Error-Adaptive and Time-Aware Maintenance of Frequency Counts over Data Streams*

Hongyan Liu[1], Ying Lu[2], Jiawei Han[2], and Jun He[3]

[1] Tsinghua University, China 100084
hyliu@tsinghua.edu.cn
[2] University of Illinois, Urbana-Champaign, USA 61801
{yinglu, hanj}@uiuc.edu
[3] Renmin University of China, China 100872
hejun@ruc.edu.cn

Abstract. Maintaining frequency counts for items over data stream has a wide range of applications such as web advertisement fraud detection. Study of this problem has attracted great attention from both researchers and practitioners. Many algorithms have been proposed. In this paper, we propose a new method, *error-adaptive* pruning method, to maintain frequency more accurately. We also propose a method called *fractionization* to record time information together with the frequency information. Using these two methods, we design three algorithms for finding frequent items and top-*k* frequent items. Experimental results show these methods are effective in terms of improving the maintenance accuracy.

1 Introduction

With the emergence of data stream applications, data mining for data streams has attracted great attention from both researchers and practitioners. Among the mining tasks for stream data, maintaining frequency counts over data streams is a basic mining problem with a wide range of applications, such as web advertisement fraud detection and network flow identification[1][2]. A number of algorithms have been proposed to tackle this problem [1] [2] [3] [4] [5] [6] [7] [8]. A comprehensive introduction to these algorithms is given in reference [2]. Most of these algorithms are designed to maintain a set of approximate frequency counts satisfying an error requirement within a theoretical memory bound, and they are mostly false-positive oriented. Usually the error bound is given by an end user. To satisfy this error bound, different algorithms use different methods to consume as less memory as possible. Among these algorithms, an algorithm called *space-saving* [2] uses an integrated approach for finding both frequent items and top-*k* frequent items. Both theoretical analysis and experimental results show that this method achieves a better performance in terms of accuracy and memory usage compared to other algorithms, such as

* This work was supported in part by the National Natural Science Foundation of China under Grant No. 70471006 and 70321001, and by the U.S. National Science Foundation NSF IIS-02-09199 and IIS-03-08215.

J.X. Yu, M. Kitsuregawa, and H.V. Leong (Eds.): WAIM 2006, LNCS 4016, pp. 484 – 495, 2006.
© Springer-Verlag Berlin Heidelberg 2006

GroupTest [3], *FREQUENT* [4], *CountSketch* [7] and *Probabilistic-InPlace* [4]. However, after studying these existing algorithms, we have following observations:

- Timestamp information is ignored at processing each data arrival. Stream data are temporally ordered, fast changing, massive, and potentially infinite sequences of data. So time dimension is an important point of view to look at the data. Also, people are usually interested in recent changes of the data streams. However, as far as we know, all of these existing algorithms for approximating the frequency counts do not take this kind of information into account.
- Precision and recall may not be enough to measure the performance of an algorithm. Many algorithms use precision and recall as important measures to judge if an algorithm is good. However, precision and recall depend on minimum support parameter (*minsup* in short) and top-*k* parameter (*k* in short). For high *minsup* or low *k* value in a skewed data stream, they are usually 1. From these two measurements, it is hard for us to know how well an algorithm does for maintaining the frequency counts as a whole. For example, if we use 10000 counters to monitor frequency counts of items over a data stream with length of 100,000 and 10000 distinct items. More than 50% of the frequency counts maintained by *space-saving* are 1, and in the meantime they also have the highest estimation error among all the counts maintained, while the exact answer tells us that only 4% of these counts are 1. But this aspect is not easy to be seen from precision and recall.

In this paper we focus on addressing these two issues described above. We propose three algorithms: *SSTime*, *Adaptive,* and *AdaTime*. Following are some contributions made in this paper:

- We propose to make use of time dimension information when designing the pruning strategy. In order to do that, we propose a method, called *fractionization* , to compress timestamp of each arrival for an item into existing count and error data. We also propose several methods to utilize this information to achieve better pruning result.
- We propose to use the sum of *maintained error* and the error of *estimation error* as well as the sum of *all errors* to measure the quality of mining algorithm. In order to improve the quality of mining results in terms of these measurements, we develop a pruning strategy, called *error-adaptive pruning*, to prune items adaptively so that the error bound can be achieved and in the meantime a low maintained error can also be achieved.
- We develop and implement an algorithm named *Adaptive* to use *error-adaptive pruning* technique to maintain frequency counts over data streams. Comprehensive experimental studies indicate that this algorithm can achieve better performance.
- We design and implement two algorithms, *SSTime and AdaTime,* to extend the existing *space-saving* algorithm and the new algorithm *Adaptive* by taking the time dimension into consideration. Experimental results show that time information is effective in terms of improving the mining quality.

The remainder of the paper is organized as follows. Section 2 describes how to keep and use time dimension, and give the description of algorithm *SSTime*. Section 3 describes the error-adaptive pruning technique, and presents two new algorithms

Adaptive and *AdaTime*. Section 4 gives three measures as a complement to existing measures for performance study and presents experimental results, and section 5 concludes the paper.

2 Keeping and Using Time Information

In this section, we describe our method to consider time information while summarizing the dynamic data stream.

2.1 Problem Definition

Let I be a set of single items, $I = \{e_1, e_2, \ldots, e_n\}$. Given a single item stream D of size N, an item e_i is called *frequent* if its frequency count, f_i, in this stream exceeds a user-specified support $\lceil \varphi N \rceil$, where φ is a user-specified threshold, called minimum support (*minsup*). An item e_j is called a *top-k frequent item* if its frequency count is among the k highest frequencies, where k is specified by user.

For a data stream application, the exact solution of finding all of the frequent items or finding all of the top-k frequent items is usually impractical due to time and space limitation. Therefore, the problem becomes finding an approximate set of frequent items and top-k items. To solve this problem, except for the parameter *minsup* and k, an error rate, ε, is also given by a user. With the relaxation of the original problem, the task of mining frequent items becomes finding all of frequent items whose estimated frequency counts exceeds $\lceil \varphi N \rceil$, where the difference between the estimated counts and their true counts is at most εN. Similarly, the task of finding top-k frequent items becomes finding k items with highest estimated frequency counts, where the difference between the estimated counts and their true counts is also at most εN.

2.2 *Fractionization:* A Method to Keep Time Information

Existing algorithms for mining frequent items in data streams can be categorized into two kinds of techniques: *counter-based* and *sketch-based*. Counter-based method use an individual counter for each item monitored. In this paper we only discuss this method.

Due to the space limitation and the big size of the stream, usually only a subset of all of items can be monitored in the main memory. Suppose we use m counters in memory to keep frequency counts, then at any point of time, only m distinct items are monitoring.

Almost all of the counter-based algorithms use the following method to maintain item's frequency. If the newcome item is currently monitored, its frequency is increased. Otherwise, an item currently monitored is pruned to make room for the new item. Although these existing algorithms are different from each other in terms of pruning method, they all neglect the time information of each item arrival. In real applications, items in data stream are changing as time changes. For example, old frequent items may become infrequent as time goes on. Therefore, a straightforward way to use time information is that whenever pruning is required, among candidates,

we choose old one instead of recent one to prune. But how can we judge which one is older than others?

The answer to this question depends on how time information for every arrival of items is recorded. If we have enough space, it is easy to record time information. But in order to achieve high accuracy, we need to use as less memory as possible for each counter. Therefore, how to put time information into existing information that a counter keeps is important.

Suppose for each counter we maintain three pieces of information for an item: *key*, *guaranteed count,* and *maximum error*, which can be represented as a triple (*item, count, error*), where each element of this triple is usually saved as an integer. Our method to save time information of each item arrival is called *Fractionization* , which means that we first transform the information of each item into a decimal fraction, and then save it as a decimal part of existing triple element such as error. In this case, we use *float* rather than *integer* to represent it. However, even by this way, we still cannot record every occurrence of an item. In order to save space, we sum all of time information of its occurrence, and then save it as a subpart of the error element.

Now the problem becomes how to express the time information of an occurrence of an item. There are many ways to do that. A simple one is that we use the occurrence order to represent the timestamp of each item arrival. For example, the timestamp of the first item in the stream is 1, and second is 2, and so on. In this way, since the length of stream increases continually, the sum of timestamp may become very big. After *fractionization* , it may become very small. To prevent this problem, before *fractionization* , we can do logarithm computation such as natural logarithm. Taking natural logarithm as an example, in order to transform the sum of timestamp into a decimal, we can get the inverse of this number. So the sum of time stamp should be greater than one. As a result, if we use natural logarithm computation, the time stamp of the first item in the stream could be 3.

In sum, we could use the following formula (1) to record the time information of a monitored item (e_i, $count_i$, $error_i$):

$$error_i = error_i + \frac{1}{\ln(\sum_{j=1}^{count_i} timestamp_j)} \tag{1}$$

Besides the linear sum of the timestamp information of an item's each monitored arrival, we can also record the square sum of the timestamp information by a similar way as shown in formula (2). Here, we put the time information in the *item* element, and the timestamp of an item's arrival can use the natural logarithmic value of its occurrence order. For example, the timestamp of the first item in a stream is *ln*(3).

$$e_i = e_i + \frac{1}{\ln(\sum_{j=1}^{count_i} timestamp_j^2)} \tag{2}$$

2.3 Algorithm: *SSTime*

To show the effectiveness of using item's time information, we integrate the time keeping and using method with the *space-saving* algorithm [2]. The algorithm called *SSTime* is outlined in Fig.1.

This algorithm is similar to *space-saving*. There are two differences between them. The first is that *SSTime* records not only the *count* and *error* information of an item, but also its time information. In Fig. 1, we use formula (1) to record the time information (line 6-8 and line 14). We can also use both formulae (1) and (2) to record more information about time. When an item is pruned from the memory, i.e., it is not monitored currently, its time information is lost at the same time (line 13-15). We can also record this information in the item that replaces it. The second difference is that when choosing the pruning item, *SSTime* takes time into consideration. Among all of items with the same (*count* + *error*), where *error* means the integer part of the counter's error element, the "oldest" item is chosen to prune first. To judge which item is old is not an easy job. In this algorithm, we use a straightforward method. The smaller the sum of timestamps of an item is, the older the item is. This method is shown in formula (3). We can also use some complex method, which will be discussed in the next section.

$$e_p = \arg\max_{e_i \in candidate}(error_i - (\text{int})error_i) \quad (3)$$

```
Algorithm: SSTime( m counters, stream D)
1       timestamp=2;
2       For each item, e_i, in D {
3       timestamp++;
4       If e_i is monitored by counter (e_i, count_i, error_i) {
5                   count_i= count_i+1;
6       temp=exp(1/(error_i-(int)error_i));
7       temp=temp+timestamp;
8       error_i=(int)error_i+1/ln(temp);
9       }
10      else {
11                  candidate={e_j | e_j has the least value of min=(count+(int)error)}
12                  Let e_p be the "oldest" item among items in candidate
13                  Replace e_p with e_i.
14                  error_i= min+1/ln(timestamp);
15                  The counter for e_i becomes (e_i, 1, error_i)
16      }
17    }
```

Fig. 1. Algorithm *SSTime*. This algorithm is an extending of the algorithm *space-saving* by incorporating time information of items to it.

With the information maintained by this algorithm, at any point of time, a query could be submitted to output all of the frequent items according to a user-specified *minsup*, or to output *k* most frequent items when the user gives the value of *k*. The method to fulfill these two kinds of queries is the same as given in *space-saving*, and we do not give them here due to the space limitation. This is the same for the other two algorithms which will be described in the following sections.

3 Error-Adaptive Pruning Method and Algorithm

3.1 *Error-Adaptive* Pruning Method

As discussed in section 1, using the pruning method proposed in *space-saving*, most of the frequency counts maintained in memory have only one guaranteed frequency count, whereas they have the highest estimated error. In other words, most of them are very untrustworthy, and the estimation error as a whole is high. In order to improve this, we propose a new pruning method, called *error-adaptive*.

The pruning method used in *space-saving* is that whenever an existing monitored item needs to be pruned, one of the items (we call them *candidate items*) with the minimum estimated count, i.e. (*count* + *error*), is selected. The problem of this method is that among these candidate items, some have very high guaranteed counts, and others have only one guaranteed count. Treating them equally during pruning will lead to high estimation error. Therefore, in our new pruning method, we try to treat them differently, and in the meantime, we need to guarantee the error rate and high recall and precision. This method is shown in Definition 2.

Definition 1. *(pruning point N) A time point is called a pruning point if at this time point, a new coming item in data stream cannot find a counter to monitor its frequency count. Let the current length of the stream is N, then this pruning point is called pruning point N.*

Definition 2. *(error-adaptive pruning method) Suppose user-specified error rate is ε, at pruning point N. Let* $ecount_i$ *be the estimated count,* $(count_i + error_i)$ *for each monitored item* e_i*. The error-adaptive pruning method selects all of items* e_j *satisfying both of the following conditions as candidate items:*

1) $\text{ecount}_j \le N/m$ where $m=\lceil 1/\varepsilon \rceil$
2) $count_j=\min(count_i)$ i=1, 2, …, m

At pruning point N, the Nth item, e_n, of the stream comes, and one of the candidate items is selected. Suppose the counter for the selected item is (e_p, $count_p$, $error_p$). Then after pruning, this counter becomes (e_n, 1, $count_p + error_p$) and is used to monitor e_n.

Using error-adaptive pruning method, we have the following lemmas.

Lemma 1. Let N be the current length of a data stream, then at any time point the following equation (4) holds.

$$N = \sum_{\forall i | e_i \text{ is monitored}} (count_i + error_i) \quad (4)$$

Proof. Each item arrival in data stream D only increases one counter's count by 1. This is obviously true when this item is currently monitored. Even when it is not monitored, it will replace one existing item. The counter for the existing item will be used to monitor the new arrival item. This counter's original *count* and *error* will be saved to *error* and its *count* will be set to 1. So the count of old arrival is kept, and the new arrival is also recorded. Hence, at any time point, the summation of any counter's *count* and *error* equals the number of item arrivals currently in data stream.

Lemma 2. At any pruning point N, there is always at least one candidate item that can be found to prune.

Proof. Lemma 1 means at any pruning point there is at least one monitored item satisfying ($count + error$) $\leq N/m$. The proof is by contradiction. Assume every item monitored has an estimated count $> N/m$, then the sum of the estimated counts of m counters must satisfy: sum($count_i + error_i$) $> N/m*m = N$, which is contradictory to Lemma 1.

Lemma 3. Using error-adaptive pruning method, the frequency count estimation error rate for any item is not greater than ε.

Proof. Items can be classified into two categories: items that are monitored currently, and items that are not monitored currently. For those monitored, if it is monitored before all of the counters are used up and have not been pruned yet, its estimation error is zero, which is obviously less than ε. If it is monitored at the pruning point N by replacing a monitored item, then its error should be less than or equal to $N\varepsilon$ according to definition 2. That is to say, its error rate ($error/N$) is not greater than ε. For those not monitored, we regard its frequency count zero. Suppose it is last pruned at the pruning point N, then according to definition 2, before its pruning, the sum of its count and error (i.e., $count + error$) must be less than or equal to $N\varepsilon$. Since its estimated count is zero, the maximum error is ($count + error$), which is not greater than $N\varepsilon$. Therefore, the lemma also holds for this case.

Using this error-adaptive pruning method for mining task given in section 2.1, the output will only include false positive, no false negative. This is already proven in algorithm *space-saving*. In *space-saving*, at every pruning point, the error for the new coming item is overestimated as the minimum estimated count, which is min($count + error$). By our method, the error estimated is no less than min($count + error$), so it is also an overestimation. Therefore, there is only false positive among output frequency count. This is also demonstrated by comprehensive experimental study results.

Based on this error-adaptive pruning method, we propose two algorithms, *Adaptive* and *AdaTime*, for finding frequent items and top-k frequent items.

3.2 Algorithm: *Adaptive*

Adative is the algorithm we design for finding frequent items and top-k frequent items based on *error-adaptive* pruning method. It is depicted in Fig.2.

In this algorithm we do not consider time information. Based on user-specified error rate ε, we use m ($=1/\varepsilon$) counters to monitor items in stream D. When a new item arrives in the stream, if it is currently monitored, its count is increased by one (lines 5-6). If it is a pruning candidate, we delete it from the candidate set (line 7). If it is not monitored and there is no candidate item in candidate set for pruning, a function, *Getcandidate*(), is called to select candidate items from all of counters based on error-adaptive pruning method described in Definition 2 (lines 13-14). Then, one candidate item is randomly picked to prune and make its counter available to the new item (line 15). If it is not monitored, but the candidate set is not empty, we choose one item from the candidates to prune instead of selecting pruning item from all of the counters again (line 15). By doing this, we could save time without affecting error rate. The items in candidate are selected during a former pruning point, say N. At that point,

each of them satisfies (*count* + *error*) ≤ $N\varepsilon$. Suppose the current pruning point is M, (M > N), then items in candidate satisfy (*count* + *error*) ≤ $N\varepsilon \le M\varepsilon$ too. After pruning an existing item, its counter is incremented and used to monitor the new item (lines 16-18).

```
Algorithm: Adaptive(m counters, stream D)
1  n = 0;
2  candidate={};
3  for each item, e_i, in stream D {
4       n = n+1;
5       if e_i is monitored by counter (e_i , count_i, error_i) {
6           count_i = count_i +1;
7           If e_i is in candidate, erase it from candidate
8       }
9       else {
10          if there is a free counter to use
11              New counter (e_i , 1, 0) for e_i;
12          else {
13              if candidate is empty
14                  candidate=GetCandidate(m, n);
15              Let e_p be one of the items in candidate
16                  Replace e_p with e_i
17              error_i=count_p+error_p;
18              The counter for e_i becomes (e_i, 1, error_i)
19          }
20      }
21  }
```

```
Function GetCandidate(m counters,
n current length of stream D)
1 min=n;
2 for each item, e_i, monitored currently {
3       if (count_i+(int)error_i <= n/m) {
4           if (count_i = =min) then
5               put e_i in candidate;
6           else if count_i <min {
7               min = count_i;
8               empty candidate;
9               put e_i in candidate;
10          }
11      }
12 }
13 return candidate;
```

Fig. 2. This is the main procedure of algorithm *Adaptive*

The function *GetCandidate*(*m*, *n*) is called to find all of the candidate items from *m* counters at pruning point *n*. This is done by traversing from counters with the minimum estimated count, ($count_i$ + $error_i$). We use the same data structure used in *Space-saving*. All of the counters with the same estimated count are attached to a bucket, and all of the buckets are linked together according to the estimated count value. Therefore, when traversing buckets from the one with the lowest estimated count, once this value is greater than $n\varepsilon$, we could stop further traverse.

3.3 Algorithm: *AdaTime*

To show the effect of the time information to the error-adaptive pruning method, we propose another algorithm, *AdaTime*, which is outlined in Fig. 3.

The major difference between algorithms *Adaptive* and *AdaTime* is shown in lines 7, 15 and 18. In line 7, we record time information together with count and error information in the counter. We can use the same method used in algorithm *SSTime*. Here we introduce another way. Suppose the timestamp for the nth arrival is $ln(n+2)$, then we could put linear sum of each timestamp of this item to *error,* and put the square sum of each timestamp in the *key* of the item. We use the *fractionization* method introduced in section 2 to do that. In line 15, instead of randomly picking one item from the candidate set, we choose the relatively old item to prune. To decide which item is older, we can use the linear sum of the timestamps and square sum of the timestamps to compute a distance between the occurrences of this item and the

new coming item. Due to the space limitation, we do not give the further detail of this method. The larger the distance is, the older the item is. Similar to line 7, in line 18, at the pruning point, time information is also recorded.

```
Algorithm: AdaTime(m counters, stream D)
1  n = 0;
2  candidate={};
3      for each item, e_i, in stream D {
4           n = n+1;
5           if e_i is monitored by counter(e_i , count_i, error_i) {
6                count_i = count_i +1;
7                Record timestamp information;
8                If e_i is in candidate, erase it from candidate
9           }
10          else {
11               if counters# < m, create a new counter  for e_i;
13               else {
14                       if candidate is empty, candidate=GetCandidate(m, n);
15                       Let e_p be "oldest" items in candidate
16                       Replace e_p with e_i by counter (e_i, 1, error_i)
17               error_i=count_p+(int) error_p;
18                       Record timestamp information;
19               }
20          }
21     }
```

Fig. 3. Algorithm *AdaTime* is an algorithm using *error-adaptive* pruning method, and it also considers time information when do pruning

4 A Performance Study

4.1 Measures

In order to evaluate performance of an algorithm completely, besides the measures such as recall, precision, space, and time, we propose three other measures to evaluate the effectiveness of various pruning method.

Let $|I|$ be the number of distinct items in a data stream, and m be the number of counters used to maintain frequency counts for these items. The first measure is the *average absolute error of all items*, or *aError* in short. It is defined in formula (5). The second is the *average absolute error of maintained counts*, or *mError* in short, as shown in formula (5).

$$aError = \frac{\sum_{i=1}^{|I|} truecount_i - count_i}{|I|} \qquad mError = \frac{\sum_{ei\ monitored} truecount_i - count_i}{m} \tag{5}$$

The third is *the average absolute error of maintained error*, or *eError* in short, as shown in formula (6).

$$eError = \frac{\sum_{ei\ monitored} error_i - (truecount_i - count_i)}{m} \tag{6}$$

We have implemented the three algorithms proposed in this paper in C language and run them on a Pentium IV 2GHz *IBM Thinkpad* laptop with 1.5G memory running Window 2003 Server system. For algorithm *SSTime* and *AdaTime*, when we implement them, we have tried several different methods to record and use time information. But due to space limitation, we only report the result of the simple method as shown in Fig. 1.

We use synthetic data generated by following a *Zipf*-like distribution [8].

4.2 Varying the Data Skew

In this set of experiments, we change the skew factor of the data stream, and measure the recall, precision, *aError*, *mError*, *eError*, and time. We fix the number of distinct items to be 100,000, the length of stream to be 10,000,000, and the error rate to be 0.0001. We compare the performance of our algorithms with s*pace-saving* which proves to have better performance than other algorithms in [2], and is implemented to our best knowledge. Since we use the data structure as used in *space-saving*, the space used by our algorithms is similar to *space-saving*. We vary the skew factor from 0.5 to 2, and the results are shown in Fig. 4 and 5.

From Fig. 4 (a) and (b) and Fig. 5 (a) we can see that algorithms *Adaptive* and *AdaTime* produce better error results than *space-saving* and *SSTime*. Furthermore, although it is hard to see from these figures, algorithm *AdaTime* is slightly better than Adaptive, and *SSTime* is slightly better than *AdaTime*.

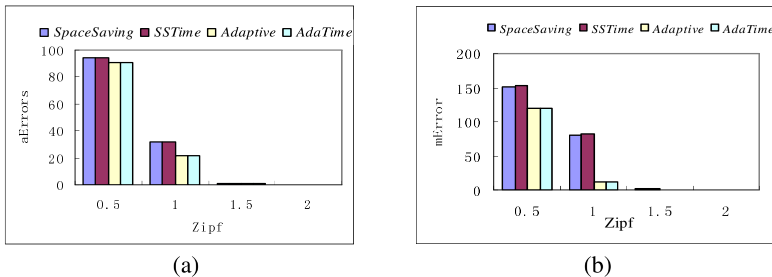


Fig. 4. These two figures show *aError* and *mError* for several data streams with length 10000000 and 100000 distinct items. Their skew factors are changed from 0.5 to 2.

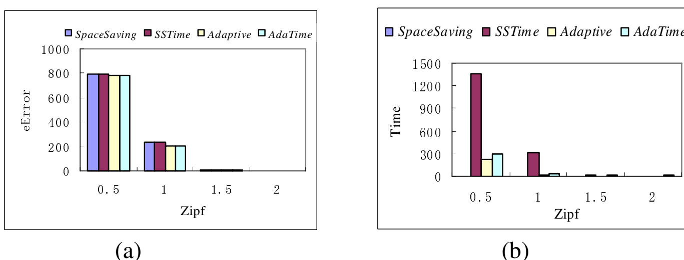


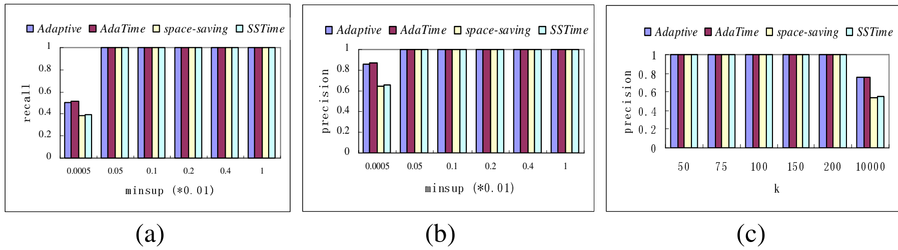
Fig. 5. These two figures show *eError* and *runtime* when running four algorithms for four data streams with length 10000000 and 100000 distinct items. Their skew factors are different.

Fig. 5 (b) indicates that among these four algorithms, *space-saving* is the fastest, and *SSTime* is slowest, while Adaptive is better than *AdaTime*. Since both recording time information and selecting candidate based on time information take more time, it is not difficult to understand this result. The reason why *SSTime* is much slower than others is that at each pruning point, every item with the min(*count* + *error*) is needed to scan and compare.

4.3 Varying the Query Parameters

In this set of experiments, we fix the number of distinct items to be 100,000, the length of stream to be 10,000,000, the error rate to be 0.0001, and skew factor to be 1. We change two parameters, *minsup* and *k*, to see the recall, precision. Since this data set is one of those used in section 4.2, the other measures for this data set remain the same as given above. The results are depicted in Fig. 6.

One can see from Fig. 6 (a) and (b), for low *minsup*, Adaptive and *AdaTime* have better recall and precision than *space-saving* and *SSTime*, whereas *AdapTime* is better than Adaptive and *SSTime* is a little better than *space-saving*. As the top-*k* query, the results for recall are the same as precision, so we do not put the figure here. Fig. 6(c) shows us that for high *k*, these algorithms have the same behavior shown in (a) and (b).



(a) (b) (c)

Fig. 6. (a) and (b) show the recall and precision of four algorithms respectively as *minsup* varies, and (c) shows the precision as *k* varies

5 Conclusions

We study the problem of maintaining frequency counts for items over data streams in this paper. We propose to use time information when pruning items, and give a *fractionization* method to represent and record the time information without spending much space. We also propose a new pruning method, *error-adaptive pruning*, to improve maintenance accuracy as a whole. Using these two methods, we design and implement three algorithms, *Adaptive*, *AdaTime,* and *SSTime*, and conduct comprehensive experiments. Our experimental results show that time information can improve the maintenance accuracy, but needs more runtime. Our results also indicate that the new pruning method is effective for improving accuracy as a whole.

References

1. G. S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proc. of 28th Intl. Conf. on Very Large Data Bases*, pages 346 – 357, 2002.
2. A. Metwally, D. Agrawal, and A. El Abbadi. Efficient. Computation of Frequent and Top-k Elements in Data Streams. *In Proceedings of the 10th ICDT. International Conference on Database Theory,* pages. 398–412, 2005.
3. G. Cormode and S.Muthukrishnan. What's Hot and What's Not: Tracking Most Frequent Items Dynamically. In *Proc. Of 22nd ACM Symposium on Principles of Database Systems (PODS)*, pages 296 – 306, 2003.
4. E. Demaine, A. Lopez-Ortiz, and J. Munro. Frequency Estimation of Internet Packet Streams with Limited Space. In *Proc. of 10th Annual European Symposium on Algorithms*, 2002.
5. C. Jin, W. Qian, C. Sha, J. Yu, and A. Zhou. Dynamically Maintaining Frequent Items Over a Data Stream. In *Proc. Of CIKM*, 2003.
6. J. Yu, Z. Chong, H. Lu, and A. Zhou. False Positive or False Negative: Mining Frequent Item Sets from High Speed Transactional Data Streams. In *Proc. of 30th VLDB*, pages 204–215, 2004.
7. M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. In *Proc. of the Int. Colloquium on Automata, Languages and Programming (ICALP)*, pages 693 – 703, 2002.
8. D. E. Knuth. *The Art of Programming*. Addison-Wesley, 1973.