Frequent Itemset Mining over Data Streams

Gurmeet Singh Manku

1 Problem Definition

We study the problem of computing *frequent elements* in a data-stream. Given support threshold $s \in [0, 1]$, an element is said to be frequent if it occurs more than sN times, where N denotes the current length of the stream. If we maintain a list of counters of the form ⟨element, count⟩, one counter per unique element encountered, we need N counters in the worst-case. Many distributions are heavy-tailed in practice, so we would need far fewer than N counters. However, the number would still exceed $1/s$, which is the maximum possible number of frequent elements. If we insist on identifying *exact* frequency counts, then $\Omega(N)$ space is necessary. This observation motivates the definition of *ϵ-approximate frequency counts* : given support threshold $s \in [0, 1]$ and an error parameter $\epsilon \in (0, s)$, the goal is to produce a list of elements along with their estimated frequencies, such that three properties are satisfied:

I. Estimated frequencies are less than the true frequencies by at most ϵN.
II. All elements whose true frequency exceeds sN are output.
III. No element whose true frequency is less than $(s - \epsilon)N$ is output.

Example Imagine a statistician who wishes to identify elements whose frequency is at least 0.1 % of the entire stream seen so far. Then the support threshold is $s = 0.1$ %. The statistician is free to set $\epsilon \in (0, s)$ to whatever she feels is a comfortable margin of error. Let us assume she chooses $\epsilon = 0.01$ % (one-tenth of s). As per Property I, estimated frequencies are less than their true frequencies by at most 0.01 %. As per Property II, all elements with frequency exceeding $s = 0.1$ % will be

G.S. Manku (✉)
Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA, USA
e-mail: manku@google.com

M. Garofalakis et al. (eds.), *Data Stream Management*,
Data-Centric Systems and Applications, DOI 10.1007/978-3-540-28608-0_10

output; there are *no false negatives*. As per Property III, 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. On the whole, the approximation has two aspects: high frequency false positives, and small errors in individual frequencies. Both kinds of errors are tolerable in real-world applications.

2 One-Pass Algorithms

We present three algorithms for computing ϵ-approximate frequency counts. To avoid floors and ceilings, we will assume that $1/\epsilon$ is an integer (if not, we can scale down ϵ to 2^{-r} where r is an integer satisfying $2^{-r} < \epsilon < 2^{-r+1}$). The data structure for all three algorithms is a list of counters of the form $\langle$element, count$\rangle$, initially empty. At any time, ϵ-approximate frequency counts can be retrieved by identifying those elements whose associated count exceeds $(s - \epsilon)N$. The algorithms differ in terms of the rules employed for creating, incrementing, decrementing and deleting counters.

MISRA–GRIES ALGORITHM ([7]). Let e denote a newly-arrived element. If a counter for e already exists, it is incremented. Otherwise, if there already exist $1/\epsilon$ counters, we *repeatedly diminish all counters* by 1 until some counter drops to zero. We then delete all counters with count zero, and create a new counter of the form $\langle e, 1\rangle$.

LOSSY COUNTING ([6]). Let e denote a newly-arrived element. If a counter for e already exists, it is incremented. Otherwise, we create a new counter of the form $\langle e, 1\rangle$. Whenever N, the current size of the stream, equals i/ϵ for some integer i, all counters are decrement by one—we discard any counter that drops to zero.

STICKY SAMPLING ([6]). The algorithm is randomized, with δ denoting the probability of failure. The algorithm maintains r, the sampling rate, which varies over the lifetime of the stream. Initially, $r = 1$. Let e denote the newly-arrived element. If a counter for e exists, it is incremented. Otherwise, we toss a coin with probability of success r. If the coin toss succeeds, we create an entry of the form $\langle e, 1\rangle$; otherwise, we ignore e.

The sampling rate r varies 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 $r = 1/2$, the next $4t$ elements are sampled at rate $r = 1/4$, and so on. Whenever the sampling rate changes, we update existing counters as follows: For each counter, we repeatedly toss an unbiased coin until the coin toss is successful, decrementing the counter for every unsuccessful outcome; if the counter drops to zero during this process, we delete the counter. Effectively, the new list of counters is identical to exactly the list that would have emerged, had we been sampling with the new rate from the very beginning.

Theorem 1 MISRA–GRIES ALGORITHM *allows retrieval of ϵ-approximate frequency counts using at most $\frac{1}{\epsilon}$ counters.*

Proof Consider a fixed element e. Whenever a counter corresponding to e is diminished by 1, $1/\epsilon - 1$ other counters are also diminished. Clearly, when N elements have been seen, a counter for e could not have been diminished by more than ϵN. □

Theorem 2 LOSSY COUNTING *allows retrieval of ϵ-approximate frequency counts using at most $\frac{1}{\epsilon}\log(\epsilon N)$ counters.*

Proof Imagine splitting the stream into buckets of size $w = 1/\epsilon$ each. Let $N = Bw$, where B denotes the total number of buckets that we have seen. For each $i \in [1, B]$, let d_i denote the number of counters which were created when bucket $B - i + 1$ was active, i.e., the length of the stream was in the range $[(B - i)w + 1, (B - i + 1)w]$. The element corresponding to such a counter must occur at least i times in buckets $B - i + 1$ through B; otherwise, the counter would have been deleted. Since the size of each bucket is w, we get the following constraints:

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

We prove the following set of inequalities by induction:

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

The base case ($j = 1$) follows from (1) directly. Assume that (2) is true for $j = 1, 2, \ldots, p - 1$. We will show that it is true for $j = p$ as well. Adding $p - 1$ inequalities of type (2) (one inequality each for i varying from 1 to $p - 1$) to an inequality of type (1) (with $j = p$) yields

$$\begin{aligned}\sum_{i=1}^{p} id_i + \sum_{i=1}^{1} d_i + \sum_{i=1}^{2} d_i + \cdots + \sum_{i=1}^{p-1} d_i \\ \leq pw + \sum_{i=1}^{1} \frac{w}{i} + \sum_{i=1}^{2} \frac{w}{i} + \cdots + \sum_{i=1}^{p-1} \frac{w}{i}.\end{aligned}$$

Upon rearrangement, we get $p \sum_{i=1}^{p} d_i \leq pw + \sum_{i=1}^{p-1} \frac{(p-i)w}{i}$, which readily simplifies to (2) for $j = p$. This completes the induction step. The maximum number of counters is $\sum_{i=1}^{B} d_i \leq \sum_{i=1}^{B} \frac{w}{i} \leq \frac{1}{\epsilon} \log B = \frac{1}{\epsilon}\log(\epsilon N)$. □

Theorem 3 STICKY SAMPLING *computes ϵ-approximate frequency counts, with probability at least $1 - \delta$, using at most $\frac{2}{\epsilon}\log(s^{-1}\delta^{-1})$ counters in expectation.*

Proof The expected number of counters is $2t = \frac{2}{\epsilon}\log(s^{-1}\delta^{-1})$. When $r \leq 1/2$, then $N = rt + rt'$, for some $t' \in [1, t)$. It follows that $\frac{1}{r} \geq \frac{t}{N}$. Any error in the estimated

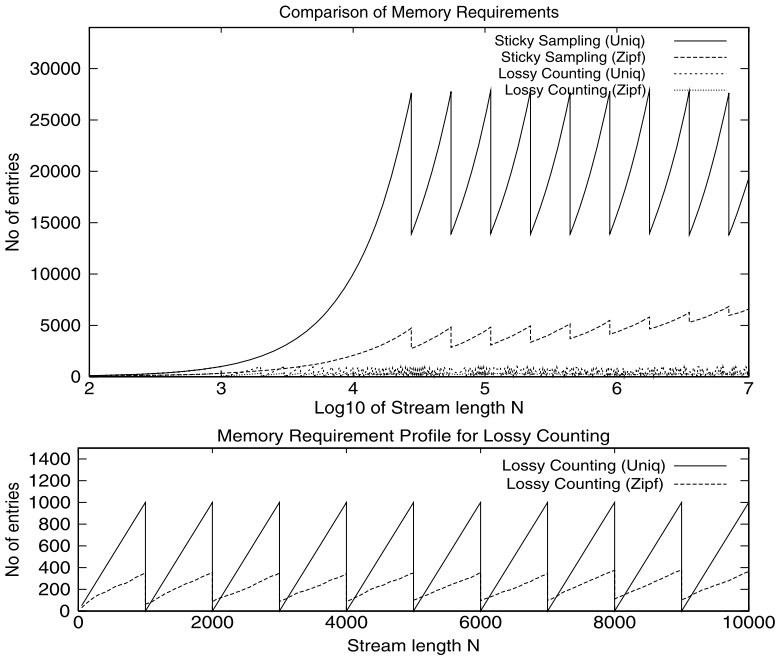

Fig. 1 Number of counters for support threshold $s = 1\ \%$, error parameter $\epsilon = 0.1\ \%$, and probability of failure $\delta = 10^{-4}$. Zipf denotes a Zipfian distribution with parameter 1.25. Uniq denotes a stream with no duplicates. The *bottom figure* magnifies a section of the barely-visible lines in the graph above

frequency of an element e corresponds to a sequence of unsuccessful coin tosses during the first few occurrences of e. The probability that this error exceeds ϵN is at most $(1 - \frac{1}{r})^{\epsilon N} \leq (1 - \frac{t}{N})^{-\epsilon N} \leq e^{-\epsilon t}$.

There are at most $1/s$ elements whose true frequency exceeds sN. The probability that the estimate frequency of *any* of them is deficient by ϵN, is at most $e^{-\epsilon t}/s$. Since $t \geq \frac{1}{\epsilon} \log(s^{-1}\delta^{-1})$, this probability is at most δ. □

Each of the algorithms has its strengths that make it useful in different contexts. The MISRA–GRIES ALGORITHM has optimal worst-case space complexity and $O(1)$ amortized cost of update per element. The update cost has been improved to $O(1)$ worst-case by Karp et al. [5]. STICKY SAMPLING is useful for identifying ϵ-approximate frequent counts over sliding windows (see Arasu and Manku [2]). LOSSY COUNTING is useful when the input stream has duplicates and when the input distribution is heavy-tailed, as borne out by Fig. 1. The kinks in the curve for STICKY SAMPLING correspond to re-sampling. They are $\log_{10} 2$ units apart on the X-axis. The kinks for LOSSY COUNTING correspond to $N = i/\epsilon$ (when deletions occur). 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 sur-

vive. For skewed distributions, both algorithms require much less space than their worst-case bounds in Theorems 2 and 3. LOSSY COUNTING is superior to MISRA–GRIES ALGORITHM for skewed data. For example, with $\epsilon = 0.01$ %, roughly 2000 entries suffice, which is only 20 % of $\frac{1}{\epsilon}$.

3 Frequent Itemset Mining

Let $\mathcal{I}$ denote the universe of all *items*. Consider a stream of *transactions*, where each transaction is a subset of $\mathcal{I}$. An itemset $X \subseteq \mathcal{I}$ is said to have support s if X is a subset of at least sN transactions, where N denotes the length of the stream, i.e., the number of transactions seen so far. The frequent itemsets problem seeks to identify all itemsets whose support exceeds a user-specified *support threshold* s.

Frequent itemsets are useful for identifying *association rules* (see Agrawal and Srikant [1] for a seminal paper that popularized the problem). Considerable work in frequent itemsets has focused on devising data structures for compactly representing frequent itemsets, and showing how the data structure can be constructed in a few passes over a large disk-resident dataset. The best-known algorithm take two passes.

Identification of frequent itemsets over data streams is useful in a data-warehousing environment where bulk updates occur at regular intervals of time, e.g., daily, weekly or monthly. *Summary data structures* that store aggregates like frequent itemsets should be maintained *incrementally* because a complete rescan of the entire warehouse-database per bulk-update is prohibitively costly. The summary data structure should be significantly smaller than the warehouse-database but need not fit in main memory.

In a data stream scenario, where only one pass is possible, we relax the problem definition to compute *ϵ-approximate frequent itemsets:* Given support threshold $s \in (0, 1)$ and error parameter $\epsilon \in (0, s)$, the goal is to produce itemsets, along with their estimated frequencies, satisfying three properties:

I. Estimated frequencies are less than the true frequencies by at most ϵN.
II. All itemsets whose true frequency exceeds sN are output.
III. No itemset whose true frequency is less than $(s - \epsilon)N$ is output.

We now develop an algorithm based upon LOSSY COUNTING for tackling the *ϵ-approximate frequent itemsets problem*. We begin by describing some modifications to LOSSY COUNTING.

3.1 *Modifications to* LOSSY COUNTING

We divide the stream into buckets of size $1/\epsilon$ each. Buckets are numbered sequentially, starting with 1. We maintain counters of the form ⟨element, count, bucket_id⟩, where bucket_id denotes the ID of the bucket that was active when this counter was

created. At bucket boundaries, we check whether count+bucket_id $\leq \epsilon N$. If so, the counter is deleted. This is equivalent to our earlier approach of decrementing counters at bucket boundaries and deleting those counters that drop to zero. The maximum possible error in the estimated frequency of an element is given by its bucket_id (which might be much less than ϵN). Furthermore, the algorithm continues to be correct even if we do not check the counters at each and every bucket boundary. However, the longer we defer the checks, the larger the space requirements due to the presence of noise (low-frequency elements in recent buckets).

3.2 Frequent Itemsets Algorithm

The input to the algorithm is a stream of transactions. The user specifies two parameters, support threshold, s, and error parameter, ϵ. We denote the current length of the stream by N. We maintain a data structure $\mathcal{D}$ consisting of a set of entries of the form (set, f, Δ), where *set* is an itemset (subset of $\mathcal{I}$), f is an integer representing the estimated frequency of *set*, and Δ is the maximum possible error in f. Initially, $\mathcal{D}$ is empty.

The stream is divided into *buckets* consisting of $w = \lceil 1/\epsilon \rceil$ transactions each. Buckets are labeled with *bucket ids*, starting from 1. We denote the *current bucket id* by $b_{current}$. We do not process the stream transaction by transaction. Instead, we fill available main memory with as many transactions as possible, and then process the resulting *batch* of transactions together. 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 $\mathcal{D}$, we create a new entry $(set, f, b_{current} - \beta)$.

A 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 small values of β, $\mathcal{D}$ is polluted by noise (subsets whose overall frequency is very low, but which occur at least $\beta + 1$ times in the last β buckets).

Two design problems emerge: *What is an efficient representation of $\mathcal{D}$? What is an efficient algorithm to implement* UPDATE_SET *and* NEW_SET*?*

3.3 Data Structure and Algorithm Design

We have three modules: TRIE, BUFFER, and SETGEN. TRIE is an efficient implementation of $\mathcal{D}$. BUFFER repeatedly reads in batches of transactions into available main memory and carries out some pre-processing. SETGEN then operates on the current batch of transactions in BUFFER. It enumerates subsets of these transactions along with their frequencies, limiting the enumeration using some pruning rules. Effectively, SETGEN implements the UPDATE_SET and NEW_SET operations to update TRIE. The challenge, it turns out, lies in designing a space-efficient TRIE and a time-efficient SETGEN.

TRIE. This module maintains the data structure $\mathcal{D}$ outlined in Sect. 3.2. 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.

To make the TRIE compact, we maintain *an array of entries* of the form $\langle item\_id, f, \Delta, level \rangle$ *corresponding to the pre-order traversal of the underlying trees*. This is equivalent to a *lexicographic ordering of all the subsets encoded by the trees*. There are no pointers from any node to its children or its siblings. The *level*'s compactly encode the underlying tree structure. Such a representation suffices because tries are always scanned sequentially, as we show later.

Tries are used by several Association Rules algorithms, hash tries [1] being a popular choice. Popular implementations of tries require pointers and variable-sized memory segments (because the number of children of a node changes over time). Our TRIE is quite different.

BUFFER. This module repeatedly fills available main memory with a batch of transactions. Each transactions is a set of item-id's. Transactions are laid out one after the other in a big array. A bitmap is used to remember transaction boundaries. A bit 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.

SETGEN. This module generates subsets of item-id's along with their frequencies in the current batch of transactions in lexicographic order. It is important that not all possible subsets 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 [1]. We describe an efficient implementation of SETGEN in greater detail later.

Overall Algorithm

BUFFER repeatedly fills available main memory with a batch of transactions, and sorts them. SETGEN operates on BUFFER to generate 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.

3.4 Efficient Implementations

In this section, we outline important design decisions that contribute to an efficient implementation.

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 all items. For example, if $|\mathcal{I}| = 10^5$, an array of size 0.4 MB suffices. If exact frequency counts are available, BUFFER first prunes away those item-id's whose frequency is less than ϵN, and then sorts the transactions, where 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. Since our compact TRIE also stores its constituent subsets in their lexicographic order, the two modules: SETGEN and TRIE work hand in hand.

We maintain TRIE not as one huge array, but as a collection of fairly large-sized chunks of memory. Instead of modifying the original trie in place, we create a new TRIE afresh. Chunks belonging to 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 belonging to the old 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 Sect. 3.2, we prune nodes in the trie more aggressively by setting the threshold for deletion to sN instead of $b_{current} \approx \epsilon N$. This is because the lower frequency nodes do not contribute to the final output.

SETGEN. This module is the bottleneck in terms of time. Therefore, it merits careful design and run-time optimizations. 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.

3.5 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 would 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. This means that the 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 tiny values of ϵ. This is a big win.

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

On the whole, the algorithm has two unique features: there is *no candidate generation phase*, which is typical of Apriori-style algorithms. Further, 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.

Experimental evaluation over a variety of datasets is available in [6].

4 Applications and Related Work

Frequency counts and frequent itemsets arise in a variety of applications. We describe two of these below.

4.1 Iceberg Queries

The idea behind Iceberg Queries[4] 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 frequent itemset algorithm developed in Sect. 3 runs in only one pass, and out-performs the highly-tuned algorithm in [4] that uses repeated hashing over multiple passes.

4.2 Network Flow Identification

Measurement and monitoring of network traffic is required for management of complex Internet backbones. In this context, identifying flows in network traffic is an important problem. A flow is defined as a sequence of transport layer (TCP/UDP) packets that share the same source+destination addresses. Estan and Verghese [3] 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 by Fang et al. [4] for Iceberg Queries.

4.3 Algorithms for Sliding Windows

Algorithms for computing approximate frequency counts over sliding windows have been developed by Arasu and Manku [2]. In a *fixed-size* sliding window, the size of the window remains unchanged. In a *variable-sized* sliding window, at each time-step, an adversary can either insert a new element, or delete the oldest element in the window. When the size of the window is W, the space-bounds for a randomized algorithm (based upon STICKY SAMPLING) are $O(\frac{1}{\epsilon}\log\frac{1}{\epsilon\delta})$ and $O(\frac{1}{\epsilon}\log\frac{1}{\epsilon\delta}\log\epsilon W)$ for fixed-size and variable-size windows respectively. The corresponding bounds for a deterministic algorithm (based upon MISRA–GRIES ALGORITHM) are $O(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon})$ and $O(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon}\log\epsilon W)$, respectively. It would be interesting to see if any of these algorithms can be adapted to compute ϵ-approximate frequent itemsets in a data stream.

References

1. R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in *Proc. of 20th Intl. Conf. on Very Large Data Bases* (1994), pp. 487–499
2. A. Arasu, G.S. Manku, Approximate counts and quantiles over sliding windows, in *Proc. ACM Symposium on Principles of Database Systems* (2004)
3. C. Estan, G. Varghese, New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice. ACM Trans. Comput. Syst. **21**(3), 270–313 (2003)
4. M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, J. Ullman, Computing iceberg queries efficiently, in *Proc. of 24th Intl. Conf. on Very Large Data Bases* (1998), pp. 299–310
5. R.M. Karp, C.H. Papadimitriou, S. Shenker, A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst. **28**, 51–55 (2003)
6. G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in *Proc. 28th VLDB* (2002), pp. 356–357
7. J. Misra, D. Gries, Finding repeated elements. Sci. Comput. Program. **2**(2), 143–152 (1982)