Approximate Frequency Counts over Data Streams

Gurmeet Singh Manku *
Stanford University
manku@cs.stanford.edu

Rajeev Motwani †
Stanford University
rajeev@cs.stanford.edu

Abstract

We present algorithms for computing frequency counts exceeding a user-specified threshold over data streams. Our algorithms are simple and have provably small memory footprints. Although the output is approximate, the error is guaranteed not to exceed a user-specified parameter. Our algorithms can easily be deployed for streams of singleton items like those found in IP network monitoring. We can also handle streams of variable sized sets of items exemplified by a sequence of market basket transactions at a retail store. For such streams, we describe an optimized implementation to compute frequent itemsets in a single pass.

1 Introduction

In several emerging applications, data takes the form of continuous *data streams*, as opposed to finite stored datasets. Examples include stock tickers, network traffic measurements, web-server logs, click streams, data feeds from sensor networks, and telecom call records. Stream processing differs from computation over traditional stored datasets in two important aspects: (a) the sheer volume of a stream over its lifetime could be huge, and (b) queries require timely answers; response times should be small. As a consequence, it is not possible to store the stream in its entirety on secondary storage and scan it when a query arrives. This motivates the design for *summary data structures* with small memory footprints that can support both one-time and continuous queries.

*Supported by NSF Grant IIS-0118173 and Stanford Graduate Fellowship

† Supported by NSF Grant IIS-0118173, an Okawa Foundation Research Grant, and grants from Microsoft and Veritas.

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

An aggregate query that is widely applicable is `SELECT R.EventType, COUNT(*) FROM R GROUP BY R.EventType`, where relation `R` models a stream of events with attribute `R.EventType` denoting the type of event. In several applications, groups with very low frequencies are uninteresting; the user is interested in only those groups whose frequency exceeds a certain threshold, usually expressed as a percentage of the size of the relation seen so far. The modified query then is `SELECT R.EventType, COUNT(*) FROM R GROUP BY R.EventType HAVING COUNT(*) > s |R|` where `s` is a user-specified percentage and `|R|` is the length of stream seen so far.

A related aggregate query is over an input stream relation whose tuples are *sets* of items rather than individual items. The goal is to compute all subsets of items, hereafter called itemsets, which occur in at least a fraction s of the stream seen so far. This is far more challenging than counting singleton tuples.

1.1 Motivating Examples

Here are four problems drawn from databases, data mining, and computer networks, where frequency counts exceeding a user-specified threshold are computed.

1. An iceberg query [FSGM$^+$98] performs an aggregate function over an attribute (or a set of attributes) of a relation and eliminates those aggregate values that are below some user-specified threshold.
2. Association rules [AS94] over a dataset consisting of sets of items, require computation of *frequent itemsets*, where an itemset is frequent if it occurs in at least a user-specified fraction of the dataset.
3. Iceberg datacubes [BR99, HPDW01] compute only those `Group By`'s of a `CUBE` operator whose aggregate frequency exceeds a user-specified threshold.
4. Traffic measurement and accounting of IP packets requires identification of flows that exceed a certain fraction of total traffic [EV01].

Existing algorithms for iceberg queries, association rules, and iceberg cubes have been optimized for finite stored data. They compute exact results, attempting to minimize the number of passes they make over the entire dataset. The best algorithms take two passes. When adapted to streams, where only one pass is allowed, and results are always expected to be available with short response times, these algorithms fail to provide any a priori guarantees on the quality of their output. In this paper, we present algorithms for computing frequency counts in a single pass with *a priori* error guarantees. Our algorithms work for variable sized transactions. We can also compute frequent *sets of items* in a single pass.

1.2 Offline Streams

In a data warehousing environment, bulk updates occur at regular intervals of time, e.g., daily, weekly or monthly. It is desirable that data structures storing aggregates like frequent itemsets and iceberg cubes, be maintained incrementally. A complete rescan of the entire database per update is prohibitively costly for multi-terabyte warehouses.

We call the sequence of updates to warehouses or backup devices an *offline stream* to distinguish it from online streams like stock tickers, network measurements and sensor data. Offline streams are characterized by regular bulk arrivals. Queries over these streams are allowed to be processed offline. However, there is not enough time to scan the entire database whenever an update comes along. These considerations motivate the design of summary data structures [AGP99] that allow incremental maintenance. These summaries should be significantly smaller than the base datasets but need not necessarily fit in main memory.

Query processing over offline and online streams is similar in that summary data structures need to be incrementally maintained because it is not possible to rescan the entire input. The difference lies in the time scale and the size of updates.

We present the first truly incremental algorithm for maintaining frequent itemsets in a data warehouse setting. Our algorithm maintains a small summary structure and does not require repeated rescans of the entire database.

1.3 Roadmap

In Section 2, we review applications that compute frequent itemsets, identifying problems with existing approaches. In Section 3, we formally state the problem we solve. In Section 4, we describe our algorithm for singleton items. In Section 5, we describe how the basic algorithm can be extended to handle sets of items. We also describe our implementation in detail. In Section 6, we report our experiments. Related work is presented in Section 7.

2 Frequency Counting Applications

In this section, we review existing algorithms for four different applications, identifying shortcomings that surface when we adapt them to stream scenarios.

2.1 Iceberg Queries

The idea behind Iceberg Queries[FSGM+98] is to identify aggregates in a GROUP BY of a SQL query that exceed a user-specified threshold τ. A prototypical query on a relation `R(c1, c2, ..., ck, rest)` with threshold τ is

```
SELECT   c1, c2, ..., ck, COUNT(rest)
FROM     R
GROUP BY c1, c2, ..., ck
HAVING   COUNT(rest) ≥ τ
```

The parameter τ is equivalent to `s |R|` where `s` is a percentage and `|R|` is the size of `R`. The algorithm presented in [FSGM+98] uses repeated hashing over multiple passes. It builds upon the work of Park et al [PCY95]. The basic idea is the following: In the first pass, a set of counters is maintained. Each incoming item is hashed onto a counter which is incremented. These counters are then compressed into a bitmap, with a 1 denoting a large counter value. In the second pass, exact frequencies for only those elements are maintained which hash to a value whose corresponding bitmap value is 1. Variations and improvements of this basic idea are explored in [FSGM+98]

It is difficult to adapt this algorithm for streams because at the end of the first pass, no frequencies are available.

2.2 Iceberg Cubes

Iceberg cubes [BR99, HPDW01] take Iceberg queries a step further by proposing a variant of the CUBE operator over a relation `R(c1, c2, ..., ck, X)`:

```
SELECT  c1, c2, ..., ck, COUNT(*), SUM(X)
FROM    R
CUBE BY c1, c2, ..., ck
HAVING  COUNT(*) ≥ τ
```

The algorithms for iceberg cubes take at least k passes, where k is the number of dimensions over which the cube has to be computed. Our algorithms can be applied as a preprocessing step to the problem of computing Iceberg Cubes. If the clause `SUM(X)` is removed, our algorithm can enable computation of approximate Iceberg Cubes in a single pass. We are currently exploring the application of our algorithms to Iceberg Cubes.

2.3 Frequent Itemsets and Association Rules

A transaction is defined to be a subset of items drawn from $\mathcal{I}$, the universe of all items. For a collection of transactions, an itemset $X \subseteq \mathcal{I}$ is said to have *support* s if X occurs as a subset in at least a fraction s of all transactions. Association rules over a set of transactions are rules of the form $X \Rightarrow Y$, where X and Y are subsets of $\mathcal{I}$ such that $X \cap Y = \phi$ and $X \cup Y$ has support exceeding a user-specified threshold s. The *confidence* of a rule $X \Rightarrow Y$ is the value $\frac{support(X \cup Y)}{support(X)}$. Usually, only those rules are produced whose confidence exceeds a user-specified threshold.

The *Apriori* algorithm [AS94] was one of the first successful solutions for Association Rules. The problem actually reduces to that of computing frequent itemsets. Once frequent itemsets have been identified, identifying rules whose confidence exceeds the stipulated confidence threshold is straightforward.

The fastest algorithms to date employ two passes. One of the first two-pass techniques was proposed by Savasere et al [SON95] where the input is partitioned into chunks that fit in main memory. The first pass computes a set of candidate frequent itemsets. The second pass computes exact supports for all candidates. It is not clear how this algorithm can be adapted for streaming data since at the end of the first pass, there are no guarantees on the accuracy of frequency counts of candidates.

Another two pass approach is based on sampling [Toi96]. Briefly, the idea is to produce a sample in one pass, compute frequent itemsets along with a *negative border* in main memory on the samples and verify the validity of the negative border. The negative border is defined by a parameter ϵ. There is a small probability that the negative border is invalid, which happens when the frequency count of a highly-frequent element in the sample is less than its true frequency count over the entire dataset by more than ϵ. It can be shown that if we want this probability to be less than some value δ, the size of the sample should be at least $\frac{c}{\epsilon^2}\log(\delta^{-1})$, for some constant $c > 1$.

Toivonen's algorithm provides strong but probabilistic guarantees on the accuracy of its frequency counts. The scheme can indeed be adapted for data streams by using reservoir sampling [Vit85]. However, there are two problems: (a) false negatives occur because the error in frequency counts is two-sided; and, (b) for small values of ϵ, the number of samples is enormous, e.g., if $\epsilon = 0.0001$, we need over 100 million samples. The second problem might be ameliorated by employing *concise samples*, an idea first studied by Gibbons and Matias [GM98]. Essentially, repeated occurrences of an element can be compressed into an (element, count) pair. Adapting concise samples to handle variable-sized transactions would amount to designing a data structure similar to FP-Trees [HPY00], which would have to be readjusted periodically as frequencies of singleton items change. This idea has not been explored by the data mining community yet. Still, the problem of false negatives in Toivonen's approach remains. Our algorithms never produce false negatives.

CARMA [Hid99] is another two-pass algorithm that gives loose guarantees on the quality of output it produces at the end of the first pass. An upper and lower bound on the frequency of each itemset is displayed to the user continuously. However, there is no guarantee that the sizes of these intervals are small. CARMA provides interval information on a *best-effort* basis with the goal of providing the user with coarse feedback while it is running.

Online/incremental maintenance of association rules for offline streams in data warehouses is an important practical problem. We provide the first solution that is guaranteed to require no more than one pass over the entire database.

2.4 Network Flow Identification

Measurement and monitoring of network traffic is required for management of complex Internet backbones. Such measurement is essential for short-term monitoring (hot spot and denial-of-service attack detection), longer term traffic engineering (rerouting traffic and upgrading selected links), and accounting (usage based pricing). Identifying flows in network traffic is important for all these applications. A flow is defined as a sequence of transport layer (TCP/UDP) packets that share the same source+destination addresses.

Estan and Verghese [EV01] recently proposed algorithms for identifying flows that exceed a certain threshold, say 1%. Their algorithms are a combination of repeated hashing and sampling, similar to those for Iceberg Queries. Our algorithm is directly applicable to the problem of network flow identification. It beats the algorithm in [EV01] in terms of space requirements.

3 Problem Definition

In this section, we describe our notation, our definition of approximation, and the goals of our algorithms.

Our algorithm will accept two user-specified parameters: a support threshold $s \in (0, 1)$, and an error parameter $\epsilon \in (0, 1)$ such that $\epsilon \ll s$. Let N denote the current length of the stream, i.e., the number of tuples seen so far. In some applications, a tuple is treated as a single item. In others, it denotes a set of items. We will use the term *item(set)* as a short-hand for *item or itemset*.

At any point of time, our algorithm can be asked to produce a list of item(set)s along with their estimated frequencies. The answers produced by our algorithm will have the following guarantees:

1. All item(set)s whose true frequency exceeds sN are output. There are *no false negatives*.
2. No item(set) whose true frequency is less than $(s - \epsilon)N$ is output.
3. Estimated frequencies are *less* than the true frequencies *by at most* ϵN.

Imagine a user who is interested in identifying all items whose frequency is at least 0.1% of the entire stream seen so far. Then $s = 0.1\%$. The user is free to set ϵ to whatever she feels is a comfortable margin of error. As a rule of thumb, she could set ϵ to one-tenth or one-twentieth the value of s and use our algorithm. Let us assume she chooses $\epsilon = 0.01\%$ (one-tenth of s). As per Property 1, all elements with frequency exceeding $s = 0.1\%$ will be output; there will be no false negatives. As per Property 2, no element with frequency below 0.09% will be output. This leaves elements with frequencies between 0.09% and 0.1%. These might or might not form part of the output. Those that make their way to the output are false positives. Further, as per property 3, all individual frequencies are less than their true frequencies by at most 0.01%.

The approximation in our algorithms has two aspects:

(a) high frequency false positives, and (b) small errors in individual frequencies. Both kinds of errors are tolerable in the applications outlined in Section 2. We say that an algorithm maintains an *ϵ-deficient synopsis* if its output satisfies the aforementioned properties.

Our goal is to devise algorithms to support ϵ-deficient synopses using as little main memory as possible.

4 Algorithms for Frequent Items

In this section, we describe algorithms for computing ϵ-deficient synopses for singleton items. Extensions to sets of items will be described in Section 5. We actually devised two algorithms for frequency counts. Both provide the approximation guarantees outlined in Section 3. The first algorithm, Sticky Sampling, is probabilistic. It fails to provide correct answers with a miniscule probability of failure. The second algorithm, Lossy Counting, is deterministic. Interestingly, we experimentally show that Lossy Counting performs better in practice, although it has a theoretically worse worst-case bound.

4.1 Sticky Sampling Algorithm

In this section, we describe a sampling based algorithm for computing an ϵ-deficient synopsis over a data stream of singleton items. The algorithm is probabilistic and is said to fail if any of the three properties in Section 3 is not satisfied. The user specifies three parameters: support s, error ϵ and probability of failure δ.

Our data structure $\mathcal{S}$ is a set of entries of the form (e, f), where f estimates the frequency of an element e belonging to the stream. Initially, $\mathcal{S}$ is empty, and the sampling rate r is set to 1. Sampling an element with rate r means that we select the element with probability $\frac{1}{r}$. For each incoming element e, if an entry for e already exists in $\mathcal{S}$, we increment the corresponding frequency f; otherwise, we sample the element with rate r. If this element is selected by sampling, we add an entry $(e, 1)$ to $\mathcal{S}$; otherwise, we ignore e and move on to the next element in the stream.

The sampling rate r varies over the lifetime of a stream as follows: Let $t = \frac{1}{\epsilon}\log(s^{-1}\delta^{-1})$. The first $2t$ elements are sampled at rate $r = 1$, the next $2t$ elements are sampled at rate $r = 2$, the next $4t$ elements are sampled at rate $r = 4$, and so on. Whenever the sampling rate changes, we also scan entries in $\mathcal{S}$, and update them as follows: For each entry (e, f), we repeatedly toss an unbiased coin until the coin toss is successful, diminishing f by one for every unsuccessful outcome; if f becomes 0 during this process, we delete the entry from $\mathcal{S}$. The number of unsuccessful coin tosses follows a geometric distribution. Its value can be efficiently computed [Vit85]. Effectively, we have transformed $\mathcal{S}$ to exactly the state it would have been in, had we been sampling with the new rate from the very beginning.

When a user requests a list of items with threshold s, we output those entries in $\mathcal{S}$ where $f \geq (s - \epsilon)N$.

Theorem 4.1 *Sticky Sampling computes an ϵ-deficient synopsis with probability at least $1 - \delta$ using at most $\frac{2}{\epsilon}\log(s^{-1}\delta^{-1})$ expected number of entries.*

Proof: The first $2t$ elements find their way into $\mathcal{S}$. When the sampling rate is $r \geq 2$, we have $N = rt + rt'$, for some $t' \in [1, t)$. It follows that $\frac{1}{r} \geq \frac{t}{N}$.

Any error in the frequency count of an element e corresponds to a sequence of unsuccessful coin tosses during the first few occurrences of e. The length of this sequence follows a geometric distribution. The probability that this length exceeds ϵN is at most $(1 - \frac{1}{r})^{\epsilon N}$, which is less than $(1 - \frac{t}{N})^{-\epsilon N}$, which in turn is less than $e^{-\epsilon t}$.

The number of elements with frequency at least s is no more than $\frac{1}{s}$. Therefore, the probability that the estimate for *any* of them is deficient by ϵN, is at most $\frac{e^{-\epsilon t}}{s}$. The probability of failure should be at most δ. This yields $t \geq \frac{1}{\epsilon}\log(s^{-1}\delta^{-1})$. Since the space requirements are $2t$, the space bound follows. $\square$

We call our algorithm *Sticky Sampling* because $\mathcal{S}$ sweeps over the stream like a magnet, attracting all elements which already have an entry in $\mathcal{S}$. Note that the space complexity for Sticky Sampling is independent of N, the current length of the stream. The idea of maintaining counts of samples was first presented by Gibbons and Matias [GM98], who used it to compress their samples and solve the Top-k problem where the k most frequent items need to be identified. Our algorithm is different in that the sampling rate r increases logarithmically proportional to the size of the stream, and that we guarantee to produce all items whose frequency exceeds s, not just the top k.

4.2 Lossy Counting Algorithm

In this section, we describe a deterministic algorithm that computes frequency counts over a stream of single item transactions, satisfying the guarantees outlined in Section 3 using at most $\frac{1}{\epsilon}\log(\epsilon N)$ space where N denotes the current length of the stream. The user specifies two parameters: support s and error ϵ. This algorithm performs better than Sticky Sampling in practice although theoretically, its worst-case space complexity is worse.

Definitions: The incoming stream is conceptually divided into *buckets* of width $w = \lceil \frac{1}{\epsilon} \rceil$ transactions each. Buckets are labeled with *bucket ids*, starting from 1. We denote the *current bucket id* by $b_{current}$, whose value is $\lceil \frac{N}{w} \rceil$. For an element e, we denote its true frequency in the stream seen so far by f_e. Note that ϵ and w are fixed while N, $b_{current}$ and f_e are running variables whose values change as the stream progresses.

Our data structure $\mathcal{D}$ is a set of entries of the form (e, f, Δ), where e is an element in the stream, f is an integer representing its estimated frequency, and Δ is the maximum possible error in f.

Algorithm: Initially, $\mathcal{D}$ is empty. Whenever a new element e arrives, we first lookup $\mathcal{D}$ to see whether an entry for e already exists or not. If the lookup succeeds, we update the entry by incrementing its frequency f by one. Otherwise, we create a new entry of the form $(e, 1, b_{current} - 1)$. We also prune $\mathcal{D}$ by deleting some of its entries at bucket boundaries, i.e., whenever $N \equiv 0 \mod w$. The rule for deletion is simple: an entry (e, f, Δ) is deleted if $f + \Delta \leq b_{current}$. When a user requests a list of items with threshold s, we output those entries in $\mathcal{S}$ where $f \geq (s - \epsilon)N$.

For an entry (e, f, Δ), f represents the exact frequency count of e ever since this entry was inserted into $\mathcal{D}$. The value of Δ assigned to a new entry is the maximum number of times e could have occurred in the first $b_{current} - 1$ buckets. This value is exactly $b_{current} - 1$. Once an entry is inserted into $\mathcal{D}$, its Δ value remains unchanged.

Lemma 4.1 *Whenever deletions occur,* $b_{current} \leq \epsilon N$.

Lemma 4.2 *Whenever an entry* (e, f, Δ) *gets deleted,* $f_e \leq b_{current}$.

Proof: We prove by induction. Base case: $b_{current} = 1$. An entry (e, f, Δ) is deleted only if $f = 1$, which is also its true frequency f_e. Thus, $f_e \leq b_{current}$.

Induction step: Consider an entry (e, f, Δ) that gets deleted for some $b_{current} > 1$. This entry was inserted when bucket $\Delta + 1$ was being processed. An entry for e could possibly have been deleted as late as the time when bucket Δ became full. By induction, the true frequency of e when that deletion occurred was no more than Δ. Further, f is the true frequency of e ever since it was inserted. It follows that f_e, the true frequency of e in buckets 1 through $b_{current}$, is at most $f + \Delta$. Combined with the deletion rule that $f + \Delta \leq b_{current}$, we get $f_e \leq b_{current}$. □

Lemma 4.3 *If e does not appear in $\mathcal{D}$, then* $f_e \leq \epsilon N$.

Proof: If the Lemma is true for an element e whenever it gets deleted, i.e., when $N \equiv 0 \mod w$, it is true for all other N also. From Lemma 4.2 and Lemma 4.1, we infer that $f_e \leq \epsilon N$ whenever it gets deleted. □

Lemma 4.4 *If* $(e, f, \Delta) \in \mathcal{D}$*, then* $f \leq f_e \leq f + \epsilon N$.

Proof: If $\Delta = 0$, $f = f_e$. Otherwise, e was possibly deleted sometimes in the first Δ buckets. From Lemma 4.2, we infer that the exact frequency of e when the last such deletion took place, is at most Δ. Therefore, $f_e \leq f + \Delta$. Since $\Delta \leq b_{current} - 1 \leq \epsilon N$, we conclude that $f \leq f_e \leq f + \epsilon N$. □

Consider elements whose true frequency exceeds ϵN. There cannot be more than $\frac{1}{\epsilon}$ such elements. Lemma 4.3 shows that all of them have entries in $\mathcal{D}$. Lemma 4.4 further shows that the estimated frequencies of all such elements are accurate to within ϵN. Thus, $\mathcal{D}$ correctly maintains an ϵ-deficient synopsis. The next theorem shows that the number of low frequency elements (which occur less than ϵN times) is not too large.

Theorem 4.2 *Lossy Counting computes an ϵ-deficient synopsis using at most $\frac{1}{\epsilon}\log(\epsilon N)$ entries.*

Proof: Let $B = b_{current}$ be the current bucket id. For each $i \in [1, B]$, let d_i denote the number of entries in $\mathcal{D}$ whose bucket id is $B - i + 1$. The element corresponding to such an entry must occur at least i times in buckets $B - i + 1$ through B; otherwise, it would have been deleted. Since the size of each bucket is w, we get the following constraints:

$$\sum_{i=1}^{j} i d_i \leq jw \qquad \text{for} \quad j = 1, 2, \ldots, B. \tag{1}$$

We claim that

$$\sum_{i=1}^{j} d_i \leq \sum_{i=1}^{j} \frac{w}{i} \qquad \text{for} \quad j = 1, 2, \ldots, B. \tag{2}$$

We prove Inequality (2) by induction. The base case $j = 1$ follows from Inequality (1) directly. Assume that Inequality (2) is true for $j = 1, 2, \ldots, p - 1$. We will show that it is true for $j = p$ as well. Adding Inequality (1) for $j = p$ to $p - 1$ instances of Inequality (2) (one each for i varying from 1 to $p - 1$) gives us $\sum_{i=1}^{p} i d_i + \sum_{i=1}^{1} d_i + \sum_{i=1}^{2} d_i + \cdots + \sum_{i=1}^{p-1} d_i \leq pw + \sum_{i=1}^{1} \frac{w}{i} + \sum_{i=1}^{2} \frac{w}{i} + \cdots + \sum_{i=1}^{p-1} \frac{w}{i}$. Upon rearrangement, we get $p \sum_{i=1}^{p} d_i \leq pw + \sum_{i=1}^{p-1} \frac{(p-i)w}{i}$ which then readily simplifies to Inequality (2) for $j = p$, thereby completing the induction step.
Since $|\mathcal{D}| = \sum_{i=1}^{B} d_i$, from Inequality 2, we get $|\mathcal{D}| \leq \sum_{i=1}^{B} \frac{w}{i} \leq \frac{1}{\epsilon}\log B = \frac{1}{\epsilon}\log(\epsilon N)$ □

If we make the assumption that in real world datasets, elements with very low frequency (at most $\frac{\epsilon N}{2}$) tend to occur more or less uniformly at random, then Lossy Counting requires no more than $\frac{7}{\epsilon}$ space, as proved in the Appendix. We note that the IBM test data generator for data mining [AS94] for generating synthetic datasets for association rules, indeed chooses low frequency items uniformly randomly from a fixed distribution.

4.3 Sticky Sampling vs Lossy Counting

In this section, we compare Sticky Sampling and Lossy Counting. We show that Lossy Counting performs significantly better for streams with random arrival distributions.

A glance at Theorems 4.1 and 4.2 suggests that Sticky Sampling should require constant space, while Lossy Counting should require space that grows logarithmically with N, the length of the stream. However, these theorems do not tell the whole story because they bound *worst-case* space complexities. For Sticky Sampling, the worst case amounts to a sequence with no duplicates, arriving in any order. For Lossy Counting, the worst case corresponds to a rather pathological sequence that we suspect would almost never occur in practice. We now show that if the arrival order of the elements is uniformly random, Lossy Counting is superior by a large factor.

We studied two streams. One had no duplicates. The other was highly skewed and followed the Zipfian distribution. Most streams in practice would be somewhere in between in terms of their frequency distributions. For both streams, we assumed uniformly random arrival order.

Table 1 compares the two algorithms for varying values of s. The value of ϵ is consistently chosen to be one-tenth of s. Worst case space complexities are obtained by plugging values into Theorems 4.1 and 4.2.

ϵ	s	*SS worst*	*LC worst*	*SS Zipf*	*LC Zipf*	*SS Uniq*	*LC Uniq*
0.1%	1.0%	27K	9K	6K	419	27K	1K
0.05%	0.5%	58K	17K	11K	709	58K	2K
0.01%	0.1%	322K	69K	37K	2K	322K	10K
0.005%	0.05%	672K	124K	62K	4K	672K	20K

Table 1: Memory requirements in terms of number of entries. LC denotes Lossy Counting. SS denotes Sticky Sampling. Worst denotes worst-case bound. Zipf denotes Zipfian distribution with parameter 1.25. Uniq denotes a stream with no duplicates. Length of stream $N = 10^7$. Probability of failure $\delta = 10^{-4}$.

Figure 1 shows the amount of space required for the two streams as a function of N, with support $s = 1\%$, error $\epsilon = 0.1\%$, and probability of failure $\delta = 10^{-4}$. The kinks in the curve for Sticky Sampling correspond to re-sampling. They are $\log_{10}(2)$ units apart on the X-axis. For kinks for Lossy Counting correspond to bucket boundaries when deletions occur.

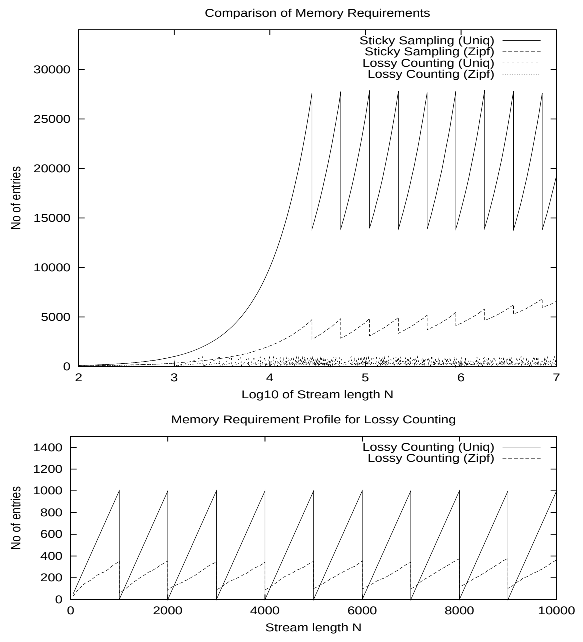


Figure 1: Memory requirements in terms of number of entries for support $s = 1\%$, error $\epsilon = 0.1\%$, probability of failure $\delta = 10^{-4}$. Zipf denotes a Zipfian distribution with parameter 1.25. Uniq denotes a stream with no duplicates. The bottom figure magnifies a section of the barely visible lines in the upper graph.

Sticky Sampling performs worse because of its tendency to remember every unique element that gets sampled. Lossy Counting, on the other hand, is good at pruning low frequency elements quickly; only high frequency elements survive. For highly skewed data, both algorithms require much less space than their worst-case bounds.

4.4 Comparison with Alternative Approaches

A well-known technique for estimating frequency counts employs uniform random sampling. If the sample size is at least $\frac{1}{\epsilon^2} \log \frac{1}{\delta}$, the relative frequency of any single element is accurate to within a fraction ϵ, with probability at least $1-\delta$. This basic idea was used by Toivonen [Toi96] to devise a sampling algorithm for association rules. Sticky Sampling beats this approach by roughly a factor of $\frac{1}{\epsilon}$.

Another algorithm for maintaining ϵ-deficient synopsis can employ ϵ-approximate quantiles [MRL99]. The key idea is that an element with frequency exceeding s will recur several times if ϵ is small relative to s. It follows that if we set $\epsilon' = \epsilon/4$, and compute ϵ'-approximate histograms, high frequency elements can be deduced to within an error of ϵ. Using the algorithm due to Greenwald and Khanna [GK01], the worst-case space requirement would be $\frac{22}{\epsilon}\log(2\epsilon N)$, worse than that for Lossy Counting, as per Theorem 4.2. Also, it is not obvious how quantile algorithms can be efficiently extended to handle variable-sized *sets of items*, a problem we consider in Section 5.

We recently learned about an as yet unpublished algorithm [KPS02] for identifying elements whose frequency exceeds a fraction ϵ of the stream. The algorithm computes exact counts in two passes. However, the essential ideas can be used to maintain an ϵ-deficient synopsis using *exactly* $\frac{1}{\epsilon}$ space. In the first pass, the algorithm maintains $\frac{1}{\epsilon}$ elements along with their counters. Initially, all counters are free. Whenever a new element arrives, we check if a counter for this element exists. If so, we simply increment it. Otherwise, if a counter is free, it is assigned to this element with initial value 1. If all counters are in use, we repeatedly diminish *all* $\frac{1}{\epsilon}$ counters by 1 until some counter becomes free, i.e., its value drops to zero. Thereafter, we assign the newly arrived element a counter with initial value 1. This algorithm works for streams of singleton items. However, there does not appear to be a straightforward adaptation to the scenario where a stream of variable-sized transactions is being analyzed and the distribution of transaction sizes is not known. Moreover, if the input stream is Zipfian, the number of elements exceeding the threshold ϵ is significantly smaller than $\frac{1}{\epsilon}$. Table 1 shows that Lossy Counting actually takes much less than $\frac{1}{\epsilon}$ space. For example, with $\epsilon = 0.01\%$, roughly 2000 entries suffice, which is only 20% of $\frac{1}{\epsilon}$. The skew in the frequencies is even more when we consider the problem of identifying frequent *sets* of items in Section 5. For example, assume that all transactions are known to have a fixed size k and that frequent subsets of size 3 are being computed. An adaptation of algorithm [KPS02] would maintain $\frac{1}{\epsilon}\binom{k}{3}$ counters. Lossy Counting would require significantly less space, as experiments in Section 6 show.

5 Frequent Sets of Items – From Theory to Practice

In this section, we develop a Lossy Counting based algorithm for computing frequency counts over streams that consist of *sets of items*. This section is less theoretical in nature than the preceding one. The focus is on system-level issues and implementation artifices for optimizing memory and speed.

5.1 Frequent Itemsets Algorithm

The input to the algorithm is a stream of transactions where each transaction is a *set of items* drawn from $\mathcal{I}$. We denote the current length of this stream by N. The user specifies two parameters: support s, and error ϵ. The challenge lies in handling variable sized transactions and avoiding explicit enumeration of all subsets of any transaction.

Our data structure $\mathcal{D}$ is a set of entries of the form (set, f, Δ), where set is a subset of items, f is an integer representing its approximate frequency, and Δ is the maximum possible error in f. Initially, $\mathcal{D}$ is empty.

Imagine dividing the incoming transaction stream into *buckets*, where each bucket consists of $w = \lceil \frac{1}{\epsilon} \rceil$ transactions. Buckets are labeled with *bucket ids*, starting from 1. We denote the *current bucket id* by $b_{current}$. We do not process the input stream transaction by transaction. Instead, we try to fill available main memory with as many transactions as possible, and then process such a *batch* of transactions together. This is where the algorithm differs from that presented in Section 4.2. Over time, the amount of main memory available might increase/decrease. Let β denote the number of buckets in main memory in the current batch being processed. We update $\mathcal{D}$ as follows:

- UPDATE_SET: For each entry $(set, f, \Delta) \in \mathcal{D}$, update f by counting the occurrences of set in the current batch. If the updated entry satisfies $f + \Delta \leq b_{current}$, we delete this entry.
- NEW_SET: If a set set has frequency $f \geq \beta$ in the current batch and set does not occur in $\mathcal{D}$, create a new entry $(set, f, b_{current} - \beta)$.

It is easy to see that every set set whose true frequency $f_{set} \geq \epsilon N$, has an entry in $\mathcal{D}$. Also, if an entry $(set, f, \Delta) \in \mathcal{D}$, then, the true frequency f_{set} satisfies the inequality $f \leq f_{set} \leq f + \Delta$. When a user requests a list of items with threshold s, we output those entries in $\mathcal{D}$ where $f \geq (s - \epsilon)N$.

It is important that β be a large number. The reason is that any subset of $\mathcal{I}$ that occurs $\beta + 1$ times or more, contributes an entry to $\mathcal{D}$. For a smaller β, more spurious subsets find their way into $\mathcal{D}$.

In the next section, we show how $\mathcal{D}$ can be represented compactly and how UPDATE_SET and NEW_SET can be implemented efficiently.

5.2 Data Structures

Our implementation has three modules: BUFFER, TRIE, and SETGEN. BUFFER repeatedly reads in a batch of transactions into available main memory. TRIE maintains the data structure $\mathcal{D}$ described earlier. SETGEN operates on the current batch of transactions. It enumerates subsets of these transactions along with their frequencies. Together with TRIE, it implements the UPDATE_SET and NEW_SET operations. The challenge lies in designing a space efficient representation of TRIE and a fast algorithm for SETGEN that avoids generating all possible subsets of itemsets.

Buffer: This module repeatedly reads in a batch of transactions into available main memory. Transactions are sets of item-id's. They are laid out one after the other in a big array. A bitmap is used to remember transaction boundaries. A bit per per item-id denotes whether this item-id is the last member of some transaction or not. After reading in a batch, BUFFER sorts each transaction by its item-id's.

Trie: This module maintains the data structure $\mathcal{D}$ outlined in Section 5.1. Conceptually, it is a forest (a set of trees) consisting of labeled nodes. Labels are of the form $\langle item\_id, f, \Delta, level \rangle$, where $item\_id$ is an item-id, f is its estimated frequency, Δ is the maximum possible error in f, and $level$ is the distance of this node from the root of the tree it belongs to. The root nodes have level 0. The level of any other node is one more than that of its parent. The children of any node are ordered by their item-id's. The root nodes in the forest are also ordered by item-id's. A node in the tree represents an itemset consisting of item-id's in that node and all its ancestors. There is a 1 to 1 mapping between entries in $\mathcal{D}$ and nodes in TRIE.

Tries are used by several Association Rules algorithms. Hash tries [AS94] are a popular choice. Usual implementations of $\mathcal{D}$ as a trie would require pointers and variable-sized memory segments (because the number of children of a node changes over time).

Our TRIE is different from traditional implementations. Since tries are the bottleneck as far as space is concerned, we designed them to be as compact as possible. We maintain TRIE as *an array of entries* of the form $\langle item\_id, f, \Delta, level \rangle$ *corresponding to the pre-order traversal of the underlying trees*. Note that this is equivalent to a *lexicographic ordering of the subsets* it encodes. There are no pointers from any node to its children or its siblings. The $level$'s compactly encode the underlying tree structure. Our representation is okay because tries are always scanned sequentially, as we show later.

SetGen: This module generates subsets of item-id's along with their frequencies in the current batch of transactions in lexicographic order. Not all possible subsets need to be generated. A glance at the description of UPDATE_SET and NEW_SET operations reveals that a subset must be enumerated iff either it occurs in TRIE or its frequency in the current batch exceeds β. SETGEN uses the following pruning rule:

> *If a subset S does not make its way into* TRIE *after application of both* UPDATE_SET *and* NEW_SET*, then no supersets of S should be considered.*

This is similar to the Apriori pruning rule. We describe an efficient implementation of SETGEN in greater detail later.

Overall Algorithm

BUFFER repeatedly fills available main memory with as many transactions as possible, and sorts them. SETGEN operates on the current batch of transactions. It generates sets of itemsets along with their frequency counts in lexicographic order. It limits the number of subsets using the pruning rule. Together, TRIE and SETGEN implement the UPDATE_SET and NEW_SET operations. In the end, TRIE stores the updated data structure $\mathcal{D}$, and BUFFER gets ready to read in the next batch.

5.3 Efficient Implementations

Buffer: If item-id's are successive integers from 1 thru $|\mathcal{I}|$, and if $\mathcal{I}$ is small enough (say, less than 1 million), we maintain *exact frequency counts* for singleton sets. If $|\mathcal{I}| = 10^5$, we need an array of size only $0.4MB$. When exact frequency counts are maintained, BUFFER first prunes away those item-id's whose frequency is less than ϵN, and then sorts the transactions. Note that N is the length of the stream up to and including the current batch of transactions.

Trie: As SETGEN generates its sequence of sets and associated frequencies, TRIE needs to be updated. Adding or deleting TRIE nodes *in situ* is made difficult by the fact that TRIE is a compact array. However, we take advantage of the fact that the sets produced by SETGEN (and therefore, the sequence of additions and deletions) are lexicographically ordered. Recall that our compact TRIE stores its constituent subsets in their lexicographic order. This lets SETGEN and TRIE work hand in hand.

We maintain TRIE not as one huge array, but as a set of fairly large-sized chunks of memory. Instead of modifying the original trie, we create a new TRIE afresh. Chunks from the old TRIE are freed as soon as they are not required. Thus, the overhead of maintaining two Tries is not significant. By the time SETGEN finishes, the chunks of the original trie have been completely discarded.

For finite streams, an important TRIE optimization pertains to the last batch of transactions when the value of β, the number of buckets in BUFFER, could be small. Instead of applying the rules in Section 5.1, we prune nodes in the trie more aggressively by setting the threshold for deletion to sT instead of $b_{current} \approx \epsilon T$. This is because the lower frequency nodes do not contribute to the final output.

SetGen: This module is the bottleneck in terms of time for our algorithm. Optimizing it has made it fairly complex. We describe the salient features of our implementation.

SETGEN employs a priority queue called *Heap* which initially contains pointers to smallest item-id's of all transactions in BUFFER. Duplicate members (pointers pointing to the same item-id) are maintained together and they constitute a single entry in *Heap*. In fact, we chain all the pointers together, deriving the space for this chain from BUFFER itself. When an item-id in BUFFER is inserted into *Heap*, the 4-byte integer used to represent an item-id is converted into a 4-byte pointer. When a heap entry is removed, the pointers are restored back to item-id's.

SETGEN repeatedly processes the smallest item-id in *Heap* to generate singleton sets. If this singleton belongs to TRIE after UPDATE_SET and NEW_SET rules have been applied, we try to generate the next set in lexicographic sequence by extending the current singleton set. This is done by invoking SETGEN recursively with a new heap created out of successors of the pointers to item-id's just removed and processed. The successors of an item-id is the item-id following it in its transaction. Last item-id's of transactions have no successors. When the recursive call returns, the smallest entry in *Heap* is removed and all successors of the currently smallest item-id are added to *Heap* by following the chain of pointers described earlier.

5.4 System Issues and Optimizations

BUFFER scans the incoming stream by memory mapping the input file. This saves time by getting rid of double copying of file blocks. The UNIX system call for memory mapping files is `mmap()`. The accompanying `madvise()` interface allows a process to inform the operating systems of its intent to read the file sequentially. We used the standard `qsort()` to sort transactions. The time taken to read and sort transactions pales in comparison with the time taken by SETGEN, obviating the need for a custom sort routine. Threading SETGEN and BUFFER does not help because SETGEN is significantly slower.

Tries are written and read sequentially. They are operational when BUFFER is being processed by SETGEN. At this time, the disk is idle. Further, the rate at which tries are scanned (read/written) is much smaller than the rate at which sequential disk I/O can be done. It is indeed possible to maintain TRIE on disk without any loss in performance. This has two important advantages:

(a) The size of a trie is not limited by the size of main memory available, as is the case with other algorithms. This means that our algorithm can function even when the amount of main memory available is quite small.

(b) Since most available memory can be devoted to BUFFER, we can work with smaller values of ϵ than other algorithms can handle. This is a big win.

TRIE is currently implemented as a pair of anonymous memory mapped segments. They can be associated with actual files, if the user so desires. Since tries are read/written sequentially, as against being accessed randomly, it is possible to compress/decompress it on the fly as sections of it are read/written to disk. Our current implementation does not attempt any compression; we use four `int`'s for node labels. Writing TRIE to disk violates a pedantic definition of single-pass algorithms. However, we should note that the term single-pass is meaningful only for disk-bound applications. Our program is cpu-bound.

Memory requirements for *Heap* are modest. Available main memory is consumed primarily by BUFFER, assuming TRIE are on disk. Our implementation allows the user to specify the size of BUFFER.

5.5 Novel Features of our Technique

Our implementation differs from Apriori and its variants in one important aspect: there is no candidate generation phase. Apriori first finds all frequent itemsets of size k before finding frequent itemsets of size $k+1$. This amounts to a breadth first search of the frequent itemsets on a lattice. Our algorithm carries out a depth first search. Incidentally, BUC [BR99] also uses repeated depth first traversals for Iceberg Cube computation. However, it makes d passes over the entire data where d is the number of dimensions in the cube.

The idea of using compact disk-based tries is novel. It allows us to compute frequent itemsets under low memory conditions. It also enables our algorithm to handle smaller values of support threshold than previously possible.

6 Experimental Results

We experimented with two kinds of datasets: streams of market-basket transactions, and text document sequences.

6.1 Frequent Itemsets over Streams of Transactions

Our experiments were carried out on the IBM test data generator [AS94]. We study two data-streams of size 1 million transactions each. One has an average transaction size of 10 with average large itemset size of 4. The other has average transaction size 15 and average large itemset size 6. Following the conventions set forth in [AS94], the names of the datasets are $T10.I4.1000K$ and $T15.I6.1000K$, where the three numbers denote the average transaction size (T), the average large itemset size (I) and the number of transactions respectively. Items were drawn from a universe of $\mathcal{I} = 10K$ unique items. The raw sizes of the two streams were 49 MB and 69 MB respectively. All experiments were carried out on a $933MHz$ Pentium III processor running Linux Kernel version 2.2.16.

In our experiments, we always fix $\epsilon = 0.1s$ (one-tenth of s). Moreover, the amount of main memory required by our programs is dominated by BUFFER, whose size is stipulated by the user. We are then left with four parameters that we study in our experiments: support s, number of transactions N, size of BUFFER, and total time taken. We measure wall clock time.

In Figure 2, we plot times taken by our algorithm for values of support s ranging from 0.1% to 1%, and BUFFER size ranging from 4 MB to 44 MB. The leftmost graphs show how decreasing s leads to increases in running time.

The kinks in the middle graphs in Figure 2 have an interesting explanation. The graphs plot running time for varying BUFFER sizes. For a fixed value of support s, the running time sometimes increases as BUFFER size increases. This happens due to a TRIE optimization we described in Section 5.3. For finite streams, when the last batch of transactions is being processed, the threshold is raised to sN. This leads to considerable savings in the running time of the last batch. In Figure 2, when BUFFER size is $20MB$, the input is split into only two (almost equal sized) batches. As BUFFER size increases, the first batch increases in size, leading to an increase in running time. Finally, when BUFFER size reaches 40 MB, it is big enough to accommodate the entire input as one batch, which is rapidly processed. This leads to a sharp decrease in running time. Increasing BUFFER size further has no effect on running time.

The rightmost graphs in Figure 2 show that the running time is linearly proportional to the length of the stream. The curve flattens in the end as processing the last batch is faster owing to the TRIE optimization mentioned in Section 5.3.

The disk bandwidth required to read the input file was always less than 1 MBps. This is a very low rate when compared with modern day disks. A single high performance SCSI disk can deliver between 20 and 160 MBps. This confirms that frequent itemset computation is cpu bound.

An interesting fact that emerged from our experiments was that the error in the output was almost zero. Over 99% of the itemsets reported in the output had 0% error. This happens because a highly frequent itemset invariably occurs within the first batch of transactions. Once it enters our data structure, it is very unlikely to get deleted. There are rarely any false positives for the same reason (the frequencies of all elements in the range $(s - \epsilon, s)$ are also accurate). This suggests that we might be able to set ϵ to a higher value and still get accurate results. A higher error rate might be observed in highly skewed data or if global data characteristics change, e.g. if the stream is sorted.

Comparison with Apriori

For comparison with the well known Apriori Algorithm [AS94], we down-loaded a publicly available package written by Christian Borgelt[1]. It is a pretty fast implementation of Apriori using prefix trees and is used in a commercial data mining package. Since the version of Linux we used did not support `mallinfo`, we re-linked Borgelt's program with the widely available `dlmalloc` library by Doug Lea[2]. We invoked `mallinfo` just before program termination to figure out its memory requirements.

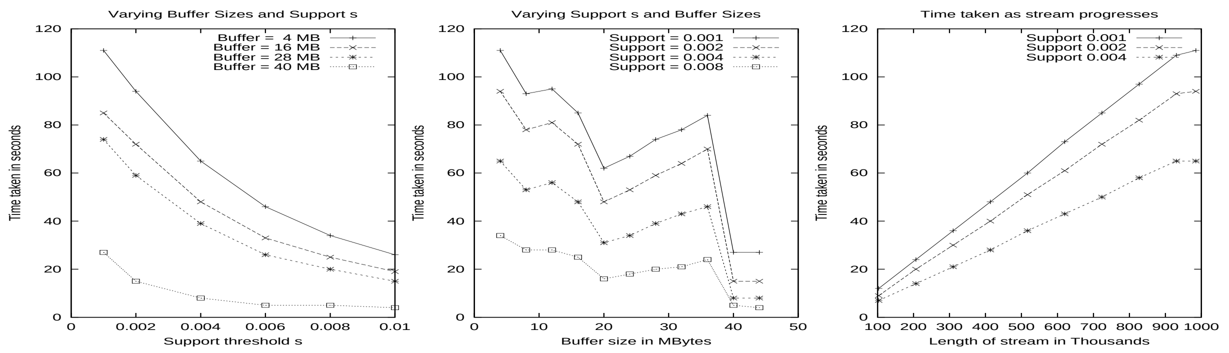
	Apriori		*Our algorithm with 4MB Buffer*		*Our algorithm with 44MB Buffer*	
Support	*Time*	*Memory*	*Time*	*Memory*	*Time*	*Memory*
0.001	99 s	81.96 MB	111 s	12.49 MB	27 s	45.16 MB
0.002	25 s	53.34 MB	94 s	9.92 MB	15 s	45.02 MB
0.004	14 s	48.09 MB	65 s	7.20 MB	8 s	45.00 MB
0.006	13 s	47.87 MB	46 s	6.03 MB	6 s	44.98 MB
0.008	13 s	47.86 MB	34 s	5.53 MB	4 s	44.95 MB
0.010	14 s	47.86 MB	26 s	5.22 MB	4 s	44.93 MB

Table 2: *Performance comparison for $T10.I4.1000K$ which has $1M$ transactions over $10K$ unique items with average transaction size* 10.

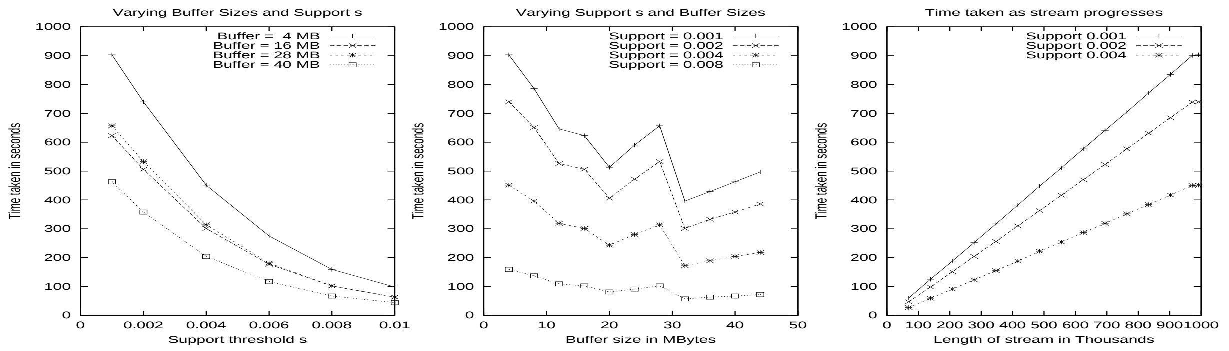
In Table 2, we compare Apriori with our algorithm for the dataset $T10.I4.1000K$, varying support from 0.1% to 1.0%. Error ϵ was set at 10% of s. We ran our algorithm twice, first with BUFFER set to 4 MB, and then, with

[1]`http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html`

[2]`http://g.oswego.edu/dl/html/malloc.html`

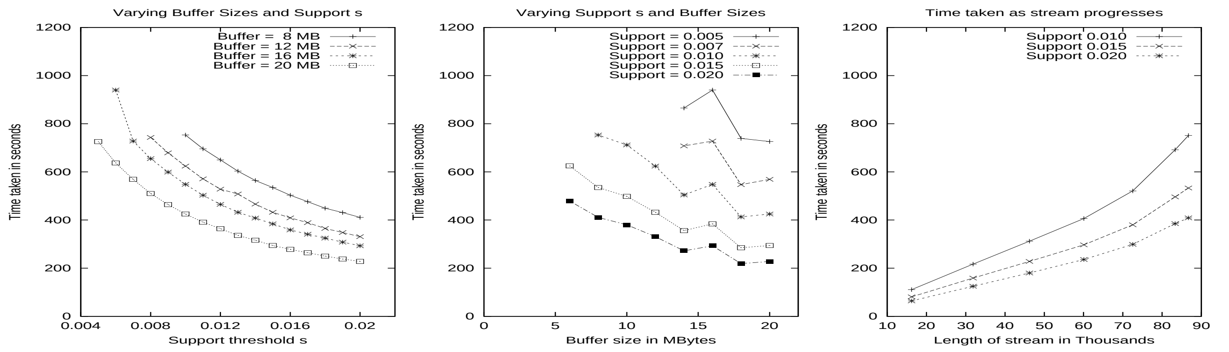


(a) Times taken for IBM test dataset $T10.I4.1000K$ with $10K$ items. No of transactions N was 1 million.

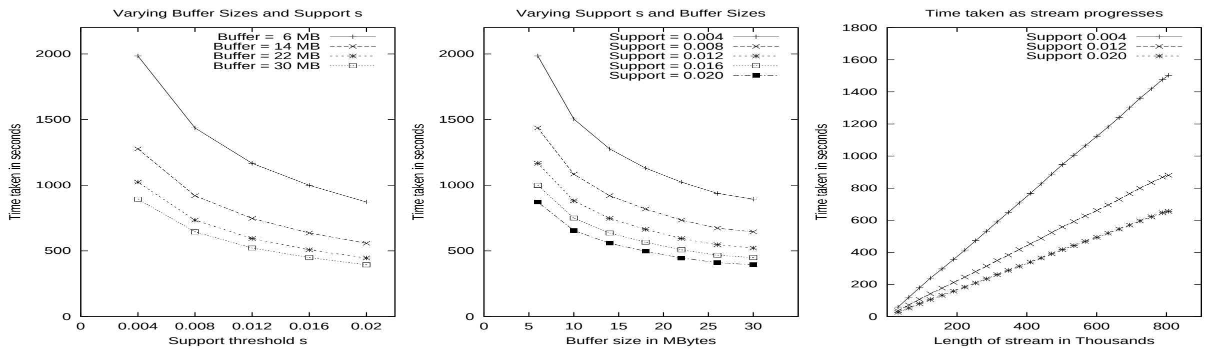


(b) Times taken for IBM test dataset $T15.I6.1000K$ with $10K$ items. No of transactions N was 1 million.

Figure 2: Experimental results for our algorithm over IBM test datasets.



(a) Times taken for frequent word-pairs in 100K web pages.



(b) Times taken for frequent word-pairs in 800K Reuters documents

Figure 3: Times taken for Iceberg Queries over Web pages and Reuters articles.

BUFFER set to 44 MB. The table shows the total memory required by the two programs. For our program, this includes the maximum cost of HEAP and TRIE, during runtime. As the value of support s increases, the memory required by TRIE decreases. This is because there are fewer itemsets with higher support. The size of the TRIE also decreases when BUFFER changes from 4 MB to 44 MB. This is because the value of β (see Section 5.1) increases. Therefore, there are fewer low frequency subsets (with frequency less than ϵ) that creep into the trie. It is interesting to observe that with BUFFER set to a small value, 4 MB, our algorithm was able to compute all frequent itemsets using much less memory than Apriori but more time. With BUFFER size 44 MB, the entire input fits in main memory. Our program beats Apriori be a factor of 2 to 3 showing that our main memory implementation is much faster.

6.2 Iceberg Queries

An iceberg query studied in [FSGM+98] was the identification of all pairs of words in a repository of $100,000$ web documents which occur at least $\tau = 500$ times together. Note that the relation R for this query is not explicitly materialized. This query is equivalent to identifying all word pairs that occur in at least $s = 0.5\%$ of all documents. We ran this query over two different datasets.

The first dataset was a collection of $100,000$ web pages crawled by WebBase, a web crawler developed at Stanford University [HRGMP00]. Words in each document were identified. Common stop-words [SB88] were removed. The resulting input file was 54 MB. Experiments for this dataset were carried out on a 933 MHz Pentium III machine running Linux Kernel version 2.2.16.

The second dataset was the well-known Reuters newswire dataset, containing $806,000$ news articles. The input file resulting from this dataset after removing stop-words was roughly 210 MB. Experiments for this dataset were carried out on a 700 MHz Pentium III machine running Linux Kernel version 2.2.16.

We study the interplay of N, the length of the stream, s, the support, time taken, and the size of BUFFER in Figure 3. The overall shape of the graphs is very similar to those for frequent itemsets over the IBM test datasets that we studied in the previous section.

For the sake of comparison with the algorithm presented in the original Iceberg Queries paper [FSGM$^+$98], we ran our program over $100,000$ web documents with support $s = 0.5\%$. This settings corresponds to the first query studied in [FSGM$^+$98] (see Figure 5 in their paper). We ran our program on exactly the same machine, a 200 MHz Sun Ultra/II with 256 MB RAM running SunOS 5.6. We fixed BUFFER at 15 MB. Our program processed the input in 4 batches, producing $396,000$ frequent word pairs. BUFFER, *Heap* and auxiliary data structures required 26 MB. The maximum size of a trie was 168 MB. Our program took 4500 seconds to complete. Fang et al [FSGM$^+$98] report that the same query required over 7000 seconds using roughly 30 MB main memory. Our algorithm is faster.

An interesting duality emerges between our approach and that of the algorithm in [FSGM$^+$98]. Our program scans the input just once, but repeatedly scans a temporary file on disk (the memory mapped TRIE). The Iceberg algorithm scans the input multiple times, but uses no temporary storage. The advantage in our approach is that it does not require a *lookahead* into the data stream.

7 Related and Future Work

Problems related to frequency counting that have been studied in the context of data streams include approximate frequency moments [AMS96], L^1 differences [FKSV99], distinct values estimation [FM85, WVZT90], bit counting [DGIM02], and top-k queries [GM98, CCFC02]. Algorithms over data streams that pertain to aggregation include approximate quantiles [MRL99, GK01], V-optimal histograms [GKS01b], wavelet based aggregate queries [GKMS01, MVW00], and correlated aggregate queries [GKS01a].

We are currently exploring the application of our basic techniques to sliding windows, data cubes, and two-pass algorithms for frequent itemsets.

8 Conclusions

We proposed novel algorithms for computing approximate frequency counts of elements in a data stream. Our algorithms require provably small main memory footprints. The problem of identifying frequent elements is at the heart of several important problems: iceberg queries, frequent itemsets, association rules, and packet flow identification. We can now solve each of them over streaming data.

We also described a highly optimized implementation for identifying frequent itemsets. In general, our algorithm produces approximate results. However, for the datasets we studied, our algorithm runs in one pass and produces exact results, beating previous algorithms in terms of time.

Our frequent itemsets algorithm can handle smaller values of support threshold than previously possible. It remains practical even in environments with moderate main memory. We believe that our algorithm provides a practical solution to the problem of maintaining association rules incrementally in a warehouse setting.

References

[AGP99] S. ACHARYA, P. B. GIBBONS, AND V. POOSALA. Aqua: A fast decision support system using approximate query answers. In *Proc. of 25th Intl. Conf. on Very Large Data Bases*, pages 754–755, 1999.

[AMS96] N. ALON, Y. MATIAS, AND M. SZEGEDY. The space complexity of approximating the frequency moments. In *Proc. of 28th Annual ACM Symp. on Theory of Computing*, pages 20–29, May 1996.

[AS94] R. AGRAWAL AND R. SRIKANT. Fast algorithms for mining association rules. In *Proc. of 20th Intl. Conf. on Very Large Data Bases*, pages 487–499, 1994.

[BR99] K. BEYER AND R. RAMAKRISHNAN. Bottom-up computation of sparse and iceberg cubes. In *Proc. of 1999 ACM SIGMOD*, pages 359–370, 1999.

[CCFC02] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. 29th Intl. Colloq. on Automata, Languages and Programming*, 2002.

[DGIM02] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. of 13th Annual ACM-SIAM Symp. on Discrete Algorithms*, January 2002.

[EV01] C. Estan and G. Verghese. New directions in traffic measurement and accounting. In *ACM SIGCOMM Internet Measurement Workshop*, November 2001.

[FKSV99] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate l1-difference algorithm for massive data streams. In *Proc. of 40th Annual Symp. on Foundations of Computer Science*, pages 501–511, 1999.

[FM85] P. Flajolet and G. N. Martin. Probabilistic counting algorithms. *J. of Comp. and Sys. Sci*, 31:182–209, 1985.

[FSGM+98] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing iceberg queries efficiently. In *Proc. of 24th Intl. Conf. on Very Large Data Bases*, pages 299–310, 1998.

[GK01] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proc. of 2001 ACM SIGMOD*, pages 58–66, 2001.

[GKMS01] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proc. of 27th Intl. Conf. on Very Large Data Bases*, 2001.

[GKS01a] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. of 2001 ACM SIGMOD*, pages 13–24, 2001.

[GKS01b] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. of 33rd Annual ACM Symp. on Theory of Computing*, pages 471–475, July 2001.

[GM98] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proc. of 1998 ACM SIGMOD*, pages 331–342, 1998.

[Hid99] C. Hidber. Online association rule mining. In *Proc. of 1999 ACM SIGMOD*, pages 145–156, 1999.

[HPDW01] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *Proc. of 2001 ACM SIGMOD*, pages 1–12, 2001.

[HPY00] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of 2000 ACM SIGMOD*, pages 1–12, 2000.

[HRGMP00] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. Webbase: A repository of web pages. *Computer Networks*, 33:277–293, 2000.

[KPS02] R. Karp, C. Papadimitriou, and S. Shenker. – *Personal Communication*, 2002.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1 edition, 1995.

[MRL99] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proc. of 1999 ACM SIGMOD*, pages 251–262, 1999.

[MVW00] Y. Matias, J. S. Vitter, and M. Wang. Dynamic maintenance of wavelet-based histograms. In *Proc. of 26th Intl. Conf. on Very Large Data Bases*, pages 101–110, 2000.

[PCY95] J. S. Park, M. S. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *Proc. of 1995 ACM SIGMOD*, pages 175–186, 1995.

[SB88] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(1), 1988.

[SON95] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of 21st Intl. Conf. on Very Large Data Bases*, pages 432–444, 1995.

[Toi96] H. Toivonen. Sampling large database for association rules. In *Proc. of 22nd Intl. Conf. on Very Large Data Bases*, pages 134–145, 1996.

[Vit85] J S Vitter. Random Sampling with a Reservoir. *ACM Tran. Math. Software*, 11(1):37–57, 1985.

[WVZT90] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. on Database Systems*, 15(2):208–229, 1990.

APPENDIX

Theorem A-1 *For Lossy Counting, if stream elements are drawn independently from a fixed probability distribution,* $\mathbf{E}[|\mathcal{D}|] < \frac{7}{\epsilon}$.

Proof: For an element e, let p_e be the probability with which it is chosen to be the next element in the stream. Consider elements with $p_e \geq \frac{\epsilon}{2}$. The number of entries contributed by these elements are no more than $\frac{2}{\epsilon}$. Moreover, all members of the last bucket might contribute an entry each to $\mathcal{D}$. There are no more than $\frac{1}{\epsilon}$ such entries. The remaining entries in $\mathcal{D}$ have elements with $p_e < \frac{\epsilon}{2}$ which were inserted before the current bucket and survived the last deletion phase as well. We will show that there are fewer than $\frac{4}{\epsilon}$ such entries. This would prove the bound claimed in the lemma.

Let $B = b_{current}$ be the current bucket id. For $i = 2, \ldots, B$, let d_i denote the number of entries in $\mathcal{D}$ with $\Delta = B - i$ and $p_e < \frac{\epsilon}{2}$. Consider an element e that contributes to d_i. The arrival of remaining elements in buckets $B - i + 1$ through B can be looked upon as a sequence of Poisson trials with probability p_e. Let X denote the number of successful trials, i.e., the number of remaining occurrences of element e. Since there are at most $w(i-1)$ trials, we get $\mathbf{E}[X] \leq p_e w(i-1) \leq \frac{i-1}{2}$. For e to contribute to d_i, we require that $X \geq i - 1$. Chernoff bound techniques (See Theorem 4.1 in [MR95]) yield the inequality $Pr[X > (1+\delta)\mathbf{E}[X]] \geq [\frac{e^\delta}{(1+\delta)^{1+\delta}}]^{\mathbf{E}[X]}$ for any $\delta > 0$. If we write $(1+\delta)\mathbf{E}[X] = i - 1$, we get $\delta \geq 1$. Therefore, $Pr[X \geq i-1] \leq (e/4)^{\mathbf{E}[X]} \leq (e/4)^{i/2}$

Thus $\mathbf{E}[d_i] \leq \frac{1}{\epsilon}(\frac{e}{4})^{i/2}$. It follows that $\mathbf{E}[\sum_{i=2}^{B} d_i] \leq \sum_{i=2}^{B} \mathbf{E}[d_i] \leq \frac{1}{\epsilon}\sum_{i=2}^{B}(\frac{e}{4})^{i/2} \leq \frac{e/4}{\epsilon(1-\sqrt{e/4})} < \frac{4}{\epsilon}$. □

The theorem is true even if the positions of the high frequency elements are chosen by an adversary; only the low frequency elements are required to be drawn from some fixed distribution.