Dynamically Maintaining Frequent Items Over A Data Stream[*]

Cheqing Jin[†] Weining Qian[†] Chaofeng Sha[†] Jeffrey X. Yu[‡] Aoying Zhou[†]
[†]Department of Computer Science and Engineering, Fudan University, P.R.C
[‡]Department of S.E.E.M., The Chinese University of Hong Kong
{cqjin,wnqian,cfsha,ayzhou}@fudan.edu.cn yu@se.cuhk.edu.hk

ABSTRACT

It is challenge to maintain frequent items over a data stream, with a small bounded memory, in a dynamic environment where both insertion/deletion of items are allowed. In this paper, we propose a new novel algorithm, called `hCount`, which can handle both insertion and deletion of items with a much less memory space than the best reported algorithm. Our algorithm is also superior in terms of precision, recall and processing time. In addition, our approach does not request the preknowledge on the size of range for a data stream, and can handle range extension dynamically. Given a little modification, algorithm `hCount` can be improved to `hCount*`, which even owns significantly better performance than before.

Categories and Subject Descriptors

H.2 [**DATABASE MANAGEMENT**]: Miscellaneous

General Terms

Algorithms

Keywords

Stream, frequent items, algorithm

1. INTRODUCTION

A data stream is an ordered sequence of items that arrives in timely manner. Currently, many real applications need to handle data streams, such as stock tickers, network traffic measurements, click streams, sensor networks and telecom call records. The volume of data stream is so large that it can hardly be stored in main memory for on-line processing. Only a summary of the whole data can be maintained in main memory for a one-pass algorithm to process [3].

[*]This work is partially supported by The National Hi-Tech R&D program under grant No. 2002AA413310.

Maintaining most frequent items is one of important issues in database, data mining and computer network etc. For example, *Iceberg queries* [14] generalize the notion of hot items in the relation to aggregate functions over an attribute (or set of attributes) in order to find aggregate values above a specified threshold. Mining *association rules* requires finding frequent itemsets [1]. Tracking measurement and accounting of IP packets require identifies of flows that exceed a certain fraction of total traffic [12]. In addition, recently, many applications request to monitor frequent items in a dynamic environment where insertion/deletion are allowed. As such an example, in order to monitor the network traffic over a router, we must insert and delete a tuple when connection starts and ends. The list of most frequent items can change significantly over time.

In this paper, we propose a novel approach, with a small memory space, to maintain a list of most frequent items above some user-specified threshold in a dynamic environment where items can be inserted and deleted.

1.1 Problem Definition

We assume that a sequence of transactions flows through the data stream. Here, a *transaction* is either inserting or deleting an item k at time point i, denoted $t_i = delete(k)$ or $t_i = insert(k)$. Without loss of generality, we assume that the items are integers in a range of $[1..M]$. Let n_k denote the net occurrence of item k. Operation $insert(k)$ increases the net occurrence of k, i.e, $n_k \leftarrow n_k + 1$, and operation $delete(k)$ decreases the net occurrence of k, i.e, $n_k \leftarrow n_k - 1$. Let N denote the sum of net occurrence of all items. i.e., $N = \Sigma_{k=1}^{M} n_k$. The frequency of any item k can be denoted as f_k, where $f_k = n_k/N$.

The problem is defined as follows. Given three user-specified parameters: a support parameter $s \in (0, 1)$, an error parameter $\epsilon \in (0, 1)$ such that $\epsilon \ll s$ and a probability parameter ρ such that ρ is near 1.

At any point of time, with a small bounded memory, output a list of items along with their estimated frequencies with the following guarantees: i) all items whose true frequency exceeds s are output; ii) no item whose true frequency is less than $s - \epsilon$ is output; and iii) estimated frequencies are more than the true frequencies by at most ϵ with high probability ρ.

1.2 Our Contribution

The main contribution of this paper is that we propose an efficient algorithm, called `hCount`, to find a list of most frequent items over a data stream. Our algorithm can handle both insert/delete operations. The space requirement of hCount is no more than $\frac{e}{\epsilon} \cdot \ln(-\frac{M}{\ln \rho})$ counters. Compared with previous best algorithm, groupTest [8], hCount is superior to groupTest in several aspects. First, given even smaller memory space, hCount outperforms groupTest in terms of performance. Second, we ensure that any item whose frequency below $s - \epsilon$ can not be output. Recall that there is no such guarantee in groupTest. Third, hCount only requires error parameter ϵ to be predefined. No threshold parameter s is needed. Finally, hCount can be easily extended to support any data streams without the requirement of knowing the range beforehand. To our knowledge, there is no such reported study in the literature.

1.3 Paper Organization

The rest of paper is organized in this way. In Section 2, we discuss the related work. In Section 3, our algorithms are discussed in detail followed by analysis. We conducted extensive experimental studies, and will report our findings in Section 4. We conclude our work in Section 5.

2. RELATED WORK

The earliest work considered the problem of finding majority – item that occurs more than half of the time of the whole data set [5, 15]. Misra and Gries also gave an algorithm to find a list of items with frequency above n/k in two passes [20]. Recent research on finding frequent items over data streams [2, 4, 6, 7, 8, 10, 16, 17, 19] can be divided into two groups: sample-based approach and hash-based approach.

Sample-based approach holds some number of counters, each of which counts the number of net occurrence of an item over a data stream. These counters are incremented whenever their corresponding items are observed, and are decremented or reallocated under certain circumstances. Sample-based algorithms [10, 16, 17, 19] are efficient for insert-only situations. In addition, because of preserving only a part of sample data, sample-based algorithms are hard to be applied to dynamic cases.

In the hash-based approach, each item in a data stream owns a respective list of counters in a hash table, and each counter may be shared by many items. A newly arriving item changes the respective counters. Unlike sample-based approach, hash-based approach can handle deletion operation by doing adverse steps with respect to insertion operation. In [6], Charikar et al. showed that, with $O(k/\epsilon^2 \log n/\delta)$ space complexity, all items whose frequency above $n/(k+1)$ can be output with probability $1 - \delta$. Recently, Cormode and Muthukrishnan showed an algorithm, called groupTest, by which they can further reduce the space complexity. The algorithm only needs $O(k(\log k + \log(1/\delta))(\log M))$ counters to output all items with frequency above $1/k + 1$ with probability $1 - \delta$.

In this paper, we show that we can significantly outperforms groupTest algorithm. Therefore, we outline groupTest algorithm below. Cormode and Muthukrishnan observed that if the underlying data distribution owns small tail property, i.e., frequent items occur most of times, then each frequent item can be a majority by putting it with a number of tail items together. Based on this observation, they reexamined the algorithm given in [11], and devised a new approach (groupTest) as follows. First, the whole universe is divided into $2k$ subsets randomly. Second, it picks out the majority from each subset if there exists one. It is important to know that groupTest may miss some correct frequent items in two cases: a) if two or more frequent items happen to appear in one subset, for at most one majority per subset can be picked out, and b) if some other items in this subset own frequency near the threshold. Cormode and Muthukrishnan solved the problem by re-separating the universe T times to ensure all frequent items can be output, which requires more memory space. Additionally, in order to avoid outputting some redundant items, groupTest needs to further split the universe into smaller subsets so that items whose frequency below but near the threshold can hardly be output. Unfortunately, this also increases the memory space.

It is worth noting that the main ideas of our work are influenced by Bloom filter, which is an efficient data structure to represent a data set to support approximate membership queries [4]. Bloom filter has been used in some applications, including database and network [13, 18, 21]. Cohen and Matias [7] revisited it and proposed efficient ways to query item's frequency over a data stream. Their work mostly focused on how to distinguish different items from a large set, rather than maintaining the frequent items list, which is the focus of this paper. Cormode and Muthukrishnan, in a technical report, reported a new sublinear space data structure called Count-Min Sketch [9] that can be used to support point queries, inner product queries and range queries. The data structure and algorithms used in Count-Min Sketch and ours shared the similarity, but were simultaneously and independently investigated with different focuses.

3. OUR ALGORITHM

In this section, we will discuss our approach in detail. A hash table, $S[m][h]$, along with h hash functions, is used in our algorithm. Each of these h hash functions maps an digit from [0..M-1] to $[0..m-1]$ uniformly and independently. Many kinds of hash functions can simulate this piece of job. Here we choose one as follows to achieve the purpose.

$$\mathcal{H}_i(k) = ((a_i \cdot k + b_i) \mod P) \mod m, 1 \le i \le h \qquad (1)$$

where a_i and b_i are two random numbers and P is a large prime number. An item k in the range has a set of *associated counters*: $\langle S[\mathcal{H}_1(k)][1], S[\mathcal{H}_2(k)][2], \cdots, S[\mathcal{H}_h(k)][h]\rangle$. These associated counters increase or decrease at the same time when encountering a transaction on item k. The determination of values of m and h will be discussed later in Proposition 1.

Two algorithms are proposed for handling tuples over stream and outputting final result separately. The algorithm hCount (Algorithm 1) maintains such a hash table, and the algorithm eFreq (Algorithm 2) checks and outputs the items with frequency above a user-specified threshold s along with their estimated frequencies. The time requirement of Algorithm eFreq is linear to the range of universe. It is acceptable when the frequency of the requests is not highly. A simple example is given below.

EXAMPLE 1. *Assume there is a data stream, whose items are within a range of* [1..16]. *Here, in order to output the most frequent items over the data stream, a hash table* $S[m][h]$ *is created where* $m = 5$ *and* $h = 4$. *With Eq. (1), the four hash functions,* $\mathcal{H}_1$, $\mathcal{H}_2$, $\mathcal{H}_3$ *and* $\mathcal{H}_4$, *can be determined by using the following four pairs of* (a_i, b_i): $(a_1, b_1) = (7, 13)$,

Algorithm 1 hCount(k, ttype)

```
1: if ttype is insert then
2:    $N = N + 1$;
3: else
4:    $N = N - 1$;
5: end if
6: for $j = 1$ to $h$ do
7:    $pos = ((a_j \cdot k + b_j) \mod P) \mod m$;
8:    if ttype is insert then
9:       $S[pos][j] = S[pos][j] + 1$;
10:   else
11:      $S[pos][j] = S[pos][j] - 1$;
12:   end if
13: end for
```

Algorithm 2 eFreq(s)

```
1: for $k = 1$ to $M$ do
2:    $c = \min_{1 \le j \le h}(S[\mathcal{H}_j(k)][j])$
3:    if $c < sN$ then output(k, c/N);
4: end for
```

$(a_2, b_2) = (22, 6)$, $(a_3, b_3) = (24, 11)$ and $(a_4, b_4) = (14, 27)$. Suppose the prime number used in Eq (1) is $P = 31$.

For simplicity, we explain the algorithm by using Example 1. Initially, all the counters of $S[m][h]$ are initialized to zero. Assume a data stream of 38 transactions is coming as indicated in Table 1, where t_i indicates the i-th transaction and k indicates the item handled by the corresponding transaction. Here, if k is a positive number, then the corresponding transaction is an insertion transaction. Otherwise, it is a deletion transaction. At time point 4, the state of hash table is shown in Figure 1 (a). The following two transactions are: $t_5 = insert(9)$ and $t_6 = delete(6)$. Accordingly, we will call `hCount(9, insert)` and `hCount(6, delete)`. The four associated counters for t_5 and t_6 are the shadow elements in Figure 1 (b) and (c), respectively. In Figure 1 (b), the four associated counters are increased by 1 (for insert). In Figure 1 (c), the four associated counters are decreased by 1 (for delete). Figure 1 (d) shows the final state for t_{38}. At any position, the net occurrence of each item can be estimated from the hash table on demand using the minimum value of its associated counters (as shown in the `eFreq` algorithm). For example, at completion of t_{38} (Table 1), we estimate the occurrence of item 6 as 2, because it is the minimum value of the four associated counters: 2, 8, 2 and 2.

Both estimated values and true values are shown in Table 2, at time point 38, based on the hash table shown in Figure 1 (d). We observe that: i) the estimated values are all greater than or equal to the true values, and ii) the gap between the true value and estimated value is very small.

In the following, we discuss how to choose m and h to maintain an ϵ-approximate frequent summary.

Proposition 1. *By our algorithm, $\frac{e}{\epsilon} \cdot \ln(-\frac{M}{\ln \rho})$ counters are used to estimate each item with error no more than ϵN with probability ρ, while $m = e/\epsilon$, and $h = \ln(-\frac{M}{\ln \rho})$.*

We sketch our proof as follows. As seen from Algorithm 2 that, for an arbitrary item k, the respective associated counters are: $\langle S[\mathcal{H}_1(k)][1],\ S[\mathcal{H}_2(k)][2],\ \cdots,\ S[\mathcal{H}_h(k)][h]\rangle$, where each associated counter contains not only net occurrence n_k for k but also occurrences of some other items that are mapped to the associated counter. Let $\langle e_1, e_2, \cdots, e_h\rangle$ denote errors of each h counters for k, then the associated counters for k are $\langle e_1 + n_k,\ e_2 + n_k,\ \cdots,\ e_h + n_k\rangle$. Provided that all hash functions are well defined, approximately $[M/m]$ items are mapped to a counter on average. It suggests that the expected value of each associated counter is N/m and the expected value of each error is no more than N/m. Let random variable Y denote this error, then

$$E[Y] \le N/m$$

In addition, from *Markov's Inequality*, we know that:

$$Pr[|Y| - \lambda E[|Y|] > 0] \le 1/\lambda$$

where λ is a positive number. Because Y is always greater than 0 and $E[Y] \le N/m$, the above formula can be written as

$$Pr[Y - \lambda N/m > 0] \le 1/\lambda$$

The above formula shows that for random variable Y, if we try once, the event that the value of Y is greater than $\lambda N/m$ happens with probability no more than $1/\lambda$. This probability is denoted as p. If we try h times, and all values are greater than $\lambda N/m$, the probability is no more than p^h. In other words, with probability $1 - p^h$, event that the value of Y is smaller than $\lambda N/m$ happens at least once. Let Y_{min} denote the minimal value among h tries, the followings can be obtained.

$$\begin{aligned} Pr[Y_{min} - \lambda N/m > 0] &\le 1/\lambda^h \\ Pr[Y_{min} - \lambda N/m < 0] &\ge 1 - 1/\lambda^h \end{aligned}$$

Let ρ denote the probability of event that all M items satisfy the above formula simultaneously. Then,

$$\rho = (1 - 1/\lambda^h)^M \approx \exp(-M/\lambda^h) \qquad (2)$$

We know that N is the net occurrence of all items and Y_{min} is the error part of each estimated value. We set the user-specified error parameter ϵ as follows.

$$\epsilon = \lambda/m \qquad (3)$$

Next, we discuss how to minimize the memory space requirement according to parameters ϵ and ρ. Let V denote the size of the hash table. Then, $V = m \cdot h$. From Eq. (2) and (3), V can be computed as follows:

$$V = \frac{1}{\epsilon} \ln(-\frac{M}{\ln \rho}) \cdot \lambda/(\ln \lambda)$$

Here, λ is a positive number and $\min_{\lambda > 0}(\frac{\lambda}{\ln \lambda}) = e$. Hence, $V = \frac{e}{\epsilon} \cdot \ln(-\frac{M}{\ln \rho})$. Therefore,

$$h = \ln(-\frac{M}{\ln \rho}), \qquad m = \frac{e}{\epsilon}$$

Proposition 1 guarantees that the maximal error of an estimated value is no more than ϵN with probability ρ. All items whose frequencies over threshold s are output. And items whose frequencies below $(s - \epsilon)$ can hardly be output.

The above shows the bounded memory space in theory. In practice, the underlying data set tends to be skewed so that a much smaller space is sufficient with high performance. For example, with 47K counters, `hCount` can support a data stream in range $[1..2^{20}]$ ($M = 2^{20}$), with $\epsilon = 0.001$ and $\rho = 0.95$. Experiments in section 4 show that only 1/10 counters can ensure this precision under some conditions. We can

t_i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
k	2	1	6	3	9	-6	16	1	13	2	4	3	-16	1	5	3	10	5	2
t_i	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
k	11	-11	2	1	3	8	2	1	-4	11	3	7	5	1	1	9	2	2	13

Table 1: A Data Stream Example.

Item	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
True occurrence	7	7	5	0	3	0	1	1	2	1	1	0	2	0	0	0
Estimated occurrence	8	8	5	0	5	2	2	1	2	3	2	1	2	2	0	1
True frequency(%)	23	23	17	0	10	0	3	3	7	3	3	0	7	0	0	0
Estimated frequency(%)	27	27	17	0	17	7	7	3	7	10	7	3	7	7	0	3

Table 2: True Value v.s. Estimated Value

also learn that `hCount` processes each transaction quickly. In theory, only $\ln(-\frac{M}{\ln \rho})$ counters need to be updated per transaction.

3.1 An Error Reduction Technique

Based on the above analysis, we know that every counter contains an error, which is the sum of occurrences of other items mapped to the same counter. The accuracy of the algorithm can be significantly improved by removing these errors where possible. We observe that, the associated counters for those rarely occur items contain the error only, because of the true zero occurrence. In our testing, we find that the accuracy of an estimated occurrence, $\hat{n}_k$, for an item k, can be improved by removing such errors, $\hat{n}_k - \tau$, where τ is called an error factor. The error factor τ is obtained as follows. Given a range $[1..M]$, we extend the range to $[1..M + \Delta]$. Here, the items in the range $[M + 1..M + \Delta]$ never occur (zero-occurrence). For each element k' in $[M + 1..M + \Delta]$, its estimated occurrence, $\hat{n}_{k'}$, is the error. The error factor is computed as the average of the estimated occurrences for all items in $[M+1..M+\Delta]$. We call the improved algorithm `hCount*` in this paper. We will report the accuracy of `hCount*` in this paper, and study the theoretical aspects as our future work.

3.2 When Data Range Is Unknown in Advance

Above we assume that the number of distinct items M is known in advance. In the following, we show that our algorithms do not rely on the predefined M and can handle data streams in any ranges dynamically.

Assume the range of a data stream is set to M_1 initially, and increases to M for $M \gg M_1$. We deal with this situation by creating a series of hash tables step by step. Initially, we created a hash table S_1 for the items in the range of $[1..M_1]$. When one item to be handled is beyond $[1..M_1]$, we create a new hash table S_2 to handle items in the range of $[M_1 + 1, M_2]$ where $M_2 = M_1^2$. For example, let $M_1 = 1000$, then the new range will be $[1001, 1000000]$. The above procedure will repeat when needed such that a new hash table S_{i+1} is created to handle data in $[M_i + 1, M_{i+1}]$ where $M_{i+1} = M_i^2$. The parameters m and h for S_i can be obtained according to Proposition 1. Obviously, within any $[M_i + 1, M_{i+1}]$, the error of each estimated value is no more than ϵ times the sum of frequencies of items. In other words, because S_i only handles a part of all items, the error can be guaranteed within ϵN for any item.

Now we consider two cases: i) we do know the predefined M, and ii) we do not know the predefined M, but increase the range repeatedly n times from $M_1, M_2, \cdots M_n$ where $M = M_n$. Then, for case i), $\lceil \log(\lceil \frac{\log M}{\log M_1} \rceil) \rceil + 1$ hash tables will be created and each hash table is smaller than the hash table for case ii). The space requirement for case i) is no more than $\lceil \log(\lceil \frac{\log M}{\log M_1} \rceil) \rceil + 1$ times the space required for case ii).

PROPOSITION 2. *The space required differs no more than* $\lceil \log(\lceil \frac{\log M}{\log M_1} \rceil) \rceil + 1$ *times between the two cases i) and ii) mentioned above.*

4. EXPERIMENTAL STUDIES

We conducted extensive testing to compare algorithms on *recall* and *precision*. The recall of a result is the proportion of the hot items that are found by the method. The precision is the proportion of items identified by the algorithm which are hot items.

The synthetic dataset we choose is zipf distribution with range [1..1000000] and the number of items is $1,000,000$. We implemented `hCount`, `hCount*` (an error factor τ is computed by using 20 extra items) and `groupTest` [8] in `C`, and conducted all experiments on a PC with a 1.7GHz CPU and 256MB main memory.

4.1 `hCount` v.s. `hCount*`

We will show the performance of our algorithms `hCount` and `hCount*`. Proposition 1 gives the minimal size of a hash table, in order to guarantee the error. In our experiments, we found that only 2,740 4-byte counters are sufficient. Figure 2 shows the change of precision with parameter h (the number of hash functions) varying from 2 to 8. `hCount*` behaves perfectly when $h \geq 4$ while `hCount` achieves its peak precision when $h = 4$. We set h to 4 in following tests.

Our algorithms are capable to output the frequent items along with their frequencies. The errors are very small. Figure 3 shows the absolute error of `hCount` and `hCount*`. `hCount*` outperforms `hCount` in terms of accuracy. As shown in Figure 3, in the worst case, the absolute error in `hCount*` is no more than 100, just 0.01% of 1,000,000. The maximal error of `hCount` is no more than 1,200, just about 0.12% of 1,000,000. The error of `hCount` decreases quickly when the underlying data distribution is more skewed. Figure 4 shows that, when precision of algorithm `hCount` decreases significantly, precision of `hCount*` is still near to 1.

1	0	1	1	1
1	0	0	1	2
1	1	0	1	1
1	0	1	1	1

(a) The state at t_4

1	0	1	1	2
1	0	0	2	2
2	1	0	1	1
1	0	1	1	2

(b) The state at t_5

1	0	1	1	1
1	0	0	2	1
1	1	0	1	1
1	0	1	0	2

(c) The state at t_6

8	3	11	6	2
7	0	1	14	8
2	5	5	10	8
8	2	6	2	12

(d) The state at t_{38}

Figure 1: The states of the hash table when handling a data stream (Table 1)

`hCount`	2,740
`hCount*`	2,740
`groupTest` (K=50, T=4)	8,400
`groupTest` (K=50, T=8)	16,800
`groupTest` (K=100, T=4)	16,800
`groupTest` (K=200, T=4)	33,600
`groupTest` (K=200, T=8)	67,200

Table 3: Space Requirement Comparison

The recall of `hCount` is 100%. It is because each item is overestimated that no item with frequency above the threshold is missed. `hCount*` also behaves well in terms of recall. Only items with frequencies, that are very close to the threshold, may be missed. Figure 3 shows that the absolute error of `hCount*` is very small, which means that it is rare that items are output wrongly. In Figure 8, recall of `hCount*` on a real data is nearly 100%.

4.2 Comparison with Previous Algorithm on Synthetic Data

The major space used in `groupTest` is a large hash table containing $2KT(1+\log M)$ counters. Other space consumption can be ignored such as the global counter and hash parameters. Similar to `groupTest`, the major space used in our algorithms is also a large hash table. Table 3 compares the space requirement among the algorithms. `hCount` and `hCount*` need least memory space, whereas `groupTest` requests different sizes according to different parameter settings. Although `groupTest` requests a larger memory space, it shows no better than our algorithms. The precision of `hCount*` is nearly 1 at any threshold (Figure 5). However, the precision of `groupTest` depends highly on its parameter K. It behaves worst among all when $K = 50$ or $K = 100$. Only when parameter $K = 200$, `groupTest` achieves similar precision as `hCount`. Note: at that time the memory requirement of `groupTest` is ten more times larger than `hCount`'s.

Additionally, our algorithm can process each transaction faster than `groupTest`. In `hCount`, only h counters need to be updated per transaction, while in `groupTest`, T tests must be done per transaction and $O(\log M)$ counters need to be updated per test. Figure 6 shows the processing time of algorithms from which we can see that `hCount` is 3 times faster than `groupTest` when the parameter T of `groupTest` is set to 4.

4.3 Real Dataset Testing

The topic detection and tracking (TDT) is an important issue in information retrieval and text mining (`http://www.ldc.upenn.edu/Projects/TDT3/`). We obtained a news collection of the news stories distributed through Reuters real-time datafeed, which contains 365,288 news stories and 100,672,866 (duplicate) words. We removed all articles such as "the" and "a" and preprocessed the data collection by term stemming. We set up a window containing of 18,000 articles at max. When a new article comes, we increase the frequencies of all words in it. If the number of news articles exceeds this threshold, the oldest news articles is removed from the window and all words in that article are deleted (decrease by 1). For every 18,000 articles, we output a list of items above a threshold (0.5% or 1%). Figure 7 and 8 show the precision and recall comparison among several algorithms. In all situations, `hCount*` behaves best. `hCount` also behaves well. When $K = 200$ and $T = 4$, `groupTest` shows similar performance to `hCount`, while the memory space requirement of `groupTest` is ten more times over `hCount`. For `groupTest`, the performance decreases significantly with a smaller memory space. Figure 7(b) shows that when parameter K decreases a half, the precision of `groupTest` also decreases significantly. We also compared with `Lossy Count` [19] and `Frequent` [10], neither of which is capable to work under a dynamic situation without major modification. We modified `Lossy Count` and `Frequent` by decreasing the counters whenever corresponding items are deleted. Their precision and recall are at a low level.

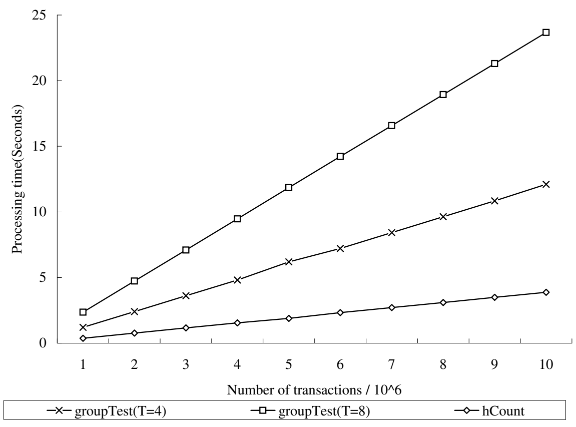


Figure 6: The processing time between `hCount` and `groupTest`

4.4 Range Extension Testing

Our algorithms can easily extend the range from what we initially set by creating a series of hash tables. In this testing, initially we set the range of universe M to a small

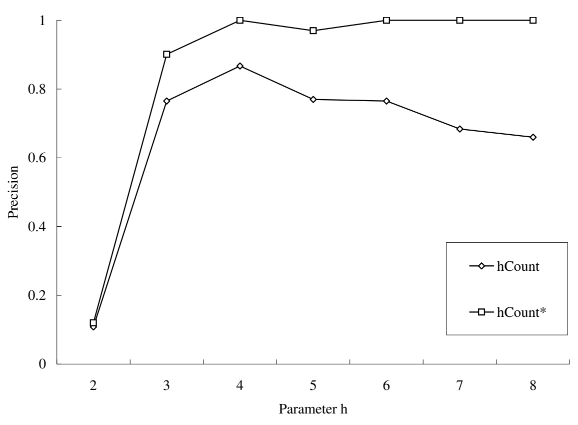


Figure 2: Precision under different h where zipf is set to 1

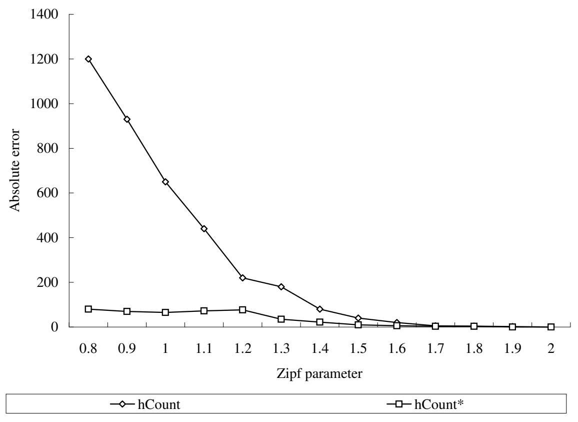
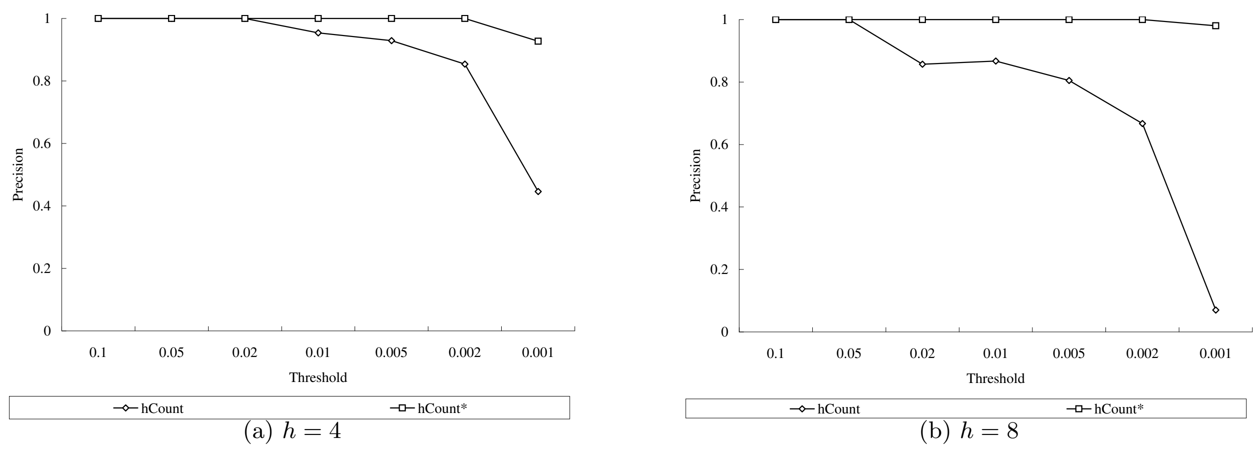
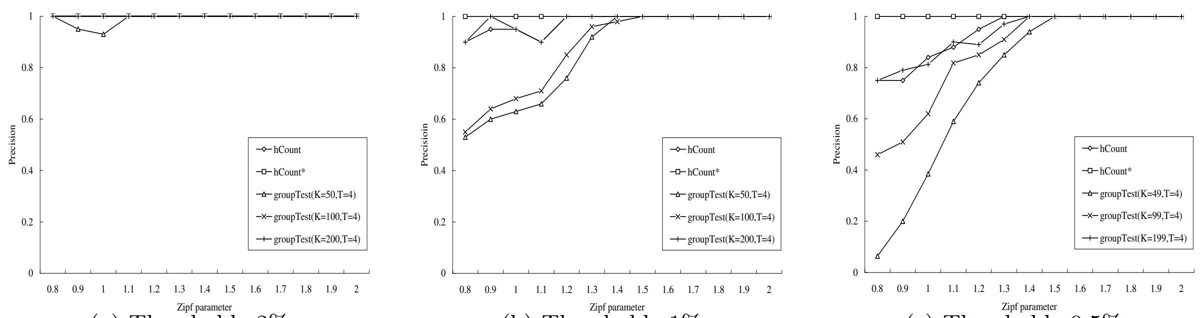


Figure 3: Absolute error between algorithms with different zipf parameter.



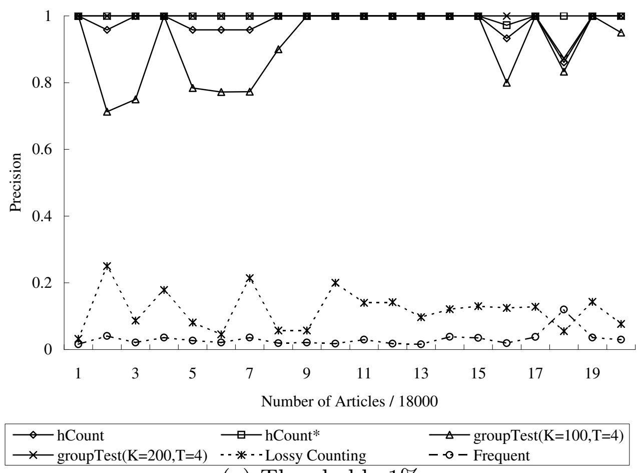
(a) $h = 4$ (b) $h = 8$

Figure 4: Precision on different threshold where zipf is set to 1

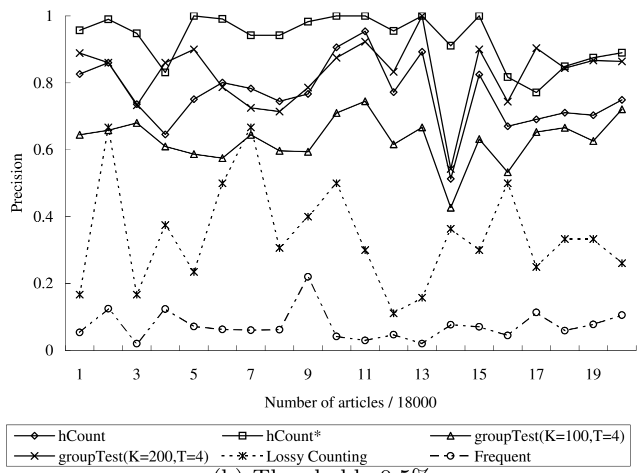


(a) Threshold=2% (b) Threshold=1% (c) Threshold=0.5%

Figure 5: Precision among `hCount`, `hCount*` and `groupTest` with different parameter setting.

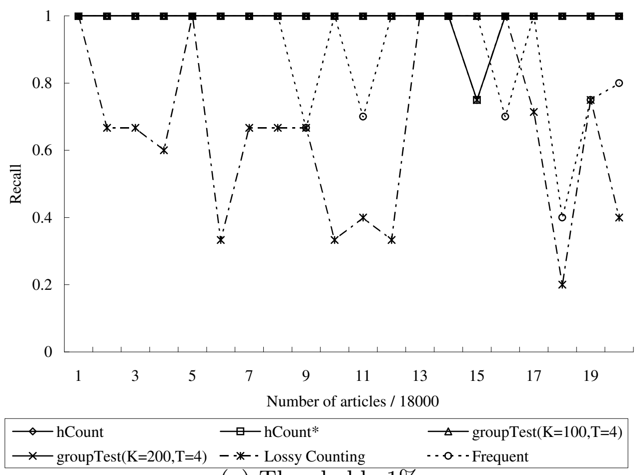


(a) Threshold=1%

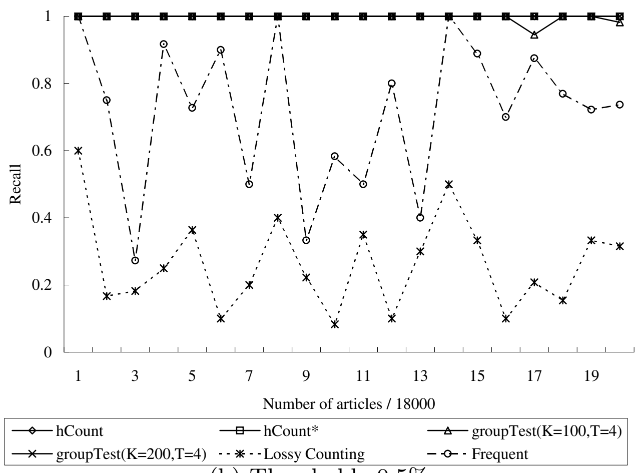


(b) Threshold=0.5%

Figure 7: Precision Comparison on real data



(a) Threshold=1%



(b) Threshold=0.5%

Figure 8: Recall comparison on real data

digit: 31, 56 and 100 respectively. For time being, hash tables are created to extend the range to about 1,000,000, 10,000,000 and 100,000,000. Figure 9 shows the precision when different memory spaces are used. We observe that if more memory space is given, our algorithm can handle a larger range with high precision reserved. When the memory space is less than 11KB, the precision of our algorithm is low. But when the memory space is doubled, the precision increases significantly, about 80% for `hCount` and 90% for `hCount*`.

5. CONCLUSIONS

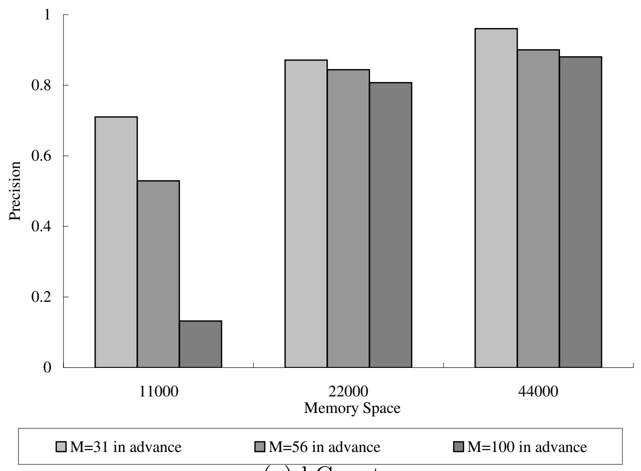
In this paper, we propose a novel hash-based approach to output a list of most frequent items over a data stream. Our approach can cope with both insertion and deletion transactions. In theory, the space required in our algorithm is no more than $\frac{e}{\epsilon} \cdot \ln(-\frac{M}{\ln \rho})$ counters, and only $\ln(-\frac{M}{\ln \rho})$ counters are needed to be updated per transaction. Our approach does not rely on the preknowledge on the range of the data stream, which is rather difficult to obtain, and can handle range extension dynamically. Our algorithms significantly outperform the best known `groupTest` algorithm in terms of precision, recall, memory consumption and processing time.

6. ACKNOWLEDGEMENTS

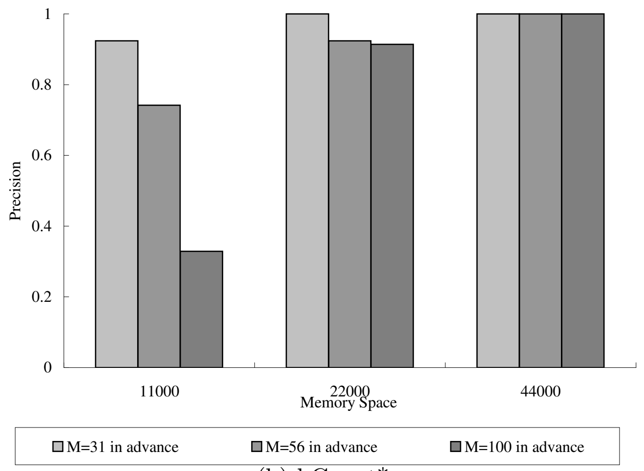
The authors thank Zhiyuan Cai and Zhihong Chong for their helpful suggestions.

7. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of VLDB*, 1994.
[2] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. of ACM STOC*, 1996.
[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issurs in data streams. In *Proc.*



(a) hCount



(b) hCount*

Figure 9: Extend the range by creating a series of hash tables.

of ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 2002.

[4] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[5] B. Boyer and J. Moore. A fast majority vote algorithm. *Technical Report 35, Institute for Computer Science, University of Texas*, 1982.

[6] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. of the 29th ICALP*, 2002.

[7] S. Cohen and Y. Matias. Spectral bloom filter. In *Proc. of ACM SIGMOD*, 2003.

[8] G. Cormode and S.Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. In *Proc. of ACM PODS*, 2003.

[9] G. Cormode and S.Muthukrishnan. Improved data stream summary: The count-min sketch and its applications. In *http://dimacs.rutgers.edu/∼graham/*, June, 2003.

[10] E. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *Proc. of 10th Annual European Symposium on Algorithms*, 2002.

[11] D.-Z. Du and F. Hwang. Combinatorial group testing and its applications. *Applied Mathematics, World Scientific*, 3, 1993.

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

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

[14] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proc. of VLDB*, 1998.

[15] M. Fischer and S. salzberg. Finding a majority among n votes: Solution to problem 81-5. *Journal of Algorithms*, 3(4):376–379, 1982.

[16] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proc. of ACM SIGMOD*, 1998.

[17] R. Karp, C. Papadimitriou, and S. Shenker. A simple algorithm for finding frequent elements in sets and bags. *Transactions on Database Systems*, 2003.

[18] Z. Li and K. A. Ross. Perf join: An alternative to two-way semijoin and bloomjoin. In *Proc. of CIKM*, 1995.

[19] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. of VLDB*, 2002.

[20] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2:143–152, 1982.

[21] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558, 1990.