Sampling Time-Based Sliding Windows in Bounded Space

Rainer Gemulla
Technische Universität Dresden
01062 Dresden, Germany
gemulla@inf.tu-dresden.de

Wolfgang Lehner
Technische Universität Dresden
01062 Dresden, Germany
lehner@inf.tu-dresden.de

ABSTRACT

Random sampling is an appealing approach to build synopses of large data streams because random samples can be used for a broad spectrum of analytical tasks. Users are often interested in analyzing only the most recent fraction of the data stream in order to avoid outdated results. In this paper, we focus on sampling schemes that sample from a sliding window over a recent time interval; such windows are a popular and highly comprehensible method to model recency. In this setting, the main challenge is to guarantee an upper bound on the space consumption of the sample while using the allotted space efficiently at the same time. The difficulty arises from the fact that the number of items in the window is unknown in advance and may vary significantly over time, so that the sampling fraction has to be adjusted dynamically. We consider uniform sampling schemes, which produce each sample of the same size with equal probability, and stratified sampling schemes, in which the window is divided into smaller strata and a uniform sample is maintained per stratum. For uniform sampling, we prove that it is impossible to guarantee a minimum sample size in bounded space. We then introduce a novel sampling scheme called bounded priority sampling (BPS), which requires only bounded space. We derive a lower bound on the expected sample size and show that BPS quickly adapts to changing data rates. For stratified sampling, we propose a merge-based stratification scheme (MBS), which maintains strata of approximately equal size. Compared to naive stratification, MBS has the advantage that the sample is evenly distributed across the window, so that no part of the window is over- or underrepresented. We conclude the paper with a feasibility study of our algorithms on large real-world datasets.

Categories and Subject Descriptors

H.2 [**Database Management**]: Miscellaneous; G.3 [**Probability and Statistics**]: Probabilistic algorithms

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06 ...$5.00.

General Terms

Algorithms,theory

1. INTRODUCTION

Random sampling techniques are at the heart of every data stream management system. In fact, it is often infeasible to process and/or store the entire data stream in high-speed applications like monitoring of sensor data, network traffic, or transaction logs. Random sampling is an appealing approach to build synopses of large data streams, since most analytical tasks can be executed on a sample either directly or in a slightly modified fashion. For example, random samples can be used to estimate sums, averages and quantiles, but they also support complex data mining tasks such as clustering.

The main challenge in data stream sampling is to maintain samples that represent only the part of the data stream relevant for analysis. Since many analytical tasks focus on only a recent part of the data stream, it is often unnecessary or infeasible to maintain a sample of the data stream in its entirety [1]. Current research addresses this problem by maintaining so-called *sequence-based* (SB) samples. In an SB sampling scheme, the probability of an item being included in the sample depends only on its position in the data stream. Popular variants are the maintenance of a uniform sample of the k most recent items [2, 9] or biased sampling schemes in which the inclusion probability of an item decays as new items arrive [9, 1]. In general, SB schemes provide for efficient sample maintenance in bounded space. Their disadvantage, however, is that they are not well-suited for *time-based* analysis. To see this, consider the following simple CQL query:

```
SELECT SUM(size) AS num_bytes
FROM packets [Range 60 Minutes]
```

This query monitors the number of bytes observed in the `packets` stream during the last 60 minutes. Suppose that we want to answer the above query in a continuous fashion by maintaining an SB sample of the k most recent items. Since the sample is sequence-based and the query is time-based, we have to choose k in such a way that the last k items are guaranteed to completely cover the 60-minute range of the query. Clearly, such a choice is impossible if no a-priori knowledge about the data stream is available. But even if we can come up with an upper bound for the number of items in the query range, SB schemes may perform poorly in practice. The reason is that—unless the data stream rate is roughly constant—the average window size is much smaller than the

upper bound, so that (with high probability) the sample contains a large fraction of outdated items not relevant for the query.

The above problems are addressed by *time-based* (TB) sampling schemes [2, 9]. In such a scheme, the inclusion probability of an item depends on its timestamp rather than on its position in the data stream. In this paper, we focus on sampling schemes that maintain a random sample of a time-based sliding window. For example, a random sample of the packets arrived during the last 60 minutes can directly be used to approximately answer the query above. In TB sampling, the main challenge is to realize an upper bound on the space consumption of the sample while using the allotted space efficiently at the same time. Space bounds are crucial in data stream management systems with many samples maintained in parallel, since they greatly simplify the task of memory management and avoid unexpected memory shortcomings at runtime. The difficulty of TB sampling arises from the fact that the number of items in the window may vary significantly over time. If the sampling fraction is kept constant at all times, the sample shrinks and grows with the number of items in the window. The size of the sample is therefore unstable and unbounded. To avoid such behavior, the sampling fraction has to be adapted on-the-fly. Intuitively, we can afford a large sampling fraction at low stream rates but only a low sampling fraction at high rates.

Uniform sampling schemes, which produce each sample of the same size with equal probability, are the most general of the available sampling schemes. Uniform samples are well understood and, in fact, many statistical estimators require an underlying uniform sample. In the context of data stream systems, uniform sampling is applied either to the data sources directly or to the output of some of the operators in the query graph. It is used to reduce or bound resource consumption, to support ad-hoc querying, to analyze why the system produced a specific output or to optimize the query graph. Though more efficient techniques exist for some of these applications, uniformity is a must if knowledge about the intended use of the sample is not available at the time the sample is created. This is because uniform samples are not tailored to a specific set of queries but provide a versatile synopsis of the underlying data.

In this paper, we are concerned with sampling schemes that maintain a uniform sample of a time-based sliding window *in bounded space*. We show that any such sampling scheme cannot provide hard sample size guarantees. Thus, any bounded-space uniform scheme produces samples of variable size, and sample size guarantees are, if available, probabilistic. We then introduce a novel uniform sampling scheme called bounded priority sampling (BPS). The scheme is based on priority sampling [2] but requires only bounded space. To the best of our knowledge, BPS is the first bounded-space uniform sampling scheme for time-based sliding windows. We analyze the distribution of the sample size and show that the algorithm quickly adapts to changing data rates. By leveraging results from the area of distinct-value estimation [3], we also show how the number of items in the window can be estimated from the sample.

For applications where the uniformity requirement is not crucial, we introduce a more space-efficient stratified sampling scheme [9]. In stratified sampling, the window is partitioned into non-overlapping time intervals called strata. For each stratum, a uniform sample is maintained. Stratified samples are easier to maintain than uniform samples, but estimation becomes more involved. For example, it is not known how to estimate the number of distinct values of an attribute from a stratified sample. If stratified samples can be used, however, estimates may become more precise [9]. In this paper, we discuss the problem of both placing stratum boundaries and maintaining the corresponding samples. We develop a merge-based stratified scheme (MBS), which maintains strata of approximately equal size. The algorithm merges adjacent strata from time to time; the decision of when to merge and which strata to merge is a major contribution of this paper. In our solution, we treat the problem as an optimization problem and give a dynamic programming algorithm to determine the optimum stratum boundaries.

The remainder of the paper is structured as follows. In Section 2, we review existing techniques from the database and data stream literature. In Section 3, we discuss the problem of maintaining a uniform sample of a time-based sliding window and give the details of the BPS algorithm. Section 4 introduces merge-based stratification, which is then used to maintain strata of approximately equal size. We present the results of our experimental evaluation in Section 5 and conclude the paper in Section 6.

2. EXISTING TECHNIQUES

In this section, we review existing sampling techniques and discuss their applicability for the setting of time-based sliding windows. Since the focus of this paper is on bounded-space sampling schemes, we also analyze the sample size and space consumption of the available schemes.

Database sampling. A variety of sample maintenance techniques have been proposed in the context of relational database systems. The most popular database sampling technique is the reservoir sampling scheme [15]. The scheme maintains a uniform random sample of size k of an insertion-only dataset by intercepting insertion requests on their way to the dataset. The idea is to add the first k inserted items directly to the sample. Subsequent items are accepted into the sample with probability $k/(N+1)$, where N is the number of items processed so far, or ignored otherwise. Accepted items replace a sample item chosen uniformly at random. Reservoir sampling has been extended to support updates and deletions [8, 7], so that one might consider using it for time-based sliding windows. The idea is to treat each arrival as an insertion into the window and each expiration as a deletion from the window. This approach does not work, however, since deletions are explicit in relational databases but implicit in time-based sliding windows. That means that the database sampling schemes require to be aware of every deletion—whether the deleted item is sampled or not—, while a window sampling scheme only observes the expiration of sampled items. For this reason, none of the available database sampling schemes can be applied to sliding windows.

Sequence-based sampling. In a sequence-based sampling scheme, the probability of an item being included in the sample depends only on its position in the data stream. Babcock et al. [2] discuss several sampling schemes that maintain a uniform random sample of the last N elements of the stream. In [9], a stratified sampling scheme for the same purpose is given. The idea is to partition the data stream into a set of equally-sized strata and to maintain a reservoir sample

of each non-expired stratum. In Section 4, we apply this idea to time-based sliding windows; the key difference to [9] is that the determination of stratum boundaries becomes much harder. An alternative approach to focus attention on the recent items is to maintain a biased sample [9, 1]. In these schemes, the probability of an item being sampled decays as new items arrive. Again, this sequence-based notion of recency does not match the time-based notion of analysis, so that sequence-based schemes can only be used if a-priori knowledge about the stream is available.

Bernoulli sampling. In [2], a modified version of Bernoulli sampling, which maintains a uniform sample of a sequence-based window, has been proposed. The method can easily be adapted to the setting of time-based windows. Let $q \in (0,1)$ be the desired sampling rate. In the adapted scheme, each item is included into the sample with probability q—independent of the other items—and excluded with probability $1-q$. Items are removed from the sample if and only if they expire. Suppose that at some arbitrary point in time, the sliding window contains N items. Then, the expected sample size is qN and the actual sample size is close to qN with high probability. The size of the sample therefore grows and shrinks with the number of items in the sliding window. One might hope that it is possible to decrease (increase) q dynamically whenever the sample size gets too large (too small). However, it has been shown recently that such a modification of q destroys the uniformity of the sample [7], so that it is impossible to control the size of a Bernoulli sample.

Priority sampling. The priority sampling scheme [2] maintains a uniform sample of size 1 from a time-based sliding window. Larger samples can be obtained by running multiple priority samplers in parallel. The idea is to assign a random priority between 0 and 1 to each arriving item. At any time, the algorithm reports the item with highest priority in the window as the sample item. Since each item has the same probability of having the highest priority, the scheme is indeed uniform.[1] In order to be able to always report the highest-priority item, it is both necessary and sufficient to store the items for which there is no element with both a larger timestamp and a higher priority. If, as above, the window contains N items at an arbitrary point in time, the scheme requires $O(\log N)$ space in expectation, the actual space requirement is also $O(\log N)$ with high probability [2]. Thus, the space consumption of priority sampling cannot be bounded from above.

To summarize, none of the available sampling schemes can be used to maintain a random sample from a time-based sliding window *in bounded space*. We address this situation and introduce both a uniform and a stratified bounded-space sampling scheme.

3. UNIFORM SAMPLING

We model a data stream as an infinite sequence $R = (e_1, e_2, \ldots)$ of items. Each item e_i has the form (t_i, d_i), where $t_i \in \Re$ denotes a timestamp and $d_i \in \mathcal{D}$ denotes the data associated with the item. The data domain $\mathcal{D}$ depends on the application; for example, $\mathcal{D}$ might correspond to a finite set of IP addresses or an infinite set of readings from one or more sensors. Throughout the paper, we assume that $t_i < t_j$ for $i < j$, that is, the timestamps of the items are strictly increasing.[2] Denote by $R(t)$ the set of items from R with a timestamp smaller than or equal to t. Denote by $W_\Delta(t) = R(t) \setminus R(t-\Delta)$ a sliding window of length Δ and denote by $N_\Delta(t) = |W_\Delta(t)|$ the size of the window at time t. For brevity, we will supress the subscript Δ in the following. Note that we use the term *window length* to refer to the timespan covered by the window (Δ, fixed) and the term *window size* to refer to the number of items in the window ($N(t)$, varying).

In this section, we study the problem of maintaining a uniform random sample from $W(t)$ in bounded space. We consider sampling schemes that maintain a data structure from which a uniform random sample $S(t)$ of the items in $W(t)$ can be extracted at any time. The distinction between data structure and sample allows to examine the space consumption and the sample size separately. A sampling scheme is called uniform if for any $A_1, A_2 \subseteq W(t)$ with $|A_1| = |A_2|$ the probability $P\{\, S(t) = A_1 \,\}$ that the scheme produces A_1 satisfies

$$P\{\, S(t) = A_1 \,\} = P\{\, S(t) = A_2 \,\}.$$

Thus, the probability that a sampling scheme produces A_1 depends only on $|A_1|$ and not on its composition.

3.1 A Negative Result

One might hope that there is a sampling scheme that is able to maintain a fixed-size uniform sample in bounded space. However, such a scheme does not exist.

THEOREM 1. *Fix some time t and set $N = N(t)$. Then, any algorithm that maintains a fixed-size uniform random sample of size k requires at least $\Omega(k \log N)$ space in expectation.*

PROOF. Let $\mathcal{A}$ be an algorithm that maintains a uniform size-k sample of a time-based sliding window and denote by $W = \{\, e_{m+1}, \ldots, e_{m+N} \,\}$ the items in the window at time t. Furthermore, denote by $t_j^- = t_{m+j} + \Delta$ the point in time when item e_{m+j} expires, $1 \le j < N$, and set $t_0^- = t$. Now, consider the case where no new items arrive in the stream until all the N items have expired. Then, let I_j be a 0/1-random variable and set $I_j = 1$ if the sample reported by $\mathcal{A}$ at time t_j^- contains item e_{m+j}. Otherwise, set $I_j = 0$. Since $\mathcal{A}$ has to store all items it eventually reports, it follows that—at time t_0^-—$\mathcal{A}$ stores at least $X = \sum I_j$ items. We have to show that $E\,[\, X \,] = \Omega(k \log N)$.

Since $\mathcal{A}$ is a uniform sampling scheme, item e_{m+1} is reported at time t_0^- with probability k/N. At time t_1^-, only $N-1$ items remain in the window and item e_{m+2} is reported with probability $k/(N-1)$. The argument can be repeated until at time t_{N-k}, all the k remaining items are reported by $\mathcal{A}$. It follows that

$$P\{\, I_j = 1 \,\} = \begin{cases} k/(N-j) & 0 \le j < N-k \\ 1 & \text{otherwise} \end{cases} \qquad (1)$$

for $0 \le j < N$. Note that only the marginal probabilities are given in (1); joint probabilities like $P\{\, I_1 = 1, I_2 = 1 \,\}$

[1]This concept is also known as min-wise sampling [11].

[2]The algorithms in this paper also work when $t_i \le t_j$ for $i < j$, but we will use the stronger assumption $t_i < t_j$ for expository reasons.

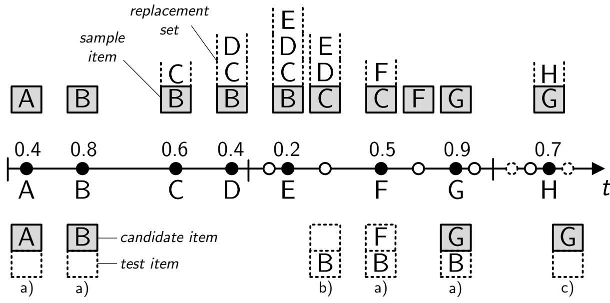

Figure 1: Illustration of PS (above timeline) and BPS (below timeline)

depend on the internals of $\mathcal{A}$. By the linearity of expected value, and since $E[I_j] = P\{I_j = 1\}$, we find that

$$E[X] = \sum_{j=0}^{N-1} E[I_j] = k(H_N - H_k + 1) = \Omega(k \ln N),$$

where $H_n = \sum_{i=1}^{n} 1/i = O(\ln n)$ denotes the nth harmonic number. □

It follows directly that it is impossible to maintain a fixed-size uniform random sample from a time-based sliding window in bounded space. By Theorem 1, such maintenance requires expected space logarithmic to the window size (which is unbounded); the worst-case space consumption is at least as large. It is not possible either to guarantee a minimum sample size because any algorithm that guarantees a minimum sample size can be used to maintain a sample of size 1. In the light of Theorem 1, we also note that the priority sampling scheme (see Section 2) is asymptotically optimal in terms of expected space. However, the algorithm has a multiplicative overhead of $\ln N$ and therefore a low space efficiency.

3.2 Bounding the Space Consumption

We now develop a bounded-space uniform sampling scheme based on priority sampling (PS). Recall that in priority sampling, a random priority p_i chosen uniformly at random from the unity interval is associated with each item $e_i \in R$. The sample $S(t)$ then consists of the item in $W(t)$ with the largest priority. In addition to the sample item, the scheme stores a set of *replacement items*, which replace the largest-priority item when it expires. This replacement set consists of all the items for which there is no item with both a larger timestamp and a higher priority. Figure 1 gives an example of the sampling process. A solid black circle represents the arrival of an item; its name and priority are given below and above, respectively. The vertical bars on the timeline indicate the window length, item expirations are indicated by white circles, and double-expirations[3] are dotted white circles. Above the timeline, the current sample item and the set of replacement items are shown. It can be seen that the number of replacement items stored by the algorithm varies over time. In fact, the replacement set is the reason for the unbounded space consumption of the sampling scheme: it contains between 0 and $N(t) - 1$ items and roughly $\ln N(t)$ items on average [2].

[3] An item that arrived at time t double-expires at time $t+2\Delta$.

We now describe our bounded-space priority sampling (BPS) scheme. The scheme also assigns random priorities to arriving items but stores at most two items in memory: a *candidate item* from $W(t)$ and a *test item* from $W(t-\Delta)$. The test item is used to determine whether or not the candidate item is reported as a sample item, see the discussion below. The maintenance of these two items is as follows:

a) *Arrival of item e_i.* If there is currently no candidate item or if the priority of e_i is larger than the priority of the candidate item, e_i becomes the new candidate item and the old candidate is discarded. Otherwise, the arriving item is ignored.

b) *Expiration of candidate item.* The expired candidate becomes the test item; we only store the timestamp and the priority of the test item. There is no candidate item until the next item arrives in the stream.

c) *Double-expiration of test item.* The test item is discarded.

The above algorithm maintains the following invariant: The candidate item always equals the highest-priority item that has arrived in the stream since the expiration of the former candidate item. This might or might not coincide with the highest-priority item in the current window and we use the test item to distinguish between these two cases. Suppose that at some time, the candidate item expires and becomes the test item. Then the candidate must have been the highest-priority item in the window right before its expiration. (If there were an item with a higher priority, this item would have replaced the candidate.) It follows that whenever the candidate item has a higher priority than the current test item, we know that the candidate is the highest-priority item *since the arrival of the test item* and therefore since the start of the current window. Similarly, whenever there is no test item stored by BPS, there hasn't been an expiration of a candidate item for at least one window length, so that the candidate also equals the highest-priority item in the window. In both cases, we report the candidate as a sample item. Otherwise, if the candidate item has a lower priority than the test item, we have no means to detect whether or not the candidate equals the highest-priority item in the window and no sample item is reported.

Before we assert the correctness of BPS and analyze its properties, we give an example of the sampling process in Figure 1. The current candidate item and test item are shown below the timeline. If the candidate item is shaded, it is reported as a sample item; otherwise, no sample item is reported. The letters below the BPS data structure refer to cases a), b) and c) above. As long as no expiration occurs, the candidate stored by BPS equals the highest-priority item in the window and is therefore reported as a sample item. The situation changes as B expires. BPS then makes item B the test item and—because there is no candidate item anymore—fails to report a sample item. This failure can be seen as a consequence of Theorem 1: BPS is a bounded-space sampling scheme and thus cannot guarantee a fixed sample size. Item F becomes the new candidate item upon its arrival. However, F is not reported because its priority is lower than the priority of the test item B. And in fact, not F but C is the highest-priority item in the window at this time. Later C expires and F does become the highest-priority item in the window. However, we still do not report

F since we are not aware of this situation. As G arrives, however, we report a sample item again because G has a higher priority than the test item B. Finally, item B is discarded from the BPS data structure as it double-expires.

3.3 Correctness and Analysis

We now establish the correctness of the BPS algorithm. Recall that BPS produces either an empty sample or a single-item sample. Given that BPS does produce a sample item, we have to show that this item is chosen uniformly and at random from the items in the current sliding window.

THEOREM 2. *BPS is a uniform sampling scheme, that is, for any $e_j \in W(t)$, we have*

$$P\{ S(t) = \{ e_j \} \mid |S(t)| = 1 \} = 1/N(t).$$

PROOF. Fix some time t and set $S = S(t)$. Denote by e_{max} the highest-priority item in $W(t)$ and suppose that e_{max} has priority p_{max}. Furthermore, denote by $e' \in W(t-\Delta)$ the candidate item stored in the BPS data structure at time $t-\Delta$ (if there is one) and let p' be the priority of e'. Note that both e_{max} and e' are random variables. There are 3 cases.

Case 1: There is no candidate item at time $t-\Delta$. Then at time t, e_{max} is the candidate item and there is no test item. We have $S = \{ e_{max} \}$.

Case 2: Item e' has a smaller priority than e_{max}. Then e_{max} is the candidate item at time t and—depending on whether e' expired before or after the arrival of e_{max}—the test item is either equal to e' or empty. In both cases, we have $S = \{ e_{max} \}$.

Case 3: Item e' has a higher priority than e_{max}. Then, e' is still the candidate item at the time of its expiration, since there is no higher-priority item in $W(t)$ that might have replaced e'. Thus, item e' becomes the test item upon its expiration and continues to be the test item up to time t—it double-expires somewhere in the interval $(t, t+\Delta)$. It follows that no item is reported at time t so that $S = \emptyset$, because the priority of the candidate item ($\leq p_{max}$) is lower than the priority p' of the test item.

To summarize, we have

$$S = \begin{cases} \{ e_{max} \} & \text{no candidate item at time } t-\Delta \\ \{ e_{max} \} & p_{max} > p' \\ \emptyset & \text{otherwise.} \end{cases} \quad (2)$$

Uniformity now follows since (2) does not depend on the values, timestamps or order of the individual items in $W(t)$. For any $e_j \in W(t)$, we have

$$P\{ S = \{ e_j \} \mid |S| = 1 \} = P\{ e_j = e_{max} \} = 1/N(t)$$

and the theorem follows. □

We now analyze the sample size of the BPS scheme. Clearly, the sample size is probabilistic and its exact distribution depends on the entire history of the data stream. However, in the light of Theorem 3 below, it becomes evident that we can still provide a *local* lower bound on the probability that the scheme produces a sample item. The lower bound is local because it changes over time; we cannot guarantee a global lower bound other than 0 that holds at any arbitrary time without a-priori knowledge of the data stream.

THEOREM 3. *The probability that BPS succeeds in producing a sample item at time t is bounded from below by*

$$P\{ |S(t)| = 1 \} \geq \frac{N(t)}{N(t-\Delta) + N(t)}.$$

PROOF. BPS produces a sample item if the highest-priority item $e_{max} \in W(t)$ has a higher priority than the candidate item e' stored in the BPS data structure right before the start of $W(t)$; see (2) above. In the worst case, e' equals the highest-priority item in $W(t-\Delta)$. Now suppose that we order the items in $W(t-\Delta) \cup W(t)$ in descending order of their priorities. BPS succeeds for sure if the first of the ordered items is an element of $W(t)$. Since the priorities are independent and identically distributed, this event occurs with probability $N(t)/(N(t-\Delta)+N(t))$ and the assertion of the theorem follows. □

If the arrival rate of the items in the data stream is constant so that $N(t) = N(t-\Delta)$, BPS succeeds with probability of at least 50%. If the rate increases or decreases, the success probability will also increase or decrease, respectively.

3.4 Sampling Multiple Items

The BPS scheme as given above can be used to maintain a single-item sample. A straightforward way to obtain larger samples is to run k independent BPS samplers $S_1, \ldots, S_k$ in parallel; we refer to this scheme as BPS with replacement (BPSWR). The sample is then set to $S = S_1 \cup \cdots \cup S_k$. We have

$$E[\,|S|\,] = \sum_{i=1}^{k} P\{ |S_i| = 1 \} \geq k \frac{N(t)}{N(t-\Delta) + N(t)}$$

by the linearity of the expected value. However, this approach has two major drawbacks. First, the sample S is a with-replacement sample, that is, each item in the window may be sampled more than once. The net sample size after duplicate removal might therefore be smaller than $|S|$. Second and more importantly, the maintenance of the k independent samples is expensive. Since a single copy of the BPS data structure requires constant time per arriving item, the per-item processing time is $O(k)$ and the total time to process a window of size N is $O(kN)$. If k is large, the overhead to maintain the sample can be significant.

We now develop a without-replacement sampling scheme called BPSWOR. In general, without-replacement samples are preferable since they contain more information about the data. The scheme is as follows: we modify BPS so as to store k candidates and k test items simultaneously. Denote by S_{cand} the set of candidates and by S_{test} the set of test items. The sampling process is similar to BPS: An arriving item e becomes a candidate when either $|S_{cand}| < k$ or e has a higher priority than the lowest-priority item in S_{cand}. In the latter case, the lowest-priority item is discarded in favor of e. As before, expiring candidates become test items and double-expiring test items are discarded. The sample $S(t)$ is then given by

$$S(t) = \text{top-k}\left(S_{cand}(t) \cup S_{test}(t) \right) \cap S_{cand}(t),$$

where top-k(A) determines the items in A with the k highest priorities. Note that for $k = 1$, BPSWR and BPSWOR coincide. $S(t)$ is then a uniform random sample of $W(t)$ without replacement; the proof is similar to the proof of

Theorem 2. Also, using an argument as in the proof of Theorem 3, we can show that $E[S(t)] \geq kN(t)/(N(t-\Delta)+N(t))$. Thus, BPSWR and BPSWOR have the same lower bound on the expected (gross) sample size. The cost of processing a window of size N is $O(kN)$ if the candidates are stored in a simple array. A more efficient approach—which also improves the cost in comparison to BPSWR—is to store the candidates in a treap, where the items are arranged in order with respect to the timestamps and in heap-order with respect to the priorities. The expected cost of BPSWOR then decreases to $O(N + k \log k \log N)$ in expectation.[4]

Note that we can also modify PS to sample without replacement. The so-modified PSWOR scheme then reports the items with the k highest priorities in the window. In order to maintain these k items incrementally, we store each item as long as there are fewer than k more recent items with a higher priority. The space consumption is still $O(k \log N)$ in expectation, but efficient maintenance of the replacement set becomes challenging. Since the focus of this paper is on bounded-space sampling schemes, we do not further elaborate on this issue.

3.5 Estimation of Window Size

For some applications, it is important to be able to estimate the window size in order to make effective use of the sample. For example, the window sum of an attribute is typically estimated as the sample average of the respective attribute multiplied by the window size. Thus—in some applications—knowledge of the window size is important to determine scale-up factors.

Exact maintenance of the number of items in the window requires that we store all the timestamps in the window in order to deal with expirations. Typically, this approach is infeasible in practice. Approximate data structures [6] do exist and can be leveraged to support the sampling process. If such alternate data structures are unavailable, we can come up with an estimate of the window size directly from the sample. Set $W_2(t) = W(t-\Delta) \cup W(t)$ and denote by $p_{(k)}$ the priority of the item with the kth highest priority in $W_2(t)$. In [3], it has been shown that an unbiased estimator for $N(t)$ is given by

$$\hat{N}_W(t) = \frac{|W(t) \cap \text{top-k}\, W_2(t)|}{k} \frac{k-1}{1-p_{(k)}}.$$

Here, the first factor estimates the fraction of non-expired items in $W_2(t)$ from the top-k items (which can be viewed as a random sample of W_2), while the second factor is an estimate of $|W_2(t)|$ itself. Now, suppose that we maintain the sample using BPSWOR. Set $S_2(t) = S_{cand} \cup S_{test}$ and denote by $p'_{(k)}$ the priority of the item with the kth highest priority in S_2. Consider the estimator

$$\hat{N}_S(t) = \frac{|S(t) \cap \text{top-k}\, S_2(t)|}{k} \frac{k-1}{1-p'_{(k)}}.$$

This estimator is similar to $\hat{N}_W(t)$ but solely accesses information available in the sample. Both estimators coincide if and only if $\text{top-k}\, S_2(t) = \text{top-k}\, W_2(t)$. This happens if at least $|W(t-\Delta) \cap \text{top-k}\, W_2(t)|$ items have been reported as the sample at time $t-\Delta$. Otherwise, the first factor in $\hat{N}_S(t)$ will overestimate the first factor in $\hat{N}_W(t)$, while the second factor will underestimate the respective factor in $\hat{N}_W(t)$. In our experiments, we found that the estimator $\hat{N}_S$ has negligible bias and low variance. Thus, both over- and underestimation seem to balance smoothly, though we do not make any formal claims here.

4. STRATIFIED SAMPLING

We now consider the problem of maintaining a stratified sample of a time-based sliding window. The general idea is to partition the window into disjoint strata and to maintain a uniform sample of each stratum [9]. Stratified sampling is often superior to uniform sampling because a stratified scheme exploits correlations between time and the quantity of interest. As will become evident later on, stratification also allows us to maintain larger samples than with BPS in the same space. The main drawback of stratified sampling is its limited applicability; for some problems, it is difficult or even impossible to compute a global solution from the different subsamples. For example, it is not known how the number of distinct values can be estimated from a stratified sample, while the problem has been studied extensively for uniform samples [5]. If, however, the desired analytical tasks can be performed on a stratified sample, stratification is often the method of choice.

We consider stratified sampling schemes, which partition the window into $l > 1$ strata and maintain a uniform sample S_i of each stratum, $1 \leq i \leq l$. Each sample has a fixed size of n items. In addition to the sample, we also store the stratum size N_i and the timestamp t_i of the upper stratum boundary; these two quantities are required for sample maintenance. The main challenge in stratified sampling is the placement of stratum boundaries because they have a significant impact on the quality of the sample.[5] In the simplest version, the stream is divided into strata of equal width (time intervals); we refer to this strategy as *equi-width stratification*. An alternative strategy is *equi-depth stratification*, where the window is partitioned into strata of equal size (number of items). Equi-depth stratification outperforms equi-width stratification when the arrival rate of the data stream varies inside a window, but the strata are much more difficult to maintain. In fact, perfect equi-depth stratification is impossible (see below), so that approximate solutions are needed. In this section, we develop a *merge-based* stratification strategy, which approximates equi-depth stratification to the best possible extent.

Figure 2 illustrates equi-width stratification with parameters $l = 4$ and $n = 1$; sampled items are represented by solid black circles. The figure displays a snapshot of the sample at 3 different points in time, which are arranged vertically and termed a), b) and c). Note that the rightmost stratum ends at the right window boundary and grows as new items arrive, while the leftmost stratum exceeds the window and may contain expired items. The maintenance of the stratified sample is significantly simpler than the maintenance of a uniform sample because arrivals and expirations are not

[4] Following an argument as in [3], at most $O(k \log N)$ items of the window are accepted into the candidate set in expectation and each accepted item incurs an expected cost of $O(\log k)$ [13]. At most k items (double-)expire while processing a window, so that the expected cost to process (double-)expirations is $O(k \log k)$.

[5] To see this, consider the simple case where all items in the window fall into only one of the l strata. In this case, a fraction of $100(l-1)/l\%$ of the available space remains unused.

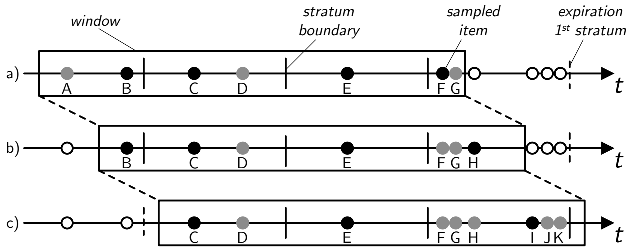

Figure 2: Equi-width stratification

intermixed within strata. Arriving items are added to the rightmost stratum and—since no expirations can occur—we can use reservoir sampling to maintain the sample incrementally (see Section 2). On the contrary, expirations only affect the leftmost stratum. We remove expired items from the respective sample; the remaining sample still represents a uniform sample of the non-expired part of the stratum [8].

4.1 Effect of Stratum Sizes

The main advantage of equi-width stratification is its simplicity, the main disadvantage is that the sampling fraction may vary widely across the strata. In the example of Figure 2c), the sampling fractions of the first, second and third stratum are given by 50%, 100% and 16%, respectively. In general, dense regions of the stream are underrepresented by an equi-width sample, while sparse regions are overrepresented. Thus, we want to stratify the data stream in such a way that each stratum has approximately the same size and therefore the same sampling fraction; we refer to this approach as equi-depth stratification. Unfortunately, perfect equi-depth stratification is not realizable in practice because the data stream is unknown in advance and we cannot move stratum boundaries arbitrarily. Before we introduce our approximate merge-based algorithm, we discuss the relationship of stratum sizes and accuracy with the help of a simple example.

Suppose that we want to estimate the window average μ of some attribute of the stream from a stratified sample and assume for simplicity that the respective attribute is normally distributed with mean μ and variance σ^2. Further suppose that at some time the window contains N items and is divided into l strata of sizes $N_1, \dots, N_l$ with $\sum N_i = N$. Then, the standard Horvitz-Thompson estimator $\hat{\mu}$ of μ is a weighted average of the per-stratum sample averages [12], that is $\hat{\mu} = \frac{1}{N}\sum_{i=1}^{l} N_i \hat{\mu}_i$, where $\hat{\mu}_i$ is the sample average of the ith stratum. The estimator has variance

$$\mathrm{Var}\left[\,\hat{\mu}\,\right] = \frac{1}{N^2}\sum_{i=1}^{l} N_i^2 \,\mathrm{Var}\left[\,\hat{\mu}_i\,\right] = \frac{\sigma^2}{nN^2}\sum_{i=1}^{l} N_i^2,$$

where we used $\mathrm{Var}\left[\,\hat{\mu}_i\,\right] = \sigma^2/n$. Thus, the variance of the estimator is proportional to the sum of the squares of the stratum sizes, or similarly, the variance of the stratum sizes:

$$\mathrm{Var}\left[\,N_1, \dots, N_l\,\right] = \sum_{i=1}^{l}\left(N_i - \frac{N}{l}\right)^2 = \frac{\sum N_i^2}{l} - \left(\frac{N}{l}\right)^2 \quad (3)$$

The variance is minimized if all strata have the same size (best case) and maximized if one stratum contains all the items in the window (worst case).

The above example is extremely simplified because we designed the stream in such a way that the variance $\mathrm{Var}\left[\,\hat{\mu}_i\,\right]$ of the estimate is equal in all strata. In general, stratification is the more efficient the higher the correlation of the attribute of interest with time gets (because time is the stratification variable). In this paper, however, we assume that no information about the intended use of the sample is available; in this case, our best guess is to assume equal variance in each stratum. Thus, the variance of the stratum sizes as given in (3) can be used to quantify the quality of a given stratification.

4.2 Merge-Based Stratification

Perfect equi-depth stratification is impossible, since we cannot reposition stratum boundaries arbitrarily. To see this, consider the state of the sample as given in Figure 2c). To achieve equi-depth stratification, we would have to (1) remove the stratum boundary between items D and E, and (2) introduce a new stratum boundary between H and I. Here, (1) represents a *merge* of the first and second stratum. In [4], Brown et al. have shown that such a merge is possible, that is, a sample of the merged stratum can be computed from the samples of the individual strata. In the example, the merged sample would contain item C with probability 2/3 and item E with probability 1/3. In contrast, (2) represents a *split* of the third stratum into two new strata, one containing items F-H and one containing items I-K. In the case of a split, it is neither possible to compute the samples of the two new strata nor to determine the stratum sizes. In the example, prior to the split, the third stratum has size 6 and the sample contains item I. Based on this information, it is impossible to come up with a sample of stratum F-H; we cannot even determine that stratum F-H contains 3 items.

Our merge-based stratified sampling scheme (MBS) approximates equi-depth stratification to the extent possible. The main idea is to merge two adjacent strata from time to time. Such a merge *reduces the information* stored about the two strata but *creates free space* at the end of the sample, which can be used for future items. In Figure 3, we illustrate MBS on the example data stream. We start as before with the 4 strata given in a). Right after the arrival of item H, we merge stratum C-D with stratum E to obtain stratum C-E. The decision of *when* and *which* strata to merge is the major challenge of the algorithm. After a merge, we use the freed space to start a new, initially empty stratum. The state of the sample after the creation of the new stratum is shown in b). Subsequent arrivals are added to the new stratum (items I, J and K). Finally, stratum A-B expires and, again, a fresh stratum is created; see c). Note that the sample is much more balanced than with equi-width stratification (Figure 2).

Before we discuss when to merge, we briefly describe how to merge. Suppose that we want to merge two adjacent strata R_1 and R_2 with $|R_1|, |R_2| \geq n$. Denote by S_i, N_i, t_i the uniform sample (of size n), the stratum size and the upper boundary of stratum R_i, $i \in \{\,1, 2\,\}$. Then, the merged stratum $R = R_1 \cup R_2$ has size $N_1 + N_2$ and upper boundary t_2. In [4], Brown et al. have shown how to merge S_1 and S_2 to obtain a uniform sample S of $R_1 \cup R_2$ with $|S| = n$. Let X be a random variable for the number of items from

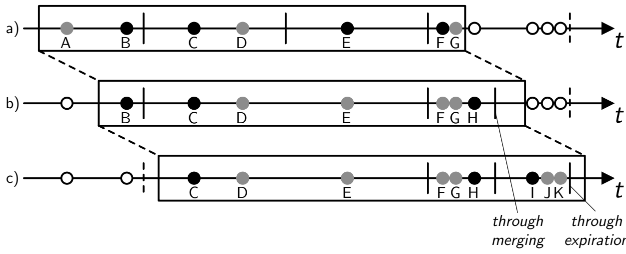

Figure 3: Merge-based stratification

R_1 in a size-n uniform sample drawn directly from R. X is hypergeometrically distributed with

$$P\{X = x\} = \binom{N_1}{x}\binom{N_2}{n-x}\Big/\binom{N_1+N_2}{n}$$

for $0 \leq k \leq n$. Since all the distribution parameters are known, we can obtain a realization x of X by throwing a dice. Then, we compute uniform subsamples S_1' and S_2' from S_1 and S_2, respectively, with $|S_1'| = x$ and $|S_2'| = n - x$. The subsamples can be computed using reservoir sampling, though more efficient sampling schemes exist for this purpose [14]. The final sample S is then set to the union of S_1' and S_2'; see [4] for a proof of the uniformity of S.

4.3 When To Merge Which Strata

The decision of when and which strata to merge is crucial for merge-based stratification. Suppose that at some time t, the window is divided into l strata $R_1, \ldots, R_l$ of size $N_1, \ldots, N_l$, respectively. During the subsequent sampling process, a new stratum is created when either (1) stratum R_1 expires or (2) two adjacent strata are merged. Observe that we have no influence on (1), but we can apply (2) as needed. We now treat the problem of when and which strata to merge as an optimization problem, where the optimization goal is *to minimize the variance of the stratum sizes at the time of the expiration of* R_1. Therefore—whenever the first stratum expires—the sample looks as much like an equi-depth sample as possible.

Denote by $R^+ = \{e_1, \ldots, e_{N^+}\}$ the set of items that arrive until the expiration of stratum R_1 (but have not yet arrived) and set $N^+ = |R^+|$.[6] At the time of R_1's expiration and before the creation of the new stratum, the window is divided into $l-1$ strata so that there are $l-2$ inner stratum boundaries. The positions of the stratum boundaries depend on both the number and point in time of any merges we perform. Our algorithm rests on the observation that for any way of putting $l-2$ stratum boundaries in the sequence

$$R_2, R_3, \ldots, R_l, e_1, e_2, \ldots, e_N+,$$

there is at least one corresponding sequence of merges that results in the respective stratification. For example, the stratification

$$R_2 \mid R_3 \mid \cdots \mid R_l, e_1, \ldots, e_N+$$

is achieved if no merge is performed (vertical bars denote boundaries), while

$$R_2 \mid \cdots \mid R_i, R_{i+1} \mid \cdots \mid R_l, e_1, \ldots, e_j \mid e_{j+1}, \ldots, e_N+$$

is achieved if stratum R_i and R_{i+1} are merged after the arrival of item e_j and before the arrival of item e_{j+1}. In general, for every stratum boundary in between $R_l, e_1, \ldots, e_{N^+}$, we drop a stratum boundary in between $R_2, \ldots, R_l$ by performing a merge operation at the respective point in time.

We can now reformulate the optimization problem: Find the partitioning of the integers

$$N_2, \ldots, N_l, \underbrace{1, \ldots, 1}_{N^+ \text{ times}}$$

into $l-1$ consecutive and non-empty partitions so that the variance (or sum of squares) of the intra-partition sums is minimized. The problem can be solved using dynamic programming in $O(l(l+N^+)^2)$ time [10]. In our specific instance of the problem, however, the last N^+ values of the sequence of integers are all equal to 1. As shown below, we can leverage this fact to construct a dynamic programming algorithm that obtains an optimum solution in only $O(l^3)$ time. Since N^+ is typically large, the improvement in performance can be significant.

The algorithm is as follows. Let $\text{opt}(k, i)$ be the minimum sum of squares when k of the $l-2$ boundaries are placed between $N_2, \ldots, N_l$ and the last one of these k boundaries is placed right after N_i; $0 \leq k \leq l-2$ and $k < i < l$. Then, $\text{opt}(k, i)$ can be decomposed into two functions

$$\text{opt}(k, i) = f(k, i) + g(k, i),$$

where $f(k, i)$ is the minimum sum of squares for the k partitions left of and including N_i and $g(k, i)$ is the minimum sum of squares for the $l-k-1$ partitions right of N_i. The decomposition significantly reduces the complexity because the computation of g does not involve any optimization. To define $g(k, i)$, observe that by definition, there are no boundaries in between $N_{i+1}, \ldots, N_l$, so that these values fall into a single partition and we can sum them up. The resulting part of the integer sequence is then

$$N_{i+1,l}, 1, \ldots, 1,$$

where $N_{a,b} = \sum_{j=a}^{b} N_j$.[7] In fact, g is minimized if all the $l-k-1$ partitions have the same size, that is, size $\frac{N_{i+1,l}+N^+}{l-k-1}$. If $N_{i+1,l}$ is larger than this average size, the minimum value of g cannot be obtained. In this case, the best choice is to put $N_{i+1,l}$ in one stratum for its own; the remaining $l-k-2$ partitions then all have size $\frac{N^+}{l-k-2}$. Thus, the function g is given by

$$g(k, i) = \begin{cases} (l-k-1)\left(\frac{N_{i+1,l}+N^+}{l-k-1}\right)^2 & N_{i+1,l} < \frac{N_{i+1,l}+N^+}{l-k-1} \\ N_{i+1,l}^2 + (l-k-2)\left(\frac{N^+}{l-k-2}\right)^2 & \text{otherwise.} \end{cases}$$

The function f can be defined recursively with

$$f(0, i) = N_{2,i}^2$$
$$f(k, i) = \min_{k \leq j < i} \left\{ f(k-1, j) + N_{j+1,i}^2 \right\}.$$

and the optimum solution is given by

$$\min_{0 \leq k \leq l-2} \min_{k < i < l} \text{opt}(k, i).$$

[6] In practice, N^+ is not known in advance; we address this issue in Section 4.4.

[7] $N_{a,b}$ can be computed in constant time with the help of an array containing the prefix sums $N_{2,2}, \ldots, N_{2,l}$ [10].

To compute the optimum solution, we iterate over k in increasing order and memoize the values of $f(k, \cdot)$; these values will be reused for the computation of $f(k+1, \cdot)$. The global solution and the corresponding stratum boundaries are tracked during the process. Since each of the loop variables k, i and j take at most l different values, the total time complexity is $O(l^3)$. The algorithm requires $O(l)$ space.

4.4 Estimation of Arriving-Item Count

The decision of when to merge is dependent on the number N^+ of items that arrive until the expiration of the first stratum. In practice, N^+ is unknown and has to be estimated. In this section, we propose a simple and fast-to-compute estimator for N^+. Especially for bursty data streams, estimation errors can occur; we therefore discuss how to make MBS robust against estimation errors.

As before, suppose that—at some time t—the sample consists of l strata of sizes $N_1, \dots, N_l$ and denote by t_i the upper boundary of the ith stratum, $1 \leq i \leq l$. Furthermore, denote by $\Delta^- = t_1 + \Delta - t$ the time span until the expiration of the first stratum. We want to predict the number of items that arrive until time $t + \Delta^-$. Denote by j the stratum index such that $t - t_j > \Delta^-$ and $t - t_{j+1} \leq \Delta^-$. An estimate $\hat{N}^+$ of N^+ is then given by

$$\hat{N}^+ = \Delta^- \frac{\sum_{i=j+1}^{l} N_i}{t - t_j}.$$

The estimate roughly equals the amount of items that arrived in the last Δ^- time units. The intuition behind this estimator is that the amount of history we use for estimation depends on how far we want to extrapolate into the future. In conjunction with the robustness techniques discussed below, this approach showed a good performance in our experiments.

Whenever a stratum expires, we compute the estimate $\hat{N}^+$ and—based on this estimate—determine the optimum sequence of merges using the algorithm given in Section 4.3. Denote by $\hat{m} \geq 0$ the total number of merges in the resulting sequence and by $\hat{N}_1^+$ the number of items that arrive before the first merge. In general, we now wait for $\hat{N}_1^+$ items to arrive in the stream and then perform a merge operation. Note that the value of $\hat{m}$ ($\hat{N}_1^+$) is a monotonically increasing (decreasing) function of $\hat{N}^+$; we perform the more merges the more items arrive before the expiration of the first stratum. Thus, underestimation may lead to too few merges and overestimation may lead to too many merges. To make MBS robust against estimation errors, we recompute the sequence of merges whenever we observe that the data stream behaves differently than predicted. There are two cases:

- $\hat{m} = 0$: We recompute $\hat{m}$ and $\hat{N}_1^+$ only if more than $\hat{N}^+$ items arrive in the stream, so that a merge may become profitable. This strategy is optimal if $\hat{N}^+ \geq N^+$ but might otherwise lead to a tardy merge.

- $\hat{m} > 0$: Denote by $\hat{t} = \frac{\hat{N}_1^+}{\hat{N}^+}\Delta^-$ the estimated time span until the arrival of the $\hat{N}_1^+$-th item. We recompute the estimates if the $\hat{N}_1^+$-th item does not arrive close to time $t + \hat{t}$. For concreteness, recomputation is triggered if either the $\hat{N}_1^+$-th item arrives before time $t + (1-\epsilon)\hat{t}$ or when fewer than $\hat{N}_1^+$ items arrived at time $t + (1+\epsilon)\hat{t}$, where $0 < \epsilon < 1$ determines the validity interval of the estimate and is usually set to a small value, say 5%.

In our experiments, the variance of the stratum sizes achieved by MBS without a-priori knowledge of N^+ was almost as low as the one achieved by MBS with a-priori knowledge of N^+.

5. EXPERIMENTS

We implemented bounded-space priority sampling with and without replacement (BPSWR/BPSWOR), priority sampling without replacement (PSWOR), Bernoulli sampling and the stratified sampling schemes in Java 1.6. The experiments have been run on a workstation PC with a 3 GHz Intel Pentium 4 processor and 2.5 GB main memory.

Almost all of the experiments have been run on real-world datasets because we felt that synthetic datasets cannot capture the complex distribution of real-world arrival rates. We used two real datasets, which reflect two different types of data streams frequently found in practice. The NETWORK dataset, which contains network traffic data, has a very bursty arrival rate with high short-term peaks. In contrast, the SEARCH dataset contains usage statistics of a search engine and the arrival rate changes slowly; it basically depends on the time of day. These two datasets allowed us to study the influence of the evolution of the arrival rates on the sampling process. The NETWORK dataset has been collected by monitoring one of our web servers for a period of 1 month. The dataset contains $8,430,904$ items, where each item represents a TCP packet and consists of a timestamp (8 bytes), a source IP and port ($4+2$ bytes), a destination IP and port ($4+2$ bytes) and the size of the user data (2 bytes). The SEARCH dataset has been collected in a period of 3 months and contains $36,389,565$ items. Each item consists of a timestamp (8 bytes) and a user id (4 bytes).

For most of our experiments, we do not report the estimation error of a specific estimate derived from the sample but rather give the key characteristics that influence the estimation error of *any* potential estimate. This way, our results are independent of the actual values associated with the items in our datasets. In the case of uniform sampling, the key characteristic is the sample size. Two uniform samples of the same size are identical in distribution, no matter which scheme has been used to compute them. Larger samples inevitably lead to a smaller estimation error. For stratified sampling, the key characteristic is the variance of the stratum sizes. This variance is a direct measure of how close stratification is to equi-depth stratification. A smaller variance typically results in less estimation error.

5.1 Summary of Experimental Results

For uniform sampling, we found that:

- BPSWOR is the method of choice when the available memory is limited and the data stream rate is varying. It then produces larger samples than Bernoulli sampling or PSWOR. Also, BPSWOR is the only scheme that does not require a-priori information about the data stream and guarantees an upper bound on the memory consumption.

- The window size ratio of the current window to both the current and previous window has a significant impact on the sample size of BPSWOR. A small ratio leads to smaller samples, while a large ratio results in larger samples. For a given ratio, the sample size has low variance and is skewed towards larger samples.

- BPSWOR is superior to BPSWR because it is significantly faster and samples without replacement.
- The window size estimate discussed in Section 3.5 has low relative error. The relative error decreases with an increasing sample size.

For stratified sampling, we found that:

- Merge-based stratification leads to significantly lower stratum size variances than equi-width stratification when the data stream is bursty. Both schemes have comparable performance when the data stream rate changes slowly.
- Merge-based stratification seems to be robust to errors in the arrival rate estimate. Results with estimated arrival rates are close to the theoretical optimum.
- When the number of strata is not too large (≤ 32), the overhead of merge-based stratification is low.

5.2 Uniform Sampling, Synthetic Data

In a first experiment, we compared Bernoulli sampling, PSWOR and BPSWOR. Neither Bernoulli sampling nor PSWOR can guarantee an upper bound on the space consumption and—without a-priori knowledge of the stream—it is not possible to parametrize them to only infrequently exceed the space bound. The goal of this experiment is to compare the sample size and space consumption of the three schemes under the assumption that such a parametrization is possible. For this purpose, we generated a synthetic data stream, where each item of the data stream consists of an 8-byte timestamp and 32 bytes of dummy data. To generate the timestamps, we modeled the arrival rate of the stream using a sine curve with a 24h-period, which takes values between $3,000$ and $5,000$ items per hour. We superimposed the probability density function (PDF) of a normal distribution with mean 24 and variance 0.5 on the sine curve; the PDF has been scaled so that it takes a maximum of $30,000$ items per hour. This models real-world scenarios where the peak arrival rate (scaled PDF) is much higher than the average arrival rate (sine curve).

We used the three sampling schemes to maintain a sample from a sliding window of 1 hour length; the window size over time is given in Figure 4a. We used a space budget of 32 kbytes; at most 819 items can be stored in 32 kbytes space. For the sampling schemes, we used parameters $k_{\text{BPSWOR}} = 585$ (number of candidate/test items), $k_{\text{PSWOR}} = 113$ (sample size) and $q_{\text{Bernoulli}} = 0.0276$ (sampling rate). The latter two parameters have been chosen so that the expected space consumption at the peak arrival rate equals 32 kbytes—as discussed above, this parametrization is only possible because we know the behavior of the stream in advance. During the sampling process, we monitored both sample size and space consumption; the results are given in Figure 4b and 4c, respectively.

Bernoulli sampling. The size of the Bernoulli sample follows the size of the window: It fluctuates around ≈ 110 items in the average case but stays close to the 819 items at peak times. The space consumption of the sample is proportional to the sample size; a large fraction of the available space remains unused in the average case.

Priority sampling. PSWOR produces a constant sample size of 113 items. The space consumption has a logarithmic dependence of the size of the window because—in addition to the sample items—PSWOR also stores the replacement set and the priority of each item.

Bounded priority sampling. BPSWOR produces a sample size of ≈ 300 items in the average case and therefore has a much better space utilization than Bernoulli sampling and PSWOR. When the peak arrives, the sample size first grows above, then falls below the 300-item average. Afterwards it stabilizes again. By Theorem 3, the sample size depends on the ratio of the number of items in the current window to the number of items in both the current and previous window together. This fraction is roughly constant in the average case but varies with the arrival of the peak load. Interestingly, the scheme almost always uses the entire available memory to store the candidate items and the test items. The space consumption slightly decreases when the peak arrives. In this case, we store fewer than k test items because—due to the increased arrival rate—candidate items are replaced by new items before their expiration and so do not become test items.

To summarize, each of the three schemes has a distinctive advantage: Bernoulli sampling does not have any memory overhead, PSWOR guarantees a fixed sample size and BPSWOR samples in bounded space. If the available memory is limited, BPSWOR is the method of choice because it produces larger sample sizes than Bernoulli sampling or PSWOR and does not require any a-priori knowledge about the data stream. For these reasons, we do not consider Bernoulli sampling and PSWOR for our real-world experiments.

5.3 Uniform Sampling, Real Data

Next, we ran BPSWR and BPSWOR on our real-world datasets with a window size of one hour. We monitored the sample size, elapsed time and the window-size estimate during the sampling process and recorded the respective values at every full hour. We did not record more frequently so as to minimize the correlation between the measurements. The experiment was repeated with space budgets ranging from 1 kbyte to 32 kbytes. For each space budget, the experiment was repeated 32 times.

Sample size. In Figure 4d, we report the distribution of the BPSWOR sample size for the NETWORK dataset; similar results were observed with BPSWR. We used a space budget of 32 kbytes, which corresponds to a value of $k = 862$. The figure shows a histogram of the relative frequencies for varying sample sizes. As can be seen, the sample size concentrates around the average of 448 items and varies in the range from 11 to 862 items. The standard deviation of the sample size is 173 and in 95% of the cases, the sample size was larger than 176 items. By Theorem 3, the sample size depends on the ratio of the size of the current window to the size of both the prior and the current window, or the window size ratio for short. In Figure 4e, we give a histogram of the window size ratios in the NETWORK dataset. As can be seen, the distribution of the window size ratio has a striking similarity to the distribution of the sample size. To further investigate this issue, we give a box-and-whisker plot of the sample size for varying ranges of window size ratios in Figure 4f. In a box-and-whisker plot, a box ranging from the first quartile to the third quartile of the distribution is drawn around the median value. From the box, whiskers extend to the minimum and maximum values as long as these values lie

within 1.5 times the interquartile distance (=height of the box); the remaining values are treated as outliers and are directly added to the plot. From the figure, it becomes evident that the window size ratio has a significant influence on the sample size. Also, for each window size ratio, the sample size has low variance and is skewed towards larger samples. The skew results from the fact that the worst-case assumption of Theorem 3 does not always hold in practice; if it does not hold, the sample size is larger.

In Figures 4g, 4h and 4i, we give the corresponding results for the SEARCH dataset. Since the items in the SEARCH dataset require less space than the NETWORK items, a larger value of $k = 1170$ was chosen. As can be seen in the figure, the sample size distribution is much tighter because the arrival rate in the dataset does not vary as rapidly. The sample size ranges from 0 items to 1170 items, where a value of 0 has only been observed when the window was actually empty. The samples size averages to 579 items and is larger than 447 items in 95% of the cases.

Performance. In Figure 4j, we compare the performance of BPSWR and BPSWOR for various space budgets on the NETWORK dataset. The figure shows the average time in milliseconds required to process a single item. It has logarithmic axes. For both algorithms, the per-item processing time increases with an increasing space budget, but BPSWOR is significantly more efficient than BPSWR. The results verify the theoretical analysis in Section 3.4. Since BPSWOR additionally samples without replacement, it is clearly superior to BPSWR.

Estimation of window size. In a final experiment with uniform sampling, we evaluated the accuracy and precision of the window size estimator given in Section 3.5 in terms of its relative error; the relative error of an estimate $\hat{N}$ of N is defined as $|\hat{N} - N|/N$. Figure 4k and 4l display the distribution of the relative error for the NETWORK and SEARCH dataset, respectively, in a kernel-density plot. The relative error is given for memory budgets of 32 kbytes, 64 kbytes and 128 kbytes for the entire sample; only the priorities are actually used for window size estimation. For both datasets and all sample sizes, the relative error almost always lies below 10% and often is much lower. As the memory budget and thus the value of k increases, the estimation error decreases; see [3] for a detailed discussion of this behavior. We conclude that our window size estimator produces low-error estimates and can be used when synopses specialized on window size estimation are unavailable.

5.4 Stratified Sampling

In the next set of experiments, we compared equi-width stratification with merge-based stratification (MBS). Recall that during the sampling process, MBS occasionally requires an estimate of the number of items that arrive until the expiration of the first stratum. To quantify the impact of estimation, we considered two versions of MBS in our experiments. MBS-N makes use of an "oracle": Whenever an estimate of the number of arriving items is required, we determine the exact number directly from the dataset so that no estimation error occurs. MBS-N can therefore be seen as the theoretical optimum of merge-based stratification. In contrast, MBS-$\hat{N}$ uses the estimation technique and robustness modifications as described in Section 4.4. The experimental setup is identical to the one used for uniform sampling, that is, we sample from the real-world datasets over a sliding window of 1 hour length. Unless stated otherwise, we used a space budget of 32 kbytes and $l = 32$ strata.

Variance of stratum sizes. We first compared the variance of the stratum sizes. In order to facilitate a meaningful variance comparison for windows of varying size, we report the coefficient of variation (CV) instead of the stratum size variance directly. The CV is defined as the standard deviation (square root of variance) normalized by the mean stratum size; a value less than 1 indicates a low-variance distribution, whereas a value larger than 1 is often considered high variance. Figure 4m displays the distribution of the CV for the NETWORK dataset using a kernel-density plot. As can be seen, equi-width stratification leads to high values of the CV, while merge-based stratification produces significantly better results. Also, MBS-N and MBS-$\hat{N}$ perform similarly, with MBS-N being slightly superior. The difference between equi-width stratification and the MBS schemes is contributed to the burstiness of the NETWORK stream in which the arrival rates vary significantly during a window length. In contrast, Figure 4n shows the distribution of the CV for the SEARCH dataset. Since the arrival rates change only slowly, equi-width stratification already produces very good results and the merge-based schemes essentially never decide to merge two adjacent strata. The three schemes produce almost identical results. Therefore, merge-based stratification is the more beneficial the more bursty the data stream is.

Accuracy of estimate (example). In a next experiment, we used the stratified sampling schemes to estimate the throughput of the NETWORK data from the sample. Here, we defined the throughput as the sum of the user-data size attribute over the entire window (see the CQL query given in the introduction). Figure 4o gives the distribution of the relative error of the estimate. The estimates derived from the merge-based schemes have a significantly lower estimation error than the estimates achieved with equi-width stratification. Thus, intelligent stratification indeed improves the quality of the sample. Note that for the SEARCH dataset, the distribution of the relative error would be almost indistinguishable for the three schemes because for this dataset, merge-based stratification does not improve upon equi-width stratification.

Number of strata (Example). The number l of strata can have a significant influence on the quality of the estimates. In Table 1, we give the average of the relative error (ARE) of the NETWORK throughput estimate for a varying number of strata. With an increasing number of strata, the ARE increases for equi-width stratification but decreases for the merge-based schemes. On the one hand, the sample size per stratum decreases as l increases and it becomes more and more important to distribute the strata evenly across the window. In fact, when the number of strata was high, equi-width stratification frequently produced empty strata and thereby wasted some of the available space. On the other hand, a large number of strata better exploits the correlations between time and the attribute of interest. Thus, the estimation error often decreases with an increasing value of l. In our experiment, the correlation of the user-data size attribute and time is low, so that the decrease in estimation error is also relatively low.

Performance. In a final experiment, we measured the average per-item processing time for the three schemes and a varying number of strata. The results for the NETWORK

ARE	4	8	16	32	64
Equi-width	2.31%	2.73%	3.44%	4.42%	5.90%
MBS-N	2.00%	1.83%	1.74%	1.70%	1.72%
MBS-$\hat{N}$	2.04%	1.88%	1.82%	1.76%	1.79%

Time (μs)	4	8	16	32	64
Equi-width	2.27	2.25	2.24	2.22	2.21
MBS-N	2.33	2.36	2.41	3.17	9.68
MBS-$\hat{N}$	2.35	2.38	2.67	4.75	18.44

Table 1: Influence of the number of strata (NETWORK)

data are given in Table 1. Clearly, equi-width stratification is the most efficient technique and the processing time does not depend upon the number of strata. The MBS schemes are slower because they occasionally have to 1) estimate the number of arriving items, 2) determine the optimum stratification and 3) merge adjacent strata. The computational effort increases as the number of strata increases. MBS-N is slightly faster than MBS-$\hat{N}$ because MBS-$\hat{N}$ reevaluates 2) if the stream behaves differently than predicted. In comparison to equi-width stratification, MBS leads to a significant performance overhead if the number of strata is large. However, when the number of strata is not too large ($l \leq 32$), the overhead is low but the quality of the resulting stratification might increase significantly.

6. CONCLUSION

We have studied bounded-space techniques for maintaining uniform and stratified samples over a time-based sliding window of a data stream. For uniform sampling, we have shown that any bounded-space sampling scheme that guarantees a lower bound on the sample size requires expected space logarithmic to the number of items in the window; the worst-case space consumption is at least as large. Our provably correct BPS scheme is the first bounded-space sampling scheme for time-based sliding windows. We have shown how BPS can be extended to efficiently sample without replacement and developed a low-variance estimator for the number of items in the window. The sample size produced by BPS is stable in general, but quick changes of the arrival rate might lead to temporarily smaller or larger samples.

For stratified sampling, we have shown how the sample can be distributed evenly across the window by merging adjacent strata from time to time. The decision of when and which strata to merge is based on a dynamic programming algorithm, which uses an estimate of the arrival rate to determine the best achievable stratum boundaries. MBS is robust against estimation errors and produces significantly more balanced samples than equi-width stratification. We found that the overhead of MBS is small as long as the number of strata is not too large. Especially for bursty data streams, the increased precision of the estimates derived from the sample compensates for the overhead in computational cost.

Repeatability Assessment Result

All the results in this paper were verified by the SIGMOD repeatability committee. Code and/or data used in the paper are available at `http://www.sigmod.org/codearchive/sigmod2008/`.

7. REFERENCES

[1] Charu C. Aggarwal. On biased reservoir sampling in the presence of stream evolution. In *Proc. VLDB*, pages 607–618, 2006.

[2] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *Proc. SODA*, pages 633–634, 2002.

[3] Kevin Beyer, Peter J. Haas, Berthold Reinwald, Yannis Sismanis, and Rainer Gemulla. On synopses for distinct-value estimation under multiset operations. In *Proc. SIGMOD*, pages 199–210, 2007.

[4] Paul G. Brown and Peter J. Haas. Techniques for warehousing of sample data. In *Proc. ICDE*, 2006.

[5] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. Towards estimation error guarantees for distinct values. In *Proc. PODS*, pages 268–279, 2000.

[6] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002.

[7] Rainer Gemulla, Wolfgang Lehner, and Peter J. Haas. Maintaining bounded-size sample synopses of evolving datasets. *The VLDB Journal*, 17(2):173–201, 2007.

[8] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. Fast incremental maintenance of approximate histograms. In *Proc. VLDB*, pages 466–475, 1997.

[9] Peter J. Haas. Data stream sampling: Basic techniques and results. In *Data Stream Management: Processing High Speed Data Streams*. Springer, 2006.

[10] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *Proc. VLDB*, pages 275–286, 1998.

[11] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Proc. SenSys*, pages 250–262, 2004.

[12] Carl-Erik Särndal, Bengt Swensson, and Jan Wretman. *Model Assisted Survey Sampling*. Springer Series in Statistics. Springer, 1991.

[13] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[14] Jeffrey Scott Vitter. Faster methods for random sampling. *Commun. ACM*, 27(7):703–718, 1984.

[15] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM TOMS*, 11(1):37–57, 1985.

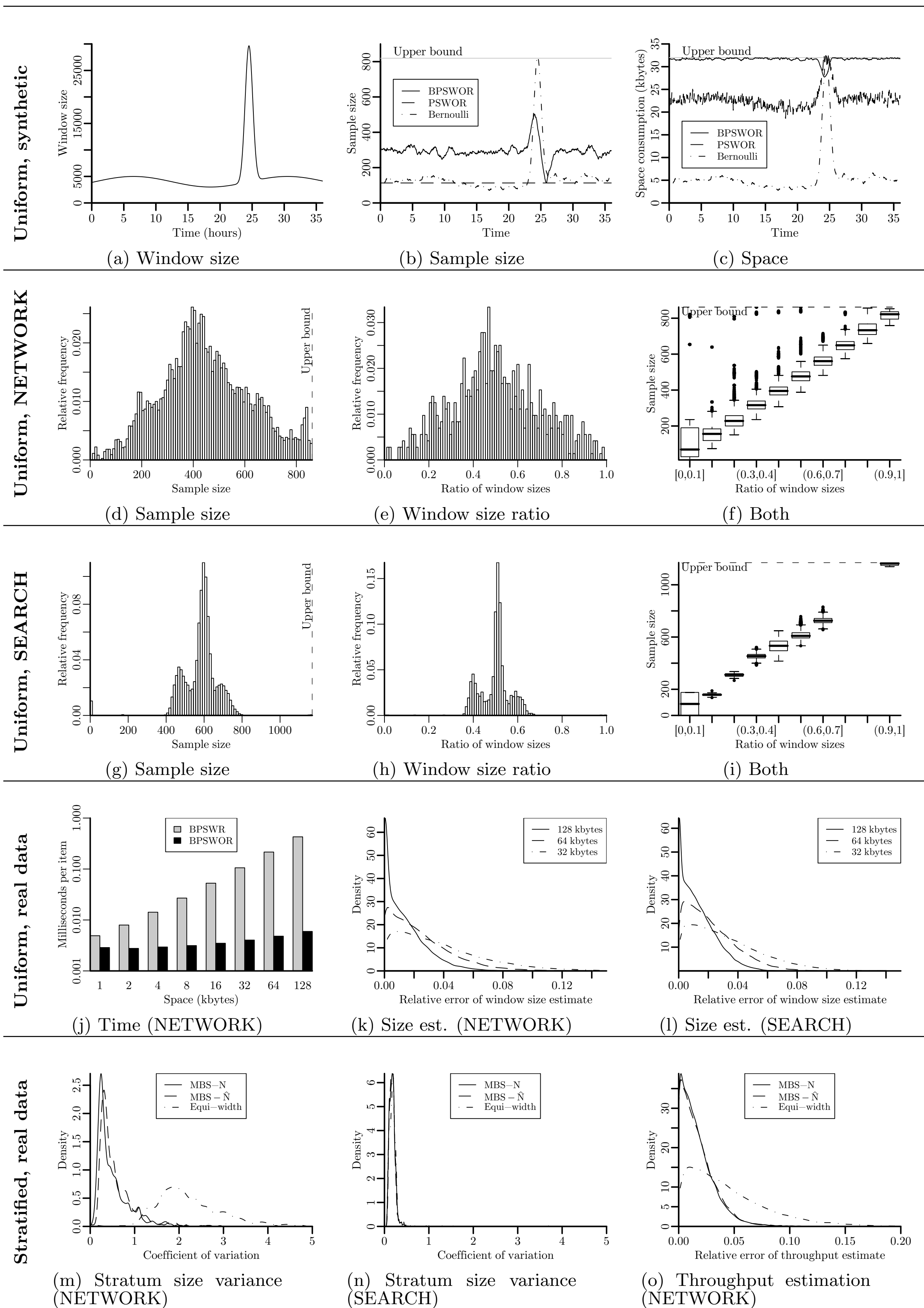

Figure 4: Experimental results (see subheadings on the left hand side)