Frequent Items in Streaming Data: An Experimental Evaluation of the State-of-the-Art

Nishad Manerikar
University of Trento
nd.manerikar@studenti.unitn.it

Themis Palpanas
University of Trento
themis@disi.unitn.eu

Abstract

The problem of detecting frequent items in streaming data is relevant to many different applications across many domains. Several algorithms, diverse in nature, have been proposed in the literature for the solution of the above problem. In this paper, we review these algorithms, and we present the results of the first extensive comparative experimental study of the most prominent algorithms in the literature. The algorithms were comprehensively tested using a common test framework on a variety of real and synthetic data. Their performance with respect to the different parameters (i.e., parameters intrinsic to the algorithms, and data related parameters) was studied. We report the results, and insights gained through these experiments.

1 Introduction

Over the past few years, there has been a substantial increase in the volume of data generated and the rate at which these data are generated by various applications. These two factors render the traditional *store first and process later* data analysis approach obsolete for several applications across many domains. Instead, a growing number of applications relies on the new paradigm of *streaming data processing* [18, 2, 19]. Consequently, the area of data stream mining has received considerable attention in the recent years.

An important problem in data stream mining is that of finding frequent items in the stream. This problem finds applications across several domains [10, 11, 9], such as financial systems, web traffic monitoring, internet advertising, retail and e-business. Furthermore, it serves as the basis for the solution of other relevant problems, like identifying frequent *itemsets* [16] and *recent* frequent items [21]. A common requirement in these settings is to identify frequent items in real time with a limited amount of memory, usually orders of

magnitude less than the size of the problem.

Several novel algorithms have been proposed in the literature to tackle this problem. There are generally two approaches: counter-based methods, and sketch-based methods. Counter-based algorithms maintain counters for and monitor a fixed number of elements of the stream. If an item arrives in the stream that is monitored, the associated counter is incremented, else the algorithm decides whether to discard the item or reassign an existing counter to this item. The prominent counter-based algorithms include Sticky Sampling and Lossy Counting (**LC**) [16], Frequent (**Freq**) [14, 12], and Space-Saving (**SS**) [17].

The other approach is to maintain a sketch of the data stream, using techniques such as hashing, to map items to a reduced set of counters. Sketch-based techniques maintain approximate frequency counts of all elements in the stream. The prominent sketch-based algorithms include CountSketch[1] (**CCFC**) [4], GroupTest (**CGT**) [6], Count Min-Sketch (**CM**) [5], and hCount (**hC**) [13].

Although similar in some aspects, each algorithm has its own characteristics and peculiarities. As far as we are aware, there has not been a comprehensive comparative study of all these algorithms. In this paper, we independently compare all approaches, using a common test framework and a common set of synthetic and real datasets, the real datasets coming from such diverse domains as retail, web blogs, and space measurements. It is interesting to note that several of the previous studies have not reported results on real datasets. This work represents a comprehensive set of experiments that provide statistically robust indicators of performance under a broad range of operating conditions. Moreover, we make sure that the results of our experiments are completely reproducible. Therefore, we make publicly available the source code for all the algorithms used in our experiments, as well as the datasets upon which we tested them [20].

In summary, in this work we make the following contributions.

- We evaluate the performance of the most prominent algorithms proposed in the literature for the problem of identifying frequent items in data streams. We compare the performance of these algorithms along several different dimensions, using a common and fair test framework.
- In our experimental framework, we use the most extensive and diverse set of synthetic and real datasets that has been employed in the related literature.
- Our experiments reveal how the parameters of each algorithm should be tweaked in order to suit the requirements of a particular application or data characteristics, and they indicate promising directions for future work in this area.

[1]We refer to the CountSketch algorithm as CCFC, after the authors' initials, to avoid confusion with the Count Min-Sketch algorithm.

- Finally, we provide a 'practitioner's guide' for helping in selecting the appropriate algorithm for a given problem scenario.

The rest of the paper is organized as follows: in Section 2, we define the problem formally; in Section 3, we give brief descriptions of the algorithms we test; in Section 4, we describe factors influencing the test designs. In Section 5, we present the tests and the results, followed by a discussion of the results in Section 6. Finally, we conclude in Section 7.

2 Problem Definition

All the algorithms make the simplifying assumption that the data stream is a set of integers. That is, each item or transaction[2] in the stream is represented by a single integer.

The Frequent Items problem (FI) is defined as follows.

Problem 1 [Frequent Items (FI)] *Given a support parameter ϕ, where $0 \leq \phi \leq 1$, find all the items in the data stream, which have a frequency of at least ϕN, where N is the number of transactions seen so far in the stream.*

Since the algorithms deal with approximate solutions to the frequent items problem, the problem is sometimes expressed in a modified form that takes into account an error parameter, ϵ. This variation of the problem, known as the ϵ-deficient problem, is posed as follows: given a support parameter ϕ, and an error parameter ϵ, find all the items in the stream which have a frequency of at least ϕN, with a tolerance of $\phi - \epsilon$. ϵ is usually chosen to be much smaller than ϕ; typically $\epsilon = \phi/10$ may be used.

The probabilistic algorithms use another input parameter, δ. This parameter represents the probability that the algorithm mis-classifies an item as frequent, when it is not, or not frequent, when it actually is.

The significance of the above two parameters (i.e., ϵ and δ) is that they represent the trade-off between the desired accuracy and the space used by the algorithm. With lower values of ϵ and δ, the algorithms guarantee a more accurate performance, but at the cost of higher space usage.

We should note that two of the algorithms, that is **CCFC** and **Freq** , are designed to address a slightly different, but related problem. They identify the top-k most frequent items in the data stream. Thus, they take as input the integer k, instead of the support ϕ.

Nevertheless, with a careful, yet straightforward, selection of the parameters, the above variations of the problem become equivalent. In our experiments, we make sure that all the algorithms solve the exact same problem, and can therefore be directly compared to each other.

[2]For the rest of this paper, we use the terms 'item' and 'transaction' interchangeably.

3 The Algorithms

3.1 Short Descriptions of the Algorithms

3.1.1 Freq

The Frequent algorithm keeps count of $k = 1/\phi$ number of items. This is based on the observation that there can be at the most $1/\phi$ items having frequency more than ϕN.

Freq keeps count of each incoming item by assigning a unique counter for each item, until all the available counters are occupied. The algorithm then decrements all counters by 1 until one of the counters becomes zero. It then uses that counter for the newest item. This step deletes all the non-frequent item counters. When the query is posed, the algorithm simply returns all k items as the frequent items.

3.1.2 LC

The Lossy Counting algorithm maintains a data structure $\mathcal{D}$, which is a set of entries of the form (e, f, Δ), where e is an element in the stream, f is an integer representing the estimated frequency and Δ is the maximum possible error in f. **LC** conceptually divides the incoming stream into buckets of width $w = 1/\epsilon$ transactions each. If an item arrives that already exists in $\mathcal{D}$, the corresponding f is incremented, else a new entry is created. $\mathcal{D}$ is pruned by deleting some of the entries at the bucket boundaries.

A query is answered by presenting as output the entries in $\mathcal{D}$ where $f \geq (\phi - \epsilon)N$.

3.1.3 CGT

The Combinatorial Group Testing algorithm is based on a combination of group testing and error correcting codes. Each item is assigned to groups using a family of hash functions. Within each group there is a group counter which indicates how many items are present in the group; and a set of $\log M$ counters with M being the largest item in the dataset. The group counter and the counters which correspond to the bits 1 in the binary representation of the item are updated accordingly.

Frequent items are identified by performing 'majority tests', i.e., by identifying items which occur more than half the time in a group.

3.1.4 CCFC

CCFC uses a data structure called CountSketch, which is an array of t hash tables each containing b buckets. Two sets of hash functions are used: one set $(h_1, \ldots, h_t)$ hashes items to buckets, and the other set

($s_1, \ldots, s_t$) hashes items to the set $\{+1, -1\}$. When an item arrives, the t buckets corresponding to that item are identified using the first set, and updated by adding +1 or -1 using the second set.

The estimated count of item q is the median of $h_i[q] \cdot s_i[q]$. For each item, CCFC uses the CountSketch data structure to estimate its count and maintain a heap of the top-k items seen so far.

3.1.5 CM

The Count-Min algorithm makes use of a new sketch structure called the Count-Min Sketch. It is a two dimensional array with width w and depth d, where w and d are determined by the parameters (ϵ, δ) supplied to the algorithm. Additionally, d pairwise independent hash functions are chosen at random, which hash each item to a column in the sketch.

When an item i arrives, one counter in each row is incremented; the counter is determined by the hash function. The estimated frequency for any item is the minimum of the values of its associated counters. For each new item, its estimated frequency is calculated, and if it is greater than the required threshold, it is added to a heap. At the end, all items whose estimated count is still above the threshold are output.

3.1.6 hC

The hCount algorithm also uses a Count-Min sketch. It maintains a sketch of size $m \times h$, where the parameters m and h are determined according to the data characteristics and allowed error. This sketch can be thought of as a hash-table of $m \times h$ counters. The algorithm uses a set of h hash functions to map each item of the dataset to h different counters, one in each column of the table.

The hash functions are of the form:

$$H_i(k) = (a_i \cdot k + b_i) \bmod \mathrm{P}) \bmod \mathrm{m}, 1 \leq \mathrm{i} \leq \mathrm{h}$$

where a_i and b_i are two random numbers, and P is a large prime number.

Thus each data item has a set of h associated counters, which are all incremented at the occurrence of that item. The estimated frequency of an item is simply the minimum of the values of all its associated counters. Clearly, the frequency of an item can only be overestimated. The error is estimated by using the data structure for calculating the frequency of a few elements which are not part of the stream. The average frequency of these as estimated by the algorithm is close to the error[3].

[3]The authors include this error correction scheme and name the enhanced algorithm hCount*. In this paper we refer to hCount* whenever we mention **hC** or hCount.

3.1.7 SS

The Space-Saving algorithm uses a data structure called Stream-Summary to monitor the frequent items. The Stream-Summary data structure consists of a linked list of a fixed number of counters, each corresponding to an item to be monitored. All counters with the same count are associated with a bucket which stores the count. Buckets are created and destroyed dynamically as new items come in. They are stored as an always-sorted doubly linked list. Each counter also stores the estimated error in the frequency count of the corresponding item, which is used later to provide guarantees about the accuracy of the frequency estimate returned by the algorithm.

When a query is posed, the algorithm scans through the buckets and returns the items whose counters are associated with the buckets that have values greater than the threshold ϕN.

4 Experimental Framework

4.1 Parameters and Performance Measures

The performance of the algorithms is affected by three sets of parameters (see Table 1).

- The intrinsic parameters of the algorithms: the tolerance ϵ, and error probability δ.
- The characteristics of the data stream: the number of items in the stream, N, the maximum value in the item domain[4], M, and the distribution of the item values (e.g., zipf parameter, Z).
- The query parameters: support, ϕ, or k (for the algorithms that cater to the the top-k items problem).

In practical applications, a reasonably high accuracy is required, and we decided to keep the parameters ϵ and δ constant throughout the experiments to reflect this requirement. Through a few preliminary tests values of $\epsilon = \phi/10$, and $\delta = 0.01$ were found to be sufficiently restrictive. We have used these values throughout, unless mentioned otherwise.

Four main indicators of performance were recorded for each experiment, as follows.

Recall : This is the fraction of the actual frequent items that the algorithm identified.

Precision : This is the fraction of the items identified by the algorithm that are actually frequent.

[4]For the datasets we used in our experiments, M represents the largest item in the stream, as well as the cardinality of the domain of the items in the stream. Even if there are M possible distinct items, such that the largest item is greater than M, we can assign the labels 1 to M to these items, so that the largest item is the same as the number of possible distinct items.

Table 1. Parameters that define the memory requirements for the different algorithms.

Algorithm	Parameter(s)
CGT	ϕ, δ, M
CCFC	k, δ, M, ϵ
CM	δ, M, ϵ
LC	ϵ
Freq	k
hC	δ, M, ϵ
SS	ϵ

Memory Used : The total memory used by the algorithm for its internal data structures.

Update Time : The total time required by the algorithm to process all items in the data stream. This is the time needed by the algorithm *only* for updating its internal data structures. This also gives a measure of the relative maximum data rates that the algorithms can handle.

The queries posed to the algorithms are expressed in terms of the threshold ϕ. For the two algorithms that are designed specifically for the top-k problem (**CCFC** and **Freq**), we used $k = 1/\phi$, as that is the maximum number of items that can have a frequency of ϕN.

4.2 Memory Considerations

Most papers describe memory bounds for the algorithms (that is, the amount of memory necessary in order to achieve a certain level of performance). These bounds are expressed as a function of the algorithm parameters listed in Table 1, and differ for each algorithm.

In our study, we performed two sets of experiments. In the first set, we allocated memory to each algorithm according to the theoretical bounds described in the corresponding papers. Note that this meant we had knowledge of the item domain cardinality, M. In the second set of experiments, we allocated the same, fixed memory budget to all the algorithms (the algorithm had to initialize its data structures using only the budgeted memory).

4.3 Implementation Details

All algorithms were implemented in *C*. Our implementation of the **CCFC** , **CM** , **CGT** , **LC** and **Freq** algorithms was based on the Massive Data Analysis Lab code-base [7]. The **hC** and **SS** algorithms were

implemented from scratch, using the same optimizations as the other algorithms. These are best-effort re-implementations based on the original papers. The code was compiled using the *gcc* compiler (version 4.1.2). The tests were run on an IBM x3250 server, with Intel Xeon Quad Core X3220 2.4GHz CPU and 4GB of main memory.

In order to calculate the recall and precision for the experiments, we also implemented a simple array to keep exact counts of all the items in the data stream. We refer to this as the **Exact** algorithm.

4.4 Datasets

4.4.1 Synthetic Data

The synthetic datasets were generated according to a Zipfian distribution. We generated datasets with the size, N, ranging between 10,000-100,000,000 items, item domain cardinality, M, 65,000-1,000,000, and Zipf parameter, Z, 0.6-3.5. The parameters used in each run are explicitly mentioned in the discussion of each experiment. We should note that (as described in Section 5) we generated several independent datasets for each particular choice of the data parameters mentioned above, and repeated each experiment for all these datasets.

4.4.2 Real Data

In our experiments, we used several real datasets coming from diverse domains. These datasets were as follows.

Kosarak: It is an anonymized click-stream dataset of a Hungarian online news portal [1]. It consists of transactions, each of which has several items, expressed as integers. In our experiments, we consider every single item in serial order. The dataset has a length of about 8,000,000 items.

Retail: It contains retail market basket data from an anonymous Belgian store [3]. As in the previous case, we consider all the items in dataset in serial order. The size of the dataset is 900,000 items.

Q148: This dataset was derived from the KDD Cup 2000 data [15], compliments of Blue Martini. The data we use are the values of the attribute "Request Processing Time Sum" (attribute number 148) from the *clicks* dataset. In order to get our final dataset, we replaced all missing values (question marks) with the value of 0. In this dataset there are approximately 235,000 items.

Nasa: For this dataset, we used the "Field Magnitude (F1)" and "Field Modulus (F2)" attributes from the Voyager 2 spacecraft Hourly Average Interplanetary Magnetic Field Data [8], compliments of NASA and

the Voyager 2 Triaxial Fluxgate Magnetometer principal investigator, Dr. Norman F. Ness. The dataset was constructed as follows. We used the data for the years 1977-2004. We removed the unknown values (values marked as 999), and multiplied all values by 1000 to convert them to integers (the original values were real numbers with precision of 3 decimal points). Finally, we concatenated the values of the two attributes, so that in our experiments, we read all the values of attribute "F1", followed by all the values of the attribute "F2". The total size of this dataset is approximately 292,000 items.

5 Experimental Results

In this section, we report the results of our experiments. Each experiment was run 20 times (5 times for the real datasets), and in all graphs we report the mean over all independent runs. In each run, the algorithms were reinitialized, and used a different seed (when applicable). For example, for those algorithms that require random numbers for hashing, new sets of numbers were generated. In addition, a new dataset (with the same characteristics) was generated for the synthetic data experiments. Graphs for each experiment are plotted using the average values over all runs, along with the 95% confidence intervals, shown as errorbars. (Note that in several cases, the confidence intervals are too narrow to be clearly visible in the graphs.)

5.1 Synthetic Datasets

In this first set of experiments, we made available the data and query characteristics to the algorithms so that they could be initialized with the author recommended memory allocation. The objective here is to compare the memory requirements of the algorithms, and their performance when using the recommended amount of memory.

5.1.1 Memory Usage

Expt. 1 *Synthetic datasets with* $N = 10^6, Z = 1.1, M = 10^6$ *were generated; with the other parameters being:* $\phi = 0.001, \delta = 0.01, \epsilon = \phi/10$.

A zipf parameter of 1.1 was chosen so that the data are not overly skewed, which would make it very easy to distinguish frequent items. But at the same time it ensures that there is a sizable group of items which are above the threshold for a reasonable range of values for the support.

As expected, the memory used by the algorithms varied greatly (see Table 3), with **Freq** using the least (136 KB), and **CM** using the most (2.6 MB). In comparison, **Exact** used 4.1 MB of memory. We study the variation in memory usage with change in ϕ in Section 5.1.4.

5.1.2 Item Domain Cardinality

Since the memory usage of most algorithms depends upon M (see Table 1), it is illustrative to look at the effect of varying M.

Expt. 2 $N = 10^6, Z = 1.1, \phi = 0.001$, *and* M *was varied from* 2^{16} *to* 2^{20}.

The memory usage is shown in Figure 1. As expected, **Freq** , **LC** , **SS** are unaffected. The other algorithms require more memory as M increases, and the increase is logarithmic.

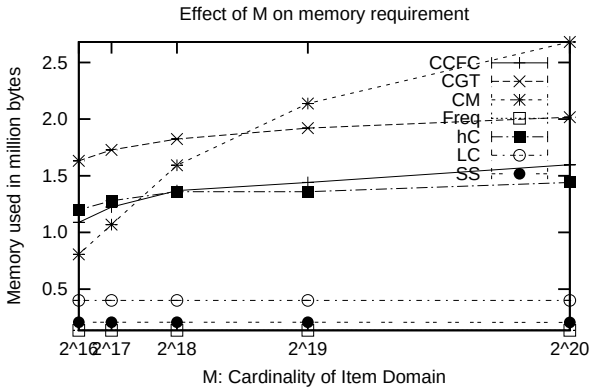

Figure 1. Effect of Cardinality of the Item Domain on Memory Requirement.

5.1.3 Number of Items

Expt. 3 $Z = 1.1, M = 10^6, \phi = 0.001$, *and* N *was increased from* 0.5 *to* 5×10^6 *in increments of* 0.5×10^6.

Ideally, with the algorithms allowed to use the optimum amount of memory, the accuracy achieved should be very high. We checked the precision and recall of the algorithms, as the number of items in the stream was increased (see Figure 3). **SS** and **LC** achieved the highest accuracy, with 100% recall and precision in every run. **hC** and **CM** achieved almost 100% on both counts. **CCFC** was slightly down on recall (around 95%), while **CGT** was slightly down on precision (93%). **Freq** had consistently very low precision (around 15%).

5.1.4 Support

The support is the defining parameter when mining for frequent items. An algorithm should be able to answer queries reliably over a wide range of support values. In this experiment we inspected the performance

of the algorithms with change in support.

Expt. 4 $N = 10^6, Z = 1.1, M = 10^6$*, and support* ϕ *was varied from 0.001 to 0.01 in increments of 0.001.*

The recall and precision achieved by the algorithms are shown in Figure 4. Performance of all algorithms was consistent over the entire range of the support values. **Freq** exhibited low precision.

It should be noted that in the experiments in this section, we allowed the algorithms to know ϕ beforehand, so that they are able to allocate memory accordingly. It is illustrative to look at how the algorithms needed to use increasing amounts of memory to cater to lower supports in order to maintain high recall and precision. This is shown in Figure 2. Quite clearly, there is an inverse proportionality relationship between the support and the memory requirements for all algorithms. This is especially pronounced in the case of **CM** , **CGT** , **CCFC** and **hC** , which are all sketch-based algorithms.

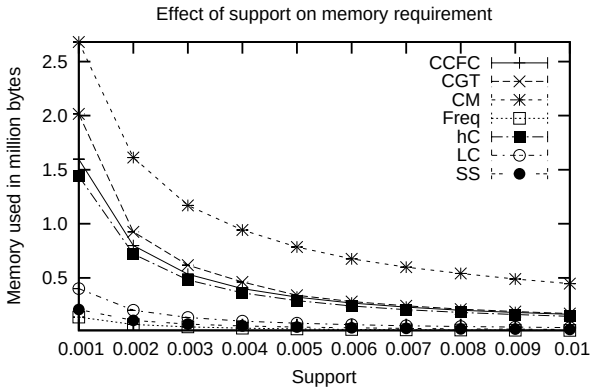

Figure 2. Effect of Support on Memory Requirement.

5.1.5 Data Distribution

The inherent assumption in mining data for frequent items is that that data are not uniform, and have features of interest. This is reflected in the skewness of the data. In this experiment, we tested the algorithms against data of varying skewness. Streams with a high skew have a few items which occur very frequently; streams with low skew have a more uniform distribution of items, and it is more difficult for the algorithms to distinguish the frequent items.

Expt. 5 $N = 10^6, \phi = 0.001, M = 10^6$*, and the Zipf parameter,* Z*, was varied between* 0.6-3.5.

Table 2. Maximum possible data rates (MB per second) that can be handled by the algorithms, based on update time for 10^8 items.

Algorithm	Maximum Data Rate (MB/sec)
CCFC	0.1603
CGT	1.2847
CM	3.8445
Freq	32.9307
hC	3.1186
LC	14.9467
SS	0.0369

The results are shown in Figure 5. As expected, all algorithms performed well for highly skewed distributions ($Z > 1.0$). **hC** , **LC** and **SS** exhibited high recall and precision even for the distributions with $Z < 1.0$. Recall for **CCFC** , and precision for **CGT** dipped noticeably for $Z < 1.0$.

5.1.6 Time

In this experiment, we measured the time required by the algorithms to update their internal data structures in response to new items arriving in the stream, and the time required to answer a query, i.e., to identify the frequent items. Up to 100 million items were fed to the algorithms one after the other without any external delays. The entire dataset was stored in main memory to ensure that there were no delays reading from disk. The cumulative time required to handle the entire stream was measured, which we call the update time.

Expt. 6 $Z = 1.1, \phi = 0.001, M = 10^6$*, and the* N *was varied between* 10^4-10^8.

Figure 9(a) shows the update times for the algorithms as a function of the number of items. As with memory, the update times vary widely. A good feature of all the algorithms is that the increase in update time scales linearly with the number of items. **Freq** has the smallest update times, but remember that its precision is always low. The best combination of accuracy and update time is offered by **LC** . The high accuracy of **SS** comes at the cost of increased update time requirements.

Based on the update time, we can calculate a 'maximum data rate' that each algorithm can handle. The calculation is based on the fact that each item in our stream is represented using 4 bytes. Combining this with the update times obtained, the maximum data rates of the algorithms are given in Table 2.

We also measured the time required to output the frequent items, which we call the query time. For all algorithms except **hC** , the query time was found to be negligible, i.e., queries were answered almost instantaneously. **hC** had a considerably large query time of around 1.3 seconds. The reason for this discrepancy, is that **hC** estimates the frequency of each individual item in the stream, and then outputs the ones above the threshold. It does not use any special data structure for keeping track of only the frequent items, as the other algorithms do. It seems that a significant speed up in query time could be achieved if something like a heap of frequent items was maintained (as in **CM**).

5.2 Synthetic Datasets, Budgeted Memory

As experiments in Section 5.1 show, almost all the algorithms perform well across different distributions and across several support thresholds. However, the comparison is in a sense unfair, as some algorithms use significantly more memory than others to achieve the same level of accuracy. In this section, we report experiments, where we allocated an equal, fixed memory budgets to all algorithms.

Due to the vastly differing internal data structures, it was impossible to restrict each algorithm to an exact memory budget number. We set the memory budget as follows. Observing that **Freq** , consistently uses the least amount of memory, we used the **Freq** memory usage as the baseline. Although **Freq** was not the best in terms of precision, this choice ensured that the algorithms were stressed. In each experiment, the memory used by **Freq** was set as the fixed budget for the other algorithms. The initialization part of the other algorithms was tweaked to cater to this requirement. We found that we could initialize all algorithms with almost equal memory, within a margin of $\pm 3\%$.

We repeated all the experiments described so far, only this time, all algorithms used the same amount of memory. The rest of the experimental settings were the same as those in Section 5.1.

5.2.1 Number of Items

With memory budgets, **CCFC** and **CGT** were the algorithms most severely affected (see Figure 6). There was a sharp fall in the recall for **CCFC** . **CGT** exhibited low recall as well as low precision. The precision of **CM** was also slightly lower. We examined the percentage reduction in memory usage for each algorithm as compared to the non-budgeted case. For the case $N = 5 \times 10^5$, these values are given in Table 3. It is interesting to note that even with these large reductions in memory, the accuracy of **SS** , **LC** and **hC** was not much affected.

Table 3. Change in memory usage with memory budgets. Memory is indicated in bytes.

Algorithm	Without Budget	Budgeted	Difference %
CCFC	1596732	136176	-91.47
CGT	2016228	136580	-93.23
CM	2680432	132296	-95.06
Freq	136024	136024	0
hC	1440104	136096	-90.55
LC	399992	136056	-65.99
SS	213485	135932	-36.33

5.2.2 Support

Again, for every support value, the memory allocated by **Freq** was used as the budget for the other algorithms. The observations are similar to the previous experiment, with **CGT** being severely affected in terms of both recall and precision (see Figure 7). Recall for **CCFC** and precision for **CM** were lower. Again, precision and recall for **SS** , **LC** and **hC** remained more or less unaffected.

5.2.3 Data Distribution

The experiments with varying Zipf parameter, Z, demonstrate that changes in data distribution affect the performance of the budgeted memory algorithms in a more pronounced manner. The recall and precision of the algorithms are shown in Figure 8. **CCFC** and **CGT** performed noticeably worse than in the non-budgeted case, although only recall was affected for **CCFC** . Also the effect of lower Z values (more uniform distribution) was even more pronounced. It was surprising to notice that *both* precision and recall for **CGT** for $Z < 1.0$ were zero. Precision for **CM** also suffered for lower Z values. **SS** , **LC** and **hC** once again performed almost as well as they did in the non-budgeted case. Predictably, for extremely skewed data ($Z > 1.5$), all algorithms performed well, despite the lower memory.

5.2.4 Time

With budgeted memory, the internal data structures used by the algorithms are smaller (less counters, smaller hash tables). Accordingly, the update time can be expected to be lower than was the case with the non-budgeted memory. The experiment in Section 5.1.6 was repeated using memory budgets and the above hypothesis proved to be true. For some of the algorithms, the reduction in update time was especially high: most notably for **CCFC** (see Figure 9(b)).

Table 4. Number of frequent items above the range of supports for the real datasets.

	Support					
Dataset	0.001	0.002	0.004	0.006	0.008	0.010
Q148	66	37	21	15	10	8
Retail	64	18	7	5	5	5
Kosarak	73	35	16	11	10	6
Nasa	215	147	76	25	19	15

5.3 Real Datasets

In this section we describe the experiments performed with real datasets. The datasets used are described in Section 4.4. In Table 4, we list for each dataset the number of items that are above the range of supports we used in the experiments.

The recall and precision of the algorithms were tested against varying support (0.001 to 0.01 in increments of 0.001). The testing was performed, as before, without and with memory budgets. For the budgeted case, the memory used by **Freq** was used as the common budget for all algorithms.

5.3.1 Q148

With non-budgeted memory, all algorithms performed well, giving almost 100% recall and precision over the entire range of support values. When we introduced memory budgets, only the performance of **SS** , **hC** and **LC** remained at high levels (see Figure 10).

On the contrary, when using budgeted memory, recall for **CCFC** and **CGT** fell, and it decreased further for higher support values. For **CGT** , precision was lower and decreased further with increasing support. **CM** exhibited similar behavior to **CGT** , but it was not as pronounced.

5.3.2 Retail

With non-budgeted memory, again all algorithms performed well. The recall for **hC** was slightly low for lower values of support, falling to 60% for $\phi = 0.001$.

This behavior was unchanged in the budgeted memory case. With budgeted memory, **CCFC** also had low recall for lower support values. **CGT** was severely affected, with recall and precision both falling to zero (see Figure 11). Precision for **CM** was markedly low, too. **LC** and **SS** once again performed consistently well.

5.3.3 Kosarak

Kosarak is the largest real dataset we used, and it also proved to be the toughest, especially for the sketch-based algorithms. For non-budgeted memory, the results for **hC** were similar to those obtained with the Retail dataset: the recall fell away for lower values of support, and this decrease was very pronounced (recall was less than 40% for all values of support less than 0.008). The other algorithms did well on recall as well as precision. **CGT** showed some dips in precision for a few values of support.

For the budgeted memory case, **CGT** was performing close to zero for both recall and precision. Recall for **hC** followed the same pattern as the non-budgeted case. Precision for **CM** was low as well (see Figure 12).

5.3.4 Nasa

Results for this set were similar in nature to the previous experiments. **LC** and **hC** performed well without and with memory budgets. In the budgeted case, all other algorithms had a lower precision, and it was markedly low for the particular point $\phi = 0.006$ (see Figure 13).

6 Discussion

Looking at the results of the experiments on synthetic and real datasets, a few general conclusions can be drawn.

6.1 Performance of the Algorithms

Even though **CCFC** and **Freq** were initially designed to solve the top-k problem, we included them in our study for completeness. The experiments indicate that **CCFC** could be adapted to the FI problem, since it performed reasonably well in our tests. On the other hand, **Freq** performed consistently low in precision.

The sketch-based algorithms **CGT** , **CM** and **hC** performed reasonably well, but were usually affected in some way at the extremes of the parameter ranges. The sketch-based algorithms (except **hC**) were also the ones that were most affected when restricted to use memory budgets.

It is also interesting to note that some of the algorithms exhibit a more stable behavior than others. This is apparent in the experiments with the synthetic datasets, where we repeatedly run each experiment, every time with a newly generated dataset (but always following the same data distribution). If we focus our attention on the confidence intervals reported in the results, we can see that the performance of **CGT** and **CCFC** has large variations among runs of the same experiment. The same, but to a lesser extent is true for

Freq and **CM** . This means that the above algorithms are rather sensitive to small variations in the input data distribution. The rest of the algorithms do not have significant variations in their performance, with **SS** exhibiting the most stable behavior of all.

Finally, it was observed that some algorithms show peculiar behavior when faced with a particular real dataset; for example, the uncharacteristically low recall of **hC** on Kosarak, or low precision of **LC** on Nasa. It would be worth exploring the reasons for this behavior.

6.1.1 Tighter Memory Constraints

It was observed in Section 5.2.1 that memory restrictions did not affect the performance of **LC** and **hC** much. We decided to stress these three algorithms further and see at how much lower levels of memory they could deliver good performance. The settings used were $N = 10^6, Z = 0.8, M = 10^6$, and the memory budgets were manually allotted. These budgets were varied from 80 KB down to 10 KB.

The results of these experiments were interesting (refer to Figure 14). The performance of **LC** degraded gradually with decreasing memory sizes. **SS** exhibited high precision and recall for memory sizes greater than 15 KB, after which its performance deteriorated drastically. **hC** on the other hand, performed well even with the low memory allocation of 10 KB, achieving recall of 97% and precision of 75%. This might be explained by the fact that **hC** does not need to make decisions about which items to monitor and which ones to discard (being a sketch-based algorithm), but rather keeps approximate counts of all items.

We further ran the same experiments on the real datasets. The results were averaged over all datasets, and are shown in Figure 15. In this case, **SS** is the one that exhibits the best overall performance. The variation in the results of **hC** is high, because it performed poorly on one of the datasets (Kosarak). All algorithms performed poorly at the low memory end (10 KB).

6.2 Memory Bounds

All the algorithms allocate at the beginning the memory needed for their internal dta structures, using the specified input parameters. Ideally, the memory requirement should be independent of any data related parameters, because in practical applications these would very often be unknown, or hard to estimate. Using these parameters means that the algorithm makes assumptions about the data distribution, or the maximum item value that may appear in the stream.

Finally, a careful observation of the experimental results with low memory budgets (Sections 5.2 and 6.1.1) reveals that it might be possible to obtain tighter theoretical memory bounds for several of the algorithms - most notably **hC** , **LC** and **SS** . The experiments demonstrate that the desirable performance

levels can be achieved with sometimes considerably lower memory requirements. This means that there is certainly room for future work on the theoretical analysis of these algorithms.

6.3 Sketch-based vs. Counter-based

Although our experiments were centered around the FI problem, it should be noted that the sketch-based algorithms apply to a broader range of problems. Maintaining a sketch implies that the algorithm stores information about all elements in the stream, and not just the frequent items. Thus, sketches act as general data stream summaries, and can be used for other types of approximate statistical analysis of the data stream, apart from being used to find the frequent items.

Thus, if an application was strictly limited to discovering frequent items, counter-based techniques (**LC** , **SS**) would be preferable due to their superior performance and ease of implementation. However, if more information about the data stream (other than just the frequent items) is required, then a sketch-based algorithm would be a better choice (**hC**).

6.4 Practical Considerations

We now examine the algorithms with respect to plausibility of use in real life applications. The first observation is that an efficient implementation also requires some general knowledge of the data characteristics, since these are sometimes an input to the algorithms (refer to Table 1). The error parameters (δ, ϵ) can be fixed at sufficiently low values without knowledge of the stream or the support that might be required in the queries later. Wherever k or ϕ are required, again a worst case estimate may be used. Knowing M, may in some cases be a problem. For **hC** , this also hides a pertinent implementation issue: for the hashing function, a large prime number and pairs of randomly generated integers are required, and for the hashing to be effective, this prime number and the random numbers need to be of the same order of magnitude as the largest number in the stream.

Another practical issue in implementing the algorithms concerns the implicit assumption in the algorithms that the data stream is a sequence of integers. This requirement is imperative for hash-based algorithms. For streams of other data types, a conversion step would be required. For example, if the data stream consisted of words, there would have to be a method of converting each word to a unique integer, say by having an intermediate hashing function.

The algorithms should also be able to handle ad-hoc queries, and dynamically adapt to changing input parameters. For example, when an algorithm is initialized and starts monitoring the data stream, the user might be interested in items above $\phi = 0.01$. If at a later point the user wants to identify items above

$\phi = 0.005$, the algorithm should have the mechanisms to perform a dynamic and smooth transition to the new requirements.

Other issues to be addressed include how query-answering and updating are to be interleaved. Whenever a query is being answered, the algorithm should not miss items in the data stream. Simple solutions include buffering the stream, or having the query process run in parallel.

7 Conclusions

The problem of identifying frequent items in streaming data is becoming increasingly relevant to many diverse domains and applications. It has also attracted lots of interest in the research community, and several algorithms have been proposed for its solution.

In this work, we experimentally evaluated the performance of several algorithms that have been proposed in the literature for mining data streams for frequent items. Over the broad range of our experiments, **hC** , **LC** and **SS** emerged as the most consistent ones, performing well across synthetic and real datasets, even with memory restrictions. They offered high precision and recall regardless of changes in support and data skew. **hC** and **SS** had a slight edge over **LC** when it came to recall and precision, but at the cost of higher query times (for **hC**) or higher update times (for **SS**).

We believe that the results of this study can help the research community focus its efforts on improving the algorithms for the FI problem, as well as help the practitioners choose the most suitable algorithm for their case among the several alternatives.

References

[1] Frequent itemset mining dataset repository, university of helsinki. http://fimi.cs.helsinki.fi/data/, 2008.

[2] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *VLDB*, pages 81–92, 2003.

[3] T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets. Using association rules for product assortment decisions: A case study. In *Knowledge Discovery and Data Mining*, pages 254–260, 1999.

[4] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 693–703, London, UK, 2002. Springer-Verlag.

[5] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[6] G. Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. *ACM Trans. Database Syst.*, 30(1):249–278, 2005.

[7] G. Cormode and S. Muthukrishnan. Massive data analysis lab, rutgers university. http://www.cs.rutgers.edu/~muthu/massdal.html, 2008.

[8] Dr. Norman F. Ness. NASA Voyager 2 Hourly Average Interplanetary Magnetic Field Data. http://nssdcftp.gsfc.nasa.gov/ spacecraft_data/ voyager/ voyager2/ magnetic_fields/ ip_1hour_ascii/, 2001.

[9] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *SIGCOMM*, pages 323–336, 2002.

[10] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *VLDB*, pages 299–310, 1998.

[11] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1999.

[12] L. Golab, D. DeHaan, E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro. Identifying frequent items in sliding windows over on-line packet streams. In *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 173–178, New York, NY, USA, 2003. ACM.

[13] C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou. Dynamically maintaining frequent items over a data stream. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 287–294, New York, NY, USA, 2003. ACM Press.

[14] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, 2003.

[15] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000. http://www.ecn.purdue.edu/KDDCUP.

[16] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357, 2002.

[17] A. Metwally, D. Agrawal, and A. E. Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Trans. Database Syst.*, 31(3):1095–1133, 2006.

[18] S. Muthukrishnan. Data streams: algorithms and applications. In *SODA*, pages 413–413, 2003.

[19] T. Palpanas, M. Vlachos, E. J. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *ICDE*, pages 338–349, 2004.

[20] Source Code, Datasets, and Additional Experimental Results. http://disi.unitn.eu/~themis/frequentitems/, 2008.

[21] F. I. Tantono, N. Manerikar, and T. Palpanas. Efficiently discovering recent frequent items in data streams. In *SSDBM*, 2008.

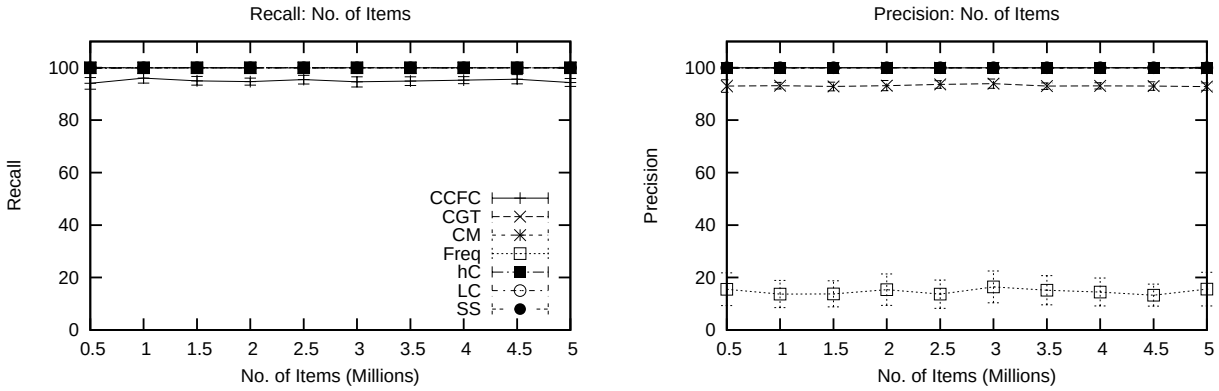

Figure 3. Effect of Number of Transactions on Recall and Precision (Items: 500,000-1,000,000; Support:0.001; Zipf: 1.1; Runs: 20)

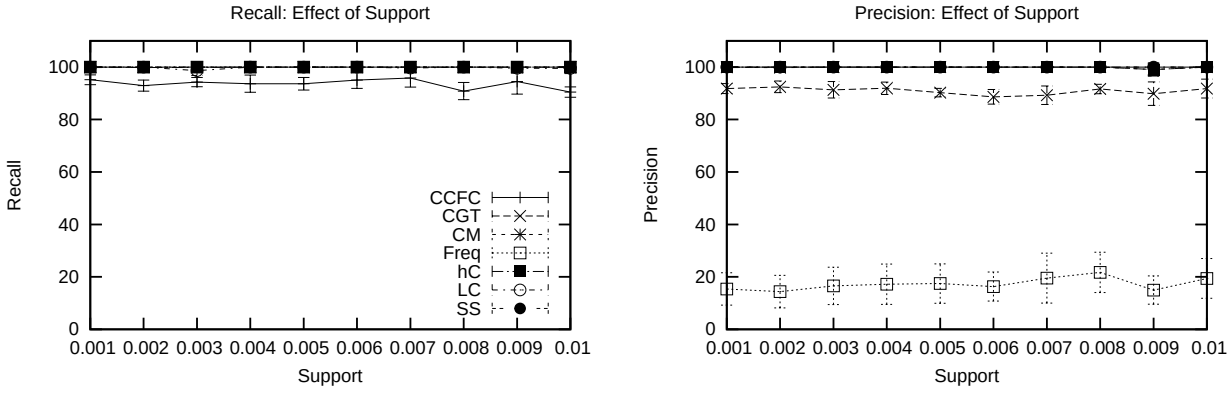

Figure 4. Effect of Support on Recall and Precision (Items: 1,000,000; Zipf: 1.1; Runs: 20)

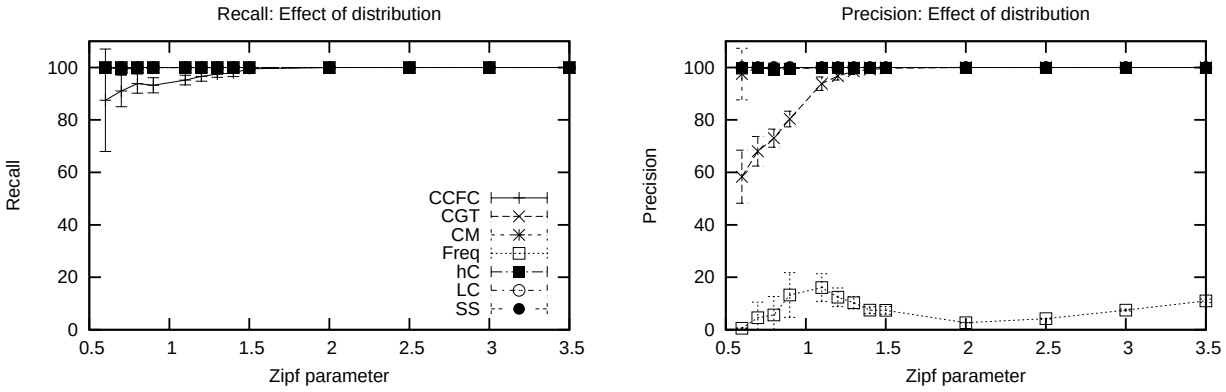

Figure 5. Effect of data distribution (skew) on Recall and Precision (Items: 1,000,000; Support:0.001; Runs: 20)

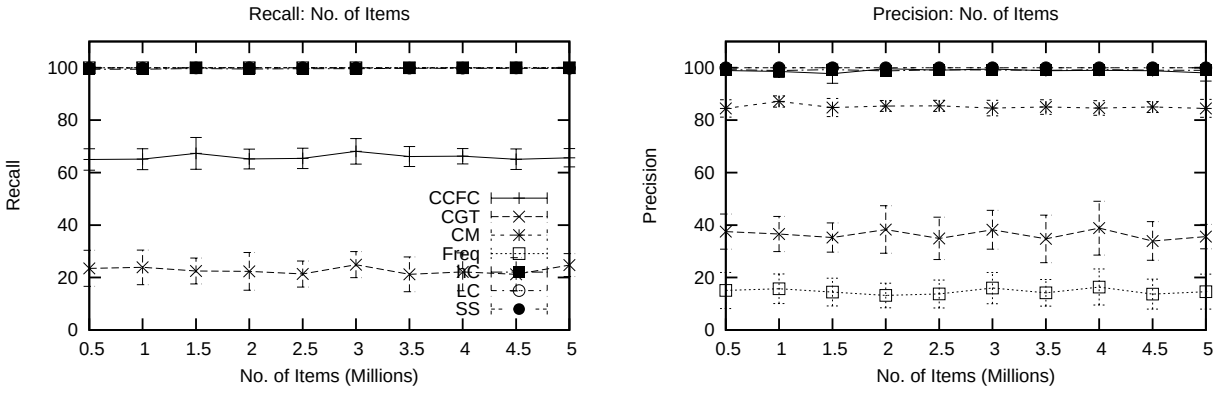

Figure 6. Effect of Number of Transactions on Recall and Precision, with Budgeted Memory. (Items: 500,000-1,000,000; Support:0.001; Zipf: 1.1; Runs: 20)

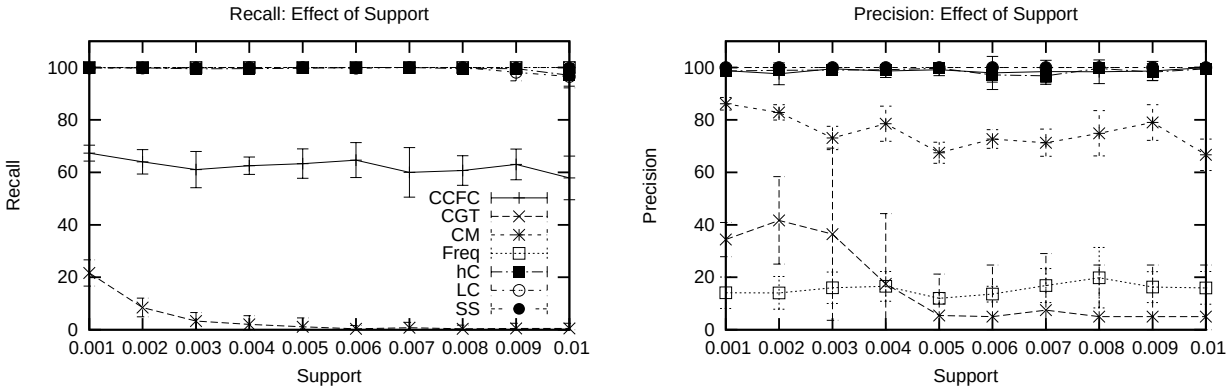

Figure 7. Effect of Support on Recall and Precision, with Budgeted Memory. (Items: 1,000,000; Zipf: 1.1; Runs: 20)

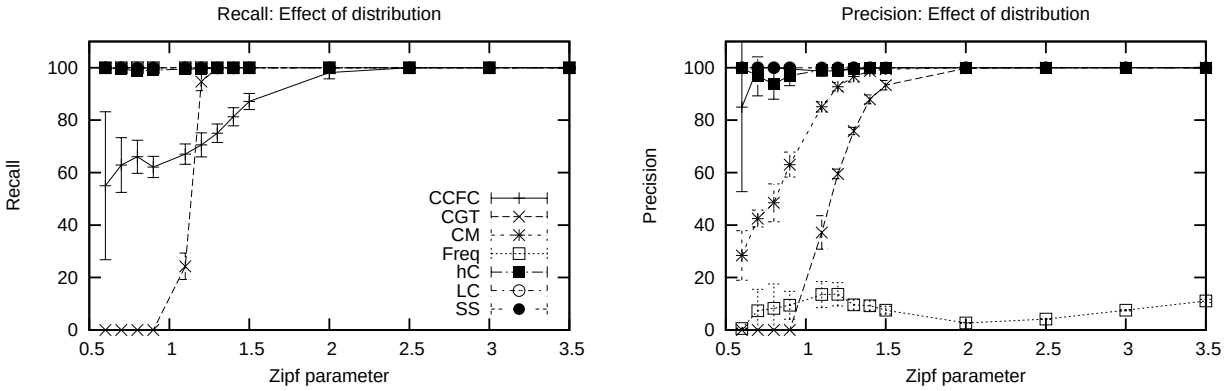

Figure 8. Effect of data distribution (skew) on Recall and Precision, with Budgeted Memory (Items: 1,000,000; Support:0.001; Runs: 20)

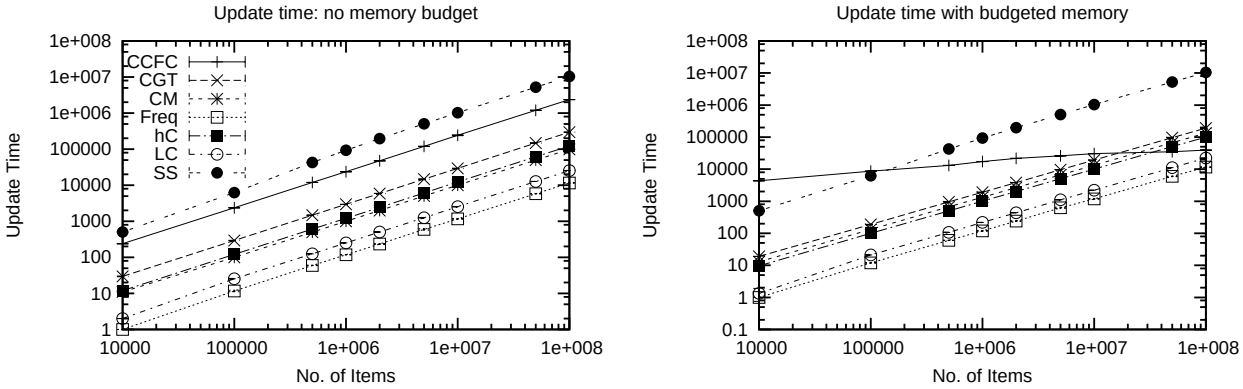

Figure 9. Update time without memory budgets and with memory budgets (Runs: 3). Note logarithmic scale on both axes.

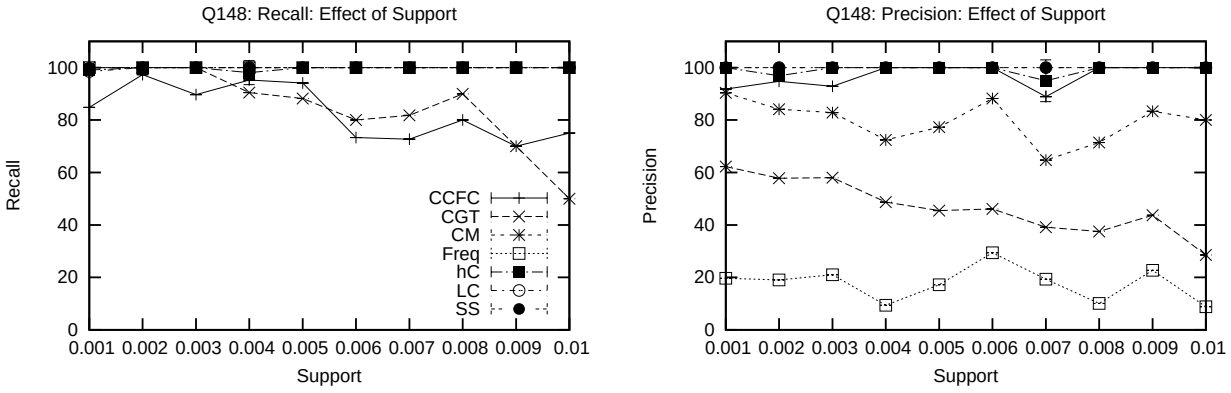

Figure 10. Dataset Q148 - Effect of Support on Recall and Precision, with Budgeted Memory (Runs:5)

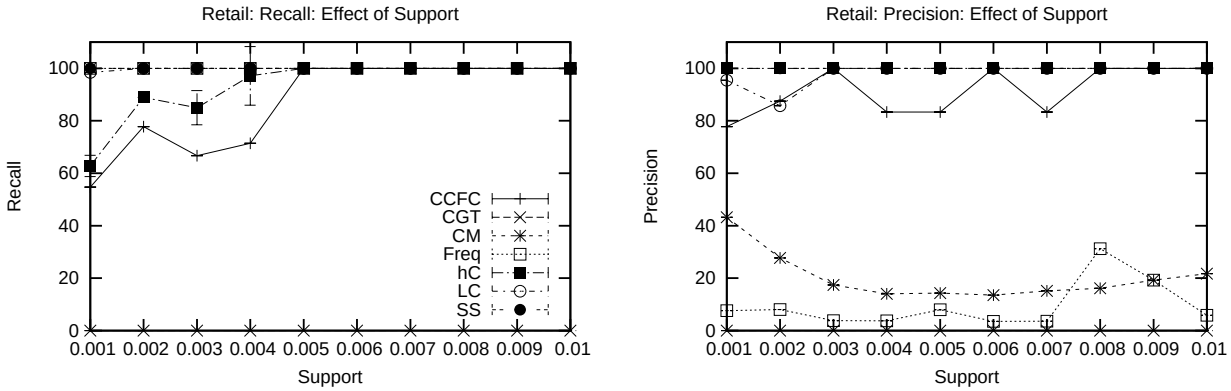

Figure 11. Dataset Retail - Effect of Support on Recall and Precision, with Budgeted Memory (Runs:5)

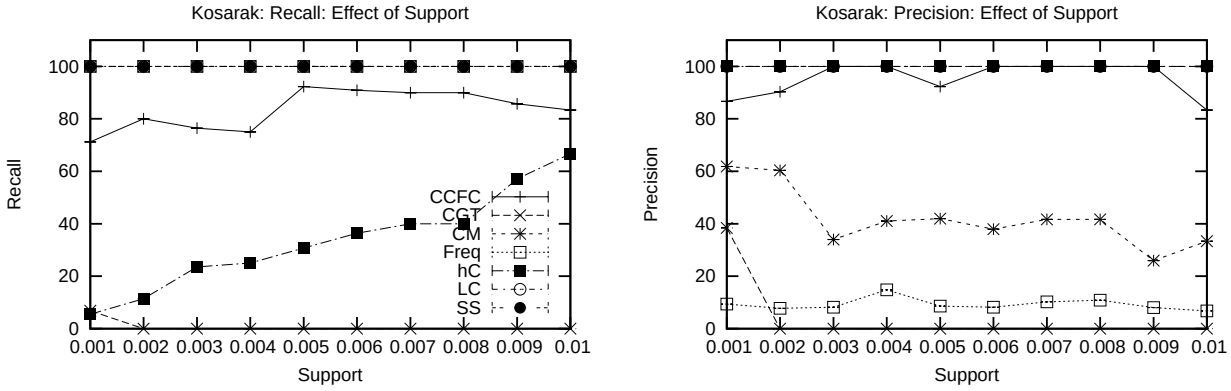

Figure 12. Dataset Kosarak - Effect of Support on Recall and Precision, with Budgeted Memory (Runs:5)

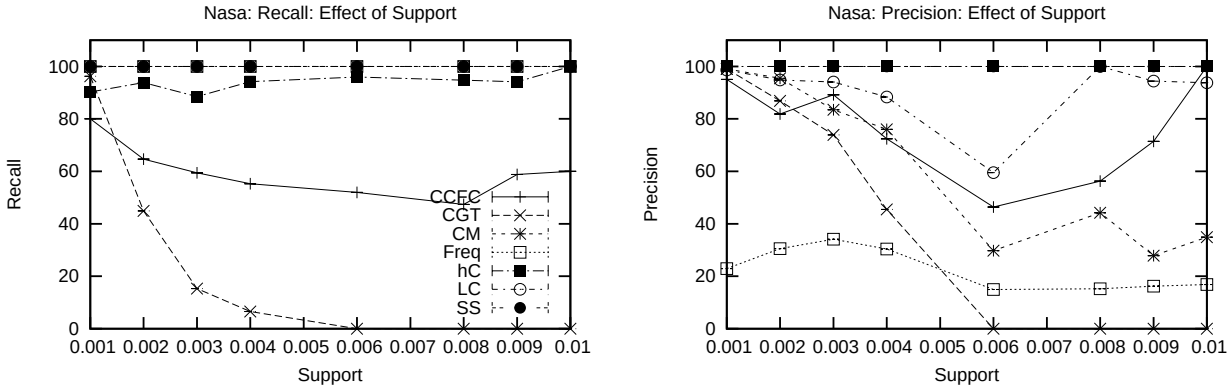

Figure 13. Dataset Nasa - Effect of Support on Recall and Precision, with Budgeted Memory (Runs:5)

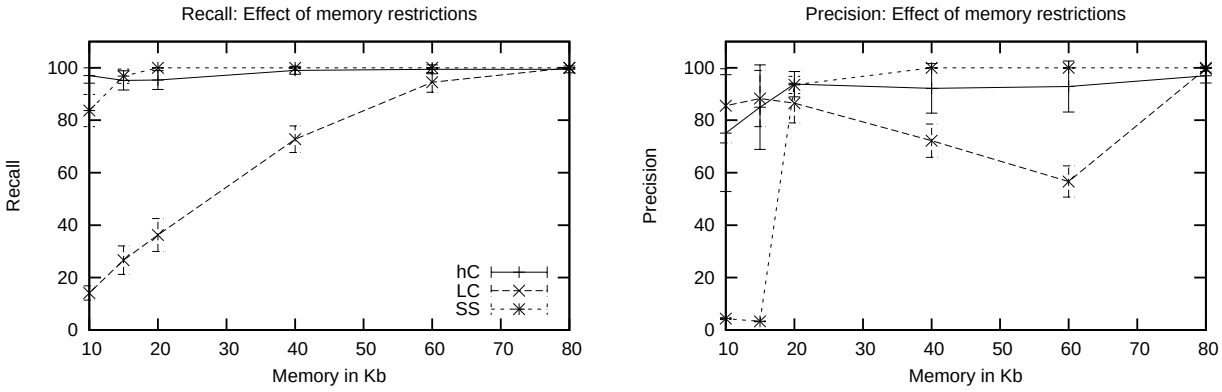

Figure 14. Effect of Memory Restrictions on Recall and Precision. (Items: 1,000,000; Zipf: 0.8; Support: 0.001; Runs: 20)

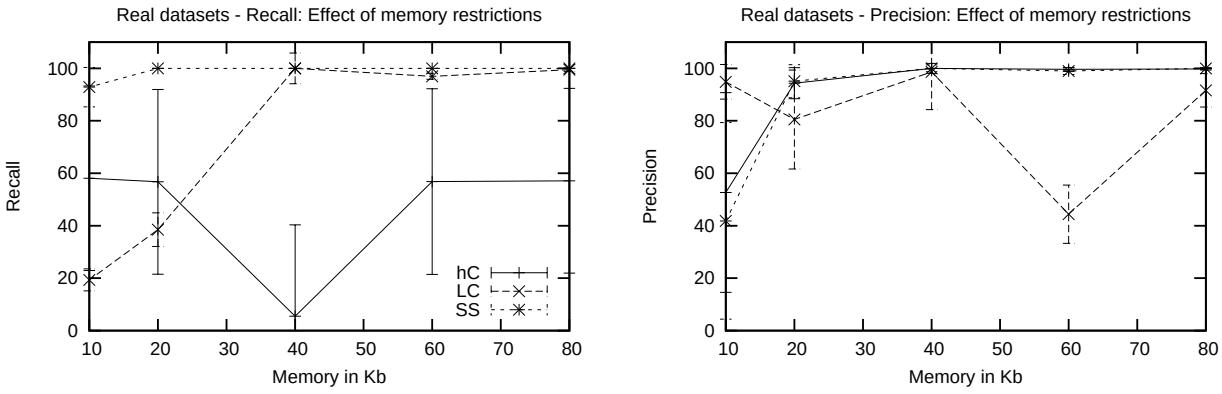

Figure 15. Real datasets: Effect of Memory Restrictions on Recall and Precision. (Support = 0.001; 5 runs over each dataset)