Rainer Gemulla, Wolfgang Lehner, Peter J. Haas

Maintaining bounded-size sample synopses of evolving datasets

Erstveröffentlichung in / First published in:

The VLDB Journal. 2008, 17, S. 173-201 [Zugriff am: 17.11.2022]. Springer. ISSN 0949-877X.

DOI: https://doi.org/10.1007/s00778-007-0065-y

Maintaining bounded-size sample synopses of evolving datasets

Rainer Gemulla · Wolfgang Lehner · Peter J. Haas

Abstract Perhaps the most flexible synopsis of a database is a uniform random sample of the data; such samples are widely used to speed up processing of analytic queries and data-mining tasks, enhance query optimization, and facilitate information integration. The ability to bound the maximum size of a sample can be very convenient from a system-design point of view, because the task of memory management is simplified, especially when many samples are maintained simultaneously. In this paper, we study methods for incrementally maintaining a bounded-size uniform random sample of the items in a dataset in the presence of an arbitrary sequence of insertions and deletions. For "stable" datasets whose size remains roughly constant over time, we provide a novel sampling scheme, called "random pairing" (RP), that maintains a bounded-size uniform sample by using newly inserted data items to compensate for previous deletions. The RP algorithm is the first extension of the 45-year-old reservoir sampling algorithm to handle deletions; RP reduces to the "passive" algorithm of Babcock et al. when the insertions and deletions correspond to a moving window over a data stream. Experiments show that, when dataset-size fluctuations over time are not too extreme, RP is the algorithm of choice with respect to speed and sample-size stability. For "growing" datasets, we consider algorithms for periodically resizing a bounded-size random sample upwards. We prove that any such algorithm cannot avoid accessing the base data, and provide a novel resizing algorithm that minimizes the time needed to increase the sample size. We also show how to merge uniform samples from disjoint datasets to obtain a uniform sample of the union of the datasets; the merged sample can be incrementally maintained. Our new RPMerge algorithm extends the HRMerge algorithm of Brown and Haas to effectively deal with deletions, thereby facilitating efficient parallel sampling.

Keywords Database sampling · Reservoir sampling · Sample maintenance · Synopsis

R. Gemulla (✉) · W. Lehner
Technische Universität Dresden, 01062 Dresden, Germany
e-mail: gemulla@inf.tu-dresden.de

W. Lehner
e-mail: lehner@inf.tu-dresden.de

P. J. Haas
IBM Almaden Research Center, San Jose, CA, USA
e-mail: phaas@us.ibm.com

1 Introduction

Because of its flexibility, sampling is widely used for quick approximate query answering [1,5,13,14,17,20,35], statistics estimation [15,37], data stream processing [18,23,43], data mining [3,22,24,27], and data integration [2,19,21,31]. Uniform random sampling, in which all samples of the same size are equally likely, is the most basic of the available database sampling schemes. Uniform sampling is ubiquitous in applications: most statistical estimators—as well as the confidence-bound formulas for these estimators—assume an underlying uniform sample. Thus uniformity is a must if it is not known in advance how the sample will be used. Uniform sampling is also a building block for more complex sampling schemes, such as stratified sampling. Methods for producing uniform samples are therefore key to modern database systems.

In the simplest setting, the basic task is to compute a uniform sample from a dataset that is stored on disk, such as a table in a relational database management system (RDBMS) or a repository of XML documents. In general, there are two alternative approaches to computing such a sample. First, the sample may be materialized on the fly as it is needed. In the setting of commercial RDBMS, Haas and König [17] have shown that there is a trade-off between the uniformity of a sampling scheme and the cost of computing it. Even if some degree of non-uniformity is acceptable, online sample materialization can still be too expensive. Moreover, it is often quite acceptable to use the same sample several times, in order to answer a set of queries or perform multiple analysis tasks. Taking advantage of this fact, an alternative approach [4,10,14,15,23] amortizes the cost of sampling over multiple uses by initially materializing a sample from a dataset and then incrementally maintaining the "sample synopsis" over time.

Incremental sample maintenance is a powerful technique, because the abstract notion of the underlying "dataset" can be interpreted very broadly in applications. Indeed, the dataset can actually be an arbitrary view, e.g., over the result of an arbitrary SQL query. Samples over views are particularly good candidates for incremental maintenance, because producing such samples on the fly can require very expensive base-data accesses. For example, most relational operators are not interchangeable with sampling [5,35], so that in most cases the sampling operator cannot be pushed down to the leaves of the query tree. Olken [35] and Olken and Rotem [36] pioneered methods for incremental maintenance of sample views in relational databases; these methods synthesize traditional view-maintenance techniques with database sampling algorithms. The idea is to, in effect, compute the "delta" (set of insertions, updates, and deletions) to the view as the underlying tables are updated and then apply general sample-maintenance methods to the resulting sequence of view modifications. Although computation of the deltas requires access to the base data, this expense is smoothly spread out over time, providing for fast query response. Also observe that the full view need never be materialized if only the sample is of interest, thereby saving space as well as time. The main deficiency of existing techniques for maintaining sample views is that they require expensive base-data accesses over and above those needed to compute the deltas.

In the context of incremental maintenance, ensuring that the sample size remains bounded at all times can be useful from a system-design point of view. Specifically, bounding the sample sizes a priori simplifies the task of memory management by avoiding unexpected overflows and expensive memory reallocation tasks; such simplification is particularly desirable when many such samples are being maintained simultaneously, as in the sample-warehouse setting of [4] or in a data-stream management system. Moreover, bounded-size sampling schemes are the method of choice for applications that need to guarantee a hard upper bound on the response time of queries over the sample. Finally, it is of theoretical interest to study such bounded-size schemes, since they are the subject of a large portion of the sampling literature. In this paper, we therefore restrict our attention to bounded-size sampling schemes; a short discussion of unbounded sampling schemes is given in Sect. 2. A key goal for a bounded-size scheme is to keep the sample size as close to the upper bound as possible, thereby minimizing wasted space and maximizing the stability and accuracy over time of estimates based on the sample.

This paper, an expanded version of [11], provides new methods for incrementally maintaining a bounded-size uniform random sample of an evolving dataset. We assume that the sample-maintenance component scans a stream of update, deletion, and insertion (UDI) transactions[1] on the dataset, and maintains the sample locally. In this setting, the main challenges in sample maintenance are (i) to enforce statistical uniformity in the presence of arbitrary insertions and deletions, (ii) to avoid accesses to the base data to the extent possible, because such accesses are typically expensive, and (iii) to maximize sampling *efficiency*, i.e., to keep the sample size as close to the upper bound as possible. We assume throughout that the sample fits in main memory. This assumption limits the applicability of our techniques in warehousing scenarios where the dataset size is so large, and the sampling rate so high, that the samples must be stored on disk. Extending our results to large disk-based samples is a topic for future research; see Sect. 7 for some tentative ideas in this direction. We also assume that an index is maintained on the sample, in order to rapidly determine whether a given item is present in the sample or not—such an index is mandatory for any implementation of sampling schemes subject to deletions.

We distinguish between "stable" datasets whose size (but not necessarily composition) remains roughly constant over time and "growing" datasets in which insertions occur more frequently than deletions over the long run. The former setting is typical of transactional database systems and databases of moving objects; the latter setting is typical of data warehouses in which historical data accumulates. As discussed above, our focus is on bounded-size sampling schemes. For stable datasets, the upper bound is usually constant, but for growing datasets, keeping the sample size below a bound that is fixed for all time is of limited practical interest. Over time, such a sample represents an increasingly small fraction of the dataset. Although a diminishing sampling fraction may not be a problem for tasks such as estimating a population sum, many other tasks—such as estimating the number of distinct values of a specified population attribute—require the sampling fraction to be bounded from below. The goal for a growing dataset is therefore to grow the sample in a stable and efficient manner, guaranteeing an upper bound

[1] We do not actually consider updates explicitly, since an update to the dataset can be trivially handled by updating the value of the corresponding sample element, if present.

on the sample size at all times and using the allotted space efficiently.

The best known method for incrementally maintaining a bounded-size uniform sample in the presence of a stream of insertions to the dataset is the classical "reservoir sampling" algorithm [8,28,33], which maintains a simple random sample of a specified size. One deficiency of this method is that it cannot handle deletions, and the most obvious modifications for handling deletions either yield procedures for which the sample size systematically shrinks to 0 over time or which require expensive base-data accesses.[2] The other main deficiency is that the class of pure insertion streams—for which reservoir sampling is designed—results in growing datasets as discussed above; thus the usefulness of the bounded reservoir sample tends to diminish over time. Surprisingly, although reservoir sampling has been around for 45 years, the algorithm apparently has never been extended to deal with either deletions or growing datasets. In this paper we provide the first such extensions of reservoir sampling.

In more detail, we address the challenges of incremental sample maintenance as follows:

1. For stable datasets, we provide a new sampling scheme, called "random pairing" (RP), that maintains a bounded-size uniform sample in the presence of arbitrary insertions and deletions, without requiring expensive base-data accesses; indeed, RP is the first bounded-size uniform sampling scheme having both of these characteristics. RP can be viewed as a generalization of both classical reservoir sampling and the "passive" stream-sampling algorithm of Babcock et al. [1]. RP is faster than all other known bounded-size schemes, because it is the only algorithm that never accesses the base data. Provided that fluctuations in the dataset size are not too extreme, the sample sizes produced by RP are as stable as those produced by expensive algorithms that require base-data accesses. Thus, if the dataset size is reasonably stable over time, RP is the algorithm of choice for incrementally maintaining a bounded uniform sample.
2. For growing datasets, we initiate the study of algorithms for periodically "resizing" a bounded-size random sample upwards, proving that any such algorithm cannot avoid accessing the base data. Prior to the current work, the only proposed approach to the resizing problem was to naively recompute the sample from scratch. We provide a novel resizing algorithm that partially enlarges the sample using the base data, and subsequently completes the resizing using only the stream of UDI transactions. Especially when access to the base data is expensive and transactions are frequent, the resizing cost can be significantly reduced relative to the naive approach by judiciously tuning the key algorithm parameter q; this parameter controls the trade-off between the time required to access the base data and the time needed to subsequently enlarge the sample using newly inserted data. We provide both a Monte Carlo-based numerical method and a quick approximate technique for choosing an optimal value of q. The numerical method serves to validate our quick approximate technique, as well as to provide a means of extending our methodology to handle more complex sampling scenarios where approximate tuning techniques may not be available.
3. For a dataset that is partitioned over several nodes, we show how to obtain a sample of the complete dataset from local samples maintained at each node, thereby facilitating efficient parallel sampling. Our new RPMerge algorithm extends the HRMerge algorithm in [4]—which was developed for an insertion-only environment—to effectively deal with deletions.

[2] A common approach is to periodically recompute the sample from scratch [15].

The remainder of this paper is organized as follows. In Sect. 2, we review existing algorithms that are pertinent to incremental sample maintenance and relate them to our new techniques. Section 3 contains a description and correctness proof of the RP algorithm. Section 4 describes our resizing algorithm and develops methods for tuning the key algorithm parameter. We then show in Sect. 5 how samples can be merged to obtain a uniform sample of the union of their datasets. In Sect. 6, we report results from an empirical performance study of the new and existing sample-maintenance algorithms; we also assess the accuracy of our approximate cost model for the resizing algorithm and compare our new merging algorithm to previous techniques. Section 7 contains our conclusions.

2 Uniform sampling schemes

In this section, we describe the sampling problem more precisely and give an overview of various new and existing sampling schemes. Following [4], call a sampling scheme *uniform* if the probability $p_R(S)$ that the scheme produces sample S when applied to dataset R satisfies $p_R(S) = p_R(S')$ whenever $|S| = |S'|$. That is, all samples of the same size are equally likely to be produced. We say that S "is a uniform sample from R" if S is produced from R using a uniform sampling scheme. We restrict attention to sampling without replacement; in general, a without-replacement sample contains more statistical information about the dataset than a with-replacement sample of the same size [41].

We focus throughout on *set-based sampling*; that is, at each time point, the dataset R of interest is a finite subset of a (possibly infinite) set $\mathcal{T} = \{t_1, t_2, \ldots\}$ of unique, distinguishable *items*, and the sample S is, in turn, a subset of R. For example, $\mathcal{T}$ might correspond to a finite set of IP addresses, an infinite sequence of unique text or XML documents, or perhaps a set of relational tuples, each having its own unique identifier.[3] Without loss of generality, we assume throughout that the dataset R is initially empty, and evolves over time as items are inserted and deleted. In general, items that are deleted may be subsequently re-inserted. Thus we consider an infinite sequence of transactions $\gamma = (\gamma_1, \gamma_2, \ldots)$, where each transaction γ_i is either of the form $+t_k$, which corresponds to the insertion of item t_k into R, or of the form $-t_k$, which corresponds to the deletion of item t_k from R. We restrict attention to "feasible" sequences such that (i) at any time point, an item appears at most once in the dataset (so that the dataset is a true set and not a multiset) and (ii) $\gamma_n = -t_k$ only if item t_k is in the dataset just prior to the processing of the nth transaction. Our goal is to ensure that, after each transaction is processed, S is a uniform sample from R. We assume throughout that, as is usual in practice, the sequence γ of insertions and deletions to the data is oblivious to the behavior of the sampling algorithm.

We first discuss two classical sampling schemes, Bernoulli sampling and reservoir sampling, which underlie all of the other sampling methods. We then discuss sampling methods that are appropriate for stable datasets and growing datasets, respectively. Finally, we discuss some recent related work on "distinct-value" (DV) sampling.

2.1 Two classical schemes

The classical Bernoulli and reservoir schemes were originally developed to deal with a sequence of insertion transactions, and are described below. The extension of Bernoulli sampling to handle deletions results in the MBERN scheme (Sect. 2.2), and the extension of reservoir sampling results in our new random-pairing algorithm (Sect. 3).

Bernoulli sampling: In the Bernoulli sampling scheme with sampling rate q, denoted BERN(q), each inserted item is included in the sample with probability q and excluded with probability $1 - q$, independent of the other items. For a dataset R, the sample size follows the binomial distribution BINOM($|R|, q$), so that $P\{|S| = k\} = \binom{|R|}{k} q^k (1-q)^{|R|-k}$ for $k = 0, 1, \ldots, |R|$ and $E[|S|] = q|R|$. Although the sample size is random, samples having the same size are equally likely, and the scheme is indeed uniform, as defined previously. The main advantages of Bernoulli sampling are simplicity and ease of parallelization. The sample size is unbounded, though sharply concentrated around the expected value of $q|R|$, and the method does not directly handle deletions.

Reservoir sampling (*RS*) : This uniform scheme maintains a random sample of fixed size M, given a sequence of insertions. The procedure, as described in [33], is as follows. Include the first M items into the sample. For each successive insertion into the dataset, include the inserted item into the sample with probability $M/|R|$, where $|R|$ is the size of the dataset just after the insertion; an included item replaces a randomly selected item in the sample. Vitter [45] significantly reduced the computational costs of RS by devising a method to directly generate the (random) number of arriving items to skip between consecutive sample inclusions, thereby avoiding the need to "flip a coin" for each item (see Sect. 3.4). Reservoir sampling has also been extended to handle very large disk-based samples [10,23].

2.2 Schemes for stable datasets

Modified Bernoulli sampling: This uniform sampling method, denoted MBERN(q), is the simplest scheme for dealing with a stable dataset. The MBERN(q) scheme treats each insertion identically to the ordinary BERN(q) scheme. Whenever an item is removed from the dataset, it is also removed from the sample, if present. As with BERN(q), the sample size is binomially distributed and is $100q\%$ of the dataset size $|R|$ on average. The variance of the sample size is $q(1-q)|R|$ for a fixed value of $|R|$; any fluctuations of $|R|$ over time further augment the variability of the sample size. If $|R|$ is known (at least approximately) beforehand—so that a suitable value of q can be chosen—and remains relatively stable over time, then the variability of the sample size might be acceptable in practice. In this paper, however, we focus on sampling schemes that produce bounded samples and which do not require any a priori knowledge about the dataset. One might attempt to obtain a bounded-size sampling method from MBERN(q) by purging the sample whenever it exceeds a specified upper bound M using Bernoulli subsampling, and then continuing the sampling process with a reduced value of q. In the earlier version of this paper [11], this method was called "Bernoulli sampling with purging" (BSP), and was asserted to be a uniform sampling scheme. In Appendix A, we show that, rather surprisingly, the BSP scheme is not, in fact, uniform, and so we do not consider BSP further.

Stream-sampling methods: Babcock et al. [1] have proposed several sampling schemes for obtaining a fixed-size uniform random sample from a moving window over a data

[3] In contrast, *multiset-based sampling* is concerned with scenarios in which multiple copies of each item may occur in both the dataset R and the sample S, so that both R and S are multisets (i.e., bags). When sampling from multisets, relatively sophisticated techniques are required to handle deletion of items; see [12] for an extension of Bernoulli sampling to multisets.

stream. This setting corresponds to the special case in which each deletion from the dataset is immediately followed by an insertion, and these algorithms do not directly generalize to arbitrary sequences of insertions and deletions. The most pertinent of the algorithms in [1] is the "passive" algorithm. This algorithm first obtains a uniform sample from the initial window. Whenever an item in the sample is deleted from the window, the corresponding newly inserted item takes the place of the deleted item in the sample. The techniques in [4] can be viewed as "approximate" stream-sampling algorithms for use in a warehouse in which "data partitions" are rolled in and out. The idea is to create samples of the data partitions that "shadow" the full partitions as they move through the warehouse. Again, these algorithms do not generalize to arbitrary, item-wise insertions and deletions, but we borrow ideas from the "merging" algorithms in [4] to parallelize the new algorithms in the current paper.

Correlated acceptance-rejection (*CAR*): The CAR algorithm of Olken and Rotem [36] maintains a uniform random sample in the presence of arbitrary insertions and deletions; this method requires access to the base data, however. CAR has been designed for the specific setting where the base data is stored in a table of a relational database. Adapted to our setting, the algorithm is as follows: Whenever an item is inserted into the dataset, CAR generates a random number N from the binomial distribution $\text{BINOM}(M, 1/|R|)$ and replaces N random items of the current sample by N copies of the new item. Therefore, CAR actually maintains a uniform sample with replacement, i.e., each item in the dataset may appear more than once in the sample. Whenever an item is deleted from the dataset, CAR replaces each occurrence of this item by a random item drawn from the population. We obtain the final uniform sample without replacement by removing duplicates; thus the gross sample size must be larger than M to compensate for duplicate removal, and there is no effective lower bound on the sample size.

CAR without replacement (*CARWOR*): This simple variant of the CAR algorithm executes standard RS at each insertion. Whenever an item is deleted from the sample S, CARWOR replaces it by a random item from $R \setminus S$. Although CARWOR maintains the sample size at its largest possible value, the algorithm relies on frequent, expensive accesses to base data.

Reservoir sampling with recomputation (*RSR*): As mentioned previously, RS is designed to deal only with insertions. The simplest modification is to execute RS as usual at each insertion. At each deletion from the dataset, we check whether the item is in the sample; if so, we remove it and continue RS with a smaller sample size. The obvious problem with this approach is that the sample size decreases monotonically to zero. We therefore modify this approach using a device as in Gibbons et al. [15]: as soon as the sample size falls below a prespecified lower bound, recompute it from scratch using, for example, sequential sampling [44]. This approach is also called the "backing sample" method. Clearly, RSR does not yield a stable sample size, and it requires repeated access to the base data. In spite of these deficiencies, RSR has been the sampling scheme of choice for bounded-size uniform sampling.

Random pairing (*RP*): Our new RP algorithm, described in Sect. 3, maintains a bounded-size uniform sample in the presence of arbitrary insertions and deletions without requiring access to the base data. In contrast to the sampling schemes above, RP compensates sample deletions using subsequent insertions; if, at any time point, all previous sample deletions have been compensated, then the sample size is as large as it can possibly be. As shown in our experiments, the RP algorithm produces samples almost as large as the algorithms above, but at a much lower cost.

2.3 Schemes for growing datasets

Modified Bernoulli sampling: The MBERN(q) sampling scheme as presented above can naturally be applied to a growing dataset. As discussed previously, the user cannot bound the sample size and, as discussed in Appendix A, subsampling cannot be applied to deal with oversized samples.

Resizing: Our novel resizing method can be used with any bounded-size sampling scheme. In this way we can grow the sample as the dataset grows, while guaranteeing an upper bound on the sample size at each time point. Resizing can even be used in conjunction with MBERN(q) sampling to increase the sampling rate q; see Sect. 4.2. Note that in contrast to MBERN(q), the combination of RP and our resizing algorithm enables a user or application to decide exactly when to allocate more storage for the sample, and also precisely how much more memory to allocate.

2.4 Distinct-value sampling

To our knowledge, the only other bounded-size uniform sampling methods that handle arbitrary sequences of insertions and deletions are the two DV-sampling algorithms recently proposed in [7,9]. These algorithms were designed for sampling from multisets; the algorithms sample uniformly from the set of distinct items of a dataset, and also provide the number of occurrences for each sampled value (or a high-accuracy approximation thereof). In our setting, where each item can occur only once in the dataset, random sampling and DV-sampling coincide, so we could attempt to adapt the DV-sampling algorithms to our purpose.

Both DV-sampling algorithms make use of a data structure which—with some success probability p—maintains a *single* item chosen randomly from the set of distinct items. To maintain multi-item samples, multiple instances of the data structure are stored, so that sampling is with replacement. Each data structure consists of $\log|\mathcal{T}|$ buckets, where $\mathcal{T}$ is the domain of the items in the dataset. The idea is that each inserted or deleted item affects exactly one of the buckets in a data structure; a (randomly chosen) hash function ensures that each item maps to the same bucket whenever it occurs in the transaction sequence. In more detail, an item is hashed to bucket i with probability $(1-r)r^{i-1}$, where $0 < r < 1$ is a parameter of the algorithm. Each bucket then consists of the *sum* of the item values inserted into it and a *counter* of the number of inserted items. Adding an item to (resp., deleting an item from) a bucket simply involves incrementing (resp., decrementing) the counter by 1 and incrementing (resp., decrementing) the sum by the item value. If there is a bucket in the data structure which contains exactly one item, then the data structure "succeeds," and this item is returned as a random sample of size 1; otherwise, the data structure fails. The point is that the stored item values do not need to be maintained individually; in the important case where a bucket contains only a single item (as indicated by a counter value of 1), the value of the sum is equal to the value of the item. Note that the lower-numbered buckets are more likely to succeed when the dataset size is small; the higher-numbered buckets handle large datasets. It has been shown [7] that for an appropriate choice of r, the success probability p is at least 14.2%.

The DV-sampling schemes are the only known bounded-size sampling methods that are "delete-proof," in that the sample-size distribution depends only on the size of the dataset, regardless of how the sample was produced. Indeed, each deletion of an item precisely cancels the effect of the prior insertion of the item. In contrast, our RP scheme works the other way around: each insertion compensates a prior deletion, and the sample size distribution depends on both the dataset size and the number of uncompensated deletions. The main disadvantage of the DV schemes is their very low space efficiency. For example, with 32-bit items and a choice of $r = \sqrt{2/3}$ as suggested in [7], we need more than 100 buckets per data structure, and thus exploit at most 1% of the memory allocated to the sample.

For stable datasets, the original DV-sampling scheme can be modified to significantly improve its space efficiency. The modified scheme maintains only a single bucket per data structure and makes use of a hash function with range $\{0, \ldots, D-1\}$, where D is the average size of the dataset. An item t affects a bucket only if the corresponding hash function satisfies $h(t) = 0$. Assuming that the data-set contains exactly D items $t_1, \ldots, t_D$, the probability p of a success is given by

$$\sum_{l=1}^{D} P\left\{ h(t_l)=0,\ h(t_{l'})\neq 0 \text{ for all } l' \neq l \right\} = \left(1 - \frac{1}{D}\right)^{D-1},$$

which is approximately equal to e^{-1} when D is large. Thus the modified scheme would still exploit only about 1/3 of the available memory, at best. Not only is the space efficiency low, but the computational costs are also extremely high. For example, with 1 million copies of the data structure, the DV scheme requires 10 trillion hash operations to insert 10 million items into the sample, which takes hours even with the fastest available hash functions. Schemes such as RP, which are tailored for set-based sampling, can perform such an insertion in a matter of minutes. For these reasons, we focus on the uniform sampling schemes outlined in the previous sections.

3 Random pairing

To motivate the idea behind the random-pairing scheme, we first consider an "obvious" passive algorithm for maintaining a bounded uniform sample S of a dataset R. The algorithm is based on reservoir sampling and avoids accessing base data by making use of new insertions to "compensate" for previous deletions. Whenever an item is deleted from the dataset, it is also deleted from the sample, if present. Whenever the sample size lies at its upper bound M, the algorithm handles insertions identically to RS; whenever the sample size lies below the upper bound and an item is inserted into the dataset, the item is also inserted into the sample. Although simple, this algorithm is unfortunately incorrect, because it fails to guarantee uniformity. To see this, suppose that, at some stage, $|S| = M < |R| = N$. Also suppose that an item t^- is then deleted from the dataset R, directly followed by an insertion of t^+. Denote by S' the sample after these two operations. If the sample is to be truly uniform, then the probability that $t^+ \in S'$ should equal M/N, conditional on $|S| = M$. Since $t^- \in S$ with probability M/N, it follows that

$$\begin{aligned} &P\left\{ t^+ \in S' \right\} \\ &\quad = P\left\{ t^- \in S, t^+ \text{ included} \right\} + P\left\{ t^- \notin S, t^+ \text{ included} \right\} \\ &\quad = \frac{M}{N} \cdot 1 + \left(1 - \frac{M}{N}\right) \cdot \frac{M}{N} > \frac{M}{N}, \end{aligned} \tag{1}$$

conditional on $|S| = M$. Thus an item inserted just after a deletion has an overly high probability of being included in the sample. The basic idea behind RP is to carefully select an inclusion probability for each inserted item so as to ensure uniformity.

3.1 Algorithm description

In the RP scheme, every deletion from the dataset is eventually compensated by a subsequent insertion. At any given time, there are 0 or more "uncompensated" deletions. The RP algorithm maintains a counter c_b that records the number of "bad" uncompensated deletions in which the deleted item was in the sample (so that the deletion also decremented the sample size by 1). The RP algorithm also maintains a counter c_g that records the number of "good" uncompensated deletions in which the deleted item was not in the sample (so that the deletion did not affect the sample). Clearly, $d = c_b + c_g$ is the total number of uncompensated deletions.

The algorithm works as follows. Deletion of an item is handled by removing the item from the sample, if present, and by incrementing the value of c_b or c_g, as appropriate. If $d = 0$, i.e., there are no uncompensated deletions, then insertions are processed as in standard RS. If $d > 0$, then we flip a coin at each insertion step, and include the incoming insertion into the sample with probability $c_b/(c_b + c_g)$; otherwise, we exclude the item from the sample. We then decrease either c_b or c_g, depending on whether the insertion has been included into the sample or not. The complete algorithm is given as Algorithm 1.

Conceptually, whenever an item is inserted and $d > 0$, the item is paired with a randomly selected uncompensated deletion, called the "partner" deletion. The inserted item is included into the sample if its partner was in the sample at the time of its deletion, and excluded otherwise. The probability that the partner was in the sample is $c_b/(c_b+c_g)$. For purposes of sample maintenance, it is not necessary to keep track of the precise identity of the random partner; it suffices to maintain the counters c_b and c_g. Note that if we repeat the calculation in (1) using RP, we now have $P\left\{t^- \notin S,\ t^+ \text{ included}\right\} = 0$, and we obtain the desired result $P\left\{t^+ \in S'\right\} = M/N$.

Typically, a sampling subsystem tracks the size of both the sample and the dataset. If so, then instead of maintaining the two additional counters c_b and c_g, it suffices to maintain a single counter d that records the number of uncompensated deletions. Specifically, set $d \leftarrow 0$ initially. After processing a transaction γ_i, update d as follows:

$$d \leftarrow \begin{cases} d+1 & \text{if } \gamma_i \text{ is a deletion} \\ \max(d-1, 0) & \text{if } \gamma_i \text{ is an insertion.} \end{cases}$$

Then, at any time point, $c_b = \min(M, |R| + d) - |S|$ and $c_g = d - c_b$.

Algorithm 1 Random pairing

```
1: $c_b$: # of uncompensated deletions that have been in the sample
2: $c_g$: # of uncompensated deletions that have not been in the sample
3: $M$: upper bound on sample size
4: $R$, $S$: dataset and sample, respectively
5: RANDOM(): returns a uniform random number between 0 and 1
6:
7: INSERT($t$):
8: if $c_b + c_g = 0$ then                // execute reservoir-sampling step
9:   if $|S| < M$ then
10:     insert $t$ into $S$
11:   else if RANDOM() $< M/(|R| + 1)$ then
12:     overwrite a randomly selected element of $S$ with $t$
13:   end if
14: else                                  // execute random-pairing step
15:   if RANDOM() $< c_b/(c_b + c_g)$ then
16:     $c_b \leftarrow c_b - 1$
17:     insert $t$ into $S$
18:   else
19:     $c_g \leftarrow c_g - 1$
20:   end if
21: end if
22:
23: DELETE($t$):
24: if $t \in S$ then
25:   $c_b \leftarrow c_b + 1$
26:   remove $t$ from $S$
27: else
28:   $c_g \leftarrow c_g + 1$
29: end if
```

3.2 An example

The RP algorithm with $M = 2$ is illustrated in Fig. 1. The figure shows all possible states of the sample, along with the probabilities of the various state transitions. The example starts after $i = 2$ items have been inserted into an empty dataset, i.e., the sample coincides with R. The insertion of item t_3 leads to the execution of a standard RS step since there are no uncompensated deletions. This step has three possible outcomes, each equally likely. Next, we remove items t_2 and t_3 from both the dataset and the sample. Thus, at $i = 5$, there are two uncompensated deletions. The insertion of t_4 triggers the execution of a *pairing step*. Item t_4 is conceptually paired with either t_3 or t_2—these scenarios are denoted by

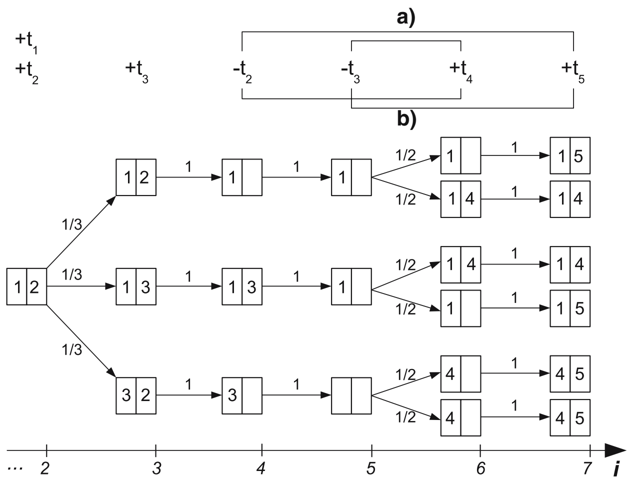

Fig. 1 Random pairing (possible outcomes and probability)

a) and *b*) respectively—and each of these pairings is equally likely. Thus t_4 compensates its partner, and is included in the sample if and only if the partner was in the sample prior to its deletion. This pairing step amounts to including t_4 with probability $c_b/(c_b + c_g)$ and excluding t_4 with probability $c_g/(c_b + c_g)$, where the values of c_b and c_g depend on which path is taken through the tree of possibilities. A pairing step is also executed at the insertion of t_5, but this time there is only one uncompensated deletion left: t_2 in scenario *a*) or t_3 in scenario *b*). The probability of seeing a given sample at a given time point is computed by multiplying the probabilities along the path from the "root" at the far left to the node that represents the sample. Observe that the sampling scheme is indeed uniform: at each time point, all samples of the same size are equally likely to have been materialized.

3.3 Correctness and sample-size properties

In this section, we formally establish the uniformity property of the RP scheme with upper bound M (≥ 1) and then derive the probability distribution, the mean, and the variance of the sample size. To establish uniformity, we actually prove a slightly stronger result that implies uniformity. Denote by R_n the dataset and by S_n sample after the nth processing step, i.e., after processing transaction γ_n. Also denote by $c_{b,n}$ and $c_{g,n}$ the value of the counters c_b and c_g after the nth step, and set $d_n = c_{b,n} + c_{g,n}$. Finally, set $u_n = \min(M, |R_n|)$,

$$v_n = \min\left(M, \max_{1\leq j\leq n} |R_j|\right) = \min(M, |R_n| + d_n), \tag{2}$$

and $l_n = \max(0, v_n - d_n)$. In light of (5) below, it can be seen that u_n and l_n are the largest and smallest possible sample sizes after the nth step, and v_n is the largest sample size attained so far. Without loss of generality, we restrict attention to sequences that start with an insertion into an empty dataset.

Theorem 1 *For any feasible sequence γ of insertions and deletions, there exist numbers $\{ p_n(k) : n \geq 1$ and $k \geq 0 \}$, depending on γ, such that*

$$P\{ S_n = A \} = p_n(|A|) \tag{3}$$

for $A \subseteq R_n$ and $n \geq 1$. Moreover,

$$\frac{p_n(k)}{p_n(k-1)} = \frac{v_n - k + 1}{d_n - v_n + k}. \tag{4}$$

for $n \geq 1$ and $k \in \{ l_n + 1, l_n + 2, \ldots, u_n \}$.

It follows from (3) that, at each step, any two samples of the same size are equally likely to be produced, so that the RP algorithm is indeed a uniform sampling scheme.

Proof Clearly, we can take $p_n(k) = 0$ for $n \geq 1$ and $k \notin \{ l_n, l_n + 1, \ldots, u_n \}$. Fix a sequence of insertions and deletions, and observe that the sample size decreases whenever c_b increases, and increases whenever c_b decreases (subject to the constraint $|S| \leq M$.) It follows directly that

$$c_{b,n} = v_n - |S_n| \tag{5}$$

for $n \geq 1$. The proof now proceeds by induction on n. The assertions of the theorem clearly hold for $n = 1$, so suppose for induction that the assertions hold for values $1, 2, \ldots, n-1$. There are two cases to consider. First, suppose that step n corresponds to the insertion of an item t, and consider a subset $A \subseteq R_n$ with $|A| = k$, where $l_n \leq k \leq u_n$. If $d_{n-1} = 0$, then $d_n = 0$ and $l_n = u_n$, so that (4) holds vacuously, and the correctness proof for standard reservoir sampling—see, e.g., [18]—establishes the assertion in (3). So assume in the following that $d_{n-1} > 0$. If $t \in A$, then, using (5), we have

$$\begin{aligned} P\{ S_n = A \} &= P\{ S_{n-1} = A - \{t\},\ t \text{ included} \} \\ &= p_{n-1}(k-1)\frac{c_{b,n-1}}{d_{n-1}} \\ &= p_{n-1}(k-1)\frac{v_{n-1} - k + 1}{d_{n-1}}. \end{aligned}$$

If $t \notin A$, then

$$\begin{aligned} P\{ S_n = A \} &= P\{ S_{n-1} = A,\ t \text{ ignored} \} \\ &= p_{n-1}(k)\frac{d_{n-1} - v_{n-1} + k}{d_{n-1}}, \end{aligned} \tag{6}$$

so that, if $A \neq \emptyset$, then

$$P\{ S_n = A \} = p_{n-1}(k-1)\frac{v_{n-1} - k + 1}{d_{n-1}}. \tag{7}$$

Here (7) follows from (6) and an inductive application of (4). This establishes the first assertion of the theorem with

$$p_n(k) = \begin{cases} p_{n-1}(k-1)\frac{v_{n-1}-k+1}{d_{n-1}} & \text{if } \max(l_n, 1) \leq k \leq u_n; \\ p_{n-1}(0)\frac{d_{n-1}-v_{n-1}}{d_{n-1}} & \text{if } k = l_n = 0; \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

To establish the second assertion of the theorem, apply (8) and then inductively apply (4), making use of the fact that—since $d_{n-1} > 0$ and an item is inserted at step n—we have $d_n = d_{n-1} - 1$ and $v_n = v_{n-1}$. Now suppose that step n corresponds to the deletion of an item t, and again consider a subset $A \subseteq R_n$ with $|A| = k \in \{ l_n, l_n + 1, \ldots, u_n \}$. Observe that

$$\begin{aligned} P\{ S_n = A \} &= P\{ S_{n-1} = A \} + P\{ S_{n-1} = A \cup \{t\} \} \\ &= p_{n-1}(k) + p_{n-1}(k+1), \end{aligned}$$

which establishes the first assertion of the theorem with

$$p_n(k) = p_{n-1}(k) + p_{n-1}(k+1) \tag{9}$$

for $l_n \le k \le u_n$. Since $d_n = d_{n-1} + 1$ and $v_n = v_{n-1}$, we then have

$$\frac{p_n(k)}{p_n(k-1)} = \frac{p_{n-1}(k) + p_{n-1}(k+1)}{p_{n-1}(k-1) + p_{n-1}(k)}$$

$$= \frac{(p_{n-1}(k)/p_{n-1}(k-1)) + (p_{n-1}(k+1)/p_{n-1}(k-1))}{1 + (p_{n-1}(k)/p_{n-1}(k-1))}$$

$$= \frac{v_{n-1} - k + 1}{d_{n-1} - v_{n-1} + k + 1} = \frac{v_n - k + 1}{d_n - v_n + k}$$

for $l_n < k \le u_n$, where we have again inductively used (4). Thus the second assertion of the theorem holds and the proof is complete. □

Observe that the RP scheme reduces to the "passive" algorithm of [1] if applied to a fixed-width moving window over a data stream. If there are no deletions, the RP scheme reduces to standard RS.

Building on Theorem 1, we can easily derive the statistical properties of the sample size at any given time point. As before, we define u_n and l_n as before to be the largest and smallest possible sample size after processing the nth transaction, and $v_n = \min(M, |R_n| + d_n)$ to be the largest sample size encountered so far.

Theorem 2 *For any feasible sequence γ of insertions and deletions and $n \ge 1$, the sample size follows the hypergeometric distribution given by*

$$P\{\,|S_n| = k\,\} = \binom{|R_n|}{k}\binom{d_n}{v_n - k} \Big/ \binom{|R_n| + d_n}{v_n} \tag{10}$$

for $l_n \le k \le u_n$, and $P\{\,|S_n| = k\,\} = 0$ otherwise. Moreover, the expected value and variance of $|S_n|$ are given by

$$E[|S_n|] = \frac{|R_n|}{|R_n| + d_n} v_n$$

and

$$\mathrm{Var}[|S_n|] = \frac{d_n v_n (|R_n| + d_n - v_n)|R_n|}{(|R_n| + d_n)^2 (|R_n| + d_n - 1)}.$$

Proof Defining $p_n(k)$ as in Theorem 1 and appealing to (3), we have

$$P\{\,|S_n| = k\,\} = \sum_{\substack{A \subseteq R_n \\ |A| = k}} P\{\,S_n = A\,\} = \binom{|R_n|}{k} p_n(k),$$

and it suffices to show that

$$p_n(k) = \binom{d_n}{v_n - k} \Big/ \binom{|R_n| + d_n}{v_n}, \quad l_n \le k \le u_n \tag{11}$$

for $n \ge 1$. Clearly, (11) holds for $n = 1$, and can be established for general $n > 1$ by a straightforward inductive argument that uses (8) and (9). The remaining assertions of the theorem follow from well known properties of the hypergeometric distribution [25, p. 238]. □

The intuition behind the result of the theorem is as follows. Suppose that whenever an item is deleted, we mark it as "deleted," but we retain such a "ghost item" in the dataset (and in the sample, if present) until the deleted item is compensated. After processing the nth transaction, the dataset contains $|R_n| + d_n$ total items, comprising $|R_n|$ real items and d_n ghost items. We can view the current sample as a uniformly selected subset of size v_n from the collection of $|R_n| + d_n$ total items. The actual sample size $|S_n|$ is simply the number of these v_n items that are real. The probability that a uniform sample of v_n items from a population of $|R_n|$ real items and d_n ghost items contains exactly k real items is well known to be given by the hypergeometric probability asserted in the theorem statement.

It can be seen from Theorem 2 that the sample size typically stays relatively close to its expected value. For example, suppose we sample 100, 000 items from a dataset consisting of 10, 000, 000 items (1%). If we delete 100, 000 items, the sample size is 99, 000 in expectation and has a standard deviation of 31.31 items; we have $98{,}900 \le |S| \le 99{,}100$ with a probability of approximately 99.8%. Moreover, when the number of uncompensated deletions is small, the expected sample size is close to its maximum possible value.

Observe that if $d_n = 0$, so that there are no uncompensated deletions, or if $|R_n| + d_n \le M$, so that the sample coincides with the population, then $E[|S|] = \min(M, |R_n|)$, $\mathrm{Var}[|S|] = 0$, and $P\{\,|S| = \min(M, |R_n|)\,\} = 1$, i.e., the sample size is deterministic.

3.4 Reducing the number of calls to Random

The RP algorithm, as displayed in Algorithm 1, calls the Random function at essentially every insertion transaction. In practice, random numbers are generated using a pseudorandom number generator (PRNG); see [30] for an overview of PRNG's. In general, the uniformity property of the sample relies on the statistical quality of the PRNG, and increased quality has its price in terms of processing cost. Because of the frequency with which the basic algorithm calls the Random function, it is worthwhile investigating the possibility of reducing the number of PRNG calls.

Revisiting Algorithm 1, one finds that there are two locations where random numbers are generated. In line 11, random numbers are used to execute plain reservoir sampling and in line 15, random numbers drive the pairing process. In both cases, results of Vitter [44,45] can be leveraged to produce a more efficient algorithm.

Reservoir-sampling step: Assume for a moment that γ consists only of insertion transactions. After the reservoir has been filled initially, RP accepts transaction γ_i with probability $M/i = M/(|R_{i-1}|+1)$ and rejects it otherwise. Suppose that RP is about to process transaction γ_i and denote by K_i the random number of rejected transactions before the next sample inclusion. We have

$$P(K_i = k) = \frac{M}{|R_{i-1}|+k+1}\prod_{j=0}^{k-1}\left(1-\frac{M}{|R_{i-1}|+j+1}\right)$$

for $k \geq 0$, where we take an empty product as 1. As mentioned in Sect. 2.1, Vitter [45] introduced a modified reservoir sampling scheme that exploits efficient acceptance-rejection (AR) methods for directly generating realizations of K_i; see [29, Sect. 8.2.4] for a general discussion of such algorithms. The basic idea is to find another random variable K_i^* that is easy and fast to generate. The AR algorithm starts by generating a realization k^* of K_i^*. With probability $p(k^*)$, this value is "accepted" and the algorithm returns with $K_i = k^*$; with probability $1 - p(k^*)$, the value is "rejected" and the process repeats. The random variable K_i^* and the probability function $p(\cdot)$ are carefully chosen so that (i) conditional on being accepted, the returned value has the same probability distribution as K_i, and (ii) the expected number of rejections before the final acceptance is small. Given such a generation method, the idea is to maintain a counter K for the number of skipped items before the next sample inclusion, initialized to an invalid value (<0). If an insertion transaction $\gamma_i = +t_l$ arrives and K is invalid, then a realization of K_i is generated and assigned to K. Next, if $K > 0$, then t_l is ignored and K is decremented. Otherwise, if $K = 0$, then t_l is included into the sample and K is invalidated. Observe that prior to the sample-inclusion/sample-exclusion decision for any insertion transaction γ_i, we have $P\{K = k\} = P\{K_i = k\}$, so that the algorithm is indeed correct.

The above idea can be transferred to the RP algorithm. In contrast to ordinary reservoir sampling, the insertion process may be interrupted by deletion transactions. However, we may simply continue to use the current value of the skip counter when the next reservoir step is executed. To see this, suppose that there are no uncompensated deletions after processing transaction γ_i and that the $(i+1)$st trans-action is a deletion, so that $d_i = 0$ and $d_{i+1} > 0$. Denote by $i^* > i$ the index of the next transaction after γ_i such that $d_{i^*} = 0$. Since RP does not execute a reservoir step for transactions $\gamma_{i+1}, \ldots, \gamma_{i^*}$, and since $|R_i| = |R_{i^*}|$, we have $P(K_{i+1} = k) = P(K_{i^*+1} = k)$, and the reservoir sampling

Algorithm 2 Random pairing (optimized)

```
1: $c_b$: # of uncompensated deletions that have been in the sample
2: $c_g$: # of uncompensated deletions that have not been in the sample
3: $M$: upper bound on sample size
4: $R$, $S$: dataset and sample, respectively
5: $K$: skip counter for reservoir sampling (initialized to $-1$)
6: $K'$: skip counter for random pairing (initialized to $-1$)
7: SKIPRS: reservoir-sampling skip function as in [45]
8: SKIPSEQ: sequential-sampling skip function as in [44]
9:
10: INSERT($t$):
11: if $c_b + c_g = 0$ then   // execute optimized reservoir-sampling step
12:   if $|S| < M$ then
13:     insert $t$ into $S$
14:   else
15:     if $K < 0$ then
16:       $K \leftarrow$ SKIPRS($M$, $|R| + 1$)
17:     end if
18:     if $K = 0$ then
19:       overwrite a randomly selected element of $S$ with $t$
20:     end if
21:     $K \leftarrow K - 1$
22:   end if
23: else   // execute optimized random-pairing step
24:   if $K' < 0$ then
25:     $K'$ = SKIPSEQ($c_b$, $c_b + c_g$)
26:   end if
27:   if $K' = 0$ then
28:     $c_b \leftarrow c_b - 1$
29:     insert $t$ into $S$
30:   else
31:     $c_g \leftarrow c_g - 1$
32:   end if
33:   $K' \leftarrow K' - 1$
34: end if
35:
36: DELETE($t$):
37: $K' \leftarrow -1$   // invalidate
38: if $t \in S$ then
39:   $c_b \leftarrow c_b + 1$
40:   remove $t$ from $S$
41: else
42:   $c_g \leftarrow c_g + 1$
43: end if
```

process can be continued at the point where it was interrupted. Algorithm 2 incorporates these optimizations into the basic RP algorithm. Here, SKIPRS denotes the reservoir-sampling skip function as described in [45].

Random-pairing step: We can exploit an idea similar to the one above. Assume that after processing transaction γ_i, there is at least one uncompensated deletion and that transactions $\gamma_{i+1}, \gamma_{i+2}, \ldots$ correspond to insertions. Denote by $c_{b,i}$ and $c_{g,i}$ the values of the sample counters after processing transaction γ_i. The item corresponding to insertion transaction γ_{i+1} is included in the sample with probability $c_{b,i}/(c_{b,i} + c_{g,i})$ and excluded otherwise. In the latter case, the next item, corresponding to γ_{i+2}, is included with probability $c_{b,i+1}/(c_{b,i+1} + c_{g,i+1}) = c_{b,i}/(c_{b,i} + c_{g,i} - 1)$, and so on. When RP is about to process transaction γ_{i+1}, let K'_{i+1} be the number of excluded items before the next sample inclusion. We have

$$P(K'_{i+1} = k') = \frac{c_{\mathrm{b},i}}{c_{\mathrm{b},i} + c_{\mathrm{g},i} - k'} \prod_{j=0}^{k'-1} \left(1 - \frac{c_{\mathrm{b},i}}{c_{\mathrm{b},i} + c_{\mathrm{g},i} - j}\right)$$

for $k' \geq 0$. In the context of sequential sampling, Vitter [44] proposed an efficient acceptance-rejection algorithm (different from the one in [45]) to directly generate a realization of K'_i. We make use of these improved methods in the RP algorithm by maintaining a skip counter K'. Unlike with the counter K discussed above, we have to recompute K' whenever a deletion transaction interrupts the insertion process. The complete procedure is given in Algorithm 2. Here, SKIPSEQ denotes the skip function as defined in [44]. Because a call to SKIPSEQ is usually more expensive than a (single) call to RANDOM, we expect this optimization to be worthwhile when the transaction stream comprises long blocks of insertions alternating with long blocks of deletions; see Sect. 6.4 for an empirical evaluation of our proposed optimizations.

4 Resizing samples

The discussion so far has focused on stable datasets, and therefore on sampling algorithms that guarantee a fixed upper bound on the sample size. We now shift attention to growing datasets. As mentioned previously, modified Bernoulli sampling can be used to maintain a sample whose size grows with the dataset, but such a sampling scheme does not control the maximum sample size. We therefore consider the problem of maintaining a sample with an upper bound that is periodically increased according to the user's needs.

4.1 A negative result

The RP algorithm can maintain a bounded sample without needing to access the base data. One might hope that there exist algorithms for resizing a sample that similarly do not need to access the base data. Theorem 3 below shows that such algorithms cannot exist.

In general, we consider algorithms that start with a uniform sample S of size at most M from a dataset R and—after some finite (possibly zero) number of arbitrary transactions on R—produce a uniform sample S' of size M' from the resulting modified dataset R', where $M < M' < |R|$. We allow the algorithms to access the base dataset R. For example, a trivial resizing scheme ignores the transactions altogether, immediately discards S, and creates a fresh sample S' by resampling R.

Algorithm 3 Sample resizing

```
1: $M$: initial sample size
2: $M'$: target sample size ($M' > M$)
3: $q$: Bernoulli sampling parameter
4: $R$: initial dataset
5: $S$: initial sample with $|S| = M$
6:
7: PHASE 1:
8: generate $U$ from BINOM($|R|, q$) distribution
9: if $U \leq M$ then
10:    $S \leftarrow$ uniform subsample of size $U$ from $S$
11:    go to Phase 2
12: else if $M < U < M'$ then
13:    $V \leftarrow$ uniform sample of size $U - M$ from $R \setminus S$
14:    $S \leftarrow S \cup V$
15:    go to Phase 2
16: else if $U \geq M'$ then
17:    $V \leftarrow$ uniform sample of size $M' - M$ from $R \setminus S$
18:    $S \leftarrow S \cup V$
19:    return $S$
20: end if
21:
22: PHASE 2:
23: while $|S| < M'$ do
24:    wait for transaction
25:    if transaction = "insert item" then
26:       insert item into $S$ with probability $q$
27:    else                          // transaction = "delete item"
28:       remove item from $S$ if present
29:    end if
30: end while
31: return $S$
```

Theorem 3 *There exists no resizing algorithm that can avoid accessing the base dataset R.*

Proof Suppose to the contrary that such an algorithm exists, and consider the case in which the transactions on R consist entirely of insertions. Fix a set $A \subseteq R'$ such that $|A| = M'$ and A contains $M + 1$ elements of R; such a set can always be constructed under our assumptions. Because the hypothesized algorithm produces uniform samples of size M' from R', we must have $P\left\{ S' = A \right\} > 0$. But clearly $P\left\{ S' = A \right\} = 0$, since $|S| \leq M$ and, by assumption, no further elements of R have been added to the sample. Thus we have a contradiction, and the result follows. □

4.2 A resizing algorithm

Note that Theorem 3 does not apply when the initial sample is created by the BERN(q) scheme, since this scheme does not produce bounded-size samples. Indeed, we can increase the size of a BERN(q) sample using subsequent insertions only, by simply applying BERN(q) sampling to these insertions. We now exploit this fact to develop a method for resizing a bounded-size sample; this method is given as Algorithm 3.

Suppose that the initial sample size is $|S| = M$ and the target sample size is $M' > M$, where $M, M' < |R|$. The basic idea is as follows. In phase 1, the algorithm converts the sample to a BERN(q) sample, possibly accessing base data in the process; we discuss the choice of q in the following section. In phase 2, the algorithm uses Bernoulli sampling (with deletions allowed) to increase the sample size to the target value M'. At this point, bounded-size sampling resumes, using the new upper bound M'. When base data accesses are expensive—e.g., when items are large or when the dataset is remote—and insertions occur frequently, this approach can be much faster than the traditional approach of recomputing the sample from scratch. In more detail, the algorithm generates a random variable U having a BINOM($|R|, q$) distribution, which represents the initial Bernoulli sample size. The algorithm uses as many items from S as possible to make up the Bernoulli sample, accessing base data only if $U > |S|$. If the initial sample size U exceeds the target size M', then the algorithm simply materializes a sample of size M' from R and terminates, in effect taking an immediate subsample of size M' from a Bernoulli sample of size U. In phase 2, the algorithm increases the sample to the desired size by using MBERN(q) sampling.

More formally, denote by S^* the effective sample at the end of phase 1. Also denote by S_k and R_k ($k \geq 0$) the elements in the sample and the dataset after k transactions have occurred in phase 2. Note that $S_0 = S^*$ if $U < M'$ and S_0 is a size-M' uniform subsample from S^* if $U \geq M'$. Finally, denote by L (≥ 0) the random number of transactions that occur during phase 2. The following theorem asserts the correctness of the resizing algorithm.

Theorem 4 *Given that $k \leq L$, where $k \geq 0$, the set S_k is a uniform sample from R_k.*

Proof (Sketch) As mentioned previously, the distribution of U in phase 1 is identical to the distribution of the sample size of a BERN(q) sample of R. The subsampling step ($U \leq M$) and the union step (otherwise) both maintain the uniformity of S, so that S^* is a BERN(q) sample from R. Using this fact, it can then be shown that every probability of the form $P\{S_k = A \mid k \leq L\}$ with $A \subseteq R_k$ depends on A only through $|A|$, and the desired result follows. For example, when $k \geq 1$ and all phase 2 transactions are insertions, fix a set $A \subseteq R_k$ comprising exactly i elements of R_0 and j elements of $R_k \setminus R_0$, where $i + j < M'$ and $j \leq k$. Then

$$\begin{aligned}&P\{S_k = A, k \leq L\}\\ &\quad = P\{S_k = A\}\\ &\quad = q^i(1-q)^{|R|-i}q^j(1-q)^{k-j} = q^{|A|}(1-q)^{|R|+k-|A|},\end{aligned}$$

and $P\{S_k = A \mid k \leq L\} = P\{S_k = A, k \leq L\}/P\{k \leq L\}$ depends on A only through $|A|$. □

We assume that the dataset is "locked" during phase 1, so that the process of incoming transactions is temporarily suspended. For ease of exposition, we assume that the sample of $R \setminus S$ is obtained using item-by-item sampling techniques as in [35]. Such techniques are applicable in a wide range of settings, and are typically much faster than a scan of the entire dataset. These techniques sample from $R \setminus S$ by first extracting a random item from the dataset. The item is accepted if it is not already in the sample (which is the usual case and is checked via the assumed index on the sample); otherwise, the item is rejected and the process starts over. As discussed in Appendix C, more sophisticated and efficient data-access methods may be available, depending upon the specific system architecture and data layout. Our goal, given cost models for a specified base-data access mechanism and the UDI stream, is to optimally balance the amount of time required to access the base data in phase 1, and the amount of time required to finish growing the sample (using new insertions) in phase 2.

The value of the parameter q determines the relative time required for phases 1 and 2. Intuitively, when base data accesses are expensive but new insertions occur frequently, we might want to choose a low value of q so as to resample as few items as possible and shift most of the work to phase 2. In contrast, when base data accesses are fast with respect to the arrival rate of new insertions, a large value of q might be preferable to minimize the complete resizing time.

As a final observation, if we are using an MBERN(q) sampling scheme to deal with a growing dataset, then we can execute phase 1 with parameter $q' > q$ to transition from MBERN(q) sampling to MBERN(q') sampling. In the following sections, however, we focus on the use of the resizing algorithm with algorithms that produce bounded-size samples.

4.3 Choosing the resizing parameter

This section addresses the key problem of choosing the parameter q in Algorithm 3. For a given choice of q, the resizing cost—i.e., the time required for resizing—is random. Indeed, the time required for phase 1 depends on the value of the binomial random variable U, and the time required for phase 2 depends on both U and on the random decisions made during the course of MBERN(q) sampling. Our goal is therefore to develop a probabilistic model of the resizing process, and choose q to minimize the expected resizing cost.[4] In Sect. 4.3.1, we develop perhaps the simplest possible model of the resizing process, assuming the base-data access

[4] Of course, any practical implementation of the resizing algorithm would estimate the cost of recomputing the sample from scratch, and choose this option if it is less expensive than the cost of the new resizing algorithm under the optimal value of q. In many scenarios, however, complete recomputation will not be the best option.

paradigm described previously and a very simple model of the UDI stream. Even for this simple model, determining q^*, the optimal value of q, is decidedly nontrivial, because the expected cost is extremely difficult, if not impossible, to evaluate analytically. One possibility, which we explore in Sect. 4.3.2, is to determine q^* using a numerical optimization algorithm. Typically, the expected resizing cost for a given value of q is extremely hard or impossible to compute exactly, and must be estimated via MonteCarlo methods. The optimization algorithm must then take the resulting uncertainty of the expected-cost observations into account, which can lead to costly computations; see Appendix B. We therefore develop (Sect. 4.3.3) an approximate model of the cost function that can be minimized analytically. The experiments in Sect. 6 indicate that both the approximate model of expected cost and the resulting choice of q closely agree with the results obtained via numerical methods, thereby justifying the use of the quick approximate analytical method. In Appendix C, we show through several examples how our techniques can be adapted to handle more complicated cost models.

4.3.1 Modeling the resizing process

There are many possible cost models for the resizing process, corresponding to different models of both the transaction stream and the base-data access mechanism. As discussed above, we focus on a relatively simple model and indicate some variations and extensions in Appendix C. Our goal in the current paper is to illustrate a general approach, illuminate the issues involved, and provide some preliminary guidance.

We first consider the cost of phase 1. During this phase, the algorithm obtains $N(U)$ items from $R \setminus S$, where $N(u) = (\min(u, M') - M)^+$ for $u \geq 0$, with $x^+ = \max(x, 0)$. As discussed above, we assume that these items are obtained using repeated simple random sampling from R with replacement, with an acceptance-rejection step to ensure that each newly sampled item is not an element of S and is distinct from all of the items sampled so far. Because of the acceptance-rejection step, the (random) number B_i of base-data accesses required to obtain the ith item has a geometric distribution with failure probability $p_i = (M + i - 1)/|R|$:

$$P\{B_i = n\} = p_i^{n-1}(1 - p_i), \quad n \geq 1.$$

The random variables B_i are mutually independent. Supposing that each base-data access takes t_a time units, the expected phase 1 cost is $t_a E_q[B]$, where $B = B_1 + B_2 + \cdots + B_{N(U)}$. We use the subscript q to emphasize the fact that the probability distribution of B depends on the distribution of U, and hence on the parameter q. We can re-express this expected cost in a more convenient form. Using standard properties of the geometric distribution [25, p. 201] and a change of summation index, we have

$$\begin{aligned} E_q[B \mid U] &= \sum_{i=1}^{N(U)} E[B_i] = \sum_{i=1}^{N(U)} \frac{|R|}{|R| - M - i + 1} \\ &= |R|\, H\left(|R| - M - N(U) + 1, |R| - M\right), \end{aligned} \quad (12)$$

where $H(n, m) = \sum_{i=n}^{m} 1/i$. In (12), the arguments to the H function are always large. Appealing to a well known approximation to the harmonic numbers, we find that $H(n, m) \approx \ln(m/n) \approx \ln(m/(n-1))$ with negligible error whenever $m, n \gg 1$. Thus we can write $E_q[B \mid U] = g(U)$, where

$$g(u) = |R| \ln\left(\frac{|R| - M}{|R| - M - N(u)}\right).$$

By the law of total expectation, we have

$$E_q[B] = E_q\left[E_q[B \mid U]\right] = E_q[g(U)],$$

and we can therefore write the expected phase 1 cost as

$$T_1(q) = t_a E_q[g(U)].$$

There are two advantages to this representation of the expected cost. First, it leads naturally to methods for numerical computation as well as for analytical approximation (see the following two sections). Second, it is easier to estimate the expected cost using Monte Carlo methods, because the random variable $E_q[B \mid U] = g(U)$ has lower variance than the random variable B. This variance reduction is a consequence of the well known variance decomposition

$$\mathrm{Var}[X] = E\left[\mathrm{Var}[X \mid Y]\right] + \mathrm{Var}\left[E[X \mid Y]\right],$$

which implies that $\mathrm{Var}\left[E[X \mid Y]\right] \leq \mathrm{Var}[X]$; the decomposition holds for any two random variables X and Y.

We now consider the cost of phase 2. In this phase, the resizing algorithm executes a random number L of Bernoulli trials until the sample size reaches the target value M'. Clearly, the cost of phase 2 is 0 if $U \geq M'$, since then phase 2 is not executed. To make further progress, we need a model of the insertion and deletion process to handle the usual case in which $U < M'$. The simplest model, which we will use here, is to assume that, during phase 2, a Bernoulli trial occurs every t_b time units; the quantity t_b primarily reflects the time between successive transactions. With probability p, the transaction is an insertion, and with probability $(1 - p)$ the transaction is a deletion. We assume that $p > 1/2$, since the dataset is growing. The parameters t_b and p can easily be estimated from observations of the arrival process. We assume as a boundary condition that when the dataset R is empty, the next transaction is, with probability 1, an insertion.

Under the foregoing assumptions, the evolution of the dataset and sample during phase 2 can be specified as a Markov chain. Specifically, denote by X_n and Y_n the size of the sample and dataset, respectively, after processing the

nth transaction during phase 2 (assuming that $U < M'$, so that phase 2 occurs). Then the stochastic process $\{(X_n, Y_n): n \geq 0\}$ is a discrete-time Markov chain with discrete state space $\mathcal{S} = \{(x, y) \in \mathbb{N} \times \mathbb{N}: x \leq y\}$, where $\mathbb{N}$ denotes the nonnegative integers. The transition probabilities are given by

$$p\,((x, y), (x + 1, y + 1)) = pq,$$
$$p\,((x, y), (x, y + 1)) = p(1 - q),$$
$$p\,((x, y), (x - 1, y - 1)) = (1 - p)x/y,$$

and

$$p\,((x, y), (x, y - 1)) = (1 - p)\,(1 - (x/y))$$

for $y \geq 1$ and $0 \leq x \leq y$, and (at the boundary of the state space) by

$$p\,((0, 0), (1, 1)) = q,$$

and

$$p\,((0, 0), (0, 1)) = (1 - q).$$

The initial state of the chain (again under the assumption that $U < M'$) is given by $(X_0, Y_0) = (U, |R|)$. Set $\Gamma = \{(x, y) \in \mathcal{S}: x = M'\}$ and denote by V the first passage time to Γ:

$$V = \inf\{n \geq 1: (X_n, Y_n) \in \Gamma\}.$$

Then set

$$L = \begin{cases} 0 & \text{if } U \geq M'; \\ V & \text{if } U < M'. \end{cases}$$

Of course, the distribution of both U and V, and hence of L, depends on q. The expected cost of phase 2 can now be written as $T_2(q) = t_b E_q[L]$, and the total resizing cost can be written as $T(q) = T_1(q) + T_2(q) = E_q[C]$, where $C = t_a g(U) + t_b L$. Several variations on this model are briefly outlined in Appendix C.

Although the above model of resizing is relatively simple, it is far from simple to find q^*, the optimal value of q. There is no apparent closed-form expression in q for the expected cost $T(q)$. Moreover, as discussed in the sequel, it appears difficult or impossible to compute $T(q)$ numerically without resorting to simulation, so that computational efficiency becomes very important.

4.3.2 Numerical methods

As mentioned above, the main obstacle to determining q^* is the difficulty of computing $T\,(q) = E_q\,[C] = E_q\,[t_a\, g(U) + t_b\, L]$, the expected resizing cost corresponding to a given choice of q. There does not appear to exist a closed-form expression for $T\,(q)$. The expected cost of phase 1 can be computed numerically without too much difficulty, based on the formula

$$E_q[g(U)] = \sum_{u=0}^{|R|} E_q[B \mid U = u]P\{U = u\}$$
$$= \sum_{u=0}^{|R|} g(u)\binom{|R|}{u} q^u (1 - q)^{|R|-u}, \qquad (13)$$

because only a small number of terms contribute significantly to the sum. Unfortunately, the quantity $E_q[L]$ is not nearly as tractable, because the distribution of L—being a first-passage time for a two-dimensional Markov chain on an infinite state space—is extremely complex. If we truncate the state space, then, in principle, $E_q[L]$ can be obtained as a solution to a system of linear equations; see, e.g., [34, p. 17]. In practice, this system is so large, and has such complex structure (because the chain is two-dimensional), that solving the system is infeasible. As discussed in Sect. 4.3.3 below, an exception occurs for the special case where $p = 1$ (no deletions). For this scenario, we can show that $E_q[L \mid U] = (M' - U)^+/q$, so that

$$E_q[L] = \sum_{u=0}^{|R|} \frac{(M' - u)^+}{q} \binom{|R|}{u} q^u (1 - q)^{|R|-u}. \qquad (14)$$

Thus we can use standard deterministic numerical optimization algorithms (see, e.g., [38, Chap. 10]) to compute q^*. Our focus, however, is on general sequences of insertions and deletions, where the standard approach fails. In Appendix B, we outline a stochastic-optimization approach that is applicable for general transaction sequences. Specifically, we describe an iterative, simulation-based "finite-difference stochastic approximation" (FDSA) algorithm that converges to q^* with probability 1.

The techniques described in Appendix B can readily be adapted to handle very complex cost models—all that is required is that the transaction process and base-data sampling mechanism can be simulated. Because these methods can be relatively expensive and complex to program, is desirable to estimate q^* with quick approximate methods.

4.3.3 An approximate optimization approach

In this section, we explore a closed-form approximation to the function $T(q)$ that is highly accurate in the insertion-only case, and agrees closely with our numerical results in the general case. This approximation immediately leads to an effective approximation of q^*.

The first step in the approximation is to assume that $U = E[U] = |R|q$ with probability 1. Our motivation is that the coefficient of variation $(\mathrm{Var}[U]/E^2[U])^{1/2}$ is of order

$O(|R|^{-1/2})$, and $|R|$ is typically very large. Thus U will be close to u_q with very high probability.

Under the above assumption, the approximate expected phase 1 cost is

$$\hat{T}_1(q) = g(|R|q) = t_a|R| \ln[(|R| - M)/(|R| - M - N(|R|q))]$$
$$= \begin{cases} 0 & \text{if } |R|q \le M; \\ t_a|R| \ln[(|R| - M)/((1-q)|R|)] & \text{if } |R|q \in (M, M'); \\ t_a|R| \ln\left[(|R| - M)/(|R| - M')\right] & \text{if } |R|q \ge M'. \end{cases}$$

To approximate the expected phase 2 cost $T_2(q)$, first suppose that $p = 1$, so that there are no deletion transactions. Then the Markov chain simplifies dramatically, and we can compute $E_q[L \mid U]$ analytically. Specifically, given that $U = u$, we have $L \equiv 0$ if $u \ge M'$ as before; if $u < M'$, then $L - (M' - u)$, the number of Bernoulli rejections before the sample size reaches M', has a negative binomial distribution:

$$P\left\{ L - (M' - u) = k \right\} = \binom{M' - u + k - 1}{M' - u - 1} (1-q)^k q^{M'-u}$$

for $k \ge 0$. Appealing to [25, p. 199], we have $E[L \mid U] = (M' - U)^+/q$ as asserted in Sect. 4.3.2, so that $E[L] \approx (M' - |R|q)^+/q$ under the assumption, as above, that $P\{U = |R|q\} = 1$. To handle the general case in which $p \in (1/2, 1]$, observe that the expected change in the dataset size after each transaction is $p \cdot 1 + (1-p) \cdot (-1) = 2p - 1$, so that the expected number of steps to increase the dataset size by 1 is roughly equal to $1/(2p-1)$. In the insertion-only case, the number of Bernoulli trials in phase 2 equals the number of items added to the dataset. Thus, roughly $1/(2p-1)$ times as many steps are required, on average, to finish phase 2 in the presence of deletions, so we multiply our insertion-only approximation of $E_q[L]$ by a factor of $1/(2p-1)$. This leads to an approximate expected phase 2 cost of

$$\hat{T}_2(q) = \frac{t_b(M' - |R|q)^+}{q(2p-1)},$$

and the expected total time required to resize a sample is approximately equal to $\hat{T}(q) = \hat{T}_1(q) + \hat{T}_2(q)$. It is easy to show that $\hat{T}$ is convex and differentiable on the interval $(M/|R|, M'/|R|)$.

We now choose $q = \hat{q}^*$, where $\hat{q}^*$ minimizes the function $\hat{T}$. Note that the our search for q^* can be restricted to the interval $[M/|R|, M'/|R|]$, because $\hat{T}(q)$ is strictly decreasing on $[0, M/|R|]$ and $\hat{T}(q) \equiv \hat{T}(M'/|R|)$ on $[M'/|R|, 1]$. Thus, to compute $\hat{q}^*$, first set

$$q_0 = \frac{(1 + 4\theta)^{1/2} - 1}{2\theta}, \tag{15}$$

where $\theta = (t_a/t_b)(|R|/M')(2p - 1)$. If $q_0 \in (M/|R|, M'/|R|)$, then q_0 satisfies $\hat{T}'(q_0) = 0$, and we take $\hat{q}^* = q_0$. Otherwise, we take $\hat{q}^*$ to be either $M/|R|$ or $M'/|R|$, depending upon which of the quantities $\hat{T}(M/|R|)$ or $\hat{T}(M'/|R|)$ is smaller.

The combined error introduced by assuming that $P\{U = |R|q\} = 1$ and by replacing the harmonic sum $H(n, m)$ by $\ln(m/(n-1))$ appears to be negligible. Indeed, when $p = 1$, we can compare our approximation to the true expected cost, where the phase 1 cost is computed as in (13)—with $g(u)$ calculated using the true harmonic sum rather than the logarithm—and the phase 2 cost is computed as in (14). In preliminary experiments, the errors that we encountered were typically on the order of 0.0001%. The more interesting question concerns the magnitude of the error introduced by our somewhat ad hoc adjustment for deletions. The experiments in Sect. 6.5 strongly indicate that this latter type of error is also negligible.

It is possible to modify the above methodology to deal with systems and applications not covered by the foregoing analysis. We give several illustrative examples in Appendix C.

5 Merging

The foregoing sections have implicitly assumed that the dataset R and sample S are each maintained at a single location and processed purely sequentially. In practice, it is often the case that R is partitioned across several nodes; see [4] for an example. In this case, it may be desirable to independently maintain a local sample of each partition and compute a global sample of the complete dataset (or, in general, of any desired union of the partitions) by merging these local samples. This approach is often superior, in terms of parallelism and communication cost, to first reconstructing R and sampling afterwards.

We therefore consider the pairwise merging problem, which is defined as follows. Given partitions R_1 and R_2 of R with $R_1 \cup R_2 = R$ and $R_1 \cap R_2 = \emptyset$, along with two mutually independent uniform samples $S_1 \subseteq R_1$ and $S_2 \subseteq R_2$, derive a uniform sample S from R by accessing S_1 and S_2 only. In some scenarios, it suffices to maintain the node samples separately and merge them on demand, e.g., in response to a user query. In other scenarios, it may be the case that R_1 and R_2 are merged into R at the same time that S_1 and S_2 are merged into S; it may then be desirable to incrementally maintain S in the presence of future transactions on R.

Brown and Haas [4] provide an algorithm, called HRMerge, that is designed to solve the merging problem in an insertion-only environment; the algorithm makes no assumptions about the method used to create the uniform samples S_1 and S_2. As described in Sect. 5.1, the algorithm subsamples each of S_1 and S_2, and returns the union of the subsamples as the merged sample S. The size of the merged sample is $|S| = \min(|S_1|, |S_2|)$. It follows that, in the presence of deletions, a naive application of HRMerge can result in very small merged samples even when the target sample sizes M_i are all equal, due to skew in the sample sizes caused by uncompensated deletions. Specifically, if a given sample has many uncompensated deletions, then this very small sample will limit the size of any merged sample in which it participates.

We provide a solution to this problem for scenarios in which each sample is incrementally maintained using the RP algorithm, perhaps with occasional resizing as described in Sect. 4. Our new extension of HRMerge, called RPMerge, yields larger merged samples and is resistant to skew; moreover, the sample produced by RPMerge is accompanied by appropriate values for the counters c_b and c_g, so that incremental maintenance can be continued. The key idea underlying RPMerge is that, conceptually, we do not immediately purge a deleted item from the dataset (or the sample, if present), but rather put it into a "transient" state, and wait until after the merge to purge transient items. Thus, at merge time, the dataset and sample contain both "real" items and transient items that have not yet been purged; we refer to the dataset and sample as being "augmented" with transient items. We run the HRMerge algorithm on the augmented samples to obtain a merged augmented sample, and then purge the transient items to produce the final merged sample. This hypothetical algorithm is illustrated in Fig. 2; real and transient items are shown as white and gray numbered circles, respectively.

The advantage of the hypothetical algorithm is that the sample size produced by the execution of HRMerge is min $(|S_1^+|, |S_2^+|)$, where S_1^+ and S_2^+ are the augmented samples. Since $|S_i^+| \geq |S_i|$ for $i = 1, 2$, we can see intuitively that the hypothetical algorithm can produce larger merged samples than naive HRMerge. As shown in Sect. 5.4, the merged

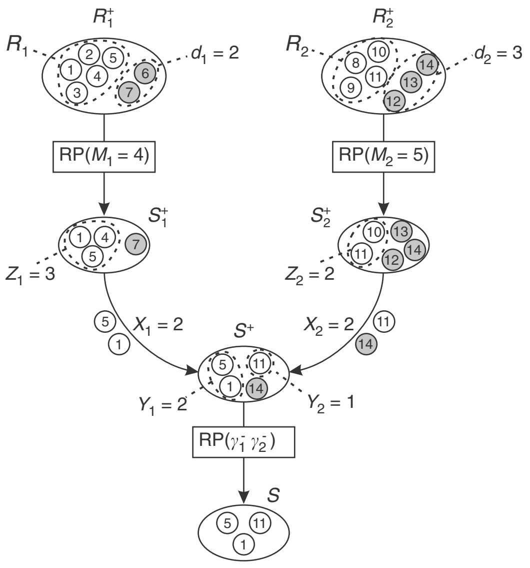

Fig. 2 A hypothetical merging algorithm

sample size depends only on the size of the individual partitions and the *global* number of uncompensated deletions. If all local samples have been generated using the same size parameter M, RPMerge is insensitive to skew in local sample sizes due to uncompensated deletions.

To obtain the actual RPMerge algorithm, we determine the probability distribution of Y_1 and Y_2—the number of real items that S_1 and S_2 ultimately contribute to S in the hypothetical algorithm. We then generate realizations y_1 and y_2 of Y_1 and Y_2 directly, and randomly sample y_1 items from S_1 and y_2 items from S_2 to form the merged sample S. The only remaining task is to determine the appropriate values for the counters c_b and c_g in the merged sample. To this end, we show that the result of running the hypothetical algorithm described above is statistically identical to the result of executing RP on the merged dataset R using a distinguished transaction sequence that is derived from the original sequences γ_1 and γ_2 that were used to create S_1 and S_2. The resulting counter values for the latter scenario are easy to determine, and are assigned to the counters for the sample produced by RPMerge. The foregoing statistical equivalence also permits easy calculation of the probability distribution for the size of the merged sample; in Sect. 5.4, we use this distribution to show that RPMerge typically produces larger sample sizes than naive HRMerge in expectation.

Sections 5.1–5.4 contain the details of the derivation and analysis of RPMerge. In Sect. 5.5, we briefly discuss other merging scenarios in which one or both of S_1 and S_2 are in the process of being resized when the merge occurs.

5.1 Naive HRMerge

The HRMerge algorithm accesses S_1 and S_2 to create a uniform sample S of size $m = \min(|S_1|, |S_2|)$. The basic idea is to select X_1 random items from S_1 and $X_2 = m - X_1$ items from S_2 to include in S, with X_1 being hypergeometrically distributed:

$$P\{X_1 = k\} = H(k; |R_1| + |R_2|, |R_1|, m),$$

where

$$H(k; N, N', M) = \binom{N'}{k}\binom{N - N'}{M - k} \Big/ \binom{N}{M}$$

denotes a hypergeometric probability.

We can apply HRMerge, unchanged, in our setting and, after merging, use the RP algorithm to incrementally maintain S; to initialize RP, set $c_b = c_g = 0$ after the merging process has been completed. Observe, however, that the size of the merged sample is limited by the smaller of the two input samples. In an insertion-only environment such as the one considered in [4], we have $|S_1| = M_1$ and $|S_2| = M_2$ after a sufficiently large number of transactions, where M_1 and M_2 are the respective sample-size bounds used by the

RP algorithm. The size of the sample produced by HRMerge is then $M = \min(M_1, M_2)$. In the presence of deletions, however, we often have $|S_1| < M_1$ and $|S_2| < M_2$, and the merged sample size is $|S| = \min(|S_1|, |S_2|) < M$. As previously discussed in Section 4.1, there is no way to increase the sample size without accessing base data. We show in the sequel that the RPMerge scheme can achieve a sample size of M even when $|S_1| < M_1$ and/or $|S_2| < M_2$; if $|S| < M$, then future insertions can be exploited to grow the sample to size M without accessing base data, since RPMerge provides RP counter values for the merged sample.

5.2 Deferring deletion transactions

Our derivation and analysis of RPMerge rest on Theorem 5 below. This result implies that, when analyzing the statistical properties of the output of the RP algorithm, we can always assume without loss of generality that all deletions are uncompensated deletions and are located at the end of the transaction sequence. Fix a sample-size bound M and, for a feasible finite sequence of transactions γ, denote by $R(\gamma)$, $S(\gamma)$, $C_b(\gamma)$, $C_g(\gamma)$, and $d(\gamma)$, the dataset, sample, counter values, and number of uncompensated deletions that result from processing the transaction sequence γ using the RP algorithm with parameter M.

Theorem 5 *For any finite feasible sequence γ, there exists a finite feasible sequence γ', comprising a subsequence of insertion transactions followed by a (possibly empty) subsequence of deletion transactions, such that*

$$R(\gamma) = R(\gamma') \quad \text{and} \quad d(\gamma) = d(\gamma') \tag{16}$$

and

$$P\left\{ S(\gamma) = A, C_b(\gamma) = c_b, C_g(\gamma) = c_g \right\} \\ = P\left\{ S(\gamma') = A, C_b(\gamma') = c_b, C_g(\gamma') = c_g \right\}$$

for all $A \subseteq R(\gamma)$ and $c_b, c_g \geq 0$ with $c_b + c_g = d(\gamma)$.

Proof Let γ' comprise $|R(\gamma)|$ insertions, one for each item in $R(\gamma)$, followed by the insertion of $d(\gamma)$ arbitrary distinct items from $\mathcal{T} \setminus R(\gamma)$, followed by the deletion of each of the latter $d(\gamma)$ items. Within each of these subsequences, the particular order in which individual items are inserted or deleted can be arbitrary. The equalities in (16) follow immediately from the construction of γ'. Let $v(\gamma)$ be the largest sample size seen so far under sequence γ and similarly for $v(\gamma')$. By (16) and (2), we have $v(\gamma) = v(\gamma')$, so that, by (11), $P\{ S(\gamma) = A \} = P\{ S(\gamma) = A \}$ for all $A \subseteq R(\gamma)$, and hence $P\{ |S(\gamma)| = k \} = P\{ |S(\gamma)| = k \}$ for all $k \geq 0$. The final assertion of the theorem now follows from (5). ◻

5.3 RPMerge

Recall that the HRMerge algorithm creates a random sample S by selecting X_1 items from S_1 and X_2 items from S_2, where the distributions of the random variables X_1 and X_2 are carefully designed to preserve uniformity. The size of S is random with $P\{ |S| \leq m \} = 1$, where $m = \min(|S_1|, |S_2|)$ as before. Our new RPMerge algorithm operates by selecting Y_1 items from S_1 and Y_2 items from S_2. The distributions of Y_1 and Y_2 are different from X_1 and X_2, and are such that S is still uniform, but the probability distribution of the sample size $|S|$ has superior properties to the distribution under HRMerge. As discussed previously, the RPMerge algorithm "simulates" a hypothetical algorithm that we now describe in detail.

Suppose that, for $j = 1, 2$, we have run the RP algorithm with parameter M_j to obtain sample $S_j \subseteq R_j$, and that there are d_j uncompensated deletions. As a technical matter, we assume henceforth that $v_j = \min(M_j, |R_j| + d_j)$, the maximum sample size seen so far, satisfies $v_j = M_j$.[5] Denote by γ_j the transaction sequence that produced R_j and S_j. Using Theorem 5, we can assume without loss of generality that $\gamma_j = \gamma_j^+ \gamma_j^-$, where γ_j^+ consists solely of insertions and γ_j^- consists solely of deletions. The sequence γ_j^+ corresponds to the insertion of the $|R_j|$ "real" items that comprise R_j, followed by the insertion of d_j "transient" items in $\mathcal{T} \setminus R_j$ that will ultimately be deleted from the dataset by processing the transactions in γ_j^-.

Our hypothetical algorithm combines the RP algorithm with the HRMerge algorithm, deferring the purging of transient items from the merged sample until the final step. Specifically, for $j = 1, 2$, the algorithm creates an intermediate augmented dataset $R_j^+ = R(\gamma_j^+)$ and augmented sample $S_j^+ = S(\gamma_j^+)$ by running RP independently on the two partitions, using sample-size bound M_j and transaction sequence γ_j^+ on partition j. Dataset R_j^+ comprises $|R_j|$ real items and d_j transient items, and sample S_j^+ comprises Z_j real items and $M_j - Z_j$ transient items, where

$$P\left\{ Z_j = k \right\} = H(k; |R_j| + d_j, |R_j|, M_j). \tag{17}$$

The hypothetical algorithm now applies the HRMerge algorithm to merge the augmented samples S_1^+ and S_2^+. HRMerge selects X_1 items from S_1^+ and X_2 items from S_2^+, where

$$P\{ X_1 = k, X_2 = M - k \} \\ = H(k; |R_1| + d_1 + |R_2| + d_2, |R_1| + d_1, M). \tag{18}$$

[5] Otherwise, we have $S_j = R_j$ and the merging problem can be trivially solved. For example, if $S_1 = R_1$, we continue the RP algorithm on S_2 using the items of S_1, in any order, as input.

Observe that, of the X_j items from S_j^+ added to S^+, precisely Y_j items are real, where

$$P\left\{Y_j = k \mid X_j, Z_j\right\} = H(k; M_j, Z_j, X_j). \tag{19}$$

The hypothetical algorithm concludes by running the RP algorithm on the sequence $\gamma_1^-\gamma_2^-$, starting with augmented sample S^+ and counter values $c_b = c_g = 0$, and using sample-size bound M. This final processing step has the effect of removing all transient items from S^+, thereby transforming S^+ into the final sample S. Figure 2 depicts the hypothetical algorithm in action.

We now determine appropriate counter values for the merged sample, as well as the probability distribution for the size of the merged sample. First observe that, by Theorem 1 and results in [4], the sample S^+ that results from the execution of HRMerge by the hypothetical algorithm is statistically identical to the sample $S(\gamma_1^+\gamma_2^+) \subseteq R(\gamma_1^+\gamma_2^+)$ obtained by running RP on the dataset R, using the sequence $\gamma_1^+\gamma_2^+$. Thus the final merged sample produced by the hypothetical algorithm is statistically identical to the sample $S(\gamma_1^+\gamma_2^+\gamma_1^-\gamma_2^-)$ produced by running RP on R using the sequence $\gamma_1^+\gamma_2^+\gamma_1^-\gamma_2^-$, starting with counter values $c_b = c_g = 0$. Observe that, after running RP on R in this manner, the final counter values are $c_b = M - |S|$ and $c_g = d_1 + d_2 - c_b$. Indeed, after the transactions in $\gamma_1^+\gamma_2^+$ have been processed, the sample size is M and the counter values for RP are given by $c_b = c_g = 0$ since no deletions have occurred so far; after processing the remaining sequence transactions in the sequence $\gamma_1^-\gamma_2^-$, the deficit $M - |S| = c_b$ corresponds precisely to the bad deletions, and the remaining $d_1 + d_2 - c_b$ deletions are good. Since the hypothetical algorithm produces output just as if RP had been run on R, we can start with these counter values when maintaining the merged sample S. Finally, by Theorem 2, the probability distribution for the size of the sample produced by running RP on R, and hence the probability distribution for sample size produced by the hypothetical algorithm, is given by

$$P\{|S| = k\} = H(k; |R_1| + |R_2| + d_1 + d_2, |R_1| + |R_2|, M) \tag{20}$$

for $0 \le k \le M$, where $M = \min(M_1, M_2)$. We show below that, by construction, the output of the actual RPMerge algorithm is statistically identical to the output of the hypothetical algorithm, and the above results on counter values and sample-size distributions apply to RPMerge as well.

We now specify the RPMerge procedure. Ignoring transient items, we see that the net effect of the hypothetical algorithm is to select, for $j = 1, 2$, precisely Y_j items from a uniform sample $S_j^* \subseteq R_j$, where the sample size $Z_j = |S_j^*|$ is distributed according to (17) and Y_j—conditionally on $|S_j^*|$ and an auxiliary random variable X_j as in (18)—is distributed according to (19). Observe that, by Theorem 2, the distribution of $|S_j^*|$ is identical to that of $|S_j|$, so we can take S_j for S_j^* in the procedure. Thus we generate X_j according to (18) and then Y_j according to $H(\cdot\,; M_j, |S_j|, X_j)$.

Algorithm 4 Sample merging

```
1: $N_j$: cardinality of partition $R_j$ ($j = 1, 2$)
2: $S_j$: sample of $R_j$
3: $M_j$: sample-size bound used by RP for generating $S_j$
4: $d_j$: number of uncompensated deletions for $R_j$
5:
6: $M \leftarrow \min(M_1, M_2)$
7: $X_1 \leftarrow$ HYPERGEOMETRIC($N_1 + N_2 + d_1 + d_2, N_1 + d_1, M$)
8: $Y_1 \leftarrow$ HYPERGEOMETRIC($M_1, |S_1|, X_1$)
9: $X_2 \leftarrow M - X_1$
10: $Y_2 \leftarrow$ HYPERGEOMETRIC($M_2, |S_2|, X_2$)
11:
12: $S \leftarrow \{ Y_1$ random items from $S_1 \} \cup \{ Y_2$ random items from $S_2 \}$
13: $c_b = M - |S|$
14: $c_g = d_1 + d_2 - c_b$
15:
16: return $(S, c_b, c_g)$
```

In Algorithm 4, we give the complete pseudocode for the merging procedure. We make use of a function HYPERGEOMETRIC that generates samples from the hypergeometric distribution; see [46, p. 101] or [26] for efficient rejection algorithms, or refer to a statistical library that includes this function, such as [6] or [16].

5.4 Comparison of expected sample sizes

The sample size $|S|$ produced by RPMerge is hypergeometrically distributed as in (20) and, by Theorem 2,

$$E[|S|] = \frac{|R_1| + |R_2|}{|R_1| + |R_2| + d_1 + d_2} M.$$

As indicated previously, the sample size produced by RPMerge can be strictly larger than that produced by a naive application of HRMerge. For example, ignoring the transient items in the scenario of Fig. 2, we see that RP produces samples $S_1 \subseteq R_1$ and $S_2 \subseteq R_2$ with $|S_1| = 3$ and $|S_2| = 2$, so that HRMerge can produce a sample of at most $\min(|S_1|, |S_2|) = 2$ items. In contrast, RPMerge has produced a sample S that contains three items.

We now show that RPMerge often performs better in terms of average sample size, also. In the common case where both samples have been generated using the same sample size parameter, the following theorem asserts that RPMerge produces samples that are at least as large, on average, as those produced by HRMerge. Indeed, the expected sample size for RPMerge is often strictly larger.

Theorem 6 *Suppose that $|R_j| > 0$ and $|R_j| + d_j > M_j$ for $j = 1, 2$. If $M_1 = M_2 = M$, then*

$$E[\min(|S_1|, |S_2|)] \leq E[|S|], \tag{21}$$

with equality holding if and only if $d_1 = d_2 = 0$.

The assumptions that $|R_j| > 0$ and $|R_j| + d_j > M_j$ for $j = 1, 2$ virtually always hold in practical cases of interest. Indeed, the latter assumption is just slightly stronger than our running assumption that $v_j = M_j$. To prove Theorem 6, we need the following lemma.

Lemma 1 *For any random variables X and Y, we have*

$$E[\min(X, Y)] \leq \min(E[X], E[Y]),$$

and the above inequality is strict if $P\{X < Y\} > 0$ and $P\{X > Y\} > 0$.

Proof Observe that

$$\begin{aligned} E[\min(X, Y)] &= E[XI(X \leq Y) + YI(X > Y)] \\ &= E[X] - E[XI(X > Y)] + E[YI(X > Y)] \\ &= E[X] - E[(X - Y)I(X > Y)], \end{aligned}$$

where $I(A) = 1$ if event A occurs and $I(A) = 0$ otherwise. Thus $E[\min(X, Y)] \leq E[X]$, and the inequality is strict if $P\{X > Y\} > 0$. Similarly, $E[\min(X, Y)] \leq E[Y]$, and the inequality is strict if $P\{X < Y\} > 0$. The desired result follows immediately. □

Proof (Theorem 6) First suppose that $d_1 = d_2 = 0$. Since $|R_j| + d_j > M$, we have $|S_1| = M$, $|S_2| = M$, and $|S| = M$, each with probability 1, so that (21) holds with equality. Otherwise, suppose that $d_1 + d_2 > 0$ and, without loss of generality, that $E[S_1] \leq E[S_2]$. By Theorem 2, the latter assumption is equivalent to

$$\frac{|R_1|}{|R_1| + d_1} M \leq \frac{|R_2|}{|R_2| + d_2} M.$$

Multiply by $(|R_1|+d_1)(|R_2|+d_2)$ and add $(|R_1|^2+|R_1|d_1)M$ to both sides of the inequality to obtain

$$|R_1|(|R_1| + |R_2| + d_1 + d_2)M \leq (|R_1| + |R_2|)(|R_1| + d_1)M.$$

Divide both sides by $(|R_1| + |R_2| + d_1 + d_2)(|R_1| + d_1)$ to show that $E[|S_1|] \leq E[|S|]$, where equality holds if and only if $E[|S_1|] = E[|S_2|]$. Using Lemma 1, we have

$$\begin{aligned} E[\min(|S_1|, |S_2|)] &\leq \min(E[|S_1|], E[|S_2|]) \qquad (22) \\ &= E[|S_1|] \leq E[|S|]. \qquad (23) \end{aligned}$$

If $E[|S_1|] < E[|S_2|]$, then the inequality in (23), and hence in (21), is strict. Otherwise, we claim that $P\{|S_1| > |S_2|\} > 0$ and $P\{|S_1| < |S_2|\} > 0$, so that, by Lemma 1, the inequality in (22)—and hence in (21)—is strict, and the desired result follows.

To see that the above claim holds, suppose that $E[|S_1|] = E[|S_2|]$. This equality and the fact that $d_1 + d_2 > 0$ together imply that both d_1 and d_2 are positive. For $j = 1, 2$, denote by $l_j = \max(0, M - d_j)$ and $u_j = \min(M, |R_j|)$ the minimum and maximum possible values for $|S_j|$. It is straightforward to show that $l_j < u_j$, given that $d_j > 0$ and, under our assumptions, $|R_j| > 0$ and $|R_j| + d_j > M$. Moreover, by Theorem 2, $P\left\{|S_j| = k\right\} > 0$ for $l_j \leq k \leq u_j$, and therefore

$$\max(l_1, l_2) < E[|S_1|] = E[|S_2|] < \min(u_1, u_2).$$

It follows that the intervals $[l_1, u_1]$ and $[l_2, u_2]$ strictly overlap, i.e., their intersection contains at least two integer values, say, i and $i + 1$. Since RP is executed independently on the two partitions, we have

$$\begin{aligned} P\{|S_1| > |S_2|\} &\geq P\{|S_1| = i + 1, |S_2| = i\} \\ &= P\{|S_1| = i + 1\} P\{|S_2| = i\} > 0. \end{aligned}$$

A symmetric argument shows that $P\{|S_1| < |S_2|\} > 0$. □

5.5 Other merging scenarios

The foregoing discussion has assumed that the two samples to be merged are each being maintained using the RP algorithm. In practice, other merging scenarios can arise if one or both of the samples is in the process of being resized, as described in Sect. 4. We briefly describe how to handle these situations.

If both S_1 and S_2 are being resized, then they can be viewed as BERN(q_1) and BERN(q_2) samples, respectively. For any sampling rate q with $0 < q \leq \min(q_1, q_2)$, the samples S_1 and S_2 can be merged to form a BERN(q) sample S using the following technique [4]: For $j = 1, 2$, take a BERN(q/q_j) subsample of S_j to create a BERN(q) sample S'_j. Then set $S = S'_1 \cup S'_2$. If $|S|$ is too small, then the merged sample S can be allowed to drift up to a new sample size as in phase 2 of the resizing algorithm; if $|S|$ is too large, then S can be subsampled using, for example, the methods given in [44].

Now suppose that S_1 is being resized and S_2 is not. We can view S_1 as having been produced by the RP algorithm with sample-size bound $M = |S_1|$, and we can take the counter values as $c_b = c_g = 0$. To obtain a merged sample S, we now run the RPMerge algorithm as described in Sect. 5.3.

6 Experiments

We conducted an experimental study to (i) evaluate the stability and performance of the RP scheme with respect to the various algorithms mentioned in Sect. 2, (ii) to compare the numerical and approximate methods for tuning the resizing parameter q, and (iii) to compare the HRMerge and RPMerge algorithms for merging samples.

In summary, we found that RP has the following desirable properties:

- When the fluctuations of the dataset size over time are not too extreme, RP produces sample sizes that are as stable as those produced by slower algorithms that access the base data.
- The speed of RP is clearly faster than any sampling scheme that requires access to the base data.

For the optimizations in Sect. 3.4, we found that

- The first optimization never slows down RP, and can speed up RP when the dataset is growing or when the dataset is stable and the transaction stream contains long sequences of insertions.
- The second optimization can slow down RP when the dataset is stable and the number of insertions that occur between a pair of successive deletions tends to be small; the optimization is beneficial when the dataset is growing—e.g., in between resizing operations—or when the dataset is stable and the transaction stream contains long sequences of insertions.

For the resizing algorithm, we found that

- The Monte Carlo-based numerical approach of Appendix B and the quick approximation approach of Sect. 4.3.3 agree very closely with respect to both the cost function $T(q)$ and the optimal parameter value q^*. These results validate the accuracy of the quick approximate tuning method.
- The time needed for resizing has low variance, so that the algorithm has stable performance.
- A good choice of q can have a significant impact on the resizing cost.

For the merging algorithm, we found that

- In most cases, RPMerge produces significantly larger samples than HRMerge.
- As predicted by theory—see (20)—the sample size produced by RPMerge is dependent on the total number of uncompensated deletions but independent of their distribution among the individual partitions.
- The relative sample-size advantage of RPMerge over HRMerge grows as the total number of uncompensated deletions increases.

6.1 Experimental setup

We implemented the new RP algorithm, as well as the CAR, CARWOR, and RSR schemes, using Java 1.6. We employed an indexed in-memory array to efficiently support the deletion of items.

All of the experiments used synthetic data; since our focus is on uniform sampling of unique data items, the actual data values are irrelevant. We ran our experiments on a variety of systems and, for most of the experiments, measured the number of operations instead of actual processing times in order to facilitate meaningful comparisons. Because the sampling algorithms in this paper can potentially be used in a wide range of application scenarios, our approach has the advantage that the results reported here can be customized to any specific scenario by appropriately costing the various operations. For example, if the base data corresponds to a single relational table, then access to this data can be costed more cheaply than if the base data is, say, a view over a join query. Unless otherwise stated, a reported result represents an average over at least 100 runs.

We assumed that the deletions and insertions are clustered into batches of b operations, and simulated the sequence of dataset operations by randomly deciding whether the next b operations are insertions or deletions. Our default value was $b = 1$, but we also ran experiments in which we systematically varied the value of b to investigate the effect of different insertion/deletion patterns.

6.2 Sample size

We evaluated the sample-size stability for the various algorithms by executing a randomly generated sequence of 5,000,000 insertion/deletion operations while incrementally maintaining a sample with a target size (and upper bound) of 100,000 items. To create a scenario in which the dataset of interest is reasonably large, we restricted the first 1,000,000 operations to be insertions only. We used a lower bound of 80,000 items for the RSR algorithm. The goal of this experiment is to illustrate the qualitative behavior of the algorithms, and so we did not average over multiple runs. For each algorithm, we plotted the sample size as it evolved over time.[6] The upper part of Fig. 3 displays results for the sampling schemes that access the base data, and the lower part displays results for the RP algorithm, which avoids base-data accesses.

As can be seen, CAR and CARWOR are optimal, since they are able to maintain the sample at its upper bound. These algorithms, however, need to access the base data. RSR also needs to access the base data, but the sample size is less stable than that of CAR or CARWOR; it fluctuates in the range

[6] We use “time” and “number of operations” synonymously.

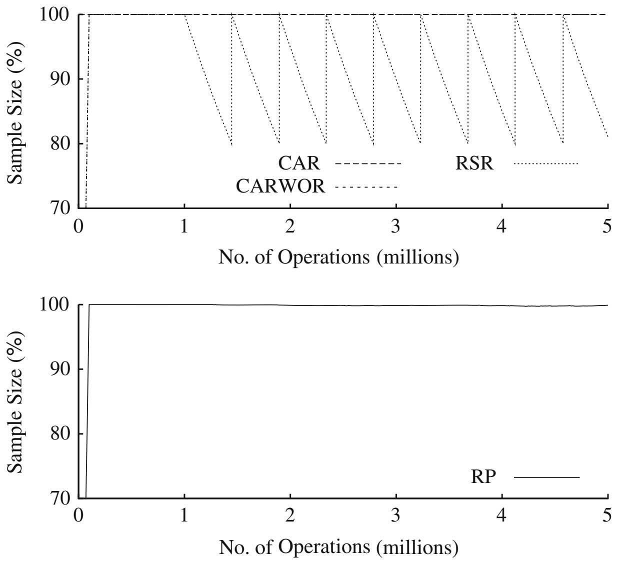

Fig. 3 Evolution of sample size over time

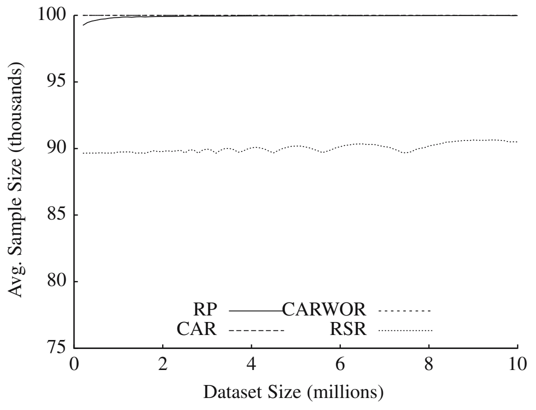

Fig. 4 Dataset size and avg. sample size

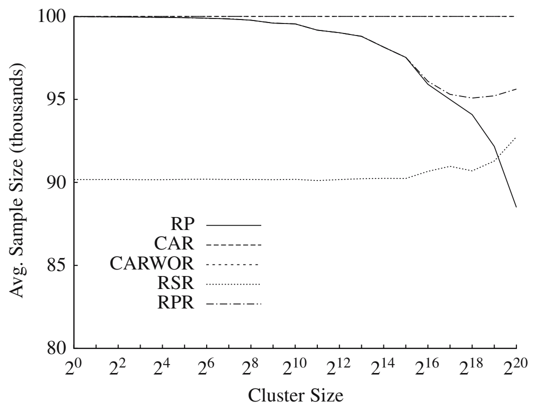

Fig. 5 Cluster size and avg. sample size

$[0.8M, M]$. We see that the sample sizes produced by RP are almost indistinguishable from those of CAR and CARWOR.

We next measured the time-average sample size for a range of dataset sizes, providing further insight into the impact of deletions. For each dataset size, we used a sequence of insertions to create both the dataset and the initial sample, and then measured changes in the sample size over time as we inserted and deleted $10{,}000{,}000$ items at random. The results are shown in Fig. 4. Again, RP performs comparably to CAR and CARWOR, in that it maintains a sample size close to the upper bound M. In contrast, the time-average sample sizes for RSR are smaller than those of the other algorithms. The reason for this behavior is that the RSR algorithm actively adjusts the sample size only periodically.

The foregoing experiments use a cluster size of $b = 1$, which means that the fluctuations in the dataset size are relatively small. We expect that, when the dataset size fluctuates strongly, so does the sample size when base-data access is disallowed. Specifically, the sample size produced by RP depends on the number of uncompensated deletions, which in turn is determined as the difference between the current dataset size and the maximum dataset size seen so far. To study this effect experimentally, we varied the magnitude of the fluctuations by varying the cluster size b. We started with a dataset consisting of $10{,}000{,}000$ items and a sample size of $100{,}000$. We then performed 2^{23} operations and averaged the sample size after every b operations. The results for different values of b are shown in Fig. 5.

As can be seen, the sample sizes produced by algorithms that access base data are independent of the cluster size, whereas those produced by RP depend on the cluster size; the higher the variance of dataset size, the lower the average sample size. Due to high peaks in dataset size, RP may fail to maintain a sufficiently large sample if the cluster size is large with respect to the dataset size. In this extreme case, base-data access is required in order to enlarge the sample. A combination of RP and resizing (RPR) can handle even this situation while minimizing accesses to the base data. Note that RPR guarantees a lower bound on the sample size, whereas RP does not.

In a final experiment, we measured the overall cost of the various sampling schemes relative to the average sample size produced by them, for various cluster sizes. (Our cost model is described in the next section.) The results are shown in Fig. 6. RSR performs worst since it is expensive and produces a non-optimal sample size; both CAR and CARWOR are more stable and less expensive. Overall, the RP-based schemes are clearly superior. As indicated above, RPR performs comparably to RP when the cluster sizes are reasonably large. The extra cost of RPR comes into play when the fluctuations in the database size are extreme. Indeed, when the cluster size exceeds 2^{20}, the sample is refilled after almost every deletion block, and RPR reduces to CARWOR.

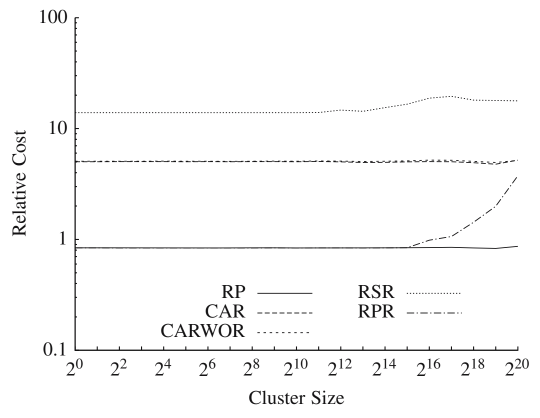

Fig. 6 Cluster size and relative cost

6.3 Performance (base-data and sample accesses)

To evaluate the relative cost of the sampling algorithms, we ran them using different dataset sizes while counting the number of dataset reads and sample writes. These two factors strongly influence the performance of the algorithms. Again, we created a sequence of 10,000,000 insertions and deletions and averaged the results over various independent runs.

Figure 7 depicts the number of accesses to base data for the different algorithms. Because it must periodically recompute the entire sample, RSR requires more base-data accesses than any other sampling scheme. Both CAR and CARWOR perform better than RSR, with CARWOR incurring more base-data accesses than CAR due to duplicate removal. All of these algorithms require fewer accesses to a larger dataset than to a smaller one because, for a bounded-size sample, the effective sampling fraction drops with increasing dataset size, so that frequency of deletions from the sample drops as well. Note, however, that if large datasets are subject to modifications more often than small ones, then this effect may vanish. Finally, observe that, because RP never requires access to the base data, its cost curve is indistinguishable from the x-axis.

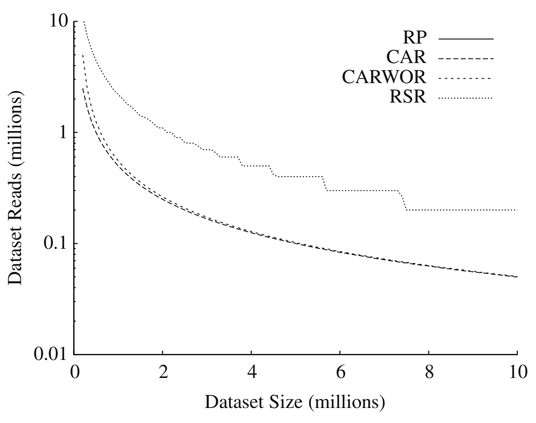

Fig. 7 Number of dataset reads

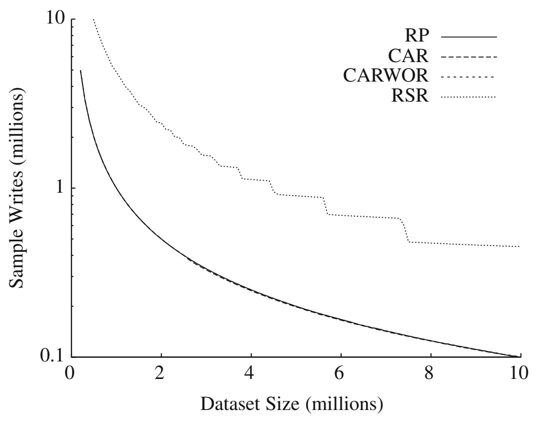

Fig. 8 Number of sample writes

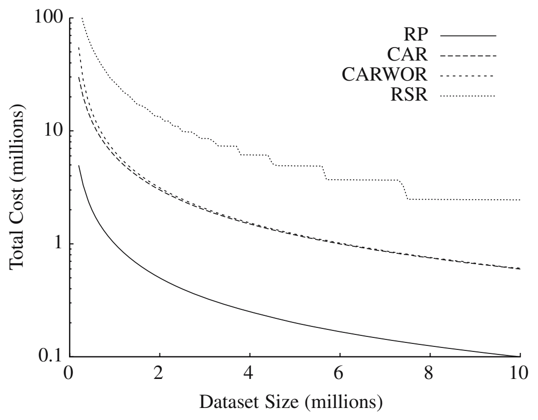

Fig. 9 Combined cost

Figure 8 shows the number of write accesses to the sample for the different sampling schemes. Again, RSR is the least efficient algorithm, because every recomputation completely flushes the current sample and refills it using base data. The other algorithms perform comparably; indeed, the curves for CAR, CARWOR, and RP coincide.

Figure 9 shows the combined cost of sample and population accesses, assuming that the latter type of access is ten times as expensive as the former. (In many applications, the relative cost of population accesses might be significantly higher.) Again, RP clearly outperforms sampling schemes that require base data access.

6.4 Performance (CPU)

We next evaluated the performance of RP in terms of CPU cost. Sampling schemes that require base-data accesses were not considered, because these accesses typically outweigh the computational cost. We implemented RP with none of the optimizations from Sect. 3.4, with the optimization of the

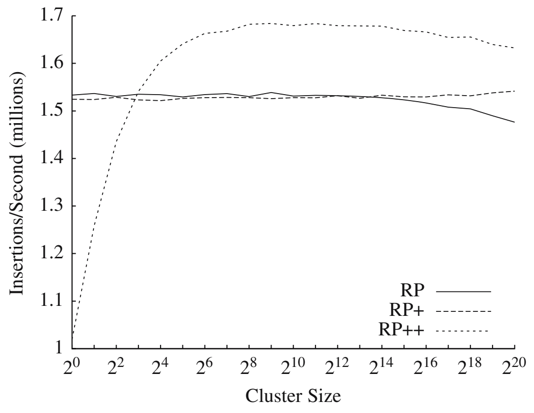

Fig. 10 Throughput, stable dataset

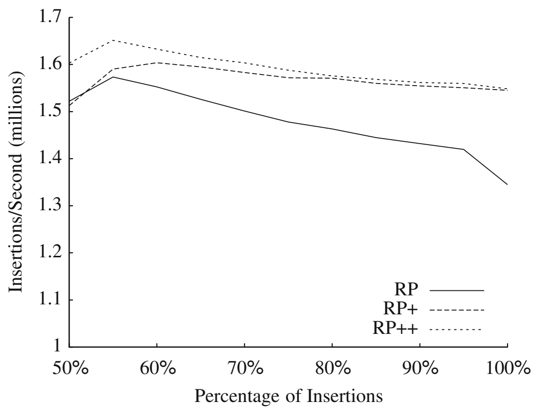

Fig. 11 Throughput, growing dataset

reservoir step (RP+), and with both optimizations (RP++). We used an indexed in-memory data structure to store the individual items. Under our experimental parameter settings, the addition of an item to the data structure takes 1.1 µs, the replacement of an item by another one 2.2 µs, the look-up cost is 0.9 µs and the removal of an item takes 3.3 µs. To generate random numbers, we used the "Mersenne Twister" of Matsumoto and Nishimura [32]; each random number takes approximately 0.6 µs to produce.[7] Each of the above times includes the measurement time.

For each of our experiments, we generated a dataset consisting of 10 million items and computed initial samples of size 100,000. We then generated a sequence of 2^{23} insertion and deletion transactions and measured the average throughput (transactions per second) separately for both types of transactions. We found that the time to process a deletion transaction is almost identical for all versions of RP, and we therefore focus our discussion on insertion transactions.

Figure 10 displays the throughput performance on a stable dataset for various cluster sizes b. First, observe that RP and RP+ perform similarly. The reason is that RP+ optimizes the reservoir step, which is executed very infrequently if the dataset is stable and does not fluctuate strongly. In contrast, when b is small, RP++ slows down the sampling process due to frequent invalidations of the skip counter K'. For $b \geq 16$, RP++ performs better than both RP and RP+.[8]

Figure 11 plots the throughput of the insertion transactions on a growing dataset. In this experiment, we fixed $b = 16$ and varied the fraction of insertion transactions from $p = 0.5$ (stable) to $p = 1$ (insertion-only). As p increases, more reservoir steps and less pairing steps are executed while running RP. Since the former are more expensive (replacement of an item) than the latter (addition of an item), the performance of RP degrades as p grows. However, the optimized schemes RP+ and RP++ partly compensate for this effect by generating (far) fewer random numbers as the fraction of insertions increases, so that these optimized algorithms outperform plain RP. Note that RP+ and RP++ coincide when $p = 1$, since, in this case, no pairing step is executed.

6.5 Resizing

We next compared the quick approximate method of Sect. 4.3.3 for computing the optimal parameter q^* to the numerical methods of Sect. 4.3.2 and Appendix B. Throughout, unless specified otherwise, we used initial and final sample sizes of $M = 100{,}000$, $M' = 200{,}000$, respectively, and also set $|R| = 1{,}000{,}000$. In addition, we set $p = 0.6$ and $t_b = 1 - ms$; recall that p represents the probability that a transaction is an insertion, and t_b is the expected time between Bernoulli trials during phase 2. The experimental results for various other choices of parameters were qualitatively similar.

Figure 12 displays the expected resizing cost $T(q)$ for various values of q, when the base-data access cost t_a equals

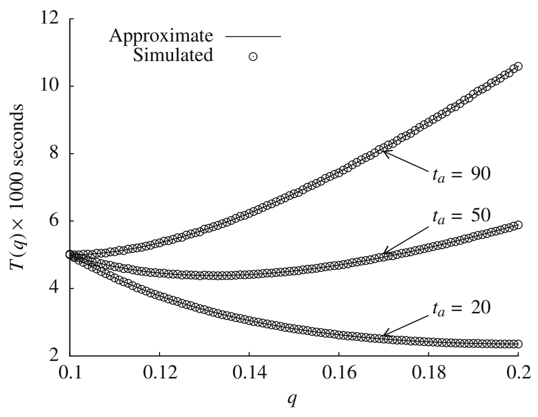

Fig. 12 Expected resizing cost $T(q)$

[7] Surprisingly, the weaker PRNG shipped with Sun's JDK requires 0.7 µs and is therefore slower.

[8] An implementation of random pairing may switch from RP+ to RP++ and vice versa, depending on the incoming transaction stream.

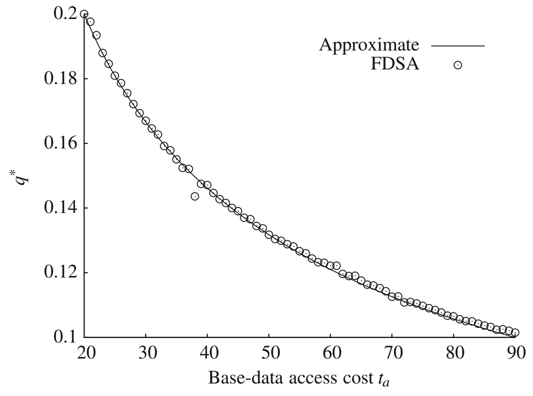

Fig. 13 Optimal value of q

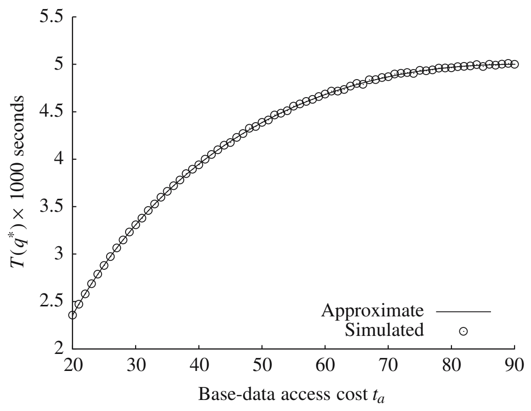

Fig. 14 Optimal resizing cost $T(q^*)$

20, 50, and 90 ms.[9] These choices of t_a illustrate the three possible behaviors (increasing, decreasing, and internal minimum point) for the T function discussed in Appendix B. For each scenario, the approximate cost $\hat{T}$ as computed in Sect. 4.3.3 is represented as a solid curve. Superimposed on this curve are points that represent estimates of T for various values of q; each point represents an average over 10 simulation replications. As expected, when the base-data access cost t_a is relatively small, the cost function achieves its minimum value at $q = 0.2$, and the optimal strategy is to increase the sample size to M' during phase 1, and not execute phase 2. When t_a is relatively large, the cost function achieves its minimum value at $q = 0.1$, and the optimal strategy is to not sample the base data at all, and increase the sample size to M' exclusively during phase 2. For an intermediate value of t_a, the optimal value of q falls in between 0.1 and 0.2—for this example, $q^* \approx 0.132$—so that the resizing work is allocated between the two phases. Note that, in this example, the expected costs corresponding to the best and worst choices of q can vary by a factor of two. Also note that the approximate and simulated costs are extremely close to each other. This high degree of consistency, which was observed for all parameter values that we investigated, increases our confidence in the quick approximate cost model. Our final observation is that the simulated resizing times showed very little variability from one simulation run to the next; the standard deviation of the resizing time was always less than 1% of the average resizing time. This indicates that the performance of the resizing algorithm is stable.

The high accuracy of the cost approximation leads us to expect that our numerical and approximate methods will also yield similar estimates for q^*. This expectation is fulfilled, as shown in Figs. 13 and 14. Figure 13 shows the optimal value q^* for various values of t_a. As before, the solid line represents values computed via the quick approximation method of Sect. 4.3.3, and the circular points represent numerical solutions, each obtained from 100 iterations of the FDSA algorithm of Appendix B. As can be seen, the results for the two different methods agree very closely, providing further justification for the quick approximate method. The slightly anomalous result at $t_a = 38$ appears to be due to a random fluctuation; Figure 14 shows, however, that even when the value of q^* produced by FDSA differs slightly from the result of the approximate closed-form model, the resulting approximated and simulated optimal resizing costs do not differ perceptibly.

[9] A complete recomputation of the sample from scratch therefore takes 4,000s, 10,000s and 18,000s, respectively.

6.6 Merging

We compared the RPMerge algorithm given in Sect. 5 to the naive application of the HRMerge algorithm in [4]. Since RPMerge leverages the fact that the samples are generated by the RP algorithm, we expect RPMerge to exhibit superior performance. In our experiments, we generated two datasets R_1 and R_2 consisting of 2 million and 1 million items, respectively. We then inserted and subsequently deleted d_1 items from R_1 and d_2 items from R_2. As we generated the datasets and performed the subsequent insertions and deletions, we maintained samples S_1 and S_2 using bounds $M_1 = M_2 = 10,000$. Finally, we merged both samples using RPMerge and HRMerge. Our reported results are averages over 100 independent runs. Note that there are d_1 and d_2 uncompensated deletions corresponding to S_1 and S_2, respectively, and that the sizes of both R_1 and R_2 remain constant as d_1 and d_2 vary.

In a first experiment, we fixed the total number of uncompensated deletions to $500,000 = d_1 + d_2$. Figure 15 displays the average sample size after merging for various values of d_1 and d_2. The sample size produced by RPMerge depends only on the sum of d_1 and d_2, and hence is insensitive to the individual values of these experimental parameters. In contrast, HRMerge produces samples whose size equals the smaller of

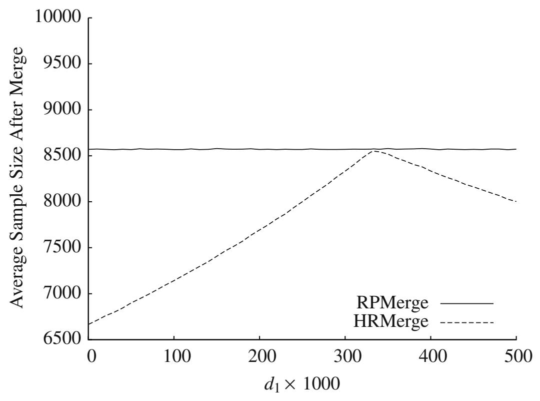

Fig. 15 Merging, $d_1 + d_2$ fixed

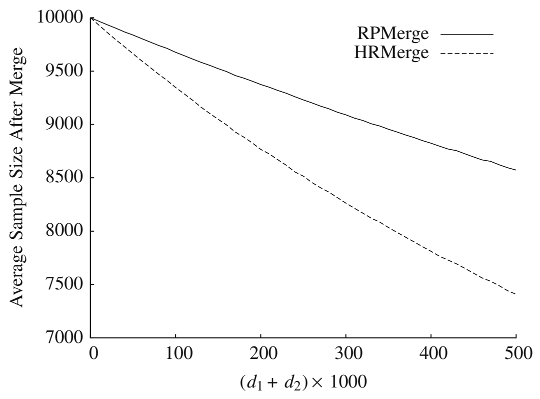

Fig. 16 Merging, $d_1/(d_1 + d_2)$ fixed

the two input sample sizes. The best performance is achieved when $E[\min(|S_1|, |S_2|)] \approx \min(E[|S_1|], E[|S_2|])$ is maximized. Since $E[|S_j|] = M|R_j|/(|R_j| + d_j)$, this optimal performance is achieved when $d_1 \approx 333,333$. In this case, RPMerge and HRMerge produce samples of approximately the same size. In all other cases, RPMerge is clearly superior.

We next evaluated the impact of the total number of uncompensated deletions on sample-size performance. We experimented with different values of $d_1 + d_2$ and set d_1 to $0.3(d_1 + d_2)$, so that 30% of the uncompensated deletions occur in the first partition. As can be seen in Fig. 16, both merging algorithms produce samples of size M when $d_1 = d_2 = 0$. The larger the number of uncompensated deletions, the greater the advantage of RPMerge over HRMerge. Together with the fact that samples produced by RPMerge can be (re)grown up to size M using subsequent insertions only, RPMerge seems to be the merging algorithm of choice.

7 Summary and conclusions

Techniques for incrementally maintaining bounded uniform samples over "datasets"—whether relational tables, views, XML repositories, or other data collections—are crucial for unlocking the full power of database sampling techniques. We have systematically studied methods for maintaining such samples under arbitrary insertions and deletions to the dataset. For stable datasets in which the dataset size does not undergo extreme fluctuations, our new RP algorithm, which generalizes both reservoir sampling and "passive" stream sampling, is the algorithm of choice with respect to speed and sample-size stability. In the presence of extreme fluctuations in the dataset size, RP can be combined with resizing or resampling algorithms to achieve acceptable sample sizes while minimizing expensive base-data accesses. For growing datasets, our new resizing algorithm permits the sample size to grow in a controlled manner. We have developed both numerical methods and approximate analytical methods for optimally tuning the algorithm to minimize the time required for resizing. When both tuning methods are applicable, they appear to yield almost identical results; the numerical methods, based on stochastic-approximation techniques, can potentially be applied even in complex scenarios where approximations are not available. For distributed environments, where the dataset is partitioned over multiple nodes and local samples are maintained at each node, we have provided a novel extension of the HRMerge algorithm that produces a sample of the complete dataset (or of any desired union of the partitions) from the local samples. Typically, our algorithm achieves larger sample sizes than are obtained via a naive application of HRMerge.

The RP algorithm as given in this paper has been designed for in-memory samples. In the future, we plan to look at methods for handling large disk-based samples, as might occur in very large data-warehouse scenarios. The key challenge here is to minimize the number of random disk accesses. One approach would be to log all of the transactions (or a well-chosen subset thereof) and to refresh the sample in a deferred manner. Future work includes finding algorithms—similar to those in [10]—that minimize both the amount of logged information and the refresh cost. Another approach would be to keep an in-memory index of the items, but store the actual values of the items on disk. The index can then be used to efficiently support the look-up operation for deleted items, while the on-disk sample might be organized to efficiently support update operations, perhaps by adapting the techniques in [23].

Appendix A: Non-uniformity of Bernoulli sampling with purging

The Bernoulli sampling with purging (BSP) scheme combines Bernoulli sampling with a technique proposed by Gibbons and Matias [14]. The idea is to use MBERN(q) sampling and to purge the sample whenever the sample size exceeds an upper bound M. In more detail, the BSP scheme starts with $q = 1$. Each purge operation consists of one or more subsampling steps, and q is decreased at each such step. Specifically, the sample is subsampled using a BERN(q'/q) scheme, where $q' < q$, and then q is set equal to q'. This procedure is repeated until the sample size falls below M, at which point the purge operation terminates and MBERN(q) sampling recommences, using the new, reduced value of q. In this appendix, we show that BSP is not a uniform sampling scheme. To simplify the discussion, we assume that $q' = pq$ for a fixed constant $p \in (0, 1)$; similar arguments apply when q'/q can vary over the subsampling steps.

The purge operation is executed whenever the sample size increases to $M + 1$. Each purge involves L Bernoulli subsampling steps with sampling rate p, where L is a geometrically distributed random variable with $P\{L = k\} = p'(1 - p')^{k-1}$. Here $p' = 1 - p^{M+1}$ denotes the probability that at least one of the M sample items is rejected so that the purge operation terminates. Denoting by S' the subsample that results from executing the purge operation on S, we have

$$\begin{aligned} &P\{S' = A\} \\ &\quad = P\{\text{all } t \in A \text{ retained, all } t \in S \setminus A \text{ purged} \mid \geq 1 \text{ item purged}\} \\ &\quad = p^{|A|}(1 - p)^{M+1-|A|}/p' \end{aligned} \tag{24}$$

for all $A \subset S$. After the purge operation terminates, the sampling process proceeds with new sampling rate qp^L.

We now give a simple example where BSP does not produce a uniform sample. Consider the sequence $\gamma = (+t_1, +t_2, -t_1, +t_3)$ and set $M = 1$. Denote by R_i the dataset, by S_i the sample and by Q_i the (random) sampling rate after processing the ith transaction. After t_1 has been inserted, we have $R_1 = S_1 = \{t_1\}$ and $Q_1 = 1$ with probability 1. The insertion of t_2 triggers a purge operation and, using (24), we find that

$$P\{S_2 = \emptyset\} = 1 - 2r,$$
$$P\{S_2 = \{t_1\}\} = P\{S_2 = \{t_2\}\} = r, \quad Q_2 = p^L$$

with $r = p(1-p)/(1-p^2)$. Transaction $-t_1$ simply removes t_1 if present in the sample, so that

$$P\{S_3 = \emptyset\} = 1 - r, \quad P\{S_3 = \{t_2\}\} = r,$$
$$Q_3 = Q_2 = p^L.$$

Transaction $+t_3$ triggers another purge operation if $S_3 = \{t_2\}$ and t_3 is accepted, which occurs with probability $Q_3 r$. It follows that

$$P\{S_4 = \{t_2\} \mid L\} = p^L(r^2 - r) + r, \quad P\{S_4 = \{t_3\} \mid L\} = p^L(r^2 - r + 1).$$

By unconditioning on L and simplifying the resulting infinite sum, we find that

$$P\{S_4 = \{t_2\}\} = \frac{p(p^2+1)}{(p+1)(p^2+p+1)}, \quad P\{S_4 = \{t_3\}\} = \frac{p}{p+1}.$$

If follows that $P\{S_4 = \{t_2\}\} < P\{S_4 = \{t_3\}\}$ for $0 < p \leq 1$, so that BSP biases the sample towards recent items. For example, a common choice is $p = 0.8$; the two probabilities are then given by ≈ 0.30 and ≈ 0.44, respectively.

The purge operation thus introduces some subtle dependencies among the sample items, and these dependencies lead to non-uniform samples when the transaction sequence contains deletions. BSP appears to work in the insertion-only setting, but this assertion has not yet been proven. We conjecture that BSP can be fixed by applying the random-pairing idea given in this paper. In this case, however, the main advantage of BSP, which is its simplicity, vanishes.

Appendix B: Stochastic optimization techniques for resizing

As discussed in Sect. 4.3.2, deterministic numerical optimization algorithms cannot be used to tune the resizing algorithm in the presence of deletions. In this appendix we develop a Monte Carlo algorithm, called FDSA, for finding the optimal value q^* of the resizing parameter. We first discuss some pertinent structural properties of the optimization problem, and then present our FDSA algorithm.

B.1 Search-space characteristics

The optimization problem has some special structure. Certain aspects of this structure facilitate numerical computation of q^*, whereas other aspects give rise to numerical challenges. We outline these considerations below.

Our first observation is that, for practical purposes, we can restrict our search for q^* to the interval $I = [M/|R|, M'/|R|]$. For $q > M'/|R|$, the cost of phase 2 is 0 with high probability, because it is unlikely that $U < M'$. Similarly, the cost of phase 1 equals its maximum possible value $g(M'/|R|)$ with high probability. The main reason that q^* might strictly exceed $M'/|R|$ would be to drive the (already extremely small) probability of the unlikely event $\{U < M'\}$ to 0; the occurrence of this unlikely event would typically incur a positive cost, because usually $t_b \gg t_a$ whenever q^* is large, so that any reduction in phase 1 cost is more than offset by an increase in phase 2 cost. In this case, however, we can achieve the same effect in practice by simply growing the sample using only the base data (eliminating phase 2) whenever $q^* \geq M'/|R|$. A similar argument holds for the lower boundary $M/|R|$.

Our next observation is derived from simulation-based estimates of the expected resizing time $T(q)$, as well as the closed-form approximations to $T(q)$ developed in Sect. 4.3.3. These results strongly indicate that the function T is non-constant and convex on the interval I. Such a cost structure implies that T is either increasing on I with $q^* = M/|R|$, decreasing on I with $q^* = M'/|R|$, or has a unique minimum $q^* \in (M/|R|, M'/|R|)$. This cost structure implies that local optimization algorithms—i.e., algorithms that search for a local, rather than a global, minimum—will suffice, so that expensive global optimization algorithms are not needed.

A final observation, again based on our simulation experiments and approximations, is that a numerical difficulty can arise when the parameters t_a, t_b, M, M' and $|R|$ are such that q^* is equal to or slightly greater than $q_l = M/|R|$. The problem is that, in such scenarios, the cost function T is often almost flat just to the right of the point q_l; see, for example, the curve corresponding to $t_a = 90$ in Fig. 12. To the left of the point q_l, however, the T function—which essentially equals the phase 2 cost function T_2 in this region—behaves roughly as $O(1/q)$ (see Sect. 4.3.3), and hence $T(q)$ rises steeply as q decreases. The net effect is that the derivative of the function T can be almost discontinuous at q_l, rapidly increasing from a large negative value to a relatively small positive value as q crosses this critical point from below. This quasi-discontinuity can cause problems for numerical optimization algorithms that try to estimate the derivative of the T function at candidate q values. In the sequel, we discuss techniques for dealing with this issue.

B.2 The FDSA algorithm

Since we cannot, in general, compute $T(q)$ exactly, an alternative, Monte Carlo-based approach is to generate realizations of the random resizing cost C via stochastic simulation, and use these observations to estimate q^*. A naive approach uses a standard deterministic numerical optimization algorithm; whenever a value of $T(q)$ is called for, we simulate the resizing algorithm multiple times, and average the resulting observations to obtain the requested function value. This naive approach is typically inefficient, requiring many expensive simulation runs. Moreover, the presence of noise can cause the optimization algorithm to behave erratically or even fail, since the algorithm was designed under the assumption that observations of the objective function are exact. For example, the standard Golden Section search routine [38, Sect. 10.1] assumes that, at all times, there are three points that "bracket" the true minimum point; in the presence of noise, this property may no longer hold, throwing the algorithm into confusion. It is possible to modify certain simple deterministic algorithms to handle noisy observations, but the performance of the modified algorithms tends to be poor, and there is little supporting theory available to provide the user with confidence that such an algorithm will converge to a solution.

A much better approach is to employ a stochastic optimization algorithm that is specifically designed to deal with noisy observations; see [42] for an excellent introduction to such algorithms. As mentioned in Sect. 7, a local optimization algorithm suffices for our problem. The most widely-used class of local stochastic optimization algorithms are the *stochastic approximation* (SA) algorithms. An SA algorithm is a steepest-descent method that rests on a recursion of the form

$$q_{k+1} = q_k - a_k \hat{G}(q_k) \tag{25}$$

for $k \geq 1$, where a_k is called the *gain* parameter and $\hat{G}(q_k)$ is an estimate of $G(q_k)$, the gradient (i.e., derivative) of the objective function T, evaluated at q_k. Each a_k is positive and $a_k \to 0$ as $k \to \infty$. Particular SA algorithms are specified by the manner in which the gain sequence $a_1, a_2, \ldots$ is determined and by the form of the estimator $\hat{G}$ used to estimate the gradient function G.

For our problem, the most suitable method is to estimate G using central finite differences.[10] Specifically, at the kth iteration, we set

$$\hat{G}(q_k) = \frac{\bar{C}_l(q_k + b_k) - \bar{C}_l(q_k - b_k)}{2b_k},$$

where $\bar{C}_l(q)$ denotes the average of l observations of the random resizing cost C, obtained from l independent simulations under parameter setting q; see [42, Chap. 6] for a detailed discussion of such "finite-difference stochastic approximation" (FDSA) algorithms, also known as Kiefer-Wolfowitz algorithms. Each b_k is positive and $b_k \to 0$ as $k \to \infty$. Perhaps surprisingly, it usually suffices to take $l = 1$ in the above formula, i.e., we use only two simulation runs at each iteration. The key insight underlying the SA approach, originally expressed in the seminal paper of Robbins and Monro [39], is that averaging across successive iterations—even though the evaluation point q changes from iteration to iteration—is a more effective use of computing resources than expending a lot of simulation effort at each iteration to get an accurate estimate of G. Under technical regularity conditions on $T(q)$ (which appear to hold in our current setting), the sequence of q_ks converges to q^* with probability 1 as $k \to \infty$, provided that the a_k and b_k sequences satisfy $\sum_{k=1}^{\infty} a_k = \infty$, $\sum_{k=1}^{\infty} a_k b_k < \infty$, and $\sum_{k=1}^{\infty} a_k^2/b_k^2 < \infty$. Following [42, Chap. 6], we use sequences of the form

$$a_k = \frac{a}{(k+A)^{\alpha}} \quad \text{and} \quad b_k = \frac{b}{k^{\beta}},$$

[10] An alternative approach would be to estimate G using "likelihood ratio" techniques, but such methods appear to be unusable in our setting due to highly unstable behavior. "Retrospective optimization" methods, which also use likelihood ratios, are also unsuitable.

where the parameters a, A, α, b, and β are chosen according to the "semiautomatic" method of [42, Sect. 6.6], starting with a value of $A = 5$. We also use the standard stabilization technique of returning as the final result not the value of the final iterate, but rather the average of the last few iterates computed.

Our only significant modification of the standard FDSA algorithm handles the difficult case where q^* is close to $M/|R|$. As discussed in the previous subsection, the gradient function G is almost discontinuous at this point, which can lead to numerical difficulties. We therefore ensure that the lower of the two endpoints used to compute the FD gradient estimate $\hat{G}$ is never less than $M/|R|$. This modification has the effect of replacing the usual central finite-difference estimate by essentially a one-sided finite-difference estimate whenever q^* is close to $M/|R|$. Because the effective value of b_k can become small in this boundary case, we actually average the results of four simulations at each endpoint, in order to further stabilize the gradient estimator.

Appendix C: Alternative cost models for resizing

Our first example concerns a more complicated model of base-data accesses. Suppose that the dataset is stored and retrieved in blocks of $b > 1$ items, as is typical for relational tables in commercial RDBMSs, and that t_a now represents the cost of accessing a block. (For simplicity, we assume that each block is completely filled, so that it contains *exactly* b items.) Also suppose that $|R| \gg |S|$, so that the probability that an item accessed during phase 1 was already in the sample prior to the start of resizing is negligible. Finally, suppose that, given the number $N(U)$ of base-data items that will be accessed during phase 1, the sampling mechanism is smart enough to precompute the IDs of the sampled items and determine in advance the set of blocks that will need to be retrieved, so that each block is only accessed once. Again, RDBMSs typically employ such a strategy in order to minimize I/Os. Letting $I_j = 1$ if at least one randomly sampled item belongs to the jth block and $I_j = 0$ otherwise ($1 \le j \le |R|/b$), the expected cost of phase 1, given that $U = u$, is

$$\begin{aligned} T_1(q;u) &= t_a E\left[\sum_{j=1}^{|R|/b} I_j\right] = t_a \sum_{j=1}^{|R|/b} E[I_j] \\ &= t_a \sum_{j=1}^{|R|/b} P\left\{ I_j > 0 \right\} = \frac{t_a|R|}{b} P\left\{ I_1 > 0 \right\} \\ &= \frac{t_a|R|}{b}(1 - P\{ I_1 = 0 \}) = \frac{t_a|R|}{b}\left(1 - \frac{\binom{|R|-b}{N(u)}}{\binom{|R|}{N(u)}}\right), \end{aligned}$$

where we have used the fact that the I_j's are identically distributed. Then $T_1(q)$ can be computed numerically as $T_1(q) = \sum_{u=0}^{|R|} T_1(q;u)\binom{|R|}{u} q^u (1-q)^{|R|-u}$, or, proceeding as in the previous section, approximated as

$$\hat{T}_1(q) = \frac{t_a|R|}{b}\left(1 - \frac{\binom{|R|-b}{N(|R|q)}}{\binom{|R|}{N(|R|q)}}\right).$$

A even simpler cost model is obtained if we assume that the data items were initially assigned to blocks in a random manner, so that we can randomly sample the dataset simply by sequentially reading pages. In this case, the conditional data access cost is computed simply as $T_1(q;u) = t_a N(u)/b$.

Our remaining examples concern more sophisticated models for the stream of arriving transactions during phase 2. For the first example, observe that our exact numerical results and approximations remain formally the same if arriving transactions are not necessarily equally spaced, but arrive according to a stationary stochastic process on the real line. Our only requirement is that the arrival times of transactions be statistically independent of whether each transaction is an insertion or a deletion. In this case, we simply define t_b as the expected time between successive Bernoulli trials. To generalize the model further, suppose that the transaction stream comprises random blocks of consecutive deletions and consecutive insertions. Specifically, at the beginning of a block, with probability p, the next K_1 transactions will be insertions, while with probability $1-p$, the next K_2 transactions will be insertions, where K_1 and K_2 are positive-integer-valued random variables with finite mean. Because the dataset is growing, we assume that $pE[K_1] - (1-p)E[K_2]$, the expected change in the dataset size due to a block of transactions, is positive. In this case, appealing to results for "renewal reward processes" [40], we find that the long-run, per-transaction rate at which the database is growing is

$$\rho = \frac{pE[K_1] - (1-p)E[K_2]}{pE[K_1] + (1-p)E[K_2]}.$$

Provided that, with high probability, K_1 and K_2 are relatively small with respect to the time scale of the problem—for example, as represented by $(M'-M)/(M'/|R|)$—we can approximate the expected phase 2 cost as $\hat{T}_2(q) = t_b(M' - |R|)^+/(\rho q)$. The approximation in Sect. 4.3.3 corresponds to the special case where $P\{K_1 = 1\} = P\{K_2 = 1\} = 1$.

References

1. Babcock, B., Datar, M., Motwani, R.: Sampling from a moving window over streaming data. In: Proc. SODA, pp. 633–634 (2002)
2. Brown, P., Haas, P., Myllymaki, J., Pirahesh, H., Reinwald, B., Sismanis, Y. : Toward automated large-scale information integration and discovery. In: Härder, T., Lehner, W. (eds.) Data Management in a Connected World, pp. 161–180. Springer, Heidelberg (2005)
3. Brown, P., Haas, P.J.: BHUNT: automatic discovery of fuzzy algebraic constraints in relational data. In: Proc. VLDB, pp. 668–679 (2003)

4. Brown, P.G., Haas, P.J.: Techniques for warehousing of sample data. In: Proc. ICDE (2006)
5. Chaudhuri, S., Motwani, R., Narasayya, V.R.: On random sampling over joins. In: Proc. ACM SIGMOD, pp. 263–274 (1999)
6. Colt Library: Open source libraries for high performance scientific and technical computing in Java. http://dsd.lbl.gov/ hoschek/colt/
7. Cormode, G., Muthukrishnan, S., Rozenbaum, I.: Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In: Proc. VLDB, pp. 25–36 (2005)
8. Fan, C., Muller, M., Rezucha, I.: Development of sampling plans by using sequential (item by item) techniques and digital computers. J. Am. Statist. Assoc. **57**, 387–402 (1962)
9. Frahling, G., Indyk, P., Sohler, C.: Sampling in dynamic data streams and applications. In: Proc. 21st Symp. Computat. Geom., pp. 142–149 (2005)
10. Gemulla, R., Lehner, W.: Deferred maintenance of disk-based random samples. In: Proc. EDBT, pp. 423–441 (2006)
11. Gemulla, R., Lehner, W., Haas, P.J.: A dip in the reservoir: Maintaining sample synopses of evolving datasets. In: Proc. VLDB, pp. 595–606 (2006)
12. Gemulla, R., Lehner, W., Haas, P.J.: Maintaining Bernoulli samples over evolving multisets. In: Proc. ACM PODS, pp. 93–102 (2007)
13. Gibbons, P., Matias, Y., Poosala, V.: AQUA project white paper. Tech. rep., Bell Laboratories, Murray Hill (1997)
14. Gibbons, P.B., Matias, Y.: New sampling-based summary statistics for improving approximate query answers. In: Proc. ACM SIGMOD, pp. 331–342 (1998)
15. Gibbons, P.B., Matias, Y., Poosala, V.: Fast incremental maintenance of approximate histograms. ACM Trans. Database Syst. **27**, 182–184 (2002)
16. GSL: GNU Scientific Library. http://www.gnu.org/software/gsl/
17. Haas, P., König, C.: A bi-level Bernoulli scheme for database sampling. In: Proc. ACM SIGMOD, pp. 275–286 (2004)
18. Haas, P.J.: Data stream sampling: Basic techniques and results. In: Garofalakis, M., Gehrke, J., Rastogi, R. (eds.) Data Stream Management: Processing High Speed Data Streams, Springer, Heidelberg (2007)
19. Halevy, A.Y., Etzioni, O., Doan, A., Ives, Z.G., Madhavan, J., McDowell, L., Tatarinov, I.: Join synopses for approximate query answering. In: Proc. CIDR (2003)
20. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online aggregation. In: Proc. ACM SIGMOD, pp. 171–182 (1997)
21. IBM Corporation: WebSphere Profile Stage User's Manual (2005)
22. Ilyas, I.F., Markl, V., Haas, P.J., Brown, P., Aboulnaga, A.: CORDS: automatic discovery of correlations and soft functional dependencies. In: Proc. ACM SIGMOD, pp. 647–658 (2004)
23. Jermaine, C., Pol, A., Arumugam, S.: Online maintenance of very large random samples. In: Proc. ACM SIGMOD, pp. 299–310 (2004)
24. John, G.H., Langley, P.: Static versus dynamic sampling for data mining. In: Proc. KDD, pp. 367–370 (2005)
25. Johnson, N.L., Kotz, S., Kemp, A.W.: Discrete Univariate Distributions, 2nd edn. Wiley, New York (1992)
26. Kachitvichyanukul, V., Schmeiser, B.: Computer generation of hypergeometric random variables. J. Stat. Comput. Simul **22**, 127–145 (1985)
27. Kivinen, J., Mannila, H.: The power of sampling in knowledge discovery. In: Proc. ACM PODS, pp. 77–85 (1994)
28. Knuth, D.E.: The Art of Computer Programming, vol. 2: Seminumerical Algorithms, 1st edn. Addison-Wesley, Reading (1969)
29. Law, A.M.: Simulation Modeling and Analysis, 4th edn. McGraw-Hill, New York (2007)
30. L'Ecuyer, P.: Uniform random number generation. In: Henderson, S.G., Nelson, B.L. (eds.) Simulation, pp. 55–81. Elsevier, Amsterdam (2006)
31. Leser, U., Naumann, F.: (Almost) hands-off information integration for the life sciences. In: Proc. CIDR, pp. 131–143 (2005)
32. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1), 3–30 (1998)
33. McLeod, A.I., Bellhouse, D.R.: A convenient algorithm for drawing a simple random sample. Appl. Statist. **32**, 182–184 (1983)
34. Norris, J.R.: Markov Chains. Cambridge University Press, Cambridge (1997)
35. Olken, F.: Random sampling from databases. Thesis LBL-32883, Information and Computing Sciences Division, Lawrence Berkeley National Laboratory (1993)
36. Olken, F., Rotem, D.: Maintenance of materialized views of sampling queries. In: Proc. ICDE (1992)
37. Poosala, V., Haas, P.J., Ioannidis, Y.E., Shekita, E.J.: Improved histograms for selectivity estimation of range predicates. In: Proc. ACM SIGMOD, pp. 294–305 (1996)
38. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C, 2nd edn. Cambridge University Press, Cambridge (1992)
39. Robbins, H., Monro, S.: A stochastic approximation method. Ann. Math. Statist. **22**, 400–407 (1951)
40. Ross, S.M.: Stochastic Processes. Wiley, New York (1983)
41. Särndal, C.E., Swensson, B., Wretman, J.: Model Assisted Survey Sampling. Springer, Heidelberg (1992)
42. Spall, J.C.: Introduction to Stochastic Search and Optimization. Wiley, New York (2003)
43. Tatbul, N., Çetintemel, U., Zdonik, S.B., Cherniack, M., Stonebraker, M.: Load shedding in a data stream manager. In: Proc. VLDB, pp. 309–320 (2003)
44. Vitter, J.S.: Faster methods for random sampling. Commun. ACM **27**(7), 703–718 (1984)
45. Vitter, J.S.: Random sampling with a reservoir. ACM Trans. Math. Softw. **11**(1), 37–57 (1985)
46. Zechner, H.: Efficient sampling from continuous and discrete distributions. Ph.D. thesis, Technical University Graz (1997)