Consistent Subset Sampling*

Konstantin Kutzkov and Rasmus Pagh

IT University of Copenhagen, Denmark

Abstract. Consistent sampling is a technique for specifying, in small space, a subset S of a potentially large universe U such that the elements in S satisfy a suitably chosen sampling condition. Given a subset $\mathcal{I} \subseteq U$ it should be possible to quickly compute $\mathcal{I} \cap S$, i.e., the elements in $\mathcal{I}$ satisfying the sampling condition. Consistent sampling has important applications in similarity estimation, and estimation of the number of distinct items in a data stream.

In this paper we generalize consistent sampling to the setting where we are interested in sampling size-k subsets occurring in some set in a collection of sets of bounded size b, where k is a small integer. This can be done by applying standard consistent sampling to the k-subsets of each set, but that approach requires time $\Theta(b^k)$. Using a carefully designed hash function, for a given sampling probability $p \in (0, 1]$, we show how to improve the time complexity to $\Theta(b^{\lceil k/2 \rceil} \log \log b + pb^k)$ in expectation, while maintaining strong concentration bounds for the sample. The space usage of our method is $\Theta(b^{\lceil k/4 \rceil})$.

We demonstrate the utility of our technique by applying it to several well-studied data mining problems. We show how to efficiently estimate the number of frequent k-itemsets in a stream of transactions and the number of bipartite cliques in a graph given as incidence stream. Further, building upon a recent work by Campagna et al., we show that our approach can be applied to frequent itemset mining in a parallel or distributed setting. We also present applications in graph stream mining.

1 Introduction

Consistent sampling is an important technique for constructing randomized sketches (or "summaries") of large data sets. The basic idea is to decide whether to sample an element x depending on whether a certain sampling condition is satisfied. Usually, consistent sampling is implemented using suitably defined hash functions and x is sampled if its hash value $h(x)$ is below some threshold. If x is encountered several times, it is therefore either *never* sampled or *always* sampled. The set of items to sample is described by the definition of the hash function, which is typically small.

Consistent sampling comes in two basic variations: In one variation (sometimes referred to as *subsampling*) there is a fixed sampling probability $p \in (0, 1)$,

* This work is supported by the Danish National Research Foundation under the Sapere Aude program.

and elements in a set must be sampled with this probability. In the alternative model the sample size is fixed, and the sampling probability must be scaled to achieve the desired sample size.

Depending on the strength of the hash function used, the sample will exhibit many of the properties of a random sample (see e.g. [5, 18]). One of the most famous applications of consistent sampling [6] is estimating the *Jaccard similarity* of two sets by the similarity of consistent samples, using the same hash function. Another well-known application is reducing the number of distinct items considered to $\Theta(1/\varepsilon^2)$ in order to make an $(1 \pm \varepsilon)$-approximation of the total number of distinct items (see [21] for the state-of-the-art result).

In this paper we consider consistent sampling of certain *implicitly defined* sets. That is, we sample from a set much larger than the size of the explicitly given database. Our main focus is on streams of sets, where we want to sample subsets of the sets in the stream.

We demonstrate the usability of our technique by designing new algorithms for several well-studied counting problems in the streaming setting. We present the first nontrivial algorithm for the problem of estimating the number of frequent k-itemsets with rigorously understood complexity and error guarantee and also give a new algorithm for counting bipartite cliques in a graph given as an incidence stream. Also, using a technique presented in [9], we show that our approach can be easily parallelized and applied to frequent itemset mining algorithms based on hashing [10, 11].

2 Preliminaries

Notation. Let $\mathcal{C} = T_1, .., T_m$ be a collection of m subsets of a ground set $\mathcal{I}$, $T_j \subseteq \mathcal{I}$, where $\mathcal{I} = \{1, \ldots, n\}$ is a set of *elements*. The sets T_j each contain at most b elements, i.e., $|T_j| \leq b$, and in the following are called *b-sets*. Let further $S \subseteq \mathcal{I}$ be a given subset. If $|S| = k$ we call S a *k-subset*. We assume that the b-sets are explicitly given as input while a k-subset can be any subset of $\mathcal{I}$ of cardinality k. In particular, a b-set with b elements contains $\binom{b}{k}$ distinct k-subsets for $k \leq b$. The *frequency* of a given k-subset is the number of b-sets containing it.

In order to simplify the presentation, we assume a lexicographic order on the elements in $\mathcal{I}$ and a unique representation of subsets as ordered vectors of elements. However, we will continue to use standard set operators to express computations on these vectors. In our algorithm we will consider only lexicographically ordered k-subsets. For two subsets I_1, I_2 we write $I_1 < I_2$ iff $i_1 < i_2$ $\forall i_1 \in I_1, i_2 \in I_2$.

The set of k-subsets of $\mathcal{I}$ is written as $\mathcal{I}^k$ and similarly, for a given b-set T_j, we write T_j^k for the family of k-subsets occurring in T_j. A family of k-subsets $\mathcal{S} \subset \mathcal{I}^k$ is called a *consistent sample* for a given sampling condition P if for each b-set T_i the set $\mathcal{S} \cap T_i^k$ is sampled, i.e., all elements satisfying the sampling condition P that occur in T_i are sampled. The sampling condition P will be defined later.

Let $[q]$ denote the set $\{0, \ldots, q-1\}$ for $q \in \mathbb{N}$. A hash function $h : \mathcal{E} \to [q]$ is t-wise independent iff $\mathbf{Pr}[h(e_1) = c_1 \wedge h(e_2) = c_2 \wedge \cdots \wedge h(e_t) = c_t] = q^{-t}$ for distinct elements $e_i \in \mathcal{E}$, $1 \leq i \leq t$, and $c_i \in [q]$. We denote by $p = 1/q$ the sampling probability we use in our algorithm. Throughout the paper we will often exchange p and $1/q$.

We assume the standard computation model and further we assume that one element of $\mathcal{I}$ can be written in one machine word.

Example. In order to simplify further reading let us consider a concrete data mining problem. Let $\mathcal{T}$ be a stream of m transactions $T_1, T_2, \ldots, T_m$ each of size b. Each such transaction is a subset of the ground set of items $\mathcal{I}$. We consider the problem of finding the set of frequent k-itemsets, i.e., subsets of k items occurring in at least t transactions for a user-defined $t \leq m$. As a concrete example consider a supermarket. The set of items are all offered goods and transactions are customers baskets. Frequent 2-itemsets will provide knowledge about goods that are frequently bought together.

The problem can be phrased in terms of the above described abstraction by associating transactions with b-sets and k-itemsets with k-subsets. Assume we want to sample 2-itemsets. A consistent sample can be described as follows: for a hash function $h : \mathcal{I} \to [q]$ we define S to be the set of all 2-subsets (i, j) such that $h(i) + h(j) = 0 \bmod q$. In each b-set we can then generate all $\binom{b}{2}$ 2-subsets and check which of them satisfy the so defined sampling condition. For a suitably defined hash function, one can show that resulting sample is "random enough" and can provide important information about the data, for example, we can use it to estimate the number of 2-itemsets occurring above a certain number of times.

3 Our contribution

3.1 Time-space trade-offs revisited.

Streaming algorithms have traditionally been mainly concerned with space usage. An algorithm with a superior space usage, for example polylogarithmic, has been considered superior to an algorithm using more space but less computation time. We would like to challenge this view, especially for time complexities that are in the polynomial (rather than polylogarithmic) range. The purpose of a scalable algorithm is to allow the largest possible problem sizes to be handled (in terms of relevant problem parameters). A streaming algorithm may fail either because the processing time is too high, or because it uses more space than what is available. Typically, streaming algorithms should work in space that is small enough to fit in fast cache memory, but there is no real advantage to using only 10% of the cache. Looking at high-end processors over the last 20 years, see for example `http://en.wikipedia.org/wiki/Comparison_of_Intel_Processors`, reveals that the largest system cache capacity and the number of instructions per second have developed rather similarly (with the doubling time for space being about 25% larger than the doubling time for number

of instructions). Assuming that this trend continues, a future processor with x times more processing power will have about $x^{0.8}$ times larger cache. So informally, whenever we have $S = o(T^{0.8})$ for an algorithm using time T and space S the space will not be the asymptotic bottleneck.

3.2 Main Result.

In this paper we consider consistent sampling of certain implicitly defined sets, focusing on size-k subsets in a collection of b-sets. The sampling is consistent in the sense that each occurrence of a k-subset satisfying the sampling condition is recorded in the sample.

Theorem 1 *For each integer $k \geq 2$ there is an algorithm computing a consistent, pairwise independent sample of k-subsets from a given b-set in expected time $O(b^{\lceil k/2 \rceil} \log\log b + pb^k)$ and space $O(b^{\lceil k/4 \rceil})$ for a given sampling probability p, such that $1/p = O(b^k)$ and p can be described in one word. An element of the sample is specified in $O(k)$ words.*

Note that for the space complexity we do not consider the size of the computed sample. We will do this when presenting concrete applications of our approach.

For low sampling rates our method, which is based on hash collisions among $k/2$-subsets, is a quadratic improvement in running time compared to the naïve method that iterates through all k-subsets in a given b-set. In addition, we obtain a quadratic improvement in space usage compared to the direct application of the hashing idea. Storing a single $2k$-wise independent hash function suffices to specify a sample, where every pair of k-subsets are sampled independently.

An important consequence of our consistent sampling algorithm is that it can be applied to b-sets revealed one at a time, thus it is well-suited for streaming problems.

4 Our approach

4.1 Intuition

A naïve consistent sampling approach works as follows: Define a pairwise independent hash function $h : \mathcal{I}^k \to [q]$, for a given b-set T generate all $\binom{b}{k}$ k-subsets $I_k \in T^k$ and sample a subset I_k iff $h(I_k) = 0$. Clearly, to decide which I_k are sampled the running time is $O(b^k)$ and the space is $O(b)$ since the space needed for the description of the hash function for reasonably small sampling probability p is negligible. A natural question is whether a better time complexity is possible.

Our idea is instead of explicitly considering all k-subsets occurring in a given b-set, to hash all elements to a value in $[q]$, $q = \lceil 1/p \rceil$ for a given sampling probability p. We show that the sampling of k-subsets is *pairwise independent*

and for many concrete applications this is sufficient to consider the resulting sample "random enough".

The construction of the hash function is at the heart of our algorithm and allows us to exploit several tricks in order to improve the running time. Let us for simplicity assume k is even. Then we sample a given k-subset if the sum (mod q) of the hash values of its first $k/2$ elements equals the sum of the hash values of its last $k/2$ elements modulo q. The simple idea is to sort all $k/2$-subsets according to hash value and then look for collisions. Using a technique similar to the one of Schroeppel and Shamir for the knapsack problem [23], we show how by a clever use of priority queues one can design an algorithm with much better time complexity than the naïve method and quadratic improvement in the space complexity of the sorting approach.

4.2 The hash function

Now we explain how we sample a given k-subset. Let $h : \mathcal{I} \to [q]$ be a $2k$-wise independent hash function, $k \geq 2$. It is well-known, see for example [12], that such a function can be described in $O(k)$ words for a reasonable sampling probability, i.e., a sampling probability that can be described in one machine word.

We take a k-subset $(a_1, \ldots, a_{\lfloor k/2 \rfloor}, a_{\lfloor k/2 \rfloor+1}, \ldots, a_k)$ in the sample iff $(h(a_1)+\cdots+h(a_{\lfloor k/2 \rfloor})) \bmod q = (h(a_{\lfloor k/2 \rfloor+1})+\cdots+h(a_k)) \bmod q$. Note that we have assumed a unique representation of subsets as sorted vectors and thus the sampling condition is uniquely defined.

For a given k-subset $I = (a_i, a_{i+1} \ldots, \ldots, a_{i+k-1})$, $i \in \mathcal{I}$, we denote by $h(a_i, a_{i+1} \ldots, \ldots, a_{i+k-1}))$ the value $(h(a_1) + h(a_{i+1}) + \cdots + h(a_{i+k-1})) \bmod q$. We define the random variable X_I to indicate whether a given k-subset $I = (a_1, \ldots, a_k)$ will be considered for sampling:

$$X_I = \begin{cases} 1, & \text{if } h(a_1 \ldots a_{\lfloor k/2 \rfloor}) = h(a_{\lfloor k/2 \rfloor+1} \ldots a_k), \\ 0, & \text{otherwise} \end{cases}$$

The following lemmas allow us to assume that from our sample we can obtain a reliable estimate with high probability:

Lemma 1 *Let I be a t-subset with $t \leq k$. Then for a given $r \in [q]$, $\mathbf{Pr}[h(I) = r] = 1/q$.*

Proof. Since h is $2k$-wise independent and uniform each of the $t \leq k$ distinct elements is hashed to a value between 0 and $q-1$ uniformly and independently from the remaining $t-1$ elements. Thus, the sum (mod q) of the hash values of I's t elements is equal with probability $1/q$ to r. □

Lemma 2 *For a given k-subset I, $\mathbf{Pr}[X_I = 1] = 1/q$.*

Proof. Let $I = I_l \cup I_r$ with $|I_l| = \lfloor k/2 \rfloor$ and $|I_r| = \lceil k/2 \rceil$. The hash value of each subset is uniquely defined, h is $2k$-wise independent, and together with the result of Lemma 1 we have $\mathbf{Pr}[h(I_l) = h(I_r) = r] = 1/q^2$ for a particular $r \in [q]$. Thus, we have $\mathbf{Pr}[h(I_l) = h(I_r) = 0 \vee \cdots \vee h(I_l) = h(I_r) = q-1] = \sum_{i=0}^{q-1} \mathbf{Pr}[h(I_l) = h(I_r) = i] = 1/q$. □

Lemma 3 *Let I_1 and I_2 be two distinct k-subsets. Then the random variables X_{I_1} and X_{I_2} are independent.*

Proof. We show that $\mathbf{Pr}[X_{I_1} = 1 \wedge X_{I_2} = 1] = \mathbf{Pr}[X_{I_1} = 1]\mathbf{Pr}[X_{I_2} = 1] = 1/q^2$ for arbitrarily chosen k-subsets I_1, I_2. This will imply pairwise independence on the events that two given k-subsets are sampled since for a given k-subset I, $\mathbf{Pr}[X_I = 1] = 1/q$ as shown in Lemma 1.

Let $I_1 = I_{1l} \cup I_{1r}$ and $I_2 = I_{2l} \cup I_{2r}$ with $|I_{il}| = \lfloor k/2 \rfloor$ and $|I_{ir}| = \lceil k/2 \rceil$. Let us assume without loss of generality that $h(I_{1l}) = r_1$ and $h(I_{2l}) = r_2$ for some $r_i \in [q]$. As shown in the previous lemmas for fixed r_1 and r_2, $\mathbf{Pr}[h(I_{1r}) = r_1] = \mathbf{Pr}[h(I_{2r}) = r_2] = 1/q$. Since h is $2k$-wise independent, all elements in $I_{1l} \cup I_{1r} \cup I_{2l} \cup I_{2r}$ are hashed independently of each other. Thus, it is easy to see that the event we hash I_{2r} to r_2 is independent from the event that we have hashed I_{1r} to r_1, thus the statement follows. □

The above lemmas imply that our sampling will be uniform and pairwise independent.

4.3 The algorithm

A pseudocode description of our algorithm is given in Figure 1. We explain how the algorithm works with a simple example. Assume we want to sample 8-subsets from a b-set $(a_1, \ldots, a_b)$ with $b > 8$. We want to find all 8-subsets $(a_1, \ldots, a_8)$ for which it holds $h(a_1, \ldots, a_4) = h(a_5, \ldots, a_8)$. As discussed, we assume a lexicographic order on the elements in $\mathcal{I}$ and we further assume b-sets are sorted according to this total order. The assumption can be removed by preprocessing and sorting the input. Since $\mathcal{I}$ is discrete, one can assume that each b-set can be sorted by the Han-Thorup algorithm in $O(b\sqrt{\log\log b})$ expected time and space $O(b)$ [16] (for the general case of sampling k-subsets even for $k = 2$ this will not dominate the complexity claimed in Theorem 1). In the following we assume the elements in each b-set are sorted. Recall we have assumed a total order on subsets, and all subsets we consider are sorted according to this total order. We will also consider only sorted subsets for sampling.

We simulate a sorting algorithm in order to find all 4-subsets with equal hash values. Let the set of 2-subsets be H. First, in ConsistentSubsetSampling we generate all $\binom{b}{2}$ 2-subsets and sort them according to their hash value in a circular list L guaranteeing access in expected constant time. We also build a priority queue P containing $\binom{b}{2}$ 4-subsets as follows: For each 2-subset $(a_i, a_j) \in H$ we find the 2-subset $(a_k, a_\ell) \in L$ such that $h(a_i, a_j, a_k, a_\ell)$ is minimized and keep track of the position of (a_k, a_ℓ) in L. Then we successively output all 4-subsets

ConsistentSubsetSampling

Input: b-set $T \subset \mathcal{I}$, a $2k$-wise independent $h : \mathcal{I} \to [q]$
Let $H = T^{k/4}$ be the $k/4$-subsets occurring in T.
Sort all $k/4$-subsets from H in a circular list L according to hash value.
Build a priority queue P with $k/2$-subsets $I = I_H \cup I_L$ according to hash value, for $I_H \in H$, $I_L \in L$.
for $i \in [q]$ **do**
$T_i^{k/2} =$ OutputNext(P, L, i)
Generate all k-subsets from $T_i^{k/2}$ satisfying the sampling condition (and consisting of k different elements).

OutputNext

Input: a circular list L, a priority queue P of $k/2$-subsets $I = (I_H \cup I_L)$ compared by hash value $h(I)$, $i \in \mathbb{N}$
while there is $k/2$-subset with a hash value i **do**
Output the next $k/2$-subset $I = (I_H \cup I_L)$ from P.
if cnt$(I_H) <$ L.length **then**
Replace I by $I_H \cup I'_L$ in P where I'_L is the $k/4$-subset following I_L in L.
Update the hash value of $I_H \cup I'_L$ and restore the PQ invariant.
cnt(I_H)++.
else
Remove $I = (I_H \cup I_L)$ from P and restore the PQ invariant.

Fig. 1: A high-level pseudocode description of the algorithm. For simplicity we assume that k is a multiple of 4. The letter H stands for "head", these are the $k/4$-subsets that will constitute the first half of $k/2$-subsets in P. We will always update the second half with a $k/4$-subset from L.

sorted according to their hash value from the priority queue by calling OutputNext. For a given collection of 4-subsets with the same hash value we generate all valid 8-subsets, i.e., we find all combinations yielding lexicographically ordered 8-subsets. Note that the "head" 2-subsets from H are never changed while we only update the "tail" 2-subsets with new 2-subsets from L. During the process we also check whether all elements in the newly created 4-subsets are different.

In OutputNext we simulate a heapsort-like algorithm for 4-subsets. We do not keep explicitly all 4-subsets in P but at most $\binom{b}{2}$ 4-subsets at a time. Once we output a given 4-subset (a_i, a_j, a_k, a_ℓ) from P, we replace it with $(a_i, a_j, a'_k, a'_\ell)$ where (a'_k, a'_ℓ) is the 2-subset in L following (a_k, a_ℓ). We also keep track whether we have not already traversed L for each 2-subset in H. If this is the case, we remove the 4-subset (a_i, a_j, a_k, a_ℓ) from P and the number of recorder entries in P is decreased by 1. At the end we update P and maintain the priority queue invariant.

In the following lemmas we will prove the correctness of the algorithm for general k and will analyze its running time. This will yield our main Theorem 1.

Lemma 4 *For $k \geq 4$ with $k \bmod 4 = 0$* Consistent Subset Sampling *outputs the $k/2$-subsets from a given b-set in sorted order according to their hash value in expected time $O(b^{k/2} \log\log b)$ and space $O(b^{k/4})$.*

Proof. Let T be the given b-subset and $\mathcal{S} = I_1, \ldots, I_{\binom{b}{k/2}}$ be the $k/2$-subsets occurring in T sorted according to hash value. For correctness we first show that the following invariant holds: After the j smallest $k/2$-subsets have been output from P, P contains the $(j+1)$th smallest $k/2$-subset in S, ties resolved arbitrarily. For $j = 1$ the statement holds by construction. Assume now that it holds for some $j \geq 1$ and we output the jth smallest $k/2$-subset $I = I_H \cup I_L$ for $k/4$-subsets I_H and I_L. We then replace it by $I' = I_H \cup I'_L$ where I'_L is the $k/4$-subset in L following I_L, or, if L has been already traversed, remove I from P. If P contains the $(j+1)$th smallest $k/2$-subset, then the invariant holds. Otherwise, we show that it must be that the $(j+1)$th smallest $k/2$-subset is I'. Since L is sorted and we traverse it in increasing order, the $k/2$-subsets $I_H \cup I_L$ with a fixed head I_H that remain to be considered have all a bigger hash value than I. The same reasoning applies to all other $k/2$-subsets in P, and since no of them is the $(j+1)$-th smallest $k/2$-subset, the only possibility is that indeed I' is the $(j+1)$-th smallest $k/2$-subset.

In L we need to explicitly store $O(b^{k/4})$ subsets. Clearly, we can assume that we access the elements in L in constant time. The time and space complexity depend on how we implement the priority queue P. We observe that for a hash function range in $b^{O(1)}$ the keys on which we compare the 2-subsets are from a universe of size $b^{O(1)}$. Thus, we can implement P as a y-fast trie [24] in $O(b^{k/4})$ space supporting updates in $O(\log \log b)$ time. This yields the claimed bounds. □

Note however, that the number of $k/2$-subsets with the same hash value might be $\omega(b^{k/4})$. We next guarantee that the worst case space usage is $O(b^{k/4})$.

Lemma 5 *For a given $r \in [q]$, k mod $4 = 0$, and sampling probability $p \in (0, 1]$, we generate all k-subsets from a set of $k/2$-subsets with a hash value r that satisfy the sampling condition in expected time $O(p^2 b^k)$ and space $O(b^{k/4})$.*

Proof. We use the following implicit representation of $k/2$-subsets with the same hash value. For a given $k/4$-subset I_P the $k/4$-subsets I_L in L occurring in $k/2$-subsets $I_P \cup I_L$ with the same hash value are contained in a subsequence of L. Therefore, instead of explicitly storing all $k/2$-subsets, for each $k/4$-subset I_P we store two indices i and j indicating that $h(I_P \cup L[k]) = r$ for $i \leq k \leq j$. Clearly, this guarantees a space usage of $O(b^{k/4})$.

We expect $pb^{k/2}$ $k/2$-subsets to have hash value r, thus the number of k-subsets that will satisfy the sampling condition is $O(p^2 b^k)$. □

The above lemmas prove Theorem 1 for the case k mod $4 = 0$. One generalizes to arbitrary $k \geq 4$ in the following way:

For even k with k mod $4 = 2$, meaning that $k/2$ is odd, it is easy to see that we need a circular list with all $\lfloor k/4 \rfloor$-subsets but the priority queue will contain $\binom{b}{\lceil k/4 \rceil}$ pairs of $k/2$-subsets (which are concatenations of $\lceil k/4 \rceil$-subsets and $\lfloor k/4 \rfloor$-subsets). For odd k we want to sample all k-subsets for which the sum of the hash values of the first $\lfloor k/2 \rfloor$ elements equals the sum of the hash

values of the last $\lceil k/2 \rceil$ elements. We can run two copies of OutputNext in parallel, one will output the $\lceil k/2 \rceil$-subsets with hash value r and the other one the $\lfloor k/2 \rfloor$-subsets with hash value r for all $r \in [q]$. Then we can generate all k-subsets satisfying the sampling condition as outlined in Lemma 5 with the only difference that we will combine $\lceil k/2 \rceil$-subsets with $\lfloor k/2 \rfloor$-subsets output by each copy of OutputNext. Clearly, the space complexity is bounded by $O(b^{\lceil k/4 \rceil})$ and the expected running time is $O(b^{\lceil k/2 \rceil} + pb^k)$. This completes the proof of Theorem 1.

A time-space trade-off. A better space complexity can be achieved by increasing the running time. The following theorem generalizes our result.

Theorem 2 *For any $k \geq 2$ and $\ell \leq k/2$ we can compute a consistent, pairwise independent sample of k-subsets from a given b-set in expected time $O(b^{\lceil k/2+\ell \rceil} \log\log b) + pb^k)$ and space $O(b^{\lceil (k-2\ell)/4 \rceil} + b)$ for a given sampling probability p, such that $1/p = O(b^k)$ and p can be described in one word.*

Proof. We need space $O(b)$ to store the b-set. Assume that we iterate over 2ℓ-subsets $(a_1, \ldots, a_{2\ell})$, without storing them and their hash values. We assume that we have fixed ℓ elements among the first $\lfloor k/2 \rfloor$ elements, and ℓ elements among the last $\lceil k/2 \rceil$ ones. We compute the value $h^\ell = (h(a_1) + \cdots + h(a_\ell) - h(a_{\lfloor k/2 \rfloor+1}) - \cdots - h(a_{\lfloor k/2 \rfloor+\ell})) \bmod q$. We now want to determine all $(k-2\ell)$-subsets for which the sum of the hash values of the first $\lfloor k/2 \rfloor - \ell$ elements equals the sum of the hash values of the last $\lceil k/2 \rceil - \ell$ elements minus the value h^ℓ. Essentially, we can sort $(k-2\ell)$-subsets according to their hash value in the same way as before and the only difference is that we subtract h^ℓ from the hash value of the last $\lceil k/2 \rceil - \ell$ elements. Thus, we can use two priority queues, where in the second one we have subtracted h^ℓ from the hash value of each $(\lceil k/2 \rceil - \ell)$-subset, output the minima and look up for collisions. Disregarding the space for storing the b-set, the outlined modification requires time $O(b^{\lceil k/2+\ell \rceil} \log\log b)$ and space $O(b^{\lceil (k-2\ell)/4 \rceil})$ to process a given b-set. □

Discussion. Let us consider the scalability of our approach to larger values of b, assuming that the time is not dominated by iterating through the sample. If we are given a processor that is x times more powerful, this will allow us to increase the value of b by a factor $x^{1/\lceil k/2 \rceil}$. This will work because the space usage of our approach will only rise by a factor $\sqrt{x}$, and, as already discussed, we expect a factor $x^{0.8}$ more space to be available. An algorithm using space $b^{\lceil k/2 \rceil}$ would likely be space-bounded rather than time-bounded, and thus only be able to increase b by a factor of $x^{0.8/\lceil k/2 \rceil}$. At the other end of the spectrum an algorithm using time x^k and constant space would only be able to increase b by a factor $x^{1/k}$.

5 Applications of Consistent Subset Sampling

In this section we discuss several algorithmic applications of Consistent Subset Sampling for well-studied data mining problems. The reader is referred to the full version of the paper[1] for more details.

A fundamental problem in data mining is the problem of frequent itemsets mining in transactional data streams. For example, transactions correspond to market baskets and we want to detects sets of goods that are frequently bought together, see [15] for an overview. A low frequency threshold may lead to an explosion of the number of frequent itemsets, therefore a good prediction of their number is needed [14]. Known approaches for the problem are all based on some heuristics [2, 19] and the worst case running time is exponential. By associating transactions with b-sets, Consistent Subset Sampling naturally applies to the problem. As a result, we obtain the first algorithm with rigorously understood complexity and approximation guarantees. The next theorem is our main result:

Theorem 3 *Let $\mathcal{T}$ be a stream of m transactions of size at most b over a set of n items and f and z be the number of frequent and different k-itemsets, $k \geq 2$, in $\mathcal{T}$, respectively. For any $\alpha, \varepsilon, \delta > 0$ there exists a randomized algorithm running in expected time $O(mb^{\lceil k/2 \rceil} \log\log b + \frac{\log m \log \delta^{-1}}{\alpha\varepsilon^2})$ and space $O(b^{\lceil k/4 \rceil} + \frac{\log m \log \delta^{-1}}{\alpha\varepsilon^2})$ in one pass over $\mathcal{T}$ returning a value $\tilde{f}$ such that*

- *if $f/z \geq \alpha$, then $\tilde{f}$ is an (ε, δ)-approximation of f.*
- *otherwise, if $f/z < \alpha$, then $\tilde{f} \leq (1+\varepsilon)f$ with probability at least $1-\delta$.*

Extending a recent technique by Campagna and the authors [9], we show how to use Consistent Subset Sampling to parallelize frequent items mining algorithm like [10, 11] when applied to transactional data streams. Here, instead of estimating the number of frequent itemsets, we show how to distribute the computation such that we achieve good load balancing among different processors.

A second application is in the area of graph mining where a graph is provided as a stream of edges. The problem of estimating the number of fixed-size subgraphs in *incidence list streams*, i.e., a stream of edges where all edges incident to a vertex are provided one after another, has become very popular in the last decade [3, 4, 7, 8, 20, 20]. By associating b-sets with the set of a vertex neighbors, we design new algorithms for the estimation of the number of k-cliques in incidence list streams for bounded degree graphs. Also, we present the first algorithm for the estimation of the number of (i^+, j)-bipartite cliques, i.e., bipartite cliques with j vertices on the right-hand side and *at least* i vertices on the left hand side. We argue that this is a problem with important real-life applications and show that a straightforward applications of Consistent Subset Sampling yields the following result:

Theorem 4 *Let $G = (V, E)$ be a graph with n vertices, m edges and bounded degree Δ revealed as a stream of incidence lists. Let further $K_{i^+,j}$ be the number*

[1] `http://arxiv.org/pdf/1404.4693.pdf`

of (i^+, j)-bicliques in G and A_j the number of j-adjacencies in G for $i \geq 1, j \geq 2$. For any $\gamma, \varepsilon, \delta \in (0, 1]$ there exits a randomized algorithm running in expected time $O(n\Delta^{\lceil j/2 \rceil} \log\log \Delta + \frac{\log n \log \delta^{-1}}{\gamma \varepsilon^2})$ and space $O(\Delta^{\lceil j/4 \rceil} + \frac{\log n \log \delta^{-1}}{\gamma \varepsilon^2})$ in one pass over the graph returning a value $\tilde{K}_{i^+,j}$ such that

- *if $K_{i^+,j}/A_j \geq \gamma$, $\tilde{K}_{i^+,j}$ is an (ε, δ)-approximation of $K_{i^+,j}$.*
- *otherwise, if $K_{i^+,j}/A_j < \gamma$, $\tilde{K}_{i^+,j} \leq (1+\varepsilon)K_{i^+,j}$ with probability at least $1 - \delta$.*

6 Conclusions

Finally, we make a few remarks about possible improvements in the running time of our consistent sampling technique. As one can see, the algorithmic core of our approach is closely related to the d-SUM problem where one is given an array of n integers and the question is to find d integers that sum up to 0. The best known randomized algorithm for 3-SUM runs in time $O(n^2(\log\log n)^2/\log^2 n)$[1], thus it is difficult to hope to design an algorithm enumerating all 3-subsets satisfying the sampling condition much faster than in $O(b^2)$ steps. Moreover, Pătraşcu and Williams [22] showed that solving d-SUM in time $n^{o(d)}$ would imply an algorithm for the 3-SAT problem running in time $O(2^{o(n)})$ contradicting the *exponential time hypothesis* [17]. It is even an open problem whether one can solve d-SUM in time $O(n^{\lceil d/2 \rceil - \alpha})$ for $d \geq 3$ and some constant $\alpha > 0$ [25].

In a recent work Dinur et al. [13] presented a new "dissection" technique for achieving a better time-space trade-off for the computational complexity of various problems. Using the approach from [13], we can slightly improve the results from Theorem 2. However, the details are beyond the scope of the present paper.

References

1. I. Baran, E. D. Demaine, M. Pătraşcu. Subquadratic Algorithms for 3SUM. *Algorithmica* 50(4): 584–596 (2008)
2. M. Boley, H. Grosskreutz. A Randomized Approach for Approximating the Number of Frequent Sets. *ICDM 2008*: 43–52
3. L. Becchetti, P. Boldi, C. Castillo, A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. *KDD 2008*: 162-24
4. I. Bordino, D. Donato, A. Gionis, S. Leonardi. Mining Large Networks with Subgraph Counting. *ICDM 2008*: 737–742
5. A. Z. Broder, M. Charikar, A. M. Frieze, M. Mitzenmacher. Min-Wise Independent Permutations. *J. Comput. Syst. Sci. 60(3)*: 630–659 (2000)
6. A. Z. Broder, S. C. Glassman, M. S. Manasse, G. Zweig. Syntactic Clustering of the Web. *Computer Networks 29(8-13)*: 1157–1166 (1997)
7. L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, C. Sohler. Counting triangles in data streams. *PODS 2006*: 253–262
8. L. S. Buriol, G. Frahling, S. Leonardi, C. Sohler. Estimating Clustering Indexes in Data Streams. *ESA 2007:* 618–632

9. A. Campagna, K. Kutzkov, R. Pagh. On Parallelizing Matrix Multiplication by the Column-Row Method. *ALENEX 2013*: 122–132.
10. M. Charikar, K. Chen, M. Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.* 312(1): 3–15 (2004)
11. G. Cormode, S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms 55(1)*: 58–75 (2005)
12. M. Dietzfelbinger, J. Gil, Y. Matias, N. Pippenger: Polynomial Hash Functions Are Reliable (Extended Abstract). *ICALP 1992*: 235–246
13. I. Dinur, O. Dunkelman, N. Keller, A. Shamir. Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and Combinatorial Search Problems. *CRYPTO 2012*: 719–740
14. F. Geerts, B. Goethals, J. Van den Bussche. Tight upper bounds on the number of candidate patterns. *ACM Trans. Database Syst.* 30(2): 333–363 (2005)
15. J. Han, M. Kamber. Data Mining: Concepts and Techniques *Morgan Kaufmann* 2000
16. Y. Han, M. Thorup. Integer Sorting in $O(n\sqrt{\log\log n})$ Expected Time and Linear Space. *FOCS 2002*: 135–144
17. R. Impagliazzo, R. Paturi, F. Zane: Which Problems Have Strongly Exponential Complexity? *J. Comput. Syst. Sci.* 63(4): 512–530 (2001)
18. P. Indyk. A Small Approximately Min-Wise Independent Family of Hash Functions. *J. Algorithms 38(1)*: 84–90 (2001)
19. R. Jin, S. McCallen, Y. Breitbart, D. Fuhry, and D. Wang. Estimating the number of frequent itemsets in a large database. In *EDBT*, pages 505–516, 2009.
20. D. M. Kane, K. Mehlhorn, T. Sauerwald, H. Sun. Counting Arbitrary Subgraphs in Data Streams. *ICALP 2012*: 598–609
21. D. M. Kane, J. Nelson, D. P. Woodruff. An optimal algorithm for the distinct elements problem. *PODS 2010*: 41–52
22. M. Pătraşcu, R. Williams. On the Possibility of Faster SAT Algorithms. *SODA 2010*: 1065–1075
23. R. Schroeppel, A. Shamir. A $T = O(2^{n/2}), S = O(2^{n/4})$ Algorithm for Certain NP-Complete Problems. *SIAM J. Comput.* 10(3): 456–464 (1981)
24. D. E. Willard. Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(N)$. *Inf. Process. Lett.* 17(2): 81–84 (1983)
25. G. J. Woeginger. Space and Time Complexity of Exact Algorithms: Some Open Problems (Invited Talk). *IWPEC 2004*: 281–290