Maintaining Highly Accurate Frequency Counts over Data Streams: A Practitioner's Perspective*

Ying Lu  Jiawei Han

Department of Computer Science
University of Illinois, Urbana-Champaign
yinglu@uiuc.edu, hanj@cs.uiuc.edu

July 5, 2005

Abstract

Maintaining frequency counts for data streams has attracted much interest among the research community recently since it provides the base for many stream mining applications. Most existing work followed the same paradigm: Given an error requirement, find an algorithm that maintains approximate frequency counts satisfying the error requirement within a theoretical memory bound. While actual algorithms are different, the following common weakness has been observed. First, most algorithms are satisfied with having the maintained counts within a certain error bound and are not concerned with maintaining the counts of individual items as accurate as possible. Second, most work is more theoretical in the sense that they focus on finding the theoretical memory bounds with less consideration for estimating the memory requirement for a given application. The bound provided can hardly be used as an accurate gauge of the true memory requirement that often varies drastically depending on the data characteristics and arrival orders.

In this paper, we study the problem of maintaining frequency counts over data streams of infinite length from a practitioner’s perspective. Given a fixed memory size, maintain frequency counts for individual items as accurately as possible. We introduce a novel one-pass deterministic algorithm, Progressive Lazy Pruning (PLP). Given a fixed memory size, PLP employs a progressive pruning technique that can make full use of available memory to maintain highly accurate approximate counts for items in data streams. The estimation error is not only bounded but also independent of the total number of arrivals. If an error requirement ε is specified, it can also maintain ε-approximate count as most existing work does. Our performance study indicates that, in comparison with one of the best existing deterministic algorithms on frequency counting, PLP achieves higher accuracy using the same amount of memory over various kind of data distribution.

*The work was supported in part by the U.S. National Science Foundation IIS-0249299, IIS-0548215. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.
1 Introduction

The problem of frequency counting is widely used in data mining applications, such as frequent patterns, association rule mining, and top-k queries [14], and it has been widely studied by researchers. With the recent emergence of data stream applications, researchers have started to focus their attention on the mining problems in the data stream environment. Data streams have the following characteristics: (1) the length of the data stream is unbounded thus it is impossible to store the entire content, and (2) data items usually arrive at a very fast speed, so a one-pass online processing algorithm is necessary as it is impossible to re-read the data items. Furthermore, with a finite amount of memory, it is impossible to keep track of the exact count of each item in a stream. Therefore, in the context of data streams, the frequency counting problem has been addressed by maintaining so-called $c$-approximate counts [2]. Given a data stream with $N$ arrivals, $c$-approximate counts are a set of pairs of (item, count) where (1) the count associated with an item is smaller than the item’s true count, but by at most $cN$, and (2) any item whose true frequency exceeds $cN$ belongs to the set; other items may or may not. A number of algorithms have been proposed [1, 3, 5, 4, 6, 8, 9, 14]. They can be grouped into two classes: Deterministic and probabilistic. Deterministic algorithms are able to maintain $c$-approximate counts with a guaranteed user-specified $c$. Probabilistic algorithms can only maintain count of the items within $cN$ of their true count with a certain probability.

The work reported in this paper was inspired by the following observations. First, most of proposed algorithms are more theoretical in the sense that they focus on finding the minimum theoretical memory bounds for a given $c$. There seems to be no work that considers how to determine required memory for a given application to guarantee the given $c$. This is because the actual memory required often varies in orders of magnitude depending on the data characteristics and arrival orders, as shown by some performance studies [14]. Second, most existing algorithms aim at keeping the maintained counts within a given error bound instead of focusing on providing count information for individual items as accurately as possible. One of the possible reasons is that most work has been rooted upon finding frequent items. However, in many data stream applications not only the frequent items but also their actual counts are of interests. For example, in financial applications, investors are not only interested in active stocks frequently traded but also the actual volume of each transaction.

With the above observations, we tackled the problem of frequency counting from a practitioner’s perspective. First, we started with the easiest parameter that a user can get: the available memory. That is, instead of giving an error requirement, the user only need to specify the size of memory available. Second, instead of counting frequent items, we aim at maintaining the frequency counts for items of the input data stream as accurately as possible using the given memory. Given an item $i$, the algorithm can return its approximate frequency count, $\text{approx.count}_i$ and estimation error $\text{error.bound}_i$. That is, we like to have as tight of an error bound as possible.

2
As the result of the first phase of our study, we present here a one-pass deterministic algorithm, *Progressive Lazy Pruning* (PLP), for maintaining highly accurate $\epsilon$-approximate frequency counts of items in data streams using a fixed size of memory. Different from most existing algorithms, where count information is often excessively pruned when there is still available memory, PLP adopts a lazy pruning approach. That is, it makes full use of available memory and conducts pruning only when it is really necessary. Furthermore, in order to maintain frequency counts which are as accurate as possible, pruning is conducted progressively, depending on the arrival of items. Instead of giving a single error bound to all items, counts of different items will have different approximation errors.

The contributions of our work can be summarized as follows.

1. Differing from most existing algorithms on frequency counting over data streams, we propose to study the problem of frequency counting in the data stream environment from a new and practical perspective: (1) Given a fixed memory space, derive algorithms that can maintain $\epsilon$-approximate frequency counts with the estimation error as low as possible, and (2) provide users with highly accurate frequency count information for individual items in data streams.

2. We present the Progressive Lazy Pruning (PLP) algorithm and its variation, Epsilon-Progressive Lazy Pruning (Epsilon-PLP). The algorithms are able to maintain $\epsilon$-approximate frequency counts with fixed amount of memory for a virtually infinite data stream whose elements are drawn from a finite set of items. The processing time for each arriving item is $O(1)$ and the bounded error is independent of the total number of arrivals. Furthermore, while the approximation error is bounded, some items may have tighter bound so that more accurate counts are available for those items.

3. We conduct a set of experiments to illustrate the effectiveness of the algorithm. Our experimental results verify the analysis of the algorithm. A comparative study with Lossy Counting, one of the best existing algorithms, indicates that our algorithm can achieve higher accuracy using the same amount of memory over various data distributions, even for situations when data is highly skewed.

The remainder of the paper is structured as follows. In Section 2 we provide the detailed description of our algorithms, Progressive Lazy Pruning and its variation Epsilon-Progressive Lazy Pruning. In addition, we also present an algorithm for getting the count information for an individual item. Section 3 presents the results of an experimental performance study that compares our algorithm with Lossy Counting, the best existing algorithm on frequency counting in data streams. A brief summary of related work is given in Section 4. Section 5 concludes the paper.
2 Maintaining Highly Accurate Frequency Counts Over Data Streams

In this section, we describe our deterministic algorithms for maintaining frequency counts over data streams using a fixed amount of memory. Given a fixed memory size, our algorithms are able to provide highly accurate count information for individual items by making full use of available memory, and have error bounds that are independent of the total number of arrivals. We first present the main algorithm, Progressive Lazy Pruning (PLP), which adopts the progressive pruning technique to ensure full memory usage and high accuracy of the maintained counts. We follow with a variation of PLP called Epsilon-Progressive Lazy Pruning, which maintains ε-approximate counts [2].

For the ease of discussion, Table 1 lists some of the notation used in this paper.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>total number of arrivals</td>
</tr>
<tr>
<td>$f_i$</td>
<td>actual frequency count for item $i$</td>
</tr>
<tr>
<td>$\tilde{f}_i$</td>
<td>maintained frequency count for item $i$</td>
</tr>
<tr>
<td>$e(c_i)$</td>
<td>the count that are missing from the approximate count ($c_i = f_i - \tilde{f}_i$)</td>
</tr>
<tr>
<td>$e(\varepsilon_i)$</td>
<td>the estimation error w.r.t. stream length, $N$ (for item $i$, $\varepsilon_i = \frac{\varepsilon}{N}$)</td>
</tr>
</tbody>
</table>

Table 1: Notation

2.1 The Progressive Lazy Pruning Algorithm

Recall that we are given a fixed amount of memory and the input data stream is infinite. The pruning of items is necessary when no memory space is available for new items. The key issue here is when and how to prune items. In this subsection, we provide an outline of our algorithm, Progressive Lazy Pruning (PLP), and describe, in detail, the core of PLP, progressive pruning, in the next subsection.

A pruning point is a point in the stream where a newly arrived item does not have an existing entry in memory and there is no free space available for the insertion of the item. Entries in memory are pruned only at pruning points. PLP assigns each pruning point a pruning id, $p.id$. When the algorithm starts, $p.id$ is set to 0 and it gets incremented by one at each pruning point. At each pruning point, items whose maintained counts are less than or equal to a certain threshold are pruned to free memory to accommodate new items. Items arrived between pruning point $k$ and $k+1$ are associated with $p.id = k$. Every memory entry has the form of triplets $(item, count, p.id)$, where item is a data item, count is its maintained frequency count, and $p.id$ is the most recent pruning id when the entry was created. PLP also stores the threshold information at each pruning point which is used for future pruning.
Algorithm 1 Progressive Lazy Pruning (PLP)

**Input:**

- \( D \): streams of single item transactions.

**Output:**

- \( M[item, cnt, p.id] \) where \( item \) is the item, \( cnt \) is the frequency count of \( i \), and \( p.id \) is the pruning id of \( i \);
- \( P[n, cnt] \) where \( n \) is the number of arrivals up to the pruning point, \( cnt \) is the maximum count of the items pruned at the pruning point.

**Description:**

1. \( n := 0; p.id := 0; \)
2. **for** each arriving item \( i \) **do**
3. \( n := n + 1; \)
4. **if** an entry \( M[i] \) is found such that \( M[i].item = i \) **then**
5. \( M[i].cnt := M[i].cnt + 1; \)
6. **else**
7. **if** no more free space for a new entry **then**
8. \( p.id := p.id + 1; \)
9. \( P[p.id].n := n; /* may be used in PruneItems */ \)
10. \( threshold := PruneItems(p.id, M, P, n); \)
11. **end if**
12. \( \text{NewEntry}(i, 1, p.id); \)
13. **end if**
14. **end for**

Algorithm \( \text{ref:algo:PLP} \) provides a conceptual outline of the Progressive Lazy Pruning algorithm. For actual implementation, much more efficient storage can be used for memory entries and pruning information. When an item \( i \) arrives, if it is found in \( M \), we increase its frequency count by one (line 4-5). Otherwise, a new entry is created with \( count = 1 \) and \( p.id \) equal to the current \( p.id \) (line 12). If there is no free space for the new entry, a new pruning point has been reached. In this case, the following steps are performed: (1) a new pruning id is assigned, (2) the total number of arrivals is set (line 7-9) and (3) Function \( \text{PruneItems} \) is called to prune some entries from \( M \) (line 10). The core of PLP is the function \( \text{PruneItems} \) which performs the actual pruning. We discuss this function in the following subsection.

### 2.2 Progressive Pruning Strategy

In this section, we describe in detail the algorithm, \( \text{Progressive Pruning} \), used by the function \( \text{PruneItems} \). We show that, given a fixed memory size, progressive pruning makes full use of available memory space to achieve high estimation accuracy. In fact, we show that, the estimation error of the frequency counts are independent of the total...
number of arrivals. We also show that, given an item \(i\), we can return its approximate count and its associated estimation error.

At each pruning point, entries with count less than or equal to a certain pruning threshold are deleted. This deletion causes the difference between the actual count, \(f_i\), and the maintained approximate count, \(\bar{f}_i\), of an item \(i\). Since an item can only be pruned and never be counted more times than the number of actual arrivals, the error is always non-negative.

One straightforward pruning strategy is to prune items having the minimum count (usually equals to 1), at each pruning point. Such a strategy may cause flushing. With increased arrivals, fewer free entries will be available thus causing pruning to occur very frequently. In the worst case, where there is only one free entry, each new arrival that does not have any existing entry would cause pruning to take place. Such flushing would increase the estimation error dramatically.

Let us use an example to illustrate. Suppose we have a data stream \(S\) of infinite length, and the memory can only hold four entries. Figure 1 shows a snapshot of the first thirty items of data stream \(S\). \(P_i\) denotes points where pruning is conducted. The tables show the memory contents in \((item, count)\) pairs before \(P_1, P_2, P_3\), and after first thirty arrivals. Item \(a\) arrived early and has accumulated count of three before \(P_1\). At \(P_1\), since the minimum count is one, items \(c\) and \(e\) are pruned. Between \(P_1\) and \(P_2\), \(f\) and \(c\) arrived twice and \(b\) arrived once. At \(P_2\), since the minimum count is two, \(a\) is still not pruned. Instead, newly arrived items \(f\) and \(c\) are pruned. The same thing repeats for the rest of pruning points. The total number of arrivals for each item for the first thirty arrivals are: \(a: 4, b: 3, c: 7, e: 2, f: 9, \) and \(g: 5\). The maximum count loss due to pruning is twelve \((1+2+2+2+1+1+1+1+1)\). It is obvious that this algorithm has strong bias toward items that arrived early and has a tendency to cause the items which arrive later be pruned repeatedly. The result is that the memory is being occupied by old items that might no longer active, and new items have hard time to have their counts maintained in memory, thus increasing the estimation error drastically.

\[
\begin{array}{cccccccccc}
P_0 & P_1 & P_2 & P_2 & P_3 & P_3 & P_4 & P_5 & P_6 & P_7 \\
S: & a, b, a, c, a, b, f, c, b, f, c, g, f, f, g, & c, f, c, f, g, f, e, c, a, c, f, g, f, g, & a, b, c, g \\
\begin{array}{c|ccc|c|ccc|c|ccc|c}
a & b & e & e & a & b & f & e & a & b & g & f & a & b & c & g \\
3 & 2 & 1 & 1 & 3 & 3 & 2 & 2 & 3 & 3 & 2 & 2 & 3 & 3 & 2 & 1
\end{array}
\end{array}
\]

Figure 1: No. of pruning points for simple pruning

To overcome this problem, we propose a strategy called progressive pruning. When using this strategy, memory entries are pruned based on two factors; (1) their maintained count, and (2) their pruning id, i.e., the time they are created.

**Definition 1. (Progressive Pruning Strategy)** Let \(k\) be the current pruning id, and \(f_{\text{min}}\)
the minimal count kept in $M$. Let $t_j$ be the minimum frequency count among all items in $M$ at pruning point $j$, i.e., $t_j = f_{\text{min,}j}$. An entry $(i, \tilde{f}_i, p_i)$ will be deleted if and only if

\[ \tilde{f}_i \leq \sum_{j=p_i+1}^{k} t_j. \]

(2.1)

The rationale of this pruning strategy is as follows. Suppose an item $i$ in memory was pruned at the first pruning point $p_1$ because its count is not greater than the pruning threshold $t_1$. Later, $i$ may arrive again before the next pruning point $p_2$ and have accumulated some counts. Let us assume that the pruning threshold at $p_2$ is $t_2$ and $i$ is again pruned because its new count is not greater than $t_2$. Then the maximum possible error introduced due to pruning for item $i$ at pruning point $p_2$ is $t_1 + t_2$. Now suppose another item $j$ has arrived before pruning point $p_1$ but it was not pruned at $p_1$. Suppose $j$'s count at pruning point $p_2$ is not greater than $t_1 + t_2$. Then if we prune $j$ at $p_2$ with threshold $t_1 + t_2$, the error introduced is $t_1 + t_2$. This error is the same as the error introduced by consecutive insertion and pruning of $i$ at pruning points $p_1$ and $p_2$. Therefore, pruning $j$ does not incur any additional error, but provides more free space for new items.

In general, we can see that an item having an entry created at pruning point $p_i$ can be pruned at point $p_j$, where $i < j$, if its count is not greater than the sum of the pruning thresholds from $p_i$ to $p_j$ without incurring any additional error. We term this kind of pruning strategy as progressive pruning due to the following feature: Items arrived early may be progressively pruned during the later pruning points. This pruning strategy avoids bias toward older items. If each item in memory gets pruned using the same threshold, then items that have arrived earlier would usually have higher counts and thus are less likely to be pruned than new items. By having pruning threshold associated with both the creation time and the count of the item, it avoids this bias.

Figure 2 shows the result of using progressive pruning strategy on the same data stream as in Figure 1. The rows in each table describes the following information: item, count, creation time, pruning id, and pruning threshold for this item. Let us take a look at the second table. Though the total count of items $f$ and $c$ is smaller than that of $a$ and $b$, but all four items are pruned at pruning point $P_3$. This is because the entries for $f$ and $c$ were created after pruning point $P_1$ while the entries for $a$ and $b$ were created since pruning point $P_0$. Therefore, the progressive pruning thresholds for them are 2 and 3 respectively. We can see that for the length of thirty items and storing only 4 entries, progressive pruning only prunes at three points ($P_1 - P_3$), and the maximum loss of counts is only four ($1+2+1$), which is much lower than the straightforward strategy.

The progressive pruning strategy is outlined in Algorithm 2.

In Algorithm 2, the pruning threshold of the current pruning point is set to the minimum frequency count of all items in $M$ (line 1). We extended $P$, the array that holds pruning point information, to include the cumulative pruning threshold sum_th. This threshold stores the maximum sum of counts that could have been pruned from the beginning to this particular pruning point (line 2). Since each item in memory has its associated pruning id, its pruning threshold can
be easily calculated (line 3-4). If the maintained count of an entry is no larger than the threshold, the entry is deleted (line 5-7).

With our progressive pruning strategy, querying the actual count of an item is an easy task. Algorithm 3 outlines the algorithm for querying the count of an item together with its associated error bound. For any item i that has an entry M[i,cnt, p.id] in memory, cnt records the actual count of i since pruning point p.id. The maximum number of counts that could have been lost due to pruning before point p.id is the sum of the pruning threshold at the pruning points prior to p.id, which is P[p.id].sum.th. For items that are not in memory, its error bound would be the sum of the pruning threshold at all pruning points.

For the sake of clarity of our basic approach, we did not provide detailed data structures for implementing PLP. Astute readers may wonder whether PLP uses more extra memory than other algorithms where each entry only needs two fields, item and count, while PLP seems need three fields, and an extra array for pruning information. We believe that the storage of a small amount of extra information does not impede PLP’s performance comparing to other algorithms. Instead, performance results show that because of this small amount of information, PLP is able to have superior performance. First, comparing to existing algorithms that also stores some extra information, PLP does not require much more overhead than them. For example, the most representative algorithm, Lossy Counting [14], also requires memory overhead to store information for pruning. Since PLP prunes only when the memory is full and the use of progressive pruning strategy free up more space at each pruning point, pruning occurs very infrequently. The experiments conducted show that for a data stream of length of more than 1,000,000, only a few prunings needs to be conducted with PLP. In addition, we only need to store the information of the pruning points that are associated with memory entries. Therefore, not much space is required. Second, recall that most existing algorithms use eager-pruning approach such that pruning is conducted even when there is memory space available. This means that there are usually memory space unused by those algorithms. This unused memory can be counted towards the space that is
Algorithm 2 Progressive Pruning

Input:
- \( p.id \): current pruning id;
- \( M[item, cnt, p.id] \) where item: item, cnt: count of \( i \), and \( p.id \): pruning id of \( i \);
- \( P[n, threshold, sum.th] \) where \( n \): number of arrivals up to the pruning point, \( threshold \): the maximum count of the items pruned at the pruning point, \( sum.th \) is the cumulative threshold up to the pruning point.

Output:
- \( M \) minus pruned entries.

Description:
1. \( P[p.id].threshold := \text{GetMinCnt}(M) \);
2. \( P[p.id].sum.th := P[p.id-1].sum.th + P[p.id].threshold \)
3. for each entry \( M[i] \) do
4.   \( threshold := P[p.id].sum.th - P[M[i], p.id].sum.th \);
5.   if \( M[i].cnt \leq threshold \) then
6.     delete \( M[i] \);
7.   end if
8. end for

not used to store items and their counts. We can look at it in this way, PLP merely makes use of this unused memory space to store some useful information for a better pruning strategy which results in better performance. Lastly, there are many optimizations can be done during actual implementation of the algorithm, such as storing the same \( p.id \) once for all the items that arrive between the same two pruning points. if we trade a bit of processing time for memory, we can further reduce the memory usage by clustering items with the same counts together so that only one count is needed for items that have the same count.

2.3 Properties of PLP Using the progressive pruning strategy, we have the following lemma for approximate counts maintained by the Progressive Lazy Pruning algorithm.

Lemma 1. Let \( M \) be the approximate counts maintained using Progressive Lazy Pruning. Let \( k \) be the maximum pruning id, and \( t_j \) the pruning threshold at pruning point \( j \), \( 1 \leq j \leq k \). If item \( i \) does not have an entry in \( M \), then the actual frequency count of \( i \) is

\[
(2.2) \quad f_i \leq \sum_{j=1}^{k} t_j.
\]

Proof: If item \( i \) does not have an entry in \( M \), then one of the following cases applies.

Case 1. The item has never arrived. In this case, \( f_i = 0 \leq \sum_{j=1}^{k} t_j \).
Algorithm 3 Querying Count Information of an Item

Input:
item: the item whose count is being queried;

Output:
(count, error): count: the approximate frequency count of item;
error: the error bound.

Description:
1: if an entry $M[x]$ is found such that $M[x].item = item$ then
2: count := $M[x].cnt$;
3: error := $P[M[x].p_id].sum_th$;
4: else
5: count := 0;
6: error := $P[p.id].sum_th$;
7: end if
8: return (count, error)

Case 2. The item was once inserted with pruning id $p_i$ and pruned off at pruning point $p'_i$, $p_i < p'_i$. In this case, according to Definition 1, $f_i = \sum_{j=p_i+1}^{p'_i} t_j = \sum_{j=1}^{k} t_j$.

Case 3. The item was inserted multiple times with pruning id $p_{i1}, p_{i2}, \ldots, p_{im}$ and pruned off at $p'_{i1}, p'_{i2}, \ldots, p'_{im}$, respectively, where $p_{i1} < p'_{i1} < p_{i2} < p'_{i2} < \cdots < p_{im} < p'_{im}$. In this case,

$$f_i = \sum_{l=1}^{m} \sum_{j=p_{il}+1}^{p'_{il}} t_j = \sum_{j=1}^{k} t_j.$$

Therefore, Lemma 1 holds.

With Lemma 1, we can prove the following theorem.

**Theorem 1.** Let $M$ be counts maintained using Progressive Lazy Pruning with progressive pruning, $k$ the total number of pruning points, and $t_j$ the pruning threshold at the $j$th pruning point where $1 \leq j \leq k$. For any item $i$, the estimated frequency count is (a) $\tilde{f}_i$, if there exists an entry $(i, \tilde{f}_i, p)$ in $M$; (b) 0, otherwise. Then we have

$$f_i - \tilde{f}_i \leq \sum_{j=1}^{k} t_j. \quad (2.3)$$

**Proof:** The proof for case (b) is a direct result of Lemma 1. For an item without an entry in $M$, $f_i \leq \sum_{j=1}^{k} t_j$. Since we set $\tilde{f}_i = 0$, Equation 2.3 holds. For case (a), since $i$ has an entry with pruning id $p_i$, $\tilde{f}_i$ must be its actual counts after pruning point $p_i$. Let the actual count of $i$ before
$p_i$ be $f_i^-$. We have the total actual count
\[ f_i = f_i^- + f_i^+. \]

Furthermore, since the entry is created with pruning id $p_i$, there must be no entry for $i$ right at pruning point of $p_i$. The reason is that every pruning point is triggered by a newly arrived item such that there is no entry for it in memory and there is also no free space to accommodate a new entry for it. According to Lemma 1,
\[ f_i^- \leq \sum_{j=1}^{p_i} t_j, \]

Therefore,
\[ f_i = f_i^- + f_i^+ \leq \sum_{j=1}^{p_i} t_j + f_i^+, \]
\[ f_i - f_i^+ \leq \sum_{j=1}^{p_i} t_j \leq \sum_{j=1}^{k} t_j. \]

With the absolute error given in Theorem 1, we can see that the absolute error of Progressive Lazy Pruning maintained estimate counts are bounded by $\sum_{j=1}^{k} t_j$.

**Corollary 1.** Let $N$ be the total number of arrivals when the maximum number of pruning points is $k$. Then the frequency estimation error of the approximate counts maintained by Progressive Lazy Pruning is
\[ \epsilon \leq \frac{\sum_{j=1}^{k} t_j}{N}. \]  

Corollary 1 is a direct result of the definition of relative error and Theorem 1.

Note that the estimation error, as expressed in Equation 2.4 has the following property.

**Property 1.** Assume data items in the input stream are independent and follow identical distribution. The estimation error of approximate frequency counts maintained by Progressive Lazy Pruning is independent of the total number of arrivals $N$.

**Proof:** According to Corollary 1, the relative estimation error is bounded by
\[ \frac{\sum_{j=1}^{k} t_j}{N}, \]
where $k$ is the total number of pruning points with $N$ arrivals, and $t_j$ is the pruning threshold at pruning point $j$. According to Definition 1, $t_j$ is the minimum frequency count among all items in $M$ at pruning point $j$. Assume that $i_j$ is the item with the minimum count $t_j$, and the entry for $i_j$ has pruning id $j'$, $j' < j$. Let the number of arrivals at pruning point $j'$ and $j$ be $n_{j'}$ and
$n_j$, respectively. We further assume that the probability of $i_j$ appearing in $I$, the set of items from where the items of the data stream is drawn, and hence in the data stream, is $Pr(i_j)$. With sufficiently large $n_j - n_{j'}$, according to the Bernoulli Law of large numbers, we can use $Pr(i_j)$ to approximate $\frac{t_j}{n_j - n_{j'}}$. That is,

$$Pr(i_j) \approx \frac{t_j}{n_j - n_{j'}},$$

or

$$t_j \approx Pr(i_j) \cdot (n_j - n_{j'}) .$$

Therefore,

$$\sum_{j=1}^{k} t_j \approx \sum_{j=1}^{k} Pr(i_j) \cdot (n_j - n_{j'})$$

$$\leq \sum_{j=1}^{k} Pr(i_j) N = N \sum_{j=1}^{k} Pr(i_j) ,$$

Combining Equation 2.4 and 2.5, we have

$$\epsilon \leq \sum_{j=1}^{k} Pr(i_j) ,$$

which is independent of $N$, the number of total arrivals. □

Property 1 states a very good feature of Progressive Lazy Pruning: With given available memory $m$, we can always maintain approximate counts over data streams with a bounded error for any number of arrivals.

### 2.4 Maintaining $\epsilon$-Approximate Count with Progressive Lazy Pruning

Recall that $\epsilon$-approximate count for a set of $N$ elements is defined as a set of (element, count) pairs such that (a) the count associated with an element is smaller than the element’s true frequency count, but by at most $\epsilon N$, and (b) any element whose true frequency exceeds $\epsilon N$ belongs to the set; other elements may or may not. We show that Algorithm Progressive Lazy Pruning with progressive pruning maintains an $\epsilon$-approximate count.

**Theorem 2.** The counts maintained for a data stream with $N$ arrivals by Progressive Lazy Pruning is $\epsilon$-approximate with $\epsilon = \frac{\sum_{j=1}^{k} t_j}{N}$.

**Proof:** Theorem 2 is a direct result of Theorem 1 and Lemma 1. Let $\epsilon = \frac{\sum_{j=1}^{k} t_j}{N}$. According to Theorem 1, we have, for item $i$ with a count $\tilde{f}_i$ in $M$,

$$f_i - \tilde{f}_i \leq \sum_{j=1}^{k} t_j = \epsilon N .$$
According to Lemma 1, for item \( i \) without entry in \( M \),

\[
f_i \leq \sum_{j=1}^{k} t_j = cN.
\]

Thus, if \( f_i > cN, i \) must be in \( M \). \( \square \)

Although Progressive Lazy Pruning maintains \( c \)-approximate counts, the error \( c \) will only be known after a sufficiently large number of data items have arrived. For certain applications with a given error requirement, \( c \), we can use the following \( c \)-pruning strategy, combining with progressive pruning to maintain \( c \)-approximate counts using the lazy pruning approach.

**Definition 2.** \((c\text{-Pruning})\) Let \( c \) be the required error bound, and \( n_k \) the number of total arrivals up to pruning point \( k \) with \( n_0 = 0 \). The \( c \)-pruning strategy deletes entries \((i, \bar{f}_i, p_k)\) at pruning point \( k \) if

\[
\bar{f}_i \leq \sum_{j=k+1}^{k} \epsilon(n_k - n_{k-1}).
\]

From the above definition, we can see that \( c \)-pruning can essentially be viewed as a special case of progressive pruning with pruning threshold at pruning point \( k \) being \((n_k - n_{k-1})\). It is easy to prove that \( c \)-pruning produces \( c \)-approximate counts.

**Theorem 3.** Let \( M \) be counts maintained using Progressive Lazy Pruning with \( c \)-pruning. Then \( M \) is \( c \)-approximate. That is, (a) the count associated with an item \( i \), \( \bar{f}_i \), is smaller than its actual frequency count \( f_i \), and \( f_i - \bar{f}_i \leq cN \), and (b) any item whose true frequency exceeds \( cN \) belongs to the set; other items may or may not.

**Proof:** As we have proved in Theorem 1, the error of Progressive Lazy Pruning maintained counts \( \bar{f}_i \) for \( i \) is \( f_i - \bar{f}_i \leq \sum_{j=1}^{k} t_j \). With \( c \)-pruning, \( t_j = \epsilon(n_j - n_{j-1}) \). Therefore,

\[
f_i - \bar{f}_i \leq \sum_{j=1}^{k} t_j = \sum_{j=1}^{k} \epsilon(n_j - n_{j-1}) = \epsilon(n_k - n_0).
\]

Since \( n_0 = 0 \), and \( n_k \leq N \), we have

\[
f_i - \bar{f}_i \leq \epsilon n_k \leq cN.
\]

Furthermore, according to Lemma 1, the actual frequency of an item which is not in \( M \) should be

\[
f_i \leq \sum_{j=1}^{k} t_j \leq cN.
\]

Thus, any item whose true frequency exceeds \( cN \) is in \( M \). Therefore, both (a) and (b) holds for counts maintained by Lazy Pruning with \( c \)-pruning. That is, the maintained counts are \( c \)-approximate. \( \square \)
Algorithm 4 ε-Pruning

Input:

\( \epsilon \): estimation error;
\( p.\text{id} \): current pruning id;
\( M[\text{item}, \text{cnt}, p.\text{id}] \) where \text{item}, \text{cnt}: count of \text{i}, and \( p.\text{id} \): pruning id of \text{i};
\( P[n, \text{threshold, sum.th}] \) where \( n \): number of arrivals up to the pruning point, \text{threshold}: the maximum count of the items pruned at the pruning point, and \( \text{sum.th} \) is the accumulative threshold up to the pruning point.

Output:

\( M \) with some entries being pruned.

Description:

1. \( P[p.\text{id}], \text{threshold} := \epsilon \cdot (P[p.\text{id}], n - P[p.\text{id}], n) \);
2. \( \text{minCnt} := \text{GetMinCnt}(M) \);
3. if \( P[p.\text{id}], \text{threshold} < 1 \) then
4. \( P[p.\text{id}], \text{threshold} := \text{minCnt} \);
5. end if
6. \( P[p.\text{id}], \text{sum.th} := P[p.\text{id}], \text{sum.th} + P[p.\text{id}], \text{threshold} \)
7. for each entry \( M[i] \) do
8. \( \text{threshold} := P[p.\text{id}], \text{sum.th} - P[M[i], p.\text{id}], \text{sum.th} \);
9. if \( M[i], \text{cnt} \leq \text{threshold} \) then
10. delete \( M[i] \);
11. end if
12. end for

One issue in implementing ε-pruning is that the given memory may not be sufficient to maintain ε-approximate counts for a given ε with any number of arrivals. Recall that the pruning threshold is \( \epsilon(n_k - n_{k-1}) \), which may not be large enough to release memory for new arrivals. For example, with a memory size of \( m \) entries, \( n_1 \) arrivals with \( m \) distinct items will fill up the available memory, which leads to a pruning point of our lazy pruning algorithm. The pruning threshold, \( t_1 \) will be \( \epsilon \cdot n_1 \) that could be less than one with small \( \epsilon \) and small \( n_1 \), which would lead to no pruning of entries. On the other hand, with the same amount of memory, pruning is possible at a later pruning point \( k \) since \( n_k - n_{k-1} \) might be much larger than \( n_1 \). To make the algorithm robust, Lazy Pruning with ε-pruning sets the pruning threshold as the progressive pruning (Definition 1) when ε-pruning is infeasible.

The algorithm of ε-pruning is outlined in Algorithm 4. It calculates the pruning threshold for the current pruning point based on the given ε and the number of new arrivals after the last pruning point (line 1). If this threshold is less than one, the pruning threshold is set to the minimum count to guarantee that some entries will be pruned off to release memory for new arrivals (line 3-5).
The minimal count in $M$ is obtained by calling Function $GetMinCnt$ (line 2). The remaining part of the algorithms is the same as Algorithm 2. That is, the pruning threshold for each item is calculated and an entry is deleted if the count associated is less than or equal to the threshold (line 6-11). One improvement can be incorporated is that whenever the pruning threshold based on $c$ is bigger than the pruning threshold of PLP, then use the threshold of PLP instead. This will help in reducing the estimation error of the items.

3 A Performance Study

To study the performance of Progressive Lazy Pruning, we implemented the algorithm in the C language and ran the programs on a UNIX machine. Item $i$ in a data stream is randomly drawn from a set of items that follows a Zipf-like distribution [13]:

$$f_i = \frac{1}{s^s \sum_{j=1}^{I} \frac{1}{j^s}},$$

where $f_i$ is the frequency of $i$, $I$ is the number of distinct items in the set, and $s$ is the skew factor. When $s = 0$, the distribution becomes uniform. With $s = 1$, it corresponds to the highly skewed pure Zipf distribution [16]. By varying $s$, different kinds of data distributions can be produced. We believe that the Zipf-like distribution is better at simulating real world applications because the items in real world applications are usually drawn from a pool of finite samples. Although it was mentioned to use non-repeatable infinite data samples, we believe that it is not necessary to conduct such tests as the performance of Progressive Lazy Pruning is predictable for streams with infinite number of samples without repeats: it will basically delete all items at every pruning point since the count of all such items is one. As such, the number of pruning points with $N$ arrivals will be $\lfloor \frac{N}{m} \rfloor$ where $m$ is available memory in terms of the number of entries. Therefore, the maximum estimation error will be about $\frac{1}{m}$.

The parameters used in the experiments are listed in Table 2.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>total number of items</td>
<td>500K-2M</td>
</tr>
<tr>
<td>$I$</td>
<td>number of distinct items</td>
<td>20K-100K</td>
</tr>
<tr>
<td>$s$</td>
<td>skew factor</td>
<td>0-1.0</td>
</tr>
<tr>
<td>$m$</td>
<td>available memory</td>
<td>4-20% of $I$</td>
</tr>
<tr>
<td>$c$</td>
<td>estimation error</td>
<td></td>
</tr>
</tbody>
</table>

3.1 The Performance of PLP The first set of experiments studies the basic performance of Progressive Lazy Pruning and the effects of the parameters, including the number of arrivals, the
number of distinct items, the available memory size, and the distribution of the items. The results are shown in Figure 3 in terms of the maximum estimation error $\epsilon$. The maximum estimation error is obtained from the number of pruning points and the pruning threshold at each pruning point, as specified in Theorem 1:

$$
\epsilon = \frac{\sum_{j=1}^{k} t_j}{N},
$$

where $N$ is the total number of arrivals, $k$ is the number of pruning points for $N$ arrivals, and $t_j$ is the pruning threshold of $j$th pruning point, $1 \leq j \leq k$. Note that $\epsilon$ defined in Equation 3.7 is the estimation bound guaranteed by the algorithm. The actual error for the estimation of an item should be less than or equal to the maximum error.

In Figure 3(a) we set the number of distinct items $I = 50,000$ and the size of memory $m = 10\% I = 5,000$. The number of arrivals varies from 100,000 to 2,000,000. The results verify the property of Progressive Lazy Pruning stated in Property 1. That is, the estimation error $\epsilon$ is independent of the total number of arrivals $N$ with sufficiently large $N$. In other words, the estimation error is only determined by the characteristics of data and the size of available memory. Since the error is independent from the number of arrivals, for the other experiments, we simply pick some values for the number of arrivals without justification.

Figure 3(b) and 3(c) depicts the maximum estimation errors when varying the number of distinct items and the size of memory used to keep the counts, respectively. In both cases, we fixed the number of arrivals $N = 500,000$. In Figure 3(b) the number of distinct items varies from 4\% to 20\% of $N$, i.e., from 20,000 to 100,000. It can be seen that with a large number of distinct items, the estimation error drops quickly.

In Figure 3(c), the memory size vary from 4\% to 20\% of the number of distinct items. From the results, we can make two observations. First, as expected, with the large size of available memory, the estimation error drops quickly. In fact, if the available memory size is equal to or greater than the number of distinct items, the estimation error could drop to zero, as we can maintain an exact count for each item. Second, even with memory being a small percentage of the number of distinct items, the algorithm can maintain quite an accurate estimation. For example, with memory that can only accommodate 4\% of the distinct items, the maximum estimation error for uniformly distributed data is about 0.0005. If the memory size increases to 10\%, the error can be reduced to 0.0002.

In summary, we can see that, for data streams with the same characteristics (i.e., the same number of distinct items and skew factor), larger memory leads to lower error, which is obvious. One thing to note is that if the memory size is same as the number of distinct items, then the error is zero. If we fix the memory as a fixed percentage of the number of distinct items, then larger number of distinct items leads to lower estimation error. Lastly, data with higher skew factor has a lower estimation error.
3.2 The Performance of Epsilon-PLP Since PLP-Epsilon adopts the same approach as PLP, its performance is expected to be similar to PLP's performance. The difference is Epsilon-PLP maintains the approximate counts with a specified estimation error whenever it is possible. On the other hand, PLP maintains the approximate counts by pruning off minimal number of items at each pruning point, and the accuracy of counts is usually high. Experimental results in Figure 4(a) verify this. The two algorithms were run with the same parameter settings, that is, number of distinct items \( I = 50,000 \), memory \( m = 5,000 \). By setting \( \epsilon = 0.0005 \), the estimation errors of Epsilon-PLP maintained counts are around 0.0005 (0.00047-0.0005) with different skew factors. On the other hand, the estimation errors for PLP in general are much lower and they also vary with different amounts of data skew.

Epsilon-PLP prunes off entries more aggressively at each pruning point. As a result, it requires less pruning points. Figure 4(b) depicts the number of pruning points for the two algorithms. This figure shows that Epsilon-PLP required 5-15\% less pruning points than PLP for the same parameter settings.

![Graphs](image)

(a) Varying Number of Arrivals  (b) Varying Number of Distinct Items  (c) Varying Memory Size

Figure 3: The Performance of PLP with Varying Parameters
Figure 4: Algorithm PLP-Epsilon
Figure 5: A Comparative Study: Lossy Counting vs. Progressive Lazy Pruning
3.3 A Comparison with Algorithm Lossy Counting

Lossy Counting (LC) is a representative deterministic one-pass algorithm that maintains approximate counts over data streams. A set of experiments were conducted to compare PLP with LC. We first ran the LC algorithm with given data sets with specified \( \epsilon \), and recorded the maximum number of memory entries used. Then we used the results to set the parameter of the available memory for algorithm PLP. One set of results is presented in Figure 5.

Figure 5(a) lists the memory required for Lossy Counting with \( \epsilon = 0.0005 \), the number of distinct items \( I = 500,000 \), and varying the total number of arrivals \( N \) from 100,000 to 2,000,000. The skew factor was varied from 0.0 to 1.0. As expected, the memory required by LC increases with \( N \) as shown in Figure 5(b). Note that, when the data is uniformly distributed (i.e., \( s = 0 \)), the required memory to maintain the estimation accuracy increases from 3.724 to 27.994 when the number of arrivals \( N \) increase from 100,000 to 2,000,000.

Figure 5(b) compares the memory requirement for LC and PLP. The specified error requirement \( \epsilon = 0.0005 \). The curves shown are the ratio between the memory requirement of Lossy Counting and Progressive Lazy Pruning. We can see that when the number of arrivals \( N = 2,000,000 \), Lossy Counting needs about 14 times more memory than Progressive Lazy Pruning when the skew factor is zero. Even with highly skewed data, the former requires about as four times as much memory.

To further illustrate the performance difference between Lossy Counting and Progressive Lazy Pruning, we conducted another set of experiments. We used the memory required by the Lossy Counting algorithm as listed in Figure 5(a) as the available memory for respective cases in Progressive Lazy Pruning algorithm, and measured the estimation error. Curves in Figure 5(c) depict the ratio between the estimation error of the two algorithms. When the number of arrivals is 2,000,000, using the same size of memory, PLP can maintain approximate counts that are nine to fourteen times more accurate than LC can.

3.4 Running Time of the Algorithms

The main operations of the Progressive Lazy Pruning algorithms include updating counts or inserting new entries upon arrival, conducting pruning at pruning points, and retrieving counts when queried.

New arrivals: To facilitate the search for the entry for a new arrival, hashing is used. Although the size of the hash table is relatively small compared to the number of entries, hence search within a chain of the entries with the same hash value has to be sequential, the total time will still be \( O(1) \) to find the entry, or to conclude that a corresponding entry does not exist. Both insertion of new entry and update of counts are trivial operations.

Answering queries: There are two types of queries, reporting the count of specific items, or reporting all items with certain specified frequency counts. In the former case, it is similar to handling new arrivals with complexity of \( O(1) \). To report all items with a predicate on frequency counts, searching of all entries is required, which results in a complexity of \( O(m) \) where \( m \) is the
total number of entries.

Pruning entries: At pruning points, entries with count less than the pruning threshold are pruned off. So, searching all entries is required if no other data structure is implemented, which leads to a complexity of $O(m)$. For applications with high arrival rate, the required pruning time may be longer than the arrival intervals. This can be easily solved as pruning could be implemented in a non-blocking fashion. That is, new arrivals can be inserted as long as some entries are released without the completion of the pruning process.

4 Related Work

The earliest deterministic algorithm on approximate frequency counts over data streams is a two-pass algorithm by Misra and Gries [15]. It requires $\frac{1}{\varepsilon}$ constant space and $O(1)$ amortized processing time per item. Later, [7, 12] made this algorithm into a one-pass algorithm that retains the $\frac{1}{\varepsilon}$ constant space bound and uses $O(1)$ processing time. But they all have no guarantee over the frequency counts of the items. Recently, many one-pass algorithms for processing frequent items in data streams have been proposed [2, 3, 5, 7, 10, 12, 14].

Among the existing work, the algorithm developed by Karp et al. [12] for iceberg queries is closest to our algorithm in terms pruning strategy. That is, they also use lazy pruning approach. However, their algorithm is aimed at keeping track of high frequency items instead of keeping track of accurate counts for items. At every pruning point, the algorithm decrements the count of every entry in memory by the minimum count among the entries. Those entries having count equal to zero after the decrement are pruned. This means that the maintained counts of entries in memory are usually much smaller than their actual counts, depending on the number of pruning points an entry has survived. Therefore, Karp's algorithm does not have any guarantee over the maintained frequency counts of items, regardless of highly frequent or infrequent items. PLP, on the other hand, is aimed at keeping track of count information of items as accurately as possible. It guarantees a bounded error that is independent of the number of arrivals and provides highly accurate count information for each individual item. At any point in time, if an item has an entry in memory, then the count stored in memory is this entry's true count since its creation time. The pruning information stored is used to provide an individual error bound for each item.

Manku and Motwani's one-pass deterministic algorithm [14], Lossy Counting, has received much attention recently. Lossy Counting is a representative eager approach algorithm. It requires users to specify two parameters: support $s$, and error $\varepsilon$. This algorithm uses $\frac{1}{\varepsilon}log(cN)$ space. They claim that even though Lossy Counting has worse space complexity than Misra's algorithm, it is more superior for skewed data. They also proposed a probabilistic sample-based algorithm called Sticky Sampling [14]. It requires $\frac{3}{\varepsilon}log(s^{-1}\delta^{-1})$ space and is with the probability of $(1 - \delta)$ within the required error bound, where $\varepsilon$, $s$, and $\delta$ are user-specified parameters for error, support and probability of failure, respectively. Their study shows that, in practice, Lossy Counting is better
than Sticky Sampling. We showed in our experimental section that, PLP has superior performance over Lossy Counting for various data distributions, including highly skewed data distribution.

There is some other related work for problems in the data stream environment, such as the top-k problem [3, 10], frequency counting over sliding windows [2], frequency counting in data streams with both insertion and deletion of items [5, 11]. Gibbons and Matias [10] presented the first sample-based algorithm for solving the top-k problem in data streams. Charikar et al. [3] uses a Count Sketch to find $k$ items whose frequency is at least $(1 - c)$ times the frequency of the $k$th most frequent items with probability $(1 - \delta)$. Arasu and Motwani [2] have recently proposed two algorithms for maintaining approximate counts and quantiles over sliding windows. Cormode et al. [5] developed a randomized algorithm for maintaining counts in data streams that consists of both insertion and deletion of items. Jin et al. [11] later proposed an algorithm called hCount that uses less memory than Cormode’s algorithm.

5 Conclusion

Frequency counting of single items over data streams is an important problem since it provides foundation for other more complex problems in the data stream environment. Its biggest challenge is to effectively use a limited amount of memory to maintain frequency counts for items from an infinite data stream as accurately as possible with bounded error. We introduced a new algorithm, Progressive Lazy Pruning (PLP), that improves over existing algorithms for frequency counting over data streams in three major ways. First, most existing algorithms are aimed at maintain counts of items such that a particular error bound is met. They do not try to provide users highly accurate count information for individual items. In practice, this is inadequate for some applications in which individual count information is very important. PLP is not only able to maintain counts with bounded error that is independent of number of arrivals, it also can provide as accurate as possible frequency counts for individual items. Second, most existing algorithms ask users for an error parameter and provide them a theoretical memory bound that is often not an accurate gauge of the true memory requirement (since the actual memory required often vary drastically depending on the data characteristics and arrival orders). While users may have some idea about the error requirement for certain applications, it is difficult for them to know how much memory is actually required to guarantee the error. PLP, on the other hand, does not require users to provide any error parameter. It maintains count information as accurately as possible with the available memory space which is always known. Furthermore, the error bound is independent from the number of arrivals. Users can immediately know whether the current amount of memory is adequate after a sufficient number of items arrived, and can increase memory if necessary. Third, most existing algorithms prune items from memory even when there is free space which leads to unnecessary accuracy loss. PLP adopts a lazy pruning approach that prunes only when there is no memory available. In terms of performance, when compared to Lossy Counting (LC), a representative one-pass deterministic algorithm, PLP achieves higher accuracy in maintained counts than LC when
using the same amount of memory for various data distributions, including highly skewed data.

In this paper we have concentrated on the basic pruning strategies to maintain ε-approximate counts. Similar to previous work, we use the number of entries in memory as a measure of the memory requirement in our performance study. There are additional techniques that can effectively maintain more count entries. For example, items with the same count can be grouped together to accommodate more items in memory. Since such techniques can be used in any method they are not discussed in detail.

The algorithms presented here are not very complicated. One immediate work is to further study the property of progressive pruning to explore other pruning strategies. We also plan to study the methods of maintaining statistics over data streams with memory constraints so that pruning can be more effective and the error bound can be further reduced.

References
