TECHNISCHE UNIVERSITÄT DRESDEN

Sampling Algorithms for Evolving Datasets

Dissertation

zur Erlangung des akademischen Grades
Doktor rerum naturalium (Dr. rer. nat.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Inf. Rainer Gemulla
geboren am 28. April 1980 in Sondershausen

verteidigt am 20. Oktober 2008

Gutachter: Prof. Dr.-Ing. Wolfgang Lehner
Technische Universität Dresden
Fakultät Informatik, Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden

Dr. Peter J. Haas
IBM Almaden Research Center, K55/B1
650 Harry Road, San Jose, CA 95120-6099
USA

Prof. Dr.-Ing. Dr. h.c. Theo Härder
Technische Universität Kaiserslautern
Fachbereich Informatik
AG Datenbanken und Informationssysteme
67653 Kaiserslautern

Dresden im Oktober 2008

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 the processing of analytic queries and data-mining tasks, to enhance query optimization, and to facilitate information integration. Most of the existing work on database sampling focuses on how to create or exploit a random sample of a static database, that is, a database that does not change over time. The assumption of a static database, however, severely limits the applicability of these techniques in practice, where data is often not static but continuously evolving. In order to maintain the statistical validity of the sample, any changes to the database have to be appropriately reflected in the sample.

In this thesis, we study efficient methods for incrementally maintaining a uniform random sample of the items in a dataset in the presence of an arbitrary sequence of insertions, updates, and deletions. We consider instances of the maintenance problem that arise when sampling from an evolving set, from an evolving multiset, from the distinct items in an evolving multiset, or from a sliding window over a data stream. Our algorithms completely avoid any accesses to the base data and can be several orders of magnitude faster than algorithms that do rely on such expensive accesses. The improved efficiency of our algorithms comes at virtually no cost: the resulting samples are provably uniform and only a small amount of auxiliary information is associated with the sample. We show that the auxiliary information not only facilitates efficient maintenance, but it can also be exploited to derive unbiased, low-variance estimators for counts, sums, averages, and the number of distinct items in the underlying dataset.

In addition to sample maintenance, we discuss methods that greatly improve the flexibility of random sampling from a system's point of view. More specifically, we initiate the study of algorithms that resize a random sample upwards or downwards. Our resizing algorithms can be exploited to dynamically control the size of the sample when the dataset grows or shrinks; they facilitate resource management and help to avoid under- or oversized samples. Furthermore, in large-scale databases with data being distributed across several remote locations, it is usually infeasible to reconstruct the entire dataset for the purpose of sampling. To address this problem, we provide efficient algorithms that directly combine the local samples maintained at each location into a sample of the global dataset. We also consider a more general problem, where the global dataset is defined as an arbitrary set or multiset expression involving the local datasets, and provide efficient solutions based on hashing.

Acknowledgments

In a recent conversation, my thesis adviser mentioned to me that he had read a thesis in which the acknowledgment section started with the words "First of all, I'd like to thank my adviser [...]." He was really excited about this—constantly repeating the "first of all" part of the introductory sentence. I'm not sure if he made good fun or if he intended to make clear to me how to start my very own acknowledgment section. I'll play it safe: First of all, I'd like to thank my adviser Wolfgang Lehner. Wolfgang sparked my interest in data management and teached me everything I know about it. He always had time for discussions; his enthusiasm made me feel that my work counts. Whenever I had problems of organizational or motivational nature, a conversation with him quickly made them disappear. Always trusting my skills, he gave me the freedom to approach scientific problems my way. Thank you for everything.

I am deeply grateful to Peter Haas, who essentially acted like a co-adviser for my work. Peter contributed significantly to this thesis and to my knowledge about sampling and probability in general. He is a great mentor, inspiring me with his passion. I want to thank Theo Härder for co-refereeing this thesis; Kevin Beyer, Berthold Reinwald, Yannis Sismanis, and Paul Brown for working with me and for fruitful discussions; and Frank Rosenthal, Simone Linke, and Benjamin Schlegel for proof-reading parts of this thesis—it draws much from their comments. I like to thank Henrike Berthold, for introducing me to the field of database sampling and for writing the proposal that made possible this work; Benjamin Schlegel and Philipp Rösch, for providing diversion and lending me their ears when I wanted to discuss something; Ines Funke, for fighting me through that jungle of bureaucracy (and, of course, for providing me with candy); Anja, Bernhard, Felix, Marcus, Norbert, Sebastian, Steffen, Stephan, Torsten, and Ulrike, for their invaluable help within the Derby/S project; and Anja, Bernd, Bernhard, Christian, Dirk, Eric, Frank, Hannes, Henrike, Maik, Marc, Martin, Matze, Peter, Steffen, Sven, and Thomas, for being great colleagues and friends.

All this work would not have been possible without the constant support of my family and friends. I like to thank my grand-parents, my parents and my sister for their rock-solid confidence in my doings. I like to thank all my friends for the great time we had together. And I want to thank my wife and my little son for helping me through this tough time, for providing encouragement, for accepting not seeing me many evenings, and for just being the family I love.

Rainer Gemulla
August 27, 2008

I am grateful to the Deutsche Forschungsgemeinschaft for providing the funding for my work under grant LE 1416/3-1.

Contents

Chapter 1

Introduction

I got this strange idea that maybe I could study the Bible
the way a scientist would do it, by using random sampling.
The rule I decided on was we were going to study
Chapter 3, Verse 16 of every book of the Bible.
This idea of sampling turned out to be
a good time-efficient way to get into a complicated subject.

— Donald Knuth (2008)

Recent studies conducted by IDC (2007, 2008) have revealed that the 2007 "digital universe" comprises about 45 gigabytes of data per person on the planet. Looking only at the data stored in large-scale data warehouses, Winter (2008) estimates that the size of the world's largest warehouse triples about every two years, thereby even exceeding Moore's law. To analyze this enormous amount of data, random sampling techniques have proven to be an invaluable tool. They have numerous applications in the context of data management, including query optimization, load balancing, approximate query processing, and data mining. In these applications, random sampling techniques are exploited in two fundamentally different ways: (i) they help compute an *exact* query result *efficiently* and/or (ii) they provide means to *approximate* the query result. In both cases, the use of sampling may significantly reduce the cost of query processing.

For an example of (i), consider the problem of deriving an "execution plan" for a query expressed in a declarative language such as SQL. There usually exist several alternative plans that all produce the same result but they can differ in their efficiency by several orders of magnitude; we clearly want to pick the plan that is most efficient. In the case of SQL, finding the optimal plan includes (but is not limited to) decisions on the indexes to use, on the order to apply predicates, on the order to process joins, and on the type of sort/join/group-by algorithm to use. Query optimizers make this decision based on estimates of the size of intermediate results. Virtually all major database vendors—including IBM, Microsoft, Oracle, Sybase, and Teradata—use random sampling to compute online and/or precompute offline statistics that can be leveraged for query size estimation. This is because a small random sample of the data often provides sufficient information to separate efficient and inefficient plans.

Perhaps the most prevalent example of (ii) is approximate query processing. The key idea behind this processing model is that the computational cost of query processing can be reduced when the underlying application does not require exact results but only a highly-accurate estimate thereof. For instance, query results visualized in a pie chart may not be required to be exact up to the last digit. Likewise, exploratory "data browsing"—carried out in order to find out which parts of a dataset contain interesting information—greatly benefits from fast approximate query answers. It is not surprising that random sampling is one of the key technologies in approximate query processing. There exists a large body of work on how to compute and exploit random samples; results obtained from a random sample can be enriched with information about their precision and, if desired, progressively refined; and sampling scales well with the size of the underlying data. Recognizing the importance of random sampling for approximate query processing, the SQL standardization committee included basic sampling clauses into the SQL/Foundation:2003 standard; these clauses are already implemented in most commercial database systems.

So far, we have outlined some applications of random sampling but we have not discussed how to actually compute the sample. The key alternatives for sample computation are "query sampling" (compute when needed) and "materialized sampling" (compute in advance). In this thesis, we concentrate almost entirely on materialized sampling. One of the key advantages of materialized sampling is that the cost of obtaining the sample amortizes over its subsequent usages. We can "invest" in sophisticated sampling designs well-suited for our specific application, even if such a sample were too costly to obtain at query time. Another distinctive advantage of materialized sampling is that access to the underlying data is not required in the estimation process. The more expensive the access to the actual data, the more important this property gets. In fact, base data accesses may even be infeasible in applications in which, for instance, the underlying dataset is not materialized or resides at a remote location.

The key challenge we face in materialized sampling is that real-world data is not static; it *evolves* over time. To ensure the statistical validity of the estimates derived from the sample, changes of the underlying data must be incorporated. The availability of efficient algorithms for sample maintenance is therefore an important factor for the practical applicability of materialized sampling. There do exist several efficient maintenance algorithms, but many of these are restricted to the class of append-only datasets, in which data once inserted is never changed or removed. In contrast, this thesis contributes novel maintenance algorithms for the *general* class of evolving datasets, in which the data is subject to insertion, update and deletion transactions. As a consequence, our algorithms extend the applicability of materialized sampling techniques to a broader spectrum of applications.

Summary of Contributions

In more detail, our main contributions are:

1. We survey the recent literature on database sampling. To the best of our knowledge, the last comprehensive survey of database sampling techniques was undertaken by Olken (1993). There has been a tremendous amount of work since then; our survey focuses on the key results, structured coarsely by their application area.

2. We review available maintenance techniques for the class of uniform random samples. In particular, we classify maintenance schemes along the following dimensions: types of datasets supported, types of transactions supported, sampling semantics, sample size guarantees, I/O cost, and memory consumption. We point out scenarios for which no efficient sampling techniques are known. We also disprove the statistical correctness of a few of the techniques proposed in the literature.

3. We present several novel maintenance algorithms for uniform samples under insertion, update, and deletion transactions. Compared to previously known algorithms, our algorithms have the advantage of being "incremental": they maintain the sample without ever accessing the underlying dataset. The main theme behind our algorithms is that of "compensating insertions": We allow the sample size to shrink whenever one of the sampled items is removed from the underlying dataset, but we compensate the loss in sample size with future insertions. We apply unique variations of this principle to obtain maintenance algorithms for set sampling, multiset sampling, distinct-item sampling, and data stream sampling.

4. We propose novel algorithms for resizing uniform samples. Techniques for resizing a sample upwards help avoid the loss in accuracy of the estimates that may result from changes of the underlying data. Conversely, techniques for resizing a sample downwards prevent oversized samples and thus overly high resource consumption. Again, expensive accesses to the underlying dataset in order to resize the sample are avoided whenever possible and minimized otherwise.

5. We present novel techniques for combining two or more uniform samples. Such techniques are particularly useful when the underlying datasets are distributed or expensive to access; information about the combined dataset can be obtained from the combined sample. More specifically, we consider the problem of computing a uniform sample from the union of two or more datasets from their local samples. We present novel algorithms that supersede previously known algorithms in terms of sample size. We also consider the more general problem of computing samples of arbitrary (multi)set expressions—including union, intersection, and difference operations—based solely on the local samples of the involved datasets. It is known that, in general, excessively large local sample sizes are required to obtain a sufficiently large combined sample. However, we extend earlier results on "min-hash samples" to show that min-hash samples

are well suited for this problem under reasonable conditions on the size of the expression.

6. We complement our results with improved estimators for certain population parameters. Our estimators exploit the information that is stored jointly with the sample in order to facilitate maintenance. Parameters of interest include sums, averages, the dataset size, and the number of distinct items in the dataset; all of our estimators are unbiased and have lower variance than previously known estimators.

Roadmap

Chapter 2 starts with a brief introduction to the theory of finite population sampling, which underlies all database sampling techniques. The main part of that chapter surveys techniques for database sampling and their applications. In chapter 3, we shift our attention to the actual computation and maintenance of random samples. The chapter reviews and classifies available maintenance techniques. Chapters 4 to 7 contain our novel results for set sampling, multiset sampling, distinct-item sampling, and data stream sampling; an entire chapter is devoted to each of these topics. Chapter 8 concludes this thesis with a summary and a discussion of open research problems.

Chapter 2

Literature Survey

The purpose of this chapter is to give a concise overview of the fundamentals of random sampling as well as its applications in the area of databases.

We start with an introduction to the theory of finite population sampling, also called survey sampling, in section 2.1. This theory provides principle methods to select a sample from a "population" or dataset of interest and to infer information about the entire population based on the information in the sample. As Särndal et al. (1991) pointed out, survey sampling has a vast area of applications: Nation-wide statistical offices make use of survey sampling to obtain information about the state of the nation; in academia, survey sampling is a major tool in areas such as sociology, psychology, or economy; and last but not least, sampling plays an important role for market research. Our discussion covers only the basics—most notably common sampling designs and estimators—but it will suffice to follow the rest of this thesis.

In the last decades, sampling has been applied to many problems that arise in the context of data management systems. To cope with the specifics of these systems, novel techniques beyond the scope of classical survey sampling have been developed. A brief comparison of survey sampling and database sampling is given in section 2.2, where we also discuss different approaches to database sampling. Section 2.3 is devoted entirely to database-related applications of sampling, including query optimization, approximate query processing and data mining. Note that this chapter does not cover sample creation and maintenance; these issues are postponed to subsequent chapters.

2.1 Finite Population Sampling

As indicated above, the theory of finite population sampling forms the basis of database and data stream sampling. We restrict our attention to those parts of the theory that are relevant for this thesis. Our notation follows common styles, but see the *index of notation* on page 263. Readers familiar with the theory can skip or skim-read this section.

2.1.1 Basic Ideas and Terminology

The data set of interest is called the *population* and the elements of the population are called *items*. Unless stated otherwise, we assume that the population is a set (i.e., it does not contain duplicates) and use

$$R = \{ r_1, \ldots, r_N \}$$

to denote a population of size N. For example, the population may comprise households of a city or employees of a company. It may also comprise tuples from a table of a relational database, tuples from a view of a relational database, XML documents from a collection of XML documents, lines of text from a log file, or items from a data stream. In this thesis, we assume that the population coincides with the dataset being sampled; the latter is usually called the sampling frame. We occasionally make use of the terms *base data*, *underlying dataset* or simply *dataset* to denote the population because these terms are more common in database literature.

A *sample* is a subset of the population. The method that is used to select the sample is called the *sampling scheme*. Following Särndal et al. (1991), a *probability sampling scheme* has the following properties:

1. It is possible (but not necessarily practicable) to define the set of samples $\mathscr{S} = \{ s_1, s_2, \ldots, s_m \}$ that can be produced by the sampling scheme.
2. For each sample $s \in \mathscr{S}$, the probability that the scheme produces s is known.
3. Every item of the population is selected with non-zero probability.

Samples produced by a probability sampling scheme are called *probability samples*. There are alternatives to probability sampling. Cochran (1977) lists haphazard sample selection, the selection of "typical" items by inspection and the restriction of the sampling process to the parts of the population that are accessible. Any of these alternative techniques might work well in specific situations, but the only way to determine whether or not the technique worked efficiently is to compare the resulting estimates with the quantity being estimated. As we will see later, the unique advantage of probability sampling is that the precision of the estimates derived from the sample can be estimated from the sample itself.

The probability distribution over the set of possible samples $\mathscr{S}$ is called the *sampling design*. Often, many possible schemes exist for a given design. For example, suppose that scheme A first selects a single item chosen uniformly and at random from the entire population and then independently selects a second item chosen uniformly and at random from the remaining part of the population. Also suppose that scheme B selects the first item as stated above. To select the second item, scheme B repeatedly samples uniformly and at random from the entire population until an item different to the first item is found. It is easy to see that both A and B lead to the same sampling design. Thus, a sampling scheme describes the *computation* of the sample, and the sample design describes the *outcome* of the sampling process.

Table 2.1: Common sampling designs

Class	Description
Uniform sampling	Select subsets of equal size with equal probability
Weighted sampling	Select items with probability proportional to item weight
Stratified sampling	Divide into strata and sample from each stratum
Systematic sampling	Select every k-th item
Cluster sampling	Divide into clusters and sample some clusters entirely

The particular design produced by A and B is called simple random sampling; it is one of the most versatile of the available sampling designs.

An *estimator* is a rule to derive an estimate of a population parameter from the sample. For any specified sample, the estimator produces a unique *estimate*. For example, to estimate the average income of a population of employees, one might take the average from a sample of the employees. Whether this specific estimator performs well or not depends on the sampling design. In general, estimator and sampling design influence each other. To analyze the properties of an estimator, one may analyze the distribution of the estimates found after having applied the estimator to each of the samples in $\mathscr{S}$.

In what follows, we first discuss common sampling designs (section 2.1.2) and then take a look at frequently used estimators for these designs (section 2.1.3).

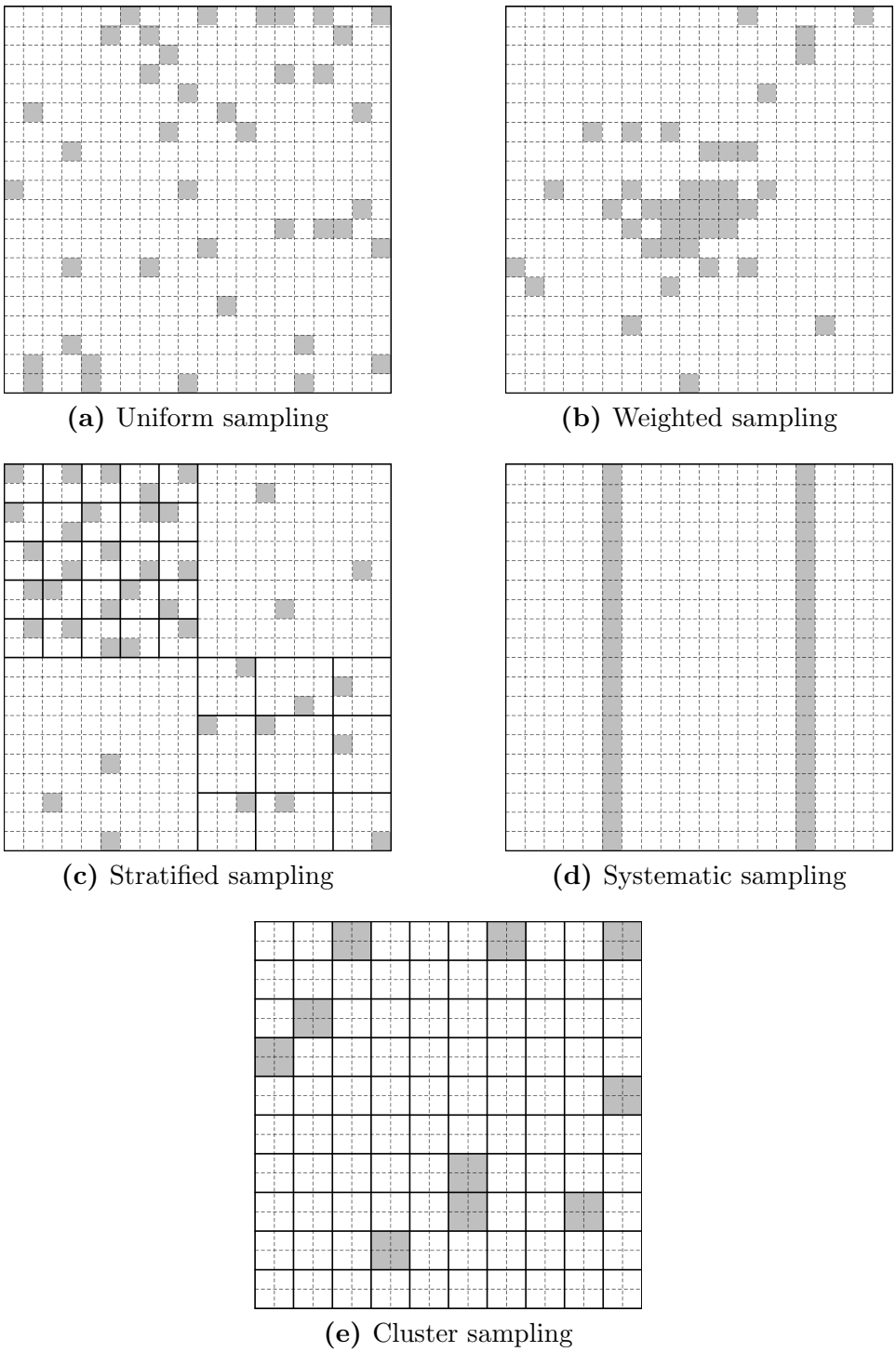
2.1.2 Sampling Designs

For a given population R and set $\mathscr{S}$ of possible samples, there is an infinite number of different sampling designs because there are infinitely many ways of assigning selection probabilities to the samples in $\mathscr{S}$. In practice, however, one does not directly decide on the sampling design but on the sampling scheme. This decision is based on both the cost of executing the scheme and the properties of the resulting sampling design. In this section, we discuss classes of sampling designs that are commonly found in practice. An overview is given in table 2.1 and figure 2.1.

A. Uniform Sampling

In a *uniform sampling* design, equally-sized subsets of the population are selected with equal probability. More formally, denote by S the random sample and fix two arbitrary subsets $A, B \subseteq R$ with $|A| = |B| = k$. Under a uniform design, it holds that

$$\Pr[S = A] = \Pr[S = B] = \frac{\Pr[|S| = k]}{\binom{N}{k}}, \tag{2.1}$$



(a) Uniform sampling

(b) Weighted sampling

(c) Stratified sampling

(d) Systematic sampling

(e) Cluster sampling

Figure 2.1: Illustration of common sampling designs ($N = 400$, $n = 40$). Adapted from Thompson (1992).

where $\Pr[S = A]$ denotes the probability of selecting subset A as the sample, and $\Pr[|S| = k]$ denotes the probability that the sample contains exactly k elements. The denominator of the final equality is a *binomial coefficient* defined as

$$\binom{N}{k} = \frac{N!}{k!(N-k)!},$$

where we take the convention $0! = 1$ and hence $\binom{0}{0} = 1$. The value of $\binom{N}{k}$ denotes the number of possible ways to select precisely k out of N items (disregarding their order); or, equivalently, the number of different subsets of size k. From (2.1), it follows immediately that every item has the same chance of being included in the sample.[1] A uniform sample of 40 items from a population of 400 items is shown in figure 2.1a.

Uniform sampling is the most basic of the available sampling designs. It is objective; no item is preferred over another one. Uniform samples capture the intuitive notion of randomness and are often considered representative. If information about R or the intended usage of the sample is unavailable at the time the sample is created, uniform sampling is the best choice. This situation occurs rarely in traditional survey sampling but is frequent in database sampling. Otherwise, when additional information is available, alternative sampling designs may be superior to uniform sampling. In this case, uniform sampling often acts as a building block for these more complex designs.

As becomes evident from equation (2.1), different uniform sampling designs differ in the distribution of the sample size $|S|$ only. The most common uniform designs are simple random sampling and Bernoulli sampling; we will come across other uniform designs in the course of this thesis.

Under the *simple random sampling* design (SRS), the sample size is a constant. For an SRS of size n, we have for each $A \subseteq R$

$$\Pr[S = A] = \begin{cases} 1/\binom{N}{n} & |A| = n \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

The sample in figure 2.1a is a simple random sample of size $n = 40$. We make use of the letter n whenever we refer to the obtained *sample size*; n is a synonym for $|S|$.

Under the *Bernoulli sampling* design (BERN), the sample size is binomially distributed. For a given sampling rate q, each item is included into the sample with probability q and excluded with probability $1 - q$; the inclusion/exclusion decisions are independent. We have

$$\Pr[S = A] = q^{|A|}(1-q)^{N-|A|} \tag{2.3}$$

for any fixed $A \subseteq R$ and

$$\Pr[|S| = k] = \binom{N}{k} q^k (1-q)^{N-k} \tag{2.4}$$

[1] The opposite does not hold, see section D on systematic sampling.

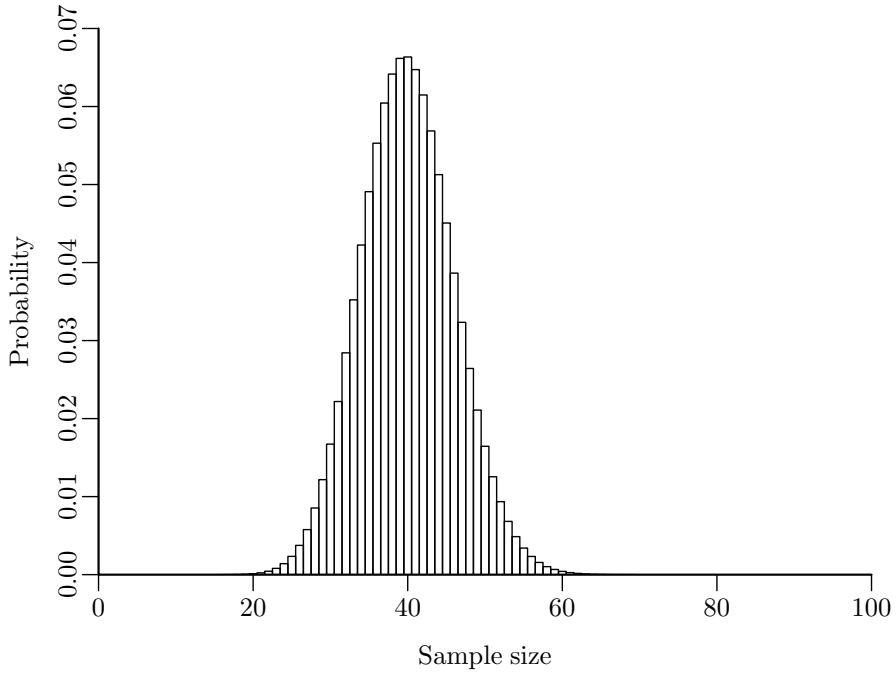


Figure 2.2: Sample size distribution of Bernoulli sampling ($N = 400$, $q = 0.1$)

for $0 \leq k \leq N$. The *binomial probability* in (2.4) is referred to frequently; we use the shortcut

$$B(k;\, N, q) \stackrel{\text{def}}{=} \binom{N}{k} q^k (1-q)^{N-k}, \tag{2.5}$$

so that $\Pr[\,|S| = k\,] = B(k;\, N, q)$. The sample size has mean qN and variance $q(1-q)N$; the distribution of the sample size for $N = 400$ and $q = 0.1$ is given in figure 2.2.

To distinguish the random sample size n from the "desired" sample size qN, we will consistently make use of the letter M, referred to as the *sample size parameter* of a design. The key difference between n and M is that n is a random variable while M is a constant.[2] Using this notation, an SRS of size M, where $1 \leq M \leq N$, is often preferable to a Bernoulli sample with $q = M/N$ for the purpose of estimation. Both designs have an expected sample size of M but the additional sample-size variance of Bernoulli sampling adds to the sampling error (Särndal et al. 1991). In practice, however, Bernoulli samples are sometimes used instead of simple random samples because the former are easier to manipulate.

Both SRS and BERN sample *without replacement*, meaning that each item sampled at most once. Alternatively, one can sample *with replacement*. In a with-replacement design, each item can be sampled more than once. Such a situation typically occurs

[2] There is no difference for SRS, though. Strictly speaking, we should have used M in our discussion of SRS and equation (2.2) but we chose n for expository reasons.

when items are drawn one by one, without removing already selected items from the population. The set $\mathscr{S}$ of possible outcomes then consists of *sequences* of items from R; order is important. A with-replacement design is called uniform if all equal-length sequences of items from R are selected with equal probability. As before, different uniform designs with replacement differ in the distribution of the sample size (length of sequence) only.

In *simple random sampling with replacement* (SRSWR), the sample size is a predefined constant M (so that $n = M$). For example, let $R = \{1, 2, 3\}$ and set $M = 2$. Possible samples are

$$\begin{array}{lll} (1,1), & (1,2), & (1,3), \\ (2,1), & (2,2), & (2,3), \\ (3,1), & (3,2), & (3,3) \end{array}$$

and each sample is selected with probability $1/9$. One can construct a uniform sample without replacement by removing duplicate items. The resulting sample is called the *net sample*, denoted by $D(S)$ for "set of distinct items in S". The unmodified sample is called the *gross sample*, denoted by S. For our example, the respective net samples are

$$\begin{array}{lll} \{1\}, & \{1,2\}, & \{1,3\}, \\ \{1,2\}, & \{2\}, & \{2,3\}, \\ \{1,3\}, & \{2,3\}, & \{3\}. \end{array}$$

The distribution of the net sample size is rather complex. After the i-th draw, denote by S_i the sample, by $|S_i|$ the number of items in the sample, including duplicates, and by $|D(S_i)|$ the number of distinct items in the sample. Then $|D(S_1)| = 1$ and

$$\Pr[\,|D(S_{i+1})| = k\,] = \frac{k}{N} \Pr[\,|D(S_i)| = k\,] + \frac{N-k+1}{N} \Pr[\,|D(S_i)| = k-1\,] \quad (2.6)$$

for $1 \leq k \leq i+1$. Figure 2.3 depicts an example of the resulting distribution. A non-recursive formula of $\Pr[\,|D(S_{i+1})| = k\,]$ is given in Tillé (2006, p. 55). The expected sample size is

$$\mathrm{E}[\,|D(S)|\,] = N\left(1 - \left(\frac{N-1}{N}\right)^M\right),$$

which evaluates to 38.11 in our example. When $M \ll N$, SRSWR performs nearly as well as SRS. Otherwise, Rao (1966) has shown that SRSWR is statistically less efficient, that is, estimates derived from the sample exhibit (slightly) higher estimation error. Nevertheless, drawing an SRSWR is sometimes less costly than drawing an SRS, even if the sample size of SRSWR is increased to compensate for the effect of duplicate items.

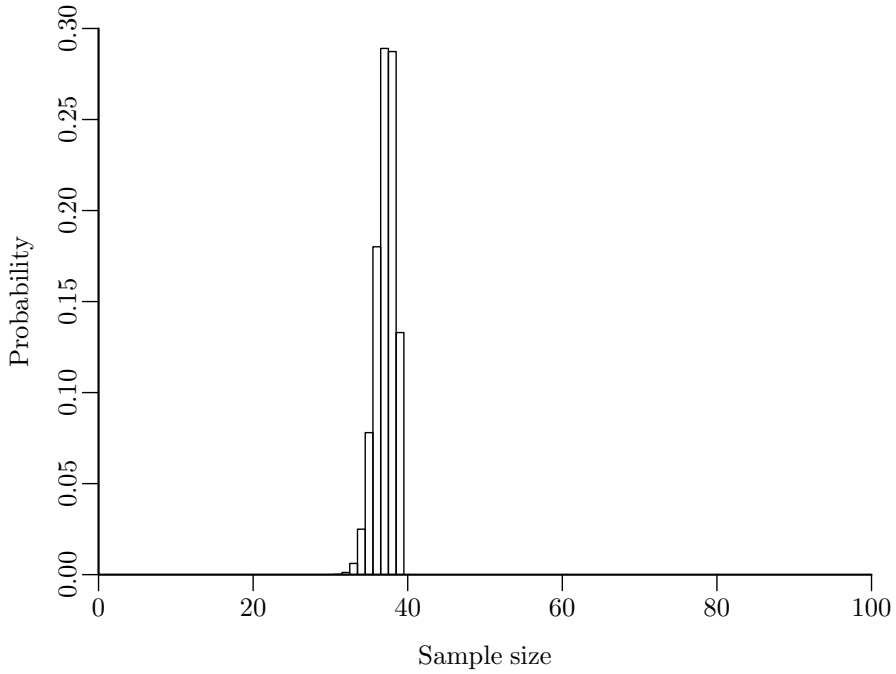


Figure 2.3: Sample size distribution of simple random sampling with replacement after duplicate removal ($N = 400$, $n = 40$)

B. Weighted Sampling

An important class of sampling designs are *weighted sampling* designs, which are also called probability-proportional-to-size (PPS) sampling designs. Unlike in uniform sampling or other equal-probability designs, the probability of an item being included into the sample varies among the items in the population. Weighted sampling is especially useful when some items in the population are considered more important than other items and the importance of each item can be quantified before the sample is created. The importance is modeled by associating a weight w_i with each item r_i in the data set, $1 \leq i \leq N$. The interpretation of the weight differs for without-replacement and with-replacement sampling. We restrict our discussion to without-replacement sampling; with-replacement designs are discussed in Särndal et al. (1991) and Thompson (1992).

Under a weighted sampling design without replacement, the inclusion probability of each item is proportional to its weight. Denote by

$$\pi_i \stackrel{\text{def}}{=} \Pr[\, r_i \in S \,]$$

the *first-order inclusion probability* of r_i. With M being the desired sample size, we have

$$\pi_i = M \frac{w_j}{\sum_j w_j} \tag{2.7}$$

assuming that the right hand side of the equation is always less than or equal to 1. Hanif and Brewer (1980) list and classify 50 different sampling schemes for weighted sampling without replacement. Most of these schemes lead to a unique sampling design. The designs differ in the higher-order inclusion probabilities, that is, the probabilities that two or more items occur jointly in the sample. For sample sizes larger than $M = 2$, most of the schemes are complex and/or lead to complex variance estimators.

We will only discuss Poisson sampling, a simple but common design. In *Poisson sampling*, each item r_i is accepted into the sample with probability π_i as in (2.7) and rejected with probability $1 - \pi_i$. The acceptance/rejection decisions are independent. We have

$$\Pr[S = A] = \prod_{r_i \in A} \pi_i \prod_{r_i \in R \setminus A} (1 - \pi_i)$$

for an arbitrary but fixed $A \subseteq R$. A realization of Poisson sampling is shown in figure 2.1b; the weight of each item is proportional to its squared distance from the center. The sample size is random; it has mean M and variance $\sum \pi_i(1 - \pi_i)$. Unless M is not too small, the sample size will stay close to M with high probability (Motwani and Raghavan 1995). In the special case where all π_i are equal, Poisson sampling is reduced to Bernoulli sampling and the sample size is binomially distributed.

C. Stratified Sampling

Under a *stratified sampling* design, the population is divided into strata in such a way that every item is contained in exactly one stratum. The strata therefore form a partitioning of the population. A separate sample is drawn independently from each stratum, typically using simple random sampling. In this case, the total sample size is constant. An example is given in figure 2.1c, where stratum boundaries are represented by solid lines. Precisely 3 items have been selected from each of the two large strata and 1 has been taken item from each of the smaller strata.

The placement of stratum boundaries and the allocation of the available sample size to the individual strata is a challenging problem, especially if multiple variables are of interest. A good overview of existing approaches is given in Cochran (1977). In general, when subpopulations are of interest (e.g., male and female), one may place each subpopulation in its own stratum. This way, it is guaranteed that the subpopulations are appropriately represented in the sample. When strata are chosen in such a way that the study variable is homogeneous within each stratum and heterogeneous between different strata, stratified sampling may produce a significant gain in precision compared to SRS.

D. Systematic Sampling

The *systematic sampling* design is an equal-probability design, that is, items are selected with equal probability. The design results from schemes that (1) order the

data set by some fixed criteria and (2) select every k-th item starting from an item chosen uniformly and at random from the first k items. Only k different samples can be selected under the design so that it is not uniform. Figure 2.1d shows a realization of systematic sampling with $k = 10$. The sample size is either $\lfloor N/k \rfloor$ or $\lceil N/k \rceil$ depending on the start item. Systematic sampling might be used in practice when drawing a uniform sample is considered too expensive or is too cumbersome to implement.

E. Cluster Sampling

Cluster sampling is used when the population is naturally divided into clusters of items (e.g., persons in a household or tuples in a disk block). The sample consists of entire clusters randomly chosen from the set of all clusters. Figure 2.1e shows a possible outcome. In the figure, cluster boundaries are represented by solid lines and simple random sampling has been used to select the clusters.

In contrast to stratified sampling, the primary goal of cluster sampling is to reduce the cost of obtaining the sample. The price is often a loss in precision, especially in the common case where the items within clusters are homogeneous and different clusters are heterogeneous. But still, the precision/cost ratio of a cluster sampling scheme may be higher than the ones of alternative schemes.

2.1.3 Estimation

Sampling is usually applied to estimate one or more *parameters* of the population. Parameters are, for example, the unemployment rate of a country's population, the labor cost of a company's employees, or the selectivity of a query on a relational table. We focus solely on real-valued parameters. More complex estimation tasks specific to databases are deferred to section 2.3.

We denote an *estimator* of population parameter θ by $\hat{\theta}$. Generally, estimator $\hat{\theta}$ is a function from a subset of R to a real value, that is, $\hat{\theta} : \mathcal{P}(R) \to \mathbb{R}$ with $\mathcal{P}(R)$ denoting the power set of R. For an arbitrary but fixed sample $s \in \mathscr{S}$, the quantity $\hat{\theta}(s)$ is a constant. The quantity $\hat{\theta}(S)$, however, is a random variable because the sample S itself is random. Clearly, we want to choose $\hat{\theta}$ such that $\hat{\theta}(S)$ approximates θ, that is, that $\hat{\theta}(S)$ is close to θ with high probability. For brevity, we will usually suppress the dependance on S and write $\hat{\theta}$ to denote $\hat{\theta}(S)$.

A. Properties of Estimators

Before we discuss estimators for common population parameters and sampling designs, we will briefly summarize some important properties of estimators in general. These properties help to determine how good an estimator performs in estimating a population parameter. They are often used to compare alternative estimators against each other. An overview is given in table 2.2.

The *expected value* of an estimator can be seen as the average value of the estimate when sampling is repeated many times. The *bias* is a measure of accuracy, the

Table 2.2: Important properties of estimators

Property	**Notation and definition**
Expected value	$\mathrm{E}[\hat{\theta}] = \sum_{s \in \mathscr{S}} \Pr[S = s]\,\hat{\theta}(s)$
Bias	$\mathrm{Bias}[\hat{\theta}] = \mathrm{E}[\hat{\theta}] - \theta$
Variance	$\mathrm{Var}[\hat{\theta}] = \mathrm{E}[(\hat{\theta} - \mathrm{E}[\hat{\theta}])^2] = \mathrm{E}[\hat{\theta}^2] - \mathrm{E}[\hat{\theta}]^2$
Standard error	$\mathrm{SE}[\hat{\theta}] = \sqrt{\mathrm{Var}[\hat{\theta}]}$
Coefficient of variation	$\mathrm{CV}[\hat{\theta}] = \dfrac{\mathrm{SE}[\hat{\theta}]}{\mathrm{E}[\hat{\theta}]}$
Mean squared error	$\mathrm{MSE}[\hat{\theta}] = \mathrm{Var}[\hat{\theta}] + \mathrm{Bias}[\hat{\theta}]^2$
Root mean squared error	$\mathrm{RMSE}[\hat{\theta}] = \sqrt{\mathrm{MSE}[\hat{\theta}]}$
Average relative error	$\mathrm{ARE}[\hat{\theta}] = \mathrm{E}\left[\dfrac{\lvert\hat{\theta} - \theta\rvert}{\theta}\right]$

degree of systematic error. It equals the expected deviation from the real population parameter. If an estimator has the desirable property of zero bias, it is said to be *unbiased*. An unbiased estimator is correct in expectation, that is, its expected value equals the population parameter it tries to estimate. Otherwise, the bias is non-zero and the estimator is *biased*. The *variance* is a measure of precision, the degree of variability. It equals the expected squared distance of the estimate from its expected value. A "large" variance means that the estimates heavily fluctuate around their expected value; a "low" variance stands for approximately stable estimates. What is considered large and low variance depends on the application and the parameter to be estimated (see below). Often, the *standard error* (square root of variance) is reported instead of the variance. The standard error is more comprehensible and has the same unit as θ. Going one step further, the *coefficient of variation* equals the standard error normalized by the expected value; it is unitless. A value strictly less than 1 indicates a low-variance distribution, whereas a value larger than 1 is often considered high variance.

The suitability of an estimator for a specific estimation problem depends on both its bias and its variance. A slightly biased estimator with a low variance may be preferable to an unbiased estimator with a high variance. The *mean squared error* (MSE) incorporates both bias and variance. It is equal to the expected squared distance to the true population parameter and is often used to compare different estimators. A low mean squared error leads to more precise estimates than a high mean squared error. The *root mean squared error* (RMSE) denotes the square root of the MSE and has the same unit as θ. The *average relative error* (ARE) denotes

the expected relative deviation of the estimate from θ. In contrast to the MSE, the ARE is normalized and unitless. It can therefore be used to explore the performance of an estimator across multiple data sets.

One of the main advantages of random sampling is that the variance of an estimator can itself be estimated from the sample. For some problems (e.g., estimating sums and averages), formulas for variance estimation are available and given below. Other, more complex problems require more sophisticated techniques, e.g., resampling methods such as bootstrapping and jackknifing.

B. Sums and Averages

Let $f : R \to \mathbb{R}$ be a function that associates a real value with each of the items in the population. For brevity, set

$$y_i = f(r_i)$$

for $1 \leq i \leq N$. We now describe how to estimate the population total

$$\tau = \sum_{r_i \in R} y_i$$

and the population average

$$\mu = \frac{1}{N} \sum_{r_i \in R} y_i = \frac{\tau}{N}.$$

All y_i, τ and μ are defined with respect to f, but we omit this dependency in our notation. Some typical choices for f are described by Haas (2009):

1. Suppose that each $r \in R$ has a numerical attribute A and let $f(r) = r.A$. Then, τ corresponds to the population total of attribute A and μ corresponds to the population average of A. For example, suppose that R comprises employees of a company and that, for each employee $r \in R$, $f(r)$ gives r's income. Then, τ corresponds to the total income of all employees and μ corresponds to the average income.

2. Let h be a predicate function such that $h(r) = 1$ whenever r satisfies a given predicate and $h(r) = 0$ otherwise. Set $f(r) = h(r)$. Then, τ corresponds to the number of items in R that satisfy the predicate and μ to the selectivity of the predicate. Picking up the above example, set $h(r) = 1$ whenever r is a manager and $h(r) = 0$ otherwise. Then, τ corresponds to the total number of managers and μ corresponds to the fraction of employees that are managers.

3. Defining $r.A$ and h as above, let $f(r) = h(r)r.A$. Then, τ corresponds to the sum of attribute A over the values that satisfy the predicate. In our example, τ corresponds to the total income of all managers.

Note that in the last case, μ does not correspond to the average value of the items that satisfy the predicate; it does not have any meaningful value. Averages over subpopulations can be expressed as a ratio of two sums—$(\sum h(r)r.A)/\sum h(r)$—and are discussed in Särndal et al. (1991, sec. 5.6).

We start with estimators for the population total τ. Horvitz and Thompson (1952) introduced an estimator that can be used with any probability design without replacement. It is given by

$$\hat{\tau}_{\text{HT}} = \sum_{r_i \in S} \frac{y_i}{\pi_i}, \tag{2.8}$$

where, as before, $\pi_i = \Pr[\, r_i \in S \,]$ denotes the first-order inclusion probability of r_i. In (2.8), each value is scaled-up or "expanded" by its inclusion probability. For this reason, the above sum is often called π*-expanded sum*.

The Horvitz-Thompson (HT) estimator is unbiased for τ. Its variance $\text{Var}[\,\hat{\tau}_{\text{HT}}\,]$ depends on the second-order inclusion probabilities

$$\pi_{ij} \stackrel{\text{def}}{=} \Pr[\, r_i \in S, r_j \in S \,]$$

and is given in table 2.3. The table also gives an unbiased estimator $\hat{\text{Var}}[\,\hat{\tau}_{\text{HT}}\,]$ of the variance of $\hat{\tau}_{\text{HT}}$ from the sample. The variance estimator is guaranteed to be non-negative when all $\pi_{ij} > 0$. It involves the computation of a double sum, which might be expensive in practice. A more serious problem is that the π_{ij} are sometimes difficult or impossible to obtain. Fortunately, the HT estimator and the respective variances simplify when used with most of the designs discussed previously. For example, when SRS is used, we have $\pi_i = n/N$ for all i and

$$\hat{\tau}_{\text{SRS}} = \frac{N}{n} \sum_{r_i \in S} y_i.$$

A sampling rate of 1% thus leads to a scale-up factor of 100.

Table 2.3 gives estimators, their variance and estimators of their variance for other sampling designs. Some of the formulas involve the *population variance*

$$\sigma^2 = \frac{1}{N-1} \sum_{r_i \in R} (y_i - \mu)^2$$

or the *sample variance*

$$s^2 = \frac{1}{n-1} \sum_{r_i \in S} (y_i - \bar{y})^2 \qquad \text{with} \qquad \bar{y} = \frac{1}{n} \sum_{r_i \in S} y_i.$$

The table gives two estimators for Bernoulli sampling: one that does not require knowledge of N and one that does. The first estimator is the standard HT estimator. The second estimator, in essence, treats the sample as if it were a simple random sample. The estimator is asymptotically unbiased and usually more efficient (Strand 1979). For stratified sampling, we denote the strata by $R_1, \ldots, R_H$ and the stratum

Table 2.3: Estimators of the population total, their variance and estimators of their variance

Design	**Required**	$\hat{\tau}$	Bias$[\hat{\tau}]$	Var$[\hat{\tau}]$	$\hat{\text{Var}}[\hat{\tau}]$
Without replacement	π_i, π_{ij}	$= \sum_{r_i \in S} \frac{y_i}{\pi_i}$	$= 0$	$= \sum_{r_i \in R} \sum_{r_j \in R} \left(\frac{\pi_{ij}}{\pi_i \pi_j} - 1 \right) y_i y_j$	$= \sum_{r_i \in S} \sum_{r_j \in S} \left(\frac{1}{\pi_i \pi_j} - \frac{1}{\pi_{ij}} \right) y_i y_j$
With replacement	p_i	$= \frac{1}{n} \sum_{r_i \in S} \frac{y_i}{p_i}$	$= 0$	$= \frac{1}{n} \sum_{r_i \in R} \left(\frac{y_i}{p_i} - \tau \right)^2 p_i$	$= \frac{1}{n(n-1)} \sum_{r_i \in S} \left(\frac{y_i}{p_i} - \hat{\tau} \right)^2$
SRS	N	$= \frac{N}{n} \sum_{r_i \in S} y_i$	$= 0$	$= N^2 \frac{\sigma^2}{n} \left(1 - \frac{n}{N} \right)$	$= N^2 \frac{s^2}{n} \left(1 - \frac{n}{N} \right)$
SRSWR	N	$= \frac{N}{n} \sum_{r_i \in S} y_i$	$= 0$	$= N(N-1) \frac{\sigma^2}{n}$	$= N^2 \frac{s^2}{n}$
Bernoulli	q	$= \frac{1}{q} \sum_{r_i \in S} y_i$	$= 0$	$= \left(\frac{1}{q} - 1 \right) \sum_{r_i \in R} y_i^2$	$= \frac{1}{q} \left(\frac{1}{q} - 1 \right) \sum_{r_i \in S} y_i^2$
Bernoulli	N	$= \frac{N}{n} \sum_{r_i \in S} y_i$	Asymptotically as SRS, see Strand (1979).		
Poisson	$\pi_i = M \frac{w_i}{\sum_i w_i}$	$= \sum_{r_i \in S} \frac{y_i}{\pi_i}$	$= 0$	$= \sum_{r_i \in R} \left(\frac{1}{\pi_i} - 1 \right) y_i^2$	$= \sum_{r_i \in S} \frac{1}{\pi_i} \left(\frac{1}{\pi_i} - 1 \right) y_i^2$
Stratified	$\hat{\tau}_h$	$= \sum_{h=1}^{H} \hat{\tau}_h$	$= 0$	$= \sum_{h=1}^{H} \text{Var}[\hat{\tau}_h]$	$= \sum_{h=1}^{H} \hat{\text{Var}}[\hat{\tau}_h]$
Stratified + SRS	N_h	$= \sum_{h=1}^{H} \frac{N_h}{n_h} \sum_{r_i \in S_h} y_i$	$= 0$	$= \sum_{h=1}^{H} N_h^2 \frac{\sigma_h^2}{n_h} \left(1 - \frac{n_h}{N_h} \right)$	$= \sum_{h=1}^{H} N_h^2 \frac{s_h^2}{n_h} \left(1 - \frac{n_h}{N_h} \right)$
Systematic / Cluster		See Särndal et al. (1991).			

samples by $S_1, \ldots, S_H$, respectively. The stratum size, sample size and total estimator of the h-th stratum are denoted by N_h, n_h and $\hat{\tau}_h$, respectively.

Hansen and Hurwitz (1943) introduced a general estimator for sampling designs with replacement. The estimator requires knowledge of the probability p_i of selecting item r_i in each of the draws. Observe that p_i differs from π_i for $n > 1$; the latter probability is given by $\pi_i = 1 - (1 - p_i)^n$. The Hansen-Hurwitz (HH) estimator is

$$\hat{\tau}_{\mathrm{HH}} = \frac{1}{n} \sum_{r_i \in S} \frac{y_i}{p_i},$$

where S is treated as a multiset. This means that each value participates in the sum as many times as it is present in S. The HH estimator is unbiased for τ. Table 2.3 gives the variance and an unbiased variance estimator for the HH estimator. The table also contains the SRSWR version of the estimator.

We now turn our attention to population averages. Making use of the HT or HH estimate of the population total, averages are obtained through division by N. We have

$$\hat{\mu}_X = \frac{\hat{\tau}_X}{N} \tag{2.9}$$

for $X \in \{\, \mathrm{HT}, \mathrm{HH} \,\}$. The variance and its estimate change by a factor of N^{-2}:

$$\mathrm{Var}[\, \hat{\mu}_X \,] = \frac{1}{N^2} \mathrm{Var}[\, \hat{\tau}_X \,] \qquad \text{and} \qquad \hat{\mathrm{V}}\mathrm{ar}[\, \hat{\mu}_X \,] = \frac{1}{N^2} \hat{\mathrm{V}}\mathrm{ar}[\, \hat{\tau}_X \,].$$

Of course, this requires that the population size N is known. Otherwise, the HT estimator of N is given by

$$\hat{N}_{\mathrm{HT}} = \sum_{r_i \in S} \frac{1}{\pi_i}$$

and the *weighted sample mean* estimator of μ is given by

$$\hat{\mu}_{\mathrm{WSM}} = \frac{\hat{\tau}_{\mathrm{HT}}}{\hat{N}_{\mathrm{HT}}} = \frac{\sum_{r_i \in S} y_i/\pi_i}{\sum_{r_i \in S} 1/\pi_i}.$$

The estimator is slightly biased but the bias tends to zero as the sample size increases. According to Särndal et al. (1991, sec. 5.7), $\hat{\mu}_{\mathrm{WSM}}$ can be superior to $\hat{\mu}_{\mathrm{HT}}$ in terms of variance. One such example is Bernoulli sampling, where the weighted sample mean coincides with the sample mean $\bar{y}$ while $\mu_{\mathrm{HT}} = n/(qN)\bar{y}$ does not.

C. Confidence Intervals

The HT and HH estimators described previously produce a *point estimate*, that is, they output a single value that is assumed to be close to the true value. In practice, one is often less interested in a single point than in an interval in which the true value is likely to lie. Confidence intervals are a prominent example of this so-called *interval estimation*. A *confidence interval* is a random interval in which the true value lies

with a specified probability. This probability is referred to as the *confidence level* and denoted by $1 - \delta$, where the "error probability" δ is usually small. The larger the value of $1 - \delta$, the broader the intervals get. A typical choice is $1 - \delta = 95\%$, which is a good tradeoff between failure of coverage of the true value and the width of the interval.

In practice, confidence intervals are often obtained from only the sample. The resulting intervals are approximate in the sense that they cover the true value with a probability of approximately $1 - \delta$. Here, we only consider *large-sample confidence intervals* for a population total τ. Large-sample confidence intervals are valid when the estimate $\hat{\tau}$ is normally distributed with mean τ. This condition of normality holds when either the population values are normally distributed or—by the central limit theorem—when the sample size is sufficiently large, say, 100 items. The desired interval is then given by

$$\hat{\tau} \pm t_{1-\delta} \hat{\mathrm{SE}}[\,\hat{\tau}\,],$$

where $t_{1-\delta}$ is $(1 - \delta/2)$-quantile of the normal distribution. For example, $t_{0.90} \approx 1.64$, $t_{0.95} \approx 1.96$ and $t_{0.99} \approx 2.58$. The width of the interval depends on both the confidence level and the standard error of the estimator. It is approximate because an estimate of the standard error is used instead of the real standard error.

D. Functions of Sums and Averages

Denote by $\tau_1, \ldots, \tau_k$ the population totals of the values of functions $f_1, \ldots, f_k$, respectively, and denote by $\hat{\tau}_1, \ldots, \hat{\tau}_k$ the corresponding HT estimators. As usual in practice, we assume that all estimators have been computed from the same sample. Statistics of the form

$$\theta = g(\tau_1, \ldots, \tau_k)$$

can be estimated by replacing each τ_j by its estimator $\hat{\tau}_j$ so that

$$\hat{\theta} = g(\hat{\tau}_1, \ldots, \hat{\tau}_k).$$

If g is a linear function of the form

$$g(x_1, \ldots, x_k) = a_0 + \sum_{j=1}^{k} a_j x_j$$

for arbitrary but fixed $a_0, \ldots, a_k \in \mathbb{R}$, the estimator $\hat{\theta}$ is unbiased. The variance of $\hat{\theta}$ depends on the covariances between the θ_j. The *covariance* between two estimators is a measure of correlation defined as

$$\begin{aligned}\mathrm{Cov}[\,\hat{\tau}_{j_1}, \hat{\tau}_{j_2}\,] &= \mathrm{E}\left[\,\big(\hat{\tau}_{j_1} - \mathrm{E}\,[\,\hat{\tau}_{j_1}\,]\big)\big(\hat{\tau}_{j_2} - \mathrm{E}\,[\,\hat{\tau}_{j_2}\,]\big)\,\right] \\ &= \sum_{r_{i_1} \in R} \sum_{r_{i_2} \in R} \left(\frac{\pi_{i_1 i_2}}{\pi_{i_1}\pi_{i_2}} - 1\right) y_{i_1 j_1} y_{i_2 j_2},\end{aligned}$$

where $y_{ij} = f_j(r_i)$. In the final equality, we used the assumption that the $\hat{\tau}_j$ are derived from the same sample, as usual in practice.[3] A positive covariance indicates that $\hat{\tau}_{j_2}$ tends to increase as $\hat{\tau}_{j_1}$ increases. Conversely, when the covariance is negative, $\hat{\tau}_{j_2}$ tends to decrease as $\hat{\tau}_{j_1}$ increases. An unbiased estimator of the covariance is given by

$$\hat{\mathrm{Cov}}[\,\hat{\theta}_{j_1}, \hat{\theta}_{j_2}\,] = \sum_{r_{i_1} \in S} \sum_{r_{i_2} \in S} \frac{1}{\pi_{i_1}\pi_{i_2}} \left(\frac{\pi_{i_1}\pi_{i_2}}{\pi_{i_1}\pi_{i_2}} - 1 \right) y_{i_1 j_1} y_{i_2 j_2}.$$

We can now express the desired variance of $\hat{\theta}$ as

$$\begin{aligned} \mathrm{Var}[\,\hat{\theta}\,] &= \sum_{j_1=1}^{k} \sum_{j_2=1}^{k} a_{j_1} a_{j_2} \,\mathrm{Cov}[\,\hat{\tau}_{j_1}, \hat{\tau}_{j_2}\,] \\ &= \sum_{j=1}^{k} a_j^2 \,\mathrm{Var}[\,\hat{\tau}_j\,] + 2 \sum_{j_1=1}^{k-1} \sum_{j_2=j_1+1}^{k} a_{j_1} a_{j_2} \,\mathrm{Cov}[\,\hat{\tau}_{j_1}, \hat{\tau}_{j_2}\,], \end{aligned} \tag{2.10}$$

where the covariance terms in the final equality might reduce or add to the total variance. The variance of $\hat{\theta}$ can be estimated by replacing in (2.10) the variance $\mathrm{Var}[\,\hat{\tau}_j\,]$ by $\hat{\mathrm{Var}}[\,\hat{\tau}_j\,]$ and $\mathrm{Cov}[\,\hat{\tau}_{j_1}, \hat{\tau}_{j_2}\,]$ by $\hat{\mathrm{Cov}}[\,\hat{\tau}_{j_1}, \hat{\tau}_{j_2}\,]$. If it is known that the sum of the covariance terms is negative or close to zero, one occasionally ignores the covariances and yields a conservative (i.e., too large) estimate of the variance.

If g is non-linear but continuous in the neighborhood of θ, the estimator $\hat{\theta}$ is approximately unbiased for sufficiently large sample sizes. Its variance can be estimated via Taylor linearization, see Särndal et al. (1991, sec. 5.5) or Haas (2009, sec. 4.5).

E. Quantiles

Random samples can also be used to make inferences about population quantiles. Denote by $r_{(1)}, r_{(2)}, \ldots, r_{(N)}$ a sequence of the items in R ordered by some criterion, so that $r_{(t)}$ denotes the (t/N)-quantile of the population. The goal is to find from the sample a confidence interval for $r_{(t)}$. Denote by $s_{(1)}, s_{(2)}, \ldots, s_{(n)}$ the ordered sequence of items in a uniform sample from R. The interval

$$[s_{(L)}, s_{(U)}]$$

with $1 \le L \le U$ and $L \le t$ contains the desired quantile with probability

$$\begin{aligned} \Pr\left[\, s_{(L)} \le r_{(t)} \le s_{(U)} \,\right] &= \Pr[\, s_{(L)} \le r_{(t)}\,] - \Pr[\, s_{(U)} \le r_{(t-1)}\,] \\ &= \left\{ \sum_{i=L}^{U-1} \binom{t}{i} \binom{N-t}{n-i} + \binom{t-1}{U-1} \binom{N-t}{N-U} \right\} \Big/ \binom{N}{n}. \end{aligned}$$

For example, with $N = 100{,}000$ and $n = 1{,}000$, the 25% quantile lies between $s_{(230)}$ and $s_{(270)}$ with a probability of approximately 85%. The situation gets much more

[3] Otherwise, if the estimates were independent, the covariance would be 0.

Table 2.4: Coarse comparison of survey sampling and database sampling

	Survey sampling	**Database sampling**
Query known	Yes	No
Exact result obtainable	No	Yes
Non-response	Yes	No
Measurement errors	Yes	No
Domain expertise available	Yes	No
Sampling designs	Sophisticated	Simple (e.g., uniform)
Sample size	Small	Large
When performed	Query time	Usually in advance
Time required	Days, weeks, months	Seconds, minutes, hours
Preprocessing feasible	No	Yes

complicated when non-uniform sample designs are used; see Krishnaiah and Rao (1988, ch. 6).

2.2 Database Sampling

Database sampling is concerned with sampling techniques tailored to database management systems and data stream management systems. In section 2.2.1, we point out the key differences from survey sampling. We then discuss the three alternative approaches to database sampling: query sampling (section 2.2.2), materialized sampling (section 2.2.3) and permuted-data sampling (section 2.2.4). Within data stream management systems, different sampling techniques are required; we discuss these techniques separately in section 2.2.5.

2.2.1 Comparison to Survey Sampling

Although database sampling techniques are built upon the survey sampling techniques of section 2.1, they differ in various respects. Our discussion of these differences draws from a similar discussion in Haas (2009). A coarse overview is given in table 2.4; not all points do always apply. We first discuss survey sampling and then proceed to database sampling.

The goal of survey sampling is to obtain information about the population that often cannot be obtained with other techniques. For example, it is infeasible to question all of the inhabitants of a large city or even the entire country. The objective of the survey is known in advance and the sample is created exclusively to achieve the objective. In fact, sampling surveys are often carried out by statisticians and domain experts, who create highly specialized sampling schemes just for the purpose of a single survey. These specialized schemes allow for very small sample sizes, which are a must because access to the data is limited and costly. Because the effort for

conducting a survey is high, the process of sampling may require from a few days up to many months to complete.

We face an entirely different situation in database sampling. In contrast to survey sampling, we are in principle able to run the query on the entire database (i.e., the population) and obtain its exact result. However, in the applications we are interested in, this approach is too time-consuming and sometimes even infeasible. Preprocessing of the data is nevertheless feasible, and many database sampling techniques make use of it in order to support efficient sampling at query time. The prospects of preprocessing are somewhat limited, though, because the query is often unknown or only vaguely known at the time the sample is created. This is due to two reasons: First, as we discuss later, even the computation of the sample at query time might be too expensive, so that the sample has to be precomputed before the query is issued. Second, the cost of computing the sample is balanced out when the sample is reused several times; it is clearly desirable to make use of a single sample for many different queries. Since domain expertise is unavailable and we consequently cannot build highly specific samples, simple sampling designs—such as uniform sampling designs—that play only a minor role in survey sampling become essential tools in database sampling. Also, to compensate for the disadvantages of simple sampling designs, database samples tend to be much larger than survey samples.

Research in database sampling mainly focuses on the questions of (i) how to quickly provide a sample at query time, and (ii) how to run database queries on the samples. Sampling schemes that are able to (iii) exploit workload information, or any other information about expected queries, are also of interest. In the remainder of this section, and in large parts of this thesis, we focus on (i); available techniques for points (ii) and (iii) are discussed in section 2.3.

2.2.2 Query Sampling

In what follows, we distinguish *exact queries* and *approximate queries*. The decision of whether or not approximate query answers are allowed is therefore left to the issuer of the query. Perhaps the oldest class of database sampling methods is represented by query sampling, which is also known as online sampling. In *query sampling*, the sample is computed from the database at query time. When an approximate query enters the system, the base data is accessed in order to build a sample for estimating the query result. Repeated runs of the same query initiate repeated computations of the sample and thus may lead to different results. Figure 2.4 illustrates the architecture of an query-sampling enhanced database system.

If implemented carefully, query sampling leads to a reduction in I/O cost because the sample can be built without reading the entire data set. As we will see below, the reduction in I/O cost might not be as high as expected at first thought. In any case, query sampling leads to reduction in CPU cost because fewer tuples have to be processed. In practice, different sampling schemes are applied depending on whether query processing is I/O-bound or CPU-bound (Haas and König 2004).

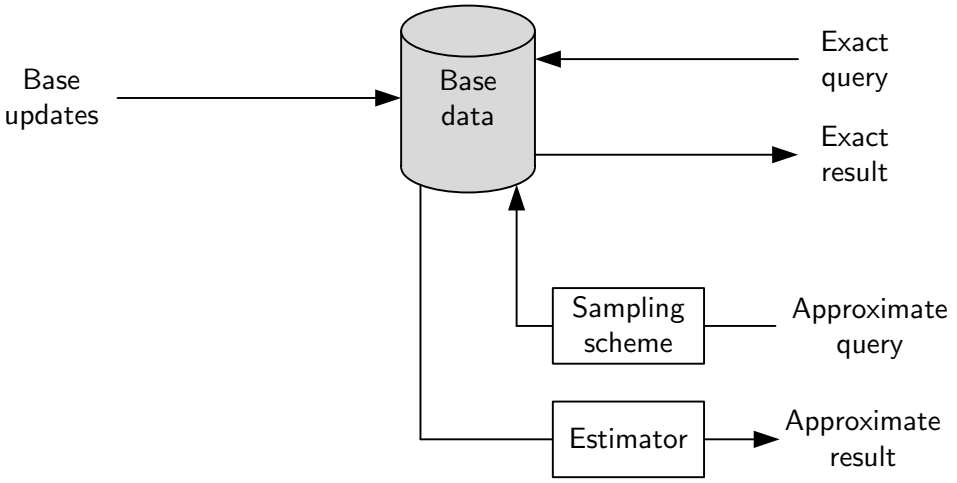


Figure 2.4: Query sampling architecture

A. Sampling Schemes

We now describe how to obtain a query sample from a relational table. Virtually all query sampling schemes lead to uniform or clustered uniform designs because the cost to execute alternative schemes, such as weighted or stratified sampling, at query time are simply too high. One distinguishes *row-level* and *block-level* schemes, depending on whether the sampling units are individual items (=rows) or blocks of items as stored on hard disk. The latter approach produces a cluster sampling design. For a given sample size, row-level sampling often leads to more precise estimates compared to block-level, sampling while block-level sampling has lower I/O cost (see below).

Row-level sampling is usually executed draw-by-draw, that is, one item at a time. Each successful draw yields an item chosen uniformly and at random from the population, which naturally leads to a with-replacement sampling scheme.[4] In the simplest setting, the items are stored in blocks of equal size, that is, each block contains *exactly* the same number of items. Then, a random item can be obtained by first selecting a random block (see below) and then picking a random item from this block. Results about the probability distribution and expected values of the number of accessed blocks to obtain a sample of a specified size can be found in Cárdenas (1975), Yao (1977), and Cheung (1982).[5]

In a more general setting, items are distributed among blocks of unequal size. If the block sizes are known, the above schemes can be modified appropriately: Each block is selected with a probability proportional to its size and, as before, a random

[4] To sample without replacement, draws that produce items already sampled are repeated.

[5] These articles do not directly discuss row-level sampling but consider the problem of retrieving k *specific* records from disk. Since their analysis is based on the assumption that every record on disk is equally likely to be one of the k specific records, it coincides with row-level sampling.

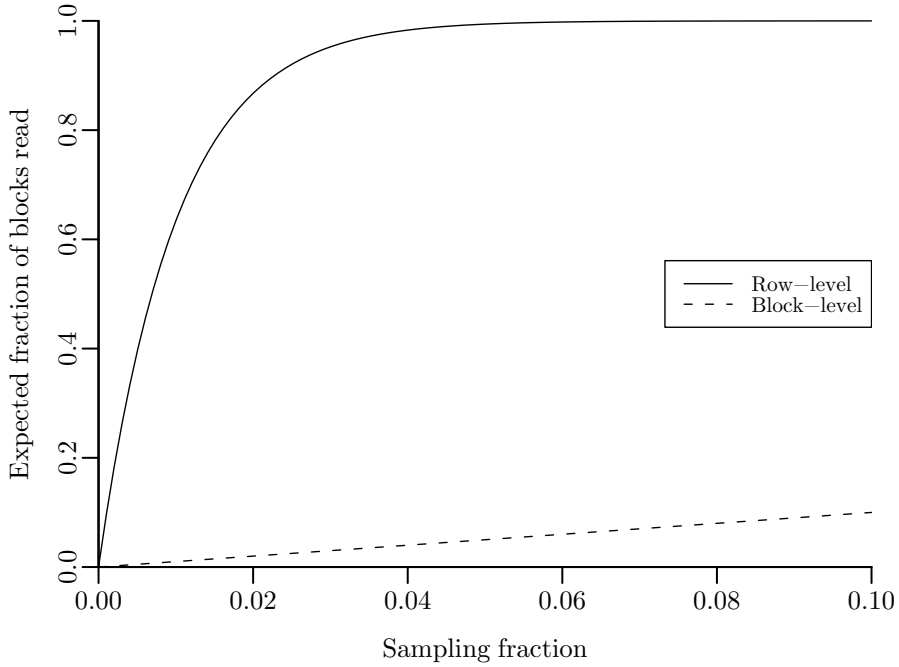
item from the block becomes the sample item. Christodoulakis (1983) gives results about the number of required block accesses. In his Ph.D. thesis, Olken (1993) discusses acceptance/rejection (A/R) schemes that compute the sample when block sizes are unknown beforehand. The key idea is to pick a block uniformly and at random. The block is "accepted" with probability

$$\frac{\text{number of items in selected block}}{\text{upper bound on number of items in any block}}$$

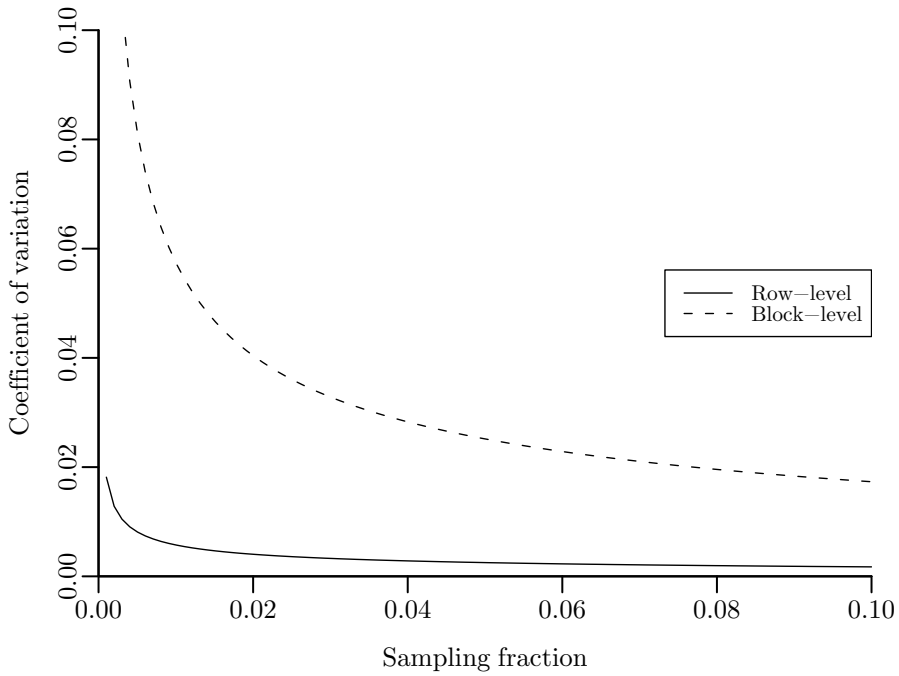
and rejected otherwise. In the case of a rejection, the current draw fails and is repeated from scratch (by choosing a new block). In case of acceptance, a tuple chosen uniformly and at random from the selected block is returned as the sample item. The A/R step ensures that each item is chosen with equal probability but it also increases the sampling cost due to rejections.

In an I/O-bound environment, the cost of reading blocks from disk dominate the cost of row-level sampling. The idea behind block-level sampling is to sample all items of selected blocks, that is, all items read from disk are used in the sample. Perhaps the simplest approach is block-level Bernoulli sampling, where each block is included in the sample with probability q and excluded with probability $1 - q$. Of course, only the blocks that are included into the sample are read from disk. Alternatively, one might sample draw-by-draw for block-level SRSWR or apply list-sequential sampling techniques (section 3.4.2) to obtain a block-level SRS. All block-level sampling schemes require that we are able to efficiently retrieve the k-th out of all blocks, where k is a random block number determined by the scheme. This is trivial if the blocks are stored contiguously. Otherwise, in-memory data structures that record where to find the blocks on disk can be exploited; see B^+ tree sampling (Olken and Rotem 1989), augmented-B^+ tree sampling (Antoshenkov 1992), hash file sampling (Olken et al. 1990), and extent map sampling (DeWitt et al. 1992).

We now give a simple example that emphasizes the advantages and disadvantages of the two approaches. Suppose that the population consists of 1,000,000 items, which are stored in blocks of 100 items each. Further, suppose that the first block consists of only 1s, the second block of only 2s, and, more generally, the i-th block consists of only is. To compare row-level and block-level sampling, we can run both schemes on the population and use the sample to estimate the population average. Figure 2.5a shows the expected number of fetched blocks for each of the schemes and for varying sampling fractions. As can be seen, row-level sampling is much more expensive than block-level sampling. In fact, when the sampling rate exceeds roughly 4%, we are reading almost all of the blocks. However, as shown in figure 2.5b, the precision of the estimate (in terms of the coefficient of variation) is much higher for row-level sampling than it is for block-level sampling. It is not immediately clear which of the two schemes is the better choice; the decision depends on both the estimator and the distribution of the data in the blocks. An in-depth treatment of this topic can be found in Haas and König (2004), who also propose a hybrid sampling scheme that combines the advantages of both approaches.



(a) I/O cost



(b) Precision of estimate

Figure 2.5: Comparison of row-level and block-level sampling

B. The SQL Standard and Commercial Implementations

A very basic form of query sampling has been included into the SQL/Foundation:2003 standard. According to the standard, each table in the `FROM` clause can be followed by a sample clause of the form

```
TABLESAMPLE BERNOULLI|SYSTEM (<rate>) [REPEATABLE (<seed>)],
```

where `<rate>` denotes the desired sampling rate in percent, `BERNOULLI` stands for row-level Bernoulli sampling, and `SYSTEM` is vendor-specific and typically refers to a block-level sampling scheme. The `REPEATABLE` clause can be used to generate the same sample in repeated executions of the query (by passing the same `<seed>`). Estimation from the sample has to be implemented by the query issuer. For example, the following query estimates the number of parts in stock based on a 1% Bernoulli sample using the HT estimator described in section 2.1.3:

```
SELECT SUM(NO_STOCKED)*100 FROM PARTS TABLESAMPLE BERNOULLI (1)
```

The `TABLESAMPLE` clause is implemented in IBM DB2 UDB (Haas 2003) and with a varied syntax in Microsoft SQL Server[6], Oracle Database[7], Teradata Database[8], and IBM Informix Dynamic Server[9]. Gryz et al. (2004) also describe how DB2 uses query sampling techniques when the `RAND()` function is encountered in the `WHERE` clause of a SQL query.

2.2.3 Materialized Sampling

In *materialized sampling*, or offline sampling, a set of precomputed samples is materialized before the actual queries arrive. The samples are taken directly from the base tables or from arbitrary views derived from the base tables. For this reason, materialized samples are sometimes called materialized sample views (Larson et al. 2007). When an approximate query enters the system, one or more of these samples are selected to answer the query. Unless different samples are selected, repeated runs of the same query produce identical results. The general architecture of a materialized-sampling system is shown in figure 2.6.

Materialized sampling has several advantages when compared to query sampling. Since the sample is computed in advance, the cost of sample computation does not affect the cost of processing an approximate query. One may therefore use more sophisticated sampling schemes than in query sampling. As a consequence, the precision of estimates derived from materialized samples can be significantly increased. Processing materialized samples also has a lower I/O cost because the samples are readily available when needed. For example, a materialized uniform

[6] http://msdn2.microsoft.com/en-us/library/ms189108.aspx
[7] http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/ap_standard_sql004.htm
[8] http://www.teradata.com/t/page/44490/
[9] http://www.redbooks.ibm.com/abstracts/tips0513.htm

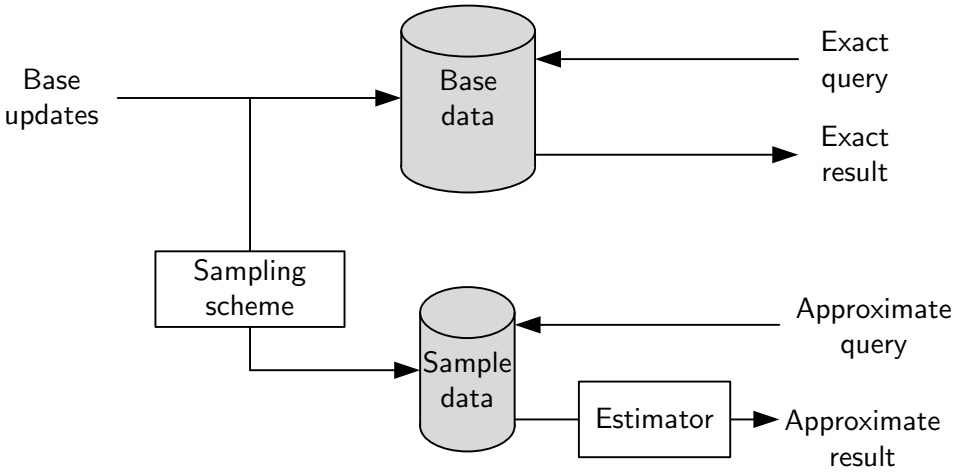


Figure 2.6: Materialized sampling architecture

sample of a relation stored on disk can be exploited to obtain estimates with the (high) precision of row-level sampling at the (low) cost of block-level sampling. The cost is further reduced when the sample is small enough to be stored in main memory. Apart from costs, materialized sampling is advantageous because it supports a wider range of queries. The reason is that materialized samples can be computed from arbitrary views over base tables. For example, it may be feasible to precompute a sample of a join with arbitrary join predicates even when the cost of obtaining such a sample at query time is too high.

The advantages of materialized sampling do not come without disadvantages. A minor point is that materialized samples require storage space. But unless the sampling fractions are extremely high, the additional space consumption will be low compared to the space required to store the base data. In fact, a large part of this thesis is concerned with sampling schemes that guarantee that the space consumption is bounded at all times. A more important point is that materialized samples have to be maintained or refreshed as the base data evolves. In a static database, samples computed once are valid forever.[10] In practice, however, databases are evolving: they are subject to insertions, updates and deletions. Each of these operations has to be reflected appropriately in the sample, which incurs maintenance costs. We do not go into further detail here because sample maintenance is one of the main topics in this thesis; it is thoroughly discussed in chapters 3 and following.

[10] In order to avoid that a bad sample obtained by an "unlucky" draw remains valid for all times, static samples should be recomputed occasionally.

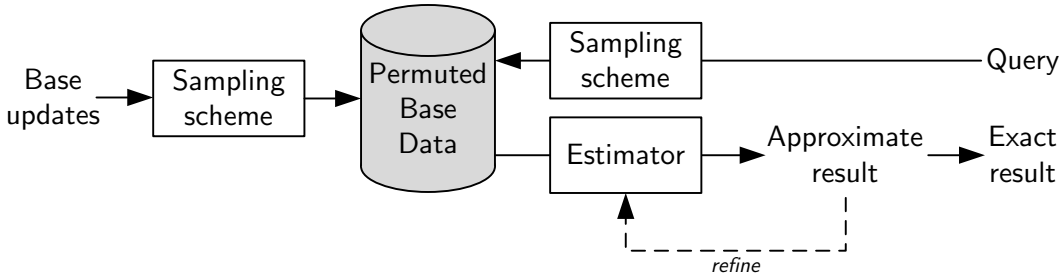


Figure 2.7: Permuted-data sampling architecture

2.2.4 Permuted-Data Sampling

In *permuted-data sampling*, the data in the database is rearranged in random order. The DBO system of Jermaine et al. (2007), which pioneered permuted-data sampling, selects each possible ordering with equal probability; but other approaches are possible. In both cases, any prefix of a table can be seen as a random sample of that table. This property is exploited in query processing, where an estimate of the final query result is maintained while the data is processed. In fact, there is no distinction between exact queries and approximate queries: exact query results are obtained by running queries to completion. The architecture of a permuted-data sampling system is depicted in figure 2.7. Part of the sampling cost emerges at update time (permutation of tables), part of it at query time (reading the permuted tables in a specified order). In this sense, permuted-data sampling is in between query sampling and materialized sampling.

Permuted-data sampling is appealing because a first estimate is quickly available and this estimate is refined as the query execution progresses. As soon as the precision is sufficient, the query (or parts of the query) can be stopped. Permuted-data sampling has its roots in the work on online aggregation by Hellerstein et al. (1997), where the focus was put on aggregation queries with optional group-by. Later, techniques for the computation of running confidence intervals have been proposed (Haas 1997, 1999; Jermaine et al. 2007). The main challenge in permuted-data sampling is the non-blocking computation of joins; the problem has been extensively studied by Haas and Hellerstein (1999), Jermaine et al. (2005), and Jermaine et al. (2007).

The main drawback of permuted-data sampling is that it has an impact on traditional query processing. This includes both queries and updates. Data in a traditional database is organized in such a way that some locality conditions are preserved. In a permuted database, locality is destroyed on purpose. Preliminary results in Jermaine et al. (2007) indicate that the overhead for running queries to completion in a permuted database is not too large. Since permuted-data sampling is a relatively new field of study, it is currently unknown how updates to the base data can be incorporated without too much overhead.

A rough overview of the advantages and disadvantages of query sampling, materialized sampling and permuted-data sampling is given in table 2.5. Parentheses

Table 2.5: Comparison of query, materialized and permuted-data sampling

	Query	**Materialized**	**Permuted**
Responsiveness	moderate	high	high
Repeated queries have different results	yes	no	no
Ability to handle complex queries	low	high	moderate
Ability to deal with data skew	low	high	(low)
Ability to report running aggregates	high	low	high
Performance impact on exact queries	no	no	yes
Performance impact on updates	no	yes	(yes)
Storage space required	no	yes	no
Access to base data required	yes	no	yes

indicate conjectures; the respective points have not yet been explored in the literature. Each approach has unique advantages and disadvantages; the decision on which one to use must be taken on a case-by-case basis. In this thesis, we focus almost entirely on materialized sampling. The reason is that query sampling techniques are too costly for many applications, and permuted-data sampling, though promising, requires a complete redesign of the available system, thereby affecting traditional query processing. Materialized samples both support the widest range of queries and have the lowest response time. These two facts alone can outweigh its main disadvantage, namely that of maintenance cost.

2.2.5 Data Stream Sampling

Data stream sampling is concerned with sampling from data streams, that is, from potentially infinite sequences of items. Samples are used to reduce or bound resource consumption, to support ad-hoc querying, to analyze why the system produced a specific output, or to optimize the query graph. Since a data stream is changing continuously, efficient sample maintenance is key to data stream sampling. Although many techniques for maintaining materialized samples from databases can also be used to sample from data streams, there are problems that are unique to data stream sampling. To avoid repetition, we focus on these problems only, that is, we do not discuss sampling problems that are solvable with database sampling techniques. The main difference in data stream sampling is that samples are often "biased" towards more recent items because recent items are considered more important by applications. Such a notion of recency does not exist in traditional database sampling and calls for novel sampling schemes. In this section, we give an overview of data stream sampling; actual sampling schemes are discussed in chapter 3.

We model a data stream as a (possibly infinite) sequence of items. The items in the stream may or may not comprise a timestamp attribute. If they do, we assume—as usual in data stream literature—that the timestamps are non-decreasing. There

are alternative data stream models, in which each arriving element of the stream constitutes an update to a global data structure (Muthukrishnan 2005). These models resemble database sampling and—for this reason—are excluded from our discussion. In the following, we will also assume that the intended use of the sample is not known or only vaguely known upon its creation. Otherwise, specialized techniques, which are not necessarily based on random sampling, exist for a wide range of analytical tasks. The discussion of these techniques by far exceeds the scope of this thesis, but see Muthukrishnan (2005), Aggarwal (2006a), and Garofalakis et al. (2009).

If sampling is performed from the entire stream, weighted sampling schemes, in which the weight increases with the recency of the item, are required. The reason is that—since the data stream is infinite—schemes such as uniform sampling produce samples that either grow infeasibly large or represent a diminishingly small fraction of the recent stream. In contrast, weighted samples have the advantage that a recent fraction of the data stream is well-represented in the sample (Haas 2009; Aggarwal 2006b).

An alternative approach is to sample from a window defined over the stream. We distinguish *stationary windows*, in which the endpoints of the window are fixed, and *sliding windows*, in which the endpoints move forward (Golab and Özsu 2003). Database sampling techniques can be applied to sample from stationary windows (Haas 2009). Sliding windows can be further classified into *sequence-based* sliding windows, which comprise the last N items arrived in the stream, and *time-based* windows, which comprise the items that arrived during the last Δ time units. We say that a sample is sequence-based or time-based if it is derived from a sequence-based or time-based window, respectively. The number of items in the window is referred to as the *window size*, while the time interval covered by a window is called the *window length*.

Using this terminology, sequence-based windows have fixed size but varying length. The fixed window size reduces the complexity of sample maintenance, but the varying window length can be problematic for time-based analyses. For example, consider the following CQL query:[11]

```
SELECT SUM(size) AS num_bytes
FROM packets [Range 60 Minutes]
GROUP BY port.
```

This query computes the number of bytes per port observed in the `packets` stream during the last 60 minutes. Suppose that we want to answer the query using a sequence-based sample of the last N items. Since the sample is sequence-based and the query is time-based, we have to choose N in such a way that the last N items are guaranteed to completely cover the 60-minute range of the query. Clearly, such a choice is impossible if no a-priori knowledge about the data stream is available. But even if we can come up with an upper bound for the number of items in the query range, sequence-based schemes may perform poorly in practice. The reason is that—unless the data stream rate is roughly constant—the average number of

[11] CQL is short for Continuous Query Language, see Arasu et al. (2006).

items in the query range is much smaller than the upper bound N, so that (with high probability) the sample contains a large fraction of outdated items not relevant for the query.

In contrast, time-based windows have variable window size but fixed window length. A time-based sliding window of the packets arrived during the last 60 minutes can directly be used to approximately answer the query above. In time-based sampling, the main challenge is to realize an upper bound on the space consumption of the sample, while using the allotted space efficiently at the same time. Space bounds are crucial in data stream management systems with many samples maintained in parallel, since they greatly simplify the task of memory management and avoid unexpected memory shortcomings at runtime. The difficulty of time-based sampling arises from the fact that the number of items in the window may vary significantly over time. If the sampling fraction is kept constant at all times, the sample shrinks and grows with the number of items in the window. The size of the sample is therefore unstable and unbounded. To avoid such behavior, the sampling fraction has to be adapted on-the-fly. Intuitively, we can afford a large sampling fraction at low stream rates but only a low sampling fraction at high rates.

As with the different approaches to sampling in databases, the decision of whether to sample a data stream using weighted sampling, sequence-based sampling or time-based sampling has to be taken on a case-by-case basis. The above discussion and the description of the actual sampling schemes in chapter 3 might help in taking that decision.

2.3 Applications of Database Sampling

In a wide area of database-related applications, sampling techniques have been used as means to reduce computational cost. This cost reduction can be achieved in two different ways. First, sampling facilitates query optimization—or, more general, algorithm selection—because knowledge obtained from a sample of the data helps to find more efficient ways to process it. Second, sampling is a key technique in approximate query processing and data mining, where the accuracy of the query answer is traded for the cost of its computation. In this section, we survey recent work in database sampling, structured by their main applications.[12] We cover query sampling, materialized sampling, and permuted-data sampling. Many of these techniques can also be applied on a sample obtained from a (sliding window of a) data stream.

We will see that, in general, there is an advantage if the sample is precomputed, or materialized, before it is actually used. Materialized samples are sometimes referred to as *sample synopses* of the data. There are alternative non-sample synopses such as histograms or wavelets; we do not cover these techniques here. In general, the

[12] In his Ph.D. thesis, Olken (1993) gives a survey about database sampling in the early 90's; most of the work discussed therein also appears here. In general, Olken's survey gives a more detailed account of this earlier work.

advantage of random sampling is that it scales well with both the amount of data and the data's dimensionality. A related approach to data summarization is given by *sketches*, which can be seen as summaries tailored to a specific purpose. For example, a sketch may maintain a data structure that can be used to estimate the number of distinct items in the dataset; the very same sketch cannot be used to estimate anything else, such as the number of distinct items in a subset of the dataset as defined by a predicate. Since they are focused on a specific estimate, sketches—when applicable—often perform better than random sampling. Again, sketches are not discussed in the next sections; we are interested in techniques that can be exploited for a variety of purposes.

2.3.1 Selectivity Estimation

Selectivity estimation is the process of estimating the selectivity or, equivalently, the size of (the result of) a query over a database. It is one of the most important applications of database sampling. Selectivity estimates are used in query optimization, where they help to determine the cost of alternative execution plans. They are also useful for load balancing in distributed systems, where a query workload is distributed amongst several nodes. In both cases, precise selectivity estimates are key to the improvement of the overall performance of query processing. In some applications, selectivity estimates are also interesting in their own right; see section 2.3.3 on approximate query processing.

Our focus is on selectivity estimation in relational database systems. We discuss predicate selectivity estimation, join selectivity estimation, and selectivity estimation for arbitrary relational algebra expressions. We then review fixed-precision sampling schemes that provide estimates with a specified precision. Finally, we discuss the usefulness of random sampling for the purpose of histogram construction and maintenance.

A. Predicate Selectivity

In the simplest instance of the selectivity estimation problem, the query has the form $\sigma(R)$, where R is a table and σ is a predicate. We want to estimate how many tuples in R satisfy predicate σ. We assume that, for each tuple $r \in R$, the membership of r in $\sigma(R)$ can be determined efficiently. Other than that, we do not impose any restrictions on σ. Let $h : \mathscr{R} \to \{\,0, 1\,\}$ denote a predicate function, that is, $h(r) = 1$ if $r \in \sigma(R)$ and $h(r) = 0$ otherwise. Then,

$$N_\sigma = |\sigma(R)| = \sum_{r \in R} h(r) \tag{2.11}$$

denotes the query size. The query selectivity is given by N_σ / N, where as before $N = |R|$.

Estimators for sums such as (2.11) have already been discussed in section 2.1.3B. Denote by S_{row} a uniform (row-level) sample of R and by S_{bl} a uniform block-level sample of R. Unbiased estimators for N_σ are given by

$$\hat{N}_{\text{row}} = \frac{N}{n}|\sigma(S_{\text{row}})| \quad \text{and} \quad \hat{N}_{\text{bl}} = \frac{N_{\text{bl}}}{n_{\text{bl}}}|\sigma(S_{\text{bl}})|,$$

where n denotes the size of S_{row} in tuples, n_{bl} the size of S_{bl} in blocks, and N_{bl} the size of R in in blocks. Block-level sampling for predicate selectivity estimation is implemented in Oracle9i Release 2 and above.[13] As discussed previously, row-level sampling is (typically) superior to block-level sampling in terms of precision per sampled tuple, but block-level sampling is superior to row-level sampling in terms of cost per sampled tuple (cf. figure 2.5). Haas and König (2004) proposed a hybrid approach that combines row-level and block-level sampling. If a materialized uniform sample S_{mat} from R is available, its usage is the best choice: the precision of the estimate corresponds to that of row-level sampling; the sampling cost corresponds to that of block-level sampling (sample stored on disk) or is even less (sample stored in memory).

For uniform sampling (row-level or materialized), Toivonen (1996) has shown that a sample of size $\ln(2/\delta)/(2\epsilon^2)$, for $0 < \epsilon, \delta < 1$, suffices to guarantee that the absolute error of the selectivity estimate exceeds ϵ with a probability of at most δ. This sample size bound is independent of both the dataset and the actual predicate. For example, to guarantee an absolute error of $\epsilon = 0.01$ with probability $1 - \delta = 95\%$, a sample size of 19,000 items is sufficient. Results for selectivity estimation in general are summarized in section D.

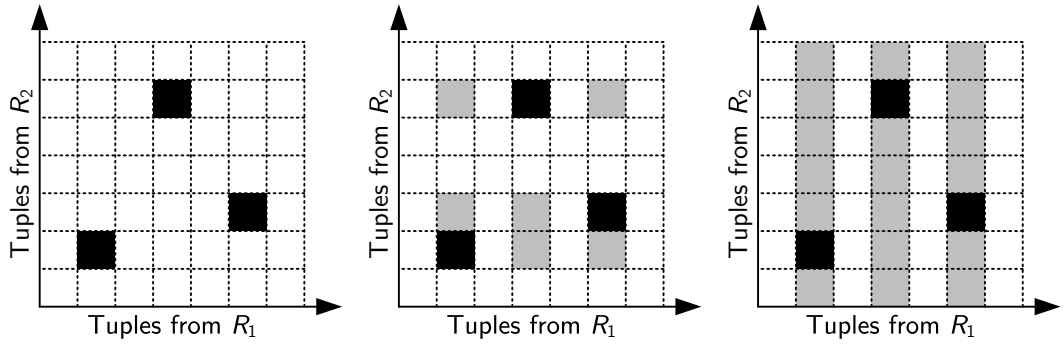
B. Join Selectivity

A join query has the form $\sigma(R_1 \times \cdots \times R_k)$, where $R_1, \ldots, R_k$ are tables and σ is a predicate. For simplicity, we assume that $k = 2$ and that σ describes an equi-join on attribute A. Thus, we consider queries of the form $J = \sigma_1(R_1) \bowtie_{R_1.A=R_2.A} \sigma_2(R_2)$. A pair $(r_1, r_2) \in R_1 \times R_2$ of tuples is said to join if and only if $(r_1, r_2) \in J$. The quantity $|J|$ denotes the join size; $|J|/(|R_1| \cdot |R_2|)$ denotes the join selectivity. For ease of presentation, we subsequently assume that $|R_1| = |R_2| = N$.

To describe the available schemes, we make use of the notation of Seshadri (1992) and Haas et al. (1993).[14] The simplest scheme is denoted *t_indep*; this scheme basically runs the row-level sampling scheme of section A with $R = R_1 \times R_2$. To obtain a sample from R, t_indep samples uniformly a tuple r_1 from R_1 and r_2 from R_2 (with replacement). The fraction of (r_1, r_2)-pairs that join is an unbiased estimate of the join selectivity.

[13] The technique is called dynamic sampling, see http://download.oracle.com/docs/cd/B10500_01/server.920/a96533/whatsnew.htm.

[14] Names consist of prefix "t_" for row-level and "p_" for block-level sampling. The suffix describes the type of sampling: independent, cross-product, or index-based.



(a) t_indep **(b)** t_cross **(c)** t_index

Figure 2.8: Space explored by various schemes for join selectivity estimation (n=3)

Suppose that t_indep samples n tuples from both R_1 and R_2. As illustrated in figure 2.8a, it then explores a fraction of n/N^2 of the full cross product. This is because every sample item is only used once: the i-th sample from R_1 is joined with the i-th sample from R_2 only, $1 \leq i \leq n$. An alternative approach is to join each newly sampled tuple with all previously sampled tuples. This method is denoted as *t_cross*; it explores a fraction of n^2/N^2 of the cross product (figure 2.8b). Alon et al. (1999) have shown that t_cross is space-optimal for worst-case data: a sample of size $O\big(N^2/J\big)$ is both necessary and sufficient to guarantee fixed relative error with constant probability. Unfortunately, this also implies that, for low-selectivity queries (small J), very large samples might be required; see section D for a way to avoid this problem.

If there is an index on, say, $R_2.A$, we can leverage the index to come up with a more efficient sampling scheme. In each step, the index-based scheme *t_index* samples a single tuple from R_1 and then computes the full join $r_1 \bowtie S$, thereby making use of the index. When n tuples have been sampled from R_1, the explored search space is n/N (figure 2.8c); a significant improvement.

Each of the above schemes can be implemented using block-level sampling; the resulting schemes are denoted as *p_indep*, *p_cross*, and *p_index*. All but the index-based schemes are due to Hou et al. (1988). The index-based schemes have been proposed by Lipton and Naughton (1990), and Hou and Ozsoyoglu (1991). A theoretical comparison between all 6 schemes has been performed by Haas et al. (1993) and Haas et al. (1996). They found that after a fixed number of sampling steps, the following relationships hold:

$$\text{index} \leq \text{block-level} \leq \text{row-level}$$

and

$$\text{index} \leq \text{cross} \leq \text{indep},$$

where we write $X \leq Y$ when the scheme X achieves a better or equal precision than scheme Y.

In some cases, the above algorithms can be problematic. One example of such a situation is when $R_1.A$ contains a few infrequent values that are frequent in $R_2.A$. This is because these infrequent values are likely to be missed by all schemes, even though they may contribute significantly to the join size. For these hard cases, Haas et al. (1995) propose an extended version of t_cross that makes use of precomputed statistics about frequent values. Ganguly et al. (1996) also focus on these cases; their bifocal sampling scheme is an ingenious combination of t_cross and t_index.

More recent schemes precompute materialized samples to further improve the selectivity estimates. Suppose that we have precomputed a sample S from the join result $R_1 \bowtie R_2$. The sample can be used to estimate the selectivities of all joins of the form $\sigma(R_1 \bowtie R_2)$. The problem of join selectivity estimation is therefore reduced to the problem of predicate selectivity estimation. Acharya et al. (1999) propose a technique called *join synopses*, which precomputes samples over key-foreign-key relationships. In more detail, for each table R, the scheme determines the set of tables $R_1, \ldots, R_k$ that are reachable from R via key-foreign-key relationships. The sample is taken from $R \bowtie R_1 \bowtie \ldots \bowtie R_k$. As shown by Kaushik et al. (2005), the so-obtained sample is a space-optimal synopsis for worst-case data. Gemulla et al. (2008) show how the space consumption of join synopses can be further reduced (although it is still the same in the worst case). For other joins than key-foreign-key joins, Estan and Naughton (2006) propose a weighted sampling scheme that does not sample from the join result. Instead, the scheme correlates the individual samples of each table in the database. This correlation allows for significantly more precise join size estimates than it would be possible with query sampling techniques. Larson et al. (2007) discusses the general advantages of materialized "sample views" in query optimization. They make use of a static sample that is refreshed whenever the quality of the estimates—as observed after running the query—indicates that the sample has become outdated.

C. Arbitrary Expressions of Relational Algebra

The problem of selectivity estimation for general relational algebra expressions has been studied in Hou et al. (1988) and Hou and Ozsoyoglu (1991). In this instance of the problem, we are given an expression E that combines a set of base tables using operations $\sigma, \bowtie, \pi, \cap, \cup$, and $\setminus$. We are interested in the query size $|E|$.

We briefly summarize the results of Hou et al.'s work. First, set intersections ($\cap$) are a special case of a natural join, where all attributes of both tables are join attributes. This implies that any query containing only σ, $\bowtie$, and $\cap$ can be evaluated using the techniques previously described. For set union ($\cup$) and set difference ($\setminus$), one can apply the principle of inclusion-exclusion. Denote by E_1 and E_2 two arbitrary expressions. We have $|E_1 \cup E_2| = |E_1| + |E_2| - |E_1 \cap E_2|$ and $|E_1 \setminus E_2| = |E_1| - |E_1 \cap E_2|$. Thus, we can rewrite queries with $\cup$ and $\setminus$ into several subqueries that do not contain these operations. Projections (π) can be handled using the distinct-count estimation techniques described in section 2.3.2.

D. Fixed-Precision Sampling

In *fixed-precision sampling*, we are given a query Q, an error bound ϵ—either absolute, relative, or combined—and a failure probability δ. The goal is to derive a sampling scheme that returns an estimate of $|Q|$ so that the probability that the estimation error exceeds ϵ is no more than δ. Fixed-precision sampling is important in practice because it can be used to determine the size of the sample required for a specific application. Clearly, a fixed-precision scheme should sample as little as possible to achieve the desired precision.

We make use of the *urn model* of Lipton and Naughton (1990) in our description. In this model, the result of the query (urn) is divided into disjoint partitions (balls). In each sampling step, the scheme selects a partition uniformly and at random and determines its exact size (number printed on ball). The selectivity estimate is given by the scaled sum of the sampled partition sizes. For example, the t_index scheme for estimating $|R_1 \bowtie R_2|$ would divide the dataset into $|R_1|$ partitions, one for each tuple in R_1. The partition of tuple $r_1 \in R_1$ is given by $r_1 \bowtie R_2$ (=number on ball r_1).

To solve the fixed-precision problem, we have to decide how many partitions we will have to sample. The *double sampling* scheme of Hou et al. (1991) takes a small pilot sample and estimates the query selectivity from the pilot sample. Based on this estimate, the sample size required to achieve the desired precision is estimated and the full sample is constructed in one run. As pointed out by Haas and Swami (1992), there is no theoretical guidance on how large the pilot sample should be. An alternative approach that does not require a pilot sample has been proposed by Lipton and Naughton (1990). Their *adaptive sampling* scheme makes use of an a-priori bound on the size of each partition. The scheme repeatedly samples a partition; it stops when the sum of the sizes of the sampled partitions exceeds a constant derived from the upper bound. Adaptive sampling is efficient when the bound is tight but may execute too many sampling steps otherwise. An improved version that avoids overly large samples is given in Lipton et al. (1990). The *sequential sampling* scheme of Haas and Swami (1992) and Haas et al. (1993) does not rely on any a-priori information. The scheme maintains an estimate of the selectivity and of the variance of the partition sizes. These estimates are used to compute large-sample confidence intervals as described in section 2.1.3C; sampling is stopped when the interval is small enough.

All of the above schemes make use of a *sanity bound*, which ensures that the sample does not get overly large when the selectivity is very low. When the sampling algorithm terminates due to the sanity bound, one can conclude that the selectivity is very low, which suffices for the purpose of query optimization.

A comparison of the various fixed-precision schemes has been performed by Ling and Sun (1995). Haas et al. (1994) and Haas and Swami (1995) investigated the cost of fixed-precision t_cross sampling for join selectivity estimation relative to the cost of actually computing the join.

E. Histogram Construction and Maintenance

Histograms are the most-widely used technique for selectivity estimation within relational database systems (Ioannidis 2003). A *histogram* of an attribute partitions the values of the attribute into a set of buckets and stores summary information for each bucket. Compared to random sampling, histograms have the advantage that they are smaller and faster to use. In fact, Kaushik et al. (2005) proved that equi-depth histograms—in which every bucket comprises the same number of values—are space-optimal (for worst-case data) for the purpose of selectivity estimation of range queries over a single attribute. Random sampling complements histograms; it is more suited for queries that involve either complex predicates or predicates on multiple attributes.

Nevertheless, it is common practice to construct histograms based on a random sample of the database. In their early work, Piatetsky-Shapiro and Connell (1984) and Muralikrishna and DeWitt (1988) suggest the use of sample sizes of roughly 1,000 tuples. More recently, Chaudhuri et al. (1998) discussed the question of how large the sample should be in order to provide strong guarantees on the error induced by sampling. Their results indicate that it is sufficient to sample $O(\log N)$ rows, a rather small sample size. Chaudhuri et al. (2004) provide algorithms that build a histogram using block-level sampling. Since it is non-trivial to maintain accurate histograms when the underlying data evolves, Gibbons et al. (1997) proposed to maintain a so-called *backing sample* instead.[15] Recomputation of the histogram from the sample is triggered when the estimates obtained from the histogram turn out to be inaccurate. The problem of maintaining a histogram is therefore reduced to the problem of maintaining a random sample.

2.3.2 Distinct-Count Estimation

The *distinct count* of a multiset R is the number of distinct items in R, also called the *zeroth frequency moment*. Accurate assessment of distinct counts is important for query optimization in relational databases. In fact, estimating the size of queries with duplicate-eliminating operations—such as projection or group-by operations—is equivalent to estimating the distinct count of the underlying data. Distinct-count estimates are also of interest for approximate query processing; see section 2.3.3.

There is a wealth of literature on distinct-count estimation in situations where a single pass over the data is feasible. An in-depth survey of the available techniques is given by Gibbons (2009). For example, the "probabilistic counting" method of Flajolet and Martin (1985) can be used to accurately estimate the distinct count using only $O(\log N)$ bits of memory, which is optimal (Alon et al. 1996). Most of these single-scan techniques are superior to random sampling in terms of both cost and precision. This does by no means imply that random sampling is useless for distinct-count estimation. In fact, when a scan of the entire dataset is impractical, random sampling is the only viable option. For example, such a situation occurs

[15] The actual sampling scheme is discussed in section 3.5.1E.

in query optimization: since both the predicates of the query and the attributes referenced by it are unknown in advance, single-pass methods that precompute (an estimate of) the distinct count cannot be used.

The use of random sampling for distinct-count estimation has a long tradition in both statistical and database literature. We only point out some of the most recent results; a survey and evaluation of the available estimators can be found in Bunge and Fitzpatrick (1993) and Haas et al. (1995).

The problem of estimating the number of distinct items from a random sample is known to be difficult. In fact, Charikar et al. (2000) have proven that, for *worst-case* data, no estimator that reads at most n items can provide a ratio error[16] less than $O(\sqrt{N/n})$ with high probability. They also give an estimator called *GEE* with $O(\sqrt{N/n})$ expected ratio error. Unfortunately, the worst-case optimality of GEE does not mean that it is the best estimator for all datasets. In fact, Haas et al. (1995) and Haas and Stokes (1998) have shown experimentally that no single estimator performs well for all datasets. In particular, some estimators exhibit good performance for low-skew datasets, while others perform well for high-skew data. Consequently, they propose a *hybrid estimator* that runs chi-square test on the sample to distinguish the low-skew and high-skew scenario; a different estimator is chosen depending on the test's outcome. Charikar et al. (2000) improve on the hybrid estimator by exchanging one of its sub-estimators with GEE.

Distinct-count estimation from a block-level sample is an even harder problem. Haas et al. (2006) briefly survey the statistical literature on estimating the number of species from a "quadrat sample", which coincides with block-level sampling from databases. They propose a generalized jackknife estimator that performs best in certain situations and, consequently, suggest a hybrid approach similar to the one above. Chaudhuri et al. (2004) propose to use any row-level estimator but to remove duplicate items within each block before doing so. This "collapse" operation is thought to reduce the effect of clustering in the blocks. They also extend the lower bound to block-level sampling: For worst-case data, no distinct-count estimator that examines at most n_B blocks can provide a ratio error of less than $O(\sqrt{N/n_b})$ with high probability. This bound is considerably weaker than the one for row-level sampling. As with selectivity estimation, materialized samples can be exploited to avoid the decrease in accuracy that results from block-level sampling, without increasing the sampling cost at query time.

In the section 2.3.3A, we also discuss (single-pass) sampling schemes that compute a sample of the distinct items in the dataset. Of course, these schemes can also be used to estimate distinct counts but, since they are more general, they have a higher computational cost.

[16]The ratio error of an estimate $\hat{D}$ of D is given by $\max(\hat{D}/D, D/\hat{D})$.

2.3.3 Approximate Query Processing

All of the previous applications can be seen as special cases of approximate query processing. In *approximate query processing*, we are given a query over a database and try to estimate the query's result based on random sampling. The advantage of approximate query processing over the exact computation of the result is that the computational cost can be reduced significantly. In applications such as online analytical processing (OLAP) or exploratory data analysis, where quick approximate results are acceptable, approximate query processing can increase the interactivity of the system. In other cases, such as data streaming, it is even infeasible to store or process the entire dataset so that approximate query processing is a must. But even if exact results are required, approximation may help to decide which exact results are likely to be of interest and which are not.

We consider SQL-style aggregate queries involving selection, projection, join and/or group-by:

$$\left.\begin{array}{l}\texttt{SELECT } A_1, \ldots, A_g, \texttt{ aggregate}(X) \\ \texttt{FROM } R_1, \ldots, R_k \\ \texttt{WHERE } \sigma \\ \texttt{GROUP BY } A_1, \ldots, A_g.\end{array}\right\} R \Bigg\} R_Q$$

The predicate σ includes both *local predicates* on $R_1, \ldots, R_k$ as well as the *join predicates* between these relations. Attributes $A_1, \ldots, A_g$ are referred to as *grouping attributes*. The group-by clause is optional, so that we also allow $g = 0$. The *aggregation attribute* X may correspond to an attribute of any input relation (e.g., the price of an order) or a function of one or more such attributes (e.g., price*tax). Aggregates of interest include count, distinct count, average, sum, standard deviation, variance, minimum, maximum, and median (or other quantiles). Not all techniques support all the aggregates and, in fact, most of the techniques specifically target sum queries. This is because sum queries are ubiquitous in applications and can be used to express other aggregates, such as counts and sums. Thus, unless stated otherwise, our subsequent discussion assumes a sum aggregate. Note that random sampling does not perform particularly well for minimum or maximum queries because these items are likely to be missed in the sample; however, it is still possible to derive an upper or lower bound, respectively.

For a given query of the form above, denote by R the dataset defined by only the `FROM` clause, restricted to the attributes referenced in the entire query (without eliminating duplicates). Similarly, denote by R_Q the dataset defined by both the `FROM` and `WHERE` clause; $R_Q = \sigma(R)$. Conceptually, the general approach to approximate the query result is to obtain a random sample S of either R or R_Q. The query result is then estimated from this intermediate sample. Approximate query answering provides methods to derive sample S as well as methods to estimate the query result from S. In particular, query sampling and permuted-data sampling techniques try to obtain S directly from $R_1, \ldots, R_k$ without generating R or R_Q. Materialized sampling

techniques precompute sample S; they do not even require access to $R_1, \ldots, R_k$ at query time.

A. Single-Table Query Without Group-By ($k = 1, g = 0$)

We first consider the simplest case of queries, that is, single-table queries without group-by. To answer this class of queries using query sampling, one obtains a uniform (row-level) sample S of R. Olken and Rotem (1986) have shown that $\sigma(S)$ is a uniform sample of $R_Q = \sigma(R)$ so that one can apply the estimation techniques of section 2.1.3 to compute the desired aggregate. The cost of row-level sampling can be avoided using permuted-data sampling—where R is stored in random order so that any prefix of R constitutes a uniform random sample—or materialized sampling, where S is precomputed. To further reduce the cost of sampling, any indexes that help to sample directly from R_Q (instead of from R) can also be exploited; one possibility is to use the B^+ tree sampling techniques of Olken and Rotem (1989).

In their well-known work on *online aggregation*, Hellerstein et al. (1997) show how to present to the user a running estimate of the query result; the estimate is refined as more and more tuples are processed. A discussion of how to compute confidence intervals in the context of online aggregation can be found in Haas (1997, 1999).

All the above approaches work well unless either (i) σ has low selectivity or (ii) the data is highly skewed. In both cases, large sample sizes are required to provide reasonably accurate estimates (Chaudhuri and Motwani 1999). Several (materialized) techniques that deal with either (i), (ii), or both have been proposed in the literature. These techniques can roughly be divided into two classes: techniques that are based on workload information and techniques that are not.

The idea behind *workload-based techniques* is to find a sampling design that performs well for a prespecified workload. Approximate queries that are "similar" to the prespecified workload are expected to profit (in terms of precision) from the workload-based design. Likewise, "dissimilar" queries are penalized. One of the first workload-based sampling techniques is due to Ganti et al. (2000). In essence, their *icicles* constitute a uniform random sample of the so-called extension of R, which is a multiset that contains R and, for any query in the workload, the set of tuples referenced by the query.[17] The key idea is that tuples that are often used in query evaluation are more likely to be present in the sample. Query evaluation is based on the assumption that the aggregate value of each tuple is independent from the tuple's probability of being sampled. This assumption may or may not hold in practice; there does not appear to be a way to verify it based on the sample. A similar approach suggested by Chaudhuri et al. (2001) avoids this problem by making use of weighted sampling; the weight of each item is proportional to the number of queries in the workload that reference it. Going one step further, the techniques in Chaudhuri et al. (2001) and Chaudhuri et al. (2007) partition table R into "fundamental regions" so that every query in the workload either references

[17] In our notation, query $\sigma(R)$ references a tuple $r \in R$ if and only if $r \in \sigma(R)$.

all or none of the tuples in a fundamental region. The fundamental regions are taken as strata for stratified sampling; sample space is allocated to the strata so that the error in the workload is minimized. The problem with this approach is that, in practice, the number of fundamental regions can be significantly higher than the space available in the sample. In this case, some (deterministically determined) subset of regions are known to be unrepresented in the sample; the resulting design neither constitutes a stratified nor a probability sampling design.

If no workload is available, or if the available (exact) workload does not permit conclusions about the approximate workload, or if penalization of "dissimilar" queries is not acceptable, workload-based techniques cannot be used. Instead, *workload-independent techniques* are needed. Chaudhuri et al. (2001) proposed a stratified scheme called *outlier indexing* to deal with highly-skewed data. It is based on the observation that tuples with an extremely low or high value in the aggregation attribute deteriorate the precision of the estimate. The scheme detects these outliers and stores them in a separate data structure; a uniform sample is taken from the rest of the table. Outlier indexing has been generalized to multiple aggregation attributes in Rösch et al. (2008). The *approximate pre-aggregation* techniques introduced by Jermaine (2003) and later Jin et al. (2006) incorporate additional summary statistics (the "pre-aggregation" part) into the sampling-based estimate (the "approximate" part). In particular, Jin et al. (2006) proposes to use existing knowledge to obtain an alternative "negative estimator". For example, suppose that we know that the aggregation attribute sums to v over the entire relation R but that the query has a predicate and thus addresses only a subset of the data. The standard (positive) estimator determines the scaled sum of the items in R that satisfy the predicate. In contrast, the negative estimator would determine the scaled sum of the items in R that do *not* satisfy the predicate and subtract the result from v. Both estimators are combined to achieve the final estimate, which can be significantly more precise than each of the estimators on their own.

In the remainder of this section, we discuss issues that arise when R is a multiset.[18] Gibbons and Matias (1998) propose to store the sample S of R in compressed form, in which the sample consists of (item, frequency)-pairs. This way, a larger sample can be stored in the available space. The paper also proposes a *concise sampling* and a *counting sampling* scheme that maintain a random sample of the dataset in compressed form; the schemes are particularly useful for finding frequent items, see section 2.3.4B.

To support distinct-count queries, the schemes of Gibbons (2001), Cormode et al. (2005), and Frahling et al. (2005) maintain a sample from the set of distinct items in the aggregation attribute X. Each distinct item is equally likely to be present in the sample. Queries in which the predicate involves attributes other than X cannot be answered from such a sample.[19] For this reason, Gibbons (2001) associates with each

[18] Recall that R is restricted to the attributes referenced by the query, which may not include the primary key of R_1.

[19] Nevertheless, these samples are useful in some applications. For example, consider a sample of IP addresses ($= X$) and a query like "How many distinct IP addresses have the form 192.*?".

sampled X-value x the set $\{ r \in R \mid r.X = x \}$ of tuples from R that have this distinct value (or, if space is an issue, a uniform sample thereof). This *distinct sample* can be used to answer a variety of distinct-count queries such as "What is the distinct number of customers that placed an order in 2008?" As discussed previously, queries of this kind are difficult to answer from uniform (multiset) samples.

B. Single-Table Query With Group-By ($k = 1, g \geq 1$)

Queries with group-bys can directly be answered with the above techniques. Suppose that we are given a group-by query and that exact computation reveals that the query result contains m different groups $G = \{ g_1, \ldots, g_m \}$, where $g_i \in A_1 \times \ldots \times A_g$ for $1 \leq i \leq m$. Then, the query result can be decomposed into m subqueries of the form

$$\begin{array}{l} \texttt{SELECT } A_1, \ldots, A_g, \texttt{ aggregate}(X) \\ \texttt{FROM } R_1, \ldots, R_k \\ \texttt{WHERE } \sigma \texttt{ AND } (A_1, \ldots, A_g) = g_i. \end{array}$$

The union of all m subqueries is identical to the query result. To estimate the result of the group-by query from a random sample S, we proceed as before but replace G by the set G' of groups that actually occur in S. Xu et al. (2008) investigate the problem of assigning confidence bounds to the so-obtained aggregates when S is a simple random sample of R. In some cases, not all the groups are represented in the sample so that $|G'| < m$; that is, some groups are missing. Small groups are especially likely to be missed by random sampling so that the problem of missing groups is often referred to as *small-group problem*.

For query sampling, the *index striding* technique of Hellerstein et al. (1997) can be used to avoid the small-group problem when an index on the grouping attributes is available. The idea is that the index can be exploited to sample uniformly from each group, in a round-robin fashion. In the context of online aggregation, where a running estimate is presented to the user, this has the additional advantage that each group is updated with the same frequency. Index striding can also be exploited to sample more tuples from groups in which the estimate is perceived to be of low precision.

Acharya et al. (2000) propose a stratified sampling design called *congressional sampling*, in which each group gets its own stratum. The scheme distributes the available sample space in such a way that each group gets a "fair" amount of space. Of course, this technique requires that (supersets of) the grouping attributes are known in advance. In contrast, the *small-group sampling* scheme of Babcock et al. (2003) materializes a uniform sample of R along with some information about small groups. In more detail, for each potential grouping attribute, the scheme scans the dataset for small groups and stores all tuples belonging to a small group into a separate "small-group table". These tables are exploited at query time to include small groups into the query result.

C. Join Queries ($k > 1$)

We restrict our attention to joins between two relations R_1 and R_2, $k = 2$. Most of the techniques below can be extended to the more general case. Also suppose that we want to estimate the sum over aggregation attribute X.

A well-known result of Olken and Rotem (1986) is that, in general, the join of two uniform samples from R_1 and R_2 does *not* constitute a uniform sample of $R_1 \bowtie R_2$. Observe that the t_cross sampling scheme discussed in the context of join selectivity estimation joins the samples of R_1 and R_2; figure 2.8b visualizes the non-uniformity of the result. Of course, one can fall back to t_indep sampling to obtain a uniform sample, figure 2.8a. Recall that t_indep samples uniformly from $R_1 \times R_2$; the join selectivity estimate equals the fraction of sampled tuple pairs that join. In approximate query processing, however, we want to compute aggregates over these joining pairs. Clearly, a significantly larger number of tuple pairs is required to achieve sufficient accuracy. We conclude that t_indep is not a promising approach for approximate query processing, although it does produce uniform samples of $R_1 \bowtie R_2$.

When there is an index on the join attribute on table R_2, Hellerstein et al. (1997) propose to use t_index sampling (figure 2.8c). The idea is to sample a random tuple from $r \in R_1$, join it with R_2 using the index, and sum over the values of X in $r_1 \bowtie R_2$. Denote the result by $X(r)$ and observe that $X(r)$ is the "contribution" of r to the query result. We can view $X(r)$ as a random sample of a modified relation R_1', which is derived from R_1 by replacing every tuple with its contribution to the query result. Thus, for the purpose of estimation, we reduced the join query to a single-table query and the estimators of section 2.1.3 can be used.

If there is no index on R_2, the situation becomes more difficult. Interestingly, t_cross sampling can be applied even though the resulting sample is non-uniform. Haas (1997) has shown that the standard sum estimator for uniform sampling, which takes the sample sum and scales up, remains unbiased for t_cross sampling. He also derived large-sample confidence intervals for estimators of the sum, average, standard deviation, or variance of X. Based on these results, Haas and Hellerstein (1998) and Haas and Hellerstein (1999) developed the *ripple join*, which is a generalization of t_cross sampling. The key idea of the ripple join is to sample β_1 tuples from R_1, then β_2 tuples from R_2, then again β_1 tuples from R_1, and so on. Whenever a block of tuples from one of the relations has been sampled, it is joined with the sample of the other relation. Thus, t_cross sampling is obtained when $\beta_1 = \beta_2 = 1$. The parameters β_1 and β_2 can be used to control the update frequency of the estimate in the context of online aggregation. Furthermore, these parameters can be tuned to reduce the number of sampling steps required to achieve the desired precision.

The ripple join as described above assumes that the samples of R_1 and R_2 both fit into memory. Jermaine et al. (2005) pointed out that the ripple join becomes unusable when the sample sizes exceed the available memory. To overcome this problem, they propose the *SMS* join, which is a combination of the ripple join and a classic sort-merge; it scales well with the size of the samples. Jermaine et al. (2007) extend the ideas behind the SMS join to more than 2 relations. Their *DBO*

system constitutes an approximate query processing system based on permuted-data sampling. In fact, the cost of both the ripple join and the SMS join is reduced significantly when the data is stored in random order because block-level sampling can be used without deteriorating precision.

Materialized samples can be exploited to reduce any join query to a single-table query, both in terms of estimation complexity and cost. The idea is to precompute and maintain a sample S of $R = R_1 \bowtie R_2$; the query is then interpreted as a single-table query on R and the required sample of R is readily available. Using this approach, all the sophisticated materialized sampling techniques for single-table queries that we discussed in the previous sections can directly be applied to join queries. If a sample of R is unavailable but there exist materialized samples of R_1 and R_2, these samples can be fed into a ripple join (or SMS join) to reduce its I/O cost.

2.3.4 Data Mining

Data mining, or more generally *knowledge discovery*, is "the nontrivial extraction of implicit, previously unknown, and potentially useful information from data" (Frawley et al. 1992). Algorithms employed in the data mining process are usually expensive in terms of both runtime cost and space consumption. Random sampling is used to improve the efficiency of data mining tasks that search for "regularities" in the data; these regularities are likely to be retained in a sample. In fact, Palmer and Faloutsos (2000) point out that many statistical vendors make use of uniform sampling to handle large datasets; see, for example, the best-practices paper of SAS Institute Inc. (1998). Along the same lines, Chen et al. (2002) state that "a number of large companies routinely run mining algorithms on a sample of their data rather than on the entire warehouse."

In our subsequent discussion, we briefly describe the application of random sampling to clustering, to the discovery of frequent items and association rules, and to the discovery of correlations and constraints.

A. Clustering

Clustering is "the art of finding groups in data" (Kaufman and Rousseeuw 1990). Clustering algorithms divide the data into groups of similar objects; objects belonging to different groups should be as dissimilar as possible. For example, a sales company might want to divide its customers into groups that exhibit a similar shopping behavior. Well-known clustering algorithms include the *k-means* algorithm of MacQueen (1967), the *PAM* algorithm of Kaufman and Rousseeuw (1990), the *BIRCH* algorithm of Zhang et al. (1996), and the *DBSCAN* algorithm of Ester et al. (1996).

To cope with large datasets, several authors have proposed to run the clustering algorithm on a random sample instead of using it on the entire dataset. Since clustering algorithms are typically expensive—they often have superlinear time

complexity—, such an approach can significantly reduce the computational cost. Optionally, objects not belonging to the sample can be assigned to their closest cluster in a postclustering step. MacQueen (1967) already proposes this sample-and-postcluster approach for k-means. Experiments by Joze-Hkajavi and Salem (1998) suggest that random sampling for k-means compares favorably to the alternative *CF tree* method of Zhang et al. (1996). Similarly, Guha et al. (1998) make use of uniform sampling to scale their *CURE* algorithm to large datasets. The *CLARA* algorithm of Kaufman and Rousseeuw (1990) draws multiple random samples, applies PAM to each of these samples, and outputs the best achieved clustering. For BIRCH, Palmer and Faloutsos (2000) propose a weighted "density-biased" sampling scheme for the case that cluster sizes are heavily skewed. The key idea is to undersample dense areas and oversample spare areas, which ensures that low-density clusters are retained in the sample. Kollios et al. (2001) extend these ideas and present a density-biased scheme in which the sampling probability is a tunable function of the local density around each point. They apply their technique for both clustering and distance-based outlier detection. Gionis et al. (2007), who propose techniques that "aggregate" multiple clusterings into a single one, also make use of random sampling to speed up their algorithms.

B. Frequent Items

Another important problem in data mining is the discovery of the frequent items—also called *heavy hitters* or *hot items*—in a dataset. For example, a sales company might be interested in their top-selling items or their most active customers. In this section, we consider *iceberg queries*, which ask for all items that have a relative frequency of at least f.[20] As Fang et al. (1998) pointed out, the naive computation of the exact result based on either hashing or sorting may be too expensive in some applications. Thus, many more sophisticated techniques, both exact and approximate, have been proposed to reduce I/O and/or memory costs. We limit our attention to sampling-based approaches; there do exist more efficient special-purpose techniques not based on random sampling, but random samples are applicable to a wider class of queries.

One can show that with a probability of at least $1 - \delta$, a uniform sample of size $\ln(\delta^{-1} f^{-1})/f$ (including duplicates) contains *all* items with relative frequency of at least f. For example, with probability of at least 99%, a sample size of 1,000 items suffices to find all items with relative frequency of at least 0.01. Of course, the sample is likely to contain in addition items with relative frequency lower than f. Fang et al. (1998) argues that, if desired, these "false-positives" can be eliminated using a scan of the dataset. They also propose several exact algorithms that use sampling in a preprocessing phase.

Gibbons and Matias (1998) propose two materialized sampling schemes called *concise sampling* and *counting sampling*. Both schemes require a constant amount

[20] The relative frequency of an item that occurs k times in a dataset of size N is given by k/N.

of space; they run a variant of Bernoulli sampling and reduce the sampling rate whenever the sample gets too large.[21] To make efficient use of the available space, the sample is stored in compressed form, that is, items that occur more than once in the sample are represented by an (item, frequency)-pair. The schemes differ in their use of the frequency counter. For concise sampling, the counter is equal to the number of times an item has been accepted into the sample. For counting sampling, however, the counter equals the number of times the item has been seen since its first acceptance into the sample. As shown by Manku and Motwani (2002), a (slightly modified) counting sample of size $2\ln(\delta^{-1}f^{-1})/\epsilon$, where $0 < \epsilon \leq f$, can be used to report with a probability of at least $1 - \delta$ all the items with a relative frequency of at least f. In contrast to uniform sampling, the scheme guarantees that it does not report items with frequency less than $f - \epsilon$.

C. Association Rules

In association rule mining (Agrawal et al. 1993), we are given a dataset of so-called transactions, where each transaction consists of several items. For example, in a supermarket application, a transaction may correspond to a shopping cart and an item corresponds to a product in the cart. A rule $X \Rightarrow Y$, with X and Y being disjoint sets of items, is a statement of the form "a transaction that contains X also contains Y." An *association rule* is a rule $X \Rightarrow Y$ in which itemset X is frequent and the rule is satisfied by most transactions that contain X. For example, the association rule $\{\,\text{bread}, \text{butter}\,\} \Rightarrow \text{milk}$ means that, with high confidence, a customer who buys bread and butter will also buy milk.

Association rule mining is related to finding frequent itemsets. In fact, the well-known *a-priori algorithm* of Agrawal and Srikant (1994) first discovers single items that are frequent, then pairs of items that are frequent, then triples of items, and so on. In each pass, the algorithm only considers itemsets that could potentially be frequent and prunes away the rest. Toivonen (1996) proposes a two-phase variant of the a-priori algorithm. In the first phase, a random sample is used to determine sets of items that are likely to be frequent. In the second phase, the algorithm scans the entire dataset and verifies the frequency of both the discovered itemsets and a carefully selected fraction of the pruned itemsets. For sufficiently large samples (see the previous section), none of the pruned itemsets will turn out to be frequent. In this case, the algorithm produces the exact result. Otherwise, when some of the pruned itemsets are frequent, a second scan of the data is performed to catch all itemsets that may have been missed.

One would somehow like to avoid scanning the database to verify the rules obtained from the sample. Clearly, if the initial sample is large, we are sure to catch all the important association rules. However, the larger the sample, the higher the cost of finding the rules supported by it. To overcome this problem, Chen et al. (2002) and Brönnimann et al. (2003) each propose a two-phase algorithm—called *FAST*

[21] A more detailed description can be found in section 3.5.1B.

and *EASE*, respectively—that works with any algorithm for mining association rules. In the first phase, the algorithms obtain a relatively large random sample of the database. In the second phase, a reduced sample is constructed in such a way that the "difference" between the frequent itemsets in the full sample and the reduced sample is minimized. The reduced sample serves as input to an association rule mining algorithm. The runtime cost and accuracy of both algorithms are compared in Brönnimann et al. (2004).

D. Correlations and Constraints

Data mining is also concerned with finding correlations and constraints in the data. The *BHUNT* algorithm of Brown and Haas (2003) discovers fuzzy algebraic constraints between pairs of attributes in a relational database. For example, such a constraint could say that the number of days between shipment and receipt of a product is likely to lie either in the interval $2-5$ days (national shipment) or the interval $12-19$ days (international shipment). To find the constraints, BHUNT first constructs candidate pairs of columns that are potentially related and interesting. Since the number of such pairs can be large, BHUNT searches for constraints in a random sample of the tuples corresponding to each of the column pairs. The *CORDS* algorithm of Ilyas et al. (2004) extends BHUNT in that it detects arbitrary correlations between columns. To do so, CORDS constructs contingency tables based on a random sample of the database; it then estimates the "mean-square contingency" as a measure of correlation. The outcome of the algorithm can be visualized in a dependency graph.

A related problem is that of constraint verification, that is, the check of whether a set of constraints is satisfied by the data. The constraints may originate, for example, from an older version of the dataset or from a dataset that is believed to be similar to the one under investigation. Constraint verification has been considered by Kivinen and Mannila (1994), who make use of random sampling to find constraints that are "clearly false". The remaining constraints are likely to hold for a large part of the database and, if desired, they can be verified exactly.

Random sampling has also been applied to efficiently estimate the similarity between objects. For large collections of Web documents, Broder (1997) proposes a variant of "min-hash sampling"[22] to estimate the resemblance and containment for any two of these documents. Datar and Muthukrishnan (2002) applied min-hash sampling to estimate rarity and similarity over data stream windows.

2.3.5 Other Applications of Database Sampling

In this section, we briefly list some other applications where database sampling appears to be useful.

[22] Min-hash sampling is discussed in section 3.5.3D.

Denning (1980) states that the U.S. Census Bureau uses random sampling to avoid inference of individuals; she proposes a variant of "min-hash sampling" to prevent such inference from database queries.

Willard (1991) applies selectivity estimates obtained from a random sample to choose between different competing algorithms. He also shows that a sample size of $O(N^{2/3})$ is cost-optimal for the class of "differential database batch queries."

Ikeji and Fotouhi (1995) propose a stratified sampling scheme to quickly produce partial results of non-aggregate queries. In contrast to aggregate queries, these early results do not have to be chosen in a uniform manner.

For parallel sorting, DeWitt et al. (1991b) make use of random samples to determine a "probabilistic splitting vector" such that every node has about the same amount of work. DeWitt et al. (1992) propose a related approach for efficient processing of parallel joins, in which random samples are used to choose the best join algorithm and distribute tuples to nodes. DeWitt et al. (1991a) apply random sampling to determine the partitions used by a "partitioned band join".

In the context of data warehousing, Brown and Haas (2006) propose a "synopsis warehouse" that consists of synopses of the partitions in a data warehouse. The synopses can be used for a variety of applications, including approximate query processing and metadata discovery. They present several sampling algorithms useful in this setting; we will discuss most of these algorithms in the course of this thesis.

Chapter 3

Maintenance of Materialized Samples

Most of the techniques for materialized sampling implicitly assume that the underlying database is static. Under this assumption, a sample once computed from the database remains valid for all times. In many practical applications, however, the database is subject to changes so that the assumption of a static database does not hold. In this situation, any modifications of the database have to be reflected in the sample to maintain its validity. Unless maintenance can be done efficiently, the cost of maintaining the sample can outweigh the advantages of materialization. In this chapter, we review and classify the available work on sample maintenance with a special focus on efficiency. We will see that maintenance can be done efficiently, but that there are still a lot of open problems that have to be resolved. Some of these open problems are investigated in chapters 4–7.

Our discussion is mainly concerned with the maintenance of uniform samples. As discussed previously, uniform sampling plays a major role in database sampling because, in database applications, the intended use of the sample is often not known in advance. The advantage of uniform sampling is that samples are representative in the sense that they are not biased towards a certain part of the underlying data. Also, many statistical estimators and confidence-bound formulas for these estimators assume the uniformity of the sample. But even when information about the intended use of the sample is available, uniform sampling is still essential. In fact, uniform samples are often used as a building block for more complex techniques that are tailored to a specific type of database queries. For example, one common approach is to store a uniform sample accompanied by some auxiliary information, which is then used to improve estimates for certain types of queries. Another example is stratified sampling, where after the division of the population into strata, uniform sampling is applied to each of the strata.

We start with a comparison of materialized samples and materialized views in the context of relational database systems in section 3.1, where we also outline the scope of this thesis. Before reviewing existing techniques for sample maintenance, we use section 3.2 to define the basic terms and notation used throughout the thesis. In section 3.3, we develop a classification of schemes for sample maintenance with respect to sampling semantics, supported transactions, sample size, and space consumption. This classification spans the space that database sampling schemes have to cover. Afterwards, in section 3.4, we discuss sampling schemes in the area of survey sampling; again, these represent the ground on which database sampling

schemes are built. Finally, in section 3.5, we give a detailed review of the existing work on sample maintenance for both database sampling and data stream sampling. During our discussion, we point out deficiencies of the existing techniques, if any, and outline our own contributions that are presented in the subsequent chapters.

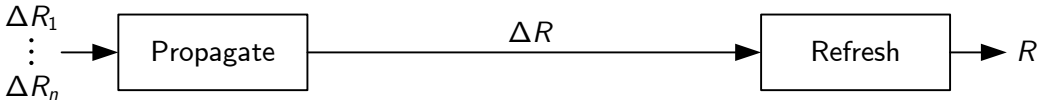
3.1 Relationship to Materialized Views

The techniques in this thesis apply to a broad spectrum of applications and systems. The following discussion is specific to the case where the sample is maintained within a relational database system.

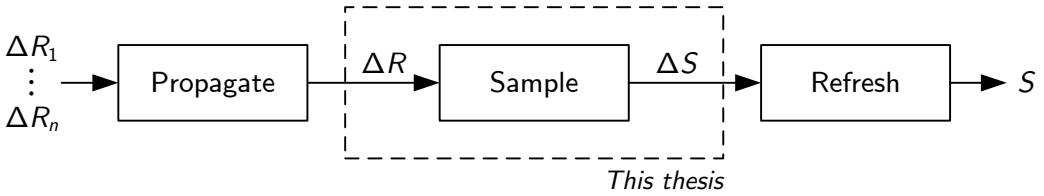
Sample maintenance within a relational database system is closely related to the problem of maintaining a materialized view (Gupta and Mumick 1999). And in fact, a materialized sample can be interpreted as a materialized view of the underlying dataset (Olken and Rotem 1992; Larson et al. 2007). This interpretation is not supported in this thesis because there are significant differences between materialized views and materialized samples. The key difference is that the view is unique while the sample is not. Given a database state and a view definition, there is a unique relation corresponding to the content of the view. The situation is different for materialized samples because, in addition to the database state and view definition, both the sampling scheme and the random inclusion/exclusion decisions carried out during the execution of the scheme determine the content of the sample. The interpretation taken in this thesis is that a materialized sample is a sample derived from a view or materialized view, but does not constitute a materialized view itself.

In materialized view maintenance, the important questions are *when* and *how* to maintain the view. The answer to the first question is called the maintenance *policy*, which is either immediate or deferred. *Immediate views* are updated immediately after the underlying data has been updated. *Deferred views* are updated at some later point in time. Examples include periodic maintenance, where the view is updated at regular time intervals, or lazy maintenance, where the view is updated whenever it is queried. The actual maintenance of the view is performed in two steps called propagate and refresh. The steps are illustrated in figure 3.1a, where the view R is derived from base relations $R_1, \ldots, R_n$. In the *propagate step*, the changes ΔR to the view R that result from the changes ΔR_i to base relations R_i, $1 \leq i \leq n$, are computed. This process may involve access to the base relations. In the *refresh step*, ΔR is applied to the materialized view.

As described above, a materialized sample is derived from a view R, just as R is derived from the base relations. There is usually no need for R to be materialized. The concept of a view maintenance policy directly applies to sample maintenance. In this thesis, we assume the immediate policy. Techniques for immediate maintenance can be used to implement deferred maintenance by keeping a log file of all transactions, although this approach may be less efficient than specialized techniques (such as those proposed in Jermaine et al. 2004; Gemulla and Lehner 2006; Pol et al. 2008; Nath and Gibbons 2008). Materialized sample maintenance consists of three steps: propagate,



(a) Materialized views



(b) Materialized samples

Figure 3.1: View and sample maintenance in an RDBMS. The view/dataset is given by $R = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$.

sample, and refresh. The steps are illustrated in figure 3.1b. The propagate and the refresh step are identical to the corresponding steps in view maintenance and available techniques can be used directly. The *sample step*, which we will discuss extensively in this thesis, computes the changes ΔS to the sample S based on the changes ΔR to the view.

The separation of the propagate and the sample step has been made primarily to emphasize the scope of this thesis. It is somewhat artificial because, when R is not materialized, one may consider computing ΔS directly from the ΔR_i. This direct approach may increase efficiency. Since the sample S can be seen as a subset of the dataset R, not all the transactions that affect R also have an effect on S. For example, consider two tables R_1 and R_2 and suppose that there is a many-to-one relationship between the two tables. Further suppose that the sample is derived from the view $R = R_1 \bowtie R_2$. Whenever an item r is inserted into R_1, a tuple $r' = \{ r \} \bowtie R_2$ is propagated to the maintenance scheme. When the sample inclusion/exclusion decision does not depend on the value of r' but solely on its existence,[1] the computation of the join between r and R_2 is unnecessary for the case where r' is excluded from the sample. Combined approaches, which handle the propagate and the sample step as a unified whole, are therefore an interesting direction for future research.

To summarize, we assume in this thesis that the changes to the dataset R, from which the sample is drawn, are propagated to the sample maintenance algorithm. If R is a relational table, then updates to the table correspond to updates of R. If R is a view defined over one or more tables, changes to R are computed using traditional view maintenance techniques. When we talk about the cost of sampling maintenance,

[1] Some of the schemes discussed in this thesis work precisely in this way.

we mean the cost of computing and applying ΔS given ΔR. This viewpoint allows us to focus on the fundamental issues of sample maintenance.

3.2 Definitions and Notation

We assume throughout this thesis that the dataset R is a subset of a possibly infinite set $\mathscr{R} = \{ r_1, r_2, \dots \}$ of unique, distinguishable items. For example, the set $\mathscr{R}$ might correspond to the domain of a relational table, to the set of all IP addresses or to a collection of XML documents. We generally allow R to be a multiset, that is, R may contain multiple copies of items from $\mathscr{R}$. We do not require R to be materialized.

Changes to R are modeled using a (possibly infinite) sequence of transactions $\gamma = (\gamma_1, \gamma_2, \dots)$. We denote by R_0 the initial dataset and by R_i, $i > 0$, the dataset that results from the applications of the first i transactions to R_0. Similarly, we denote by N_i the number of items in dataset R_i, including duplicates. Without loss of generality, we assume that the initial dataset R_0 is empty and thus $N_0 = 0$. Each transaction γ_i is of one of the following three types:

- An *insertion transaction* $+r$ corresponds to the insertion of item $r \in \mathscr{R}$:

$$R_i = R_{i-1} \uplus \{ r \} .$$

- A *deletion transaction* $-r$ corresponds to the deletion of item $r \in \mathscr{R}$:

$$R_i = R_{i-1} \setminus^{+} \{ r \} .$$

- An *update transaction* $r \to r'$ corresponds to the replacement of item $r \in \mathscr{R}$ by item $r' \in \mathscr{R}$:

$$R_i = R_{i-1} \setminus^{+} \{ r \} \uplus \{ r' \} .$$

Here, $\uplus$ and $\setminus^{+}$ denote the multiset versions of the set operators $\cup$ (set union) and $\setminus$ (set difference). Thus, when item r occurs multiple times in R_{i-1} upon processing a deletion or update transaction, only a single copy of the item is updated or deleted. We require the stream of transactions to be *feasible*: whenever a transaction removes or updates an item, it must be present in the dataset at the start of the transaction. In some scenarios (e.g., set sampling), additional restrictions on the sequence of transactions apply and will be discussed when needed. In some applications, multiple transactions are clustered, that is, they occur simultaneously. In this case, we break up the clustering and perform the operations in arbitrary order (or in an order specified by the sampling scheme).

With these definitions at hand, we define a *maintenance scheme* as a randomized algorithm that maintains a sample S_i from R_i for all i. As usual in practice, we assume throughout this thesis that the sequence γ is not known in advance; the maintenance scheme cannot look ahead of the transaction currently processed. On the same lines, we assume that the application that produces γ is oblivious to the

state of the sample; γ is independent from the inclusion/exclusion decisions of the maintenance scheme.

In general, it is desirable that a maintenance scheme computes S_i from S_{i-1} and γ_i only, but there are schemes that require access to R_i. Perhaps the simplest such scheme discards the current sample when γ_i arrives and computes a fresh sample from R_i by using some sampling scheme. This approach is clearly inefficient, but it shows that every sampling scheme can be used to maintain a sample if access to the dataset is available. Similarly, a maintenance scheme can be used to compute a sample from R by treating every element of R as an insertion transaction into an initially empty dataset. Thus, the concepts of sampling scheme and maintenance scheme are related. Whenever the intended meaning can be derived from the context, we drop this distinction and refer to both schemes as sampling scheme.

3.3 Properties of Maintenance Schemes

Recall that a sampling design is specified by a probability distribution over the set of all possible samples. When a maintenance scheme is used to maintain a sample of an evolving dataset, the set of possible samples—and therefore, the sampling design—is also evolving. Thus, a maintenance scheme results in a *sequence of sampling designs*, one for each transaction in the sequence of transactions. One might extend sampling theory by temporal aspects, but there is little to be gained for the purpose of this thesis. Instead, we are interested in properties valid for all the designs produced by the scheme. These time-invariant properties are discussed in the following. They span a "space" that can be used to classify the available maintenance schemes. Dimensions of the space include sampling semantics, supported types of transactions, sample size properties and memory consumption.

3.3.1 Sampling Designs

We only consider sampling designs where $S_i \subseteq R_i$, that is, the sample does not contain any items not present in the underlying dataset. A maintenance scheme is said to be be *uniform* if the resulting sampling designs are all uniform. That is, for every $i \geq 0$,

$$\Pr[\, S_i = A \,] = \frac{\Pr[\, |S_i| = |A| \,]}{\binom{N_i}{|A|}} \tag{3.1}$$

for an arbitrary but fixed $A \subseteq R_i$. For many maintenance schemes, the distribution of sample size is not fixed and varies from transaction to transaction. Also note that equation (3.1) only gives the marginal distribution of the samples. Samples S_{i-1} and S_i are usually dependent because most maintenance schemes reuse information in S_{i-1} to compute S_i. Often, but not always, we have

$$\Pr[\, S_i = A \mid S_{i-1} = B \,] \neq \frac{\Pr[\, |S_i| = |A| \,]}{\binom{N_i}{|A|}}$$

for arbitrary but fixed $A \subseteq R_i$, $B \subseteq R_{i-1}$. This inter-sample dependency usually does not pose a problem; it may even have advantages. For example, when the sample is used to monitor the evolution of a population parameter over time, inter-sample dependencies can be exploited to provide more precise estimates; see the discussion in Krishnaiah and Rao (1988, ch. 8).

A maintenance scheme is said to sample *with replacement* if all S_i are with-replacement samples. A maintenance scheme is said to sample *without replacement* if all the S_i are without-replacement samples. For weighted sampling, we assume that the weights associated with the items in $\mathscr{R}$ are fixed. Then, a maintenance scheme is said to be *weighted* if for every item $r_j \in R_i$, $i \geq 0$, it holds

$$\Pr[\, r_j \in S_i \,] \propto w_j$$

with w_j being the weight of r_j and $\propto$ denoting proportionality. Again, the distribution of the sample size may change over time. The situation is more difficult for stratified and cluster sampling designs because the strata and clusters may be evolving themselves. We do not go into further detail here, but see section 7.2 for an example of a stratified maintenance scheme.

3.3.2 Datasets and Sampling Semantics

We distinguish three different sampling semantics; some of them only apply to certain types of datasets: set sampling, multiset sampling and distinct-item sampling.

A. Set Sampling

The most common form of sampling in classical sampling theory is *set sampling*. In set sampling, the dataset is a true set, that is, it does not contain any duplicates. The sample may contain duplicate items if sampling is with replacement. A maintenance scheme supports set sampling if sequences resulting in true sets are supported by the scheme. In such a sequence, an insertion transaction $\gamma_i = +r$ is allowed to occur only if $r \notin R_{i-1}$. Set sampling is the most basic sampling semantics and is supported by virtually all maintenance schemes.

B. Multiset Sampling

In the context of database systems, sampling is sometimes performed from multisets instead of sets. For example, suppose that we want to maintain a sample from a table of a relational database. For improved efficiency, only the columns relevant for estimation purposes are to be included into the sample. Then, the primary key, which ensures the uniqueness of the items in the table, is often omitted so that sampling is performed from a table that now may contain duplicate items. In *multiset sampling*, every item from $\mathscr{R}$ may occur more than once in the dataset. Multiset sampling is more general than set sampling; every multiset sampling scheme can directly be used for set sampling.

The semantics of multiset sampling is more involved than the one of set sampling. Even if sampling is without replacement, the sample may contain duplicates because the dataset does. To transfer the ideas of set sampling to multiset sampling, we say that a multiset sampling scheme *corresponds* to a set sampling scheme if the following two procedures yield equal sampling designs:

1. Run the multiset sampling scheme on the dataset R.
2. Label all items in R with a unique identifier. Run the set sampling scheme on the labeled version of R. Finally, remove the labels from the items in the sample.

For example, consider the dataset $R = \{ A, A, B \}$ and the following multiset sampling scheme: Select an item from R uniformly and at random. Include the selected item into the sample and remove a single copy of the selected item from R. Then, independently select another item uniformly and at random from the remaining dataset and include the selected item into the sample. The resulting distribution of the multiset sample S is

$$\begin{aligned} \Pr[\, S = \{ A, A \}\,] &= 1/3 \\ \Pr[\, S = \{ A, B \}\,] &= 2/3. \end{aligned} \tag{3.2}$$

The above sampling scheme corresponds to simple random sampling of size 2. To see this, we first label the items in R and obtain dataset $R' = \{ A_1, A_2, B_3 \}$. Then, we compute a simple random sample S' from R'. The probability distribution of S' is given by

$$\begin{aligned} \Pr\left[\, S' = \{ A_1, A_2 \}\,\right] &= 1/3 \\ \Pr\left[\, S' = \{ A_1, B_3 \}\,\right] &= 1/3 \\ \Pr\left[\, S' = \{ A_2, B_3 \}\,\right] &= 1/3. \end{aligned}$$

Finally, we remove the labels from S' and obtain the distribution in (3.2). Using the method described above, we can apply properties of set sampling schemes to multiset sampling schemes. In particular, a multiset sampling scheme is *uniform* if and only if there exists a corresponding set sampling scheme that is uniform (as in the example above).

C. Distinct-Item Sampling

In distinct-item sampling, the dataset is also a multiset, but sampling is performed from the distinct items in the dataset. Distinct-item sampling is applied when the number of copies of an item should not affect its probability of being sampled. If sampling is without replacement, the sample does not contain duplicates.

Again, we make use of a correspondence to set sampling. A distinct-item sampling scheme *corresponds* to a set sampling scheme if the following two procedures yield equal sampling designs:

1. Run the distinct-item sampling scheme on R.
2. Run the set sampling scheme on $D(R)$, the set of distinct items in R.

For example, set $R = \{ A, A, B \}$ and thus $D(R) = \{ A, B \}$. Any distinct-item scheme that corresponds to simple random sampling of size 1 must produce the following sampling design:

$$\Pr[S = \{ A \}] = 1/2$$
$$\Pr[S = \{ B \}] = 1/2.$$

A *uniform* distinct-item sampling scheme corresponds to a set sampling scheme that is uniform. In such a scheme, each distinct item is sampled with equal probability.

3.3.3 Supported Transactions and Maintenance Costs

Probably the most important aspect of a maintenance scheme is which types of transactions it supports. A maintenance scheme *supports* insertions, deletions, or updates if it can handle all transactions of the respective type. As will become evident later on, not every scheme supports all types of transactions. In some applications, datasets are append-only sets and it suffices that the scheme supports insertion transactions. In other applications, datasets are subject to each of the three transaction types and more sophisticated maintenance schemes are required.

If a specific maintenance scheme supports the types of transactions required by the application, it is important to assess the costs associated with processing the transactions. In fact, it may be the case that a scheme that supports only insertions is faster at processing insertions than a scheme that supports all three transaction types. Maintenance costs include CPU costs, I/O costs for accessing the sample, and, if existent, I/O costs for accessing the underlying dataset R. Another cost is that of space consumption, which is discussed in detail in section 3.3.5.

We argue below that, if a scheme requires access to R, the I/O cost of these accesses often by far outweigh the other costs. For this reason, we classify sampling schemes into separate categories depending on whether or not and how often they access base data.

A. Cost of Base Data Accesses

Schemes that access the base data have a significant overhead, often several orders of magnitude higher than the CPU cost or the I/O cost of accessing the sample. This may have several reasons:

1. Dataset R is much larger than its sample and usually resides on hard disk. A scan of even a small fraction of R may therefore lead to significant I/O costs. In contrast, the sample is often stored in main memory (or resides in the cache) so that sample accesses incur no or little I/O costs.

2. Dataset R is accessed in random fashion. In fact, access to R is often required to extract items chosen uniformly and at random. As discussed in section 2.2.2 on query sampling, the extraction of random items can be expensive, even if only a few items are required.

3. Dataset R is stored at a remote location. In this case, accesses to R may lead to significant communication costs. Additionally, system resources at the remote location are utilized.

4. Dataset R is not materialized. For example, one might sample from the join of two tables $R_1 \bowtie R_2$ without materializing the join result. Access to R then triggers the execution of a join or parts thereof.

In some cases, schemes that access R are not applicable at all:

5. Dataset R is not accessible. When sampling from a data stream window, for instance, the content of the window itself is discarded due to resource limitations in the data stream system. Instead, the sample is used to represent the items in the window.

From the discussion above, it becomes evident that access to dataset R should be avoided, if possible.

B. Incremental Maintenance

We now classify maintenance schemes with respect to the type of access to R they require. For each type of transaction T, we distinguish three classes.

- A maintenance scheme is *non-incremental* with respect to T if for all feasible sequences γ and for all transactions γ_i of type T, the computation of S_i is solely based on R_i (or R_{i-1} and γ_i). Non-incremental schemes do not reuse the current sample, they recompute it from scratch.

- A maintenance scheme is *incremental* with respect to T if for all feasible sequences γ and all transactions γ_i of type T, the scheme computes S_i without ever accessing the dataset R_i. Often, the computation is based on S_{i-1} and γ_i. Sometimes, auxiliary data structures that enable strongly incremental maintenance are stored and maintained with the sample. We view these structures as an intrinsic part of the maintenance scheme.[2]

- A maintenance scheme is *semi-incremental* with respect to T if it is neither non-incremental nor incremental. Semi-incremental schemes may access both S_{i-1} and R_i. Note that a scheme is semi-incremental even if it accesses R_i infrequently.

[2]This is different to the corresponding concept of self-maintainability of a materialized view, which applies to the view only, without additional data structures.

From the viewpoint of efficiency, incremental schemes are preferable over semi-incremental schemes, which in turn are preferable over non-incremental schemes.

A maintenance scheme that supports insertions automatically supports (at least) non-incremental maintenance under updates and deletions. If an update or deletion occurs, the scheme would discard the current sample and recompute it from scratch by reinserting all items of the modified dataset. Similarly, a scheme that is incremental under both insertions and deletions is also incremental under updates. This is because updates can be expressed as a deletion followed by an insertion. In general, schemes that support updates directly, that is, schemes that do not fall back to deletion and reinsertion, are more efficient.

C. Implicit Deletions

Thus far, we have assumed that the sequence γ is complete, that is, γ explicitly contains all the transactions that modify R. In some applications, however, deletions are not given explicitly. Instead, a "deletion predicate" is used to determine whether or not a given item has been deleted. Deletions are *implicit* in the sense that they are implied by the deletion predicate. Only deletions of items that are stored in the sample at the time of their deletion are visible to the maintenance scheme; the scheme is oblivious to deletions of items that are not stored in the sample. A maintenance scheme that supports deletion transactions does not necessarily support implicit deletions.

We do not further investigate implicit deletions in general. Instead, we will consider only a special case of implicit deletions that occurs when sampling from a time-based sliding window of a data stream. In this setting, items that arrive in the stream are valid for a certain amount of time; they expire afterwards. Implicit deletions occur when an item expires because only expirations of items stored in the sample are visible to the sampling scheme.[3]

3.3.4 Sample Size

An important aspect of a maintenance scheme is the sample size or the distribution of the sample size produced by the scheme. The sample size is defined as the number of items stored in the sample, including duplicates. The problem of finding the optimum sample size for a given application has been studied extensively in statistical literature, see Cochran (1977) for an overview. The essence is that, on the one hand, larger samples contain more information about the data than smaller samples do; thus, they lead to more precise estimates. On the other hand, one does not want the sample to become too large because both the cost of sampling and the cost of obtaining the estimate from the sample increase with the sample size.

[3] This does not hold for sequence-based windows. In such a window, arrivals and expirations are synchronized, that is, an expiration occurs if and only if a new item arrives in the stream. The sampling scheme is not oblivious to deletions. In fact, this is the reason why sequence-based samples are easier to maintain.

The requirements on the sample size depend on both the application and properties of the dataset. To facilitate analysis of the algorithms, 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, which accumulate historical data.

In this section, we discuss classes of maintenance schemes with respect to sample-size variability and sample-size bounds. We also discuss their suitability for stable and growing datasets. Since a maintenance scheme may behave differently for different transaction sequences, denote by Γ an arbitrary but fixed set of transaction sequences. For example, Γ might correspond to the set of all sequences over domain $\mathscr{R}$, to the set of insertion-only sequences, or to the set of all sequences in which the population size never exceeds a certain quantity. All subsequent properties are with respect to such a set Γ, but we omit any reference to Γ for brevity. Later, when we state that a sampling scheme exhibits a certain property, the set Γ must be defined if it not clear from the context.

A. Variability

A *fixed-size* maintenance scheme produces samples of constant size. For with-replacement sampling, we have

$$|S_i| = M$$

for some arbitrary but fixed sample size $M > 0$. For without-replacement sampling, the sample size cannot be larger than the size of the dataset and

$$|S_i| = \min\{ M, N_i \}.$$

Fixed-size samples have the advantage that the cost of obtaining an estimate from the sample is bounded from above for many estimation problems. For example, if the population sum of an attribute is estimated using a Horvitz-Thompson estimator, estimation requires summing up at most M scaled values. Fixed-size samples are ideal for stable datasets because the sampling fraction remains roughly constant. For growing datasets, however, keeping the sample size fixed for all times 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.

In a *semi-variable-size* maintenance scheme, the sample size changes as new transactions are processed, but each individual sample design has a constant sample size. We have

$$|S_i| = M_i$$

for a sequence of arbitrary but fixed sample sizes $M_1, M_2, \ldots$ with $M_i \geq 0$. The M_i may depend on γ. Compared to fixed-size sampling, semi-variable-size samples have the advantage that the sample size can be adjusted dynamically. For example, the sample size may be increased whenever the sampling fraction drops below a lower bound. Such an approach retains the advantages of fixed-size sampling, but avoids its disadvantage for growing datasets. Thus, techniques for sample resizing—either upwards or downwards—are of interest in some applications.

Finally, a *variable-size* maintenance scheme produces samples of a probabilistic size, that is, the variability of the sample size results from the sample design. The sample size is a random variable. Its distribution typically depends on γ and changes over time. Although a variable sample size is disadvantageous, variable-size schemes often have a lower maintenance cost than fixed-size or semi-variable-size schemes. This cost reduction may more than make up for the variability of the sample size, especially if the variability is low. For this reason, variable-size schemes play a very important role in sample maintenance.

B. Bounds

When the sample size is not fixed, it is of practical interest whether or not the sample size can be bounded. Lower bounds ensure that the sample contains at least a certain amount of information about the dataset, while upper bounds are effective to control the cost of sample maintenance and estimation. In fact, upper bounds on the sample size are especially useful in applications that need to guarantee a hard upper bound on the response time of queries over the sample.

A maintenance scheme is *bounded in size* from below if there exists a constant $L > 0$ such that for all i

$$|S_i| \geq \min(|R_i|, L).$$

A scheme is bounded in size from above if there exists a constant M such that for all i

$$|S_i| \leq M.$$

A maintenance scheme is *unbounded in size* from below/above if such a bound does not exist. Using similar arguments as above, bounded-size sampling schemes are the method of choice for stable datasets. In general, unbounded sampling schemes are needed for growing datasets. Note that unbounded-size sampling and fixed-size sampling are mutually exclusive, but all other combinations of variability and bounds are feasible; see table 3.1

The bounds as discussed above are both global (valid at all times) and strict (never exceeded). In the course of this thesis, we will also consider *local bounds*, which are valid under certain conditions only, and *probabilistic bounds*, which are valid with a certain probability. If global and strict bounds do not exist or are very broad, local and/or probabilistic bounds can help understand further the behavior of a maintenance scheme.

Table 3.1: Feasible combinations of sample size and sample footprint

		Size			Footprint	
		fixed	semi-var.	variable	bounded	unbounded
Size	bounded	✓	✓	✓	✓	✓
	unbounded	-	✓	✓	-	✓

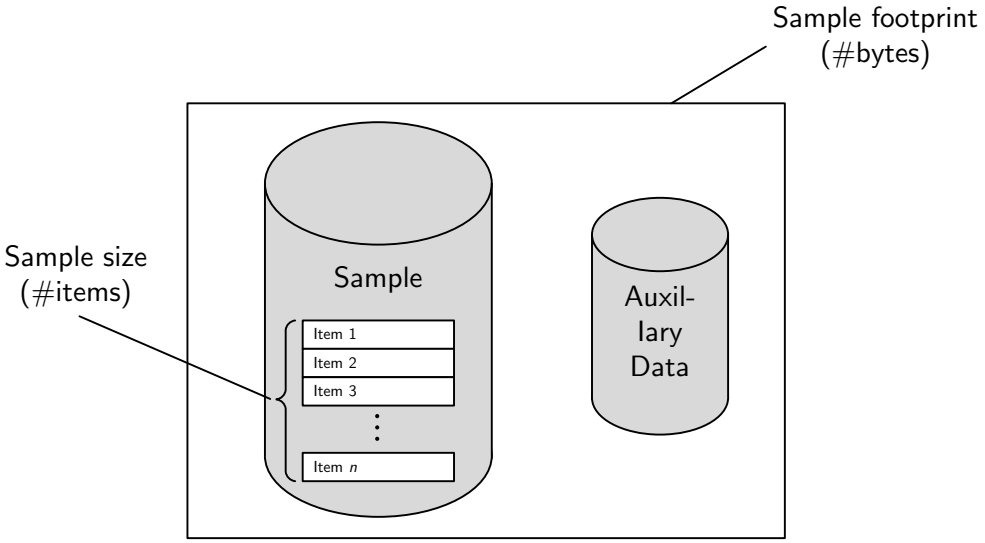


Figure 3.2: Sample size and sample footprint

3.3.5 Sample Footprint

The *footprint* of a sample is the space consumption of the sample in bytes. If auxiliary data structures are kept to enable sample maintenance, the space consumption of these structures is included in the footprint. The sample footprint is related to the sample size, but has a different focus:

1. The sample size is measured in number of items, the footprint uses bytes.
2. The sample size does not include any auxiliary data, the footprint does.
3. The sample size is a measure of how much information is contained in the sample, the footprint is a measure of how much space is required to store the sample.

Both quantities are illustrated in figure 3.2.

We distinguish maintenance schemes with *fixed footprint*, *semi-variable footprint* and *variable footprint*. The definitions correspond to the respective definitions for sample-size variability. Similarly, the footprint of a maintenance scheme can be bounded or unbounded. Ensuring that the sample footprint remains bounded from

above at all times can be useful from a system-design point of view. Specifically, a-priori bounds for the sample footprint simplify 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 Brown and Haas (2006) or in a data stream management system.

The sample size and sample footprint of a maintenance scheme do not necessarily have the same class of variability or bounds. For example, there are fixed-size maintenance schemes that have a variable footprint, and there are bounded-size sampling schemes that have an unbounded footprint. Unbounded-size sampling and bounded-footprint sampling are mutually exclusive, all other combinations are feasible; see table 3.1.

3.3.6 Summary

We discussed several properties that a maintenance scheme can have. These properties range from sampling semantics and properties of the sampling designs—such as uniformity or sample-size distribution—over the types of transactions that a scheme supports, as well as the cost associated with these transactions, to the amount of space required to store the sample with all its auxiliary data structures. They allow us to concisely summarize the key characteristics of a maintenance scheme, and we will make use of them in our subsequent discussion.

3.4 Schemes for Survey Sampling

In traditional survey sampling, two different types of sampling schemes are distinguished (Särndal et al. 1991). *Draw-sequential* schemes compute the sample draw-by-draw. In each draw, a new sample item is selected from either the entire population or a subset of it. The selection is randomized and the selection probabilities may change after each draw. As a consequence, the population is accessed in random order. In contrast, *list-sequential* schemes scan the population sequentially and decide for each element whether or not it is selected; this decision is randomized. Again, the selection probabilities are often not fixed a priori but depend on the outcome of earlier selections.

The notion of list-sequential sample selection is similar but not equal to our notion of incremental sample maintenance (with respect to insertions). The difference is that the survey-sampling schemes typically require some knowledge about the population before the selection is carried out. Such knowledge may include the size of the population or the value of an auxiliary variable associated with the population items. Furthermore, some list-sequential schemes reorder the population before sample selection is carried out. Incremental schemes are stronger in that they do not require a-priori knowledge or reordering. In fact, every incremental scheme is also list-sequential, while the opposite does not hold.

3.4.1 Draw-Sequential Schemes

Suppose that the items in the population are stored in an indexed list and denote by $1, \ldots, N$ the set of indexes. The simplest draw-sequential scheme uniformly selects a random integer between 1 and N in each draw and adds the item having the selected index to the sample. The process is repeated—with draws being independent—until a sample of the desired size has been computed. The method is a special case of the sampling scheme of Muller (1958). Since each index can be selected more than once, the method leads to simple random sampling with replacement. The number of distinct items in the sample is distributed as given in equation (2.6) on page 11. Variations of this simple draw-sequential scheme, in which the random selection of an item is replaced by more sophisticated procedures, were already discussed in the context of query sampling; see section 2.2.2.

To obtain a simple random sample of size M without replacement, we may repeat the drawing of items until M distinct items have been selected. Items that are already present in the sample are discarded. Denote by B_n a random variable for the number of sampling steps required to obtain the n-th distinct item, $1 \leq n \leq M$. Clearly, we have $B_1 = 1$ because the first drawn item is always distinct. Since each of the subsequent items is accepted into the sample only if it has not been sampled already, B_n is geometrically distributed with

$$\begin{aligned} \Pr[\, B_n = k \,] &= \Pr[\, k-1 \text{ duplicates followed by a new item} \,] \\ &= \left(\frac{n-1}{N}\right)^{k-1} \frac{N-n+1}{N} \end{aligned}$$

for $k \geq 1$. The B_n are independent. The total number of repetitions is given by $B = \sum B_n$. Using the linearity of the expected value, standard properties of the geometric distribution (Johnson et al. 1992, pp. 201–207) and a change in the summation index, we find that

$$\mathrm{E}[\, B \,] = \sum_{n=1}^{M} \mathrm{E}[\, B_n \,] = 1 + \sum_{n=2}^{M} \frac{N}{N-n+1} = 1 + N(H_{N-1} - H_{N-M}). \tag{3.3}$$

Here, the quantity

$$H_n \stackrel{\text{def}}{=} \sum_{i=1}^{n} \frac{1}{i}$$

denotes the n-th *harmonic number*. A well-known approximation for H_n is given by

$$H_n = \ln n + \gamma + \epsilon, \quad \epsilon \leq \frac{1}{2n}, \tag{3.4}$$

where $\gamma = 0.5772\ldots$ denotes Euler's constant (Knuth 1997). For brevity, we define the *partial harmonic number*

$$H_{n,m} = \sum_{i=n}^{m} \frac{1}{i} = H_m - H_{n-1}.$$

By (3.4), we find that

$$H_{n,m} \approx \ln\left(\frac{m}{n-1}\right) \approx \ln\left(\frac{m+1}{n-1}\right) \tag{3.5}$$

with negligible error whenever $m, n \gg 1$. Thus, for large N,

$$\mathrm{E}[B] = NH_{N-M+1,N-1} \approx N \ln\left(\frac{N}{N-M}\right).$$

For example, to sample 1,000 distinct items out of 10,000 items, approximately 1,053 sampling steps are required in expectation. Compared to with-replacement sampling, the overhead in the number of draws is negligible when $N \gg M$, but the repeated sample lookups can be expensive. In fact, if the sample lookup is performed by a sequential scan of the sample, the algorithm requires $\Omega(M^2)$ total time.

Many draw-sequential schemes try to improve upon the above algorithm by avoiding any repetitions or sample lookups. A scheme suggested by Goodman and Hedetniemi (1977)[4] maintains a list of unselected indexes initialized to $1, \ldots, N$. In each draw, a random index is selected and removed from the list, and the population item having the selected index is added to the sample. The scheme has the obvious drawback of requiring $\Omega(N)$ space. Ernvall and Nevalainen (1982) and Teuhola and Nevalainen (1982) improve upon the algorithm by storing only the difference between the set of all indexes and the set of unselected indexes. The algorithms run in $O(M)$ expected time and $O(M^2)$ worst-case time. They require $O(M)$ space.

As we will see below, draw-sequential algorithms are superseded by carefully implemented list-sequential algorithms.

3.4.2 List-Sequential Schemes

As above, we describe schemes that select M integers from the sequence of indexes $1, ..., N$. In this section, we only describe list-sequential schemes that are not incremental. When applicable, these schemes are often superior to incremental schemes in terms of efficiency because they exploit knowledge of the population size.

The standard list-sequential algorithm has been proposed by Fan et al. (1962, Method 1) and Jones (1962). It was rediscovered by Bebbington (1975). The idea is to process the indexes sequentially. Each index is accepted with probability

$$\frac{M-n}{N-i+1} = \frac{\#\text{ remaining samples}}{\#\text{ remaining indexes}},$$

where n is the number of already accepted indexes and i is the current index. Whenever an index is accepted, the respective population item is retrieved and added to the sample. The process stops as soon as M items have been selected. The algorithm accesses the population sequentially, and the original order of the population items is retained in the sample. As proven by Fan et al. (1962), the

[4] As cited by Ernvall and Nevalainen (1982).

method produces a simple random sample without replacement. The number of sampling steps lies in the range from M to N and averages to $\frac{M}{M+1}(N+1)$ (Ahrens and Dieter 1985).

An example with $M = 2$ and $N = 4$ is shown in figure 3.3a. The figure displays a probability tree that contains all possible states of the sample along with the probability of state transitions. The probability of reaching a node can be obtained by multiplying the transition probabilities along the path from the root of the tree to the respective node. Each sample is selected with probability $1/6$; the average number of sampling steps is $10/3 \approx 3.33$.

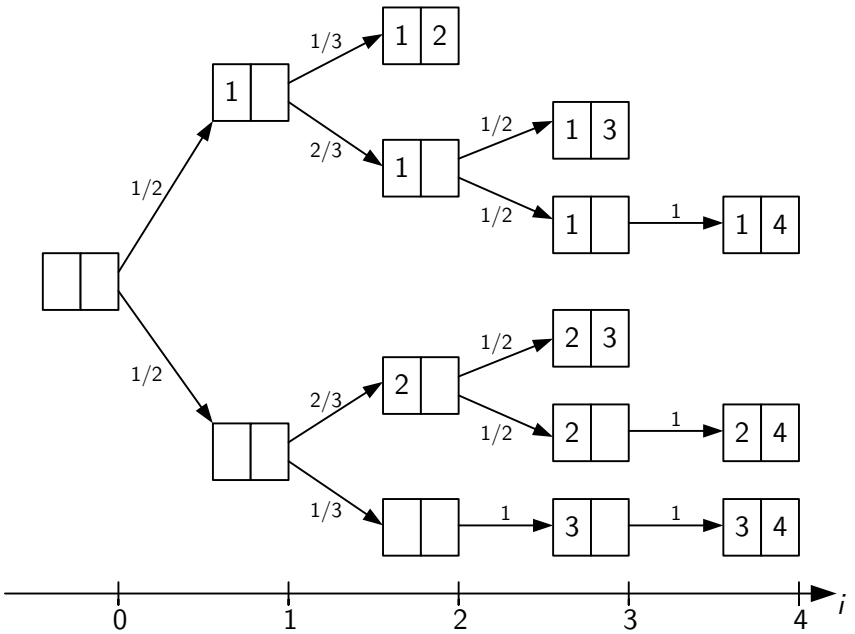
The large number of sampling steps has given rise to the development of more efficient schemes. The general idea of the improved schemes is to compute a "skip counter" that indicates the number of items to skip before the next item is processed. Skipped items do not incur any cost, and the computation of the skip counters can be done very efficiently. Suppose that we are about to process the i-th index and that n items have already been selected into the sample. Index i is then accepted with probability $(M-n)/(N-i+1)$. If index i is rejected, index $i+1$ is accepted with probability $(M-n)/(N-i)$ and rejected otherwise. In the latter case, index $i+2$ is accepted with probability $(M-n)/(N-i-1)$ and so on. Denote by $Z_{i,n}$ a random variable for the number of excluded items before the next sample inclusion. We have

$$\Pr[\, Z_{i,n} = z \,] = \frac{M-n}{N-i+1-z} \prod_{z'=0}^{z-1} \left(1 - \frac{M-n}{N-i+1-z'}\right) \tag{3.6}$$

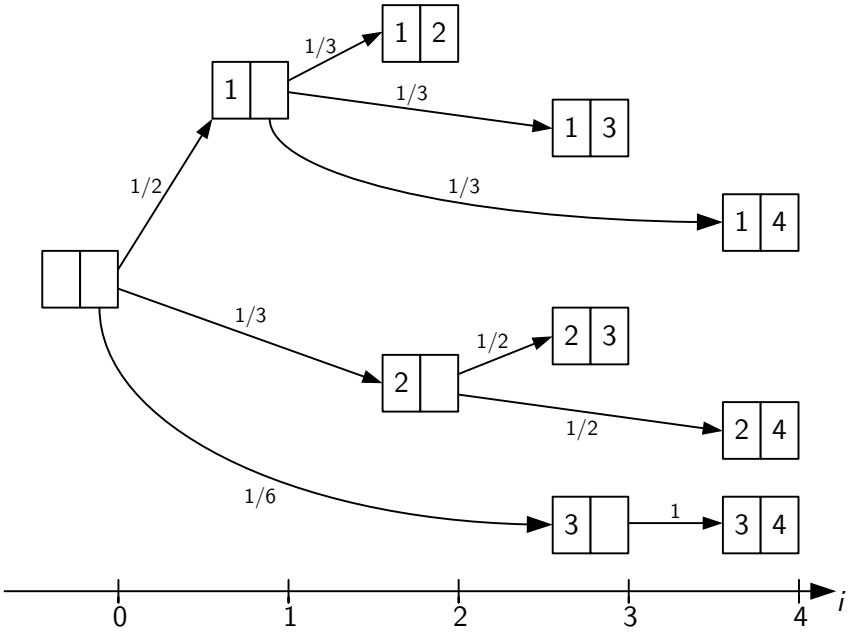
for $z \geq 0$, where we take an empty product as 1. We can now state the skip-based algorithm. The algorithm first generates a realization z_1 of $Z_{1,0}$ and includes the item with index $i_1 = z_1 + 1$ into the sample. Thus, the first z_1 indexes are skipped. Next, the algorithm generates a realization z_2 of $Z_{i_1+1,1}$ and includes item $i_2 = i_1 + z_2 + 1$ into the sample. The process is repeated until M items have been selected. The entire algorithm is illustrated in figure 3.3b; the number of sampling steps is precisely $M = 2$. The efficiency of the skip-based algorithm depends on how efficiently the realizations of $Z_{i,n}$ can be generated. A detailed overview of such methods is given in Devroye (1986, ch. XII.3); we summarize the main results below.

In Devroye (1986) and Bissell (1986), algorithms that generate a realization of $Z_{i,n}$ using the inversion method are proposed; see Devroye (1986, ch. II.2) for a general discussion of this method. The idea is to generate a uniform random number U on the interval $[0, 1]$. The desired value of the skip counter is then given by the smallest integer $z \geq 0$ that satisfies $U < \Pr[\, Z_{i,n} \leq z \,]$. The value of z is found by a sequential search, starting with $z = 0$. Devroye (1986) also gives a more intelligent method that avoids starting from $z = 0$ by computing a lower bound on z. In any case, the inversion method reduces the number of required random numbers to M, but the computation of the $Z_{i,n}$ is still expensive.

A more efficient acceptance-rejection (AR) algorithm has been proposed by Vitter (1984); see Devroye (1986, chapter II.3) for a general discussion of such algorithms. The basic idea is to find another random variable Z^* that is easy and fast to generate.



(a) Standard algorithm



(b) Skip-based algorithm

Figure 3.3: Illustration of list-sequential sampling

The AR algorithm starts by generating a realization z^* of Z^*. With probability $p(z^*)$, this value is "accepted" and the algorithm returns with $Z = z^*$; with probability $1 - p(z^*)$, the value is "rejected" and the process repeated. The random variable Z^* 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 Z, and (ii) the expected number of rejections before the final acceptance is small. To compute the M skip counters required to obtain the size-M sample, Vitter's AR method requires $O(M)$ expected time; a significant improvement over the standard method. An optimized version of the algorithm is given in Vitter (1987) and Nair (1990).

Another direction is taken by Ahrens and Dieter (1985), who make use of a Bernoulli sampling scheme that is also based on skip counters. Their method exploits the fact that skip counters for Bernoulli sampling have a simpler distribution and are thus easier to compute than for sequential sampling (section 3.4.3B). Therefore, the idea is to generate a Bernoulli sample that is slightly larger than M with sufficiently high probability. The sample size is then reduced using, for example, the standard method described above. Finally, the items corresponding to the indexes in the reduced sample are retrieved from disk. The method is slightly faster than Vitter's method (Vitter 1987), but it has the disadvantage that index computation and population access cannot be interweaved. It also requires $O(M)$ space.

3.4.3 Incremental Schemes

We now discuss list-sequential schemes that are also incremental. These schemes treat the population as a sequence of insertion transactions into an initially empty dataset. Thus, the population is processed sequentially but—in contrast to the schemes discussed above—incremental schemes are able to output at any time a uniform random sample of that part of the population that has been processed already. This is done by either maintaining the sample directly or by maintaining a data structure from which the sample can be extracted.

The main motivation of incremental schemes in the context of survey sampling is that, in some applications, the population size is unknown prior to sampling. Of course, the population size can be determined in a first scan, and non-incremental schemes can be used to compute the sample in a second scan. However, a direct, single-scan approach is often more efficient.

A summary of the schemes discussed in this section is given in the upper part of table 3.2, page 74. The table is structured according to the properties developed in section 3.3. A check mark (✓) stands for incremental, a circle (○) is for semi-incremental, and a dash (-) means non-incremental or unsupported.

Making use of the notation of section 3.2, we describe the schemes in terms of processing a sequence γ that consists of insertions only. To simplify our discussion, we assume that $\gamma_i = +r_i$ so that $R_i = \{ r_1, r_2, \ldots, r_i \}$. Both the dataset and the sample are initially empty, $S_0 = R_0 = \emptyset$.

A. Bernoulli Sampling, BERN(q)

Perhaps the simplest incremental scheme is the Bernoulli sampling scheme. In Bernoulli sampling with sampling rate q, denoted BERN(q), each arriving item is included into the sample with probability q and excluded with probability $1 - q$, independent of the other items. We have

$$S_{i+1} \leftarrow \begin{cases} S_i \cup \{ r_{i+1} \} & \text{with probability } q \\ S_i & \text{with probability } 1 - q. \end{cases} \tag{3.7}$$

The Bernoulli sampling scheme leads to the Bernoulli sampling design; see section 2.1.2 for a discussion of the properties of this design. When the intended meaning becomes apparent from the context, we will write Bernoulli sampling to denote either the Bernoulli design or a scheme that leads to this design. The Bernoulli scheme as described above has been proposed by Fan et al. (1962, Method 3). Replacing $\cup$ by $\uplus$ in (3.7), it can also be used when both S and R are multisets because the sampling decisions do not depend on the actual values of the processed items.

B. Bernoulli Sampling With Skipping

As with list-sequential sampling, Bernoulli sampling can be implemented more efficiently by maintaining a skip counter in addition to the sample. As pointed out by Fan et al. (1962, Method 9) and Ahrens and Dieter (1985), the skip counter Z follows a geometric distribution with

$$\Pr[Z = z] = (1 - q)^z q \tag{3.8}$$

for $z \geq 0$. Using the inversion method of Devroye (1986, ch. X.2), a realization of Z can be computed in constant time by setting

$$Z = \left\lfloor \frac{\log U}{\log(1 - q)} \right\rfloor$$

with U being a uniform random variable in $(0, 1)$. The expected number of skipped transactions is $\mathrm{E}[Z] = (1 - q)/q$, which is $O(q^{-1})$ so that savings can be substantial if q is small. Given a sequence $z_0, z_1, \ldots$ of independent realizations of Z according to (3.8), the complete algorithm is

$$(S_{i+1}, Z_{i+1}) \leftarrow \begin{cases} (S_i \cup \{ r_{i+1} \}, z_{i+1}) & Z_i = 0 \\ (S_i, Z_i - 1) & Z_i > 0, \end{cases} \tag{3.9}$$

starting with $Z_0 = z_0$. The z_i encode the generation of skip counters; only the z_i that are actually used have to be generated.

C. Reservoir Sampling, RS(M)

The reservoir sampling scheme, denoted RS(M), maintains a uniform sample of fixed size M, given a sequence of insertions. The procedure inserts the first M items directly into the sample. For each successive transaction $\gamma_i = +r_i$, $i > M$, the item is inserted into the sample with probability M/N_i, where $N_i = N_{i-1} + 1 = i$ is the size of the dataset just after the insertion. An included item replaces a randomly selected item in the sample. The complete algorithms is as follows

$$S_{i+1} \leftarrow \begin{cases} S_i \cup \{ r_{i+1} \} & N_i + 1 \leq M \\ \textsc{Replace}(S_i, r_{i+1}) & \text{with probability } M/(N_i + 1) \\ S_i & \text{with probability } (N_i - M + 1)/(N_i + 1), \end{cases} \tag{3.10}$$

where the function $\textsc{Replace}(S, r)$ replaces a random item in S by r and returns the result. The algorithm requires knowledge of N_i, which thus has to be maintained with the sample. A slightly different version of reservoir sampling appeared in Knuth (1981). According to Knuth, this method is due to Alan G. Waterman. The version given here was first proposed by McLeod and Bellhouse (1983); their paper also contains a proof of uniformity.

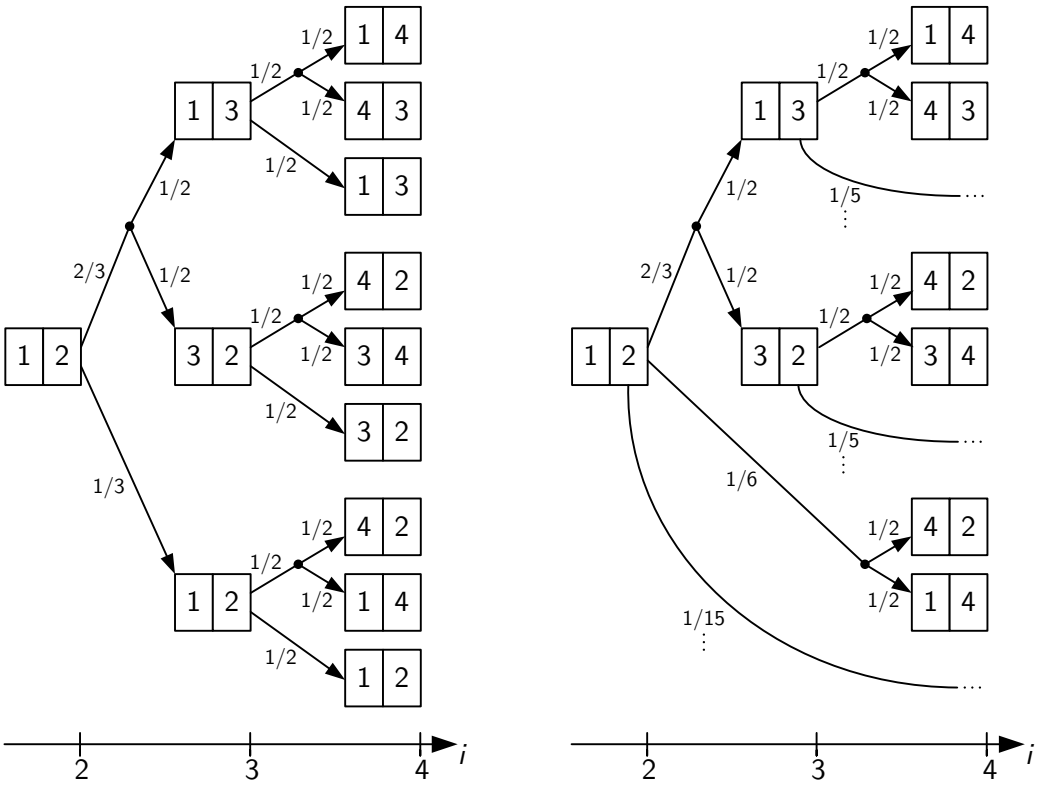
An illustration of the algorithm for $M = 2$ is given in figure 3.4a. The first two steps have been omitted; the sample $\{ 1, 2 \}$ is output with probability 1 after these two steps. The upper children represent acceptances, the lower child represents a rejection. In case of acceptance, another random decision is carried out to determine at which position the new item is written. After 4 items have been processed, each distinct sample is selected with probability 1/6; there are multiple paths to some of the sample states.

Ignoring the first M steps required to initially fill the sample, the algorithm performs $MH_{M+1,N} \approx M \ln(N/M)$ item replacements in expectation to process $N > M$ items (Knuth 1981). In the example, the expected number of replacements up to $N = 4$ is $0.58\overline{3}$.

D. Reservoir Sampling With Skipping

In his well-known paper, Vitter (1985) developed an acceptance-rejection algorithm that generates skip counters for reservoir sampling in constant time. The problem is also discussed in Pinkham (1987) and Li (1994). As before, right after item r_i has been processed, denote by Z_i a random variable for the number of rejected items until the next sample insertion. Item r_{i+1} is accepted with probability $M/(N_i + 1)$ or rejected otherwise. In case of rejection, item r_{i+2} is accepted with probability $M/(N_i + 2)$. If r_{i+2} is rejected as well, item r_{i+3} is selected with probability $M/(N_i + 3)$ and so on. The distribution of Z_i is therefore

$$\Pr[\, Z_i = z \,] = \frac{M}{N_i + z + 1} \prod_{z'=0}^{z-1} \left(1 - \frac{M}{N_i + z' + 1} \right) \tag{3.11}$$



(a) Standard algorithm **(b)** Skip-based algorithm

Figure 3.4: Illustration of reservoir sampling

for $z \geq 0$. Given a sequence $z_M, z_{M+1}, \ldots$ of independent realizations of random vairables $Z_M, Z_{M+1}, \ldots$, respectively, the skip-based algorithm is given by

$$(S_{i+1}, Z_{i+1}) \leftarrow \begin{cases} (S_i \cup \{\, r_{i+1} \,\}, z_M) & N_i + 1 \leq M \\ (\textsc{Replace}(S_i,\, r_{i+1}), z_{i+1}) & Z_i = 0 \\ (S_i, Z_i - 1) & Z_i > 0. \end{cases} \tag{3.12}$$

Again, only the z_i that are actually used have to be generated. The algorithm is illustrated in figure 3.4b. There are infinitely many possible values for each skip counter; skip counter values that refer to item r_6 and beyond are omitted (as indicated by vertical ellipsis points).

E. Min-Wise Sampling, MIN(M)

Min-wise sampling (Fan et al. 1962, Method 4) is a precursor to reservoir sampling.[5] The algorithm makes use of a sequence $U = (U_1, U_2, \ldots)$ of independent and identically-distributed (i.i.d.) random variables, typically uniform. The U_i act as random tags in the sequence of arriving items $r_1, r_2, \ldots$; item r_i is given tag U_i. In min-wise sampling of size M, denoted MIN(M), the sample consists of the items with the M lowest tags processed thus far. To enable maintenance, it is necessary to store the tags of the items that have been included into the sample. The algorithm is

$$S_{i+1} \leftarrow \begin{cases} (S_i \cup \{\, (r_{i+1}, U_{i+1}) \,\} & |S_i| < M \\ S_i \setminus \{\, \operatorname*{argmax}_{(r,u) \in S_i} u \,\} \cup \{\, (r_{i+1}, U_{i+1}) \,\} & |S_i| = M,\ U_{i+1} < \max\limits_{(r,u) \in S_i} u \\ S_i & \text{otherwise.} \end{cases}$$

Uniformity follows by symmetry; every size-M subset of R has the same chance of having the smallest tags. Compared to reservoir sampling, min-wise sampling has the disadvantage that its sample footprint is larger because additional space is required to store the tags. It also has higher CPU cost. In a naive implementation of a sampling step, the sample is scanned sequentially for the item with the largest tag, which gives $O(M)$ time per transaction. By keeping track of the largest item, the cost of *rejected* transactions can be reduced to $O(1)$. An even more economic approach is to store the sample in a heap; the cost for *accepted* items then reduces to $O(\log M)$, which is still more than the $O(1)$ of reservoir sampling. Min-wise sampling has the advantage, however, that the number of items in the base data does not have to be known when executing a sampling step. As we will see later, this property is important for scenarios such as distinct-item sampling or sliding-window sampling. Min-wise sampling can be implemented using geometric skips (Devroye 1986; Li 1994), but it then loses its unique advantages in the aforementioned scenarios.

[5] In fact, min-wise sampling was known under the name reservoir sampling (Knuth 1969; Devroye 1986). To avoid confusion, we do not follow this nomenclature.

Table 3.2: Uniform maintenance schemes for set/multiset sampling

		Set	Multiset	Insert	Update	Delete	Size	Footprint	Skip
Survey sampling	BERN(q)	✓	✓	✓	-	-	Unbounded	Unbounded	✓
	RS(M)	✓	✓	✓	-	-	Fixed	Fixed[a]	✓
	MIN(M)	✓	✓	✓	-	-	Fixed	Fixed[a]	✓
Database sampling	MBERN(q)	✓	-	✓	✓	✓	Unbounded	Unbounded	✓
	MBERNP(M)	✓	-	✓	✓	-	Bounded	Bounded[a]	✓
	MRS(M)	✓	-	✓	✓	-	Fixed	Fixed[a]	✓
	MRSR(L, M)	✓	-	✓	✓	○	Bounded	Bounded[a]	(✓)
	MRST(L, M)	✓	-	✓	✓	○[b]	Bounded	Fixed[a]	✓
	CAR(M)	✓	-	✓	✓	○	Bounded[c]	Fixed[a]	(✓)
	CARWOR(M)	✓	-	✓	✓	○	Fixed	Fixed[a]	(✓)
	MBERNM(q)	✓	✓	✓	○	○	Unbounded	Unbounded	✓
Novel schemes	RP(M)	✓	-	✓	✓	✓	Bounded	Bounded[a]	✓
	ABERN(q)	✓	✓	✓	✓	✓	Unbounded	Unbounded	-

[a] Assuming constant space for storing an item / a counter / a label / the sampling rate.

[b] ✓ when deletions occur infrequently.

[c] After conversion to a without-replacement sample.

3.5 Schemes For Database Sampling

The main difference between database sampling schemes and survey sampling schemes is that, in addition to insertion transactions, the former also consider update and/or deletion transactions applied on the dataset. These additional types of transactions significantly increase the complexity of sample maintenance. Nevertheless, many of the schemes we are going to review here build upon the incremental survey sampling schemes, often in a non-trivial way.

We assume throughout that the sample is small enough to fit in main memory so that random accesses to the sample are cheap. This assumption limits the applicability of database sampling in warehousing scenarios where the dataset size can be so large, and the sampling rate so high, that the samples must be stored on disk or flash drives. Extending database sampling schemes to large disk-based samples is a topic for future research; see chapter 8. 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 discuss set sampling schemes, multiset sampling schemes, distinct-item sampling schemes and data stream sampling schemes, each class in its own section. Similarly, chapters 4–7 are each devoted to one of these classes. A short summary and discussion of the contribution of this thesis is given at the end of each section. An overview of the schemes is given in tables 3.2, 3.4, and 3.7. Detailed information about our *naming conventions* as well as a list of all algorithms can be found in the *index of algorithm names* on page 269.

As before, we assume throughout this thesis that $S_0 = R_0 = \emptyset$. Also denote by $N_i = |R_i|$ the size of the dataset and by $n_i = |S_i|$ the (random) size of the sample after the i-th transaction. Clearly, $n_0 = N_0 = 0$. The dataset size, including potential duplicates, can be maintained incrementally. If transaction γ_{i+1} corresponds to an insertion, then $N_{i+1} = N_i + 1$. If it corresponds to a deletion, then $N_{i+1} = N_i - 1$. If γ_{i+1} is an update transaction, the dataset size remains unchanged and $N_{i+1} = N_i$.

3.5.1 Set Sampling

We start with a discussion of available set sampling schemes. We consider two extensions of Bernoulli sampling (one unbounded, one bounded), three extensions of reservoir sampling (two of them supporting deletions), and two adapted versions of the correlated acceptance/rejection scheme of Olken and Rotem (1992).

A. Modified Bernoulli Sampling, MBERN(q)

It is straightforward to modify Bernoulli sampling on sets to handle updates and deletions; the modified version is denoted as MBERN(q). The algorithm remains unchanged for insertions; each inserted item is accepted with probability q and

rejected with probability $1 - q$; see equations (3.7) and (3.9). If transaction γ_{i+1} corresponds to an update of the form $r \to r'$, set

$$S_{i+1} \leftarrow \begin{cases} S_i \setminus \{ r \} \cup r' & r \in S_i \\ S_i & \text{otherwise,} \end{cases}$$

that is, simply update the sample copy of the item if present. The output of the sampling process is the same as if r were replaced by r' right from the beginning. If transaction γ_{i+1} corresponds to a deletion $-r$, set

$$S_{i+1} \leftarrow \begin{cases} S_i \setminus \{ r \} & r \in S_i \\ S_i & \text{otherwise,} \end{cases}$$

that is, remove the deleted item if present in the sample. Thus the deletion operation "annihilates" item r; it is as if item r were never inserted into R. It is possible to run MBERN(q) using skip counters to process insertion transactions; the value of the skip counter remains unmodified for both update and deletion transactions.

B. Modified Bernoulli Sampling With Purging, MBERNP(M)

The MBERN(q) scheme described above is unbounded in both size and space. To bound the sample size, the scheme can be combined with the "counting sample" technique proposed by Gibbons and Matias (1998); the combined scheme is denoted MBERNP(M). 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 MBERNP(M) scheme starts with $q = 1$. Each purge operation consists of one or more subsampling steps,[6] and the sampling rate q is decreased at each such step. Specifically, the sample is subsampled using a BERN(q'/q) scheme, where $q' < q$, and q is set equal to q' afterwards. 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. The method works well for insertion and updates but, as we will show below, it leads to non-uniform samples when the transaction sequence contains deletions. Thus, unlike common practice, MBERNP(M) can only be used on sequences that do not contain any deletions.

In the remainder of this section, we show that MBERNP(M) does not support deletion transactions. 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 $M + 1$, due to an accepted insertion transaction. Each purge involves L Bernoulli subsampling steps with sampling rate p, where L is a geometrically distributed random variable with

$$\Pr[\, L = k \,] = p'(1 - p')^{k-1}$$

[6] *Subsampling* is the process of applying a sampling scheme on a sample, that is, on a dataset that is already sampled. Subsampling is used to decrease the size of a sample.

for $k \geq 1$. 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 for any $A \subset S$

$$\begin{aligned}\Pr\left[\, S' = A \,\right] &= \Pr\left[\,\text{all } r \in A \text{ retained, all } r \in S \setminus A \text{ purged} \mid \geq 1 \text{ item purged}\,\right] \\ &= \frac{p^{|A|}(1-p)^{M+1-|A|}}{p'}. \end{aligned} \tag{3.13}$$

After the purge operation has terminated, the sampling process proceeds with the new sampling rate qp^L.

We now give a simple example where MBERNP(M) does not produce a uniform sample; an illustration is given in figure 3.5. Consider the sequence $\gamma = (+r_1, +r_2, -r_1, +r_3)$ 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 i-th transaction, starting with $R_0 = S_0 = \emptyset$ and $Q_0 = 1$. The first insertion $+r_1$ is directly included into the sample; the sampling rate remains unmodified, $Q_1 = 1$. The insertion of r_2 triggers a purge operation and, using (3.13) with $A = \{\, r_1 \,\}$, we find that r_2 is selected as the sample item with probability $p_1 = p(1-p)/(1-p^2)$. The same holds for r_1; the sample becomes empty with probability $1 - 2p_1$. Also, the sampling rate is adjusted depending on the number L of purges so that $Q_2 = p^L$. Transaction $-r_1$ simply removes r_1 if present in the sample; $Q_3 = Q_2$. Transaction $+r_3$ is accepted with probability $Q_3 = p^L$ and rejected otherwise. In the case of a rejection, the sample remains unmodified. Otherwise, if r_3 is accepted, it is included into the sample and, when additionally $S_3 = \{\, r_2 \,\}$, another purge operation is triggered. By multiplying the probabilities along the paths in figure 3.5, summing up and replacing Q_3 by p^L, we yield

$$\begin{aligned}\Pr\left[\, S_4 = \{\, r_2 \,\} \mid L \,\right] &= p_1(1 - Q_3) + p_1 Q_3 p_1 \\ &= p^L(p_1^2 - p_1) + p_1, \end{aligned}$$

and

$$\begin{aligned}\Pr\left[\, S_4 = \{\, r_3 \,\} \mid L \,\right] &= (1 - 2p_1)Q_3 + p_1 Q_3 + p_1 Q_3 p_1 \\ &= p^L(p_1^2 - p_1 + 1). \end{aligned}$$

We can now uncondition on L and simplify the resulting infinite sum

$$\begin{aligned}\Pr[S_4 = \{r_3\}] &= \sum_{k=1}^{\infty} \Pr[L = k] \Pr[S_4 = \{r_3\} \mid L = k] \\ &= \sum_{k=1}^{\infty} (1-p^2)(p^2)^{k-1} p^k (p_1^2 - p_1 + 1) \\ &= \frac{1-p^2}{p^2}(p_1^2 - p_1 + 1) \sum_{k=1}^{\infty} (p^3)^k \\ &= \frac{p}{p+1},\end{aligned}$$

where we used the limit of the geometric series

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

for $|x| < 1$. Similarly,

$$\begin{aligned}\Pr[S_4 = \{r_2\}] &= \sum_{k=1}^{\infty} \Pr[L = k] \left(p^k(p_1^2 - p_1) + p_1\right) \\ &= \frac{p}{p+1} \frac{p^2+1}{p^2+p+1}.\end{aligned}$$

It follows that

$$\Pr[S_4 = \{r_2\}] < \Pr[S_4 = \{r_3\}]$$

for $p > 0$, so that MBERNP(M) 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. MBERNP(M) appears to work in the insertion-only setting, but this assertion has not yet been proven. Also, in the insertion-only setting, the scheme is superseded by modified reservoir sampling, which we consider next. As a final note, Tao et al. (2007) proposed an extension of MBERNP(M) that can be shown to produce non-uniform samples even in the insertion-only setting. The proof is similar to the one above. In short, the sequence $\gamma = (+r_1, +r_2, +r_3)$ with $M = 1$ leads to

$$\begin{aligned}\Pr[S_3 = \{r_1\}] &= 3/8 \\ \Pr[S_3 = \{r_2\}] &= 3/8 \\ \Pr[S_3 = \{r_3\}] &= 1/4,\end{aligned}$$

which is clearly non-uniform.

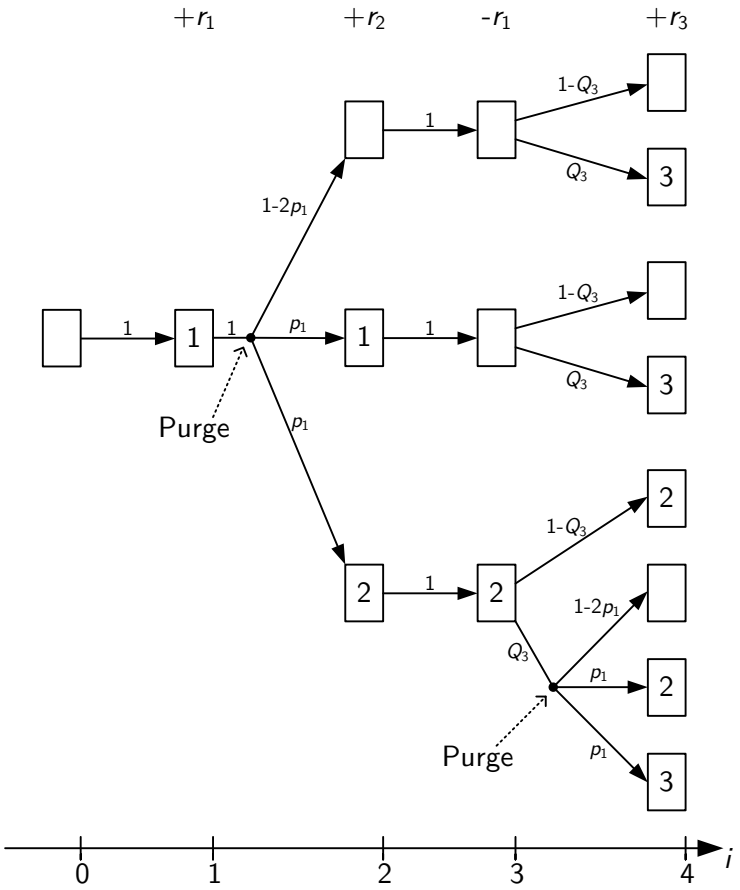


Figure 3.5: Counterexample for modified Bernoulli sampling with purging

C. Modified Reservoir Sampling, MRS(M)

Reservoir sampling can be extended to process update transactions in an incremental way. The modified scheme, denoted as MRS(M), processes insertion transactions as before; see equations (3.10) and (3.12). If an update transaction $\gamma_{i+1} = r \to r'$ arrives, item r—if present in the sample—is replaced by r':

$$S_{i+1} \leftarrow \begin{cases} S_i \setminus \{ r \} \cup \{ r' \} & r \in S_i \\ S_i & \text{otherwise.} \end{cases}$$

Again, insert operations can be sped up using skip counters. Since the dataset size, which determines the inclusion probability of inserted items, is not affected by update transactions, the value of the skip counter remains unchanged for updates.

D. Modified Reservoir Sampling and Deletions

The extension of MRS(M) to support deletion transactions is more complex. Suppose that $\gamma_{i+1} = -r^-$ is the first deletion transaction that occurs in γ. Also suppose that the reservoir has been filled initially before γ_{i+1} arrives, that is, $|S_i| = M$ and $N_i > M$. There are two different cases that can occur when processing deletion transaction γ_{i+1}:

1. Item r^- is not present in the sample, $r^- \notin S_i$. This event occurs with probability $1 - n_i/N_i$.

2. Item r^- is present in the sample, $r^- \in S_i$. This event occurs with probability n_i/N_i.

Case 1 is easy to handle. By the uniformity of S_i, every size-M subset from R_i is equally likely to be chosen as the sample S_i. It follows that all size-M subsets that do not contain item r^- are also chosen with equal probability. Thus, S_i represents a size-M uniform random sample of $R_{i+1} = R_i \setminus \{ r^- \}$ and we therefore set $S_{i+1} \leftarrow S_i$.

In case 2, we have no option but to remove r^- from the sample. Using the same argument as above, one finds that the resulting sample $S_i \setminus \{ r^- \}$ is a uniform random sample of R_{i+1}. However, the sample size has been reduced by one, and it is not immediately clear how to proceed.

Ideally, we would want to use subsequent insertions to "compensate" for previous deletions, that is, to cancel out their effect of reducing the sample size. An "obvious" algorithm would be as follows: Whenever the sample size matches its upper bound M, the algorithm handles insertions identically to RS(M). Whenever the sample size lies below the upper bound M 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 the deletion of item r^- is directly followed by an insertion of item r^+. Denote by S' the sample after these two operations. If

the sample is to be truly uniform, then the probability that $r^+ \in S'$ should equal M/N, conditional on $|S| = M$. Since $r^- \in S$ with probability M/N, it follows that

$$\begin{aligned}\Pr\left[r^+ \in S'\right] &= \Pr\left[r^- \in S,\ r^+ \text{ included}\right] + \Pr\left[r^- \notin S,\ r^+ \text{ included}\right] \\ &= \frac{M}{N} \cdot 1 + \left(1 - \frac{M}{N}\right) \cdot \frac{M}{N} > \frac{M}{N},\end{aligned} \tag{3.14}$$

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 our random pairing algorithm, which is introduced in chapter 4, is to carefully select an inclusion probability for each inserted item so as to ensure uniformity. In what follows, we discuss alternative approaches that have been proposed in the literature.

E. Modified Reservoir Sampling With Recomputation, MRSR(L, M)

An alternative approach to extend MRS(M) to support deletions is to simply continue reservoir sampling with a smaller sample size $n < M$ whenever a deletion has caused the sample to shrink. This leads to a slightly different procedure for handling insertion transactions because we now have to account for situations where the sample size is smaller than M. For insertion $\gamma_{i+1} = +r$, we have

$$S_{i+1} \leftarrow \begin{cases} S_i \cup \{\, r \,\} & N_i + 1 \leq M,\ n_i = N_i \\ \textsc{Replace}(S_i,\, r) & \text{with probability } n_i/(N_i + 1) \\ S_i & \text{with probability } (N_i - n_i + 1)/(N_i + 1). \end{cases} \tag{3.15}$$

This procedure is a direct extension of equation (3.10). The 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. (1997): as soon as the sample size falls below a prespecified lower bound L, recompute it from scratch using, for example, a list-sequential sampling scheme. The resulting method can be described as modified reservoir sampling with recomputation, denoted as MRSR(L, M). It is also called the "backing sample" method. The complete algorithm to process deletion $\gamma_{i+1} = -r$ is

$$S_{i+1} \leftarrow \begin{cases} S_i \setminus \{\, r \,\} & r \in S_i,\ N_i \leq L \vee n_i > L \\ S_i & r \notin S_i \\ \textsc{Srs}(R_{i+1},\, M) & \text{otherwise,} \end{cases}$$

where $\textsc{Srs}(R_{i+1}, M)$ computes a size-M simple random sample of R_{i+1}. MRSR(L, M) is semi-incremental with respect to deletions because access to the base data is required from time to time. It is incremental with respect to insertions and updates.

One may use skip counters with MRSR(L, M), but the value of the counter has to be recomputed with every deletion. This is because the dataset size and/or sample size changes so that the inclusion/exclusion probabilities assumed when generating the skip counter do not hold anymore. The use of a skip counter is therefore only beneficial when the number of deletions is low.

F. Modified Reservoir Sampling With Tagging (And Recomputation), MRST(L, M)

Tao et al. (2007) proposed another approach, denoted as MRST$(0, M)$, to extend MRS(M) with deletion support.[7] The basic idea is to tag deleted items instead of physically removing them from the sample. A deletion is therefore interpreted as an update that sets the "deleted flag" of the respective tuple; and updates are directly supported by MRS(M). Denote by N_i^+ the cumulated number of insertion transactions that have arrived in the stream up to and including transaction γ_i; $N_0^+ = 0$. If γ_{i+1} is an insertion $+r$, we set

$$S_{i+1} \leftarrow \begin{cases} S_i \cup \{\,(r, \text{false})\,\} & N_i^+ + 1 \leq M \\ \textsc{Replace}(S_i, (r, \text{false})) & \text{with probability } M/(N_i^+ + 1) \\ S_i & \text{with probability } (N_i^+ - M + 1)/(N_i^+ + 1), \end{cases}$$

and $N_{i+1}^+ = N_i^+ + 1$. Here, each element in the sample is a tuple (r, d), where r denotes the item and d the deletion flag. All items are marked as non-deleted at the time of their insertion. If $\gamma_{i+1} = -r$ is a deletion, we set

$$S_{i+1} \leftarrow \begin{cases} S_i \setminus \{\,(r, \text{false})\,\} \cup \{\,(r, \text{true})\,\} & (r, \text{false}) \in S_i \\ S_i & \text{otherwise,} \end{cases}$$

and $N_{i+1}^+ = N_i^+$. The net sample, which contains only the non-deleted items, is given by

$$S_i^* = \{\, r \mid (r, \text{false}) \in S_i \,\}.$$

As observed by Tao et al. (2007), the net sample size follows a hypergeometric distribution and averages to

$$\mathrm{E}\,[\,|S_i^*|\,] = \frac{N_i}{N_i^+} M.$$

The values of N_i and N_i^+ depend on the sequence γ. To shed some light on the expected sample size,[8] pick a sequence γ that satisfies

$$\lim_{i \to \infty} \frac{N_i^+}{i} = p^+$$

for some constant $0.5 \leq p^+ \leq 1$. The value of p^+ can be seen as the fraction of insertions in γ over the long run. Let $N_0 = |R_0|$ be the initial size of the dataset before applying γ and let S_0 be a size-M uniform sample of R_0. Then, $N_i = N_0 + N_i^+ - (i - N_i^+)$ and we obtain

$$\lim_{i \to \infty} \frac{N_i}{N_i^+} = \lim_{i \to \infty} \frac{N_0 + N_i^+ - (i - N_i^+)}{N_i^+} = \frac{2p^+ - 1}{p^+}.$$

[7] The algorithm has been developed independently from our random pairing algorithm given in chapter 4.

[8] The key ideas of the following derivation are due to Peter J. Haas.

Table 3.3: Expected sample size for MRST$(0, M)$

p^+	0.5	0.6	0.7	0.8	0.9	1
$\mathrm{E}[\,\lvert S^*\rvert\,]$	0	$0.33M$	$0.57M$	$0.75M$	0.88M	M

In the final equality, we made use of the fact that $\lim_{i\to\infty} N_i^+ = \infty$ under our assumptions. The expected net sample size in the limit is

$$\lim_{i\to\infty} \mathrm{E}[\,\lvert S_i^*\rvert\,] = \frac{2p^+ - 1}{p^+} M.$$

Table 3.3 gives the limit of $\mathrm{E}[\,\lvert S_i^*\rvert\,]$ for some choices of p^+. The algorithm shows good performance if deletions occur infrequently but it fails in the case of a stable dataset (where $p^+ = 0.5$). For this reason, we modify MRST$(0, M)$ so that it recomputes the sample whenever the net sample size has fallen below a prespecified lower bound L. The modified algorithm is denoted by MRST(L, M); it coincides with MRST$(0, M)$ for $L = 0$.

G. Correlated Acceptance/Rejection Sampling, CAR(M)

The correlated acceptance/rejection algorithm of Olken and Rotem (1992), denoted as CAR(M), maintains a size-M simple random sample with replacement. It is incremental with respect to insertions and updates, and semi-incremental with respect to deletions. CAR(M) 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 essentially maintains M independent MRSR$(1, 1)$ samples, although it is slightly more efficient. CAR(M) processes an insertion $\gamma_{i+1} = +r$ as follows. First, it generates a random number X_{i+1} from the binomial$(M,\, 1/N_{i+1})$ distribution, that is,

$$\Pr[\,X_{i+1} = k\,] = B(k;\, M, 1/N_{i+1})$$

for $0 \le k \le M$. Fast methods to generate X_{i+1} can be found in Devroye (1986, ch. X.4). Based on the observation that X_{i+1} is distributed as the number of MRSR$(1, 1)$ samples that accept r, CAR(M) replaces X_{i+1} random items of the current sample by X_{i+1} copies of r. Extending the previous Replace function by an additional argument for the number of items to replace, we have

$$S_{i+1} \leftarrow \textsc{Replace}(S_i,\, r,\, X_{i+1}).$$

To process a deletion $\gamma_{i+1} = -r$, CAR(M) replaces each occurrence of r by a random item drawn from the population:

$$S_{i+1} \leftarrow \biguplus_{r' \in S_i} \begin{cases} r' & r' \neq r \\ \textsc{Srs}(R_{i+1},\, 1) & r' = r \end{cases},$$

where samples obtained by repeated calls to SRS are independent. As discussed in section 2.1.2, we can obtain a uniform sample without replacement by removing duplicates.

H. Correlated Acceptance/Rejection Sampling Without Replacement, CARWOR(M)

The CAR(M) algorithm can be modified to sample directly without replacement. In this modified version, denoted as CARWOR(M), insertions and updates are processed as in MRS(M). A deletion transaction $\gamma_{i+1} = -r_j$ is processed as follows:

$$S_{i+1} \leftarrow \begin{cases} S_i \setminus \{ r_j \} \cup \text{SRS}(R_{i+1} \setminus S_i, 1) & r_j \in S_i \\ S_i & r_j \notin S_i \end{cases},$$

that is, whenever an item is deleted from the sample, we resample a replacement item from the base data. CARWOR(M) is similar to MRSR(M, M), but the sample is refilled instead of being entirely recomputed.

I. Summary and Roadmap

The classic survey sampling schemes only apply to insertion-only datasets; they are mainly used to compute the initial samples that are subsequently maintained by one of the other schemes.

The MBERN(q) scheme extends the applicability of Bernoulli sampling to arbitrary sequences of insertions, updates and deletions. The scheme has zero space overhead, does not require access to the base data and is simple to implement. In applications where the sample size variability and unboundedness of Bernoulli sampling are acceptable, MBERN(q) is clearly the method of choice.

The various bounded-size schemes differ with respect to cost and sample-size stability. For sequences that do not contain any deletion transactions, MRS(M) is the method of choice because it provides both a fixed sample size and a fixed footprint without ever accessing the base data. Sequences with deletions are more difficult to handle. In fact, all of the available schemes require access to the base data to compensate for the reduced sample size caused by deletions. CARWOR(M) provides a fixed sample size but accesses base data at every sample deletion. MRSR(L, M) and MRST(L, M) are more efficient because base-data access is deferred until the sample has become too small. This improved efficiency comes at the cost of sample-size stability.

An ideal bounded-size scheme for stable datasets would not require any base data accesses while at the same time providing a fixed sample size. Unfortunately, such a scheme does not exist: every sample deletion either leads to a decrease of the sample size or, if compensated, to a base data access. In chapter 4, however, we introduce a novel bounded-size scheme for stable datasets that comes close to the ideal scheme. Our scheme is called *random pairing*, denoted by RP(M); its properties are given in table 3.2. RP(M) is an extension of reservoir sampling that makes use of newly

inserted items to compensate for prior deletions. The scheme does not require access to the base data, even if the transaction sequence contains deletions. Experiments show that, when the fluctuations of the dataset size are not too extreme, random pairing is the algorithm of choice with respect to speed and sample-size stability.

Also in chapter 4, we consider algorithms for periodically resizing a bounded-size random sample upwards; these algorithms are well-suited for growing datasets. Compared to MBERN(q), which in expectation also produces larger samples when the dataset is growing, such algorithms have the advantage that the sample can be grown in a controlled manner and in arbitrary increments (i.e., not necessarily linear to the dataset size). We prove that any resizing algorithm cannot avoid accessing the base data and we provide a novel resizing algorithm that minimizes the time needed to increase the sample size.

3.5.2 Multiset Sampling

Relatively little is known about sampling from evolving multisets. The survey sampling schemes BERN(q), RS(M), and MIN(M) can all be used to maintain a sample of an insertion-only multiset. This is because the inclusion/exclusion decisions are independent of the value of the inserted item. To derive the multiset versions of the algorithms, simply replace each set union ($\cup$) by a multiset union ($\uplus$) in the respective descriptions. When the transaction sequence contains updates or deletions, however, maintenance becomes much harder because both the dataset R and the sample S may contain multiple copies of the updated or deleted item. As discussed in section 3.2, update and deletion transactions refer to only a single one of these copies, so it is not immediately clear how to proceed. In fact, the only known uniform sampling scheme is a semi-incremental scheme based on MBERN(q). We describe the scheme below, but first we introduce some helpful notation.

Denote by $X_i(r)$ the frequency of item r in the sample and by $N_i(r)$ its frequency in the dataset just after transaction γ_i has been processed. The quantity $X_i(r)$ is typically a random variable, while $N_i(r)$ is completely determined by the sequence γ. We subsequently assume that the sample is stored in compressed form, as proposed by Gibbons and Matias (1999). In the compressed representation, each element of the sample comprises a pair $(r, X_i(r))$ when $X_i(r) > 1$ or a singleton (r) when $X_i(r) = 1$. An item r with $X_i(r) = 0$ does not appear in the sample. For example, the compressed form of the set

$$\{ A, A, A, B \}$$

is given by

$$\{ (A, 3), (B) \}.$$

The compressed representation has the advantage of requiring less storage space than the multiset representation. In other words, the footprint of the sample is reduced; the sample size remains unaffected. The compression also leads to a concise way to refer to sample insertions and deletions. We will write

$$X_{i+1}(r) \leftarrow X_i(r) + 1$$

to denote the sample inclusion of item r,

$$S_{i+1} \leftarrow \begin{cases} S_i \cup \{ (r) \} & X_i(r) = 0 \\ S_i \setminus \{ (r) \} \cup \{ (r, 2) \} & X_i(r) = 1 \\ S_i \setminus \{ (r, j) \} \cup \{ (r, j+1) \} & X_i(r) = j > 1. \end{cases}$$

Similarity, we write $X_{i+1}(r) \leftarrow X_i(r) - 1$ to denote a sample deletion.

Gibbons and Matias (1999) used sample compression to enforce an upper bound on the footprint of the sample. In a way similar to MBERNP(q), their *concise sampling* algorithm purges the sample whenever its footprint exceeds a given space budget. As shown by Brown and Haas (2006), the approach does not produce uniform samples, so that we do not consider it here. Brown and Haas also propose a *hybrid* version of RS(M) that uses the compressed representation as long as the entire dataset can be represented within the space budget, but it switches back to a non-compressed representation afterwards.[9] This approach is especially beneficial for datasets with few distinct items; it supports insertion transactions only.

A. Modified Bernoulli Sampling for Multisets, MBERNM(q)

In the multiset setting, S_i is a Bernoulli sample of R_i with sampling rate q if and only if each $X_i(r)$ is binomially distributed with

$$\Pr [X_i(r) = k] = B(k; N_i(r), q)$$

and the random variables $X_i(r)$, $r \in \mathscr{R}$, are mutually independent. Thus, each item is maintained independently of the other items; the value of $X_i(r)$ remains unaffected if an item $r' \neq r$ is inserted or removed. For $i = 0$, both the dataset and the sample are empty and we have $N_i(r) = X_i(r) = 0$ for all $r \in \mathscr{R}$.

With these definitions at hand, we now describe an algorithm due to Gemulla et al. (2007) for maintaining a Bernoulli sample of a multiset. The algorithm is denoted MBERNM(q). If γ_{i+1} corresponds to an insertion $+r$, then

$$X_{i+1}(r) \leftarrow \begin{cases} X_i(r) + 1 & \text{with probability } q \\ X_i(r) & \text{with probability } 1 - q. \end{cases} \tag{3.16}$$

If γ_{i+1} corresponds to a deletion $-r$, then

$$X_{i+1}(r) \leftarrow \begin{cases} X_i(r) - 1 & \text{with probability } X_i(r)/N_i(r) \\ X_i(r) & \text{with probability } 1 - X_i(r)/N_i(r). \end{cases} \tag{3.17}$$

Laxly speaking, when $X_i(r)$ of the $N_i(r)$ copies of item r are present in the sample, the "deleted copy" is one of the copies in the sample with probability $X_i(r)/N_i(r)$.

[9] The paper also proposes a hybrid BERN(q) scheme that switches to reservoir sampling as soon as the Bernoulli sample size exceeds an upper bound. It can be shown that this approach is non-uniform, see section 4.2.1B.

Update transactions are modeled as a deletion followed by an insertion. A formal proof of the correctness of the algorithm is given in Gemulla et al. (2007).

MBERNM(q) is semi-incremental with respect to deletions because processing a deletion requires knowledge of the quantity $N_i(r)$, which has to be obtained from the underlying dataset R_i. One might consider maintaining the counters $N_i(r)$ locally for each distinct item in the dataset, but, of course, this is equivalent to storing the entire dataset in compressed form.

B. Summary and Roadmap

The classic survey sampling schemes can be used to maintain a sample of an insertion-only multiset. As discussed above, neither the survey sampling nor the set sampling schemes can be used when the dataset is additionally subject to update or deletion transactions. The only known scheme that does support these types of transactions is MBERNM(q), a scheme that relies on frequent base-data accesses.

In chapter 5, we propose a novel sampling scheme called *augmented Bernoulli sampling*, ABERN(q), that is able to maintain a Bernoulli sample of an evolving multiset without ever accessing base data. The basic properties of the scheme are given in table 3.2. ABERN(q) maintains for every sample item a so-called "tracking counter." We show that these counters can be exploited to estimate population frequencies, sums, averages, and the number of distinct items in an unbiased manner, with lower variance than the usual estimators based on a Bernoulli sample.

3.5.3 Distinct-Item Sampling

Distinct-item sampling schemes sample uniformly from $D(R)$, the set of distinct items in R. All previously known schemes are based on hashing, that is, they make use of a set of hash functions $h_1, \ldots, h_k$ for some $k \geq 1$. The sampling schemes are deterministic in the sense that they produce the same result when run repeatedly on the same transaction sequence using the same set of hash functions. The "randomness" of the sample thus depends only on the degree of randomness in the hash functions.

Ideally, hash functions $h_1, \ldots, h_k$ are independent and truly random, that is, they act like independent random functions from $\mathscr{R}$ to the hash range. In practice, however, truly random hash functions are unusable because their description requires space linear to the size of $\mathscr{R}$ (or at least R). The "practitioner's approach", therefore, is to use weaker hash functions instead and to assume that these functions behave as if they were truly random. Interestingly, this assumption has recently received some theoretical justification (Mitzenmacher and Vadhan 2008). The "theorist's approach" is to design the sampling algorithms in such a way that they only require a "certain degree" of randomness in the hash functions. The downside is that these algorithms may have a significant overhead in space and/or time, which makes them unappealing in practice. Thus, practice and theory differ widely; we describe algorithms of both types. A more detailed discussion of hash functions is given in section 6.1.

Table 3.4: Uniform maintenance schemes for distinct-item sampling

	Insert	Update	Delete	Size	Footprint	Hash functions (#)
BERND(q)	✓	-	-	Unbounded	Unbounded	truly random (1)
MBERND(q)	✓	✓	✓	Unbounded	Unbounded	truly random (1)
BERNDP(M)	✓	-	-	Bounded	Bounded[a]	truly random[c] (1)
MIND(M)	✓	-	-	Fixed	Fixed[a]	M-min-wise (1)
MINDWR(M)	✓	-	-	Bounded[b]	Fixed[a]	min-wise (M)
DIS(M)	✓	✓	✓	Bounded	Bounded[a]	truly random[c] (M)
AMIND(M)	✓	○[d]	○[d]	Bounded	Fixed[a]	M-min-wise (1)
ABERND(q)	✓	✓	✓	Unbounded	Unbounded	none

[a] Assuming constant space for storing an item / a counter / a hash function.
[b] After conversion to a without-replacement sample.
[c] See discussion in text.
[d] ✓ when deletions occur infrequently.

A summary of the algorithms and their requirements on the hash functions is given in table 3.4. Note that when R is a set, distinct-item sampling and set sampling coincide: all distinct-item schemes can thus be used to maintain a uniform sample from a set.

A. Bernoulli Sampling for Distinct Items, BERND(q)

The BERN(q) scheme can be adapted to sample uniformly from the distinct items of an insertion-only dataset. Recall that BERN(q) accepts each newly-inserted item r with probability q and rejects it with probability $1 - q$; see equation (3.7). In an actual implementation, the decision of whether or not to sample r may be taken by first generating a uniform random number $U \in (0, 1)$ and—since $\Pr[U < u] = u$ for $u \in [0, 1]$—accepting the item if and only if $U < q$. Given a truly random hash function $h : \mathscr{R} \to (0, 1)$, we can treat the hash value $h(r)$ of r as if it were a uniform random number and thus replace U by $h(r)$ in the algorithm. For an insertion $\gamma_{i+1} = +r$, set

$$S_{i+1} \leftarrow \begin{cases} S_i \cup \{ r \} & h(r) < q \\ S_i & \text{otherwise.} \end{cases}$$

Here, the sample is treated as a set so that each item is sampled at most once. The key idea of this hash-based Bernoulli scheme, which we denote BERND(q), is that repeated insertions of the same item lead to the same random hash value. Thus, each item is either accepted at its first insertion or it is never going to be accepted. It

follows directly that the probability that an item is sampled does not depend on its frequency in R. Since each item is also sampled independently and $\Pr[\,h(r) < q\,] = q$ for all $r \in \mathscr{R}$, the sample is indeed uniform; its size follows the binomial$(|D(R)|, q)$ distribution.

The above scheme is well-known in the literature. Denning (1980) makes use of it to prevent inference of individuals from sensitive database queries; Duffield and Grossglauser (2001) uses it as a building block for a network sampling scheme ("trajectory sampling"); Duffield et al. (2001) proposes a weighted version of the scheme ("size-dependent sampling"); Gibbons and Tirthapura (2001) apply it to estimate simple functions on the union of data streams ("coordinated 1-sampling"); and Hadjieleftheriou et al. (2008) uses it to estimate the selectivity of weighted-set-similarity queries ("hashed samples"). The reason for the widespread usage of BERND(q) is that the scheme (and all other hash-based schemes) can be used to *correlate the sampling process* of several datasets by simply using the same hash functions. We will also make use of sample correlation through hashing in chapter 6.

Note that BERND(q) requires the hash function to be truly random in order to produce uniform samples. Less randomness is required when the sample is solely used to estimate counts, sums, and averages over the distinct-items. In fact, pairwise independent hash functions (section 6.1C) suffice to obtain unbiasedness and variance of the "Bernoulli estimator" as given in table 2.3 on page 18. This nice property does not hold for any of the subsequent schemes.

B. Modified Bernoulli Sampling for Distinct Items, MBERND(q)

The BERND(q) scheme does not directly support update and deletion transactions. The reason is that maintaining its invariant

$$r \in S_i \Leftrightarrow r \in R_i \wedge h(r) < q$$

would require access to the base data in the case of updates and deletions. For example, a dataset deletion should lead to a sample deletion if and only if the *last* copy of the item has been deleted from the dataset. The frequency of the deleted item, however, cannot be obtained from the BERND(q) sample. Fortunately, we can extend BERND(q) so that it stores the frequency of each sampled item along with the item itself; each sample item is then a pair $(r, N_i(r))$. We denote the modified scheme by MBERND(q). The frequency counters can be maintained incrementally as follows. An insertion $\gamma_{i+1} = +r$ is processed as

$$S_{i+1} \leftarrow \begin{cases} S_i \cup \{\,(r,1)\,\} & h(r) < q,\ N_i(r) = 0 \\ S_i \setminus \{\,(r, N_i(r))\,\} \cup \{\,(r,\ N_i(r)+1)\,\} & h(r) < q,\ N_i(r) > 0 \\ S_i & \text{otherwise,} \end{cases}$$

and a deletion $\gamma_{i+1} = -r$ as

$$S_{i+1} \leftarrow \begin{cases} S_i \setminus \{ (r, N_i(r)) \} & h(r) < q,\ N_i(r) = 1 \\ S_i \setminus \{ (r, N_i(r)) \} \cup \{ (r,\ N_i(r) - 1) \} & h(r) < q,\ N_i(r) > 1 \\ S_i & \text{otherwise,} \end{cases}$$

and an update as a deletion followed by an insertion. In addition to facilitating efficient maintenance, knowledge of the frequencies of sampled items can also be exploited for estimation purposes.

The idea of storing a counter with each sampled item can be extended further. In fact, arbitrary data structures can be associated with each sampled item, provided that the data structures are themselves incrementally maintainable. For example, consider a table of sales transactions, where each transaction consists of a customer and a price. We can then compute and maintain an MBERND(q) sample of the distinct customers and associate with each customer the (exact) total of all their transactions. Such a sample is useful, for example, to estimate the fraction of customers that have a total of less than a given quantity. This remarkable observation is due to Gibbons (2001), who maintains, for each distinct item, a uniform sample of the respective base data items. Of course, this idea can also be applied to BERND(q) as well as to most of the subsequent schemes, with DIS(M) being the exception.

C. Bernoulli Sampling for Distinct Items With Purging, BERNDP(M)

In a way similar to MBERNP(M), an upper bound on the sample size of BERND(q) can be realized by purging the sample from time to time. The bounded scheme, denoted as BERNDP(M), starts with an initial value of $q = 1$. Whenever the sample size exceeds M items due to a successful sample insertion, the sample is first subsampled using BERND(q'/q) sampling for some $q' < q$, and the value of q is then set to q'. This so-called purging step is repeated until the sample size has fallen below M. A clever implementation of this scheme for the case $q' = 0.5q$ is given by Gibbons (2001).[10] The scheme cannot be used when the transaction sequence contains updates or deletions. In fact, BERNDP(M) reduces to MBERNP(M) when the dataset does not contain any duplicates; the non-uniformity arguments against MBERNP(M) directly apply.

We now show that that pairwise-independent hash functions are not sufficient to guarantee the uniformity of the sample; they do not even lead to an equal-probability sampling design. Denote by h the hash function and suppose that h is chosen uniformly from the $\mathscr{H}_p$ family discussed in section 6.1C, that is, $h(x) = ax + b \mod p$ for a fixed prime p and randomly chosen integers $0 \le a, b < p$.[11] Then, h is

[10] Gibbons calls his scheme "distinct sampling". We do not make use of this name to avoid possible confusion with distinct-item sampling in general.

[11] This is almost the hash function used by Gibbons, who additionally required that $a > 1$. We explicitly allow $a = 0$ because otherwise the hash values would not be pairwise independent. However, the ongoing analysis remains unaffected because, whenever $a = 0$ and $N > M$, all items have the same hash value and the resulting sample will be empty.

Table 3.5: Frequencies of each possible sample for BERNDP(1)

	$N = 1$	$N = 2$	$N = 3$	$N = 4$	$N = 5$
$S = \emptyset$	0	22,361,431	16,773,121	17,705,179	17,238,455
$S = \{ 1 \}$	67,092,481	22,365,525	16,773,120	12,114,152	10,017,341
$S = \{ 2 \}$		22,365,525	16,773,120	12,579,499	9,318,632
$S = \{ 3 \}$			16,773,120	12,579,499	11,182,080
$S = \{ 4 \}$				12,114,152	9,318,632
$S = \{ 5 \}$					10,017,341
Uniform?	✓	✓	✓	-	-

pairwise independent, that is, for all pairs of integers $0 \leq r_1, r_2 < p$ and pairs of hash values $0 \leq v_1, v_2 < p$, we have

$$\Pr[\, h(r_1) = v_1, h(r_2) = v_2 \,] = \frac{1}{p^2}. \tag{3.18}$$

We now fix a value of $p = 8{,}191$ and run BERNDP(1), with $q' = 0.5q$, successively with each of the p^2 possible hash functions (*all* combinations of a and b). If BERNDP(1) were uniform, all equally-sized samples would be reported with the same probability. Table 3.5 shows the distribution of the samples for $M = 1$ (rows) for various datasets (columns). Each dataset consists of N items, where N varies across columns, and comprises integers $\{ 1, \ldots, N \}$. As can be seen, BERNDP(1) does not produce uniform samples when $N > 3$. For example, when $N = 5$, sample $\{ 3 \}$ is reported more than 19% more frequently than sample $\{ 2 \}$. Similar behavior can be seen for other choices of p. Of course, when h is truly random, the algorithm does produce uniform samples.

D. Min-Hash Sampling, MIND(M)

Min-wise sampling can be adapted to sample from the distinct items of an insertion-only dataset. The adapted algorithm maintains a sample of size $\min(|D(R)|, M)$ and is denoted by MIND(M). As with the distinct-item versions of Bernoulli sampling, MIND(M) makes use of a random hash function h. The idea is to run min-wise sampling as described in section 3.4.2 but to replace the random tags by the hash value of the arriving item. For this reason, the algorithm is referred to as min-hash sampling. For insertion $\gamma_{i+1} = +r$, we set

$$S_{i+1} \leftarrow \begin{cases} S_i \cup \{ r \} & |S_i| < M \\ S_i \setminus \{ \operatorname*{argmax}_{r' \in S_i} h(r') \} \cup \{ r \} & |S_i| = M,\ r \notin S_i,\ h(r) < \max\limits_{r' \in S_i} h(r') \\ S_i & \text{otherwise,} \end{cases}$$

where S_i is treated as a set. Updates and deletions are not supported. The efficiency of the algorithm can be improved by storing the hash values together with the

sampled items. MIND(M) does not require h to be truly uniform; it suffices if h is M-min-wise independent; see the discussion in section 6.1D. The algorithm is used by Broder (1997) to estimate resemblance and containment between multiple datasets.

E. Min-Hash Sampling With Replacement, MINDWR(M)

A with-replacement version of MIND(M) is given in Datar and Muthukrishnan (2002) and Dasu et al. (2002). The MINDWR(M) scheme maintains M independent MIND(1) samples. A different hash function is used for each sample. Compared to MIND(M), MINDWR(M) has the advantage that it requires only min-wise (instead of M-min-wise) independent hash functions. It has the disadvantage that sampling is with replacement. Moreover, M hash values (instead of a single one) have to be computed for each incoming item so that the CPU cost is significantly higher; it grows linearly with M.

F. Dynamic Inverse Sampling, DIS(M)

Cormode et al. (2005) proposed a bounded-size scheme that can handle arbitrary insertions and deletions; a similar scheme is given by Frahling et al. (2005). The former scheme is called dynamic inverse sampling, denoted as DIS(M). It makes use of M data structures and M independent hash functions, one per data structure. Each data structure maintains—with some success probability $p > 0$—a *single* item chosen uniformly and at random from the set of distinct items in R. The sample size is thus random in $[0, M]$ and sampling is with replacement.

Each data structure consists of $O(\log|\mathscr{R}|)$ buckets. Each inserted or deleted item affects exactly one of the buckets in a data structure; the hash function associated with the data structure 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 $(1-\alpha)\alpha^{i-1}$, where $0 < \alpha < 1$ is a parameter of the algorithm.[12] Each bucket consists of the *sum* of the items (treated as integers) inserted into it, a *counter* of the number of inserted items and a data structure for *collision detection*. Adding an item to (resp., deleting an item from) a bucket simply involves incrementing (resp., decrementing) the counter by 1, incrementing (resp., decrementing) the sum by the item value and updating the collision detection structure accordingly. If there is a bucket in the data structure that contains exactly one distinct item, then the data structure "succeeds," and this item is returned as a random sample of size 1; otherwise, the data structure fails. Note that the lower-numbered buckets are more likely to succeed when the dataset size is small; the higher-numbered buckets handle large datasets. The point is that the stored items do not need to be maintained individually; in the important case where a bucket contains only a single distinct item (as indicated by the collision-detection data structure), the value of the sum is

[12] Suppose that hash function h maps uniformly into the range $1, \ldots, H$. Then, the bucket number of item r is given by $\lceil \log_{1/\alpha}(H/h(r)) \rceil$.

Table 3.6: Frequencies of each possible sample for DIS(1)

	$N = 1$	$N = 2$	$N = 3$	$N = 4$	$N = 5$
$S = \emptyset$	0	6,776,653	3,388,337	5,553,787	4,520,971
$S = \{1\}$	67,092,481	30,157,914	23,029,192	16,806,972	14,415,533
$S = \{2\}$		30,157,914	17,645,760	13,962,375	10,684,929
$S = \{3\}$			23,029,192	13,962,375	12,370,586
$S = \{4\}$				16,806,972	10,684,929
$S = \{5\}$					14,415,533
Uniform?	✓	✓	-	-	-

equal to the value of the counter times the value of the item; the item can thus be extracted. Cormode et al. (2005) have shown that, for an appropriate choice of α, the success probability p is at least 14.2%.

DIS(M) is the only known bounded-size sampling method that is "delete-proof" in that the sample-size distribution depends only on the size of the dataset, regardless of how it has been produced. Indeed, each deletion of an item precisely cancels the effect of the prior insertion of the item. The main disadvantage of DIS(M) is its low space efficiency. For example, with 32-bit items and a choice of $\alpha = \sqrt{2/3}$ as suggested by Cormode et al. (2005), we need more than 100 buckets per data structure, and thus exploit at most 1% of the memory allocated to the sample. Also, as in MINDWR(M) sampling, processing an insertion or deletion transaction requires the computation of M hash values (instead of just one).

DIS(1) has been developed with the intention of making BERNDP(1) delete-proof.[13] We thus conjecture that pairwise independent hash functions do not suffice to guarantee uniformity. To validate our conjecture, we make use of the same experimental setup that was used to prove the non-uniformity of BERNDP(1) under pairwise independent hash functions.[14] Table 3.6 shows our results; DIS(1) indeed produces non-uniform samples for $N > 2$. Of course, DIS(1) does produce uniform samples when truly random hash functions are used.

G. Summary and Roadmap

BERND(q) and MBERND(q) are the methods of choice for maintaining a Bernoulli sample of the distinct items in an evolving dataset. These methods have low space and time overhead and they are easy to implement.

[13] Private communication. In essence, DIS(1) stores information about the buckets that BERNDP(1) would have purged; this information can be used to "undo" purge operations. Similar ideas can be exploited for delete-proof distinct-count estimation, see for example Ganguly (2007).

[14] We used a value of $\alpha = \sqrt{2/3}$ and the "greedy" version of DIS(1). Other values of α as well as usage of the basic version led to similar results.

Bounded-size sampling is more complex. For insertion-only datasets, MIND(M) is the method of choice. It supersedes BERNDP(M) and MINDWR(M) because the latter schemes do not provide a fixed sample size. When the dataset is subject to update and/or deletion transactions, DIS(M) is the only available scheme that is incremental, although its space efficiency is low.

Most of the above distinct-item schemes are based on truly random hash functions, which are infeasible in practice. In chapter 6, we take a closer look at available hash functions and discuss their suitability for random sampling, arguing that hash functions that "look" truly uniform suffice in practice. The main part of chapter 6 is concerned with MIND(M) sampling. We propose a simple extension that augments the sample with frequency counters. Our extension is denoted as AMIND(M) and adds support for a limited number of updates and deletions. We provide an unbiased, low-variance estimator for the number of distinct items in the population (or a subset thereof) from an MIND(M) or AMIND(M) sample. Such estimators are required to scale-up other estimates from the sample to the entire dataset. Finally, we show that two or more AMIND(M) samples can be combined to obtain an AMIND(M) sample of arbitrary unions, intersections, and differences of their underlying datasets.

In table 3.4, we also list a scheme denoted as ABERND(q). As discussed in chapter 5, the scheme is based on our ABERN(q) *multiset* sampling scheme, from which a distinct-item sample can be extracted whenever needed. Interestingly, ABERND(q) is the only distinct-item scheme that does not make use of hashing. However, the size of the underlying ABERN(q) sample depends on the size of the entire dataset instead of just the number of distinct items in it. When the dataset contains many frequent items and a multiset sample is not required, MBERND(q) is usually a better choice.

3.5.4 Data Stream Sampling

We now proceed to the discussion of uniform sampling from sliding windows over data streams. We consider both sequence-based and time-based windows; see section 2.2.5 for the respective definitions. Sampling is used for data stream windows when it is infeasible to store the entire window. As a consequence, we are only interested in incremental schemes; semi-incremental or even non-incremental schemes cannot be used because the content of the window is not accessible.

We model a data stream as an infinite sequence $R = (e_1, e_2, \ldots)$ of items. Each item e_i has the form (i, t_i, r_i), where i is the position of the item in the data stream, $t_i \in \mathbb{R}$ denotes a timestamp, and $r_i \in \mathscr{R}$ denotes the data associated with the item. Throughout this thesis, we assume that $t_i < t_j$ for $i < j$, that is, the timestamps of the items are strictly increasing.[15] As before, the data domain $\mathscr{R}$ depends on the application; for example, $\mathscr{R}$ might correspond to a finite set of IP addresses or to an infinite set of readings from one or more sensors.

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

Table 3.7: Uniform maintenance schemes for data stream sampling

	Sequence-based	Time-based	Size	Footprint
BERNW(q)	✓	✓	Unbounded	Unbounded
PASSIVE(M)	✓	-	Fixed	Fixed[a]
CHAIN(M)	✓	-	Bounded[b]	Unbounded
PS(M)	✓	✓	Bounded[b]	Unbounded
PSWOR(M)	✓	✓	Fixed	Unbounded
BPS(M)	✓	✓	Bounded	Bounded[a]
BPSWOR(M)	✓	✓	Bounded	Bounded[a]

[a] Assuming constant space for storing an item / a counter / a priority.

[b] After conversion to a without-replacement sample.

Denote be R_i the set $\{e_1, \ldots, e_i\}$ of the first i items in R. After the arrival of i items, denote by $W_{i,N} = R_i \setminus R_{i-N}$ a *sequence-based window* of size N; the corresponding sample is denoted S_i. The length of the window, that is, the time span covered by the items in the window, is denoted by $\Delta_{i,N} = t_i - t_{N-i}$. Unless all timestamps in the stream are equidistant, the window length $\Delta_{i,N}$ varies over time. The window size N, however, is constant so that, for without-replacement sampling, the sample size cannot exceed N. We therefore make use of a slightly different notion of bounded-size and bounded-space sampling. We say that a sequence-based scheme is bounded in size if it can guarantee that the sample size is always strictly less than N. The situation is more subtle for space bounds; we take the view that a sequence-based scheme is bounded in space if it never stores all N items simultaneously.

A similar notation is employed for *time-based windows*. Denote by $R(t)$ the set of items from R with a timestamp smaller than or equal to t. Denote by $W_\Delta(t) = R(t) \setminus R(t-\Delta)$ a time-based sliding window of length Δ, by $N_\Delta(t) = |W_\Delta(t)|$ the size, and by $S(t)$ the sample of the window at time t. Unless all timestamps in the stream are equidistant, the window size $N_\Delta(t)$ varies over time. The window length Δ is constant.

A summary of the schemes discussed in the following is given in table 3.7. All schemes but PSWOR(M) are due to Babcock et al. (2002); they are discussed in more detail in Haas (2009). Note that some of the schemes cannot be used with time-based windows. In fact, sampling from time-based windows is considerably harder than sampling from sequence-based windows.

A. Bernoulli Sampling for Sliding Windows, BERNW(q)

Bernoulli sampling can be used to sample both sequence-based and time-based sliding windows. The procedure is the same in both cases and, in fact, similar to the different versions of Bernoulli sampling we have discussed already. Each arriving item is accepted into the sample with probability q and rejected with probability $1 - q$. An accepted item is kept in the sample until it expires (either after N successive insertions or after Δ time units have elapsed), in which case it is removed from the sample. Since each item is sampled independently from all the other items, the uniformity of the sample is maintained.

For sequence-based windows, the sample size follows a binomial(N, q) distribution and averages to Nq. Since the variance of the binomial distribution is low, the sample size will stay close to its expected value with high probability. Nevertheless, the scheme is unbounded because $\Pr[\,|S_i| = N\,] = q^N > 0$ for $i \geq N$. For time-based windows, the sample size has a binomial$(N(t), q)$ distribution. It averages to $N(t)q$ so that the sample grows and shrinks with the window size.

B. Passive Sampling, PASSIVE(M)

The "passive" sampling scheme, denoted as PASSIVE(M), is an extension of reservoir sampling to sequence-based windows. The idea is to build an initial sample by applying $RS(M)$ to the first N items of the stream. Afterwards, an arriving item is included into the sample if and only if its arrival coincides with the expiration of a sample item, which is in turn removed from the sample. The output of PASSIVE(M) is a size-M uniform sample. To see this, observe that the distribution of the sample is identical to the one of MRS(M), which is uniform, applied to the sequence $(+e_1, \ldots, +e_N, e_1 \rightarrow e_{N+1}, e_2 \rightarrow e_{N+2}, \ldots)$. A disadvantage of PASSIVE(M) for some applications is that the sample is highly periodic in that, for $i \geq N$, items $e_i, e_{i+N}, e_{i+2N}, \ldots$ are either all accepted into the sample or all rejected. In fact, Haas (2009) pointed out that when the stream is periodic with period N, the content of the sample does not change after the N initial steps, that is, $S_N = S_{N+1} = S_{N+2} = \ldots$.

C. Chain Sampling, CHAIN(M)

Chain sampling, denoted as CHAIN(M), is a fixed-size sampling scheme with replacement that, at the expense of some memory, does not show the periodic behavior of PASSIVE(M). We describe the CHAIN(1) scheme; samples of size M are obtained by running M independent copies of the CHAIN(1) scheme in parallel.

The key idea of chain sampling is to store, in addition to the sample item itself, a list of items that replace the sample item when it expires. This so-called chain retains the arrival order of the items in the stream, that is, new items are appended at the end of the chain. The sample item is also stored in the chain so that "sample

item" is simply a shortcut for "first item in the chain." The algorithm is as follows.[16] An arriving item e_i is accepted with probability $1/\min(i, N)$. If accepted, e_i replaces the entire chain and the algorithm decides on an item e_Z that replaces e_i upon its expiration; index Z is set to a value chosen uniformly and at random from $[i+1, i+N-1]$. Now, suppose that e_i is rejected, that is, e_i does not replace the entire chain. Then, item e_i is appended to the chain if and only if $Z = i$. In this case, Z is reinitialized to a random integer in $[i+1, i+N-1]$. Finally, the scheme checks whether or not the sample item has expired; if yes, it is removed and the now-first item in the chain becomes the new sample item.

To see why the algorithm works, suppose for induction that sample S_{i-1} is uniform; this assumption trivially holds for $i - 1 = 1$ because $S_1 = \{ e_1 \}$ with probability 1. When item e_i arrives, it becomes the sample item if and only if it replaces the entire chain—an event that happens with probability $1/\min(i, N)$. Thus,

$$\Pr[S_i = \{ e_i \}] = \Pr[e_i \text{ accepted}] = \frac{1}{\min(i, N)}. \tag{3.19}$$

Otherwise, we check whether or not the current sample item equals e_{i-N} and thus expires. If yes, the chain has not been entirely replaced by an arriving item since the arrival of e_{i-N}. Since Z has been set to a random index in $[i-N+1, i-1]$ at that time, and item e_Z has been appended to the chain upon its arrival, every item in $e_{i-N+1}, \ldots, e_{i-1}$ is equally likely to be second in the chain and thus to become the new sample item. Otherwise, if the sample item does not equal e_{i-N} and thus does not expire, it follows from the uniformity of S_{i-1} that each item in $e_{i-N+1}, \ldots, e_{i-1}$ is equally likely to be the sample item. Putting both observations together, we find that whether or not the sample item expires, each item in $e_{i-N+1}, \ldots, e_{i-1}$ is equally likely to be sampled. Since there are $\min(i, N) - 1$ such items, we have for $j \in [i-N+1, i-1]$

$$\Pr[S_i = \{ e_j \}] = \frac{\Pr[e_i \text{ rejected}]}{\min(i, N) - 1} = \frac{1}{\min(i, N)}$$

and uniformity follows together with (3.19).

The entire chain may consist of up to N elements so that the scheme is unbounded. In practice, however, Babcock et al. (2002) and Haas (2009) have shown that the expected length of the chain is at most $e = O(1)$, and that the actual length does not exceed $O(\log N)$ with high probability. Thus, on average, the independence improvement of CHAIN(M) over PASSIVE(M) comes at a multiplicative cost of at most $e \approx 3$.

[16] The chain sampling scheme given here is a slightly modified version of the original scheme of Babcock et al. (2002). The scheme has been modified to emphasize its similarity to priority sampling.

D. Priority Sampling, PS(M)

The priority sampling scheme, denoted as PS(M), can be seen as an extension of chain sampling to time-based sliding windows. Again, we describe the PS(1) scheme; samples of size M are obtained by running M independent copies of the scheme.

The reason why chain sampling cannot be used with time-based sliding windows is that the window size $N(t)$ is variable and unknown. To ensure uniformity, an arriving item must be chosen as the sample item with probability $1/N(t)$, a probability that we do not know.[17] Priority sampling avoids this problem by making use of an idea similar to min-wise sampling. The scheme assigns a random priority chosen uniformly from the unit interval $(0, 1)$ to each arriving item. The idea is to report at any time the item with the highest priority in the window as the sample item. Uniformity then follows because each item has the same probability of having the highest priority.

To maintain the highest-priority item incrementally, PS(1) makes use of a chain of elements of increasing timestamp and decreasing priority. The sample item can always be found at the beginning of the chain, the other items replace the sample item upon its expiration. The algorithm is as follows. Suppose that item $e_i = (i, t_i, r_i)$ with priority p_i arrives in the stream. Then, we remove all items with a timestamp less than or equal to $t_i - \Delta$ from the beginning of the chain. These items have expired so that they can be discarded. We also remove from the end of the chain all items from the end of the chain that have a priority of less than p_i. These items cannot become the highest-priority item anymore because e_i is "stronger": it has a higher priority and expires at a later point in time. Note that with probability $1/N(t)$, priority p_i is the highest in the window and all items are removed from the chain. As a final step, e_i is appended to the end of the chain. The scheme maintains the invariant that, at any time, all and only the window items for which there does not exist an item with both a larger timestamp and a higher priority are present in the chain. In order to be able to always report the highest-priority item—even when no items subsequently arrive in the stream—, knowledge of these items is both necessary and sufficient.

Babcock et al. (2002) have shown that, at an arbitrary point in time t, the scheme requires $O(\log N(t))$ space in expectation. The actual space requirement is also $O(\log N(t))$ with high probability. Thus, the space consumption of priority sampling cannot be bounded from above. When priority sampling is applied to a sequence-based window ($t_i = i$ and $\Delta = N$), the scheme requires more space than chain sampling in expectation ($O(\log N)$ vs. $O(1)$). The reason is that chain sampling exploits the knowledge of constant window size, while priority sampling does not.

E. Priority Sampling Without Replacement, PSWOR(M)

Priority sampling can be modified to sample without replacement. The resulting scheme, denoted PSWOR(M), has been mentioned in Gemulla and Lehner (2008).

[17] An estimator for $N(t)$ is given in chapter 7. This estimator can be used to determine scale-up factors; it is not suited to drive the sampling process.

The idea is straightforward: Instead of maintaining only the highest-priority item in the window, we maintain the items with the M highest priorities. In order to maintain these M items incrementally, we store each arriving item as long as there are fewer than M more recent items with a higher priority. One can show that the space consumption is still $O(M \log N(t))$ in expectation, but efficient maintenance of the chain becomes challenging.

We give a semi-naive implementation that requires $O(N(t))$ time per item. In the implementation, items are arranged in M "levels" labeled $0, \ldots, M-1$. An item is stored at level l if exactly l more recent items with a higher priority have arrived. Now suppose that item e_i arrives in the stream. We first compare e_i to every item stored in the data structure. Suppose that e_i is compared to item e_j having level l. If e_i has a higher priority than e_j, we promote e_j to level $l+1$. Item e_j is removed from the data structure either if it has expired or if it is promoted while being at level $M-1$. After all items have been processed, item e_i is inserted at level 0.

The semi-naive scheme can be improved by maintaining a per-level index over the priorities of the items. This way, one does not need to compare e_i to each and every item stored in the data structure. An interesting open problem is to estimate the maintenance cost of this improved scheme.

F. Summary and Roadmap

Bernoulli samples can be maintained from both sequence-based and time-based windows using the BERNW(q) scheme.

Again, bounded-size sampling is more complex. For sequence-based sampling, the PASSIVE(M) scheme is clearly the best choice when inter-sample dependencies are acceptable. This is because it is simple, efficient, and it has a fixed footprint. Otherwise, when inter-sample dependencies have to be minimized, CHAIN(M) is the method of choice because it has a lower expected space overhead than its competitors PS(M) and PSWOR(M). For time-based windows, PS(M) and PSWOR(M) are the only options. Although both schemes maintain a fixed-size sample, they have an unbounded footprint.

In chapter 7, we introduce a novel with-replacement sampling scheme for time-based windows called bounded priority sampling, BPS(M). We also propose a variant of the scheme that samples without replacement, BPSWOR(M). Both schemes are summarized in table 3.7. As the names suggest, these schemes are built upon priority sampling but, in contrast to priority sampling, they have a bounded footprint. The samples produced by both the BPS(M) and BPSWOR(M) schemes have a random size. This is unavoidable because—as we will prove—it is impossible to guarantee a minimum sample size in bounded space. We can, however, give a lower bound on the expected sample size at each point of time. We also propose a stratified sampling scheme for time-based windows that leads to larger samples but can only be used when uniformity is not required by the application. In these applications, the stratified scheme is a compelling alternative to BPS(M) and BPSWOR(M).

Chapter 4

Set Sampling

In this chapter,[1] we develop algorithms that can be used to maintain a bounded sample of an evolving set, where boundedness refers to both sample size and sample footprint. The main challenges in this setting are (i) to enforce statistical uniformity, (ii) to avoid accesses to the base data to the extent possible, because such accesses are typically expensive (see section 3.3.3A), and (iii) to maximize sampling *efficiency*, i.e., to keep the sample size as close to the upper bound as possible. In addition to sample maintenance, we also discuss methods for reducing or increasing the sample size dynamically as well as methods for merging two or more samples into a single sample of the union of their underlying datasets.

In section 4.1, we provide a novel sampling scheme, called *random pairing*, RP(M), that maintains a bounded, provably uniform sample in the presence of arbitrary insertions, updates and deletions. RP(M) can be viewed as a generalization of the modified reservoir sampling scheme and the passive sampling scheme discussed in chapter 3. In contrast to previous schemes that handle deletions, RP(M) does not require access to the base data. Provided that fluctuations in the dataset size are not too extreme, the sample sizes produced by RP(M) are as stable as those produced by alternative algorithms that require base-data accesses. Thus, if the dataset size is reasonably stable over time, RP(M) is the algorithm of choice for incrementally maintaining a bounded uniform sample.

In section 4.2, 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. Such methods are of interest for growing datasets because they can be exploited to grow the sample in a controlled manner, whenever needed. 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 called *RPRES* that partially enlarges the sample using the base data, and subsequently completes the resizing using only the stream of insertion, update and deletion transactions. Especially when access to the base data is expensive and transactions are frequent, the resizing cost can be significantly reduced relative to the

[1] Parts of the material in this chapter have been developed jointly with Peter J. Haas and Wolfgang Lehner. The chapter is based on Gemulla et al. (2006) and Gemulla et al. (2008). The copyright of Gemulla et al. (2006) is held by the VLDB Endowment; the original publication is available at http://portal.acm.org/citation.cfm?id=1164179. The copyright of Gemulla et al. (2008) is held by Springer; the original publication is available at www.springerlink.com and http://dx.doi.org/10.1007/s00778-007-0065-y.

naive approach by judiciously tuning the key algorithm parameter d; 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. Finally, we give a subsampling algorithm termed *RPSUB* that can be used to reduce the size of the sample; the algorithm is useful to handle potential memory bottlenecks and to undo prior sample enlargements.

Algorithms for merging two or more samples are discussed in section 4.3. These algorithms are particularly useful when the dataset is partitioned over several nodes. In this case, sample merging can be used 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 MERGE algorithm of Brown and Haas (2006)—which was developed for an insertion-only environment—to effectively deal with deletions. We show analytically that RPMERGE produces larger sample sizes than MERGE in expectation; the merged sample thus contains more information about the complete dataset. Additionally, when RPMERGE is used, the merged sample can be maintained incrementally using the random pairing algorithm.

4.1 Uniform Sampling

In this section, we provide the details of the random pairing algorithm. We present a proof of its correctness, analyze its sample size properties, show how a skip counter can be incorporated, and present the results of our experimental evaluation.

We assume throughout that the sample fits in main memory. This assumption limits, in terms of efficiency, 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 the discussion in chapter 8. 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.

4.1.1 Random Pairing

As before, denote by M the sample size parameter and recall the "obvious" extension of MRS(M) to support deletion transactions (section 3.5.1D). The idea of the obvious algorithm is to compensate sample deletions–which may cause the sample size to fall below M—by refilling the sample with newly inserted items. Unfortunately, the algorithm does not produce a uniform sample because items inserted after deletions have an overly high probability of being sampled. The basic idea behind RP(M) is to carefully select an inclusion probability for each inserted item so as to ensure uniformity.

In the following, we do not explicitly discuss update transactions; they are treated as in MRS(M), that is, the updated item is also updated in the sample if present. For brevity, we will frequently omit the sample size parameter and refer to RP(M)

simply as RP; the parameter M is treated as a constant chosen before the sampling process starts.

A. Algorithmic 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 $d = 0$, that is, there are no uncompensated deletions, then insertions are processed as in standard reservoir sampling. If $d > 0$, 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 4.1.

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. 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. In fact, the probability that the partner was in the sample matches precisely the probability $c_b/(c_b + c_g)$ used by RP.

Let's repeat the calculation in (3.14) that was used to show that the obvious algorithm is incorrect. Suppose that RP(M) has been used to obtain a sample S of dataset R, with $|R| > M$, produced by a transaction sequence that does not contain any deletions ($d = 0$). Since RP(M) and RS(M) coincide when no deletions occurs, sample S is uniform and $|S| = M$. Now, suppose that item r^- is deleted from R and item r^+ is subsequently inserted into R. Denote by S' the state of sample after these two operations have been processed by RP(M). We have

$$\begin{aligned}\Pr\left[r^+ \in S'\right] &= \Pr\left[r^- \in S,\ r^+ \text{ included}\right] + \Pr\left[r^- \notin S,\ r^+ \text{ included}\right] \\ &= \frac{M}{N}\cdot\frac{1}{1} + \left(1 - \frac{M}{N}\right)\cdot\frac{0}{1} = \frac{M}{N},\end{aligned}$$

as desired. A complete proof of the correctness of RP is given in section C.

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

Algorithm 4.1 Random pairing (basic version)

```
1: c_b: number of uncompensated deletions that have been in the sample
2: c_g: number 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(r):
8: if c_b + c_g = 0 then                          // execute reservoir-sampling step
9:    if |S| < M then
10:      insert r into S
11:   else if RANDOM() < M/(|R| + 1) then
12:      overwrite a randomly selected element of S with r
13:   end if
14: else                                          // execute random-pairing step
15:   if RANDOM() < c_b/(c_b + c_g) then
16:      c_b ← c_b − 1
17:      insert r into S
18:   else
19:      c_g ← c_g − 1
20:   end if
21: end if
22:
23: DELETE(r):
24: if r ∈ S then
25:   c_b ← c_b + 1
26:   remove r from S
27: else
28:   c_g ← c_g + 1
29: end if
```

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,

$$c_{\mathrm{b}} = \min(M, |R| + d) - |S| \quad \text{and} \quad c_{\mathrm{g}} = d - c_{\mathrm{b}}. \tag{4.1}$$

To see this, observe that $\min(M, |R| + d)$ corresponds to the maximum sample size seen so far; the number of bad deletions is just the difference to the current sample size $|S|$.

B. Example

The RP(M) algorithm with $M = 2$ is illustrated in figure 4.1. 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 r_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 r_2 and r_3 from both the dataset and the sample. Thus, at $i = 5$, there are two uncompensated deletions. The insertion of r_4 triggers the execution of a *pairing step*. Item r_4 is conceptually paired with either r_3 or r_2—these scenarios are denoted by (a) and (b) respectively—and each of these pairings is equally likely. Thus r_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 r_4 with probability $c_{\mathrm{b}}/(c_{\mathrm{b}} + c_{\mathrm{g}})$ and excluding r_4 with probability $c_{\mathrm{g}}/(c_{\mathrm{b}} + c_{\mathrm{g}})$, 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 r_5, but this time there is only one uncompensated deletion left: r_2 in scenario (a) or r_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.

C. Proof of Correctness

We now formally establish the uniformity property of the RP(M) scheme with upper bound $M \geq 1$; we actually prove a slightly stronger result that implies uniformity. As before, denote by R_i the dataset and by S_i sample after the i-th processing step,

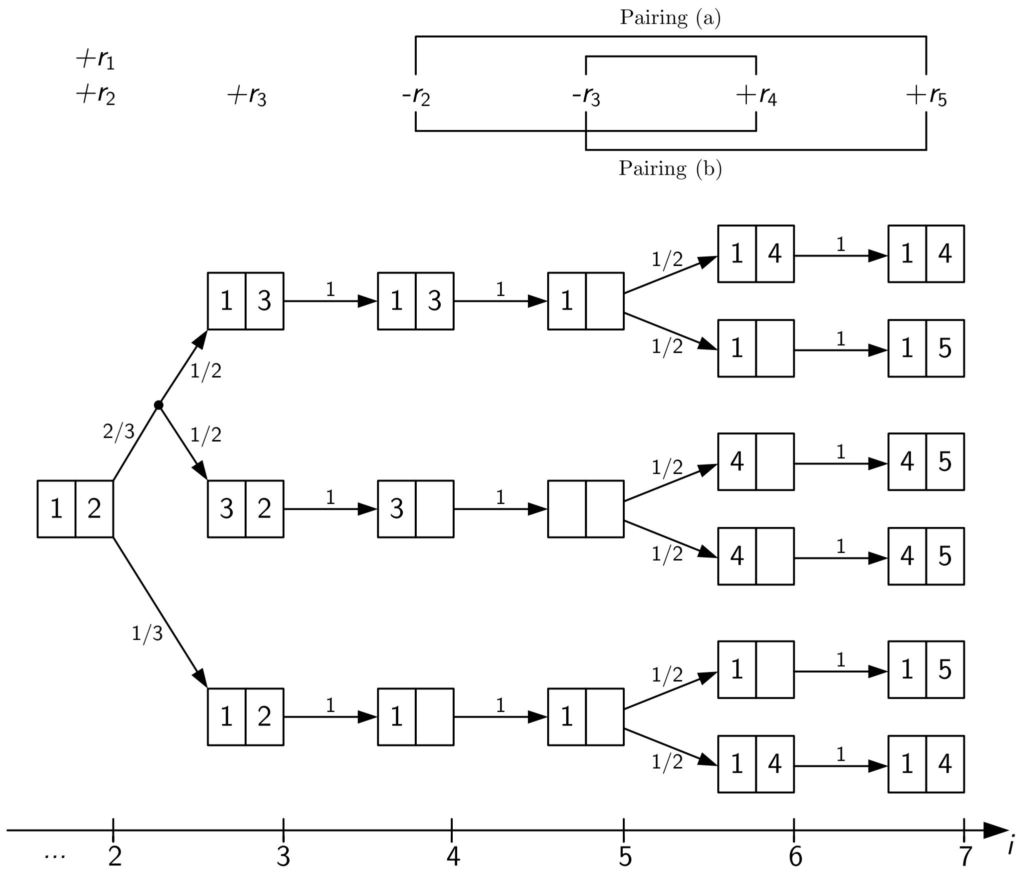


Figure 4.1: Illustration of random pairing

i.e., after processing transaction γ_i. Also denote by $c_{\mathrm{b},i}$ and $c_{\mathrm{g},i}$ the value of the counters c_b and c_g after the i-th step, and set $d_i = c_{\mathrm{b},i} + c_{\mathrm{g},i}$. Finally, set

$$\begin{aligned} v_i &= \min\bigl(M, \max_{1\le j\le i} |R_j|\bigr) = \min(M, |R_i| + d_i), \\ l_i &= \max(0, v_i - d_i), \\ u_i &= \min(M, |R_i|). \end{aligned} \tag{4.2}$$

In light of (4.5) below, it can be seen that l_i and u_i are the smallest and largest possible sample sizes after the i-th step, and v_i 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 4.1. *For any feasible sequence γ of insertions and deletions, there exist numbers $\{\, p_i(k) \colon i \ge 1$ and $k \ge 0 \,\}$, depending on γ, such that*

$$\Pr[\, S_i = A \,] = p_i(|A|) \tag{4.3}$$

for $A \subseteq R_i$ and $i \ge 1$. Moreover,

$$\frac{p_i(k)}{p_i(k-1)} = \frac{v_i - k + 1}{d_i - v_i + k}. \tag{4.4}$$

for $i \ge 1$ and $k \in \{\, l_i + 1, l_i + 2, \ldots, u_i \,\}$.

It follows from (4.3) 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 $p_i(k) = 0$ for $i \ge 1$ and $k \notin \{\, l_i, l_i + 1, \ldots, u_i \,\}$. 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 $|S| \le M$.) It follows directly that

$$c_{\mathrm{b},i} = v_i - |S_i| \tag{4.5}$$

for $i \ge 1$. The proof now proceeds by induction on i. The assertions of the theorem clearly hold for $i = 1$, so suppose for induction that the assertions hold for values $1, 2, \ldots, i-1$. There are two cases to consider. First, suppose that step i corresponds to the insertion of an item r, and consider a subset $A \subseteq R_i$ with $|A| = k$, where $l_i \le k \le u_i$. If $d_{i-1} = 0$, then $d_i = 0$ and $l_i = u_i$, so that (4.4) holds vacuously, and the correctness proof for standard reservoir sampling—see, e.g., Haas (2009)—establishes the assertion in (4.3). So assume in the following that $d_{i-1} > 0$. If $r \in A$, then, using (4.5), we have

$$\begin{aligned} \Pr[\, S_i = A \,] &= \Pr[\, S_{i-1} = A \setminus \{\, r \,\},\ r \text{ included} \,] \\ &= p_{i-1}(k-1)\frac{c_{\mathrm{b},i-1}}{d_{i-1}} = p_{i-1}(k-1)\frac{v_{i-1} - k + 1}{d_{i-1}}. \end{aligned}$$

If $r \notin A$, then

$$\begin{aligned} \Pr[\, S_i = A \,] &= \Pr[\, S_{i-1} = A,\ r \text{ ignored} \,] \\ &= p_{i-1}(k) \frac{d_{i-1} - v_{i-1} + k}{d_{i-1}}, \end{aligned} \tag{4.6}$$

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

$$\Pr[\, S_i = A \,] = p_{i-1}(k-1) \frac{v_{i-1} - k + 1}{d_{i-1}}. \tag{4.7}$$

Here (4.7) follows from (4.6) and an inductive application of (4.4). This establishes the first assertion of the theorem with

$$p_i(k) = \begin{cases} p_{i-1}(k-1) \frac{v_{i-1}-k+1}{d_{i-1}} & \text{if } \max(l_i, 1) \leq k \leq u_i; \\ p_{i-1}(0) \frac{d_{i-1}-v_{i-1}}{d_{i-1}} & \text{if } k = l_i = 0; \\ 0 & \text{otherwise.} \end{cases} \tag{4.8}$$

To establish the second assertion of the theorem, apply (4.8) and then inductively apply (4.4), making use of the fact that—since $d_{i-1} > 0$ and an item is inserted at step i—we have $d_i = d_{i-1} - 1$ and $v_i = v_{i-1}$. Now suppose that step i corresponds to the deletion of an item r, and again consider a subset $A \subseteq R_i$ with $|A| = k \in \{\, l_i, l_i + 1, \ldots, u_i \,\}$. Observe that

$$\begin{aligned} \Pr[\, S_i = A \,] &= \Pr[\, S_{i-1} = A \,] + \Pr[\, S_{i-1} = A \cup \{\, r \,\} \,] \\ &= p_{i-1}(k) + p_{i-1}(k+1), \end{aligned}$$

which establishes the first assertion of the theorem with

$$p_i(k) = p_{i-1}(k) + p_{i-1}(k+1) \tag{4.9}$$

for $l_i \leq k \leq u_i$. Since $d_i = d_{i-1} + 1$ and $v_i = v_{i-1}$, we then have

$$\begin{aligned} \frac{p_i(k)}{p_i(k-1)} &= \frac{p_{i-1}(k) + p_{i-1}(k+1)}{p_{i-1}(k-1) + p_{i-1}(k)} \\ &= \frac{\big[p_{i-1}(k)/p_{i-1}(k-1)\big] + \big[p_{i-1}(k+1)/p_{i-1}(k-1)\big]}{1 + \big[p_{i-1}(k)/p_{i-1}(k-1)\big]} \\ &= \frac{v_{i-1} - k + 1}{d_{i-1} - v_{i-1} + k + 1} = \frac{v_i - k + 1}{d_i - v_i + k} \end{aligned}$$

for $l_i < k \leq u_i$, where we have again inductively used (4.4). Thus the second assertion of the theorem holds and the proof is complete. □

Observe that RP(M) reduces to PASSIVE(M) when applied to a sequence-based sliding window over a data stream. If there are no deletions, RP(M) scheme reduces to standard RS(M).

D. Sample Size Properties

Building on theorem 4.1, we can derive the statistical properties of the sample size at any given time point. In theorem 4.2 below, we establish the probability distribution, the mean, and the variance of the sample size. As before, see (4.2), we define l_i and u_i to be the smallest and largest possible sample size after processing the i-th transaction, and v_i to be the largest sample size encountered so far.

As a shortcut, denote by

$$H\big(k;\, N, N', M\big) = \binom{N'}{k}\binom{N - N'}{M - k} \Big/ \binom{N}{M} \tag{4.10}$$

a hypergeometric probability. Intuitively, $H(k;\, N, N', M)$ denotes the probability that when sampling M out of N' white and $N - N'$ black balls, exactly k of them are white. In the theorem below, the white balls correspond to the items in R_i, the black balls correspond to uncompensated deletions.

Theorem 4.2. *For any feasible sequence γ of insertions and deletions and $i \geq 1$, the sample size follows the hypergeometric distribution given by*

$$\Pr\left[\, |S_i| = k \,\right] = H(k;\, |R_i| + d_i, |R_i|, v_i) \tag{4.11}$$

for $l_i \leq k \leq u_i$, and $\Pr\left[\, |S_i| = k \,\right] = 0$ otherwise. Moreover, the expected value and variance of $|S_i|$ are given by

$$\mathrm{E}\left[\, |S_i| \,\right] = \frac{|R_i|}{|R_i| + d_i} v_i$$

and

$$\mathrm{Var}\left[\, |S_i| \,\right] = \frac{d_i v_i (|R_i| + d_i - v_i)|R_i|}{(|R_i| + d_i)^2 (|R_i| + d_i - 1)}.$$

Proof. Defining $p_i(k)$ as in theorem 4.1 and appealing to (4.3), we have

$$\Pr\left[\, |S_i| = k \,\right] = \sum_{\substack{A \subseteq R_i \\ |A| = k}} \Pr\left[\, S_i = A \,\right] = \binom{|R_i|}{k} p_i(k),$$

and it suffices to show that

$$p_i(k) = \binom{d_i}{v_i - k} \Big/ \binom{|R_i| + d_i}{v_i} \tag{4.12}$$

for $l_i \leq k \leq u_i$ and $i \geq 1$. Clearly, (4.12) holds for $i = 1$, and can be established for general $i > 1$ by a straightforward inductive argument that uses (4.8) and (4.9). The remaining assertions of the theorem follow from well-known properties of the hypergeometric distribution (Johnson et al. 1992, p. 250). □

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 "transient item" in the dataset (and in the sample, if present) until the deleted item is compensated. After processing the i-th transaction, the dataset contains $|R_i| + d_i$ total items, comprising $|R_i|$ real items and d_i transient items. We can view the current sample as a uniformly selected subset of size v_i from the collection of $|R_i| + d_i$ total items. The actual sample size $|S_i|$ is simply the number of these v_i items that are real. As mentioned above, the probability that a uniform sample of v_i items from a population of $|R_i|$ real items and d_i transient 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 4.2 that the sample size is concentrated around 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 $98{,}900 \leq |S| \leq 99.100$ with a probability of approximately 99.8%. Moreover, when the number of uncompensated deletions is small, the expected sample size is close to the maximum possible sample size.

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

4.1.2 Random Pairing With Skipping

The RP algorithm, as displayed in algorithm 4.1, calls the RANDOM function at essentially every insertion transaction. In practice, random numbers are generated using a pseudorandom number generator (PRNG); see L'Ecuyer (2006) 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 4.1, 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, we can leverage results on skip-based sampling to produce a more efficient algorithm. Note that in contrast to the skip-based survey sampling schemes of section 3.4, we can not simply skip over arriving transactions when RP is used. This is because RP proceeds differently for insertion and deletion transactions and therefore must access every transaction to distinguish between both cases. We can, however, reduce the cost of generating random numbers so that RP becomes more CPU-efficient.

A. Reservoir Step

Assume for a moment that γ consists only of insertion transactions. In this setting, RP coincides with reservoir sampling so that we can make use of the skip counters for reservoir sampling. Recall that skip-based reservoir sampling makes use of a random variable Z_i that denotes the number of rejected items until the next sample insertion; Z_i is defined in equation (3.11). When processing $\gamma_{i+1} = +r$, item r is included into the sample if and only if $Z_i = 0$. In this case, Z_{i+1} is regenerated; otherwise, we set $Z_{i+1} \leftarrow Z_i - 1$ and ignore item r.

The same idea can be exploited for arbitrary sequences in which the insertion process may be interrupted by deletion transactions. In fact, we can continue to use the skip counter without modification as soon as the next reservoir step is executed. To see this, suppose that there are no uncompensated deletions after processing transaction γ_i and that γ_{i+1} is a deletion; we have $d_i = 0$ 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 $|R_i| = |R_{i^*}|$ by theorem 4.2, we have $\Pr[\, Z_{i+1} = k \,] = \Pr[\, Z_{i^*+1} = k \,]$ according to (3.11), and the skip-based reservoir sampling process can be continued at the point where it was interrupted.

Algorithm 4.2 incorporates these optimizations into the basic RP algorithm. The skip counter is denoted by Z. When $Z \geq 0$, the variable contains the number of insertions to skip; a value of $Z < 0$ indicates that the skip counter has to be regenerated. The function SkipRS takes care of regeneration; it can be implemented efficiently using on of the algorithms of Vitter (1985).

B. Pairing Step

We can exploit an idea similar to the one above to optimize the pairing step. Assume that after processing transaction γ_i, there are $d_i > 0$ uncompensated deletions and that transactions $\gamma_{i+1}, \gamma_{i+2}, \ldots, \gamma_{i+d_i}$ all correspond to insertions. As before, denote by $c_{\mathrm{b},i}$ and $c_{\mathrm{g},i}$ the values of the sample counters after processing transaction γ_i, where $c_{\mathrm{b},i} + c_{\mathrm{g},i} = d_i$. We argue that out of the d_i insertions, the items that RP includes into the sample form a uniform random sample of size $c_{\mathrm{b},i}$. To see this, observe that the item corresponding to insertion transaction γ_{i+1} is included in the sample with probability $c_{\mathrm{b},i}/d_i$ and excluded otherwise. In case of inclusion, the next item, corresponding to γ_{i+2}, is included with probability $c_{\mathrm{b},i+1}/(c_{\mathrm{b},i+1} + c_{\mathrm{g},i+1}) = (c_{\mathrm{b},i} - 1)/(d_i - 1)$. In case of exclusion, the next item is included with probability $c_{\mathrm{b},i+1}/(c_{\mathrm{b},i+1} + c_{\mathrm{g},i+1}) = c_{\mathrm{b},i}/(d_i - 1)$, and so on. These probabilities match the inclusion probabilities used in the list-sequential schemes of section 3.4.2 with sample size $c_{\mathrm{b},i}$ and population size d_i: the numerator of the inclusion probability denotes the remaining sample items, the denominator equals the remaining population items.

We make use of the improved methods for list-sequential sampling in the RP algorithm by maintaining a skip counter Z' for the pairing step; Z' is distributed as given in equation (3.6). Unlike with the counter Z for the reservoir step, we have to

Algorithm 4.2 Random pairing (optimized version)

```
1:  c_b: number of uncompensated deletions that have been in the sample
2:  c_g: number 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:  Z: skip counter for reservoir sampling (initialized to −1)
6:  Z′: skip counter for random pairing (initialized to −1)
7:  SkipRS: reservoir-sampling skip function as in Vitter (1985)
8:  SkipSeq: list-sequential-sampling skip function as in Vitter (1984)
9:
10: Insert(r):
11: if c_b + c_g = 0 then                    // execute optimized reservoir-sampling step
12:   if |S| < M then
13:     insert r into S
14:   else
15:     if Z < 0 then
16:       Z ← SkipRS(M, |R| + 1)
17:     end if
18:     if Z = 0 then
19:       overwrite a randomly selected element of S with r
20:     end if
21:     Z ← Z − 1
22:   end if
23: else                                     // execute optimized random-pairing step
24:   if Z′ < 0 then
25:     Z′ = SkipSeq(c_b, c_b + c_g)
26:   end if
27:   if Z′ = 0 then
28:     c_b ← c_b − 1
29:     insert r into S
30:   else
31:     c_g ← c_g − 1
32:   end if
33:   Z′ ← Z′ − 1
34: end if
35:
36: Delete(r):
37: Z′ ← −1                                  // invalidate
38: if r ∈ S then
39:   c_b ← c_b + 1
40:   remove r from S
41: else
42:   c_g ← c_g + 1
43: end if
```

recompute Z' whenever a deletion transaction interrupts the insertion process.[2] The complete procedure is given in algorithm 4.2. Again, a negative value of Z' indicates that recomputation is required. SkipSeq generates the skip counter using one of the algorithms of Vitter (1984). 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 section 4.1.3D for an empirical evaluation of our proposed optimizations.

4.1.3 Experiments

We conducted an experimental study to evaluate the stability and performance of the RP scheme with respect to the various bounded-size algorithms mentioned in Section 3.5.1. 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, we found that

- The optimization of the reservoir step 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 optimization of the pairing step 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 or when the dataset is stable and the transaction stream contains long sequences of insertions.

A. Setup

We implemented the RP algorithm, as well as the CAR, CARWOR, MRSR, and MRST schemes, using Java 1.6. We employed an indexed in-memory array to efficiently support the deletion of items. Since the order of the items within the sample is commonly of no interest, we maintained a dense array, i.e., if an item gets deleted, we moved the last item of the sample to its position. This avoids maintaining a data structure to capture free slots within the array.

[2] More precisely, it suffices to recompute Z' when the insertion process resumes. Suppose that Z' is recomputed while processing insertion γ_{i+1}: we set $Z' = Z_{1,0}$ where $Z_{1,0}$ follows the distribution in (3.6) with $k = c_{\mathrm{b},i}$ and $N = d_i$.

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 thesis 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.

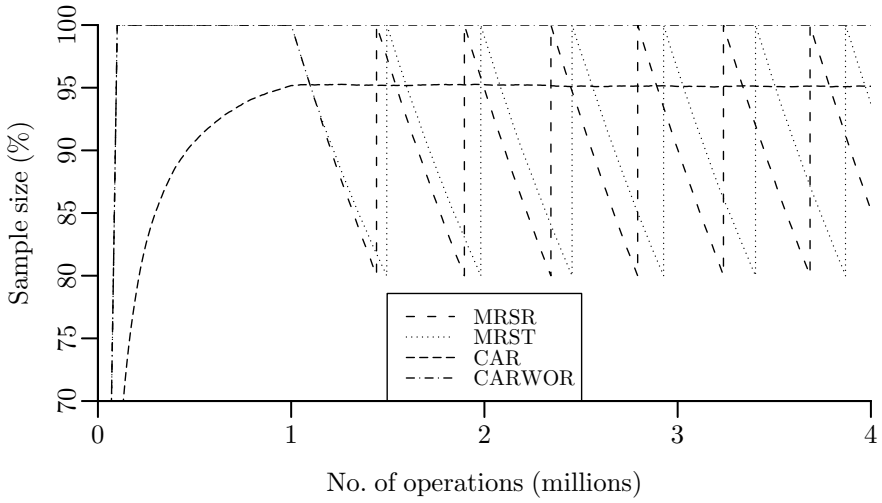
We assumed that the deletions and insertions are clustered into batches of b operations, and simulated the sequence of dataset operations by randomly deciding whether the next b operations are insertions or deletions. Our default value was $b = 1$, but we also ran experiments in which we systematically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

B. Sample Size

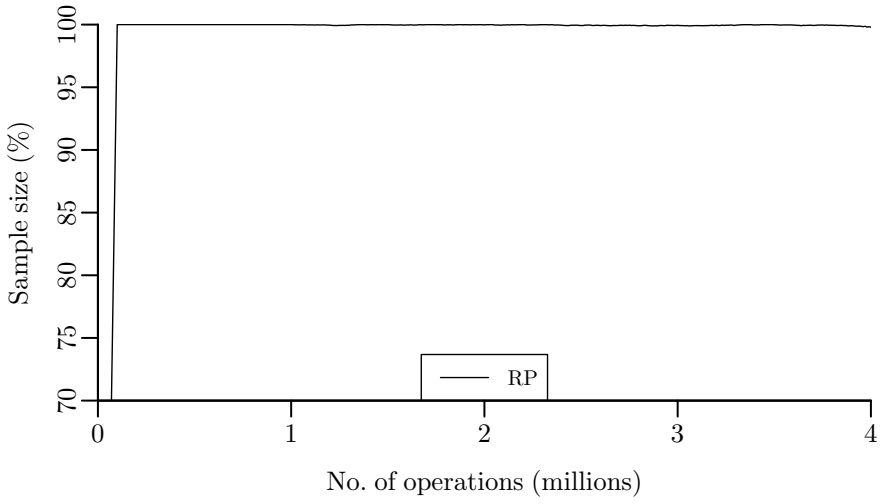
We evaluated the sample-size stability for the various algorithms by executing a randomly generated sequence of 4,000,000 insertion/deletion operations while incrementally maintaining a sample with a target size (and upper bound) of 100,000 items. To create a scenario in which the dataset of interest 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 MRSR and MRST algorithms. 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. The upper part of figure 4.2 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, CARWOR is optimal, since it is able to maintain the sample at its upper bound. CAR produces smaller samples after duplicate elimination, but the sample size is stable and close to the upper bound. Both algorithms, however, need to access the base data. MRSR and MRST also need to access the base data, but the sample size is less stable than that of CAR or CARWOR; it fluctuates in the range $[0.8M, M]$. MRST is slightly superior to MRSR in that the sample size decrease is slower. We see that the sample sizes produced by RP are almost indistinguishable from those of 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 figure 4.3. Again, RP performs comparably to CARWOR, in that it maintains a sample size close to the upper bound



(a) Weakly incremental schemes



(b) Strongly incremental schemes

Figure 4.2: Evolution of sample size over time

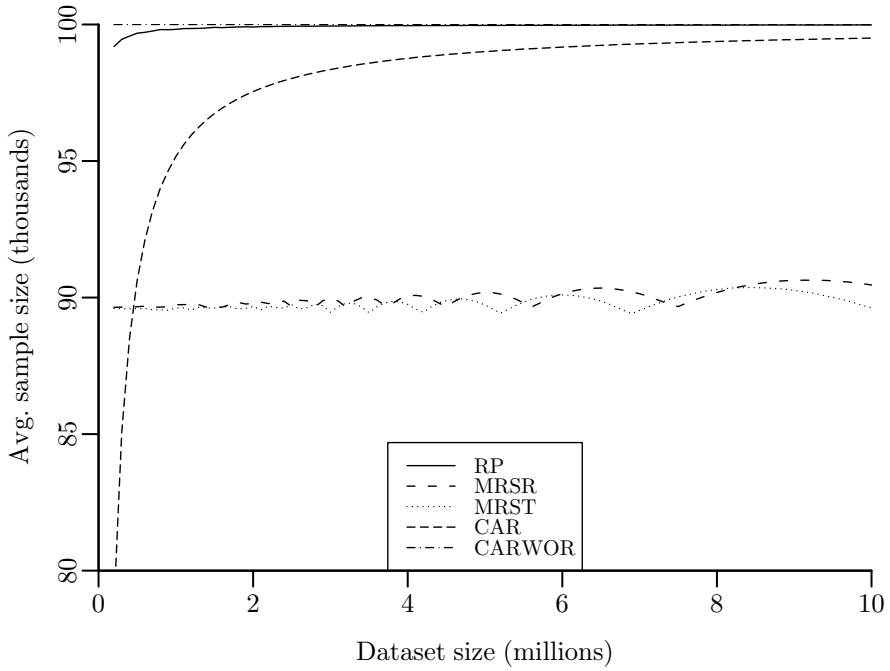


Figure 4.3: Dataset size and average sample size

M. CAR is almost as effective when the dataset size is large so that few duplicates are expected in the sample. In contrast, the time-average sample sizes for MRSR and MRST are smaller than those of the other algorithms. The reason for this behavior is that the both algorithms 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 figure 4.4.

As can be seen, the sample sizes produced by algorithms that access base data are almost independent of the cluster size, whereas those produced by RP depend on the cluster size; the higher the variance of the dataset size, the lower the average sample size. Due to high peaks in the 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

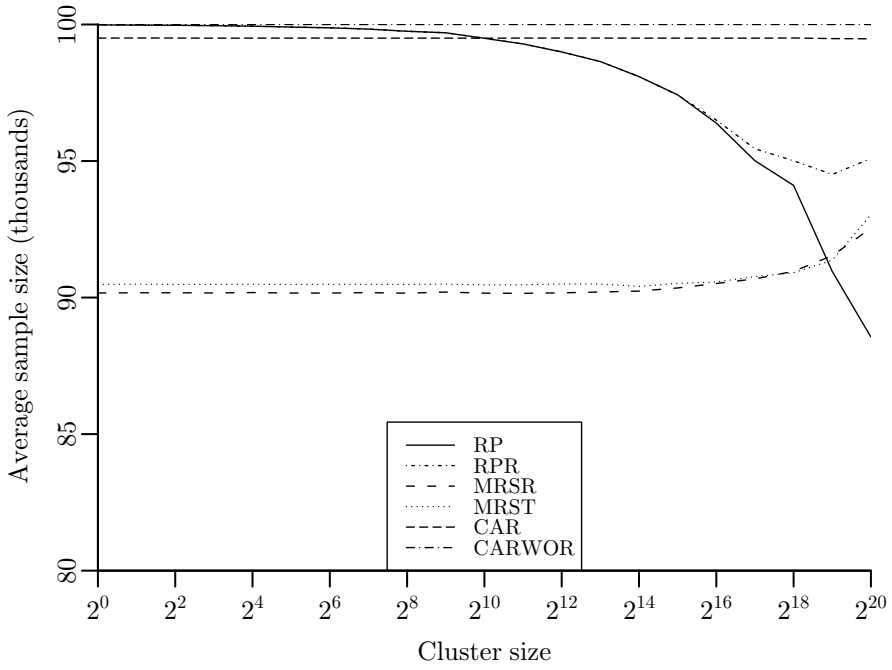


Figure 4.4: Cluster size and average sample size

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 figure 4.5. MRSR and MRST perform worst since they are expensive and produce 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} (not shown), the sample is recomputed from scratch after almost every deletion block, and RPR reduces to MRSR.

C. 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 4.6 depicts the number of accesses to base data for the different algorithms. Because they must periodically recompute the entire sample, MRSR and MRST

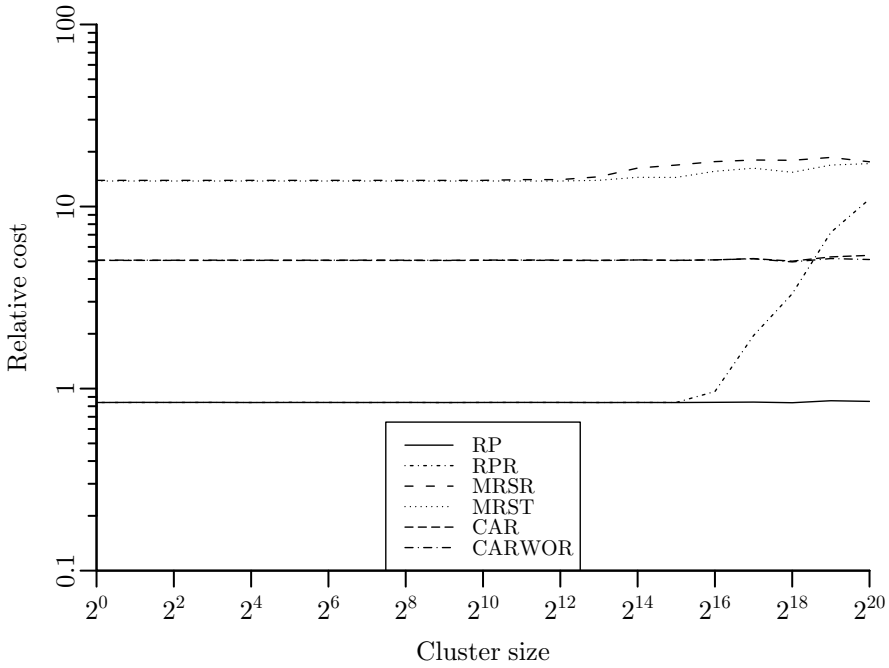


Figure 4.5: Cluster size and relative cost

require more base-data accesses than any other sampling scheme. Both CAR and CARWOR perform better than MRSR, with CARWOR incurring slightly 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 the 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.

Figure 4.7 shows the number of write accesses to the sample for the different sampling schemes. Again, MRSR and MRST are the least efficient algorithms, because every recomputation completely flushes the current sample and recomputes it using base data. The other algorithms perform comparably; indeed, the curves for CAR, CARWOR, and RP coincide.

Fig. 4.8 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.

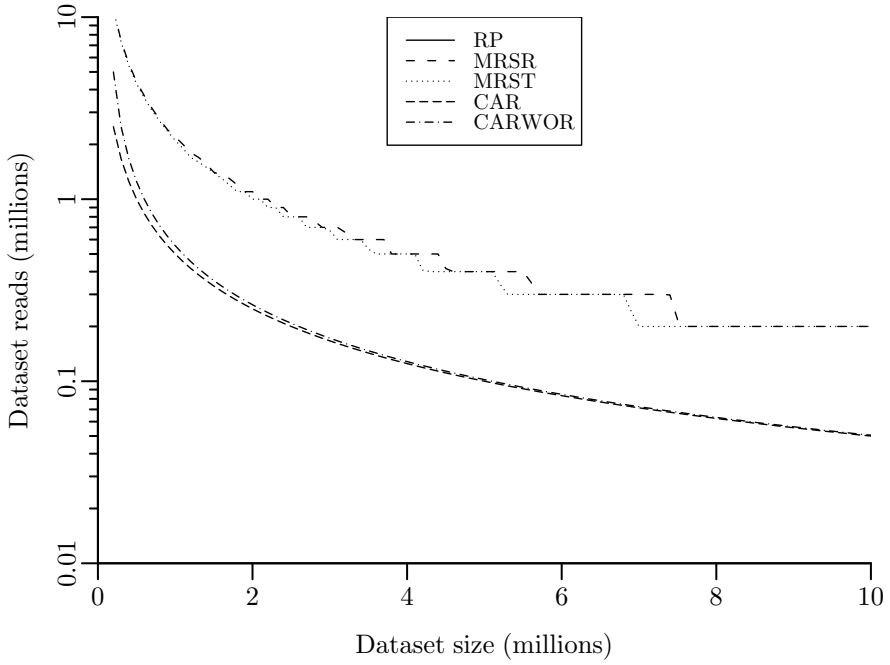


Figure 4.6: Number of dataset reads

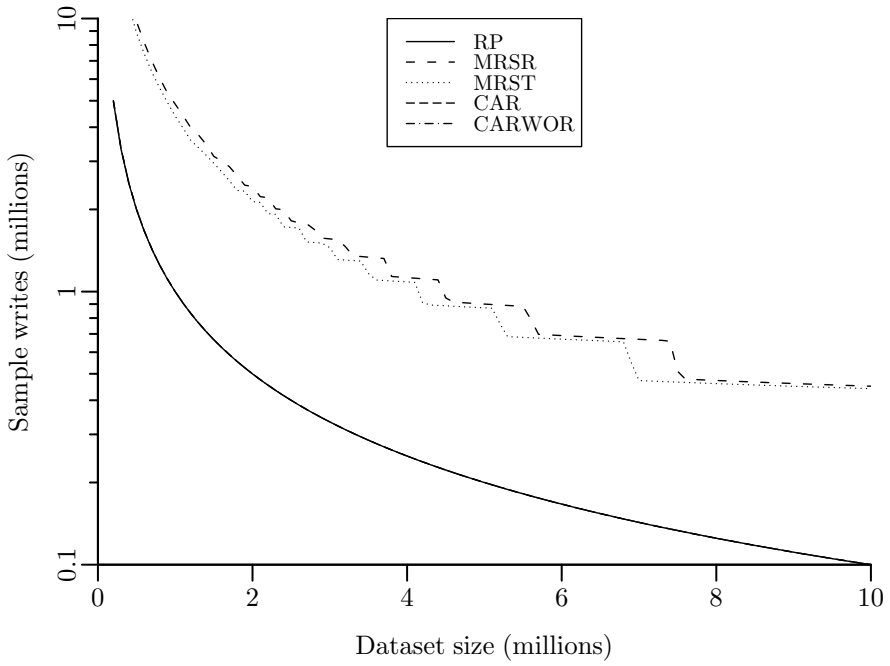


Figure 4.7: Number of sample writes

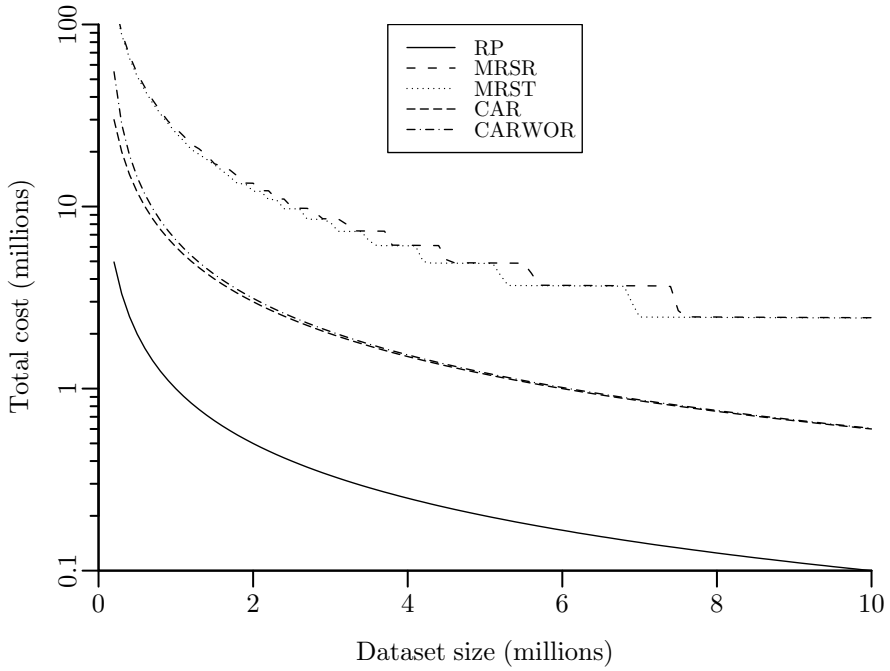


Figure 4.8: Combined cost

D. 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 section 4.1.2, with the optimization of the 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 $1.1\mu s$, the replacement of an item by another one $2.2\mu s$, the look-up cost is $0.9\mu s$ and the removal of an item takes $3.3\mu s$. To generate random numbers, we used the "Mersenne Twister" of Matsumoto and Nishimura (1998); each random number takes approximately $0.6\mu s$ to produce.[3] 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.

[3] Surprisingly, the weaker PRNG shipped with Sun's JDK requires $0.7\mu s$ and is therefore slower.

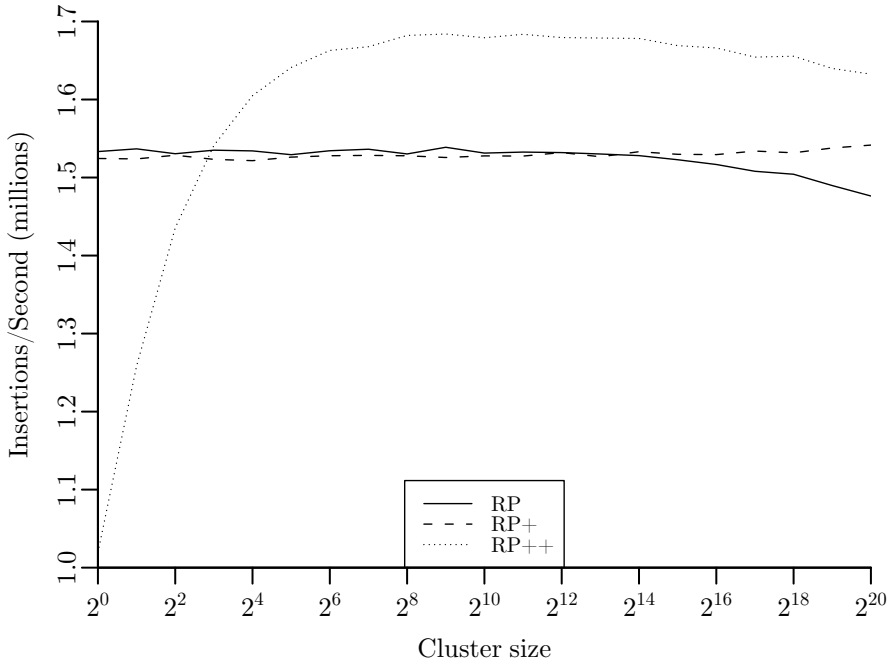


Figure 4.9: Throughput, stable dataset

Figure 4.9 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 Z'. For $b \geq 16$, RP++ performs better than both RP and RP+.[4]

Figure 4.10 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.

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

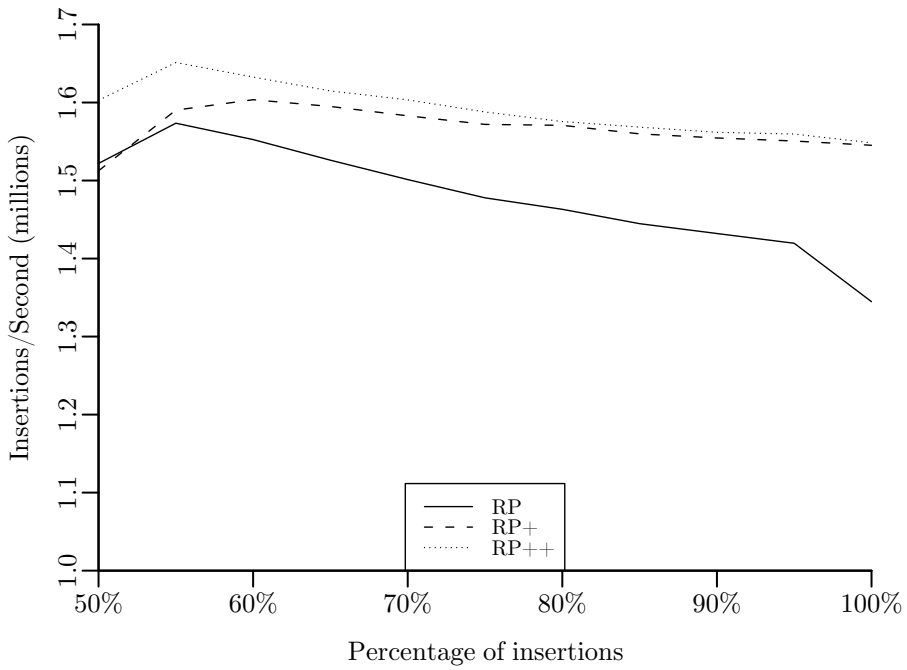


Figure 4.10: Throughput, growing dataset

4.2 Sample Resizing

The discussion so far has focused on stable datasets, and therefore on sampling algorithms that guarantee a fixed upper bound on the sample size. For growing datasets, however, one might want to increase the sample size from time to time in order to avoid undersized samples. As discussed in section 3.5.1, the Bernoulli scheme MBERN(q) 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. Furthermore, we study the related problem of subsampling, that is, the problem of reducing the sample size and its upper bound. Subsampling techniques can be used in practice when, for example, the memory available for the sample decreases.

The outline of this section is as follows: We start with schemes that increase the size of a bounded sample. In section 4.2.1, we show that any such scheme cannot avoid accessing the base data. We then disprove the correctness of an earlier algorithm given in Gemulla et al. (2008) and describe a novel, provably uniform scheme called RPRES. In section 4.2.2, we discuss optimum parametrization of RPRES. Experimental results are presented in section 4.2.3. We conclude this section with a discussion of subsampling in section 4.2.4, where we derive the RPSUB algorithm.

4.2.1 Resizing Upwards

The discussion in this section applies to arbitrary bounded-size uniform samples, not just the ones created by the random pairing algorithm. In general, we consider algorithms that start with a uniform sample S of size 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. Of course, we are interested in schemes that do the resizing in a more efficient manner.

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 4.3 below shows that such algorithms cannot exist.

Theorem 4.3. *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 $\Pr[\, S' = A \,] > 0$. But clearly $\Pr[\, S' = A \,] = 0$, 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. □

B. Non-Uniformity of Bernoulli Resizing

Gemulla et al. (2008) proposed a resizing algorithm called Bernoulli resizing, denoted BERNRES, that makes use of Bernoulli sampling to enlarge the sample. In this section, we show that BERNRES does not produce uniform samples.

We first describe the algorithm. BERNRES has 2 phases. In phase 1, the algorithm converts the size-M sample to an MBERN(q) sample for some arbitrary value $0 < q < 1$, possibly accessing base data in the process. If after the conversion the sample size equals M' or is larger, the algorithm returns M' randomly chosen items as the enlarged sample and bounded-size sampling resumes using the new upper bound M'. Otherwise, BERNRES proceeds to phase 2, in which it uses MBERN(q) to increase the sample size to the target value M' and then resumes bounded-size sampling.

We now give a counterexample in which BERNRES does not produce uniform samples. We consider the transaction sequence $\gamma = (+r_1, +r_2, +r_3)$. Suppose that we have already processed transactions $+r_1$ and $+r_2$ using a sample-size bound of

$M = 1$. We now want to resize the sample to size $M' = 2$ using BERNRES. The first step of BERNRES is to convert the current sample to a MBERN(q) sample. The choice of q is arbitrary so that we set $q = 0.5$ to simplify the formulas. Denoting by S the sample after conversion, we have according to (2.3) and since $R = \{ r_1, r_2 \}$:

$$\Pr[S = \emptyset] = \Pr[S = \{ r_1 \}] = \Pr[S = \{ r_2 \}] = \Pr[S = \{ r_1, r_2 \}] = \frac{1}{4}.$$

There are two cases: (i) $|S| < 2$ and (ii) $|S| = 2$. We discuss both cases separately. Our goal is to determine the distribution of sample S' that results from S after processing transaction $+r_3$. In case (i), BERNRES uses MBERN(0.5) sampling to process $+r_3$, that is, r_3 is added to S with probability 0.5 or ignored otherwise. We therefore have

$$\begin{aligned}
&\Pr\left[S' = \emptyset \mid |S| < 2 \right] = \Pr\left[S' = \{ r_3 \} \mid |S| < 2 \right] \\
&\quad = \Pr\left[S' = \{ r_1 \} \mid |S| < 2 \right] = \Pr\left[S' = \{ r_1, r_3 \} \mid |S| < 2 \right] \\
&\quad = \Pr\left[S' = \{ r_2 \} \mid |S| < 2 \right] = \Pr\left[S' = \{ r_2, r_3 \} \mid |S| < 2 \right] \\
&\quad = \frac{1}{6}.
\end{aligned}$$

We can see that S' conditionally on $|S| < 2$ is not uniform because sample $\{ r_1, r_2 \}$ is chosen with probability 0 while samples $\{ r_1, r_3 \}$ and $\{ r_2, r_3 \}$ are both chosen with probability $1/6 > 0$. For case (ii), where $|S| = 2$, bounded-size sampling recommences so that

$$\begin{aligned}
&\Pr\left[S' = \{ r_1, r_2 \} \mid |S| = 2 \right] = \Pr\left[S' = \{ r_1, r_3 \} \mid |S| = 2 \right] \\
&\quad = \Pr\left[S' = \{ r_2, r_3 \} \mid |S| = 2 \right] = \frac{1}{3}.
\end{aligned}$$

We can now uncondition on S to obtain

$$\begin{aligned}
\Pr[S = \{ r_1, r_2 \}] &= \Pr[|S| < 2] \Pr\left[S' = \{ r_1, r_2 \} \mid |S| < 2 \right] \\
&\quad + \Pr[|S| = 2] \Pr\left[S' = \{ r_1, r_2 \} \mid |S| = 2 \right] \\
&= \frac{3}{4} 0 + \frac{1}{4} \frac{1}{3} = \frac{1}{12} \\
\Pr[S = \{ r_1, r_3 \}] &= \Pr[|S| < 2] \Pr\left[S' = \{ r_1, r_3 \} \mid |S| < 2 \right] \\
&\quad + \Pr[|S| = 2] \Pr\left[S' = \{ r_1, r_3 \} \mid |S| = 2 \right] \\
&= \frac{3}{4} \frac{1}{6} + \frac{1}{4} \frac{1}{3} = \frac{3}{16}.
\end{aligned}$$

Both probabilities differ so that S' does not constitute a uniform random sample of R. The reason for the non-uniformity is that, when $|S| < 2$, S has a different distribution than a real MBERN(q) sample so that MBERN(q) cannot be used in phase 2.

C. Algorithmic Description

We now describe our RPRES algorithm. It is based on BERNRES but avoids the conversion into a Bernoulli sample that led to the incorrectness of BERNRES.

Suppose that the initial sample size is $|S| = M$ with probability 1 and the target sample size is $M' > M$, where $M, M' < |R|$.[5] We say that a sample S is an $\mathrm{RP}(M', d)$ sample of R if it is produced by running $\mathrm{RP}(M')$ on a sequence γ that produces R and contains d uncompensated deletions. Such sequence exists for any value of d; for example, the sequence may consist of $|R|$ insertions, one for each item in R, followed by the insertion of d "transient" items from $\mathscr{R} \setminus R$, which are subsequently deleted. The key idea of RPRES is to convert sample S to an $\mathrm{RP}(M', d)$ sample, where d is treated as a parameter of the conversion process. The conversion may require access to the base data, but—as we discuss in the following section—the probability and amount of such accesses depend on d. After conversion, subsequent transactions are processed using $\mathrm{RP}(M')$ so that, after a sufficiently large number of insertions, the sample size reaches its target value M'. When base data accesses are expensive and insertions occur frequently, this approach can be much faster than the traditional approach of recomputing the sample from scratch.

RPRES comprises two phases. In phase 1 (conversion), the algorithm generates a hypergeometric$(|R| + d, |R|, M')$ random variable U, which—according to theorem 4.2—represents the initial $\mathrm{RP}(M', d)$ sample size. The distribution of U depends on parameter d, e.g., $\mathrm{E}[U] = M'|R|/(|R| + d)$. For $d = 0$, we have $U = M'$ with probability 1; for larger values of d, the (expected) value of U decreases. The algorithm now uses as many items from S as possible to make up the $\mathrm{RP}(M', d)$ sample, accessing base data only if $U > |S|$. Phase 1 concludes with the computation of the counters c_{g} and c_{b} according to (4.1). In phase 2 (growth), the algorithm increases the sample to the desired size by using $\mathrm{RP}(M')$ sampling and subsequent insertions. Since the dataset is growing, the sample size will eventually reach its target value M'. The phase ends as soon as the number of uncompensated deletions reaches 0, in which case the sample size is guaranteed to equal M'. We also end phase 2 when the dataset becomes empty, although this situation is unlikely to occur in practice. If it does, resizing should be abandoned in favor of a fresh $\mathrm{RP}(M')$ sample maintained as the dataset grows again.

In algorithm 4.3, we give the complete pseudocode of the resizing procedure. We make use of a function HYPERGEOMETRIC that generates samples from the hypergeometric distribution; see Zechner (1997, p. 101) or Kachitvichyanukul and Schmeiser (1985) for efficient rejection algorithms, or refer to a statistical library that includes this function, such as the Colt Library (2004) or the GNU Scientific Library (2008).

[5] It is important that the initial sample size equals M with probability 1 because otherwise subtle issues with uniformity can occur. An example of such issues is given in section 4.2.4, where we discuss subsampling. However, when RP is used to maintain the sample, the assumption that $\Pr[|S| = M] = 1$ is fulfilled whenever $d = 0$ and—because the dataset is growing—easily satisfied in practice.

Algorithm 4.3 Resizing

M: initial sample size
M': target sample size ($M' > M$)
R: initial dataset
S: initial sample with $|S| = M$
d: resizing parameter

PHASE 1: *// convert to RP(M',d) sample*
$U \leftarrow \text{HYPERGEOMETRIC}(|R| + d, |R|, M')$
if $U \leq M$ **then**
    $S \leftarrow$ uniform subsample of size U from S
else
    $V \leftarrow$ uniform sample of size $U - M$ from $R \setminus S$
    $S \leftarrow S \cup V$
end if
$c_\text{g} \leftarrow M' - |S|$
$c_\text{b} \leftarrow d - c_\text{g}$
go to phase 2

PHASE 2: *// grow sample to size M'*
while $c_\text{g} + c_\text{b} > 0 \wedge |R| > 0$ **do**
    wait for transaction
    process transaction using $\text{RP}(M')$ and counters c_g and c_b
end while
return S

We now assert the correctness of the resizing algorithm. Denote by S^* the effective sample at the end of phase 1. Also denote by S_i and R_i ($i \geq 0$) the elements in the sample and the dataset after i transactions have occurred in phase 2. Finally, denote by L (≥ 0) the number of transactions that occur during phase 2; the value of L is completely determined by the transaction sequence and the value of parameter d.

Theorem 4.4. *For any i, where $0 \leq i \leq L$, the set S_i is a uniform sample from R_i.*

Proof. According to theorem 4.2, the distribution of U in phase 1 is identical to the distribution of the sample size of an $\mathrm{RP}(M', d)$ sample from R. The subsampling step ($U \leq M$) and the union step (otherwise) both maintain the uniformity of S, so that S^* is also uniform. Both facts together imply that S^* is an $\mathrm{RP}(M', d)$ sample from R. From this fact and from (4.1), we conclude that sample S^* along with counters c_g and c_b forms an $\mathrm{RP}(M')$ sample of R. The assertion of the theorem now follows from the correctness of $\mathrm{RP}(M')$. □

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 draw-sequential sampling techniques as the ones discussed in section 3.4.1. Such techniques are applicable in a wide range of settings and are typically much faster than a scan of the entire dataset. Recall that draw-sequential 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); otherwise, the item is rejected and the process starts over. As discussed in Gemulla et al. (2008), 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 sequence of transactions, 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 d 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 large value of d 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 small value of d might be preferable to minimize the complete resizing time.

D. Example

The RPRES algorithm is illustrated in figure 4.11, where we make use of the counterexample that was used to show the non-uniformity of BERNRES. The transaction sequence is given by $\gamma = (+r_1, +r_2, +r_3)$ and the sample is resized from $M = 1$ to $M' = 2$ after transaction $+r_1$ and $+r_2$ have been processed. We use a value of $d = 1$ for the resizing algorithm. Right before resizing is initiated, the sample S is equal to either $\{ r_1 \}$ or $\{ r_2 \}$, each with probability 1/2. For $|R| = 2$ and our choice

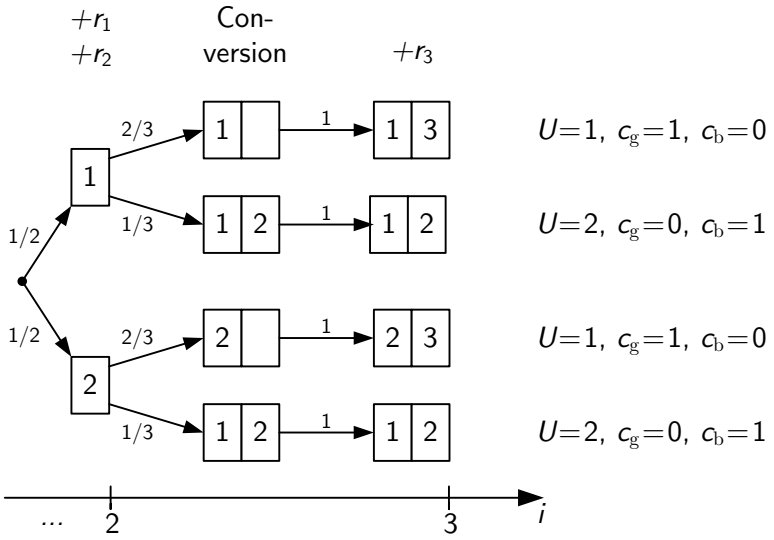


Figure 4.11: Illustration of resizing

of $d = 1$, we have $\Pr[U = 1] = 2/3$ and $\Pr[U = 2] = 1/3$. When $U = 1 = |S|$, the sample remains unmodified and the counters are set to $c_g = M' - U = 1$ and $c_b = d - c_g = 0$. Otherwise, when $U = 2$, a random item is selected from the base data and added to the sample and the counters are set to $c_g = 0$ and $c_b = 1$. It can be seen from the figure that, right after the conversion, the sample is indeed uniform. When transaction $+r_3$ arrives, a pairing step is executed. The decision of whether to accept or reject r_3 depends on the values of the counters, which in turn depend on U. In our example, item r_3 is added to the sample for $U = 1$ and ignored for $U = 2$. At this point of time, we have $c_g = c_b = 0$ and the resizing algorithm terminates. Note that the resizing algorithm accesses the base data with probability $1/3$; with probability $2/3$, no such access is required.

4.2.2 Parametrization of Resizing

This section addresses the key problem of choosing the parameter d in algorithm 4.3. For a given choice of d, 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 hypergeometric random variable U, and the time required for phase 2 depends on both d and the transaction sequence. Our goal is therefore to develop a probabilistic model of the resizing process, and choose d to minimize the expected resizing cost.[6] We develop perhaps the simplest possible model, based on the base-data access paradigm described previously and a very simple model of the transaction stream. A

[6] 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 d. In many scenarios, however, complete recomputation will not be the best option.

discussion of more complex cost models and their implications on the resizing cost can be found in Gemulla et al. (2008). Given a cost model, numerical methods can be used to determine d^*, the optimal value of d. Since numerical optimization can be too expensive at runtime, we also consider an approximate model of the cost function that can be minimized analytically. The experiments in section 4.2.3 indicate that both the approximate model of expected cost and the resulting choice of d closely agree with the results obtained via numerical methods, thereby justifying the use of the quick approximate analytical method.

A. Modeling the Resizing Process

We first consider the cost of phase 1. During this phase, the algorithm obtains $N(U)$ items from $R \setminus S$, where

$$N(u) = \max(u - M, 0)$$

for $0 \leq u \leq M'$. 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. Following an argument as in section 3.4.1, the (random) number B_k of base-data accesses required to obtain the k-th item has a geometric distribution

$$\Pr[B_k = n] = p_k^{n-1}(1 - p_k), \qquad n \geq 1,$$

where the failure probability $p_k = (M + k - 1)/|R|$ denotes the probability of drawing an item that is already present in the sample. The random variables B_k are mutually independent. Supposing that each base-data access takes t_a time units, the expected phase 1 cost is $t_a \operatorname{E}_d[B]$, where $B = B_1 + B_2 + \cdots + B_{N(U)}$. We use the subscript d to emphasize the fact that the probability distribution of B depends on the distribution of U, and hence on the parameter d. We can re-express this expected cost in a more convenient form. Using again an argument as in section 3.4.1, we have

$$\operatorname{E}_d[B \mid U] = |R| \, H_{|R|-M-N(U)+1,\,|R|-M},$$

where $H_{n,m} = \sum_{i=n}^{m} 1/i$. Using approximation (3.5) for partial harmonic numbers, we can write $\operatorname{E}_d[B \mid U] = g(U)$, where

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

By the law of total expectation, we have

$$\operatorname{E}_d[B] = \operatorname{E}_d\big[\operatorname{E}_d[B \mid U]\big] = \operatorname{E}_d[g(U)],$$

and we can therefore write the expected phase 1 cost as

$$T_1(d) = t_a \operatorname{E}_d[g(U)].$$

We now consider the cost of phase 2. In this phase, the resizing algorithm executes L steps of the random pairing algorithm, where L depends on the transaction sequence. To make further progress, we need a model of the insertion and deletion process. The simplest model, which we will use here, is to assume that, during phase 2, a sampling step 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 $q = (1 - p)$ 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.

The distribution the number L of transactions in phase 2 can be obtained by an analogy to a ruin problem, see for example Feller (1968). In the classical ruin problem, a gambler wins or loses a dollar with probability p and $q = 1 - p$, respectively. The gambler is given initial capital z and the game ends when the gambler's capital reduces to zero (=ruin) or reaches value a (=win). We are interested in the expected number of steps until the gambler either wins or is ruined. In our setting, we have $z = |R|$ and a=$|R| + d$. The expected value of the duration of the ruin problem is given by (Feller 1968, p. 348)

$$\mathrm{E}_d\,[\,L\,] = \frac{|R|}{q-p} - \frac{|R|+d}{q-p}\,\frac{1-(q/p)^{|R|}}{1-(q/p)^{|R|+d}}. \tag{4.13}$$

The expected cost of phase 2 can now be written as

$$T_2(d) = t_\mathrm{b}\,\mathrm{E}_d\,[\,L\,],$$

and the total resizing cost can be written as

$$T(d) = T_1(d) + T_2(d) = \mathrm{E}_d\,[\,t_\mathrm{a} g(U) + t_\mathrm{b} L\,].$$

B. Finding an Optimum Parametrization

We can now apply numerical methods to find the optimum value d^* for d so that $T(d)$ is minimized. The expected cost of phase 1 can be computed numerically, based on the formula

$$\begin{aligned}\mathrm{E}_d\,[\,g(U)\,] &= \sum_{u=l}^{M'} \mathrm{E}_d\,[\,B \mid U = u\,]\,\Pr\,[\,U = u\,] \\ &= \sum_{u=l}^{M'} g(u) H\big(u;\,|R|+d, |R|, M'\big)\,,\end{aligned} \tag{4.14}$$

where $l = \max(M' - d, 0)$ denotes the minimum value of U under our assumption that $M' < |R|$. The above sum can be evaluated quite efficiently because only a small number of terms contribute significantly to the sum. The expected cost of phase 2 is given in (4.13) and can be easily computed. Given both formulas, we can

use standard numerical optimization algorithms to compute d^*, see, for example, (Press et al. 1992, ch. 10).

We now explore a closed-form approximation to the function $T(d)$ that is highly accurate and agrees closely with our numerical results. This approximation immediately leads to an effective approximation of d^*. The first step in the approximation is to assume that $U = \mathrm{E}[U] = M'|R|/(|R| + d)$ with probability 1; see theorem 4.2 for the expected value and variance of U. Our motivation is that the coefficient of variation

$$\mathrm{CV}[U] = \sqrt{\frac{\mathrm{Var}[U]}{\mathrm{E}^2[U]}} = \sqrt{\frac{d(|R| + d - M')}{M'|R|(|R| + d - 1)}}$$

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

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

$$\hat{T}_1(d) = g(\mathrm{E}[U]) = g\left(M'\frac{|R|}{|R| + d}\right) = t_\mathrm{a}|R| \ln\left[\frac{|R| - M}{|R| - M - N\left(M'\frac{|R|}{|R|+d}\right)}\right].$$

Making use of the fact that $N(u) = 0$ for $u \le M$, and since

$$M'\frac{|R|}{|R| + d} \le M \quad \text{for} \quad d \ge |R|\frac{M' - M}{M},$$

we find that

$$\hat{T}_1(d) = \begin{cases} t_\mathrm{a}|R| \ln\left[\dfrac{(|R| - M)(|R| + d)}{|R|(|R| + d - M')}\right] & d < |R|\frac{M'-M}{M} \\ 0 & d \ge |R|\frac{M'-M}{M}. \end{cases}$$

It is easy to show that $\hat{T}_1(d)$ is monotonically decreasing, convex, and differentiable on the interval $[0, |R|\frac{M'-M}{M})$.

To approximate the expected phase 2 cost $T_2(d)$, observe that the expected change of 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 $1/(2p - 1)$. Thus, roughly $d/(2p - 1)$ steps are required, on average, to increase the dataset size by d and therefore to finish phase 2. This leads to an approximate expected phase 2 cost of

$$\hat{T}_2(d) = t_\mathrm{b}\frac{d}{2p - 1}.$$

The above equation is precisely the limit of (4.13) as $|R| \to \infty$; in practice, the approximation is accurate when $|R|$ is not too small. The expected total time required to resize a sample is approximately equal to $\hat{T}(q) = \hat{T}_1(q) + \hat{T}_2(q)$.

We now choose $d = \hat{d}^*$, where $\hat{d}^*$ minimizes the function $\hat{T}$. Note that our search for $\hat{d}^*$ can be restricted to the interval $[0, |R|\frac{M'-M}{M}]$, because $\hat{T}_2(d)$ is increasing and $\hat{T}_1(d) = 0$ for $d \geq \frac{M'-M}{M}$. Thus, to compute $\hat{d}^*$, first set

$$d_0 = \frac{M}{2} - |R| + \sqrt{\frac{M^2}{4} + \frac{t_a}{t_b}|R|M'(2p-1)}. \tag{4.15}$$

If $d_0 \in [0, |R|\frac{M'-M}{M}]$, then d_0 satisfies $\hat{T}'(d_0) = 0$, and we take $\hat{d}^* = d_0$. Otherwise, we take $\hat{d}^*$ to be either 0 or $|R|\frac{M'-M}{M}$, depending upon which of the quantities $\hat{T}(0)$ or $\hat{T}(|R|\frac{M'-M}{M})$ is smaller. In our experiments, we found that the combined error introduced by assuming that $\Pr[U = \mathrm{E}[U]] = 1$ and by replacing the harmonic sum $H_{n,m}$ by $\ln\big(m/(n-1)\big)$ appears to be negligible.

4.2.3 Experiments

We conducted an experimental study to evaluate the performance of the RPRES algorithm. We found that

- The numerical method and the quick approximation approach agree very closely with respect to both the cost function $T(d)$ and the optimal parameter value d^*. 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 d can have a significant impact on the resizing cost.

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 arrivals during phase 2. The experimental results for various other choices of parameters were qualitatively similar.

Figure 4.12 displays the expected resizing cost $T(d)$ for various values of d, when the base-data access cost t_a equals 20, 50, and 90 milliseconds.[7] There are three possible behaviors of the T function in the search interval: increasing, decreasing, and internal minimum point. Our choices of t_a illustrate these three possible behaviors. For each scenario, the approximate cost $\hat{T}$ is represented as a solid curve. Superimposed on this curve are points that represent the exact (expected) cost T for various values of d. As expected, when the base-data access cost t_a is relatively small, the cost function achieves its minimum value at $d = 0$, 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 $d = 1{,}000{,}000$, and the optimal strategy is to not sample the base data at all, and increase the

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

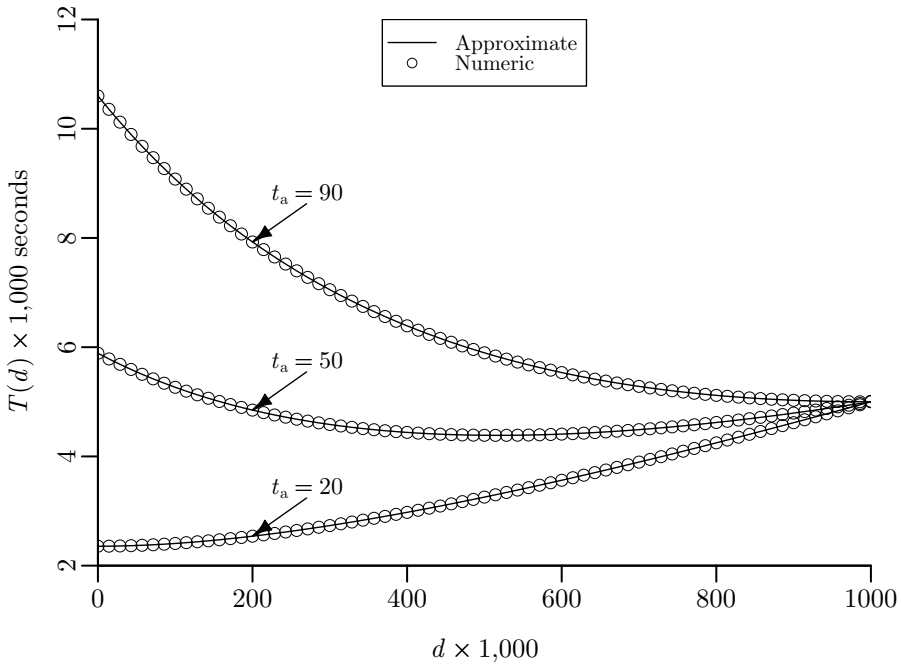


Figure 4.12: Expected resizing cost $T(d)$

sample size to M' exclusively during phase 2. For an intermediate value of t_a, the optimal value of d falls in between 0 and 1,000,000—here, $d^* \approx 517{,}745$—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 d can vary by a factor of two. Moreover, the approximate and exact 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.

The high accuracy of the cost approximation leads us to expect that our numerical and approximate methods will also yield similar estimates for d^*. This expectation is fulfilled, as shown in figures 4.13 and 4.14. Figure 4.13 shows the optimal value d^* for various values of t_a, while Figure 4.14 shows the (exact) expected resizing cost for the numeric and approximate value of d^*. As before, the solid line represents values computed via the quick approximate method and the circular points represent the numerical solutions. The approximate method seems to slightly underestimate the exact value of d^*. However, even when the value of d^* produced by the numerical method differs slightly from the result of the approximate closed-form model, the resulting resizing costs do not differ perceptibly. The reason is that the cost curve—as shown in figure 4.12—is flat around the optimum value of d^*.

To evaluate the stability of RPRES with respect to its performance, we run as a final experiment 100 independent repetitions of a Java implementation of RPRES. We used the three scenarios $t_a = 20$, $t_a = 50$ and $t_a = 90$ and set d to its respective

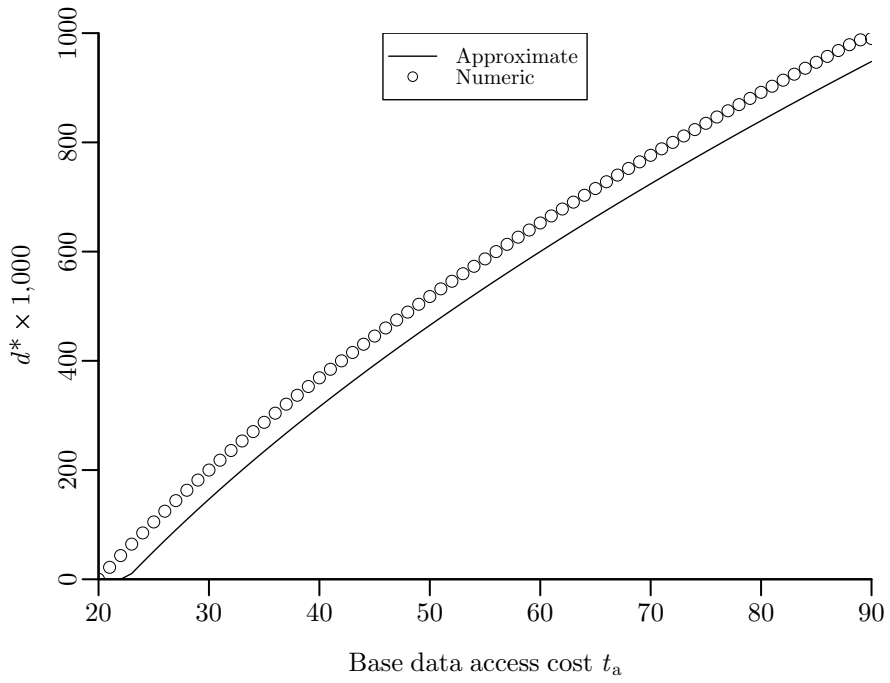


Figure 4.13: Optimal value of d

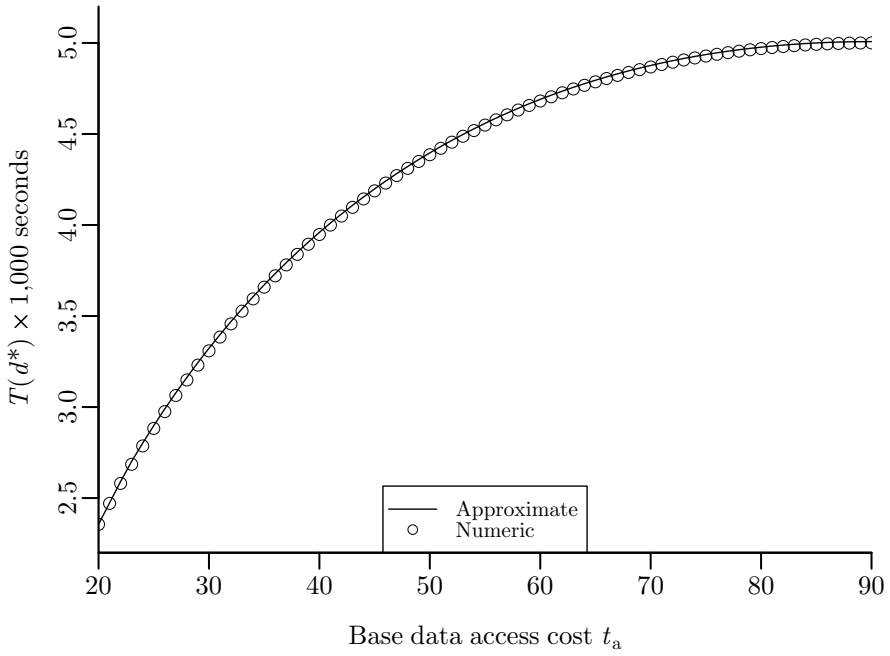


Figure 4.14: Optimal expected resizing cost $T(d^*)$

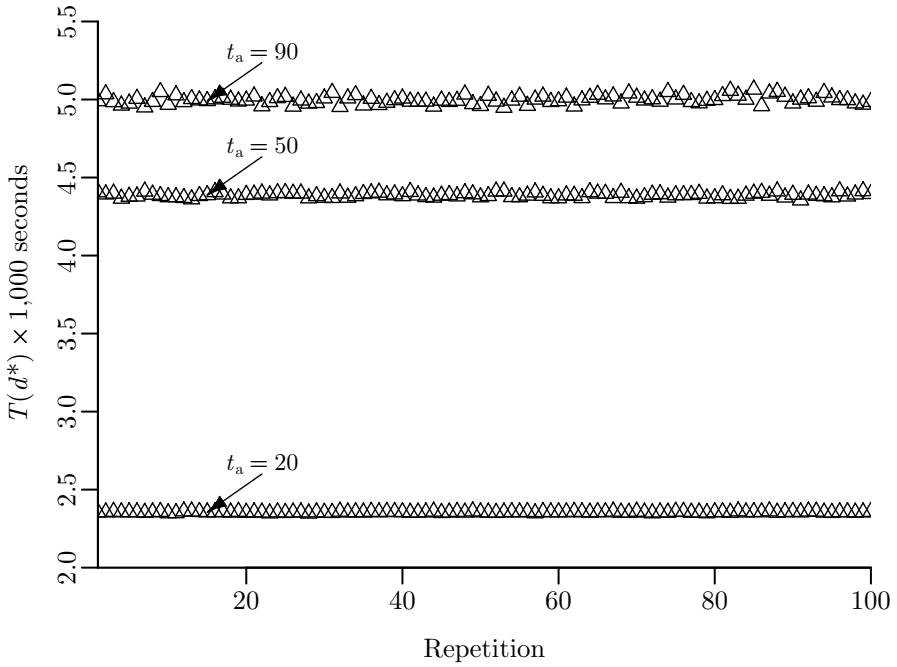


Figure 4.15: Actual resizing cost at optimum parametrization

optimum value. Within each run, we monitored the number of base data accesses and the total number of arriving transactions until the resizing process ended. The results are given in figure 4.15, where we print a symbol for each individual run. As can be seen, the running times are very stable, with little variation across multiple runs. We conclude that the actual running time of the RPRES algorithm stays close to its expected value with high probability, so that RPRES exhibits a stable performance.

4.2.4 Resizing Downwards

We now turn attention to the subsampling problem, which can be formalized as follows: Given an RP(M) sample S_i with counters $d_i = c_{\mathrm{b},i} + c_{\mathrm{g},i}$, derive an RP($M'$) sample S'_i with $M' < M$ accompanied with appropriate counters $d'_i = c'_{\mathrm{b},i} + c'_{\mathrm{g},i}$. As a technical matter, we assume henceforth that $|R_i| + d_i > M$ so that the maximum sample size seen so far equals M.[8]

Suppose that after processing transaction γ_i, the memory bound is reduced from M to M'. We first consider the case where $d_i \leq M - M'$. Then, the minimum sample size l_i as defined in (4.2) is at least M' so that $\Pr[\,|S_i| \geq M'\,] = 1$. To form S'_i, we simply select M' random items from S_i; this random selection can be executed

[8] Otherwise, we have $S_i = R_i$ and the subsampling problem can be trivially solved: simply run RP(M') using the elements in S_i as input.

using one of the list-sequential algorithms of section 3.4.2. Observe that since S_i is uniform so is S'_i. Together with $|S'_i| = M'$, it follows that S'_i is statistically identical to a fresh RP(M') sample computed from R_i so that we can set both $c'_{\mathrm{b},i}$ and $c'_{\mathrm{g},i}$ to 0. In what follows, we will refer to this subsampling algorithm as SUB(M') or simply SUB.

We now consider the case where $d_i > M - M'$ so that $\Pr[\,|S_i| < M'\,] > 0$. An "obvious" algorithm for this case would be as follows: When $|S_i| \geq M'$, apply SUB as before; this reduces the sample size to M' as desired. Otherwise, when $|S_i| < M'$, there seems to be no need to perform any immediate action; we wait until—at some time $i' > i$—$|S_{i'}| = M'$ and then apply SUB. Unfortunately, the obvious algorithm turns out to be incorrect. To see this, consider the example given in figure 4.1 on page 106 and suppose that the sample size is reduced from $M = 2$ to $M' = 1$ just after transaction $\gamma_5 = -r_3$ has been processed. At that time, we have

$$\Pr[\,S_5 = \emptyset, c_\mathrm{g} = 0, c_\mathrm{b} = 2\,] = \frac{1}{3},$$
$$\Pr[\,S_5 = \{\,r_1\,\}, c_\mathrm{g} = 1, c_\mathrm{b} = 1\,] = \frac{2}{3}.$$

The obvious subsampling algorithm leads to

$$\Pr\left[\,S'_5 = \{\,r_1\,\}, c'_{\mathrm{b},i} = 0, c'_{\mathrm{g},i} = 0\,\right] = \frac{2}{3},$$
$$\Pr\left[\,S'_5 = \emptyset, c'_{\mathrm{b},i} = 2, c'_{\mathrm{g},i} = 0\,\right] = \frac{1}{3},$$

where SUB is applied only when $S_5 = \{\,r_1\,\}$. Now, suppose that transaction $\gamma_6 = +r_4$ is processed. Then, by applying the random pairing algorithm, we have

$$\begin{aligned}
\Pr\left[\,S'_6 = \{\,r_1\,\}\,\right] &= \Pr\left[\,S'_5 = \{\,r_1\,\}, r_4 \text{ rejected}\,\right] = \frac{2}{3}\frac{1}{2} = \frac{1}{3},\\
\Pr\left[\,S'_6 = \{\,r_4\,\}\,\right] &= \Pr\left[\,S'_5 = \{\,r_1\,\}, r_4 \text{ accepted}\,\right] + \Pr\left[\,S'_5 = \emptyset, r_4 \text{ accepted}\,\right]\\
&= \frac{2}{3}\frac{1}{2} + \frac{1}{3}\frac{2}{2} = \frac{2}{3},
\end{aligned}$$

which is clearly non-uniform. Thus, we have to find another algorithm for reducing the upper bound whenever $\Pr[\,|S_i| < M'\,] > 0$ and S'_i is subject to further maintenance.

The key idea underlying our RPSUB subsampling algorithm is the following: Suppose that, conceptually, we do not purge deleted items from the dataset, but rather put them into a "transient" state. Then, the dataset contains both "real" and "transient items" that have not yet been purged; we refer to the dataset as being "augmented" with transient items. Further suppose that we are somehow able to come up with a size-M uniform sample from the *augmented* dataset. (We will show that, again conceptually, such a sample can be obtained from sample S.) Since $M > M'$, the size of the augmented sample is larger than M' with probability 1, and we can we can apply SUB to obtain an *augmented* subsample. The final subsample

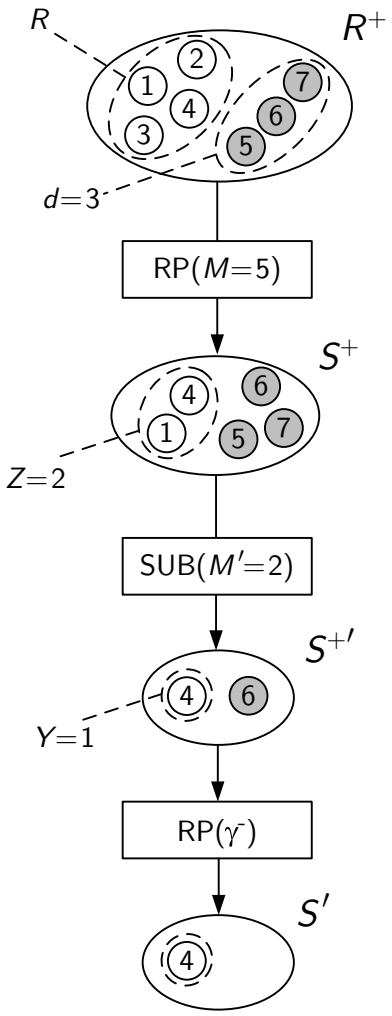


Figure 4.16: A hypothetical subsampling algorithm

S' is formed by removing all transient items from the augmented subsample. This hypothetical algorithm is illustrated in figure 4.16; real and transient items are shown as white and gray numbered circles, respectively.

To obtain the actual RPSUB algorithm, we determine the probability distribution of Y—the number of real items that S ultimately contributes to S' in the hypothetical algorithm. We then generate a realization y of Y directly, and randomly sample y items from S to form subsample S'. The only remaining task is to determine the appropriate values for the counters c_b and c_g in the subsample. 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 R using a distinguished transaction sequence that is derived from the original sequence γ that was used to

create S. The resulting counter values for the latter scenario are easy to determine, and are assigned to the counters for the sample produced by RPSUB.

A. Deferring Deletion Transactions

The derivation and analysis of the RPSUB algorithm rest on theorem 4.5 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 thus 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 $R(\gamma)$, $S(\gamma)$, $C_{\mathrm{b}}(\gamma)$, $C_{\mathrm{g}}(\gamma)$, and $d(\gamma)$, the dataset, sample, counter values, and number of uncompensated deletions that result from processing the transaction sequence γ using the RP(M) algorithm.

Theorem 4.5. *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*

$$R(\gamma) = R(\gamma') \qquad \textit{and} \qquad d(\gamma) = d(\gamma') \tag{4.16}$$

and

$$\Pr\left[\, S(\gamma) = A, C_b(\gamma) = c_b, C_g(\gamma) = c_g \,\right] = \Pr\left[\, S(\gamma') = A, C_b(\gamma') = c_b, C_g(\gamma') = c_g \,\right]$$

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 $\mathscr{R} \setminus R(\gamma)$, 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 (4.16) 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 (4.16) and (4.2), we have $v(\gamma) = v(\gamma')$, so that, by (4.12),

$$\Pr\left[\, S(\gamma) = A \,\right] = \Pr\left[\, S(\gamma') = A \,\right]$$

for all $A \subseteq R(\gamma)$, and hence

$$\Pr\left[\, |S(\gamma)| = k \,\right] = \Pr\left[\, |S(\gamma')| = k \,\right]$$

for all $k \geq 0$. The final assertion of the theorem now follows from (4.5). □

B. Algorithmic Description

We now describe the hypothetical and, subsequently, our RPSUB algorithm in full detail. Suppose that we have run the RP(M) algorithm to obtain sample $S \subseteq R$, and that there are d uncompensated deletions. Denote by γ the transaction sequence that

produced R and S. Using theorem 4.5, we can assume without loss of generality that $\gamma = \gamma^+\gamma^-$, where γ^+ consists solely of insertions and γ^- consists solely of deletions. The sequence γ^+ corresponds to the insertion of the $|R|$ "real" items that comprise R, followed by the insertion of d "transient" items in $\mathscr{R} \setminus R_j$ that will ultimately be deleted from the dataset by processing the transactions in γ^-.

The hypothetical algorithm creates an intermediate augmented dataset $R^+ = R(\gamma^+)$ and augmented sample $S^+ = S(\gamma^+)$ by running $\mathrm{RP}(M)$ on γ^+. Dataset R^+ comprises $|R|$ real items and d transient items, and sample S^+ comprises Z real items and $M - Z$ transient items, where

$$\Pr[\, Z = k \,] = H(k;\, |R| + d, |R|, M) \tag{4.17}$$

according to (4.10); see the discussion right after theorem 4.2. The hypothetical algorithm now applies the SUB algorithm to obtain the subsample $S^{+\prime}$. To do so, SUB selects M' random items from S^+. Observe that, of the M' items from S^+ added to $S^{+\prime}$, precisely Y items are real, where

$$\Pr[\, Y = k \mid Z \,] = H\big(k;\, M, Z, M'\big)\,. \tag{4.18}$$

The hypothetical algorithm concludes by running the $\mathrm{RP}(M')$ algorithm on the sequence γ^-, starting with augmented sample $S^{+\prime}$ and counter values $c_\mathrm{b} = c_\mathrm{g} = 0$. This final processing step has the effect of removing all transient items from $S^{+\prime}$, thereby transforming $S^{+\prime}$ into the final sample S'. Figure 4.16 depicts the hypothetical algorithm in action.

We now determine appropriate counter values for the subsample, as well as the probability distribution for its size. First observe that, by theorem 4.1, the sample $S^{+\prime}$ that results from the execution of SUB by the hypothetical algorithm is statistically identical to the sample $S(\gamma^+) \subseteq R(\gamma^+)$ obtained by running $\mathrm{RP}(M')$ on the dataset R^+. Thus the final subsample produced by the hypothetical algorithm is statistically identical to the sample $S(\gamma^+\gamma^-)$ produced by running $\mathrm{RP}(M')$ on R using the sequence $\gamma^+\gamma^-$, starting with counter values $c_\mathrm{b} = c_\mathrm{g} = 0$. Observe that, after running $\mathrm{RP}(M')$ on R in this manner, the final counter values are $c_\mathrm{b} = M' - |S'|$ and $c_\mathrm{g} = d - c_\mathrm{b}$. Indeed, after the transactions in γ^+ have been processed, the sample size is M' and the counter values for RP are given by $c_\mathrm{b} = c_\mathrm{g} = 0$ since no deletions have occurred so far; after processing the remaining transactions in the sequence γ^-, the deficit $M' - |S'| = c_\mathrm{b}$ corresponds precisely to the bad deletions, and the remaining $d - c_\mathrm{b}$ deletions are good. Since the hypothetical algorithm produces output just as if $\mathrm{RP}(M')$ had been run on R, we can start with these counter values when maintaining the subsample S'. Finally, by theorem 4.2, the probability distribution for the size of the sample S', and hence the probability distribution for sample size produced by the hypothetical algorithm, is given by

$$\Pr\big[\, |S'| = k \,\big] = H\big(k;\, |R| + d, |R|, M'\big) \tag{4.19}$$

for $0 \le k \le M'$. We show below that, by construction, the output of the actual RPSUB algorithm is statistically identical to the output of the hypothetical algorithm,

Algorithm 4.4 Subsampling for random pairing

S: sample of R
d: number of uncompensated deletions for R
M: sample-size bound used by RP for generating S
M': new sample-size bound, $M' < M$

$Y \leftarrow \text{HYPERGEOMETRIC}(M, |S|, M')$
$S' \leftarrow \{\, Y \text{ random items from } S \,\}$
$c'_{\mathrm{b}} = M' - |S'|$
$c'_{\mathrm{g}} = d - c'_{\mathrm{b}}$

return $(S', c'_{\mathrm{b}}, c'_{\mathrm{g}})$

and the above results on counter values and sample-size distributions apply to RPSUB as well.

We now specify the RPSUB procedure. Ignoring transient items, we see that the net effect of the hypothetical algorithm is to select precisely Y items from a uniform sample $S^* \subseteq R$ of size Z, where the $Z = |S^*|$ is distributed according to (4.17) and Y—conditionally on $|S^*|$—is distributed according to (4.18). Observe that, by theorem 4.2, the distribution of $Z = |S^*|$ is identical to that of $|S|$, so we can take S for S^* in the procedure. Thus we generate Y according to

$$\Pr[\, Y = k \,] = H\big(k;\, M, |S|, M'\big) \tag{4.20}$$

and then select Y random items from S to obtain S'. The complete pseudocode of the subsampling procedure is given as algorithm 4.4, where we make use of the function HYPERGEOMETRIC to generate samples from the hypergeometric distribution; see section 4.2.1C.

4.3 Sample Merging

In the foregoing discussion, we 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 Brown and Haas (2006) 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 (2006) provide an algorithm, called MERGE, 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. MERGE can also be used in the deletion setting but—as we will see in section 4.3.1—has the disadvantage that the size of the merged sample is sensitive to skew in the local sample sizes caused by uncompensated deletions. 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 section 4.2. Our new extension of MERGE, 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 similar to that of RPSUB. Recall that, conceptually, RPSUB defers the processing of deletion transactions until after the SUB algorithm has been applied. The dataset and sample prior to processing the deletion are treated as being augmented with "transient" items that have not yet been purged. Building upon this idea, our RPMERGE algorithm, again conceptually, defers processing of deletion transactions on both participating datasets until after MERGE has been applied: We run the MERGE 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 figure 4.17; real and transient items are shown as white and gray numbered circles, respectively. To obtain the actual RPMERGE algorithm, we determine the probability distribution of Y_1 and Y_2, then generate realizations y_1 and y_2 of Y_1 and Y_2 directly, and then select y_1 random items from S_1 and y_2 random items from S_2 to form the merged sample S. As with RPSUB, the values of the counters c_b and c_g are easy to determine by making use of a statistical equivalence of RPMERGE and a fresh RP sample drawn from R using a transaction sequence where all deletions occur at the end. The foregoing statistical equivalence also permits easy calculation of the probability distribution for the size of the merged sample; in section 4.3.2B we use this distribution to show that RPMERGE typically produces larger sample sizes than naive MERGE in expectation.

4.3.1 General Merging

The MERGE algorithm as described by Brown and Haas (2006) 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:

$$\Pr[X_1 = k] = H(k; |R_1| + |R_2|, |R_1|, m)\,. \tag{4.21}$$

In fact, $H(k; |R_1| + |R_2|, |R_1|, m)$ is equal to the probability that exactly k out of m random items from $R_1 \cup R_2$ belong to R_1. The resulting sample is therefore statistically equivalent to a size-m uniform sample from $R_1 \cup R_2$ so that MERGE is indeed correct.

We can apply MERGE, unchanged, in our setting and, after merging, use the RP(m) algorithm to incrementally maintain S; to initialize RP(m), set $c_\mathrm{b} \leftarrow 0$ and $c_\mathrm{g} \leftarrow 0$ after the merging process has been completed. Observe that the size (and upper bound) 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 Brown and Haas (2006), 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 MERGE 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 $m < M$. Specifically, MERGE is sensitive to skew in the sample sizes: if either S_1 or S_2 has many uncompensated deletions, then this very small sample limits the size of the merged sample. As previously discussed in section 4.2.1A, there is no way to increase the sample size above m 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 $|S_2| < M_2$. Otherwise, if $|S| < M$, future insertions can be exploited to grow the sample to size M without accessing base data because RPMERGE provides RP(M) counter values for the merged sample.

4.3.2 Merging for Random Pairing

For dataset R_j, $j \in \{1, 2\}$, denote by γ_j the transaction sequence that produced sample S_j and let d_j be the number of uncompensated deletions in γ_j. We assume henceforth that $v_j = \min(M_j, |R_j| + d_j)$, $j \in \{1, 2\}$, the maximum sample size seen so far, satisfies $v_j = M_j$.[9]

A. Algorithmic Description

We first describe the hypothetical algorithm in detail. An illustration of the algorithm is shown in figure 4.17. Using theorem 4.5, we can assume without loss of generality that $\gamma_j = \gamma_j^+ \gamma_j^-$, where γ_j^+ consists of $|R_j| + d_j$ insertions and γ_j^- consists of d_j deletions. As before, denote by $R_j^+ = R(\gamma_j^+)$ the augmented dataset and by S_j^+ the augmented sample that is obtained by running RP(M_j) on R_j^+ using sequence γ_j^+. Observe that under our assumptions $|S_j^+| = M_j$. As a first step, the hypothetical algorithm applies MERGE to the augmented samples S_1^+ and S_2^+. MERGE selects X_1 random items from S_1^+ and $M - X_1$ items from S_2^+, where according to (4.21)

$$\Pr[X_1 = k] = H\big(k; |R_1^+| + |R_2^+|, |R_1^+|, M\big) \tag{4.22}$$

[9] 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.

Algorithm 4.5 Merging for random pairing

S_j: sample of R_j $(j \in \{1, 2\})$
d_j: number of uncompensated deletions for R_j
M_j: sample-size bound used by RP for generating S_j

$M \leftarrow \min(M_1, M_2)$
$X_1 \leftarrow \text{HYPERGEOMETRIC}(|R_1| + |R_2| + d_1 + d_2, |R_1| + d_1, M)$
$Y_1 \leftarrow \text{HYPERGEOMETRIC}(M_1, |S_1|, X_1)$
$X_2 \leftarrow M - X_1$
$Y_2 \leftarrow \text{HYPERGEOMETRIC}(M_2, |S_2|, X_2)$

$S \leftarrow \{ Y_1 \text{ random items from } S_1 \} \cup \{ Y_2 \text{ random items from } S_2 \}$
$c_\mathrm{b} = M - |S|$
$c_\mathrm{g} = d_1 + d_2 - c_\mathrm{b}$

return $(S, c_\mathrm{b}, c_\mathrm{g})$

with $M = \min(M_1, M_2)$. Denote by $S_j^{+\prime}$ the obtained augmented subsample of S_j^+. The MERGE algorithm then computes the union of the augmented subsamples to obtain the merged augmented sample $S^+ = S_1^{+\prime} \cup S_2^{+\prime}$, which is by construction a size-M uniform random sample of $R^+ = R_1^+ \cup R_2^+$. Since $R^+ = R(\gamma_1^+\gamma_2^+)$ and both γ_1^+ and γ_2^+ consist of only insertions, S^+ is statistically equivalent to an $RP(M)$ sample with counters $c_\mathrm{b} = c_\mathrm{g} = 0$ computed directly from R^+. The hypothetical algorithm now obtains the final sample S by running $\mathrm{RP}(M)$ on S^+ using sequence $\gamma_1^-\gamma_2^-$ and initial counter values $c_\mathrm{g} = c_\mathrm{b} = 0$. Sample S is statistically equivalent to an execution of $\mathrm{RP}(M)$ on sequence $\gamma = \gamma_1^+\gamma_2^+\gamma_1^-\gamma_2^-$ and—since $R(\gamma) = R$—it follows that S is an $\mathrm{RP}(M)$ sample of R. Because the number of bad deletions that occurred during processing $\gamma_1^-\gamma_2^-$ equals the difference in sample sizes $|S^+| - |S| = M - |S|$, this execution of $\mathrm{RP}(M)$ results in counters $c_\mathrm{b} = M - |S|$ and $c_\mathrm{b} = d_1 + d_2 - c_\mathrm{g}$.

To specify RPMERGE, observe that the number Y_j of real items from R_j in S as obtained by the hypothetical algorithm is distributed as the number of real items in a size-X_j subsample of R_j^+. We can therefore derive the distribution of Y_j using arguments as in section 4.2.4; the similarity of RPSUB and RPMERGE can also be seen by direct comparison of figures 4.16 and 4.17. According to (4.20), the distribution of Y_j—conditionally on X_j—is given by

$$\Pr[\, Y_j = k \mid X_j \,] = H(k;\, M_j, |S_j|, X_j)\,. \tag{4.23}$$

The RPMERGE algorithm thus generates X_1 according to (4.22), sets $X_2 = M - X_1$ and then generates Y_j according to (4.23). The complete pseudocode for the merging procedure is given in algorithm 4.5, where we make of the function HYPERGEOMETRIC to generate samples from the hypergeometric distribution; see section 4.2.1C.

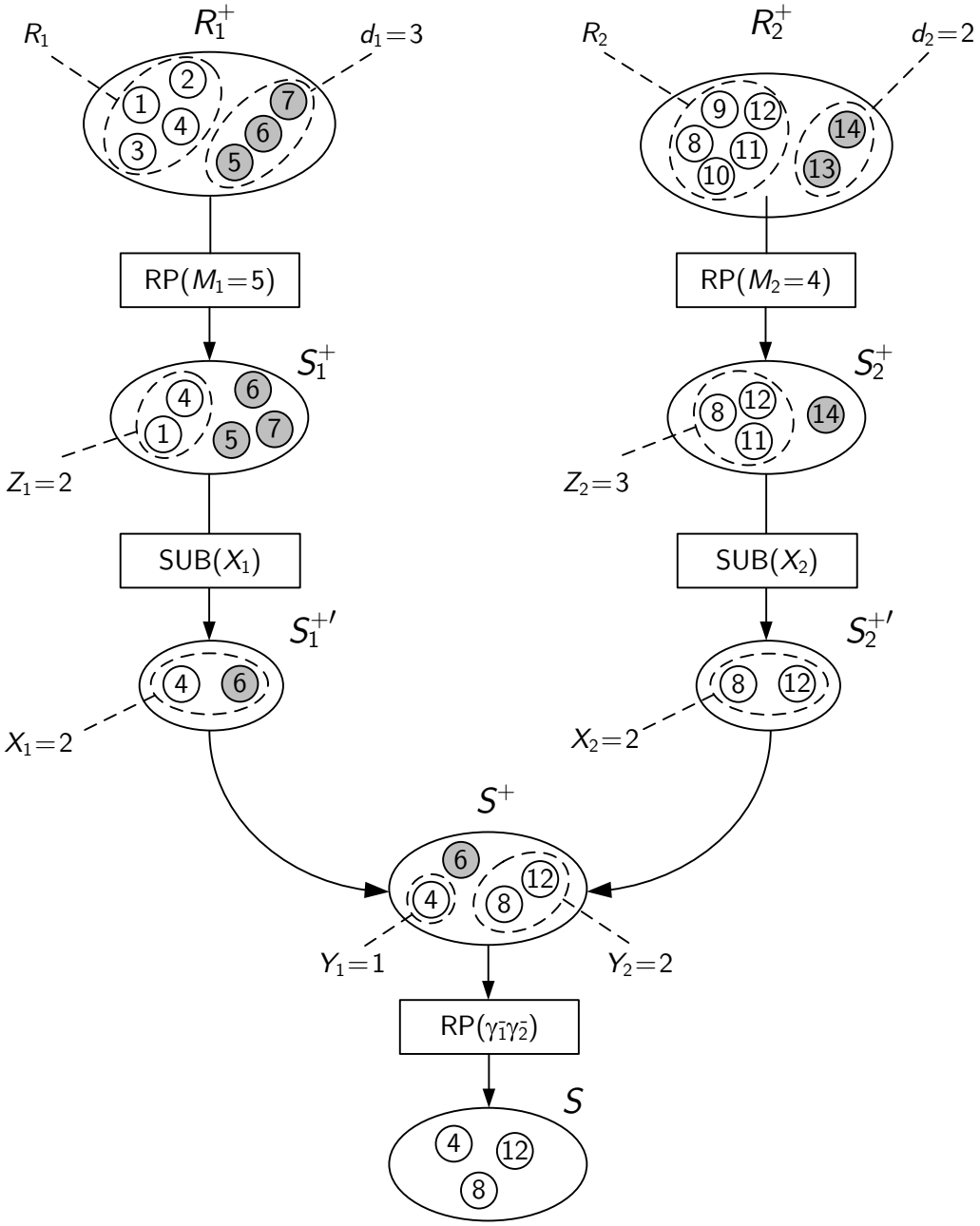


Figure 4.17: A hypothetical merging algorithm

B. Sample Size Properties

By theorem 4.2, the sample size $|S|$ produced by RPMERGE is hypergeometrically distributed with

$$\Pr[\,|S| = k\,] = H(k;\, |R_1| + |R_2| + d_1 + d_2, |R_1| + |R_2|, d_1 + d_2) \tag{4.24}$$

and

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

As indicated previously, the sample size produced by RPMERGE can be strictly larger than that produced by a naive application of MERGE. For example, ignoring the transient items in the scenario of figure 4.17, we see that RP produces samples $S_1 \subseteq R_1$ and $S_2 \subseteq R_2$ with $|S_1| = 2$ and $|S_2| = 3$, so that MERGE 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 MERGE. Indeed, the expected sample size for RPMERGE is often strictly larger.

Theorem 4.6. *Suppose that* $|R_j| > 0$ *and* $|R_j| + d_j > M_j$ *for* $j = 1, 2$. *If* $M_1 = M_2 = M$, *then*

$$\mathrm{E}[\,\min(|S_1|, |S_2|)\,] \le \mathrm{E}[\,|S|\,], \tag{4.25}$$

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 4.6, we need the following lemma.

Lemma 4.1. *For any random variables* X *and* Y, *we have*

$$\mathrm{E}[\,\min(X, Y)\,] \le \min(\mathrm{E}[\,X\,], \mathrm{E}[\,Y\,]),$$

and the above inequality is strict if $\Pr[\,X < Y\,] > 0$ *and* $\Pr[\,X > Y\,] > 0$.

Proof. Observe that

$$\begin{aligned}
\mathrm{E}[\,\min(X, Y)\,] &= \mathrm{E}[\,X I_{X \le Y} + Y I_{X > Y}\,] \\
&= \mathrm{E}[\,X\,] - \mathrm{E}[\,X I_{X > Y}\,] + \mathrm{E}[\,Y I_{X > Y}\,] \\
&= \mathrm{E}[\,X\,] - \mathrm{E}[\,(X - Y) I_{X > Y}\,],
\end{aligned}$$

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

Proof. (of theorem 4.6) 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 (4.25) holds with equality. Otherwise, suppose that $d_1 + d_2 > 0$ and, without loss of generality, that $\mathrm{E}[\,S_1\,] \leq \mathrm{E}[\,S_2\,]$. By theorem 4.2, 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 $\mathrm{E}[\,|S_1|\,] \leq \mathrm{E}[\,|S|\,]$, where equality holds if and only if $\mathrm{E}[\,|S_1|\,] = \mathrm{E}[\,|S_2|\,]$. Using lemma 4.1, we have

$$\begin{aligned} \mathrm{E}[\,\min(|S_1|, |S_2|)\,] &\leq \min\big(\mathrm{E}[\,|S_1|\,], \mathrm{E}[\,|S_2|\,]\big) && (4.26) \\ &= \mathrm{E}[\,|S_1|\,] \leq \mathrm{E}[\,|S|\,]. && (4.27) \end{aligned}$$

If $\mathrm{E}[\,|S_1|\,] < \mathrm{E}[\,|S_2|\,]$, then the inequality in (4.27), and hence in (4.25), is strict. Otherwise, we claim that $\Pr[\,|S_1| > |S_2|\,] > 0$ and $\Pr[\,|S_1| < |S_2|\,] > 0$, so that, by lemma 4.1, the inequality in (4.26)—and hence in (4.25)—is strict, and the desired result follows.

To see that the above claim holds, suppose that $\mathrm{E}[\,|S_1|\,] = \mathrm{E}[\,|S_2|\,]$. 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 4.2, $\Pr[\,|S_j| = k\,] > 0$ for $l_j \leq k \leq u_j$, and therefore

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

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

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

A symmetric argument shows that $\Pr[\,|S_1| < |S_2|\,] > 0$. □

4.3.3 Experiments

We compared the RPMERGE algorithm to the naive application of the MERGE algorithm. We found that:

- In most cases, RPMERGE produces significantly larger samples than MERGE.

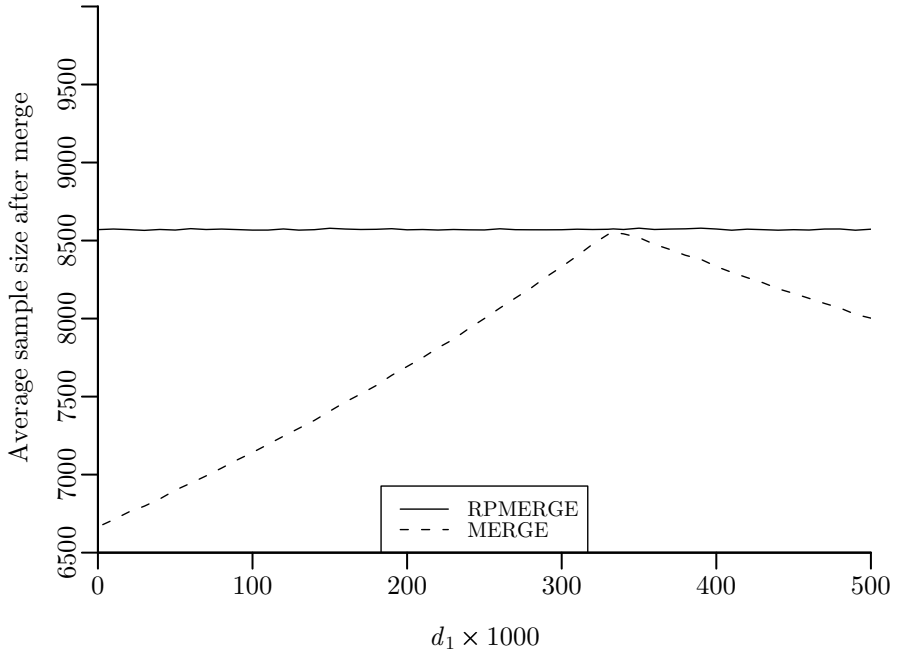


Figure 4.18: Merging, $d_1 + d_2$ fixed

- As predicted by theory—see (4.24)—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 MERGE grows as the total number of uncompensated deletions increases.

These results are not surprising: 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 MERGE. 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 4.18 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

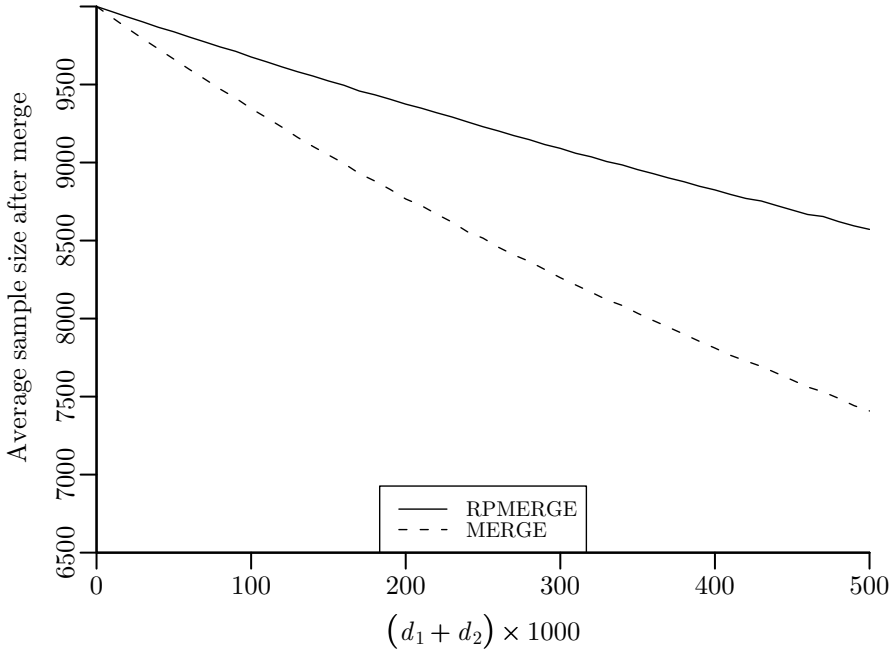


Figure 4.19: Merging, $d_1/(d_1 + d_2)$ fixed

of these experimental parameters. In contrast, MERGE produces samples whose size equals the smaller of the two input sample sizes. The best performance is achieved when $\mathrm{E}[\min(|S_1|, |S_2|)] \approx \min(\mathrm{E}[|S_1|], \mathrm{E}[|S_2|])$ is maximized. Since $\mathrm{E}[|S_j|] = M|R_j|/(|R_j|+d_j)$, this optimal performance is achieved when $d_1 \approx 333{,}333$. In this case, RPMERGE and MERGE 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 the 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 figure 4.19, 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 MERGE. 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.

4.4 Summary

We have systematically studied methods for maintaining bounded-size 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 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 can potentially be applied even in complex scenarios where approximations are not available. We also studied the inverse problem of reducing the size of the sample dynamically. Our RPSUB algorithm is provably uniform and can be used to free some sample space when resources become scarce. 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 MERGE 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 MERGE.

Chapter 5

Multiset Sampling

In this chapter,[1] we are concerned with maintenance schemes for evolving *multi*sets. Specifically we propose an algorithm for incrementally maintaining a Bernoulli sample over such a multiset. We derive improved estimators for use with our samples, and we discuss issues that arise when samples are resized or merged.

Our sampling scheme, as introduced in section 5.1, is called *augmented Bernoulli sampling*, ABERN(q), and can handle arbitrary insertion, update and deletion transactions without ever accessing the underlying base data. In order to maintain the sample, our method augments the sample with "tracking counters," originally introduced in Gibbons and Matias (1998) for the purpose of estimating the population frequencies of "hot" items. We show that these counters not only facilitate maintenance, but can also be exploited to obtain unbiased estimators of population frequencies, sums, and averages, where these estimators have lower variance than the usual estimators based on an ordinary Bernoulli sample. Furthermore, we show how to estimate the number of distinct items in the multiset in an unbiased manner. Our distinct-item estimator is based on the observation that a distinct-item Bernoulli sample can be extracted from our ABERN(q) sample; this is not possible with ordinary Bernoulli samples. We derive the standard error for each of our estimators, and provide formulas for estimating these standard errors.

In section 5.2, we discuss the problem of dynamically changing the sampling rate q of our augmented samples. We argue that increasing the value of q so as to enlarge the sample size requires a scan of almost the entire dataset. Fortunately, this situation occurs rarely in practice because ABERN(q) samples already grow linearly with the dataset size. A more interesting problem is to reduce the value of q in order to reduce the sample size and avoid oversized samples. We give a subsampling algorithm that performs such a reduction of q without accessing the base data; the resulting sample can be incrementally maintained. As discussed previously, subsampling cannot be used to enforce strict bounds on the sample size because such an approach would lead to non-uniform samples. However, we show how subsampling can be used to provide tight probabilistic bounds.

Finally, in section 5.3, we examine issues that arise when incrementally maintaining samples that result from merging operations. As a negative result, we show that, in

[1] The material in this chapter has been developed jointly with Peter J. Haas and Wolfgang Lehner. The chapter is based on Gemulla et al. (2007) with copyright held by ACM. The original publication is available at http://portal.acm.org/citation.cfm?id=1265530.1265544.

the general case of non-disjoint parent datasets, it is impossible to merge augmented samples to obtain a new augmented sample from the union of the corresponding parent datasets. However, the augmented samples can still be used to obtain a plain, non-maintainable Bernoulli sample from this union.

5.1 Uniform Sampling

The only previously known algorithm for maintaining a Bernoulli sample in the presence of insertion, update and deletion transactions is the MBERNM(q) scheme given in section 3.5.2A. Recall that MBERNM(q) stores a tuple $(r, X_i(r))$ for each sample item, where $X_i(r)$ is the frequency of r in the sample. When an insertion $+r$ arrives, the value of $X_i(r)$ is incremented with probability q and retained otherwise; an increase of $X_i(r)$ from 0 to 1 corresponds to the insertion of tuple $(r, 1)$ into the sample. In order to support deletion transactions, the scheme requires knowledge of the frequency $N_i(r)$ of r in the base data: When processing transaction $-r$, the value of $X_i(r)$ is decremented with probability $X_i(r)/N_i(r)$ and retained otherwise. Again, a decrease from $X_i(r)$ from 1 to 0 corresponds to the removal of item r from the sample. Since deletions of non-sampled items do not affect the sample, one would like somehow to maintain only the $N_i(r)$ counters corresponding to items r that are in the sample. Such maintenance is impossible without accessing the base data, because insertions into the dataset of an item that is not currently in the sample, but will eventually be in the sample, cannot be properly accounted for.

Our augmented Bernoulli sampling scheme improves upon MBERNM(q) in that it does not require knowledge of $N_i(r)$ to process deletion transactions. The sample can therefore be maintained without ever accessing the base data. We describe the scheme in section 5.1.1 and turn our attention to improved estimators for our samples in section 5.1.2.

5.1.1 Augmented Bernoulli Sampling

Borrowing an idea from Gibbons and Matias (1998), our new maintenance method rests on the fact that it suffices to maintain a "tracking counter" $Y_i(r)$ for each item r in the sample. Whenever $X_i(r)$ is positive, the counter $Y_i(r)$ records the number of net insertions of r into the dataset that have occurred since the insertion of the first of the current $X_i(r)$ sample items; the dataset insertion corresponding to the first of these $X_i(r)$ sample inclusions is counted as part of $Y_i(r)$. An example is shown in figure 5.2, where a sequence consisting of 8 insertions of item r has been used. In the figure, sample items and rejected items are shown as gray and white circles, respectively.

The general layout of the sample S_i is as follows: for each distinct item $r \in \mathscr{R}$ that occurs in the sample at least once, S_i contains the triple $(r, X_i(r), Y_i(r))$; the sample is therefore "augmented" with tracking counters. To save space, we store the entry for r as $(r, X_i(r), Y_i(r))$ if $Y_i(r) > 1$ and simply as (r) if $X_i(r) = Y_i(r) = 1$.

The resulting space savings can be significant when there are many unique values in the dataset.

A. Algorithmic Description

In Bernoulli sampling, each item is sampled independently from all the other items.[2] Without loss of generality, therefore, we fix an item r and focus on the maintenance of $X_i(r)$ and $Y_i(r)$ as the transaction sequence γ is processed. We assume in this section that γ consists solely of insertions and deletions of item r and we represent the state of S_i as (X_i, Y_i), that is, we suppress the dependence on r in our notation.[3] We have $X_i = Y_i = 0$ whenever $r \notin S_i$. As before, we assume that both the dataset and the sample are initially empty so that $X_0 = Y_0 = 0$.

The new algorithm works as follows: For an insertion transaction $\gamma_{i+1} = +r$, set

$$(X_{i+1}, Y_{i+1}) \leftarrow \begin{cases} (X_i + 1, Y_i + 1) & \text{if } \Phi_{i+1} = 1 \\ (X_i, Y_i + 1) & \text{if } \Phi_{i+1} = 0, X_i > 0 \\ (0, 0) & \text{otherwise,} \end{cases} \tag{5.1}$$

where Φ_{i+1} is a 0/1 random variable such that $\Pr[\,\Phi_{i+1} = 1\,] = q$. For a deletion $\gamma_{i+1} = -r$, set

$$(X_{i+1}, Y_{i+1}) \leftarrow \begin{cases} (0, 0) & \text{if } X_i = 0 \\ (0, 0) & \text{if } X_i = Y_i = 1 \\ (X_i - 1, Y_i - 1) & \text{if } X_i \geq 1, Y_i > 1, \Psi_{i+1} = 1 \\ (X_i, Y_i - 1) & \text{otherwise,} \end{cases} \tag{5.2}$$

where Ψ_{i+1} is a 0/1 random variable such that

$$\Pr[\,\Psi_{i+1} = 1\,] = \frac{X_i - 1}{Y_i - 1}.$$

Note that $\Pr[\,\Psi_{i+1} = 1\,] = 0$ whenever $X_i = 1$; we have $X_{i+1} \geq 1$ whenever $Y_i > 1$. As before, item r is removed from the sample if $X_i > 0, X_{i+1} = 0$ and added to the sample if $X_i = 0, X_{i+1} > 0$. The processing of γ_{i+1} is solely based on S_i and γ_{i+1}, that is, access to the dataset R is not required at any time.

The complete pseudocode of the algorithm is given as algorithm 5.1, where we write $S[r]$ to denote the (X_i, Y_i)-pair of item r.

[2] Thus our algorithm can also be used to maintain a Poisson sample, in which the inclusion probability q varies for each distinct item.

[3] An exception to this rule is made in algorithm 5.1, where we give the complete pseudocode of our algorithm.

Algorithm 5.1 Augmented Bernoulli sampling

```
1: q: the sampling rate
2: S: the augmented Bernoulli sample
3: RANDOM(): returns a uniform random number between 0 and 1
4:
5: INSERT(r):
6: if S contains r then
7:    (X, Y) ← S[r]
8:    if RANDOM() < q then                          // r ∈ S, insertion accepted
9:       X ← X + 1
10:   end if
11:   Y ← Y + 1
12:   S[r] ← (X, Y)
13: else if RANDOM() < q then                       // r ∉ S, insertion accepted
14:   S[r] ← (1, 1)                                 // add r to the sample
15: end if
16:
17: DELETE(r):
18: if S contains r then
19:   (X, Y) ← S[r]
20:   if Y = 1 then                                 // last seen occurrence of r
21:      remove r from S
22:   else                                          // more than one occurrence
23:      if RANDOM() < (X−1)/(Y−1) then             // deletion accepted
24:         X ← X − 1
25:      end if
26:      Y ← Y − 1
27:      S[r] = (X, Y)
28:   end if
29: end if
```

B. Example

Figure 5.1 depicts a probability tree for our algorithm with $\gamma = (+r, +r, +r, -r)$ and $q = 0.25$. Each node of the tree represents a possible state of the sample; for example, $(1, 2)$ stands for $S = \{ (r, 1, 2) \}$. As before, edges represent state transitions and are weighted by the respective transition probability. To determine the probability of reaching a given node, multiply the probabilities along the path from the root to the node. To compute the probability that the sample is in a specified state after γ has been processed, sum the probabilities of all leaf nodes that correspond to the state. For example,

$$\Pr[S_4 = \emptyset] = \left(\frac{3}{4}\right)^2 \frac{1}{4} + \left(\frac{3}{4}\right)^3 .$$

Summing up all such probabilities, we find that

$$\begin{aligned}
\Pr[X_4 = 0] &= \frac{9}{16} = (1-q)^2 = B(0;\, 2, q) \\
\Pr[X_4 = 1] &= \frac{6}{16} = 2q(1-q) = B(1;\, 2, q) \\
\Pr[X_4 = 2] &= \frac{1}{16} = q^2 = B(2;\, 2, q) .
\end{aligned}$$

Thus X_4 is binomially distributed, and we have a Bernoulli sample of R_4.

C. Proof of Correctness

To show that ABERN(q) indeed maintains a true Bernoulli sample, we first derive the probability distribution of the tracking counter Y_i. Recall that N_i denotes the multiplicity of item r in dataset R_i or, equivalently, the number of "non-annihilated" insertion transactions after processing the sequence $\gamma_1, \gamma_2, \ldots, \gamma_i$.

Lemma 5.1. *For $i \geq 0$ and any integer $0 \leq k \leq N_i$,*

$$\Pr[Y_i = m] = \begin{cases} (1-q)^{N_i} & \text{if } m = 0 \\ q(1-q)^{N_i - m} & \text{otherwise.} \end{cases} \tag{5.3}$$

Proof. Our proof is by induction on i. We have $Y_i = N_i = 0$ when $i = 0$, and (5.3) holds trivially. Suppose for induction that (5.3) holds for i. If transaction γ_{i+1} is an insertion, then $N_{i+1} = N_i + 1$ and

$$\Pr[Y_{i+1} = m] = \begin{cases} (1-q)\Pr[Y_i = 0] & \text{if } m = 0 \\ q \Pr[Y_i = 0] & \text{if } m = 1 \\ \Pr[Y_i = m-1] & \text{otherwise.} \end{cases}$$

If transaction γ_{i+1} is a deletion, then $N_{i+1} = N_i - 1$ and

$$\Pr[Y_{i+1} = m] = \begin{cases} \Pr[Y_i = 0] + \Pr[Y_i = 1] & \text{if } m = 0 \\ \Pr[Y_i = m+1] & \text{otherwise.} \end{cases}$$

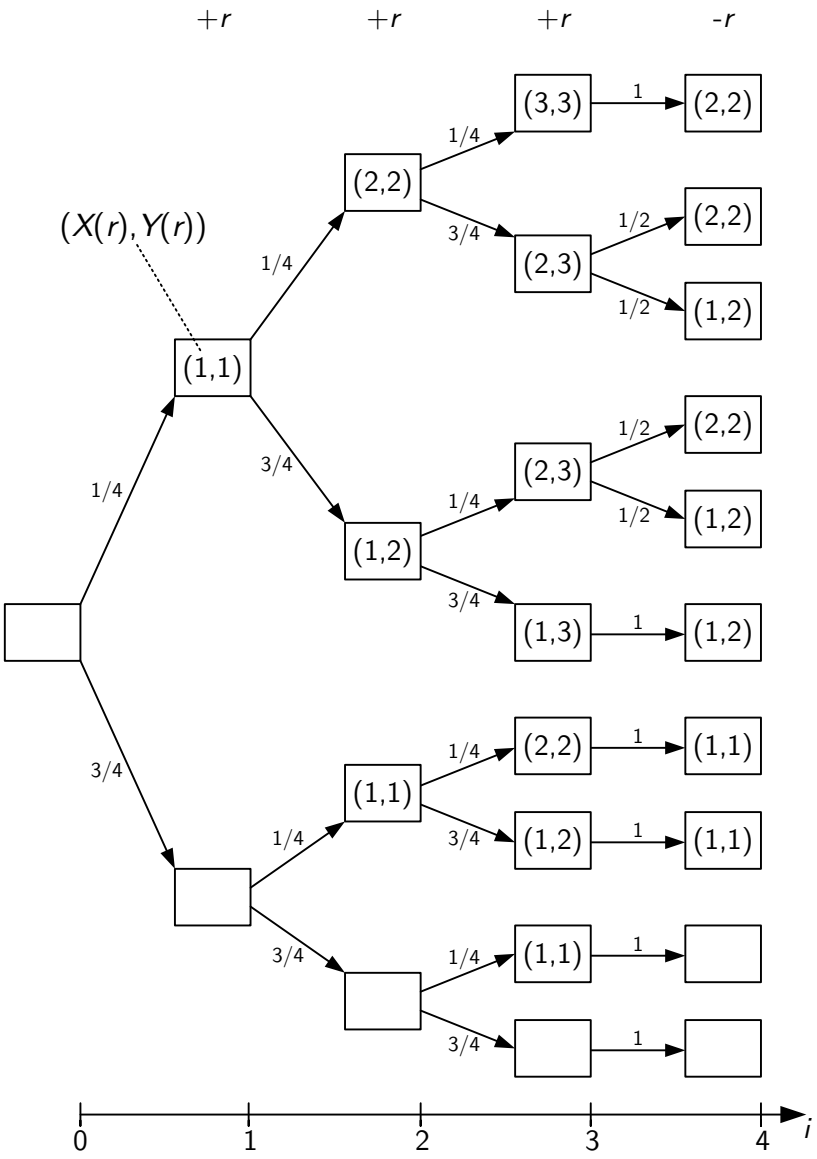


Figure 5.1: Illustration of augmented Bernoulli sampling with $q = 25\%$

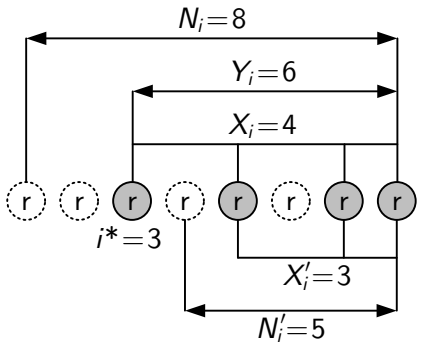


Figure 5.2: Notation used in the proof of theorem 5.1

The identity in (5.3) now follows by applying the induction hypothesis, together with some straightforward algebra. □

The assertion of lemma 5.1 is particularly easy to understand in the special case where all transactions are insertions, so that there are no annihilated transactions and $N_i = i$. In this scenario, $Y_i = 0$ if all i inserted items are excluded from the sample, which occurs with probability $(1-q)^i$, and $Y_i = m > 0$ if the first $i-m$ items are excluded and the next item is included, which occurs with probability $q(1-q)^{i-m}$.

We now establish the correctness of our sample-maintenance algorithm. Recall that

$$B(k;\, N, q) = \binom{N}{k} q^k (1-q)^{N-k}$$

denotes a binomial probability. In fact, $B(k;\, N, q)$ corresponds to the probability that exactly k heads occur when flipping N weighted coins with probability of heads q. Figure 5.2 illustrates the notation used in the proof of the following theorem.

Theorem 5.1. *For $i \geq 0$ and any integer $0 \leq k \leq N_i$, ABERN(q) maintains the invariant*

$$\Pr[\, X_i = k \,] = B(k;\, N_i, q)\,. \tag{5.4}$$

Proof. Although we could prove the validity of (5.4) by enhancing the inductive proof of lemma 5.1, we choose to give an intuitive probabilistic argument that provides insight into why the algorithm works. Fix $i \geq 0$ and observe that $X_i = 0$ if and only if $Y_i = 0$, so that, using (5.3), we have

$$\Pr[\, X_i = 0 \,] = \Pr[\, Y_i = 0 \,] = B(0;\, N_i, q)\,,$$

as asserted in (5.4).

Now consider a scenario in which $X_i \geq 1$, and denote by i^* the index of the transaction at which the first of the X_i instances of item r was inserted into the sample. It follows from the definition of i^* that, for $j = i^*+1, i^*+2, \ldots, i$, the

cumulative number of insertions in the sequence $\gamma_{i^*,j} = (\gamma_{i^*+1}, \gamma_{i^*+2}, \ldots, \gamma_j)$ is at least as great as the cumulative number of deletions, so that the net number of insertions is always nonnegative. We can therefore view each deletion in $\gamma_{i^*,i}$ as annihilating a previous insertion that is also an element of $\gamma_{i^*,i}$, and the net effect of processing the transactions in $\gamma_{i^*,i}$ can be viewed as inserting the set of non-annihilated "new" items into the dataset R_{i^*}. For $j = i^* + 1, i^* + 2, \ldots, i$, denote by R'_j and S'_j the set of non-annihilated new items in the dataset and sample, respectively, just after processing transaction γ_j, and set $N'_j = |R'_j|$ and $X'_j = |S'_j|$. Observe that $X'_j = X_j - 1 \geq 0$, since the sample S_j consists of the single "old" item that was included at transaction γ_{i^*}, together with the $X_j - 1$ new items in S'_j. Also observe that $N'_j = Y_j - 1$, since Y_j counts the item inserted into the dataset at γ_{i^*} together with the non-annihilated new items inserted into the dataset while processing the transaction sequence $\gamma_{i^*,j}$.

We claim that X'_i can be viewed as the size of a sample S'_i that is obtained from R'_i by processing the transactions in $\gamma_{i^*,i}$ using the MBERNM(q) algorithm of section 3.5.2. Indeed, for $j = i^*, i^* + 1, \ldots, i - 1$, suppose that transaction γ_{j+1} is an insertion. It follows from (5.1) that X_j, and hence X'_j, is incremented if and only if the random variable Φ_{j+1} equals 1, i.e.,

$$X'_{j+1} \leftarrow \begin{cases} X'_j + 1 & \text{with probability } q \\ X'_j & \text{with probability } 1 - q. \end{cases} \tag{5.5}$$

If γ_{j+1} is a deletion, then, by (5.2),

$$X'_{j+1} \leftarrow \begin{cases} X'_j - 1 & \text{if } \Psi_{i+1} = 1 \\ X'_j & \text{otherwise,} \end{cases} \tag{5.6}$$

where

$$\Pr[\,\Psi_{j+1} = 1\,] = \frac{X_j - 1}{Y_j - 1} = \frac{X'_j}{N'_j}.$$

Our claim follows upon comparison of (5.5) to (3.16) and of (5.6) to (3.17).

The foregoing claim and the correctness of MBERNM(q)—as asserted in Gemulla et al. (2007)—together imply that S'_i is a true Bernoulli sample of R'_i. We therefore have

$$\begin{aligned} \Pr[\,X_i = k \mid Y_i = m\,] &= \Pr\left[\,X'_i = k - 1 \mid N'_i = m - 1\,\right] \\ &= B(k - 1;\, m - 1, q) \end{aligned} \tag{5.7}$$

for $1 \leq k \leq m \leq N_i$. Combining (5.3) and (5.7), we find that

$$\begin{aligned}
\Pr[\, X_i = k \,] &= \sum_{m=k}^{N_i} \Pr[\, X_i = k \mid Y_i = m \,] \Pr[\, Y_i = m \,] \\
&= \sum_{m=k}^{N_i} B(k-1;\, m-1, q)\, q(1-q)^{N_i - m} \\
&= q^k (1-q)^{N_i - k} \sum_{m=k}^{N_i} \binom{m-1}{k-1} \\
&= B(k;\, N_i, q)
\end{aligned}$$

for $k \geq 1$, and the desired result follows. □

Thus, the new algorithm works by tracking the net number of insertions into the dataset only after the item is first inserted into the sample, i.e., from transaction γ_{i^*} onwards. As mentioned above, this idea originally appeared as part of the counting-sample method in Gibbons and Matias (1998). A counting sample comprises pairs of the form $(r, Y_i(r))$, where $Y_i(r)$ is defined as above. As mentioned by Gibbons and Matias (1998), a Bernoulli sample can be extracted from a counting sample via a coin-flipping step. Our algorithms amortize this subsampling cost over all transactions, thereby facilitating faster on-demand materialization of the sample, and hence faster production of estimates based on the sample. Since we maintain the value of X_i directly instead of randomly generating it every time the sample is accessed, we expect that estimates derived from our augmented Bernoulli sample are more stable statistically, especially when they are computed frequently.

5.1.2 Estimation

In this section, we derive unbiased estimators for item frequencies, sums, averages, ratios, and distinct-item counts. We show how to exploit the tracking counters to obtain a variance reduction.

A. Frequencies

We again assume a single item r. Ignoring the tracking counters in our augmented Bernoulli sample, the frequency N_i of an item r can be estimated as $\hat{N}_{X_i} = X_i/q$. This estimator is the standard Horvitz-Thompson (HT) estimator for a Bernoulli sample from R; see section 2.1.3. The HT estimator $\hat{N}_{X_i}$ is unbiased and has variance

$$\mathrm{Var}[\, \hat{N}_{X_i} \,] = (1-q)N_i/q.$$

To motivate our improved estimator, write

$$\hat{N}_{X_i} = \frac{X_i - 1}{q} + \frac{1}{q}. \tag{5.8}$$

Recall the definitions of i^* and R'_i from the proof of theorem 5.1, figure 5.2. From (5.3), it follows that the number $L_i = N_i - Y_i$ of (non-annihilated) items inserted into R during the processing of $\gamma_1, \gamma_2, \ldots, \gamma_{i^*}$ has a geometric distribution:

$$\Pr[L_i = l] = q(1-q)^{l-1} \tag{5.9}$$

for $l \geq 1$. The expected number of items is therefore $\mathrm{E}[L_i] = 1/q$. Thus the second term in (5.8) is an estimator of the number of items in $R_{i^*} = R_i \setminus R'_i$. As described in the proof of theorem 5.1, $X_i - 1$ is the size of a Bernoulli sample from R'_i, so that the first term in (5.8) is simply the (unbiased) HT estimator of the frequency of r in R'_i. Since we have maintained an augmented sample $S_i = \{(X_i, Y_i)\}$, however, we know the value of the number of items inserted into R'_i *exactly*: this number is simply $Y_i - 1$. Thus, intuitively, we can reduce the variance of the estimator $\hat{N}_{X_i}$ by replacing the first term in (5.8) by the quantity that it is trying to estimate, yielding the improved estimator $Y_i - 1 + (1/q)$. This estimator is not quite correct, however, because there is a positive probability that $Y_i = 0$, in which case the above reasoning does not hold. When $Y_i = 0$, item r is not in the sample, and we have no information at all about N_i. The simplest choice in this case is to estimate N_i as 0, just as in the HT estimator; we show below that this choice ensures unbiasedness. Thus the final form of our improved estimator[4]

$$\hat{N}_{Y_i} = \begin{cases} 0 & \text{if } Y_i = 0 \\ Y_i - 1 + (1/q) & \text{otherwise} \end{cases} \tag{5.10}$$

The following result shows that $\hat{N}_{Y_i}$ is indeed unbiased.

Theorem 5.2. $\mathrm{E}[\hat{N}_{Y_i}] = N_i$ *for* $i \geq 0$.

Proof. Fixing r and suppressing the subscript i in our notation, we have

$$\begin{aligned} \mathrm{E}[\hat{N}_Y] &= \Pr[Y = 0]\,\mathrm{E}[\hat{N}_Y \mid Y = 0] + \Pr[Y > 0]\,\mathrm{E}[\hat{N}_Y \mid Y > 0] \\ &= \Pr[Y > 0]\,\mathrm{E}[Y \mid Y > 0] + \Pr[Y > 0]\,\frac{1-q}{q} \end{aligned} \tag{5.11}$$

Thus, we have to compute $\Pr[Y > 0]$ and $\mathrm{E}[Y \mid Y > 0]$. From (5.3), the former quantity is given by

$$\Pr[Y > 0] = 1 - \Pr[Y = 0] = 1 - (1-q)^N. \tag{5.12}$$

and the latter quantity by

$$\begin{aligned} \mathrm{E}[Y \mid Y > 0] &= \sum_{m=1}^{N} m \Pr[Y = m \mid Y > 0] = \sum_{m=1}^{N} m \frac{\Pr[Y = m]}{\Pr[Y > 0]} \\ &= \sum_{m=1}^{N} \frac{mq(1-q)^{N-m}}{1-(1-q)^N} = \frac{Nq - (1-q)\big(1-(1-q)^N\big)}{q\big(1-(1-q)^N\big)}. \end{aligned} \tag{5.13}$$

[4] We recently learned that both $\hat{N}_{Y_i}$ and $\mathrm{Var}[\hat{N}_{Y_i}]$ have been derived independently by Estan and Naughton (2006, app. B) in the context of join-size estimation.

To derive the final equality, observe that

$$\sum_{i=1}^{N} i x^{N-i} = \sum_{i=0}^{N-1} (N-i) x^i = N \sum_{i=0}^{N-1} x^i - \sum_{i=0}^{N-1} i x^i = N \sum_{i=0}^{N-1} x^i - \sum_{i=1}^{N-1} \sum_{j=i}^{N-1} x^j$$

and apply the identity $\sum_{i=0}^{N-1} x^i = N(1-x^N)/(1-x)$ of the geometric series to obtain

$$\sum_{i=1}^{N} i x^{N-i} = \frac{N(1-x) - x(1-x^N)}{(1-x)^2}.$$

The assertion of the theorem follows after substituting (5.12) and (5.13) into (5.11). □

Calculations similar to the proof of theorem 5.2 show that the variance of $\hat{N}_Y$ is given by

$$\mathrm{Var}[\,\hat{N}_Y\,] = \frac{1-q-(1-q)^{N+1}}{q^2}, \tag{5.14}$$

where we continue to suppress i in our notation. Note that $\mathrm{Var}[\,\hat{N}_Y\,] < (1-q)/q^2$, i.e., the variance is bounded from above in N. The reason for this advantageous behavior is that, as N grows, the value of the estimator $\hat{N}_Y$ becomes increasingly dominated by the value Y, a quantity that embodies exact knowledge.

We now compare the variance of the unbiased estimators $\hat{N}_X$ and $\hat{N}_Y$.

Theorem 5.3. $\mathrm{Var}[\,\hat{N}_Y\,] \leq \mathrm{Var}[\,\hat{N}_X\,]$, *with equality holding only if* $N = 0$ *or* 1.

Proof. Starting with the well-known Bernoulli inequality

$$(1-q)^N \geq 1 - Nq, \tag{5.15}$$

we find that

$$N \geq \frac{1-(1-q)^N}{q}.$$

The desired result follows after multiplying both sides of the above inequality by $(1-q)/q$, and observing that equality holds in (5.15) only when N equals 0 or 1. □

For fixed q, we have $\mathrm{Var}[\,\hat{N}_Y\,]/\,\mathrm{Var}[\,\hat{N}_X\,] \approx 1/(qN)$ as N becomes large, and the variance reduction can be substantial.

Following standard statistical practice, we can estimate $\mathrm{Var}[\,\hat{N}_Y\,]$ using the (biased) estimator

$$\hat{\mathrm{Var}}[\,\hat{N}_Y\,] = \frac{1-q-(1-q)^{\hat{N}_Y+1}}{q^2},$$

The bias of the estimator converges to 0 as $q \to 1$.

B. Sums, Averages, and Ratios

The foregoing results for frequencies lead immediately to unbiased estimators for sums and averages, as well as estimator for ratios. In particular, suppose we are given a function $f : \mathscr{R} \mapsto \mathbb{R}$ and we wish to estimate the sum of $f(r)$ over all items r in the dataset. That is, we wish to estimate $\alpha(f) = \sum_{r\in\mathscr{R}} f(r)N(r)$, where $N(r)$ is the multiplicity of item r in R. The standard HT estimator of $\alpha(f)$ is

$$\hat{\alpha}_X(f) = \sum_{r\in D(S)} f(r)\hat{N}_X(r) = \sum_{r\in\mathscr{R}} f(r)\hat{N}_X(r),$$

where $\hat{N}_X(r)$ corresponds to the estimator $\hat{N}_X$ in (5.8), evaluated with respect to item r, and $D(S)$ denotes the set of distinct items in the sample. The linearity of the expectation operator immediately implies that $\mathrm{E}[\,\hat{\alpha}_X(f)\,] = \alpha(f)$, so that $\hat{\alpha}_X$ is unbiased. Because items are sampled independently, the estimators $\{\,\hat{N}_X(r) : r \in \mathscr{R}\,\}$ are mutually independent, and

$$\mathrm{Var}[\,\hat{\alpha}_X(f)\,] = \sum_{r\in\mathscr{R}} f^2(r)\frac{(1-q)N(r)}{q}.$$

Similarly, an improved estimator is given by

$$\hat{\alpha}_Y(f) = \sum_{r\in D(S)} f(r)\hat{N}_Y(r).$$

It follows from theorem 5.2 that $\hat{\alpha}_Y(f)$ is unbiased, and, by (5.14),

$$\mathrm{Var}[\,\hat{\alpha}_Y(f)\,] = \sum_{r\in\mathscr{R}} f^2(r)\frac{1-q-(1-q)^{N(r)+1}}{q^2}.$$

Theorem 5.3 implies that $\mathrm{Var}[\,\hat{\alpha}_Y(f)\,] \le \mathrm{Var}[\,\hat{\alpha}_X(f)\,]$. We can obtain a natural (biased) estimator of $\mathrm{Var}[\,\hat{\alpha}_Y(f)\,]$ as

$$\hat{\mathrm{Var}}[\,\hat{\alpha}_Y(g)\,] = \sum_{r\in D(S)} f^2(r)\frac{1-q-(1-q)^{\hat{N}_Y(r)+1}}{q^2\big(1-(1-q)^{\hat{N}_Y(r)}\big)}.$$

This estimator is "almost" the HT estimator of $\mathrm{Var}[\,\hat{\alpha}_Y(f)\,]$, except that $N(r)$ is replaced by its estimate $\hat{N}_Y(r)$ in each term of the sum.

The foregoing results extend in a straightforward way to averages of the form $\mu = (1/|R|)\sum_{r\in\mathscr{R}} f(r)N(r)$, where $|R| = \sum_{r\in\mathscr{R}} N(r)$. Since $|R|$ is usually known in applications or maintained with the sample, it can be treated as a deterministic constant, adding a multiplicative factor of $1/|R|$ to the estimators and a factor of $1/|R|^2$ to the variances and variance estimators.

A less trivial scenario arises when estimating a ratio of the form

$$\rho = \frac{\sum_{r\in\mathscr{R}} f(r)N(r)}{\sum_{r\in\mathscr{R}} g(r)N(r)} = \frac{\alpha(f)}{\alpha(g)},$$

where f and g are arbitrary real-valued functions on $\mathscr{R}$. As special case of such an estimator, take g to be a 0/1 function that corresponds to a predicate defined over $\mathscr{R}$, and take $f(t) = f^*(r)g(r)$, where f^* is an arbitrary real-valued function on $\mathscr{R}$. Then ρ corresponds to the average value of f^* over those elements of R that satisfy the predicate corresponding to g. The ratio estimation problem has been extensively studied (Särndal et al. 1991), and an exhaustive discussion is beyond the scope of this thesis; we content ourselves here with briefly presenting some of the most pertinent results. The usual ratio estimator

$$\hat{\rho}_\Theta = \frac{\hat{\alpha}_\Theta(f)}{\hat{\alpha}_\Theta(g)} = \frac{\sum_{r \in D(S)} f(r)\hat{N}_\Theta(r)}{\sum_{r \in D(S)} g(r)\hat{N}_\Theta(r)},$$

where Θ equals X or Y, is biased, but the bias converges to 0 as q increases. A number of schemes have been proposed to reduce the bias when q is very small; see Särndal et al. (1991). When q is not too small, a Taylor-series argument yields an approximate expression for $\mathrm{Var}[\,\hat{\rho}_\Theta\,]$:

$$\mathrm{Var}[\,\hat{\rho}_\Theta\,] \approx \frac{1}{\alpha^2(g)}\Big(\mathrm{Var}\,[\,\hat{\alpha}_\Theta(f)\,] + \rho^2\,\mathrm{Var}\,[\,\hat{\alpha}_\Theta(g)\,] - 2\rho\,\mathrm{Cov}\,[\,\hat{\alpha}_\Theta(f), \hat{\alpha}_\Theta(g)\,]\Big),$$

where $\mathrm{Cov}\,[\,W, Z\,]$ denotes the covariance of random variables W and Z. Note that, using the independence of the sampling for different distinct items, we have

$$\mathrm{Cov}\,[\,\hat{\alpha}_\Theta(f), \hat{\alpha}_\Theta(g)\,] = \sum_{r \in \mathscr{R}} f(r)g(r)\,\mathrm{Var}[\,\hat{N}_\Theta(r)\,]$$

which can be estimated by

$$\hat{\mathrm{Cov}}\,[\,\hat{\alpha}_\Theta(f), \hat{\alpha}_\Theta(g)\,] = \sum_{r \in D(S)} f(r)g(r)\frac{1 - q - (1-q)^{\hat{N}_Y(r)+1}}{q^2\big(1 - (1-q)^{\hat{N}_Y(r)}\big)}$$

The usual method for estimating the variance $\mathrm{Var}[\,\hat{\rho}_\Theta\,]$ simply replaces ρ by $\hat{\rho}_\Theta$, $\alpha(g)$ by $\hat{\alpha}_\Theta(g)$, Cov by $\hat{\mathrm{C}}\mathrm{ov}$, and each Var by $\hat{\mathrm{V}}\mathrm{ar}$ in the formula for $\mathrm{Var}[\,\hat{\rho}_\Theta\,]$. Of course, we can always obtain standard errors or estimators of standard errors by taking the square root of the corresponding variances or estimated variances.

C. Distinct-Item Counts

In this section we show that, perhaps surprisingly, an augmented Bernoulli sample can also be used to estimate the number of distinct items in R. Although an augmented Bernoulli sample will probably not perform as well as synopses that are designed specifically for this task—see Gibbons (2009) for an overview of specialized methods—the techniques in this section can be useful when special-purpose synopses are not available.

Given an augmented Bernoulli sample S, define an (ordinary) random subset $S' \subseteq \mathscr{R}$ by examining each $r \in D(S)$ and including r in S' with probability $p(r)$, where

$$p(r) = \begin{cases} 1 & Y(r) = 1 \\ q & Y(r) > 1. \end{cases}.$$

Denote by $D(R)$ the set of distinct items in R.

Theorem 5.4. *The random subset S' is a BERN(q) sample of $D(R)$.*

Proof. Fix an item r and observe that $\Pr[\, r \in S' \,] = 0$ if r is not in S, and hence if $Y(r) = 0$. Then, writing Y for $Y(r)$ and using (5.3),

$$\begin{aligned} \Pr\big[\, r \in S' \,\big] &= \Pr[\, Y = 1 \,] + q \Pr[\, Y > 1 \,] \\ &= \Pr[\, Y = 1 \,] + q\big(\Pr[\, Y > 0 \,] - \Pr[\, Y = 1 \,]\big) \\ &= (1 - q) \Pr[\, Y = 1 \,] + q \Pr[\, Y > 0 \,] \\ &= (1 - q) q (1 - q)^{N-1} + q\big(1 - (1 - q)^N\big) = q \end{aligned}$$

Since items are included or excluded from S' independently of each other, the desired result holds. □

We can therefore obtain an unbiased estimate of $D = |D(R)|$, the number of distinct items in R, by using a standard HT estimator, namely,

$$\hat{D}_{\mathrm{HT}} = \frac{|S'|}{q}.$$

The variance of this estimator is $(1 - q)D/q$ and can be estimated by $(1 - q)\hat{D}_{\mathrm{HT}}/q$.

We propose a different unbiased estimator here, which estimates D directly from S and yields lower variance. The idea is to apply the "conditional Monte Carlo principle," which asserts that, if W is an unbiased estimator of some unknown parameter θ and Z is a random variable that represents "additional information," then the random variable $W' = \mathrm{E}[\, W \mid Z \,]$ is a better estimator than W in that, by the law of total expectation, $\mathrm{E}[\, W' \,] = \mathrm{E}\big[\, \mathrm{E}[\, W \mid Z \,] \,\big] = \mathrm{E}[\, W \,] = \theta$, and, moreover, $\mathrm{Var}[\, W' \,] \leq \mathrm{Var}[\, W \,]$. The variance reduction is a consequence of the well known and easily derived variance decomposition

$$\mathrm{Var}[\, W \,] = \mathrm{E}\big[\, \mathrm{Var}[\, W \mid Z \,] \,\big] + \mathrm{Var}\big[\, \mathrm{E}[\, W \mid Z \,] \,\big],$$

which holds for any two random variables W and Z. To apply this principle, we take $W = \hat{D}_{\mathrm{HT}}$ and $Z = S$, so that our proposed estimator is $\hat{D}_Y = \mathrm{E}[\, \hat{D}_{\mathrm{HT}} \mid S \,]$. We can

express $\hat{D}_Y$ in a more tractable form, as follows. Denote by I_A the indicator variable that equals 1 if event A occurs and equals 0 otherwise. Then

$$\begin{aligned}\hat{D}_Y &= \mathrm{E}\,[\,\hat{D}_{\mathrm{HT}} \mid S\,] = \mathrm{E}\left[\,\frac{1}{q}\sum_{r\in D(R)} I_{r\in S'} \;\middle|\; S\,\right] \\ &= \frac{1}{q}\sum_{r\in D(R)} \Pr[\,r\in S' \mid S\,] = \frac{1}{q}\sum_{r\in D(R)} I_{r\in D(S)}\Pr[\,r\in S' \mid Y(t)\,] \\ &= \frac{1}{q}\sum_{r\in D(R)} I_{r\in D(S)}p(r) = \frac{1}{q}\sum_{r\in D(S)} p(r).\end{aligned}$$

Intuitively, each distinct item in S is included in S' with probability $p(r)$ and the estimator $\hat{D}_{\mathrm{HT}}$ estimates D by counting all included items followed by a division by q. On average, every item contributes a quantity of $p(r)/q$ to the distinct-item count, and $\hat{D}_Y$ simply adds up the expected contributions. As shown above, $\hat{D}_Y$ is unbiased for D. Moreover, the variance of this estimator can be explicitly computed as

$$\begin{aligned}\mathrm{Var}[\,\hat{D}_Y\,] &= \sum_{r\in D(R)} (1-q)^{N(r)}/q \\ &\le \sum_{r\in D(R)} (1-q)/q = (1-q)D/q = \mathrm{Var}[\,\hat{D}_{\mathrm{HT}}\,],\end{aligned}$$

where equality holds if and only if $D = |R|$. In the usual way, the variance of $\hat{D}_Y$ can be estimated by the "almost" HT estimator

$$\hat{\mathrm{Var}}[\,\hat{D}_Y\,] = \sum_{r\in D(S)} \frac{(1-q)^{\hat{N}_Y(t)}}{q\big(1-(1-q)^{\hat{N}_Y(t)}\big)},$$

where $\hat{N}_Y(r)$ is the estimator of $N(r)$ given in (5.10). Note that the memory requirement for our technique is unbounded, unlike the estimator in Gibbons (2001) and other specialized distinct-item estimation methods as in Gibbons (2009).

As a final note, observe that—for practical values of q—we cannot extract a BERN(q) sample of $D(R)$ from a plain, non-augmented Bernoulli sample, that is, by accessing $X(r)$ only.

Theorem 5.5. *For $0 < q < 0.5$, there exists no algorithm that can extract a BERN(q) sample of $D(R)$ from a BERN(q) sample of any multiset R.*

Proof. The proof is by contradiction. Suppose that there is an algorithm A that takes as input a BERN(q) sample S from R and outputs a BERN(q) sample S' from $D(R)$. Observe that A must output item r with probability 0 if $r \notin R$ and with probability q if $r \in R$. In the following, we derive the probability function $p(x)$, which corresponds to the probability that A outputs r when $X(r) = x$. The assertion of the theorem follows because, for certain x, $p(x)$ does not denote a valid probability when $0 < q < 0.5$.

Set $\mathscr{R} = \{ r_0, r_1, r_2, \dots \}$ and suppose that R contains each item r_i exactly i items. We can use dataset R to determine the values of $p(x)$. Clearly, we have $p(0) = 0$ because A must not output r_0 and $X(r_0) = 0$ with probability 1, see (5.4). For $N > 0$, we have

$$\Pr[\, A \text{ outputs } r_N \,] = \sum_{x=0}^{N} p(x) \Pr[\, X(r_N) = x \,] = \sum_{x=0}^{N} p(x) B(x;\, N, q) = q, \quad (5.16)$$

where the last inequality follows from the assumed correctness of A. We can use (5.16) to determine the values of $p(1), p(2), \dots$, in that order, by setting $N = 1, 2, \dots$, respectively. We find that

$$p(x) = \begin{cases} 0 & x = 0 \\ 1 & x = 1 \\ (2q - 1)/q & x = 2 \\ \dots \end{cases}.$$

Now observe that $p(2) < 0$ for $q \in (0, 0.5)$. The assertion of the theorem follows because $p(2)$ does not denote a valid probability. □

The probabilities $p(x)$ seem to be valid when $q \geq 0.5$ but we did not further investigate this issue.

5.2 Sample Resizing

We now turn attention to the the problem of resizing an ABERN(q) sample dynamically. Since, in expectation, the sample grows linearly with the size of the dataset and since many estimators require only sublinear growth of the dataset to maintain their accuracy, resizing the sample upwards is typically of limited interest in practice. If such a resize is still required—that is, we want to switch to a sampling rate $q' > q$—, it appears that a scan of almost the entire base data is unavoidable. The reason is that the draw-sequential model that we used for resizing an RP(M) sample (section 4.2.1) cannot be used with ABERN(q) or, more general, multisets. To see this, suppose that $R = \{ a, b, b \}$ and $S = \{ a, b \}$. Further suppose that we want to increase the size of S to 3 by drawing random items from R. The problem is that in order to determine whether all copies of the selected item are already in the sample (a drawn) or the not (b drawn), we require knowledge of the *exact* frequency of the item in the base data. Since this frequency is expensive to obtain, draw-sequential methods are expected to be inefficient. Thus, whenever the sample is resized upwards, we compute the new ABERN(q') sample from scratch using a complete scan of the underlying dataset.

The remainder of this section is concerned with the subsampling problem: Given an ABERN(q) sample S of a dataset R, derive an ABERN(q') sample S' from R, where $q' < q$, without accessing R. Subsampling has applications in practice whenever the

sampling process is run on a system with bounded processing or space capabilities. In more detail, whenever some criterion is satisfied, one may reduce the sampling rate in order to effectively reduce the size of the sample. Before we discuss algorithms for subsampling, we briefly point out restrictions on the nature of the aforementioned criterion. This discussion applies to arbitrary Bernoulli samples, not just ABERN(q) samples.

A. Controlling the Sample Size

Several techniques in literature make use of subsampling to enforce an upper bound on the sample size (Gibbons and Matias 1998; Brown and Haas 2006; Tao et al. 2007). Though subsampling itself maintains the uniformity of the sample, this does not hold when subsampling is used to *bound* the size of an evolving sample; see the discussion in section 3.5.1B. The reason is that usage of subsampling in this fashion in effect "converts" the Bernoulli sample to a bounded-size sample, and Bernoulli maintenance algorithms cannot be used anymore. Therefore, we cannot use subsampling to enforce a strict upper bound on the sample size.

However, subsampling can be used to enforce a *probabilistic* bound on the sample size. The trick is to make the decision of whether or not to initiate subsampling dependent on only the size of the *dataset*, and therefore independent from the size or composition of the sample. Note that we do not require that the upper bound is fixed for all times; we simply treat it as a (monotonically increasing) function $M : \mathbb{N} \to \mathbb{N}$ of the dataset size.[5] For example, when $M(N_i) = \log N_i$, the sample size is logarithmic in the dataset size. This is a distinctive advantage over methods such as RP(M); these methods, however, are able to provide strict bounds.

We now show how to actually obtain the probabilistic upper bound on the sample size using subsampling. Fix some time i and denote by $N = N_i$ the size of the dataset and by $M = M(N_i)$ the upper bound at time i. Recall that the size of a Bernoulli sample has a binomial(N, q) distribution, where q is the sampling rate at time i. The probability that the sample size does not exceed M is given by (Johnson et al. 2005, p. 110)

$$\Pr[\,|S| \leq M\,] = \sum_{k=M+1}^{N} B(k;\, N, q) = I_q(M+1, N-M), \tag{5.17}$$

where $I_x(a, b)$ denotes the *regularized incomplete beta function* (also called incomplete beta function ratio). An efficient algorithm for the computation of the most significant digits of $I_x(a, b)$ is given by Didonato and Morris, Jr (1992). Now, to ensure that the sample exceeds bound M with probability at most δ, we equate (5.17) to $1 - \delta$ and solve for q using numerical methods; a closed-form approximation of the solution is given by Brown and Haas (2006). With q_0 being the obtained solution, we initiate subsampling with target sampling rate $q' \leq q_0$ when $q_0 < q$, that is, when the current

[5] Other choices for M are possible, e.g., M may be a function of time. In the following, we only require that M is oblivious to the state of the sample.

sampling rate is too high. To avoid overly frequent subsampling steps, the target sampling rate q' after subsampling should be set to a value slightly less than q_0.

The failure probability δ can be chosen small without too much overhead. For example, suppose that $|R| = 10{,}000{,}000$, $M = 10{,}000$ and $\delta = 0.01$. The value of q_0 is then given by 0.00095; the expected sample size of 9,500 items is close to M, and the actual sample size will not exceed M with probability 99%.

B. Algorithmic Description

The subsampling problem is trivial if we are only trying to produce an ordinary Bernoulli subsample that does not need to be incrementally maintained. With $q' < q$ being the desired sampling rate after subsampling, the subsample S' can be obtained by running BERN(q'/q) sampling on S; see for example Brown and Haas (2006). Counting duplicate items separately, each item is present in S with probability q and accepted into S' with probability q'/q; the probability that it occurs in S' thus equals $q \cdot q'/q = q$, as desired. The challenge in the general setting is to assign an appropriate value to the tracking counters of the subsample, so that incremental maintenance can be continued.

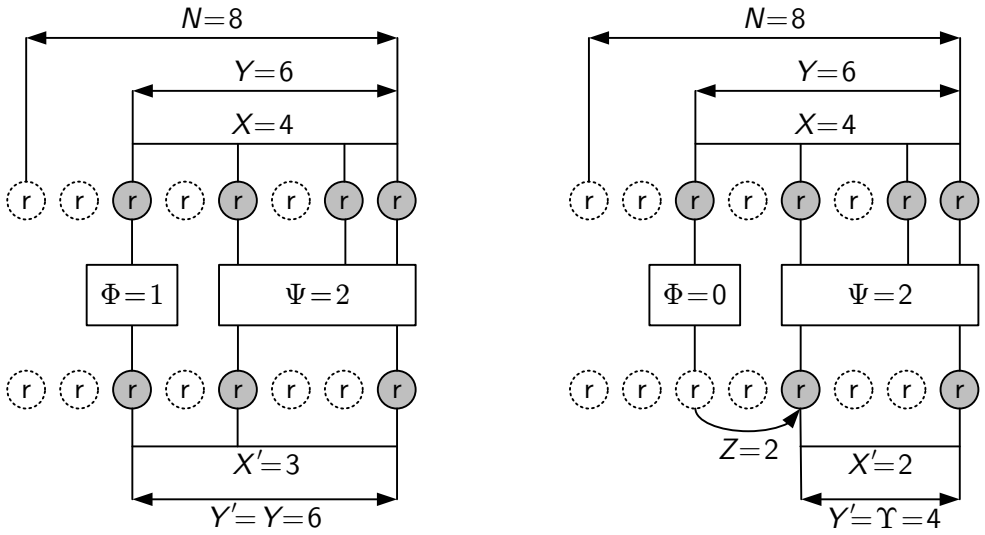
We now describe the subsampling algorithm for ABERN(q). Let S_i be the sample after processing transactions $\gamma_1, \ldots, \gamma_i$ with sampling rate q. Again, we suppress the subscript i and fix an item r, so that S_i is given by $S = (X, Y)$. Given S, we want to generate $S' = (X', Y')$ having the correct distribution. Our subsampling algorithm is illustrated in figure 5.3. In the figure, the original sample has been computed from a transaction sequence that consists of 8 insertions of item r. Sample items are shown as gray circles, rejected items are white. The upper and lower part of the figure display the state of the sample before and after subsampling, respectively.

The algorithm is as follows: Set $q^* = q'/q$. Let Φ be a Bernoulli(q^*) random variable, i.e., a 0/1 random variable with $\Pr[\,\Phi = 1\,] = q^*$ and $\Pr[\,\Phi = 0\,] = 1 - q^*$. Let Ψ be a random variable such that $\Pr[\,\Psi = k'\,] = B(k'; X - 1, q^*)$ for $0 \le k' < X$. Observe that $\Psi \equiv 0$ when $X = 1$. The random variable Φ has the interpretation that $\Phi = 1$ if and only if the first of the X items that were inserted into S is retained in S'; the random variable Ψ is the number of the remaining $X - 1$ items that are retained. The algorithm sets

$$X' \leftarrow \begin{cases} 0 & \text{if } X = 0 \\ \Phi + \Psi & \text{otherwise.} \end{cases} \tag{5.18}$$

To compute Y', let Υ be another random variable with

$$\Pr[\,\Upsilon = m'\,] = \frac{X'}{m'} \prod_{i=m'+1}^{Y-1} \left(1 - \frac{X'}{i}\right)$$



(a) First item retained ($\Phi = 1$) **(b)** First item rejected ($\Phi = 0$)

Figure 5.3: Illustration of subsampling

for $X' \le m' < Y$. (By convention, we take an empty product as equal to 1.) The algorithm sets

$$Y' \leftarrow \begin{cases} 0 & \text{if } X' = 0 \\ Y & \text{if } X' > 0 \text{ and } \Phi = 1 \\ \Upsilon & \text{otherwise.} \end{cases} \tag{5.19}$$

Some insight into this choice of Y' is given in the proof below.

C. Proof of Correctness

We now establish the correctness of the subsampling algorithm.

Theorem 5.6. *The above subsampling algorithm produces an ABERN(q') sample.*

Proof. We must show that the random pair (X', Y') has the proper distribution, given (X, Y), after determining what this proper distribution is. As with theorem 5.1, we give a proof that is somewhat informal, but provides insight into the workings of the algorithm. The variables used in this proof are illustrated in figure 5.3. Denote by $\gamma = (\gamma_1, \ldots, \gamma_N)$ a sequence of N insertions of item r. Since, by theorem 5.1, the distribution of X and Y depend only on N, we can assume without loss of generality that S has been generated from γ. (That is, S is based on N non-annihilated insertions.)

First consider the distribution of X'. Clearly, X' must equal 0 if X equals 0. Otherwise, as indicated above, the correct way to obtain X' from X is to take a BERN(q^*) subsample of the X items in S. Therefore, the random variable X' must

have a binomial(X, q^*) distribution, i.e., $\Pr[\, X' = k' \,] = B(k'; X, q^*)$. Recall that, in general, a binomial(m, q) random variable can be represented as the sum of m independent and identically distributed (i.i.d.) Bernoulli(q) random variables. Thus Ψ can be viewed as a sum of $X - 1$ i.i.d. Bernoulli(q) random variables, so that $\Phi + \Psi$ is distributed as a sum of X such variables, and hence has a binomial(X, q^*) distribution. Therefore X', as defined by (5.18), indeed has the proper distribution.

To complete the proof, it suffices to show that the conditional distribution of Y' is the proper one, given that we have taken a BERN(q^*) subsample that resulted in X' items being retained. To determine the proper distribution, first observe that, trivially, Y' must equal 0 if X' equals 0. To analyze the case where $X' > 0$, let $j^* = N - Y + 1$ be the index of the transaction corresponding to the first insertion of an item into S; thus the remaining $X - 1$ items in S were inserted during transactions $\gamma_{j^*+1}, \gamma_{j^*+2}, \ldots, \gamma_N$. Clearly, $Y' = Y$ if and only if the item inserted into S at transaction γ_{j^*} is retained in the subsample S'; this event occurs with probability q^*. With probability $1 - q^*$, we have $Y' \neq Y$. In this case, we can compute the proper distribution of Y' as follows. The transactions corresponding to the X' items in the final subsample represent a subset (of size X') of the $Y - 1$ transactions $\gamma_{j^*+1}, \gamma_{j^*+2}, \ldots, \gamma_N$. By symmetry, all possible subsets of size X' are equally likely. Thus, if we have an urn containing $Y - 1$ balls, of which X' are black (gray in figure 5.3b) and $Y - 1 - X'$ are white, and if Z is the random variable that represents the number of sequential draws (without replacement) required to produce the first black ball, then the first of the X' transactions has an index distributed as $j^* + Z$, so that Y' must be distributed as $N - (j^* + Z) + 1 = Y - Z$. For Z to equal $l > 1$, the first ball must be white, which happens with probability $1 - \big(X'/(Y-1)\big)$, then the next ball must be white, which happens with probability $1 - \big(X'/(Y-2)\big)$ (since there is one less white ball in the urn after the first draw), and so on, up through the $(l-1)$st ball; the l-th ball must be black, which happens with probability $X'/(Y - l)$. Similarly, $Z = 1$ with probability $X'/(Y - 1)$. Thus

$$\begin{aligned}\Pr[\, Z = l \,] &= \frac{X'}{Y - l} \prod_{i=1}^{l-1} \left(1 - \frac{X'}{Y - i}\right) \\ &= \frac{X'}{Y - l} \prod_{i=Y-l+1}^{Y-1} \left(1 - \frac{X'}{i}\right)\end{aligned}$$

for $l \geq 1$, where the second equality results after a change of index from i to $Y - i$ in the product. Since $Y' = Y - Z = m'$ if and only if $Z = Y - m'$, we must have

$$\begin{aligned}\Pr[\, Y' = m' \mid Y' \neq Y \,] &= \Pr[\, Z = Y - m' \,] \\ &= \frac{X'}{m'} \prod_{i=m'+1}^{Y-1} \left(1 - \frac{X'}{i}\right).\end{aligned}$$

By inspection, the random variable Y' defined via (5.19) has precisely the correct distribution. As mentioned previously, Φ has the interpretation that $\Phi = 1$ if and

only if $Y' = Y$, i.e., if and only if the item inserted into S at transaction γ_{j^*} is retained in S'. □

Algorithm 5.2 gives the pseudocode for our subsampling algorithm. We have displayed simple versions of the functions ComputeΨ and ComputeϒΥ, which generate samples of the random variables Ψ and Υ, respectively. More efficient algorithms are given in Devroye (1986, p. 521 and p. 619).

5.3 Sample Merging

We now discuss the merging problem. Given partitions R_1 and R_2 of R with $R_1 \uplus R_2 = R$, along with two independent ABERN(q) samples S_1 and S_2, the goal is to derive an ABERN(q) sample S from R by accessing S_1 and S_2 only.[6] As mentioned previously, merging is used in practice when R is distributed across several nodes; see Brown and Haas (2006) for an example.

If S is not subject to further maintenance, we can set $X(r) \leftarrow X_1(r) + X_2(r)$ for all $r \in S_1 \cup S_2$ to produce a plain BERN(q) sample; see Brown and Haas (2006). Here $X_1(r)$ and $X_2(r)$ denote the frequency of item r in the respective subsamples, and $X(r)$ denotes the frequency of item r in the merged sample. A harder version of the problem is to derive a merged sample S that includes a tracking counter, so that maintenance of S can be continued. First suppose that we know a priori that $R_1 \cap R_2 = \emptyset$. Then it is easy to show that, for all when $r \in S_i$, setting $X(r) \leftarrow X_i(r)$ and $Y(r) \leftarrow Y_i(r)$ yields the desired augmented Bernoulli sample. Otherwise, the hard merging problem cannot be solved, as shown by the following negative result.

Theorem 5.7. *If $R_1 \cap R_2 \neq \emptyset$ and $0 < q < 1$, then there exists no algorithm that can compute an ABERN(q) sample S of $R = R_1 \uplus R_2$ by accessing S_1 and S_2 only.*

Proof. The proof is by contradiction, so suppose that there exists a merging algorithm A and let S be the ABERN(q) sample produced by A. Fix an item r in the intersection of R_1 and R_2. There is always such an item, since $R_1 \cap R_2 \neq \emptyset$. Denote by N, N_1, and N_2 the frequency of item r in R, R_1, and R_2, respectively. Furthermore, let Y, Y_1, and Y_2 be the value of the tracking counters in S, S_1, and S_2, respectively.

We first show that A cannot ever set $Y > Y_1 + Y_2$. From (5.3), we must have $\Pr[Y_1 = N_1] = \Pr[Y_2 = N_2] = q$ so that, by the independence of Y_1 and Y_2,

$$\Pr[Y_1 = N_1, Y_2 = N_2] = q^2. \tag{5.20}$$

Suppose that Algorithm A were to set $Y > Y_1 + Y_2$ with positive probability when it observed, say, input values $Y_1 = k_1$ and $Y_2 = k_2$, that is,

$$\Pr[Y > k_1 + k_2 \mid Y_1 = k_1, Y_2 = k_2] > 0$$

[6] Here, $\uplus$ has multiset semantics, so that it need not be the case that $R_1 \cap R_2 = \emptyset$.

Algorithm 5.2 Subsampling for augmented Bernoulli sampling

```
 1: q: current sampling rate
 2: q': desired sampling rate, q' ≤ q
 3: S: the augmented Bernoulli sample
 4: RANDOM(): returns a uniform random number between 0 and 1
 5:
 6: for all r ∈ S do
 7:    (X, Y) ← S[r]
 8:    X ← COMPUTEΨ(X, q*)
 9:    if RANDOM() < q* then                          // first item included
10:       X ← X + 1
11:       S[r] = (X, Y)
12:    else if X = 0 then                             // all items excluded
13:       remove r from S
14:    else                                           // first item excluded
15:       Y ← COMPUTEϒ(X, Y)
16:       S[r] = (X, Y)
17:    end if
18: end for
19:
20: COMPUTEΨ(X, q*)                                   // simple version
21: Ψ ← 0
22: for 1 ≤ i ≤ X − 1 do
23:    if RAND() < q* then
24:       Ψ ← Ψ + 1
25:    end if
26: end for
27: return Ψ
28:
29: COMPUTEϒ(X, Y)                                    // simple version
30: ϒ ← Y − 1
31: while RAND() < X/ϒ do
32:    ϒ ← ϒ − 1
33: end while
34: return ϒ
```

If item r happened to occur exactly k_1 and k_2 times in R_1 and R_2, respectively, so that $N_1 = k_1$ and $N_2 = k_2$, then

$$\begin{aligned}
&\Pr[\, Y > N \,] \\
&\geq \Pr[\, Y > k_1 + k_2, Y_1 = k_1, Y_2 = k_2 \,] \\
&= \Pr[\, Y > k_1 + k_2 \mid Y_1 = k_1, Y_2 = k_2 \,] \Pr[\, Y_1 = k_1, Y_2 = k_2 \,] \\
&= \Pr[\, Y > k_1 + k_2 \mid Y_1 = k_1, Y_2 = k_2 \,] \, q^2 > 0,
\end{aligned}$$

where we have used (5.20). But $Y \in \{\, 0, 1, \ldots, N \,\}$ by definition, so that Algorithm A cannot ever set $Y > Y_1 + Y_2$.

We now show that A must set $Y > Y_1 + Y_2$ with positive probability, a contradiction. Observe that

$$\begin{aligned}
\Pr[\, Y = N \,] = \Pr\big[\, Y = N, (Y_1, Y_2) = (N_1, N_2) \,\big] \\
+ \Pr\big[\, Y = N, (Y_1, Y_2) \neq (N_1, N_2) \,\big].
\end{aligned} \tag{5.21}$$

By (5.3) and (5.20), we must have

$$\begin{aligned}
\Pr[\, Y = N \,] = q > q^2 = \Pr\big[\, (Y_1, Y_2) = (N_1, N_2) \,\big] \\
\geq \Pr\big[\, Y = N, (Y_1, Y_2) = (N_1, N_2) \,\big],
\end{aligned}$$

which implies that

$$\Pr\big[\, Y = N, (Y_1, Y_2) \neq (N_1, N_2) \,\big] > 0 \tag{5.22}$$

for the equality in (5.21) to hold. Since $N = N_1 + N_2$ and, by definition, $Y_1 \leq N_1$ and $Y_2 \leq N_2$, the assertion in (5.22) is equivalent to $\Pr[\, Y = N, Y_1 + Y_2 < N \,] > 0$, so that $\Pr[\, Y > Y_1 + Y_2 \,] > 0$. □

5.4 Summary

We have provided a scheme for maintaining a Bernoulli sample of an evolving multiset. Our maintenance algorithm can handle arbitrary insertion, update and deletion transactions, and avoids ever accessing the underlying dataset. We have shown that the tracking counters used for maintenance can also be exploited to estimate frequencies, population sums, population averages, and distinct-item counts in an unbiased manner, with variance lower (often much lower) than the standard estimates based on a Bernoulli sample. We have also indicated how to estimate the variance (and hence the standard error) for these estimates, and we have briefly described how to apply results from ratio-estimation theory to our new estimators. We have also described how to obtain an augmented sample from another such sample using subsampling, and we have identified the (rather limited) conditions under which augmented samples can be merged to obtain an augmented sample of the union of the underlying datasets.

Chapter 6

Distinct-Item Sampling

In this chapter,[1] we are concerned with bounded-size sampling schemes that maintain uniform distinct-item samples from an evolving dataset. We also show how to estimate scale-up factors, how to resize the sample, and how to combine multiple samples to compute a sample of any set or multiset expression involving their base datasets. For the latter reason, our algorithms are also valuable in the context of set sampling because typical set sampling methods cannot perform such combinations.

Recall that virtually all distinct items schemes are based on hashing—the only exception known to the authors is the AMIND(q) scheme proposed in the previous chapter. Before we turn our attention to distinct-item sampling from evolving datasets, we briefly discuss the suitability of the available hash functions for sampling. Many of these hash functions have been developed with applications such as hash tables in mind. In these applications, the main goal of hashing is to minimize collisions; it is not immediately clear how the hash functions interact with distinct-item sampling. In section 6.1, we therefore review the most common classes of hash functions. We also conduct a simple experiment that empirically evaluates the uniformity of the sample obtained by using each of these hash functions. Interestingly, our experiment suggests that the best compromise between practicability and uniformity of the sample is achieved by cryptographic hash functions.

The heart of this chapter starts in section 6.2, where we extend min-hash sampling with support for deletions. The resulting scheme is called augmented min-hash sampling, AMIND(M), because the sample is augmented with counters of the multiplicity of each distinct item. Our method is similar to the reservoir sampling with tagging scheme by Tao et al. (2007), that is, deleted items are occasionally retained in the sample in order to ensure uniformity. In our analysis, we show that only deletions of the last copy of a distinct item may have a (negative) impact on the sample size and that the sample stays reasonably large when the fraction of deletions is not too high. Also in section 6.2, we consider the problem of estimating the number of distinct items in the dataset for both plain MIND(M) and our AMIND(M) scheme. We present several estimators for this problem and analyze their theoretical properties.

[1] Parts of the material in this chapter have been developed jointly with Kevin Beyer, Peter J. Haas, Berthold Reinwald, and Yannis Sismanis. The chapter is based on Beyer et al. (2007) with copyright held by ACM. The original publication is available at http://portal.acm.org/citation.cfm?id=1247480.1247504.

It turns out that our estimators are unbiased and have low variance; they can even be used to estimate the number of distinct items that satisfy a given predicate.

In section 6.3, we discuss the problem of resizing the sample upwards and downwards. Unfortunately, it appears that every algorithm that resizes the sample upwards must scan almost the entire base data. We give a slightly more efficient algorithm than naive recomputation from scratch. Reducing the sample size is significantly easier and can be done without accessing base data.

One of the most interesting properties of AMIND(M) samples is that they can be combined to obtain samples of arbitrary unions, intersections and differences of their underlying datasets. In fact, we show in section 6.4 that AMIND(M) samples are closed under these operations. Therefore, AMIND(M) samples are much more powerful in this respect than the set and multiset sampling schemes discussed earlier. As might be expected, the size of the resulting sample depends on the selectivity of the expression used to combine the samples so that our techniques can only be used when the result of the expression is not too small.

6.1 Hash Functions

We consider hash functions that have domain $[N]$ and range $[H]$, where both N and H are fixed positive integers and $[k]$ denotes the set $\{0, \ldots, k-1\}$. Let $\mathscr{H} = \{h_1, h_2, \ldots\}$ be a family of hash functions from $[N]$ to $[H]$. In practice, the family $\mathscr{H}$ represents the set of functions that are generated by a specific hashing scheme. Whenever a hash function is required, it is chosen uniformly and at random from $\mathscr{H}$, typically by assigning random values to parameters of the hashing scheme. We are therefore interested in properties of a single hash function h chosen uniformly and at random from $\mathscr{H}$.

A. Truly Random Hash Functions

If $\mathscr{H}$ is the set of all functions from $[N]$ to $[H]$, then h is said to be *truly random*. This means that for arbitrary subsets $\{r_1, \ldots, r_n\} \subseteq [N]$ and arbitrary (not necessarily distinct) hash values $v_1, \ldots, v_n \in [H]$, it holds

$$\Pr[h(r_1) = v_1, h(r_2) = v_2, \ldots, h(r_n) = v_n] = \frac{1}{H^n}.$$

Random hash functions can be used with any of the hash-based sampling schemes of section 3.5.3, and the resulting samples are guaranteed to be truly uniform. The downside is that the space required to store a random hash function is too large for all practical purposes. Indeed, the number of hash functions in $\mathscr{H}$ is $|\mathscr{H}| = H^N$ so that $\Omega(N \log H)$ bits are required to store h. Thus, smaller families of hash functions are needed. These smaller families naturally lead to a smaller amount of "randomness" in h (or, more precisely, a smaller amount of entropy), but can still provide good results in practice.

B. Universal Hash Functions

We first discuss families for which the loss in randomness can be quantified. A family $\mathscr{H}$ is said to be 2-*universal* if for all $r_1, r_2 \in [N]$ with $r_1 \neq r_2$

$$\Pr[\, h(r_1) = h(r_2) \,] \leq \frac{1}{H}, \tag{6.1}$$

that is, a collision of two hash values is at most as likely as a collision of two hash values obtained from a truly random hash function. The concept has been introduced in a well-known paper by Carter and Wegman (1977), who also give some 2-universal hash functions. For example, for $H \leq N \leq p$ with p prime, the family

$$\mathscr{H}_{p,H} = \{\, h_{a,b} \mid a, b \in [p], a > 0 \,\} \quad \text{with} \quad h_{a,b}(x) = \big((ax + b) \bmod p\big) \bmod H$$

is 2-universal. To describe a hash function from $\mathscr{H}_{p,H}$, only $O(\log p)$ bits to store parameters a and b are required. According to Bertrand's postulate, there exists a prime between N and $2N$, so that we require $O(\log N)$ bits for a suitably chosen prime p. Although universal hash functions are well-suited for hashing into a hash table, their usefulness for random sampling appears limited. Indeed, equation (6.1) does not even guarantee that each hash value is chosen with equal probability for each input item. For example, when $H \geq N$, the family that consists of only the identity function is 2-universal. Since this family contains only a single function, h does not provide any randomness and thus cannot be used to drive the random sampling process.[2]

C. Pairwise Independent Hash Functions

A stronger concept than 2-universality is that of pairwise independence (also called strong 2-universality). A family $\mathscr{H}$ is *pairwise independent* if for all $r_1, r_2 \in [N]$ with $r_1 \neq r_2$ and for all $v_1, v_2 \in [H]$

$$\Pr[\, h(r_1) = v_1, h(r_2) = v_2 \,] = \frac{1}{H^2},$$

that is, h is indistinguishable from a truly random function for inputs of size *at most* 2. The 2-universal family $\mathscr{H}_{p,N}$ given above is "almost" pairwise independent; minor modifications lead to the pairwise independent family

$$\mathscr{H}_p = \{\, h_{a,b} \mid a, b \in [p] \,\} \quad \text{with} \quad h_{a,b}(x) = (ax + b) \bmod p$$

for $N \leq H = p$, with p prime. A summary of available pairwise independent hash functions is given in Thorup (2000). Again, for purposes such as hashing into hash tables, pairwise independent hash functions work well in practice. Mitzenmacher and Vadhan (2008) pointed out that this behavior results from the "randomness" of the data items presented to the hash functions. Laxly speaking, when the data contains

[2] There are many other deterministic hash functions; none of them is suitable for random sampling.

enough randomness, the output of the hash function will approximately look like a random function. However, there is no known distinct-item sampling scheme that is able to produce *provably* uniform samples based on pairwise independent hash functions; it is also unknown whether such a scheme exists.

The concept of pairwise independence can be generalized to k-wise independence in the obvious way: A family of hash functions is k-wise independent, when the hash values of every subset of $[N]$ with size at most k are indistinguishable from a truly random hash function. An efficient 4-wise independent hash function is given in Thorup and Zhang (2004). When the base data is known to contain at most k items and k-independent hash functions are used, all the distinct-item schemes of section 3.5.3 produce truly uniform samples. However, using arguments as above, such hash functions require $\Omega(k \log H)$ bits to describe; too much in practice.

D. Min-Wise Independent Hash Functions

A family of hash functions is *min-wise independent* if it is collision-free (thus $N \leq H$) and if for all $A \subseteq [N]$ and for all $r \in A$

$$\Pr\big[\, h(r) = \min_{r' \in A} h(r') \,\big] = \frac{1}{|A|},$$

that is, for every subset of the domain, each item is equally likely to have the minimum hash value. As discussed previously, min-hash sampling with replacement (page 92) in conjunction with a set of min-wise independent hash-functions produces truly uniform samples. Bounds on the size of min-wise independent families are given in Broder et al. (2000) and an explicit, optimal construction algorithm is given in Takei et al. (2000). The size of exact min-wise independent families is exponential in N so that a hash function requires $\Omega(N)$ space to describe, but Indyk (1999) and Broder et al. (2000) also discuss smaller approximate families.

From a theoretical point of view, the concept of min-wise independence can be generalized to that of k-min-wise independence, meaning that any sequence of k items is equally likely to have the k-smallest hash values. However, it is unknown whether non-trivial (i.e., not truly random) k-min-wise independent families exist. Also, from the arguments above, their space consumption is at least linear in N, so that, from a practical point of view, these functions are infeasible.

E. Cryptographic Hash Functions

We now turn attention to families of hash functions that perform extremely well in practice but do not provide theoretical guarantees. More specifically, we consider cryptographic hash functions that are based on block ciphers, see Menezes et al. (1996, ch. 9.5). A *block cipher* c with block length m is a function that maps an m-bit input, called *plaintext*, to an m-bit output, called *ciphertext*; it is parametrized by a K-bit key k. Assume for a moment that $N = H = 2^m$ and denote by $c_k(r)$ the ciphertext of $r \in [N]$ using key $k \in [2^K]$. Then

$$\mathscr{H}_c = \big\{\, c_k \mid k \in [2^K] \,\big\}$$

is the family of hash functions generated by c. To select a hash function from $\mathscr{H}_c$, the key k is picked uniformly and at random from the set of all keys. A hash function therefore requires $O(K)$ bits to describe.

Our choice for c is the Advanced Encryption Standard (AES), see NIST Federal Information Processing Standards (2001, FIPS 197). AES has a 128-bit block length and we make use of the version of AES that also has 128-bit keys. Hellekalek and Wegenkittl (2003) showed that AES behaves empirically like a high-quality pseudo-random number generator when applied in an iterative fashion. Their experimental results render AES therefore a promising candidate for sampling.

We now relax our initial assumption that both N and H equal 2^{128}. When $N < 2^{128}$, zero padding is used to inflate the size of the input. When $N > 2^{128}$, the input is partitioned into blocks of 128 bits length and a technique called *cipher-block-chaining* is used to compute the final hash value; see Menezes et al. (1996, p. 353) for a discussion of the algorithm. Regarding the hash range, H is often set to 2^{32} in practice, in which case we simply take the last 32 bits of the ciphertext as the hash value.

F. A Simple Experiment

We conducted a simple experiment that provides some insight into the suitability of the different hash functions for random sampling. Our goal is to find out which hash functions provide uniform samples with min-hash sampling and which do not. The results in this section are of an empirical nature, that is, we can determine whether a hash function is "bad" but positive results may not generalize. The experiment is as follows:

1. Fix a dataset R of size N, a sample size $M < N$ and select a family $\mathscr{H}$ of hash functions to be tested.

2. Generate $5\binom{N}{M}$ independent MIND(M) samples of R. Each sample is generated with a fresh hash function selected uniformly and at random from all the hash functions in $\mathscr{H}$.

3. Count the number of occurrences of each of the $\binom{N}{M}$ possible samples. In expectation, each sample should occur 5 times.

4. Run a chi-square test to check whether the observed frequencies are consistent with their expected values.

The results of the test are given in table 6.1. The numbers in parentheses correspond to the p-value of the chi-square distribution that corresponds to the outcome of the sampling process (see below).

We now describe the experimental setup in more detail. We used three different types of datasets: (i) increasing sequences of integers, (ii) real data, and (iii) random numbers. For a given choice of N, the sequence dataset consists of the integers $\{0, 1, \ldots, N-1\}$, the real dataset has been obtained from the "quakes" dataset

Table 6.1: Results of uniformity tests of MIND(M) sampling

Dataset	N	M	Random	Pairwise	Min-wise	AES
Sequence	1,000	1	✓(0.10)	- (10^{-5})	✓(0.34)	✓(0.42)
	100	2	✓(0.77)	- (0)	✓(0.96)	✓(0.34)
	25	5	✓(0.77)	- (0)	✓(0.61)	✓(0.19)
Real	1,000	1	✓(0.49)	✓(0.43)	✓(0.92)	✓(0.30)
	100	2	✓(0.74)	✓(0.73)	✓(0.24)	✓(0.84)
	25	5	✓(0.86)	- (0.01)	✓(0.52)	✓(0.12)
Random	1,000	1	✓(0.63)	✓(0.26)	✓(0.45)	✓(0.45)
	100	2	✓(0.11)	✓(0.38)	✓(0.74)	✓(0.79)
	25	5	✓(0.26)	✓(0.14)	✓(0.52)	✓(0.90)

shipped with The R Project (2008),[3] and the random dataset consists of random integers in the interval $[0, 2^{31} - 2]$. The datasets cover the spectrum from low over moderate to high entropy and allow us to study whether the distribution of the input data has an effect on the uniformity of the samples. The samples were generated using a family of truly random hash functions, a family of pairwise independent hash functions ($\mathscr{H}_{2^{31}-1}$), a family of min-wise independent hash functions (Broder et al. 2000) and a family of cryptographic hash functions based on AES. The chi-square test was run at the 5% significance level, meaning that the outcome of the sampling process is considered non-uniform if the probability of receiving from a truly uniform scheme a result at least as extreme as the one observed is below 5%. This probability is called *p-value* and is given in table 6.1. Note that direct comparison of the p-values is not meaningful, it only matters whether the p-value is less than 5% or not. To avoid rejecting a hash function by mere coincidence, we repeated failed tests up to 3 times.

We now discuss the results shown in table 6.1. First, random hash functions lead to provably uniform samples so that they naturally passed all the uniformity tests. The same holds for min-wise independent hash functions when $M = 1$. Interestingly, the min-wise independent hash function used in the experiments passed all the test and therefore seems to be a good candidate for sampling. However, as discussed previously, both random and min-wise independent hash functions require space linear in N to describe. Pairwise independent hash functions do not produce uniform samples on the sequence dataset, they pass some tests for the real data and all tests for random data. It seems that pairwise independent hash functions can be used when the dataset has sufficient entropy, see Mitzenmacher and Vadhan (2008). The samples produced by AES passed all our tests. Since both the pairwise independent hash function and the AES hash function require constant space, AES seems to be

[3] The dataset consists of information about 1,000 earthquakes near Fiji. We combined the longitude and latitude into a single integer ($\text{long} \cdot 1{,}000{,}000 + \text{lat} \cdot 100$) and then used the first N of these integers as the population.

the method of choice for practical purposes. It provides a good tradeoff between uniformity of the samples and the space required to store the hash function.

6.2 Uniform Sampling

In the remainder of this chapter, we focus on the min-hash sampling scheme MIND(M) discussed in section 3.5.3D. Recall that in MIND(M) sampling, a random hash function h is used to assign a hash value to each arriving item. The sample then consists of the distinct items in R with the M smallest hash values. The scheme supports only insertion transactions, but in section 6.2.1, we leverage an idea from Tao et al. (2007) to provide support for update and deletion transactions. As mentioned previously, the main obstacle of MIND(M) sampling is that it is not immediately clear how to estimate the number D of distinct items in the dataset. We discuss this problem in section 6.2.2, where we provide an unbiased, low-variance estimator of the distinct-item count. Results of an experimental study are given in section 6.2.3.

We subsequently assume that—in an empirical sense—the hash function being used with MIND(M) sampling behaves like a truly uniform hash function. In other words, we assume that when H is the domain of the hash functions, the sequence of hash values $h(r_1), h(r_2), \ldots, h(r_D)$ looks like a realization of independent and identically distributed samples from the discrete uniform distribution on $[H]$. Provided that H is sufficiently greater than D, there will be no collision between the hash values with high probability. In fact, a "birthday problem" argument shows that collisions will be avoided when $H = \Omega(D^2)$, see Motwani and Raghavan (1995, p. 45). We assume henceforth that, for all practical purposes, any hash function that arises in our discussion is collision-free.

6.2.1 Min-Hash Sampling With Deletions

We now propose a simple extension of MIND(M) that adds support for deletions. We refer to the scheme as augmented min-hash sampling, AMIND(M). We do not consider updates explicitly, because they can be decomposed into a deletion and an insertion transaction. Also, we omit the transaction number (subscript i) in our discussion for simplicity.

A. Algorithmic Description

The key idea of AMIND(M) is to maintain a uniform sample of *all* distinct items that have been inserted into the dataset, even the ones that have been deleted in the meantime. The non-deleted sample items then constitute a uniform sample of $D(R)$, the set of distinct items in R. To distinguish between deleted and non-deleted items, we associate a frequency counter $N(r)$ with each sampled item—just as we did in MBERND(q) sampling, section 3.5.3B. Each element of the sample is therefore a tuple $(r, N(r))$, where $N(r)$ denotes the frequency of item r in R. In contrast to MBERND(q), however, item r is also kept in the sample when its frequency count

reduces to zero. Items with a frequency count of at least 1 represent non-deleted items, while items with a frequency count of 0 represent deleted items.

In more detail, the AMIND(M) sample consists of the distinct items with the M smallest hash values seen thus far. The algorithm is as straightforward: Suppose that we are about to process insertion $+r$. If $r \in S$, we increase its frequency count. Otherwise, when $r \notin S$, we proceed as in MIND(M) sampling: We add item r directly to sample when the sample has not yet been completely filled ($|S_i| < M$) or replace the sample item r_{max} with the largest hash value when $h(r) < h(r_{max})$. In both cases, the frequency count of item r is initialized to 1. Now suppose that deletion $-r$ arrives. If $r \in S$, we decrease its frequency count but keep the item in the sample, even when its count became zero. If $r \notin S$, we ignore the deletion.

Algorithm 6.1 gives the pseudocode of the sampling scheme. In order to support efficient maintenance, an actual implementation would make use of a "randomized" treap to store the items in S; see Seidel and Aragon (1996) for description and analysis of the data structure. In a nutshell, a treap is a mixture of a tree and a heap. As in a tree, the items are sorted with respect to their value so that a specific item can be found efficiently. As in a heap, the items are sorted in heap order with respect to their hash value. Our treap appears randomized because the hash values are chosen at random. For a randomized treap of size M, both lookups, insertions and deletions have expected cost $O(\log M)$. The element with the largest hash value resides at the root of the treap and can thus be found in $O(1)$ time. A discussion of the total cost of algorithm 6.1 can be found in section C.

We refer to the sample maintained by AMIND(M) as *gross sample* because the sample may contain deleted items, which cannot be exploited for estimation purposes. Denote as before by R^+ the set of all items that have ever been inserted into R (including duplicates). We have $D(R) \subseteq D(R^+)$ and $S \subseteq D(R^+)$. Since deletions never modify the items that are sampled (they modify only frequencies), the gross sample is a uniform random sample from $D(R^+)$. In fact, the sample contains the items from $D(R^+)$ with the M smallest hash values and—ignoring frequency counts—is identical to a MIND(M) sample from R^+. To obtain the desired sample from $D(R)$, we compute the *net sample*

$$S^* = \{\, (r, N(r)) \in S \mid N(r) > 0 \,\},$$

which contains only the non-deleted items. S^* is a uniform random sample of $D(R)$ by the uniformity of S and the fact that only the items from S that also belong to R are retained in S^*.

B. Sample Size Properties

The AMIND(M) scheme can be seen as the distinct-item version of the modified reservoir sampling scheme with tagging (MRST) by Tao et al. (2007), see the discussion in section 3.5.1F. When AMIND(M) is executed on a dataset that does not contain duplicates, it has the same sample size properties as MRST($0, M$): deletions cause the sample size to shrink. However, when the dataset contains

Algorithm 6.1 Augmented min-hash sampling

```
1: M: upper bound on the sample size
2: R, S: dataset and sample, respectively
3: S[r]: frequency counter of item r ∈ S (can be 0)
4: h: hash function from domain ℛ to [H]
5: MaxHash(S): returns the item in S that has the maximum hash value
6:
7: Insert(r):
8: if r ∈ S then                                   // repeated insertion
9:    S[r] ← S[r] + 1
10: else
11:    if |S| < M then                              // build phase
12:       add r to S and set S[r] = 1
13:    else                                         // sampling phase
14:       r_max ← MaxHash(S)
15:       if h(r) < h(r_max) then
16:          remove r_max from S
17:          add r to S and set S[r] = 1
18:       end if
19:    end if
20: end if
21:
22: Delete(r):
23: if r ∈ S then
24:    S[r] ← S[r] − 1
25: end if
```

duplicates, the effect of deletions on the size of an AMIND(M) sample may be negligible. The reason is that only deletions of the last copy of an item have a potential impact on the net sample size because only these deletions reduce the frequency of the item to 0. All other deletions do not affect the size of the net sample. For example, set $M = 1$ and consider the transaction sequence $+r_1+r_1+r_2$. We then have

$$\Pr[\, S_3 = \{\, (r_1, 2)\,\}\,] = \Pr[\, S_3 = \{\, (r_2, 1)\,\}\,] = \frac{1}{2}.$$

Now suppose that deletion $-r_1$ arrives in the stream of transactions. Then,

$$\Pr[\, S_4 = \{\, (r_1, 1)\,\}\,] = \Pr[\, S_4 = \{\, (r_2, 1)\,\}\,] = \frac{1}{2}$$

so that $|S_4^*| = |S_4| = 1$ with probability 1. Thus, in contrast to MRST(0, M), the net sample size does not directly depend on the total number of insertions and deletions. Instead, it depends on the number $D^+ = |D(R^+)|$ of insertions of new items—that is, items that have not been inserted before—and the number of deletions of last copies of an item. Continuing the example above, deletion $-r_2$ leads to

$$\Pr[\, S_5 = \{\, (r_1, 1)\,\}\,] = \Pr[\, S_5 = \{\, (r_2, 0)\,\}\,] = \frac{1}{2}$$

so that $|S_5^*| = 1$ with probability 0.5 and $|S_5^*| = 0$ otherwise. Using arguments as in section 4.1.1, one finds that the sample size follows the hypergeometric distribution

$$\Pr[\, |S^*| = k\,] = H(k; D^+, D, M), \tag{6.2}$$

where $D = |D(R)|$ and $H(k; N, N', M)$ is defined as in equation (4.10). The sample size has expected value

$$\mathrm{E}[\, |S^*|\,] = \frac{D}{D^+} M$$

and variance

$$\mathrm{Var}[\, |S^*|\,] = \frac{D(D^+ - D)M(D^+ - M)}{(D^+)^2(D^+ - 1)}.$$

C. Runtime Cost

We now analyze the CPU cost of maintaining an AMIND(M) sample. As mentioned before, we store the sample S in a randomized treap to facilitate efficient maintenance. In the proof of the theorem below, we assume that $D^+ > M$ (so that the sample does not coincide with the dataset) and that the computation of a hash value takes $O(1)$ time.

Theorem 6.1. *AMIND(M) requires $O((M/D^+)\log M)$ expected time to process an insertion or deletion transaction.*

Proof. We first consider a deletion $-r$. Making use of the fact that $r \in S$ if and only if $h(r) \leq h(r_{max})$, the membership check in line 23 of algorithm 6.1 can be implemented in $O(1)$ time. The extraction of item r_{max} takes $O(1)$ time because r_{max} resides at the root of the treap. When $r \in S$, its frequency counter is updated. This requires $O(\log M)$ expected time: we have to locate item r in the treap of M items. Since $r \in S$ with probability M/D^+, the total expected cost is

$$\Pr[\, r \in S \,]\, O(\log M) + \Pr[\, r \notin S \,]\, O(1) = O\big((M/D^+) \log M\big)$$

as claimed in the theorem.

Basically the same arguments can be applied to the insertion case. The sample updates in lines 9, 12, 16, and 17 each take $O(\log M)$ expected time. Observe that the sample is updated if and only if $h(r) \leq h(r_{max})$. If r has been inserted at an earlier point of time, then this event occurs with probability $\Pr[\, r \in S \,] = M/D^+$. Otherwise, when r is new, the probability of accepting r and thereby accessing the sample is $M/(D^+ + 1) < M/D^+$ by the uniformity of AMIND(M) sampling. The assertion of the theorem now follows because the expected cost is $O(\log M)$ with probability at most M/D^+ and $O(1)$ otherwise. □

When $D^+ \leq M$, the expected cost per transaction is $O(\log D^+)$.

6.2.2 Estimation of Distinct-Item Counts

For some sampling applications, it is crucial to be able to estimate the number D of distinct items in the dataset. The value of D is used as a scale-up factor to interpolate estimates from the sample to the entire dataset. For example, suppose that the sample is taken over the distinct customers of a trading company and that the exact total of the customer's transactions is associated with each sample item. In section 3.5.3B, we used such a sample to estimate the *fraction* f of customers that contributed a total of less than a given quantity. If we instead want to determine the *number* of these customers, we have to multiply f by D, the total number of customers. Unfortunately, we cannot maintain D incrementally without storing the entire set of distinct items, which is clearly infeasible.[4] Therefore, we need more efficient estimation schemes.

There are two principal approaches to compute an estimate of D:

- *External estimation.* Maintain in addition to the sample a special-purpose synopsis solely for estimating D. A very good survey of the available synopses and their underlying ideas is given by Gibbons (2009). All the techniques are based on hashing, but—in contrast to distinct-item sampling—there are schemes that can provide probabilistic guarantees on the estimation error even when the hash functions being used are only pairwise independent (Alon et al. 1999; Bar-Yossef et al. 2002).

[4] This differs from set and multiset sampling, where the scale up factor equals the dataset size N, including duplicates, and can be maintained efficiently.

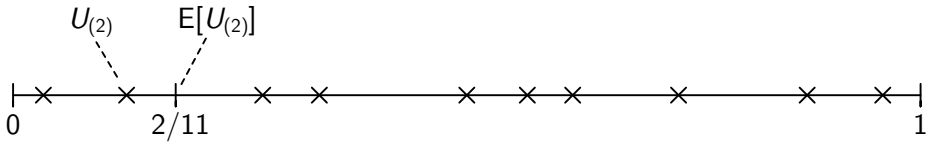


Figure 6.1: 10 random points on the unit interval

- *Internal estimation.* Estimate the number of distinct values directly from the information stored in the sample. As discussed below, the number of distinct values can be estimated by looking at the hash values of the sampled items and, in the case of AMIND(M), at the frequency counters associated with each item.

External estimation has the advantage that D can be estimated up to an arbitrary precision by throwing in sufficient resources. It has the disadvantage, however, that the additional synopsis requires itself space and time to maintain. In contrast, internal estimation does not incur any maintenance overhead, but its suitability depends on how accurate we can estimate the distinct-item count from the sample. As shown below, we can indeed provide highly accurate estimates provided that M is not too small. For this reason, we focus solely on internal estimation.

We subsequently assume that $D \geq M$.[5] For the insertion-only case, we review the basic estimator of Bar-Yossef et al. (2002), which is biased, and then show that a slightly modified version of the estimator yields an unbiased estimator with lower mean squared error. Using results from Cohen (1997) and Beyer et al. (2007), we find that the relative error of the unbiased estimator is bounded from above (as $D \to \infty$). We also show that, if $D \gg M$, the basic estimator closely resembles the maximum likelihood estimator of D. Finally, we extend our analysis to more general settings. Specifically, we extend the estimator to deal with transaction sequences that contain deletions, and we consider the problem of estimating the number of distinct values in subsets of the dataset.

A. The Basic Estimator

Interestingly, some of the synopses for distinct-item estimation are closely related to the MIND(M) sampling scheme. Instead of directly maintaining the M smallest items, these synopses maintain just the hash values of these items (Bar-Yossef et al. 2002; Beyer et al. 2007). The motivation behind these techniques can be viewed as follows. If $D \gg 1$ points are placed randomly and uniformly on the unit interval, then, by symmetry, the expected distance between any two neighboring points is $1/(D+1) \approx 1/D$, so that the expected value of $U_{(M)}$, the M-th smallest point, is

[5] Otherwise, the problem is trivial because the net sample size equals the number of distinct items in the dataset.

$\mathrm{E}[U_{(M)}] = \sum_{j=1}^{M} 1/(D+1) \approx M/D$; see figure 6.1. Thus $D \approx M/\mathrm{E}[U_{(M)}]$. The simplest estimator of $\mathrm{E}[U_{(M)}]$ is simply $U_{(M)}$ itself,[6] and yields the *basic estimator*

$$\hat{D}_M^{\mathrm{BE}} = \frac{M}{U_{(M)}}.$$

The connection between the above idea and the distinct-count estimation problem rests on the observation that—under our assumptions—the hash function "looks like" a uniform random number generator. In particular, let $r_1, r_2, \ldots, r_D$ be an enumeration of the distinct values in dataset R and let h be a hash function as before. The sequence $h(r_1), h(r_2), \ldots, h(r_D)$ will look like the realization of a sequence of independent and identically distributed (i.i.d.) samples from the discrete uniform distribution on $[H]$. Since we have set $H = \Omega(D^2)$, hash range H it is sufficiently greater than D so that the sequence

$$U_1 = \frac{h(r_1)}{H}, U_2 = \frac{h(r_2)}{H}, \ldots, U_D = \frac{h(r_D)}{H}$$

will approximate the realization of a sequence of i.i.d. samples from the continuous uniform distribution on $[0, 1]$. Thus, in practice, the estimator $\hat{D}_k^{\mathrm{BE}}$ can be applied with $U_{(M)}$ taken as the M-th smallest hash value (normalized by a factor of $1/H$). Note that the function $f(x) = 1/x$ is strictly convex on $(0, \infty)$, so that

$$\mathrm{E}[\hat{D}_M^{\mathrm{BE}}] = \mathrm{E}\left[\frac{M}{U_{(M)}}\right] > \frac{M}{\mathrm{E}[U_{(M)}]} \approx D$$

by Jensen's inequality. That is, the estimator $\hat{D}_M^{\mathrm{BE}}$ is biased upwards for each possible value of D. It was proposed by Bar-Yossef et al. (2002), along with conservative error bounds based on Chebyshev's inequality.

B. An Unbiased Estimator

In what follows, we provide an unbiased estimator that also has lower mean squared error than $\hat{D}_M^{\mathrm{BE}}$. The unbiased estimator is given by

$$\hat{D}_M^{\mathrm{UB}} = \frac{M-1}{U_{(M)}}, \tag{6.3}$$

a slight modification of $\hat{D}_M^{\mathrm{BE}}$. In this section, we analyze properties such as the unbiasedness and moments of $\hat{D}_M^{\mathrm{UB}}$.

Let $U_1, U_2, \ldots, U_D$ be the normalized hash values of the distinct items in the dataset; we model these values as a sequence of independent and identically distributed (i.i.d.) random variables from the uniform$[0, 1]$ distribution—see the discussion above. Denote by $U_{(M)}$ the M-th smallest of $U_1, U_2, \ldots, U_D$. We already know that

[6] In the statistical literature, this estimator is called the *method-of-moments* estimator of $\mathrm{E}[U_{(M)}]$.

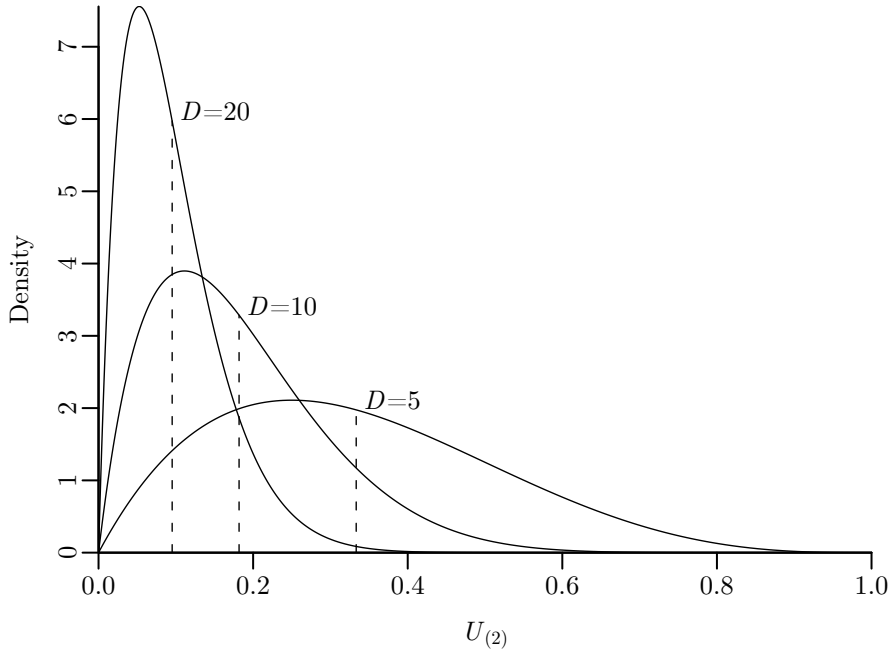


Figure 6.2: Distribution and expected value of $U_{(2)}$, the second-smallest hash value

$\mathrm{E}[U_{(M)}] = M/(D+1)$. Unfortunately, this does not help in analyzing $\hat{D}_M^{\mathrm{UB}}$ because $U_{(M)}$ occurs in the *denominator* of (6.3). Instead, our goal is to compute $\mathrm{E}[1/U_{(M)}]$, which requires knowledge of the distribution of $U_{(M)}$.

To make further progress, observe that $U_{(M)}$ is the M-th *order statistic* of $U_1, \ldots, U_D$. Results from the theory of order statistics (David and Nagaraja 2003, sec. 2.1) imply that $U_{(M)}$ follows the beta distribution with parameters M and $D - M + 1$. That is, the probability density function (pdf) of $U_{(M)}$ is given by

$$f_{M,D}(t) = \frac{t^{M-1}(1-t)^{D-M}}{B(M, D-M+1)}, \tag{6.4}$$

where

$$B(a,b) = \int_0^1 t^{a-1}(1-t)^{b-1}\,dt \tag{6.5}$$

$$= a^{-1}\binom{a+b-1}{a}^{-1} \tag{6.6}$$

denotes the *beta function* (Johnson et al. 1992, p. 7); equality (6.6) holds when a and b are integers. Figure 6.2 provides some insight into the distribution of the $U_{(M)}$ for $M = 2$ and different choices of D.

The probability that $U_{(M)}$ lies in a given interval is equal to the area under the respective part of the density function. Thus,

$$\Pr[\, U_{(M)} \leq x \,] = \int_0^x f_{M,D}(t)dt = I_x(M, D - M + 1) \tag{6.7}$$

where

$$\begin{aligned} I_x(a, b) &= \frac{1}{B(a, b)} \int_0^x t^{a-1}(1 - t)^{b-1} dt \\ &= \sum_{i=a}^{a+b-1} \binom{a + b - 1}{i} x^i (1 - x)^{a+b-i-1}. \end{aligned} \tag{6.8}$$

denotes the *regularized incomplete beta function*. It is defined for all real a and b (Johnson et al. 1992, p. 15) but equation (6.8) applies only when a and b are integer. As mentioned previously, an efficient algorithm for the computation of the most significant digits of $I_x(a, b)$ is given in Didonato and Morris, Jr (1992). Applying identity (6.8) to (6.7), we find that

$$\Pr[\, U_{(M)} \leq x \,] = \sum_{i=M}^{D} \binom{D}{i} x^i (1 - x)^{D-i},$$

which is simply the probability that at least M of the U_i values are smaller than x.

To facilitate the analysis of $\hat{D}_M^{\mathrm{UB}}$, we first derive the moments of $1/U_{(M)}$. For any real value r in $[0, M - 1)$, we have by definition of expected value, (6.4) and (6.5)

$$\mathrm{E}\,[\, U_{(M)}^{-r} \,] = \int_0^1 \frac{1}{t^r} f_{M,D}(t)dt = \frac{B(M - r, D - M + 1)}{B(M, D - M + 1)}.$$

If r is an integer, we can exploit identity (6.8) to obtain

$$\mathrm{E}\,[\, U_{(M)}^{-r} \,] = \frac{D^{\underline{r}}}{(M - 1)^{\underline{r}}}, \tag{6.9}$$

where $a^{\underline{b}}$ denotes the falling power $a(a - 1) \cdots (a - b + 1)$. Regarding $\hat{D}_M^{\mathrm{UB}}$, we find that

$$\mathrm{E}\,[\, \hat{D}_M^{\mathrm{UB}} \,] = \mathrm{E}\left[\, \frac{M - 1}{U_{(M)}} \,\right] = (M - 1)\,\mathrm{E}\,[\, U_{(M)}^{-1} \,] = D, \tag{6.10}$$

so that $\hat{D}_k^{\mathrm{UB}}$ is indeed unbiased for D. Using the definition of variance and (6.9), we obtain

$$\mathrm{Var}[\, \hat{D}_M^{\mathrm{UB}} \,] = (M - 1)^2\,\mathrm{E}\,[\, U_{(M)}^{-2} \,] - (M - 1)^2\,\mathrm{E}\,[\, U_{(M)}^{-1} \,]^2 = \frac{D(D - M + 1)}{M - 2}. \tag{6.11}$$

Since $\hat{D}_M^{\mathrm{UB}}$ is unbiased, its mean squared error (MSE) is equal to its variance.

For comparison, note that, since $\hat{D}_M^{\mathrm{BE}} = M\hat{D}_M^{\mathrm{UB}}/(M-1)$ and by (6.9), $\mathrm{E}[\,\hat{D}_M^{\mathrm{BE}}\,] = MD/(M-1)$ and

$$\mathrm{MSE}[\,\hat{D}_M^{\mathrm{BE}}\,] = \Big(\frac{M}{M-1}\Big)^2 \mathrm{Var}[\,\hat{D}_M^{\mathrm{UB}}\,] + \Big(\frac{D}{M-1}\Big)^2.$$

Thus, as discussed earlier, $\hat{D}_M^{\mathrm{BE}}$ is biased high for D, and has infinite mean when $M = 1$. Moreover, it can be seen that $\hat{D}_M^{\mathrm{UB}}$ has lower MSE than $\hat{D}_M^{\mathrm{BE}}$.

We now provide probabilistic (relative) error bounds for the estimator $\hat{D}_M^{\mathrm{UB}}$. Specifically, given a failure probability $0 < \delta < 1$, we give a value of ϵ such that $\hat{D}_M^{\mathrm{UB}}$ lies in the interval $[(1-\epsilon)D, (1+\epsilon)D]$ with confidence probability $1-\delta$.

Theorem 6.2. *For $0 < \epsilon < 1$ and $M \geq 1$,*

$$\Pr\left[\frac{|\hat{D}_M^{UB} - D|}{D} \leq \epsilon\right] = I_{u(D,M,\epsilon)}(M, D-M+1) - I_{l(D,M,\epsilon)}(M, D-M+1) \quad (6.12)$$

where

$$u(D,M,\epsilon) = \frac{M-1}{(1-\epsilon)D} \quad \textit{and} \quad l(D,M,\epsilon) = \frac{M-1}{(1+\epsilon)D}. \quad (6.13)$$

Proof. The desired result follows directly from (6.7) after using (6.3) to obtain

$$\begin{aligned}\Pr\left[\frac{|\hat{D}_M^{\mathrm{UB}} - D|}{D} \leq \epsilon\right] &= \Pr\big[\,(1-\epsilon)D \leq \hat{D}_M^{\mathrm{UB}} \leq (1+\epsilon)D\,\big] \\ &= \Pr\left[\frac{M-1}{(1+\epsilon)D} \leq U_{(M)} \leq \frac{M-1}{(1-\epsilon)D}\right].\end{aligned}$$

□

Error bounds for a given value of δ can be obtained by equating the right side of (6.12) to $1-\delta$ and solving for ϵ using numerical root-finding algorithm. Although useful for theoretical analysis, these bounds cannot be used directly in practice, since they involve the unknown parameter D. Using a standard approach from statistics, practical approximate error bounds based on the observed value of $U_{(M)}$ can be obtained by replacing D with $\hat{D}_M^{\mathrm{UB}}$ in the above formulas.

Figure 6.3 displays the error bound ϵ as a function of $1-\delta$ for $D = 1{,}000{,}000$ and several values of M. The dashed and solid curves represent the confidence intervals for the basic estimator $\hat{D}_M^{\mathrm{BE}}$ and the unbiased estimator $\hat{D}_M^{\mathrm{UB}}$, respectively. As expected, $\hat{D}_M^{\mathrm{UB}}$ is superior to $\hat{D}_M^{\mathrm{BE}}$ when M is small; for example, when $M = 16$ and $1-\delta = 0.95$, use of the unbiased estimator yields close to a 20% reduction in ϵ. As M increases, $M - 1 \approx M$ and both estimators perform similarly.

To further examine the behavior of the unbiased estimator, we derive the average relative error (ARE), which is defined as the expected value of the relative error $|\hat{D}_M^{\mathrm{UB}} - D|/D$. As discussed in section 2.1.3A, the ARE is a common metric for comparing the performance of statistical estimators.

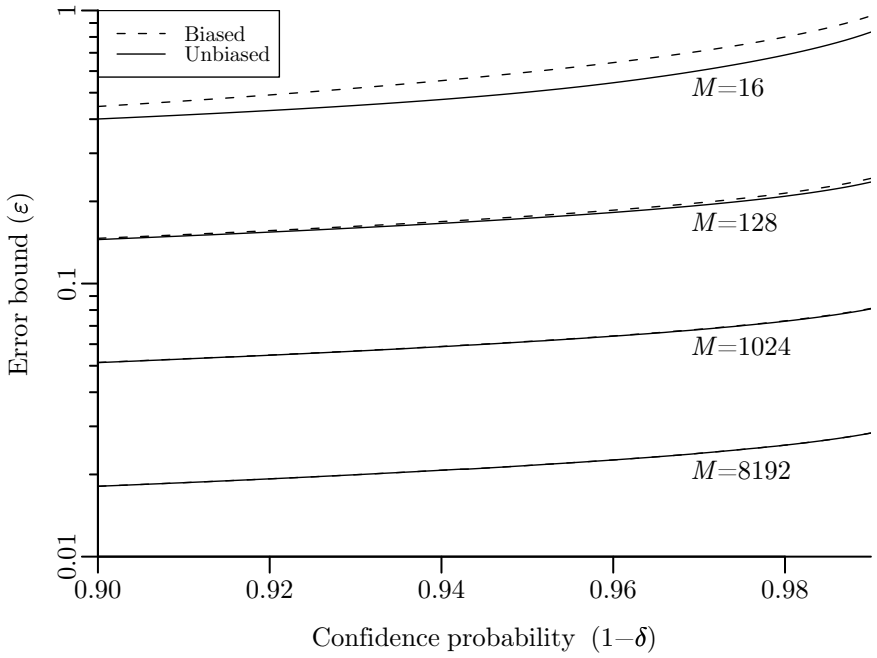


Figure 6.3: Error bounds for $D = 1{,}000{,}000$

Theorem 6.3. *The average relative error of $\hat{D}_M^{UB}$ is given by*

$$\mathrm{ARE}[\,\hat{D}_M^{UB}\,] = 2\binom{D}{M-1}\left(\frac{M-1}{D}\right)^{M-1}\left(1-\frac{M-1}{D}\right)^{D-M+2} \tag{6.14}$$

Proof. We have

$$\begin{aligned}
\mathrm{ARE}[\,\hat{D}_M^{\mathrm{UB}}\,] &= \mathrm{E}\left[\frac{|\hat{D}_M^{\mathrm{UB}}-D|}{D}\right] = \frac{1}{D}\int_0^1 \left|\frac{M-1}{t}-D\right| f_{M,D}(t)\,dt \\
&= \frac{1}{D}\int_0^{(M-1)/D}\Big(\frac{M-1}{t}-D\Big) f_{M,D}(t)\,dt \\
&\qquad + \frac{1}{D}\int_{(M-1)/D}^1 \Big(D-\frac{M-1}{t}\Big) f_{M,D}(t)\,dt \\
&= 2I_{(M-1)/D}(M-1, D-M+1) - 2I_{(M-1)/D}(M, D-M+1),
\end{aligned}$$

where the last equality is obtained after expanding the integrals and applying the identity $(M-1)(tD)^{-1} f_{M,D}(t) = f_{M-1,D-1}(t)$. The desired result follows after applying (6.8). □

Figure 6.4 displays the ARE for several choices of M and D. As can be seen, the ARE converges to an upper bound as D becomes large. For this reason, we study the behavior of $\hat{D}_M^{\mathrm{UB}}$ as D becomes large.

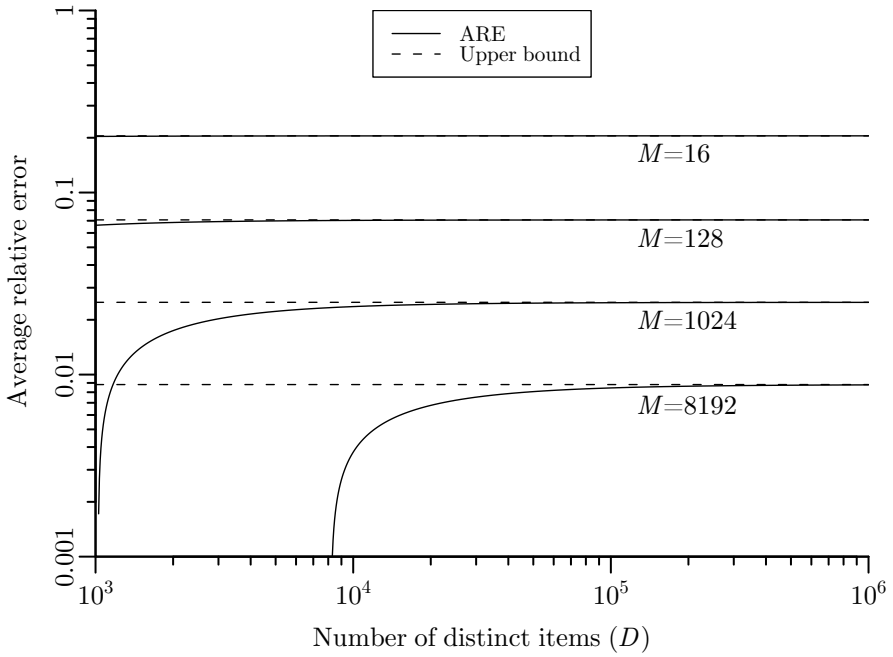


Figure 6.4: Average relative error of the unbiased estimator $\hat{D}_M^{\mathrm{UB}}$

C. Asymptotic Analysis of the Unbiased Estimator

Our discussion so far has provided formulas for probabilistic (relative) error bounds and the average relative error of $\hat{D}_M^{\mathrm{UB}}$ given that D is known. In practice, however, D is the quantity that is being estimated and, of course, unknown. Thus, we are interested in error bounds that do not depend on the value of D. And in fact, we can obtain these bounds by computing the limit of (6.12) and (6.14) as $D \to \infty$. An alternative approach yielding the same result has been taken by Beyer et al. (2007). They have shown that, as $D \to \infty$, $U_{(M)}$ converges in distribution to the sum of M independent random variables, each having an exponential distribution with rate parameter D. Based on this observation, we can use earlier results from Cohen (1997), who proposed estimator $\hat{D}_M^{\mathrm{UB}}$ for precisely this situation. It follows that

$$\begin{aligned}\Pr\left[\frac{|\hat{D}_M^{\mathrm{UB}} - D|}{D} \leq \epsilon\right] &\approx e^{-\frac{M-1}{1+\epsilon}}\left(1 + \sum_{i=1}^{M-1}\frac{(M-1)^i}{(1+\epsilon)^i i!}\right) - e^{-\frac{M-1}{1-\epsilon}}\left(1 + \sum_{i=1}^{M-1}\frac{(M-1)^i}{(1-\epsilon)^i i!}\right) \\ &= P_{\frac{M-1}{1-\epsilon}}(M) - P_{\frac{M-1}{1+\epsilon}}(M),\end{aligned}$$

where $P_x(a)$ is the *regularized (lower) incomplete gamma function* (Johnson et al. 1992, p. 14), and

$$\mathrm{ARE}[\,\hat{D}_M^{\mathrm{UB}}\,] \approx \frac{2(M-1)^{M-2}}{(M-2)!e^{M-1}} \approx \sqrt{\frac{2}{\pi(M-2)}}.$$

Though slightly conservative, the asymptotic error bounds have the advantageous property that, unlike the exact bounds, they do not involve the unknown quantity D. (In the context of distinct value estimation, asymptotic bounds can be exploited to choose M so as to provide a-priori bounds on the estimation error.)

D. Maximum Likelihood Estimator

The classical statistical approach to estimating unknown parameters is the method of maximum likelihood (ML); see Serfling (1980, sec. 4.2). We apply this approach by casting our estimation problem as a parameter estimation problem. Specifically, recall that $U_{(M)}$ has the probability density $f_{M,D}$ given in (6.4). The ML estimate of D is defined as the value $\hat{D}$ that maximizes the (log) likelihood $L(D; U_{(M)})$ of the observation $U_{(M)}$, defined as $L(D; U_{(M)}) = \ln f_{M,D}(U_{(M)})$. We find this maximizing value by solving the equation $L'(D; U_{(M)}) = 0$, where the prime denotes differentiation with respect to D. We have

$$L'(D; U_{(M)}) = \ln(1 - U_{(M)}) - \Psi(D - M + 1) + \Psi(D + 1),$$

where $\Psi(n)$ denotes the digamma function (Johnson et al. 1992, p. 7). For x sufficiently large and $n > 0$ integer, $\Psi(n) = H_{n-1} - \gamma \approx \ln(n - 1)$, where as before H_{n-1} denotes a harmonic number and γ denotes Euler's constant. Applying this approximation, we obtain

$$\hat{D}_M^{\mathrm{ML}} \approx \frac{M}{U_{(M)}},$$

so that the ML estimator roughly resembles the basic estimator $\hat{D}_M^{\mathrm{BE}}$ provided that $D \gg M$. In fact, our experiments indicated that $\hat{D}_M^{\mathrm{ML}}$ and $\hat{D}_M^{\mathrm{BE}}$ are indistinguishable from a practical point of view. It follows that $\hat{D}_M^{\mathrm{ML}}$ is asymptotically equivalent to $\hat{D}_M^{\mathrm{UB}}$ as $M \to \infty$. A basic result for ML estimators (Serfling 1980, sec. 4.2.2) implies that, for $D \gg M \gg 0$, the estimator $\hat{D}_M^{\mathrm{UB}}$ has, to a good approximation, the minimal possible variance for any estimator of D.

E. General Transaction Sequences

The above estimators do not apply to our AMIND(M) scheme and thus cannot be used with sequences that contain update and deletion transactions. The reason is that AMIND(M) maintains a sample S from $D(R^+)$, not $D(R)$; the net sample S^* from $D(R)$ is extracted when needed. It follows that $\hat{D}_M^{\mathrm{UB}}$ turns into an unbiased estimator of $D^+ = |D(R^+)|$ and, because $D^+ \geq D$, can be biased for D. In this section, we develop our final estimator $\hat{D}_M$, the AMIND(M) version of $\hat{D}_M^{\mathrm{UB}}$.

Denote by $K = |S^*|$ the number of non-deleted items in S. From (6.2), it follows that K has a hypergeometric distribution

$$\Pr[K = k] = H(k; D^+, D, M). \tag{6.15}$$

We now use K to estimate D. We know that $\hat{D}_M^{\mathrm{UB}}$ is an unbiased estimator of D^+; we would like to "correct" this estimator via multiplication by $\rho = D/D^+$. We do not know ρ, but a reasonable estimate is

$$\hat{\rho} = \frac{K}{M}, \tag{6.16}$$

the fraction of sample elements in S that belong to R. This leads to our proposed *generalized estimator*

$$\hat{D}_M = \frac{K}{M}\Big(\frac{M-1}{U_{(M)}}\Big). \tag{6.17}$$

We now establish some basic properties of the estimator. For $d \geq m \geq 1$, set

$$\Delta(d, m, \epsilon) = I_{u(d,m,\epsilon)}(m, d-m+1) - I_{l(d,m,\epsilon)}(m, d-m+1),$$

where as before $I_x(a, b)$ is the regularized incomplete beta function, and $u(d, m, \epsilon)$ and $l(d, m, \epsilon)$ are defined as in (6.13). Take $\Delta(\infty, m, \epsilon) = 0$.

Theorem 6.4. *The estimator $\hat{D}_M$ satisfies $\mathrm{E}[\,\hat{D}_M\,] = D$ for $M > 1$,*

$$\mathrm{Var}[\,\hat{D}_M\,] = \frac{D(M(D^+ - M + 1) - D^+ + D)}{M(M-2)}$$

for $M > 2$, and, if $D > 0$, $\epsilon \in (0, 1)$, and $M \geq 1$,

$$\Pr\left[\frac{|\hat{D}_M - D|}{D} \leq \epsilon \;\middle|\; K = k\right] = \Delta(MD/k, M, \epsilon) \tag{6.18}$$

for $0 \leq k \leq \min(M, D)$, and

$$\Pr\left[\frac{|\hat{D}_M - D|}{D} \leq \epsilon\right] = \sum_{k=0}^{\min(M,D)} \Delta(MD/k, M, \epsilon) H(k; D^+, D, M). \tag{6.19}$$

Proof. A can be seen from (6.2), the distribution of K does not depend on the hash values $\{\,h(r) \colon r \in R^+\,\}$. It follows that the random variables K and $U_{(M)}$ are statistically independent, as are $\hat{\rho}$ and $U_{(M)}$, where $\hat{\rho} = K/M$ as above. By (6.15) and standard properties of the hypergeometric distribution, we have

$$\mathrm{E}[\,K\,] = M\frac{D}{D^+} \tag{6.20}$$

and

$$\mathrm{Var}[\,K\,] = \frac{D(D^+ - D)M(D^+ - M)}{(D^+)^2(D^+ - 1)}. \tag{6.21}$$

It follows from (6.20) that $\mathrm{E}[\,\hat{\rho}\,] = \rho$. Using independence and the unbiasedness of $\hat{D}_M^{\mathrm{UB}}$, we find that

$$\mathrm{E}[\,\hat{D}_M\,] = \mathrm{E}[\,\hat{\rho}\hat{D}_M^{\mathrm{UB}}\,] = \mathrm{E}[\,\hat{\rho}\,]\,\mathrm{E}[\,\hat{D}_M^{\mathrm{UB}}\,] = \rho D^+ = D.$$

The formula for $\mathrm{Var}[\,\hat{D}_M\,]$ follows after some straightforward algebra starting with the definition of variance and using equations (6.9), (6.20), and (6.21). To obtain the relation in (6.18), use the fact that K and $\hat{D}_M^{\mathrm{UB}}$ are independent, and write

$$\Pr\left[\frac{|\hat{D}_M - D|}{D} \leq \epsilon \;\middle|\; K_\cap = k\right] = \Pr\left[\frac{|(k/M)\hat{D}_M^{\mathrm{UB}} - D|}{D} \leq \epsilon\right]$$
$$= \Pr\left[\frac{|\hat{D}_M^{\mathrm{UB}} - D^*|}{D^*} \leq \epsilon\right],$$

where $D^* = (M/k)D$. The desired result then follows by mimicking the proof of theorem 6.2. The final relation in (6.19) follows from (6.18) by unconditioning on K and using (6.15). □

Thus $\hat{D}_M$ is unbiased for D. It also follows from the proof that the estimator $\hat{\rho}$ is unbiased for the fraction of non-deleted elements ρ. Using (6.19), we can compute exact confidence bounds numerically, analogously to the insertion-only case. To obtain practical approximate bounds based on observation of K and $U_{(M)}$, use the representation in (6.18), but replace D by $\hat{D}_M$.

Figure 6.5 displays the influence of deletions on the error bound of the estimate. The figure has been obtained by fixing $D = 1{,}000{,}000$ and varying the value of ρ (and thus D^+). For example, a value of $\rho = 0.1$ means that we (conceptually) inserted 10,000,000 distinct items into the dataset and subsequently removed 9,000,000 of these items, so that 10% of the items are still present. The figure plots the relative error that is achieved with a confidence probability of $1 - \delta = 95\%$. As can be seen, the error bound at first remains stable as ρ decreases, but then quickly approaches 1. The reason for this behavior is that the estimate $\hat{\rho}$, which is required to estimate D, is highly accurate when $\rho > 1/M$ (roughly) but becomes unusable as ρ falls below $1/M$. In this case, the number K of non-deleted items in the sample will be zero with high probability, which in turn leads to a estimate of $\hat{D}_M = 0$. Fortunately, unless M is very small, a large fraction of the dataset has to be deleted to see this behavior. For example, when $M = 8,192$, the decrease in accuracy is visible when $\rho < 0.001$, in which case more than $99,9\%$ of the dataset have been deleted.

F. Predicates

We now return to our initial example, where we were interested in the number of customers that contributed less than a given quantity to the company's total. In settings such as this one, we are not interested in statements about the entire population but about a certain subset of it. Let p be a predicate on the distinct items in R and $\sigma_p(A)$ the elements of A that satisfy p. We assume that all information required to evaluate p is stored together with the sampled items. Our goal is to estimate $\theta = |D(\sigma_p(R)|)$, the number of distinct items in R that satisfy p. There are two ways to compute an estimate $\hat{\theta}$ of θ. A natural way is to extract the net sample

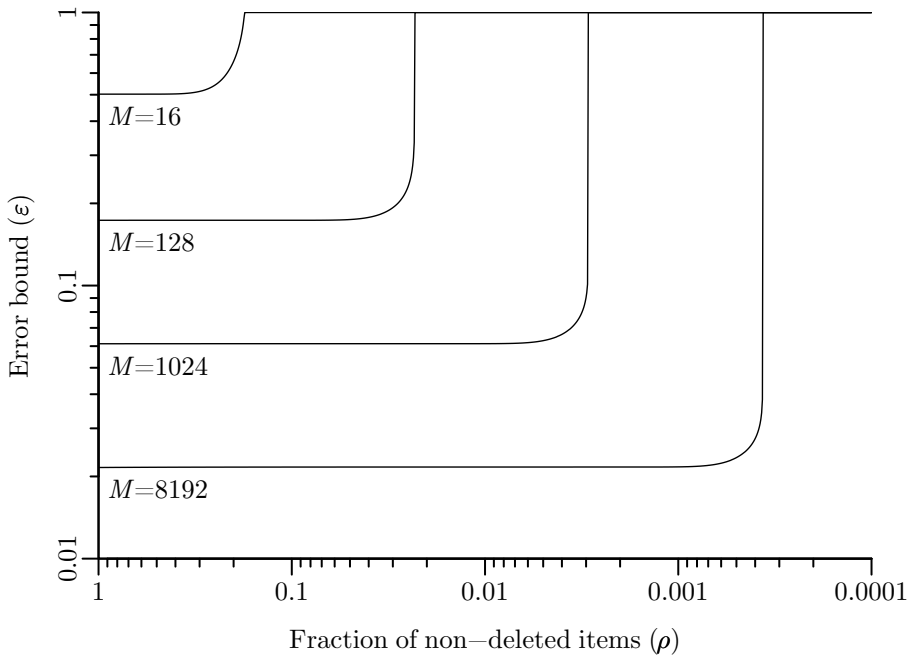


Figure 6.5: Error bounds of $\hat{D}_M$ for $D = 1{,}000{,}000$ and $1 - \delta = 95\%$

S^* from S, determine the number of customers in S^* that satisfy the predicate, and to scale up the resulting value:

$$\hat{\theta}_1 = \frac{|\sigma_p(S^*)|}{|S^*|}\hat{D}_M = \frac{|\sigma_p(S^*)|}{M} \cdot \frac{M-1}{U_{(M)}}.$$

The second way is to (conceptually) delete all items that do not satisfy predicate p from R and S, and then estimate the number of distinct items in the remaining dataset. Denote by $R' = R \uplus \sigma_p(R)$ the relation and by $S' = S \uplus \sigma_p(R)$ the sample after the deletions. Then,

$$\hat{\theta}_2 = \hat{D}'_M = \frac{K'}{M} \cdot \frac{M-1}{U'_{(M)}},$$

where as before K' denote the number of non-deleted items in S' and $U'_{(M)}$ denotes the maximum hash value. Now observe that when running AMIND(M), we have $K' = |\sigma_p(S^*)|$ and $U'_{(M)} = U_{(M)}$. We infer that $\hat{\theta}_1 = \hat{\theta}_2$ so that we can apply theorem 6.4 on θ_2 to obtain properties of θ_1. In fact, from the viewpoint of analysis of estimators, there is no difference between deletions and predicates. As a consequence, the analysis of the preceding section (for sequences with deletions) also applies to plain MIND(M) sampling (with predicates). Finally, note that, in the trading-company example, both $\hat{\theta}_1$ and $\hat{\theta}_2$ are independent from the actual customers in $\sigma_p(S^*)$. This independence simplifies the derivation of probabilistic error bounds for a wide range

of more complex estimates such as, say, the actual contribution of the subset of the customers to the company's total (a.k.a. The Long Tail).

6.2.3 Experiments

The foregoing analysis was made under the assumption that the hash values follow a continuous uniform$[0, 1]$ distribution. In practice, however, hash functions have a discrete range, so that only a finite number of different hash values exists. Normalizing the hash values to $[0, 1]$, it follows that, at best, the hash values only approximate the continuous uniform$[0, 1]$ distribution. Second and more importantly, the hash function must distribute its hash values uniformly on its entire range. In this section, we investigate empirically the validity of our analysis in the real world. Since our experiments in section 6.1F showed that AES is a promising candidate for hash-based sampling, we compare our theoretical results with practical results obtained by AES.

We used one real-world and two synthetic datasets. The real-world dataset has been extracted from "The Gutenberg Webster's Unabridged Dictionary".[7] It comprises $229,480$ words and $21,983$ distinct words. The first synthetic dataset consists of a sequence of integers, while the second dataset comprises random numbers in $[0, 2^{32} - 1]$. Both datasets also contain $21,983$ distinct items.

In our first experiment, we repeatedly computed an AMIND(M) sample from the dataset and then used the sample to estimate the total number of distinct items. We used various values for the sample size M and repeated each individual experiment 100 times. The results are shown in figure 6.6, where we plot the mean estimate along with a 2σ-interval. Here, $\sigma = \mathrm{SE}[\hat{D}_M^{\mathrm{UB}}]$ denotes the standard error of the estimate. Note that by Chebyshev's inequality, at least 75% of the estimates lie within the 2σ-interval. The leftmost bar of each group contains the expected values that correspond to equations (6.10) and (6.11) of our analysis. Each of the remaining bars corresponds to a dataset used in our experiments. As can be seen, the theoretical and observed values agree closely, no matter which dataset has been used. The slight variations visible in the figure result from the fact that we are actually estimating the mean and standard error of the AES estimates based on 100 samples.

In our second and final experiment, we proceeded as before but fixed $M = 1{,}024$. We then estimated the number of distinct values in the dataset after applying a predicate. Such a setting requires usage of the generalized estimator of section 6.2.2E. Figure 6.7 shows our results for varying selectivities. Again, the theoretical values agree closely with the observed values for all three datasets. Admittedly, these experiments are extremely simple and do not allow generalized conclusions. However, they do provide some evidence that our analysis is applicable when AES is used as a hash function. More empirical results for AES (and other hash functions) in the context of distinct-item estimation from minimum hash values are given in Beyer et al. (2007).

[7] Available at http://www.gutenberg.org/etext/673.

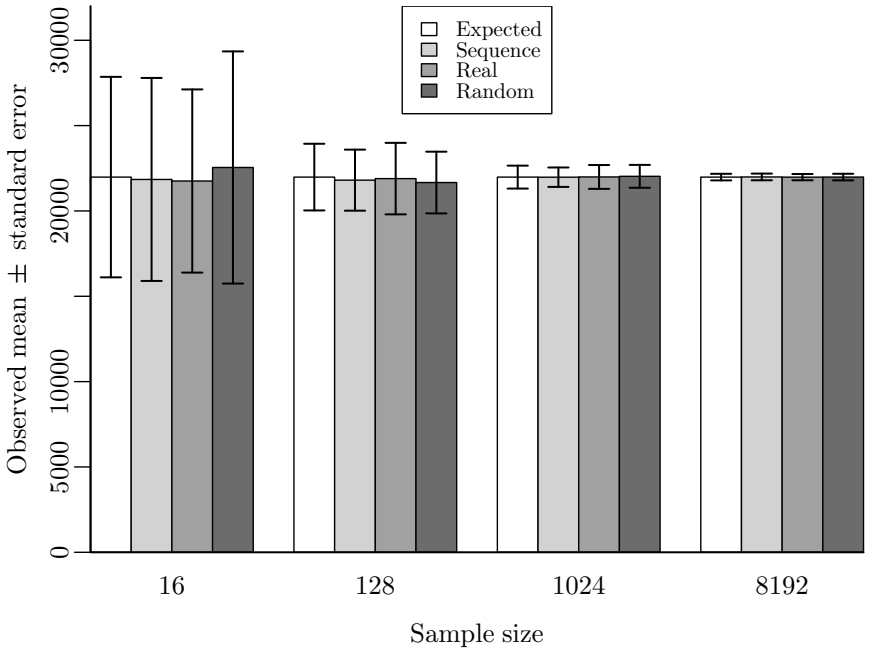


Figure 6.6: Performance of unbiased estimator

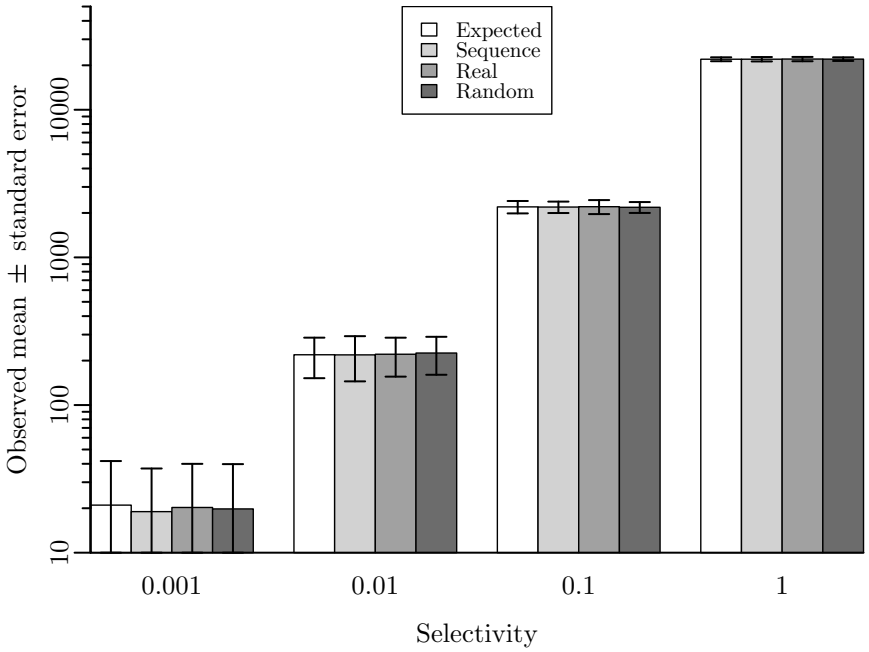


Figure 6.7: Performance of generalized estimator

6.3 Sample Resizing

In this section, we turn attention to the problem of resizing a MIND(M) sample either upwards or downwards. As discussed in previous chapters, resizing algorithms help to avoid under- and oversized samples, respectively, in case of changes of the dataset size. Though our discussion is concerned with plain MIND(M) sampling, it can easily be adapted to the setting of AMIND(M) sampling.

6.3.1 Resizing Upwards

We first consider algorithms for enlarging the sample size. Using an argument as in the proof of theorem 4.3, one can show that any algorithm that increases the size of the MIND(M) sample must access the base data with non-zero probability. In fact, we argue that any resizing algorithm essentially requires a complete scan of R so that resizing upwards is expensive. Suppose that we want to increase the sample size from M to M', where $M < M' \leq D$. The resulting MIND(M') sample then consists of the distinct items with the M' smallest hash values. To obtain the sample, we have to extract the items with the hash values of ranks $M+1, M+2, \ldots, M'$. Thus, we cannot simply extract $M' - M$ random distinct items from R, we need a *specific* set of items that is determined by the hash function being used. To find these items, we have to look at every item in $R \setminus S$ because every item we ignore might be one of the desired items.

Since we have to scan R in any case, a naive algorithm is to completely discard the sample and recompute a fresh MIND(M') sample from scratch. A slightly more efficient approach is to sample $M^+ = M' - M$ (specific) items from $R \setminus S$. To do so, denote by h_{max} the maximum hash value in S. The idea is to run MIND(M^+) sampling on R but only feed the items with a hash value larger than h_{max} into the sampling scheme. Denoting by S^+ the resulting sample, the desired enlarged sample is given by $S' = S \cup S^+$. The advantage of the improved approach is a reduced CPU cost. Using results from Beyer et al. (2007), one finds that the expected cost to compute the MIND(M^+) sample is $O(N + M^+ \log M^+ \log D)$. The merge requires at most $O(M' \log M')$ expected time (because we have to build a randomized treap of S'). This compares favorably to the expected cost of $O(N + M' \log M' \log D)$ for recomputing the MIND(M') sample from scratch. Of course, when reading the base data requires expensive I/O operations, the savings in CPU cost may be negligible.

6.3.2 Resizing Downwards

Resizing the sample downwards can be done efficiently. Our algorithm is based on the following theorem.

Theorem 6.5. *Let S be a MIND(M) sample of some dataset R and let S' be a MIND(M') sample of S, where $M' < M$, obtained using the same hash function. Then, S' is a MIND(M') sample of R.*

Proof. Observe that S' contains the items of S having the M' smallest hash values. Since by definition S contains the items of $D(R)$ having the $M > M'$ smallest hash values, it follows immediately that S' also comprises the items of $D(R)$ having the M' smallest hash values. This is exactly the definition of a MIND(M') samples and the theorem follows. □

Thus, to resize a MIND(M) sample downwards, we run MIND(M') sampling on S, which requires $O(M + M' \log M' \log M)$ expected time according to Beyer et al. (2007). When $M^- = M - M'$ is small, it is more efficient to repeatedly remove the root of the treap, which is known to have the largest hash value, instead of computing a new sample. Because each removal requires at most $O(\log M)$ expected time and M^- items have to be removed, the total expected resizing cost of the modified scheme is $O(M^- \log M)$.

6.4 Sample Combination

The discussion up until now has focused on AMIND(M) sampling from a single base dataset. We now consider the more general case where we are given multiple local datasets, which may be distributed across several nodes, along with their AMIND(M) samples.[8] In the context of set and multiset sampling, we have already discussed the case that the local samples form a partitioning of a "global" dataset that corresponds to the union of the local datasets. Then, in order to derive a sample of the global dataset, it is often more efficient to merge the local samples instead of reconstructing the global dataset and sampling afterwards. In distinct-item sampling, we can push this approach one step further: Our goal is to compute an AMIND(M) sample of a *compound* dataset, that is, a dataset that is created from the local datasets using the *set and/or multiset* operations of union, intersection, and difference. Since the resulting sample is a true AMIND(M) sample, our previous algorithms for sample maintenance, estimating distinct-item counts, and sample resizing apply directly.

6.4.1 Multiset Unions

Consider two datasets A and B along with their AMIND(M) samples S_A and S_B. For convenience, we treat the elements in S_A and S_B as being items instead of (item, counter)-pairs in our formulas. We wish to compute $S_\uplus$, an AMIND(M) sample of $A \uplus B$.

For $V \in \{ A, B \}$, denote by γ_V the sequence that were used to create S_V and let V^+ denote the multiset of all items ever inserted in γ_V. Observe that—by definition of an AMIND(M) sample—S_V comprises the items in $D(V^+)$ with the M smallest hash values. Retaining sample items with frequency 0 and omitting all counter values, we find that S_V is thus a MIND(M) sample of V^+. The distinction between V and

[8] We assume throughout that all local samples have been computed using the same hash function h and sample size bound M. When the sample sizes bounds of the local samples vary, we resize the samples to a common bound using the techniques in section 6.3.

V^+ is important: S_V is an AMIND(M) sample of V but a MIND(M) sample of V^+. In fact, we can say that *any* MIND(M) sample from a set $X \supseteq V$ together with the counter values matching each sample item's frequency in V is an AMIND(M) sample of V.

We leverage the above observation in our algorithm: To derive $S_\uplus$, we compute in a first step a MIND(M) sample of $A^+ \uplus B^+ \supseteq A \uplus B$, and then determine appropriate counter values. Define by $S_A \oplus S_B$ the set comprising the items with the M smallest hash values in $D(S_A \uplus S_B)$. Observe that the $\oplus$ operator is symmetric and associative.

Theorem 6.6. *The set $S_\oplus = S_A \oplus S_B$ is a MIND(M) sample of $A^+ \uplus B^+$.*

Proof. Denote by $G = A^+ \oplus B^+$ the set of items with the M smallest hash values in $D(A^+ \uplus B^+)$. We have to show that $S_\oplus = G$. Observe that G contains the items with the M' smallest values in $D(A^+)$ for some $M' \leq M$, and these M' values therefore are also contained in S_A, i.e., $G \cap D(A^+) \subseteq S_A$. Similarly, $G \cap D(B^+) \subseteq S_B$, so that $G \subseteq S_A \cup S_B$. For any $r \in (S_A \cup S_B) \setminus G$, we have that $h(r) > \max_{r' \in G} h(r')$ by definition of G and because $r \in A^+ \uplus B^+$. Thus G in fact comprises the M smallest hash values in $S_A \cup S_B$, so that $S_\oplus = G$. Now observe that, by definition, G is precisely the MIND(M) synopsis of $A^+ \uplus B^+$. □

Given $S_\oplus$, we can obtain the desired sample $S_\uplus$ by adding counter values. The following lemma helps us to obtain the frequency in A and B of all items in $S_\oplus$.

Lemma 6.1. *For all $r \in S_\oplus$ and $V \in \{ A, B \}$, we have $r \in S_V$ if and only if $r \in V^+$.*

Proof. Let $r \in S_\oplus$, so that r is among the items with the M smallest hash values of $D(A^+ \uplus B^+)$. Then r is among the items with the M smallest hash values in $D(V^+)$ if $r \in V^+$, so that $r \in S_V$ if $r \in V^+$. Conversely, if $r \in S_V$, then $r \in V^+$ because $S_V \subseteq D(V^+)$. □

Denote by $N_V(r)$ the frequency of item r in multiset V. Because S_V is an AMIND(M) sample of V, we have $N_{S_V}(r) = N_V(r)$. Combining this observation with lemma 6.1, we find that for all $r \in S_\oplus$

$$N_V(r) = \begin{cases} N_{S_V}(r) & r \in S_V \\ 0 & \text{otherwise,} \end{cases} \tag{6.22}$$

so that $N_V(r)$ can be computed from S_V. Now observe that

$$N_{A \uplus B}(r) = N_A(r) + N_B(r) \tag{6.23}$$

denotes the frequency of item R in $A \uplus B$. Therefore, the final sample is given by

$$S_\uplus = \{ (r, N_{A \uplus B}(r)) \mid r \in S_\oplus \}. \tag{6.24}$$

Here, $N_{A \uplus B}$ is used as a function to combine the counter values obtained from both samples.

Theorem 6.7. *The sample $S_{\uplus}$ is an AMIND(M) sample of $A \uplus B$.*

Proof. According to (6.24), $S_{\oplus}$ and $S_{\uplus}$ contain the same items. It follows from theorem 6.6 that $S_{\uplus}$ contains the items with the M smallest hash values in $D(A^+ \uplus B^+) \supseteq D(A \uplus B)$. From lemma 6.1 and equations (6.22), (6.23), and (6.24) we conclude that $N_{S_{\uplus}}(r) = N_{A \uplus B}(r)$ for all $r \in S_{\uplus}$. Thus, $S_{\uplus}$ is an AMIND(M) sample of $A \uplus B$ and the theorem follows. □

The size of the resulting (net) sample is equal to M when $A^+ = A$ and $B^+ = B$, but may otherwise be smaller than M. Properties of the sample size are discussed in section 6.4.3.

6.4.2 Other Operations

As before, consider two datasets A and B with corresponding AMIND(M) samples S_A and S_B. We first show how to compute samples $S_{\uplus}$ and $S_{\setminus^+}$ of the multiset intersection and multiset difference, respectively, of A and B. The key idea remains the same as for multiset unions: we first compute $S_{\oplus}$, which is a MIND(M) sample of $A^+ \uplus B^+$, and then compute the frequencies of the items in $A \uplus B$ or $A \setminus^+ B$, respectively. Thus, we apply the above algorithm for multiset unions, but replace the formula used to combine the counter values by

$$N_{A \uplus B}(r) = \min(N_A(r), N_B(r))$$

for multiset intersection and

$$N_{A \setminus^+ B}(r) = \max(N_A(r) - N_B(r), 0)$$

for multiset difference.

Using the same idea, we can also support the set operations $\{\cup, \cap, \setminus\}$. To do so, we cap the counter values of the participating datasets and the result by 1, thereby eliminating duplicates. The corresponding functions for combining the frequencies are given by

$$\begin{aligned} N_{A \cup B}(r) &= \min(1, N_A(r) + N_B(r)), \\ N_{A \cap B}(r) &= \min\big[1, \min(N_A(r), N_B(r))\big], \\ N_{A \setminus B}(r) &= \min(1, N_A(r)) - \min(1, N_B(r)). \end{aligned}$$

We can now generalize our results to obtain an AMIND(M) sample over arbitrary expressions involving the operations $\{\uplus, \uplus, \setminus^+, \cup, \cap, \setminus\}$. Given an expression E over datasets $A_1, \ldots, A_n$ and the respective samples $S_{A_1}, \ldots, S_{A_n}$, we set $S_{\oplus} = S_{A_1} \oplus \cdots \oplus S_{A_n}$. To construct S_E, we obtain the frequency of each item in E by recursively applying the above formulas on $N_E(r)$. The recursion stops when only item frequencies of the base datasets remain in the formula. For example, the expression $(A_1 \uplus A_2) \setminus^+ A_3$ becomes

$$\begin{aligned} N_{(A_1 \uplus A_2) \setminus^+ A_3}(r) &= \max\big[N_{A_1 \uplus A_2}(r) - N_{A_3}(r), 0\big] \\ &= \max\big[\min(N_{A_1}(r), N_{A_2}(r)) - N_{A_3}(r), 0\big]. \end{aligned}$$

The final sample S_E is then given by

$$S_E = \{\, (r, N_E(r)) \mid r \in S_\oplus \,\}\,.$$

Of course, the number of items in the net sample S_E^*—that is, the number of non-zero items in S_E—depends on the number of deletions in the base datasets as well as on the fraction of items that satisfy expression E. In the next section, we look more closely at the distribution of the net sample size.

6.4.3 Analysis of Sample Size

Let E be an (multi)set expression on $A_1, \ldots, A_n$ and set $A = A_1 \uplus \cdots \uplus A_n$. Moreover, define $A_1^+, \ldots, A_n^+$ as before and set $A^+ = A_1^+ \uplus \cdots \uplus A_n^+$. Then, the items in $S_\oplus$ and thus the items in S_E, including zero items, form a size-M uniform sample of $D(A^+)$. To derive the size of the net sample S_E^*, we are interested in the number of items in $D(A^+)$ that also belong to the result of expression E. Set $D = |D(A)|$, $D^+ = |D(A^+)|$ and let $D_E = |D(E)|$ denote the number of distinct items in the result of E. Applying (6.2), we find that the size of the net sample S_E^* follows the hypergeometric distribution with

$$\Pr[\,|S_E^*| = k\,] = H(k; D^+, D_E, M).$$

It follows that, in expectation, the sample size equals

$$\mathrm{E}[\,|S_E^*|\,] = \frac{D_E}{D^+} M = \frac{D_E}{D} \cdot \frac{D}{D^+} \cdot M.$$

As expected, the effective sample size depends on the selectivity of E (that is, D_E/D) and the fraction of non-deleted items (D/D^+). When E is highly selective and/or there have been many deletions, the resulting sample will be small. For this reason, our techniques can only be used when E is sufficiently large. Note that, however, small result sizes may not be a problem in the context of distinct-item estimation, where we are only interested in an estimate of D_E. As indicated in figure 6.5 and more closely examined in Beyer et al. (2007), these estimates can still be quite accurate.

6.5 Summary

We have extended the well-known min-hash sampling scheme for distinct-item sampling with support for deletions. The key idea behind our augmented min-hash sampling scheme, AMIND(M), is to augment the sample with the multiplicities of the sampled items. As perhaps expected, the sample size distribution of our scheme does not depend on the total number of deletions but only on the number of deletions that ultimately remove a distinct item from the dataset (i.e., the last copy). If these deletions are relatively infrequent, they are effectively compensated over the long run. A large part of our discussion focused around the problem of estimating the number

of distinct items in the base data or a subset thereof. This problem is important because distinct-count estimates are required to determine scale-up factors from the sample to the full dataset. We reviewed earlier estimators from related problem areas and used them to develop an unbiased, asymptotically optimal estimator. The key ideas behind our estimator can also be exploited for the sole purpose of distinct-item estimation, that is, without even maintaining the sample. We have also shown how multiple AMIND(M) samples can be combined to derive an AMIND(M) sample of an arbitrary combination of their underlying datasets, where the combination consists of (multi)set operations of union, intersection and difference. This flexibility makes our AMIND(M) scheme interesting for traditional set sampling.

Chapter 7

Data Stream Sampling

With this chapter,[1] we leave the area of database sampling and turn our attention to sampling from a time-based sliding window defined over a data stream. Our main focus lies on bounded sampling schemes; we provide a novel uniform sampling scheme and discuss its properties in terms of sample size and computational cost. We also develop a stratified scheme that overcomes some of the limitations of uniform sampling for sliding windows.

In section 7.1, we prove as a negative result that no bounded-space uniform sampling scheme over a time-based sliding window can guarantee a minimum sample size. As a byproduct, we infer that priority sampling, PS(M),—which maintains a fixed-size sample and is consequently unbounded in space—is optimal in terms of space consumption. Nevertheless, in spite of being optimal, PS(M) has a high space overhead that leads to a low space efficiency. To sample in bounded space, we develop a related scheme called bounded priority sampling, BPS(M), which can be seen as a modification of priority sampling, although the underlying idea is different. BPS(M) cannot provide strict sample size guarantees but it is able to provide strong probabilistic ones. In fact, our experiments indicate that—in addition to enforcing space bounds—BPS(M) has a higher space efficiency than PS(M). Finally, we show how to sample without replacement and how to estimate the window size directly from the sample by applying our techniques from chapter 6.

In section 7.2, we propose a stratified sampling scheme for time-based sliding windows. Recall that in stratified sampling, the dataset is divided into a set of disjoint strata and a sample is taken from each of these strata. Usually, the strata are constructed as required by the application and we can make use of arbitrary sampling schemes in each stratum. For this reason, we did not explicitly discuss stratified sampling in earlier chapters. However, the situation is different for data stream sampling because we can exploit stratification to facilitate efficient sample maintenance. In fact, out merge-based stratification algorithm divides the window into strata of approximately equal size; it then maintains a uniform sample of each stratum. The algorithm merges adjacent strata from time to time; the main challenge is to decide when and which strata to merge. In our solution, we treat the problem as an optimization problem and give a dynamic programming algorithm to determine

[1] The material in this chapter has been developed jointly with Wolfgang Lehner. The chapter is based on Gemulla and Lehner (2008) with copyright held by ACM. The original publication is available at http://portal.acm.org/citation.cfm?id=1376616.1376657.

the optimum stratum boundaries. The resulting algorithm, termed MBS, has an even higher space efficiency than BPS(M). The downside is that stratified samples cannot be used with all applications so that the advantages of MBS cannot always be exploited.

7.1 Uniform Sampling

We start with the discussion of uniform sampling. Our negative result about bounded-space uniform sampling is stated and proven in section 7.1.1. We briefly review priority sampling in section 7.1.2, before introducing and analyzing our bounded priority sampling scheme in section 7.1.3. Afterwards, we give an estimator for the window size in section 7.1.4 and outline some optimizations for our scheme in section 7.1.5. Section 7.1.6 concludes with an overview of our experimental study.

We pick up our notation from section 3.5.4. In short, the data stream R is modeled as a sequence of items $e_1, e_2, \ldots$. Each item e_i has form (t_i, r_i),[2] where $t_i \in \mathbb{R}$ is a timestamp and $r_i \in \mathscr{R}$ is the item's value. The set of items that arrive until time t (including) is denoted $R(t)$. A time-based sliding window of *length* Δ is denoted by $W_\Delta(t) = R(t) \setminus R(t-\Delta)$ or $W(t)$ for short. The window *size*—the number of items in the window—is denoted by $N(t) = |W(t)|$. The uniform sample from $W(t)$ is denoted by $S(t)$.

7.1.1 A Negative Result

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

Theorem 7.1. *Fix some time t and set $N = N(t)$. Any algorithm that maintains a fixed-size uniform random sample of size M has to store at least $\Omega(M \log N)$ items in expectation.*

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

Since $\mathscr{A}$ is a uniform sampling scheme, item e_{m+1} is reported at time t_0^- with probability M/N. At time t_1^-, only $N-1$ items remain in the window and item

[2] We do not use triple (i, t_i, r_i) as in section 3.5.4 because the item index i is unimportant for time-based windows.

e_{m+2} is reported with probability $M/(N-1)$. The argument can be repeated until at time t^-_{N-M}, all the M remaining items are reported by $\mathscr{A}$. It follows that

$$\Pr[I_j = 1] = \begin{cases} M/(N-j) & 0 \le j < N-M \\ 1 & \text{otherwise} \end{cases} \tag{7.1}$$

for $0 \le j < N$. Note that only the marginal probabilities are given in (7.1); joint probabilities like $\Pr[I_1 = 1, I_2 = 1]$ depend on the internals of $\mathscr{A}$. By the linearity of expected value, and since $\mathrm{E}[I_j] = \Pr[I_j = 1]$, we find that

$$\mathrm{E}[X] = \sum_{j=0}^{N-1} \mathrm{E}[I_j] = M(H_N - H_M + 1) = \Omega(M \log N)\,.$$

This completes the proof. □

It follows directly that it is impossible to maintain a fixed-size uniform random sample from a time-based sliding window in bounded space. By theorem 7.1, such maintenance requires that we store a number of items logarithmic to the window size. Because the window size is unbounded, so is the expected space consumption. Of course, the worst-case space consumption is at least as large; it is also unbounded. Moreover, it is impossible to guarantee a minimum sample size in bounded space because any algorithm that guarantees a minimum sample size can be used to maintain a sample of size 1.

At a first glance, our negative results seem to imply that efficient sample maintenance in bounded space is infeasible. This is because we cannot provide any strict guarantees on the sample size. However, we will see later that we can still provide strong probabilistic guarantees. Our algorithms completely avoid the $\Omega(\log N)$ multiplicative overhead of theorem 7.1; they have a superior space efficiency than any fixed-size sampling scheme.

7.1.2 Priority Sampling Revisited

We briefly review priority sampling because it forms the basis of our schemes. Recall that priority sampling, or PS(1), maintains a size-1 uniform sample; larger samples are obtained by maintaining multiple samples in parallel. From theorem 7.1, we immediately learn that the priority sampling requires unbounded space.

The idea behind priority sampling is as follows: A random priority p_i chosen uniformly and at random from the unity interval is associated with each item $e_i \in R$. The sample $S(t)$ then consists of the item in $W(t)$ with the highest priority.[3] In addition to the sample item, the scheme stores a set of *replacement items*, which replace the highest-priority item when it expires. This replacement set contains all the items for which there is no item with both a larger timestamp and a higher

[3] That's similar to MIND(1), where we keep the item with the smallest hash value.

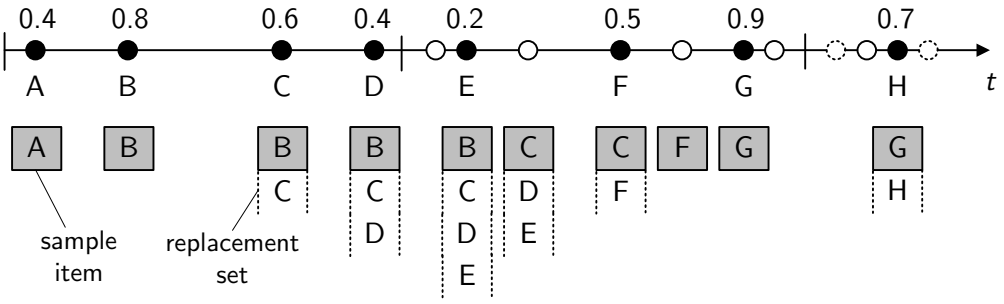


Figure 7.1: Illustration of priority sampling

priority because only these items may eventually become the highest-priority item at some later point of time.

Figure 7.1 gives an example of the sampling process. A solid black circle represents the arrival of an item; its name and priority are given below and above, respectively. The vertical bars on the timeline indicate the window length, item expirations are indicated by white circles, and double-expirations[4] are dotted white circles. Below the timeline, the current sample item and the set of replacement items are shown. It can be seen that the number of replacement items stored by the algorithm varies over time. In fact, the replacement set is the reason for the unbounded space consumption: it contains between 0 and $N(t) - 1$ items and roughly $\ln N(t)$ items on average (Babcock et al. 2002). From theorem 7.1, we infer that priority sampling is asymptotically optimal in terms of expected space consumption; we cannot hope to find a better fixed-size scheme.

7.1.3 Bounded Priority Sampling

In this section, we develop our bounded priority sampling scheme, BPS(M), which is based on priority sampling. In what follows, we describe the BPS(1) scheme. To obtain a larger with-replacement sample, one may run multiple independent BPS(1) samples in parallel. A more efficient without-replacement scheme is given in section E.

A. Algorithmic Description

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

(a) *Arrival of item e_i.* If there is currently no candidate item or if the priority of e_i is larger than the priority of the candidate item, e_i becomes the new

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

candidate item and the old candidate is discarded. Otherwise, the arriving item is ignored.

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

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

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

The complete pseudocode of BPS(1) is given as algorithm 7.1. The internal data structures of algorithm 7.1 are updated only when a new item arrives in the stream (Arrival) or when the sample is queried (Report). This has the advantage that the sample does not have to be monitored when no data arrives in the stream. In both cases, arrival and query, expired candidates become test items and double-expired test items are discarded (RemoveExpired).

B. Example

Before we assert the correctness of BPS and analyze its properties, we give an example of the sampling process in figure 7.2. The current candidate item and test item are shown below the timeline. If the candidate item is shaded, it is reported as a sample item; otherwise, no sample item is reported. The letters below the BPS data structure refer to cases (a), (b) and (c) above. As long as no expiration occurs, the candidate stored by BPS equals the highest-priority item in the window and is therefore reported as a sample item. The situation changes as B expires. BPS then makes item B the test item and—because there is no candidate item anymore—fails to report a sample item. This failure can be seen as a consequence of theorem 7.1: BPS is a bounded-space sampling scheme and thus cannot guarantee a fixed sample size. Item F becomes the new candidate item upon its arrival. However, F is not

Algorithm 7.1 Bounded priority sampling

```
1: Δ: window length
2: t: current time
3: (t, r): timestamp and value of arriving item
4: e_cand = (t_cand, r_cand, p_cand): timestamp, value and priority of candidate item
5: e_test = (t_test, p_test): timestamp and priority of test item
6: RANDOM(): returns a uniform random number between 0 and 1
7:
8: ARRIVAL(t, r):
9: call REMOVEEXPIRED(t)
10: p ← RANDOM()
11: if e_cand = empty ∨ p > p_cand then
12:    e_cand ← (t, r, p)
13: end if
14:
15: REPORT(t):
16: call REMOVEEXPIRED(t)
17: if e_cand = empty ∨ e_test = empty then
18:    return e_cand
19: else if p_cand > p_test then
20:    return e_cand
21: else
22:    return empty
23: end if
24:
25: REMOVEEXPIRED(t) :
26: if e_cand ≠ empty ∧ t_cand ≤ t − Δ then
27:    e_test ← (t_cand, p_cand)
28:    e_cand ← empty
29: end if
30: if e_test ≠ empty ∧ t_test ≤ t − 2Δ then
31:    e_test ← empty
32: end if
```

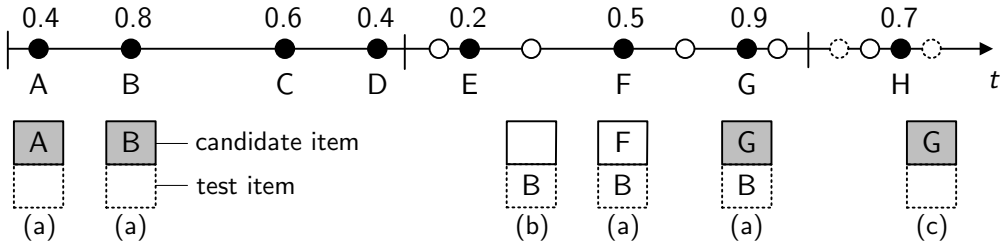


Figure 7.2: Illustration of bounded priority sampling

reported because its priority is lower than the priority of the test item B. And in fact, not F but C is the highest-priority item in the window at this time. Later C expires and F does become the highest-priority item in the window, but we still do not report F since we are not aware of this situation. As G arrives, however, we report a sample item again because G has a higher priority than the test item B. Finally, item B is discarded from the BPS data structure as it double-expires.

C. Proof of Correctness

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

Theorem 7.2. *BPS is a uniform sampling scheme, that is, for any $e \in W(t)$, we have*

$$\Pr\big[\, S(t) = \{\, e \,\} \,\big|\, |S(t)| = 1 \,\big] = \frac{1}{N(t)}.$$

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

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

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

3. Item e' has a higher priority than $e_{\max}$. Then, e' is still the candidate item at the time of its expiration, since there is no higher-priority item in $W(t)$ that might have replaced e'. Thus, item e' becomes the test item upon its expiration

and continues to be the test item up to time t—it double-expires somewhere in the interval $(t, t+\Delta)$. It follows that no item is reported at time t so that $S = \emptyset$, because the priority of the candidate item ($\leq p_{\max}$) is lower than the priority p' of the test item.

To summarize, we have

$$S = \begin{cases} \{ e_{\max} \} & \text{no candidate item at time } t - \Delta \\ \{ e_{\max} \} & p_{\max} > p' \\ \emptyset & \text{otherwise.} \end{cases} \tag{7.2}$$

Uniformity now follows since (7.2) does not depend on the values, timestamps or order of the individual items in $W(t)$. For any $e \in W(t)$, we have

$$\Pr[\, S = \{ e \} \mid |S| = 1 \,] = \Pr[\, e = e_{\max} \,] = \frac{1}{N(t)}$$

and the theorem follows. □

D. Sample Size Properties

We now analyze the sample size of the BPS scheme. Clearly, the sample size is probabilistic and its exact distribution depends on the entire history of the data stream. However, in the light of theorem 7.3 below, it becomes evident that we can still provide a *local* lower bound on the probability that the scheme produces a sample item. The lower bound is local because it changes over time; we cannot guarantee a global lower bound other than 0 that holds at any arbitrary time without a-priori knowledge of the data stream.

Theorem 7.3. *The probability that BPS succeeds in producing a sample item at time t is bounded from below by*

$$\Pr[\, |S(t)| = 1 \,] \geq \frac{N(t)}{N(t-\Delta) + N(t)}.$$

Proof. BPS produces a sample item if the highest-priority item $e_{\max} \in W(t)$ has a higher priority than the candidate item e' stored in the BPS data structure right before the start of $W(t)$; see (7.2) above. In the worst case, e' equals the highest-priority item in $W(t-\Delta)$. Now suppose that we order the items in $W(t-\Delta) \cup W(t)$ in descending order of their priorities. BPS succeeds for sure if the first of the ordered items is an element of $W(t)$. Since the priorities are independent and identically distributed, this event occurs with probability $N(t)/(N(t-\Delta) + N(t))$ and the assertion of the theorem follows. □

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

E. Sampling Multiple Items

The BPS scheme as given above can be used to maintain a single-item sample. As mentioned before, larger samples can be obtained by maintaining M independent BPS samples $S_1, \ldots, S_M$ in parallel. The sample is then set to $S(t) = S_1(t) \uplus \cdots \uplus S_M(t)$. We have

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

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

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

$$S_{\mathrm{WOR}}(t) = \text{top-M}\left(S_{\mathrm{cand}}(t) \cup S_{\mathrm{test}}(t)\right) \cap S_{\mathrm{cand}}(t),$$

where top-M(A) determines the items in A with the M highest priorities. The sample thus consists of the candidate items that belong to the M highest-priority items currently stored in the data structure. Note that for $M = 1$, BPS(M) and BPSWOR(M) coincide. $S_{\mathrm{WOR}}(t)$ is a uniform random sample of $W(t)$ without replacement; the proof is similar to the proof of theorem 7.2. Also, using an argument as in the proof of theorem 7.3, we can show that

$$\mathrm{E}\,[\,S_{\mathrm{WOR}}(t)\,] \geq M \frac{N(t)}{N(t-\Delta) + N(t)}.$$

Thus, BPS(M) and BPSWOR(M) have the same lower bound on the expected gross sample size. The cost of processing a window of size N is $O(MN)$ if the candidates are stored in a simple array. A more efficient approach—which also improves the cost in comparison to BPS(M)—is to store the candidates in a treap; see the discussion on page 182. In the treap, the items are arranged in order with respect to the

timestamps and in heap-order with respect to the priorities. The expected cost of BPSWOR then decreases to $O(N + M \log M \log N)$ in expectation.[5]

7.1.4 Estimation of Window Size

Sampling from a sliding window is similar to sampling from the distinct items in a dataset in that we do not know the size of the underlying window or dataset, respectively. For some applications, however, it is important to be able to at least estimate the window size in order to make effective use of the sample. For example, the window sum of an attribute is typically estimated as the sample average of the respective attribute multiplied by the window size. Thus—in some applications—knowledge of the window size is important to determine scale-up factors.

In our discussion on distinct-item sampling, we pointed out that the dataset size can be estimated using either external or internal estimation. The same arguments apply here: Exact maintenance of the number of items in the window requires that we store all the timestamps in the window in order to deal with expirations. Typically, this approach is infeasible in practice. Approximate data structures do exist and can be leveraged to support the sampling process (Datar and Muthukrishnan 2002). If such alternate data structures are unavailable or infeasible to maintain, we can come up with an estimate of the window size directly from the sample. Set $W_{2\Delta}(t) = W_\Delta(t-\Delta) \cup W_\Delta(t)$ and denote by $p_{(M)}$ the priority of the item with the M-th highest priority in $W_{2\Delta}(t)$. We make use of the distinct-count results of section 6.2.2:[6] According to (6.17), an unbiased estimator for $N(t)$ is given by

$$\hat{N}_W(t) = \frac{|W_\Delta(t) \cap \text{top-M}\, W_{2\Delta}(t)|}{M} \cdot \frac{M-1}{1-p_{(M)}}.$$

Here, the first factor estimates the fraction of non-expired items in $W_{2\Delta}(t)$ from the top-M items (which can be viewed as a random sample of $W_{2\Delta}$), while the second factor is an estimate of $|W_{2\Delta}(t)|$ itself. Now, suppose that we maintain the sample using BPSWOR(M). Set $S_2(t) = S_{\text{cand}}(t) \cup S_{\text{test}}(t)$ and denote by $p'_{(M)}$ the priority of the item with the M-th highest priority in S_2. Consider the estimator

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

This estimator is similar to $\hat{N}_W(t)$ but solely accesses information available in the sample. Both estimators coincide if and only if top-M $S_2(t) =$ top-M $W_{2\Delta}(t)$. This happens if at least $|W_\Delta(t-\Delta) \cap \text{top-M}\, W_{2\Delta}(t)|$ items have been reported as the sample at time $t-\Delta$. Otherwise, the first factor in $\hat{N}_S(t)$ will overestimate the first

[5] Following an argument as in Beyer et al. (2007), at most $O(M \log N)$ items of the window are accepted into the candidate set in expectation and each accepted item incurs an expected cost of $O(\log M)$. At most M items (double-)expire while processing a window, so that the expected cost to process (double-)expirations is $O(M \log M)$.

[6] The results in section 6.2.2 also apply to priority sampling without replacement.

factor in $\hat{N}_W(t)$, while the second factor will underestimate the respective factor in $\hat{N}_W(t)$. In our experiments, we found that the estimator $\hat{N}_S$ has negligible bias and low variance. Thus, both over- and underestimation seem to balance smoothly, although we do not make any formal claims here.

7.1.5 Optimizations

We now briefly present two optimizations that improve upon the BPS and BPSWOR schemes presented above.

Early expiration. Fix some time t and suppose that the application using the sample can guarantee that it does not query the sample until some time t', where $t < t' < t + \Delta$. This situation might occur, for example, when the sample is queried on a periodical basis. If such knowledge is available, we can improve the probability that BPS produces a sample item at time $t + \Delta$ and later. The idea is to check at time t whether the current candidate item e expires before time t'. If so, we *immediately* make e the test item without waiting for its expiration. Then, at time $t + \Delta$, the sample is known to contain the highest-priority item of $W(t + \Delta)$; see the first case in (7.2). To make this knowledge available to the sampling algorithm, we backdate item e to time $t - \Delta$ right before we make it the test item; e then double-expires at time $t + \Delta$. This approach does not affect the correctness of the scheme, but increases the sample size if the sample is queried infrequently. It can also be used with BPSWOR.

Switching sampling on and off. When BPSWOR is used for sample maintenance, the sample size can also be increased at times where the stream rate is very low. We say that an item is discarded if (1) the item arrives but is not made a candidate or (2) the item is evicted from the candidate set due to the arrival of a new higher-priority item. Now suppose that BPSWOR did not discard an item for a time span of at least Δ. It is easy to see that the candidate set then contains all the items in the window. Thus we report the entire candidate set as the sample when the timestamp of the last eviction is more then Δ time units in the past. With this optimization, sampling is "switched off" as soon as the algorithm detects that the entire window fits into the available space. It is "switched on" again when the window size exceeds the available space.

7.1.6 Experiments

We conducted an experimental study of BPS and BPSWOR for both synthetic and real-world datasets. In summary, we found that:

- BPSWOR is the method of choice when the available memory is limited and the data stream rate is varying. It then produces larger samples than both Bernoulli sampling and PSWOR. Also, BPSWOR is the only scheme that does

not require a-priori information about the data stream and guarantees an upper bound on the memory consumption.

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

- BPSWOR is superior to BPS because it is significantly faster and samples without replacement.

- The window-size estimate of section 7.1.4 has low relative error. The relative error decreases with an increasing sample size.

A. Setup

We implemented BPS(M), BPSWOR(M), PS(M), PSWOR(M) and BERNW(q) in Java 1.6. The experiments have been run on a workstation PC with a 3 GHz Intel Pentium 4 processor and 2.5 GB main memory.

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

As before, we do not report the estimation error of a specific estimate derived from the sample but rather report the (distribution of the) sample size. The motivation behind this approach is that two uniform samples of the same size are identical in distribution, no matter which scheme has been used to compute them. Larger samples inevitably lead to a smaller estimation errors: schemes that produce larger samples are superior to schemes that produce smaller samples.

B. Synthetic Data

In a first experiment, we compared BERNW, PSWOR and BPSWOR. Neither BERNW nor PSWOR can guarantee an upper bound on the space consumption and—without a-priori knowledge of the stream—it is impossible to parametrize them to only infrequently exceed the space bound. The goal of this experiment is to compare the sample size and space consumption of the three schemes under the assumption that such a parametrization is possible. For this purpose, we generated a synthetic data stream, where each item of the data stream consists of an 8-byte timestamp and 32 bytes of dummy data. To generate the timestamps, we modeled the arrival rate of the stream using a sine curve with a 24h-period, which takes values between 3,000 and 5,000 items per hour. We superimposed the probability density function (PDF) of a normal distribution with mean 24 and variance 0.5 on the sine curve; the PDF has been scaled so that it takes a maximum of 30,000 items per hour. This models real-world scenarios where the peak arrival rate (scaled PDF) is much higher than the average arrival rate (sine curve).

We used the three sampling schemes to maintain a sample from a sliding window of 1 hour length; the window size over time is given in figure 7.3. We used a space budget of 32 kbytes; at most 819 items can be stored in 32 kbytes space. For the sampling schemes, we used parameters $M_{\text{BPSWOR}} = 585$ (number of candidate/test items), $M_{\text{PSWOR}} = 113$ (sample size) and $q_{\text{BERNW}} = 0.0276$ (sampling rate). The latter two parameters have been chosen so that the expected space consumption at the peak window size equals 32 kbytes—as discussed above, this parametrization is only possible because we know the behavior of the stream in advance. During the sampling process, we monitored both sample size and space consumption; the results are given in figure 7.4.

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

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

Bounded priority sampling. BPSWOR produces a sample size of ≈ 300 items in the average case and therefore has a much better space utilization than BERNW and PSWOR. When the peak arrives, the sample size first grows above, then falls below the 300-item average. Afterwards it stabilizes again. By theorem 7.3, the sample size depends on the ratio of the number of items in the current window to the number of items in both the current and previous window together. This fraction is roughly constant in the average case but varies with the arrival of the peak load.

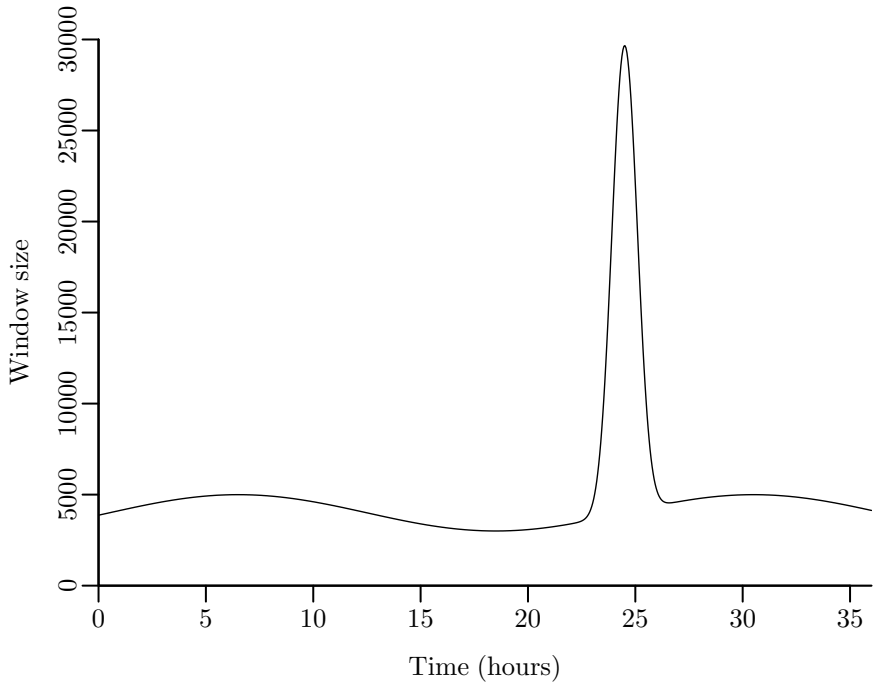


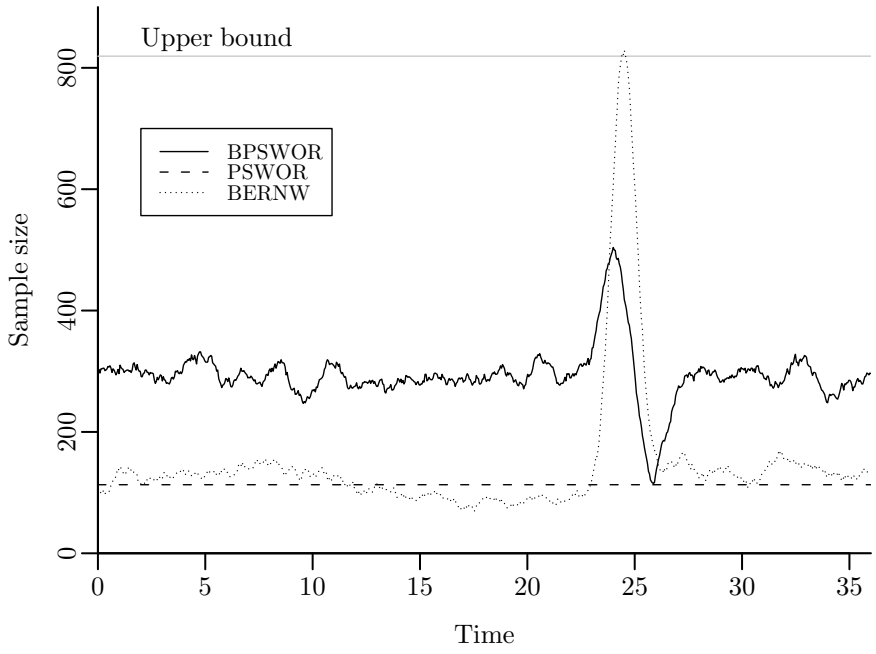
Figure 7.3: Progression of the window size over time (synthetic data)

Interestingly, the scheme almost always uses the entire available memory to store the candidate items and the test items. The space consumption slightly decreases when the peak arrives. In this case, we store fewer than M test items because—due to the increased arrival rate—candidate items are replaced by new items before their expiration and so do not become test items.

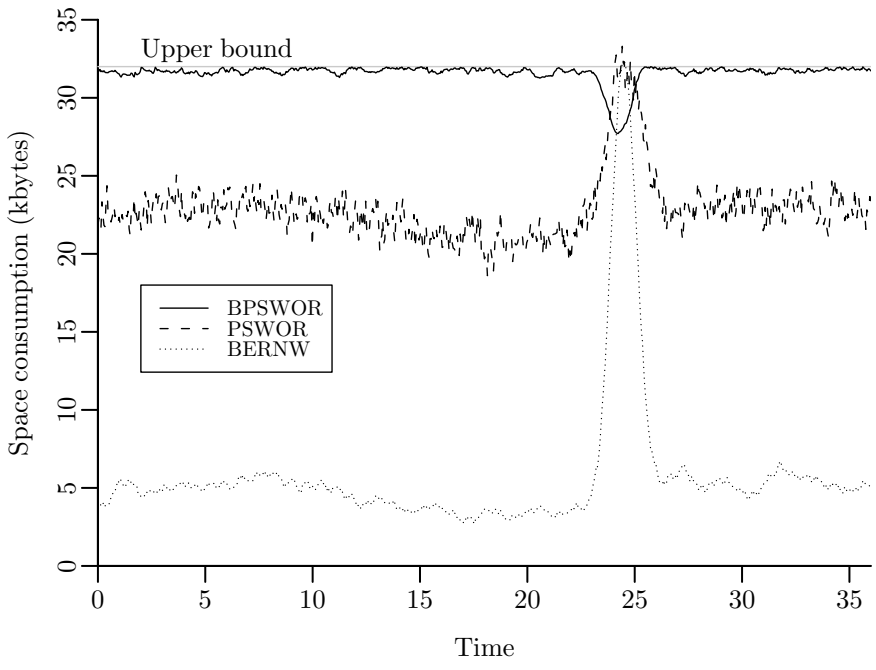
To summarize, each of the three schemes has a distinctive advantage: BERNW does not have any memory overhead, PSWOR guarantees a fixed sample size and BPSWOR samples in bounded space. If the available memory is limited, BPSWOR is the method of choice because it produces larger sample sizes than BERNW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

C. Real data

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



(a) Sample size

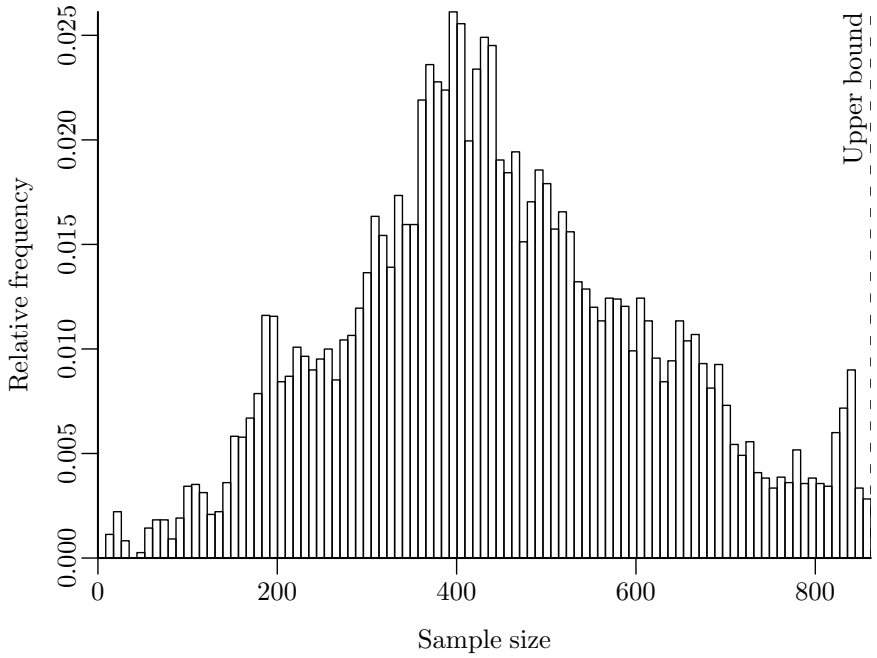


(b) Space consumption

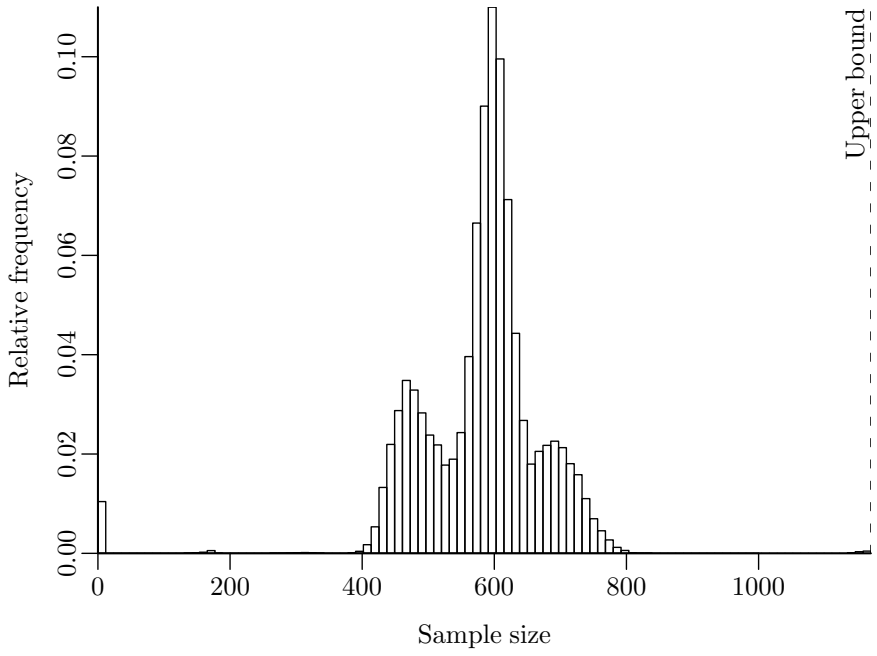
Figure 7.4: Progression of sample size and space consumption over time (synthetic data)

Sample size. In figure 7.5a, we report the distribution of the BPSWOR sample size for the NETWORK dataset; similar results were observed with BPS. We used a space budget of 32 kbytes, which corresponds to a value of $M = 862$. The figure shows a histogram of the relative frequencies for varying sample sizes. As can be seen, the sample size concentrates around the average of 448 items and varies in the range from 11 to 862 items. The standard deviation of the sample size is 173 and in 95% of the cases, the sample size was larger than 176 items. By theorem 7.3, the sample size depends on the ratio of the size of the current window to the size of both the prior and the current window, or the window-size ratio for short. In figure 7.6a, we give a histogram of the window-size ratios in the NETWORK dataset. As can be seen, the distribution of the window-size ratio has a striking similarity to the distribution of the sample size. To further investigate this issue, we give a box-and-whisker plot of the sample size for varying ranges of window-size ratios in figure 7.7a. In a box-and-whisker plot, a box ranging from the first quartile to the third quartile of the distribution is drawn around the median value. From the box, whiskers extend to the minimum and maximum values as long as these values lie within 1.5 times the interquartile distance (=height of the box); the remaining values are treated as outliers and are directly added to the plot. From the figure, it becomes evident that the window-size ratio has a significant influence on the sample size. Also, for each window-size ratio, the sample size has low variance and is skewed towards larger samples. The skew results from the fact that the worst-case assumption of theorem 7.3 does not always hold in practice; if it does not hold, the sample size is larger.

In figures 7.5b, 7.6b and 7.7b, we give the corresponding results for the SEARCH dataset. Since the items in the SEARCH dataset require less space than the NETWORK items, a larger value of $M = 1,170$ was chosen. As can be seen in the figures, the sample size distribution is much tighter because the arrival rate in the dataset does not vary as rapidly. The sample size ranges from 0 items to $1,170$ items, where a value of 0 has only been observed when the window was actually empty. The samples size averages to 579 items and is larger than 447 items in 95% of the cases.

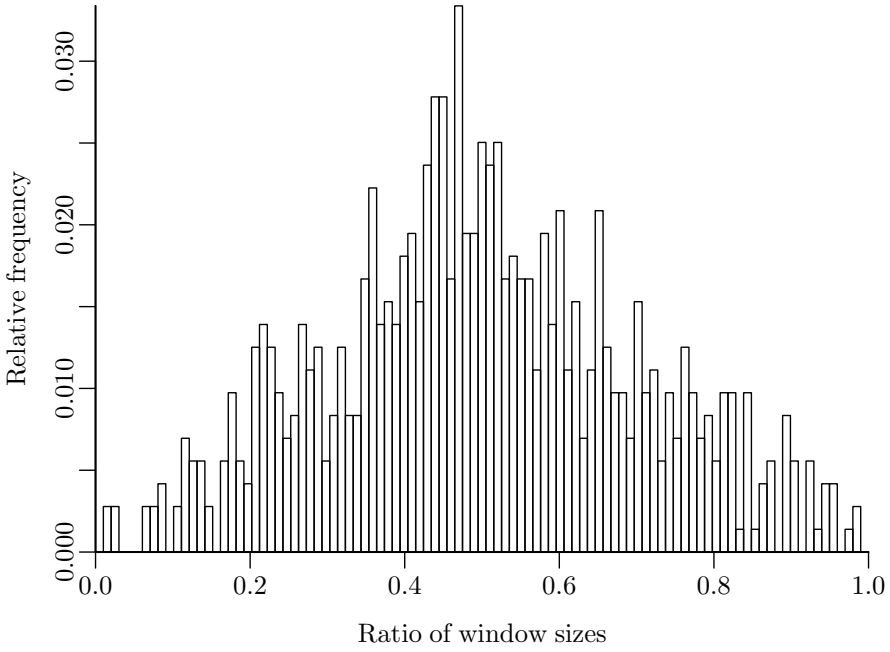


(a) NETWORK data

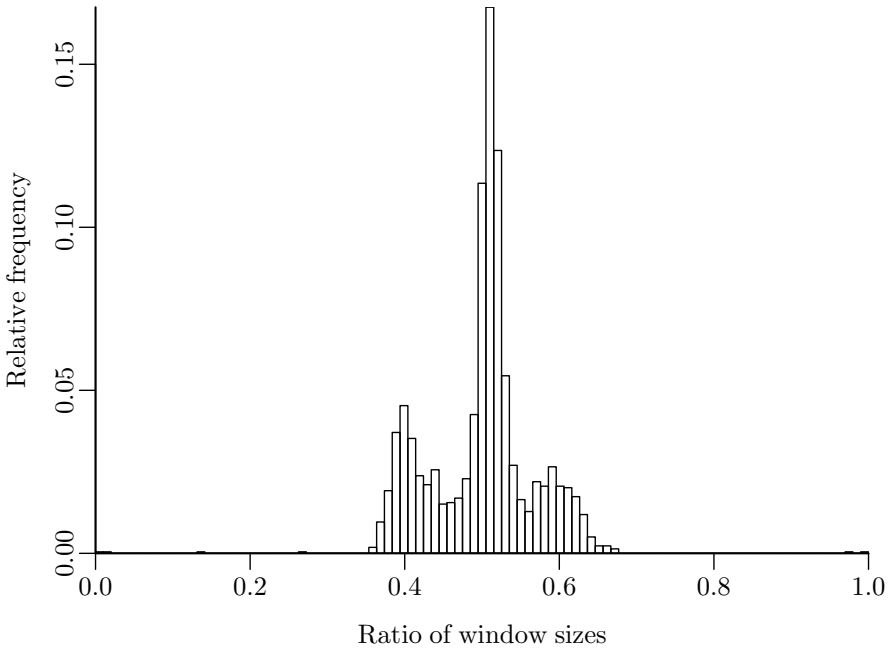


(b) SEARCH data

Figure 7.5: Distribution of sample size (real data)

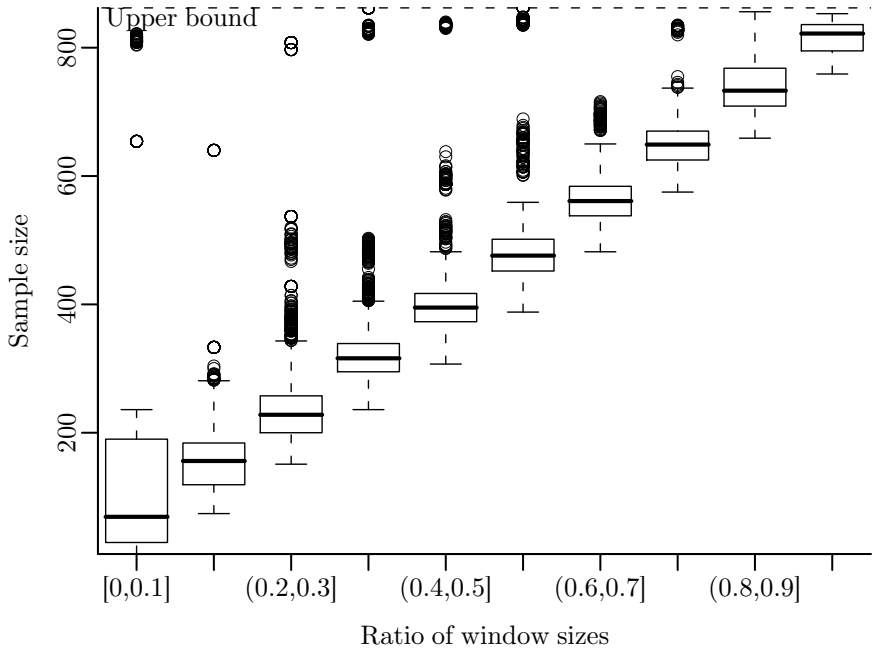


(a) NETWORK data

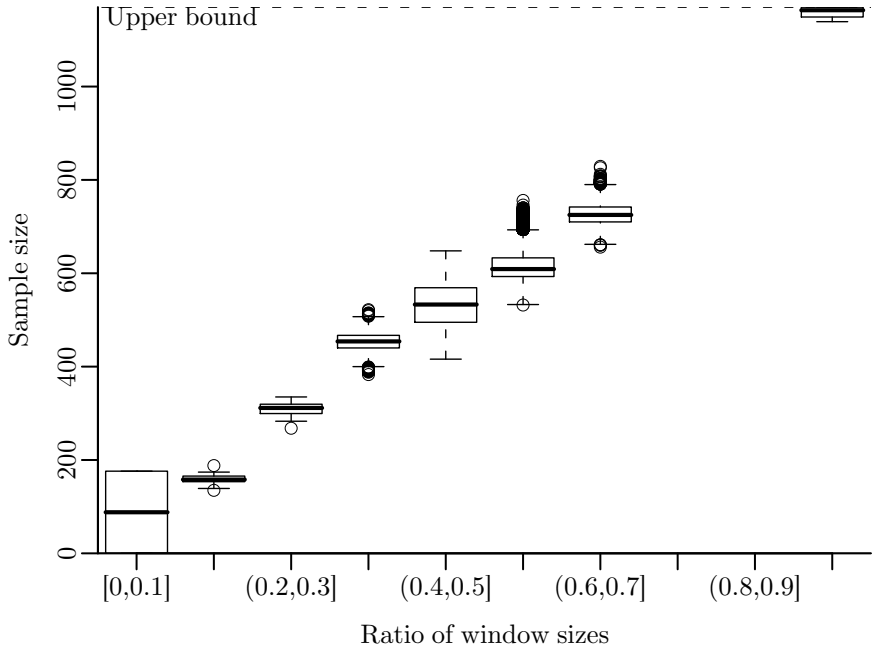


(b) SEARCH data

Figure 7.6: Distribution of window-size ratio (real data)



(a) NETWORK data



(b) SEARCH data

Figure 7.7: Sample size distribution and window size ratio (real data)

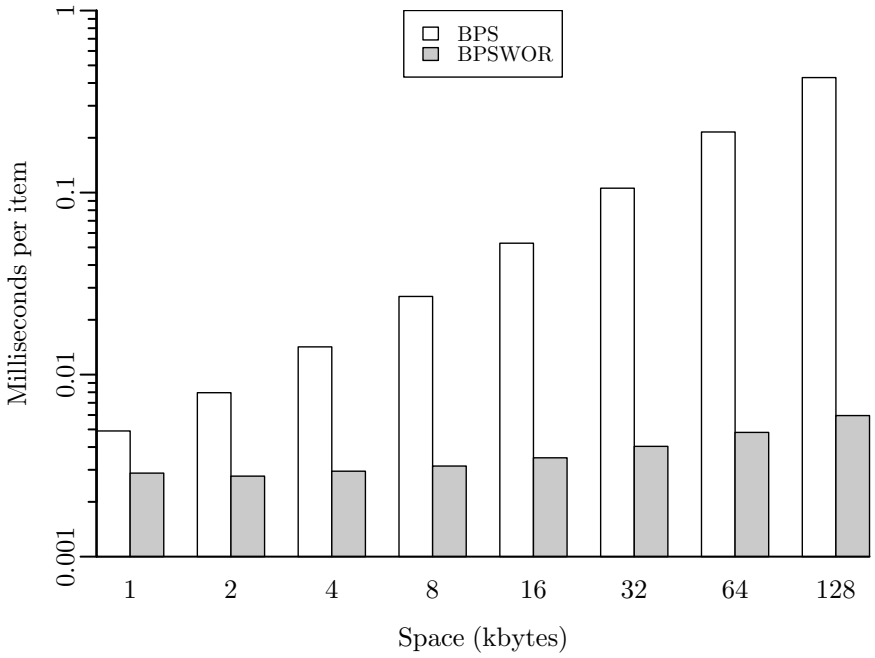
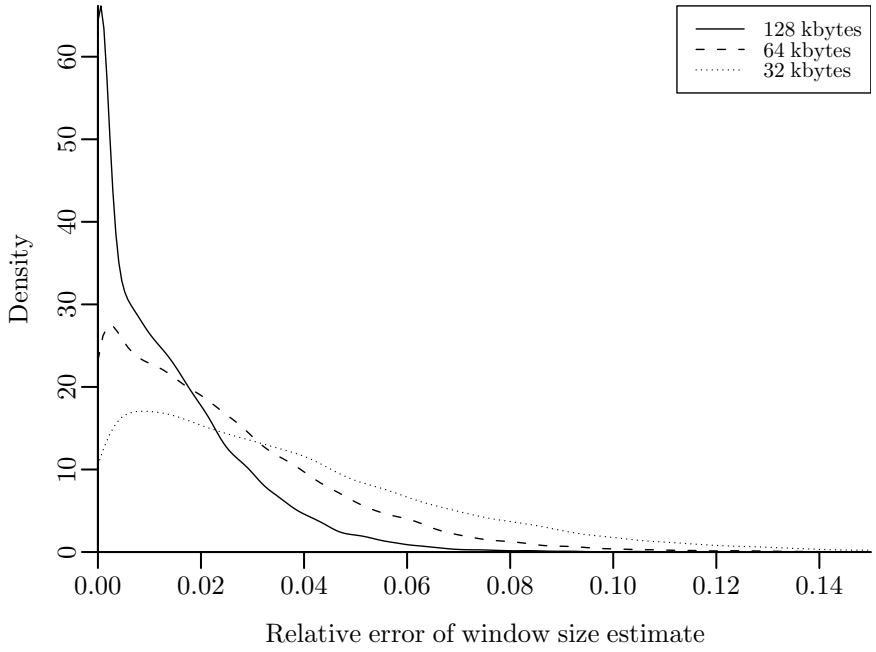


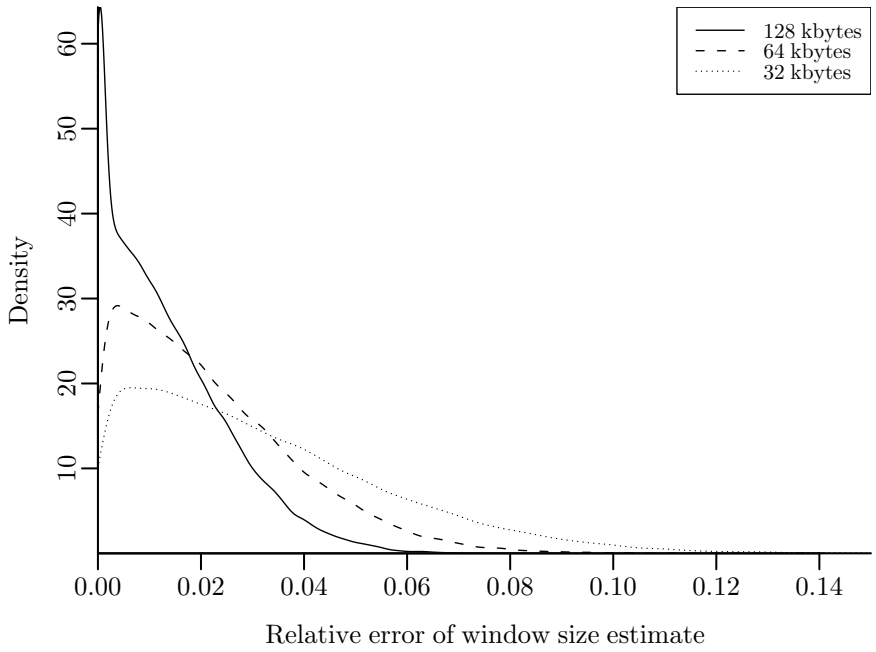
Figure 7.8: Execution time (NETWORK data)

Performance. In figure 7.8, we compare the performance of BPS and BPSWOR for various space budgets on the NETWORK dataset. The figure shows the average time in milliseconds required to process a single item. It has logarithmic axes. For both algorithms, the per-item processing time increases with an increasing space budget, but BPSWOR is significantly more efficient than BPS. The results verify the theoretical analysis in section 7.1.3E. Since BPSWOR additionally samples without replacement, it is clearly superior to BPS.

Estimation of window size. In a final experiment, we evaluated the accuracy and precision of the window-size estimator given in section 7.1.4 in terms of its relative error; the relative error of an estimate $\hat{N}$ of N is defined as $|\hat{N} - N|/N$. Figure 7.9 displays the distribution of the relative error for the NETWORK and SEARCH dataset, respectively, in a kernel-density plot. The relative error is given for memory budgets of 32 kbytes, 64 kbytes and 128 kbytes for the entire sample; although only the priorities are used for window-size estimation. For both datasets and all sample sizes, the relative error almost always lies below 10% and often is much lower. As the memory budget and thus the value of M increases, the estimation error decreases; see section 6.2.2 for a detailed discussion of this behavior. We conclude that our window size estimator produces low-error estimates and can be used when specialized synopses for window-size estimation are unavailable.



(a) NETWORK data



(b) SEARCH data

Figure 7.9: Estimation of window size (real data)

7.2 Stratified Sampling

We now consider the problem of maintaining a stratified sample of a time-based sliding window. The general idea is to partition the window into disjoint strata and to maintain a uniform sample of each stratum (Haas 2009). Stratified sampling is often superior to uniform sampling because a stratified scheme exploits correlations between time and the quantity of interest. As will become evident later on, stratification also allows us to maintain larger samples than BPSWOR in the same space. The main drawback of stratified sampling is its limited applicability; for some problems, it is difficult or even impossible to compute a global solution from the different subsamples. For example, it is not known how the number of distinct items (of an attribute) can be estimated from a stratified sample, while the problem has been studied extensively for uniform samples (see section 2.3.2). If, however, the desired analytical tasks can be performed on a stratified sample, stratification is often the method of choice.

We consider stratified sampling schemes, which partition the window into $H > 1$ strata and maintain a uniform sample S_h of each stratum, $1 \leq h \leq H$. Each sample has a fixed size of M items so that the total sample size is HM. In addition to the sample, we also store the stratum size N_h and the timestamp t_h of the upper stratum boundary; these two quantities are required for sample maintenance. The main challenge in stratified sampling is the placement of stratum boundaries because they have a significant impact on the quality of the sample.[7] In the simplest version, the stream is divided into strata of equal width (time intervals); we refer to this strategy as *equi-width stratification*. An alternative strategy is *equi-depth stratification*, where the window is partitioned into strata of equal size (number of items). As shown below, equi-depth stratification outperforms equi-width stratification when the arrival rate of the data stream varies inside a window, but the strata are more difficult to maintain. In fact, perfect equi-depth stratification is impossible to maintain so that approximate solutions are needed. In this section, we develop a *merge-based stratification* strategy that approximates equi-depth stratification to the best possible extent.

Figure 7.10 illustrates equi-width stratification with parameters $H = 4$ and $M = 1$; sampled items are represented by solid black circles. The figure displays a snapshot of the sample at 3 different points in time, which are arranged vertically and termed (a), (b) and (c). Note that the rightmost stratum ends at the right window boundary and grows as new items arrive, while the leftmost stratum exceeds the window and may contain expired items. The maintenance of the stratified sample is significantly simpler than the maintenance of a uniform sample because arrivals and expirations are not intermixed within strata. Arriving items are added to the rightmost stratum and—since no expirations can occur—we can use RS(M) to maintain the sample incrementally. On the contrary, expirations only affect the

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

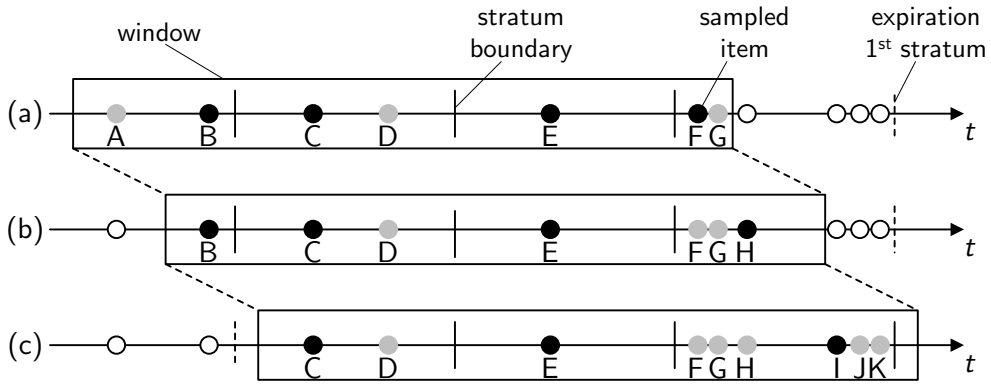


Figure 7.10: Illustration of equi-width stratification

leftmost stratum. We remove expired items from the respective sample; the remaining sample still represents a uniform sample of the non-expired part of the stratum (as discussed in section 3.5.1D).

7.2.1 Effect of Stratum Sizes

The main advantage of equi-width stratification is its simplicity, the main disadvantage is that the sampling fraction may vary widely across the strata. In the example of figure 7.10c, the sampling fractions of the first, second and third stratum are given by 50%, 100% and 16%, respectively. In general, dense regions of the stream are underrepresented by an equi-width sample, while sparse regions are overrepresented. Thus, we want to stratify the data stream in such a way that each stratum has approximately the same size and therefore the same sampling fraction. Before we discuss equi-depth stratification and our approximate algorithms, we investigate the relationship of stratum sizes and accuracy with the help of a simple example.

Suppose that we want to estimate the window average μ of some attribute of the stream from a stratified sample and assume for simplicity that the respective attribute is normally distributed with mean μ and variance σ^2. Further suppose that at some time the window contains N items and is divided into H strata of sizes $N_1, \ldots, N_H$ with $\sum N_h = N$. Then, the standard Horvitz-Thompson estimator $\hat{\mu}$ of μ is a weighted average of the per-stratum sample averages (see table 2.3), that is

$$\hat{\mu} = \frac{1}{N} \sum_{h=1}^{H} N_h \hat{\mu}_h,$$

where $\hat{\mu}_h$ is the sample average of the h-th stratum. The estimator has variance

$$\mathrm{Var}[\,\hat{\mu}\,] = \frac{1}{N^2} \sum_{h=1}^{H} N_h^2 \,\mathrm{Var}[\,\hat{\mu}_h\,] = \frac{\sigma^2}{MN^2} \sum_{h=1}^{H} N_h^2,$$

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

$$\text{Var}\,[\,N_1, \ldots, N_H\,] = \sum_{h=1}^{H} \left(N_h - \frac{N}{H}\right)^2 = \frac{\sum N_h^2}{H} - \left(\frac{N}{H}\right)^2 \tag{7.3}$$

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

The above example is extremely simplified because we designed the stream in such a way that the variance $\text{Var}[\,\hat{\mu}_h\,]$ of the estimate is equal in all strata. In general, stratification is the more efficient the higher the correlation of the attribute of interest with time gets (because time is the stratification variable). In our discussion, however, we assume that no information about the intended use of the sample is available; in this case, our best guess is to assume equal variance in each stratum. Thus, the variance of the stratum sizes as given in (7.3) can be used to quantify the quality of a given stratification.

7.2.2 Merge-Based Stratification

Perfect equi-depth stratification is impossible, since we cannot reposition stratum boundaries arbitrarily. To see this, consider the state of the sample as given in figure 7.10c. To achieve equi-depth stratification, we would have to (1) remove the stratum boundary between items D and E, and (2) introduce a new stratum boundary between H and I. Here, (1) represents a *merge* of the first and second stratum. A sample of the merged stratum can be computed from the samples of the individual strata using one of the algorithms in section 4.3; preferably the MERGE algorithm of Brown and Haas (2006). In the example, the merged sample would contain item C with probability 2/3 and item E with probability 1/3. In contrast, (2) represents a *split* of the third stratum into two new strata, one containing items F-H and one containing items I-K. In the case of a split, it is neither possible to compute the samples of the two new strata nor to determine the stratum sizes. In the example, prior to the split, the third stratum has size 6 and the sample contains item I. Based on this information, it is impossible to come up with a sample of stratum F-H; we cannot even determine that stratum F-H contains 3 items.

A. Algorithmic Description

Our merge-based stratified sampling scheme (MBS) approximates equi-depth stratification to the extent possible. The main idea is to merge two adjacent strata from time to time. Such a merge *reduces the information* stored about the two strata but *creates free space* at the end of the sample, which can be used for future items. In figure 7.11, we illustrate MBS on the example data stream. We start as before with the 4 strata given in (a). Right after the arrival of item H, we merge stratum C-D with stratum E to obtain stratum C-E. The decision of *when* and *which* strata to

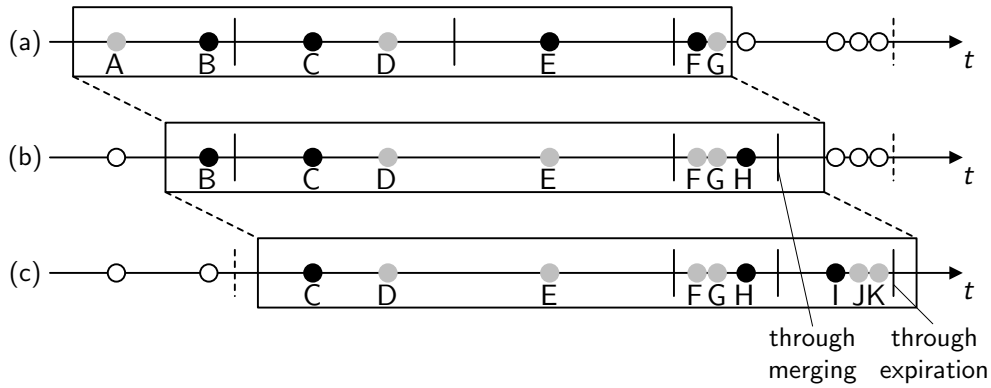


Figure 7.11: Illustration of merge-based stratification

merge is the major challenge in designing the algorithm. After a merge, we use the freed space to start a new, initially empty stratum. The state of the sample after the creation of the new stratum is shown in (b). Subsequent arrivals are added to the new stratum (items I, J and K). Finally, stratum A-B expires and, again, a fresh stratum is created; see (c). Note that the sample is much more balanced than with equi-width stratification (figure 7.10).

Before we discuss when to merge, we briefly describe how to merge. Suppose that we want to merge two adjacent strata R_h and R_{h+1} and $1 < h < H$. Denote by S_h, N_h, t_h the uniform sample (of size M), the stratum size and the upper boundary of stratum R_h. Then, the merged stratum $R_h \cup R_{h+1}$ has size $N_h + N_{h+1}$ and upper boundary t_{h+1}. The sample S is computed from S_h and S_{h+1} using the MERGE algorithm of Brown and Haas (2006) as discussed in section 4.3.1.

B. When To Merge Which Strata

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

Denote by $R^+ = \{ e_1, \ldots, e_{N^+} \}$ the set of items that arrive until the expiration of stratum R_1 (but have not yet arrived) and set $N^+ = |R^+|$.[8] At the time of R_1's expiration and before the creation of the new stratum, the window is divided into $H - 1$ strata so that there are $H - 2$ inner stratum boundaries. The positions of the

[8] In practice, N^+ is not known in advance; we address this issue in section C.

stratum boundaries depend on both the number and point in time of any merges we perform. Our algorithm rests on the observation that for any way of putting $l-2$ stratum boundaries in the sequence

$$R_2, R_3, \ldots, R_H, e_1, e_2, \ldots, e_{N^+},$$

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

$$R_2 \mid R_3 \mid \cdots \mid R_H, e_1, \ldots, e_{N^+}$$

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

$$R_2 \mid \cdots \mid R_h, R_{h+1} \mid \cdots \mid R_H, e_1, \ldots, e_i \mid e_{i+1}, \ldots, e_N+$$

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

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

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

into $H-1$ consecutive and non-empty partitions so that the variance (or sum of squares) of the intra-partition sums is minimized. The problem can be solved using dynamic programming in $O\big(H(H+N^+)^2\big)$ time (Jagadish et al. 1998). In our specific instance of the problem, however, the last N^+ values of the sequence of integers are all equal to 1. As shown below, we can leverage this fact to construct a dynamic programming algorithm that obtains an optimum solution in only $O\big(H^3\big)$ time. Since N^+ is typically large, the improvement in performance can be significant.

The algorithm is as follows. Let $\text{opt}(k,h)$ be the minimum sum of squares when $k-1$ of the $H-2$ boundaries are placed between $N_2, \ldots, N_H$ and one boundary is placed right after N_h; $0 \leq k \leq H-2$ and $k < h < H$. Then, $\text{opt}(k,h)$ can be decomposed into two functions

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

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

$$N_{h+1,H}, 1, \ldots, 1,$$

where $N_{i,j} = \sum_{h=i}^{j} N_h$.[9] In fact, g is minimized if all the $H-k-1$ partitions have the same size

$$\frac{N_{h+1,H} + N^+}{H-k-1}.$$

If $N_{h+1,H}$ is larger than this average size, the minimum value of g cannot be obtained. In this case, the best choice is to put $N_{h+1,H}$ in one stratum for its own; the remaining $H-k-2$ partitions then all have size

$$\frac{N^+}{H-k-2}.$$

Thus, the function g is given by

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

The function f can be defined recursively with

$$\begin{aligned} f(0,h) &= N_{2,h}^2 \\ f(k,h) &= \min_{k \le j < h} \left\{ f(k-1,j) + N_{j+1,h}^2 \right\}. \end{aligned}$$

and the optimum solution is given by

$$\min_{0 \le k \le H-2} \min_{k < h < H} \mathrm{opt}(k,h).$$

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

C. Estimation of Arriving Item Count

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

As before, suppose that—at some time t—the sample consists of H strata of sizes $N_1, \ldots, N_H$ and denote by t_h the upper boundary of the h-th stratum, $1 \le h \le H$. Furthermore, denote by $\Delta^- = t_1 + \Delta - t$ the time span until the expiration of the first stratum. We want to predict the number of items that arrive until time $t + \Delta^-$.

[9] $N_{i,j}$ can be computed in constant time with the help of an array containing the prefix sums $N_{2,2}, \ldots, N_{2,H}$ (Jagadish et al. 1998).

Denote by j the stratum index such that $t - t_j > \Delta^-$ and $t - t_{j+1} \leq \Delta^-$. An estimate $\hat{N}^+$ of N^+ is then given by

$$\hat{N}^+ = \Delta^- \frac{\sum_{h=j+1}^{H} N_h}{t - t_j}.$$

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

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

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

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

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

7.2.3 Experiments

We run a set of experiments to evaluate whether merge-based stratification is superior to equi-width stratification in practical scenarios. In summary, we found that:

- Merge-based stratification leads to significantly lower stratum size variances than equi-width stratification when the data stream is bursty. Both schemes have comparable performance when the data stream rate changes slowly.

- Merge-based stratification seems to be robust to errors in the arrival rate estimate. Results with estimated arrival rates are close to the theoretical optimum.
- When the number of strata is not too large ($H \leq 32$), the overhead of merge-based stratification is low.

We used the same setup as in section 7.1.6; specifically, we used the NETWORK and SEARCH datasets. For most of our experiments, we report the variance of the stratum sizes, which is the key characteristic for stratified sampling over data streams. The variance is a direct measure of how close stratification is to optimal equi-depth stratification. A smaller variance typically results in less estimation error; this behavior was also be visible in our experiments.

Recall that during the sampling process, MBS occasionally requires an estimate of the number of items that arrive until the expiration of the first stratum. To quantify the impact of estimation, we considered two versions of MBS in our experiments. MBS-N makes use of an "oracle": Whenever an estimate of the number of arriving items is required, we determine the exact number directly from the dataset so that no estimation error occurs. MBS-N can therefore be seen as the theoretical optimum of merge-based stratification. In contrast, MBS-$\hat{N}$ uses the estimation technique and robustness modifications as described in section 7.2.2C. The experimental setup is identical to the one used for uniform sampling, that is, we sample from the real-world datasets over a sliding window of 1 hour length. Unless stated otherwise, we used a space budget of 32 kbytes and $H = 32$ strata.

Variance of stratum sizes. We first compared the variance of the stratum sizes achieved by the three different stratification schemes. In order to facilitate a meaningful variance comparison for windows of varying size, we report the coefficient of variation (CV) instead of the stratum-size variance directly. The CV is defined as the standard deviation (square root of variance) normalized by the mean stratum size:

$$\mathrm{CV}\,[\,N_1,\ldots,N_H\,] = \frac{\sqrt{\mathrm{Var}\,[\,N_1,\ldots,N_H\,]}}{N/H}.$$

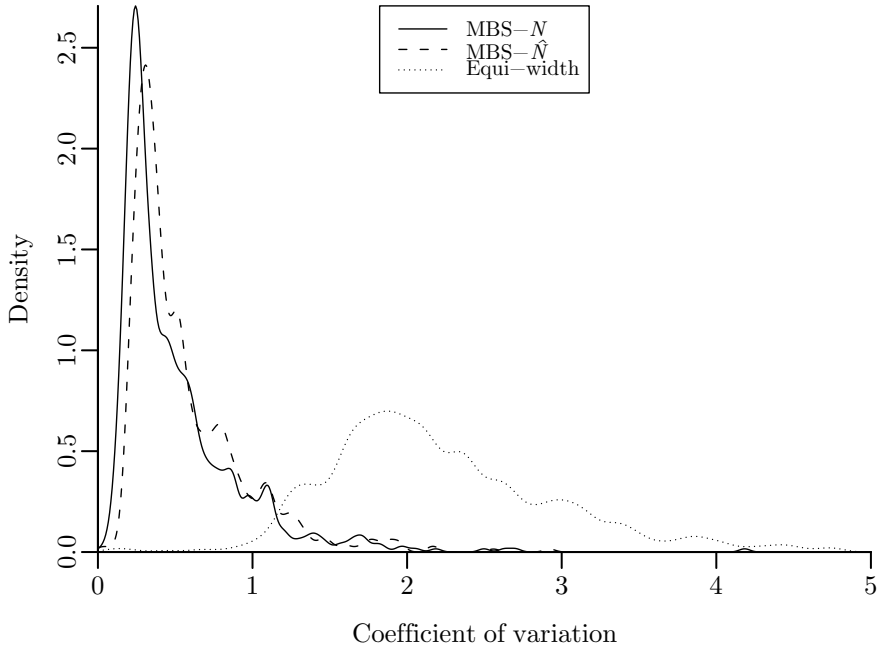
A value less than 1 indicates a low-variance distribution, whereas a value larger than 1 is often considered high variance. Figure 7.12a displays the distribution of the CV for the NETWORK dataset using a kernel-density plot. As can be seen, equi-width stratification leads to high values of the CV, while merge-based stratification produces significantly better results. Also, MBS-N and MBS-$\hat{\text{N}}$ perform similarly, with MBS-N being slightly superior. The difference between equi-width stratification and the MBS schemes is contributed to the burstiness of the NETWORK stream in which the arrival rates vary significantly during a window length. In contrast, Figure 7.12b shows the distribution of the CV for the SEARCH dataset. Since the arrival rates change only slowly, equi-width stratification already produces very good results and the merge-based schemes essentially never decide to merge two adjacent strata. The

three schemes produce almost identical results. Therefore, merge-based stratification is the more beneficial the more bursty the data stream is.

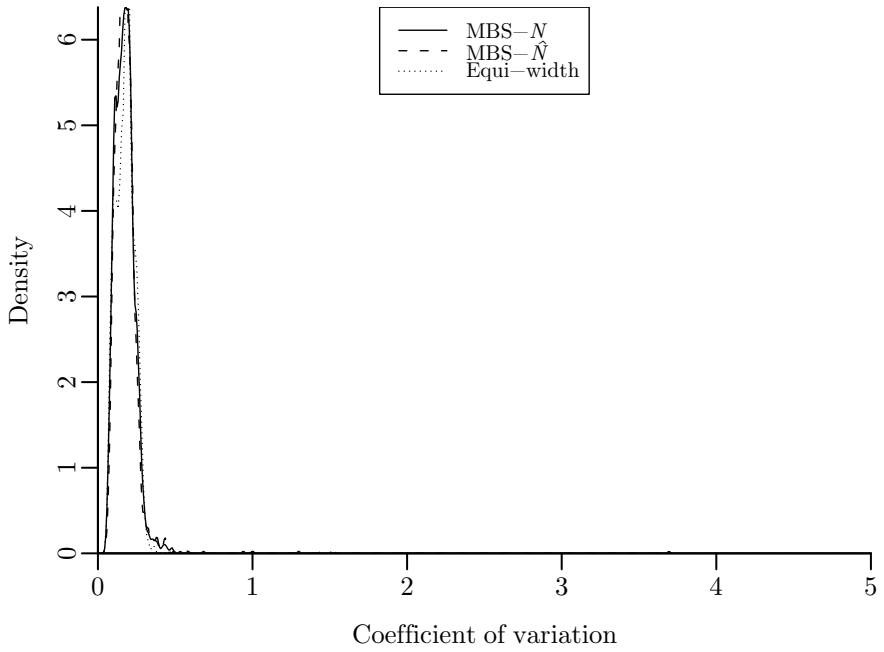
Accuracy of estimate (example). In a next experiment, we used the stratified sampling schemes to estimate the throughput of the NETWORK data from the sample. Here, we defined the throughput as the sum of the user-data size attribute over the entire window (see the description of the NETWORK dataset). Figure 7.13 gives the distribution of the relative error of the estimate. The estimates derived from the merge-based schemes have a significantly lower estimation error than the estimates achieved with equi-width stratification. Thus, intelligent stratification indeed improves the quality of the sample. Note that for the SEARCH dataset, the distribution of the relative error would be almost indistinguishable for the three schemes because for this dataset, merge-based stratification does not improve upon equi-width stratification.

Number of strata (Example). The number H of strata can have a significant influence on the quality of the estimates. In table 7.1, we give the average relative error (ARE) of the NETWORK throughput estimate for a varying number of strata. With an increasing number of strata, the ARE increases for equi-width stratification but decreases for the merge-based schemes. On the one hand, the sample size per stratum decreases as H increases and it becomes more and more important to distribute the strata evenly across the window. In fact, when the number of strata was high, equi-width stratification frequently produced empty strata and thereby wasted some of the available space. On the other hand, a large number of strata better exploits the correlations between time and the attribute of interest. Thus, the estimation error often decreases with an increasing value of H. In our experiment, the correlation of the user-data size attribute and time is low, so that the decrease in estimation error is also relatively low.

Performance. In a final experiment, we measured the average per-item processing time for the three schemes and a varying number of strata. The results for the NETWORK data are given in table 7.1. Clearly, equi-width stratification is the most efficient technique and the processing time does not depend upon the number of strata. The MBS schemes are slower because they occasionally have to (1) estimate the number of arriving items, (2) determine the optimum stratification and (3) merge adjacent strata. The computational effort increases as the number of strata increases. MBS-N is slightly faster than MBS-$\hat{N}$ because MBS-$\hat{N}$ reevaluates (2) if the stream behaves differently than predicted. In comparison to equi-width stratification, MBS leads to a significant performance overhead if the number of strata is large. However, when the number of strata is not too large ($H \leq 32$), the overhead is low but the quality of the resulting stratification might increase significantly.



(a) NETWORK data



(b) SEARCH data

Figure 7.12: Coefficient of variation of stratum sizes

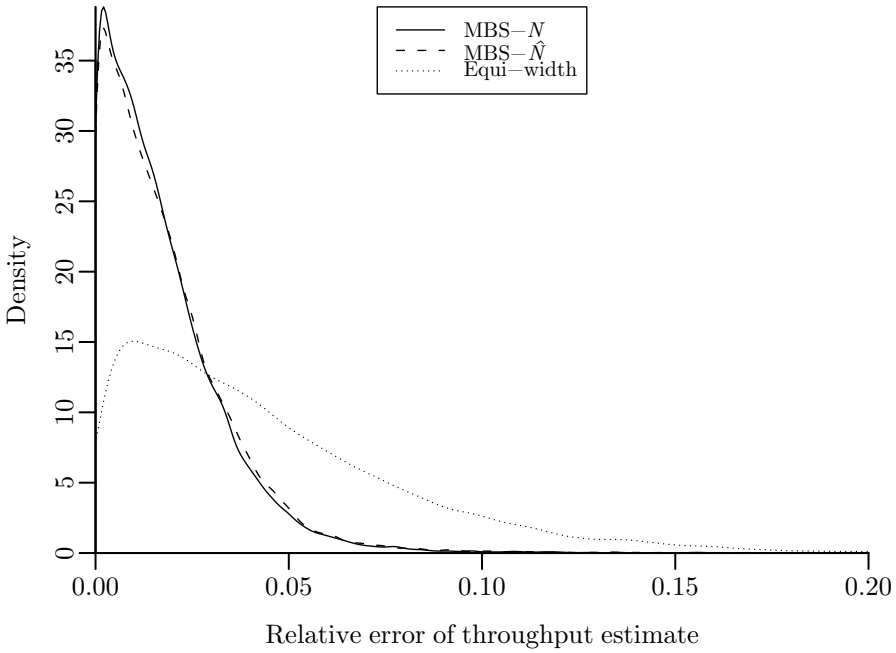


Figure 7.13: NETWORK dataset, throughput estimation

Table 7.1: Influence of the number of strata (NETWORK dataset)

		4	8	16	32	64
ARE	Equi-width	2.31%	2.73%	3.44%	4.42%	5.90%
	MBS-N	2.00%	1.83%	1.74%	1.70%	1.72%
	MBS-$\hat{\text{N}}$	2.04%	1.88%	1.82%	1.76%	1.79%
Time (μs)	Equi-width	2.27	2.25	2.24	2.22	2.21
	MBS-N	2.33	2.36	2.41	3.17	9.68
	MBS-$\hat{\text{N}}$	2.35	2.38	2.67	4.75	18.44

7.3 Summary

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

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

Chapter 8

Conclusion

Due to its wide range of applications, random sampling has been and continues to be an important research area in data management. A quick look at the bibliography of this thesis reveals that roughly 80 sampling-related papers appeared in the last decade, counting only major database conferences and journals, and roughly 35 of them appeared in the last 4 years. Many of the sampling techniques proposed in these papers have been developed for static datasets, in which a sample once computed remains valid for all times. In practice, however, datasets evolve and any changes to the data have to be appropriately reflected in the sample to maintain its statistical validity. In this thesis, we addressed this problem and provided efficient methods for sample maintenance. Our algorithms can potentially be leveraged to extend the applicability of many previous techniques to the class of evolving datasets.

More specifically, we considered the problem of maintaining a uniform random sample of an evolving dataset; such samples are a building block of more sophisticated techniques. We proposed novel maintenance algorithms including *random pairing* (for sets), *augmented Bernoulli sampling* (for multisets), *augmented min-hash sampling* (for distinct items), *bounded priority sampling* (for data streams), and *merge-based stratified sampling* (also for data streams). The key property of all our algorithms is that they are "incremental", that is, they completely avoid any expensive accesses to the underlying dataset. Instead, sampling decisions are based solely on the stream of insertion, update, and deletion transactions applied to the data. In addition to our maintenance schemes, we discussed the related problems of how to resize a random sample and how to combine several samples into a single one. We proposed novel estimators for counts, averages, sums, ratios, and the number of distinct items. Our estimators exploit the maintenance information stored along with the samples and they have lower estimation errors than known estimators, which do not exploit this information.

Future Work

There are a lot of open problems in this specific area of database sampling; we list some of them that we consider interesting to look at.

Sampling algorithms. Previous algorithms—together with the algorithms in this thesis—already cover a large fraction of the problem space for uniform sample

maintenance. However, the following problems have not yet been addressed in the literature:

- We presented bounded sampling algorithms for set sampling, distinct-item sampling, and data stream sampling. Is there any corresponding algorithm for multiset sampling?
- All known distinct-item sampling schemes assume at least min-wise independent hash functions.[1] Is there any maintenance scheme that works with pairwise independent hash functions?
- The existing bounded distinct-item schemes either support only a "small" number of deletions or have large space overhead. Is it possible to find a compromise?
- We discussed a special case of sampling with "implicit deletions" that occurs with time-based windows over a data stream. Are there any algorithms for the more general setting as outlined in section 3.3.3C?
- We considered uniform sampling. What about weighted sampling?[2]

Sampling costs. We used simple cost models to assess the cost of sample maintenance, and we generally assumed that the cost of sample maintenance is low. Both issues may deserve further exploration:

- Our algorithms were designed under the assumption that samples are stored in main memory so that random accesses are cheap. Can the algorithms be modified so as to efficiently maintain large samples stored on hard disks or flash drives?[3]
- We assumed that the dataset and its sample are stored at the same location so that we can access the stream of transactions at virtually no cost. How can maintenance techniques be integrated into distributed systems, where this assumption might not hold?
- How does the cost of sample maintenance relate to the cost of maintaining the underlying dataset? In other words, what is the overhead of maintaining a random sample?

[1] In practice, min-wise independence can be approximated using k-wise independent hash functions, $k \gg 2$, see Indyk (1999).

[2] One can modifiy BERN(q) and ABERN(q) to implement Poisson sampling from a set and a multiset, respectively. For bounded-size sampling, the problem is decidedly hard; it has received a lot of attention in the statistical literature.

[3] Efficient techniques for the class of append-only datasets have been proposed by Jermaine et al. (2004), Pol et al. (2008), Gemulla et al. (2006), and Nath and Gibbons (2008). All these approaches implement a deferred refresh policy.

- Materialized sampling is clearly preferable at query time but it might not be a good option when the sample is accessed rarely but the dataset is updated frequently. Can we be more specific about the relative cost of these two approaches?

Sampling infrastructure. It is certainly a non-trivial issue to implement random sampling techniques in an actual system. Here are some challenges that may arise when doing so:

- We treated the "propagate step" and the "sample step" for incremental sample maintenance separately. What would a combined approach look like? What efficiency benefits are possible?
- Our algorithms process each insertion, update, and deletion transaction separately. Can we be more efficient when we are given batches of transactions?
- How does sampling interact with transaction processing in a relational database system? For example, there are situations in which several concurrent writers can operate on different parts of the base data without any conflict. When a materialized sample is defined on the dataset, can we avoid locking the entire sample and thereby blocking one of the writers?

Many more open problems could probably be listed here. As mentioned previously, database sampling is a very active area of research, and it is likely that some of the problems above are going to be addressed in the near future. This thesis can be seen as a step towards efficient algorithms for sample maintenance under general transactions, but it is certainly not the final step.

Bibliography

Acharya, S., P. B. Gibbons, and V. Poosala (2000). Congressional samples for approximate answering of group-by queries. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 487–498.

Acharya, S., P. B. Gibbons, V. Poosala, and S. Ramaswamy (1999). Join synopses for approximate query answering. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 275–286.

Aggarwal, C. C. (Ed.) (2006a). *Data Streams: Models and Algorithms (Advances in Database Systems)*. Springer.

Aggarwal, C. C. (2006b). On biased reservoir sampling in the presence of stream evolution. In *Proc. of the 2006 Intl. Conf. on Very Large Data Bases*, pp. 607–618.

Agrawal, R., T. Imielinski, and A. N. Swami (1993). Mining association rules between sets of items in large databases. In *Proc. of the 1993 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 207–216.

Agrawal, R. and R. Srikant (1994). Fast algorithms for mining association rules. In *Proc. of the 1994 Intl. Conf. on Very Large Data Bases*, pp. 487–499.

Ahrens, J. H. and U. Dieter (1985). Sequential random sampling. *ACM Transactions on Mathematical Software 11*(2), 157–169.

Alon, N., P. B. Gibbons, Y. Matias, and M. Szegedy (1999). Tracking join and self-join sizes in limited storage. In *Proc. of the 1999 ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 10–20.

Alon, N., Y. Matias, and M. Szegedy (1996). The space complexity of approximating the frequency moments. In *Proc. of the 1996 Annual ACM Symp. on Theory of Computing*, pp. 20–29.

Alon, N., Y. Matias, and M. Szegedy (1999). The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences 58*(1), 137–147.

Antoshenkov, G. (1992). Random sampling from pseudo-ranked b+ trees. In *Proc. of the 1992 Intl. Conf. on Very Large Data Bases*, pp. 375–382.

Arasu, A., S. Babu, and J. Widom (2006). The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal 15*(2), 121–142.

Babcock, B., S. Chaudhuri, and G. Das (2003). Dynamic sample selection for approximate query processing. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 539–550.

Babcock, B., M. Datar, and R. Motwani (2002). Sampling from a moving window over streaming data. In *Proc. of the 2002 Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 633–634.

Bar-Yossef, Z., T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan (2002). Counting distinct elements in a data stream. In *Proc. of the 2002 Intl. Workshop on Randomization and Approximation Techniques*, pp. 1–10.

Bebbington, A. C. (1975). A simple method of drawing a sample without replacement. *Applied Statistics 24*(1), 136.

Beyer, K., P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla (2007). On synopses for distinct-value estimation under multiset operations. In *Proc. of the 2007 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 199–210.

Bissell, A. F. (1986). Ordered random selection without replacement. *Applied Statistics 35*(1), 73–75.

Broder, A. Z. (1997). On the resemblance and containment of documents. In *Proc. of the 1997 Compression and Complexity of Sequences*, pp. 21–29.

Broder, A. Z., M. Charikar, A. Frieze, and M. Mitzenmacher (2000). Min-wise independent permutations. *Journal of Computer and System Sciences 60*(3), 630–659.

Brönnimann, H., B. Chen, M. Dash, P. Haas, Y. Qiao, and P. Scheuermann (2004). Efficient data-reduction methods for on-line association rule discovery. In K. S. Hillol Kargupta, Anupam Joshi and Y. Yesha (Eds.), *Data Mining: Next Generation Challenges and Future Directions*. AAAI Press.

Brönnimann, H., B. Chen, M. Dash, P. Haas, and P. Scheuermann (2003). Efficient data reduction with ease. In *Proc. of the 2003 Intl. Conf. on Knowledge Discovery and Data Mining*, pp. 59–68.

Brown, P. and P. J. Haas (2003). Bhunt: Automatic discovery of fuzzy algebraic constraints in relational data. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pp. 668–679.

Brown, P. G. and P. J. Haas (2006). Techniques for warehousing of sample data. In *Proc. of the 2006 Intl. Conf. on Data Engineering*, pp. 6.

Bunge, J. and M. Fitzpatrick (1993). Estimating the number of species: A review. *Journal of the American Statistical Association 88*(421), 364–373.

Cárdenas, A. F. (1975). Analysis and performance of inverted data base structures. *Communications of the ACM 18*(5), 253–263.

Carter, J. L. and M. N. Wegman (1977). Universal classes of hash functions (extended abstract). In *Proc. of the 1977 Annual ACM Symp. on Theory of Computing*, pp. 106–112. ACM Press.

Charikar, M., S. Chaudhuri, R. Motwani, and V. Narasayya (2000). Towards estimation error guarantees for distinct values. In *Proc. of the 2000 ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 268–279.

Chaudhuri, S., G. Das, M. Datar, and R. M. V. R. Narasayya (2001). Overcoming limitations of sampling for aggregation queries. In *Proc. of the 2001 Intl. Conf. on Data Engineering*, pp. 534–544.

Chaudhuri, S., G. Das, and V. Narasayya (2001). A robust, optimization-based approach for approximate answering of aggregate queries. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 295–306.

Chaudhuri, S., G. Das, and V. Narasayya (2007). Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems 32*(2), 9.

Chaudhuri, S., G. Das, and U. Srivastava (2004). Effective use of block-level sampling in statistics estimation. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 287–298.

Chaudhuri, S. and R. Motwani (1999). On sampling and relational operators. *IEEE Data Engineering Bulletin 22*(4), 41–46.

Chaudhuri, S., R. Motwani, and V. Narasayya (1998). Random sampling for histogram construction: how much is enough? In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 436–447.

Chen, B., P. Haas, and P. Scheuermann (2002). A new two-phase sampling based algorithm for discovering association rules. In *Proc. of the 2002 Intl. Conf. on Knowledge Discovery and Data Mining*, pp. 462–468.

Cheung, T.-Y. (1982). Estimating block accesses and number of records in file management. *Communications of the ACM 25*(7), 484–487.

Christodoulakis, S. (1983). Estimating block transfers and join sizes. In *Proc. of the 1983 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 40–54.

Cochran, W. G. (1977). *Sampling Techniques* (3rd ed.). Wiley Series in Probability & Mathematical Statistics. John Wiley & Sons.

Cohen, E. (1997). Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences 55*(3), 441–453.

Colt Library (2004). Open source libraries for high performance scientific and technical computing in Java. http://dsd.lbl.gov/~hoschek/colt/.

Cormode, G., S. Muthukrishnan, and I. Rozenbaum (2005). Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In *Proc. of the 2005 Intl. Conf. on Very Large Data Bases*, pp. 25–36.

Dasu, T., T. Johnson, S. Muthukrishnan, and V. Shkapenyuk (2002). Mining database structure; or, how to build a data quality browser. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 240–251.

Datar, M. and S. Muthukrishnan (2002). Estimating rarity and similarity over data stream windows. In *Proc. of the 2002 Annual European Symp. on Algorithms*, pp. 323–334. Springer-Verlag.

David, H. A. and H. N. Nagaraja (2003, Aug). *Order Statistics* (3rd ed.). Wiley Series in Probability and Statistics. Wiley.

Denning, D. E. (1980). Secure statistical databases with random sample queries. *ACM Transactions on Database Systems 5*(3), 291–315.

Devroye, L. (1986). *Non-Uniform Random Variate Generation*. New York: Springer.

DeWitt, D. J., J. F. Naughton, and D. A. Schneider (1991a). An evaluation of non-equijoin algorithms. In *Proc. of the 1991 Intl. Conf. on Very Large Data Bases*, pp. 443–452.

DeWitt, D. J., J. F. Naughton, and D. A. Schneider (1991b). Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proc. of the 1991 Intl. Conf. Parallel and Distributed Information Systems*, Los Alamitos, CA, USA, pp. 280–291. IEEE Computer Society Press.

DeWitt, D. J., J. F. Naughton, D. A. Schneider, and S. Seshadri (1992). Practical skew handling in parallel joins. In *Proc. of the 1992 Intl. Conf. on Very Large Data Bases*, pp. 27–40.

Didonato, A. R. and A. H. Morris, Jr (1992). Algorithm 708; significant digit computation of the incomplete beta function ratios. *ACM Transactions on Mathematical Software 18*(3), 360–373.

Duffield, N., C. Lund, and M. Thorup (2001). Charging from sampled network usage. In *Proc. of the 2001 ACM SIGCOMM Workshop on Internet Measurement*, New York, NY, USA, pp. 245–256. ACM.

Duffield, N. G. and M. Grossglauser (2001). Trajectory sampling for direct traffic observation. *IEEE/ACM Transactions on Networking 9*(3), 280–292.

Ernvall, J. and O. Nevalainen (1982). An algorithm for unbiased random sampling. *The Computer Journal 25*(1), 45–47.

Estan, C. and J. F. Naughton (2006). End-biased samples for join cardinality estimation. In *Proc. of the 2006 Intl. Conf. on Data Engineering*, pp. 20.

Ester, M., H.-P. Kriegel, J. Sander, and X. Xu (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of the 1996 Intl. Conf. on Knowledge Discovery and Data Mining*, pp. 226–231.

Fan, C. T., M. E. Muller, and I. Rezucha (1962). Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *Journal of the American Statistical Association 57*(298), 387–402.

Fang, M., N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman (1998). Computing iceberg queries efficiently. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pp. 299–310.

Feller, W. (1968). *An Introduction to Probability Theory and Its Applications* (3 ed.). Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons.

Flajolet, P. and G. N. Martin (1985). Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences 31*(2), 182–209.

Frahling, G., P. Indyk, and C. Sohler (2005). Sampling in dynamic data streams and applications. In *Proc. of the 2005 Annual ACM Symp. on Computational Geometry*, pp. 142–149.

Frawley, W. J., G. Piatetsky-Shapiro, and C. J. Matheus (1992). Knowledge discovery in databases: An overview. *AI Magazine 13*, 57–70.

Ganguly, S. (2007). Counting distinct items over update streams. *Theoretical Computer Science 378*(3), 211–222.

Ganguly, S., P. B. Gibbons, Y. Matias, and A. Silberschatz (1996). Bifocal sampling for skew-resistant join size estimation. In *Proc. of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 271–281.

Ganti, V., M.-L. Lee, and R. Ramakrishnan (2000). ICICLES: Self-tuning samples for approximate query answering. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases*, pp. 176–187.

Garofalakis, M., J. Gehrke, and R. Rastogi (Eds.) (2009). *Data Stream Management: Processing High-Speed Data Streams*. Data-Centric Systems and Applications. Springer.

Gemulla, R. and W. Lehner (2006). Deferred maintenance of disk-based random samples. In *Proc. of the 2006 Intl. Conf. on Extending Database Technology*, pp. 423–441.

Gemulla, R. and W. Lehner (2008). Sampling time-based sliding windows in bounded space. In *Proc. of the 2008 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 379–392.

Gemulla, R., W. Lehner, and P. J. Haas (2006). A dip in the reservoir: maintaining sample synopses of evolving datasets. In *Proc. of the 2006 Intl. Conf. on Very Large Data Bases*, pp. 595–606.

Gemulla, R., W. Lehner, and P. J. Haas (2007). Maintaining bernoulli samples over evolving multisets. In *Proc. of the 2007 ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 93–102.

Gemulla, R., W. Lehner, and P. J. Haas (2008). Maintaining bounded-size sample synopses of evolving datasets. *The VLDB Journal 17*(2), 173–201.

Gemulla, R., P. Rösch, and W. Lehner (2008). Linked bernoulli synopses: Sampling along foreign keys. In *Proc. of the 2008 Intl. Conf. on Statistical and Scientific Database Management*, pp. 6–23.

Gibbons, P. B. (2001). Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *The VLDB Journal*, pp. 541–550.

Gibbons, P. B. (2009). Distinct-values estimation over data streams. In M. Garofalakis, J. Gehrke, and R. Rastogi (Eds.), *Data Stream Management: Processing High Speed Data Streams*. Springer.

Gibbons, P. B. and Y. Matias (1998). New sampling-based summary statistics for improving approximate query answers. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 331–342.

Gibbons, P. B. and Y. Matias (1999). Synopsis data structures for massive data sets. In *Proc. of the 1999 Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 909–910.

Gibbons, P. B., Y. Matias, and V. Poosala (1997). Fast incremental maintenance of approximate histograms. In *Proc. of the 1997 Intl. Conf. on Very Large Data Bases*, pp. 466–475.

Gibbons, P. B. and S. Tirthapura (2001). Estimating simple functions on the union of data streams. In *Proc. of the 2001 Annual ACM Symp. on Parallel Algorithms and Architectures*, pp. 281–291.

Gionis, A., H. Mannila, and P. Tsaparas (2007). Clustering aggregation. *ACM Transactions on Knowledge Discovery from Data 1*(1), 4.

GNU Scientific Library (2008). GNU Scientific Library. http://www.gnu.org/software/gsl/.

Golab, L. and M. T. Özsu (2003). Issues in data stream management. *SIGMOD Record 32*(2), 5–14.

Goodman, S. E. and S. T. Hedetniemi (1977). *Introduction to the Design and Analysis of Algorithms*. New York, NY, USA: McGraw-Hill, Inc.

Gryz, J., J. Guo, L. Liu, and C. Zuzarte (2004). Query sampling in DB2 Universal Database. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 839–843.

Guha, S., R. Rastogi, and K. Shim (1998). CURE: An efficient clustering algorithm for large databases. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 73–84.

Gupta, A. and I. S. Mumick (Eds.) (1999). *Materialized Views: Techniques, Implementations, and Applications*. Cambridge, MA, USA: The MIT Press.

Haas, P. J. (1997). Large-sample and deterministic confidence intervals for online aggregation. In *Proc. of the 1997 Intl. Conf. on Statistical and Scientific Database Management*, pp. 51–63.

Haas, P. J. (1999). Hoeffding inequalities for join-selectivity estimation and online aggregation. Technical Report RJ 10040, IBM Almaden Research Center. (Revised version).

Haas, P. J. (2003). Speeding up DB2 UDB using sampling. Technical report, IBM Almaden Research Center. http://www.almaden.ibm.com/cs/people/peterh/idugjbig.pdf.

Haas, P. J. (2009). Data stream sampling: Basic techniques and results. In M. Garofalakis, J. Gehrke, and R. Rastogi (Eds.), *Data Stream Management: Processing High Speed Data Streams*. Springer.

Haas, P. J. and J. M. Hellerstein (1998). Join algorithms for online aggregation. Technical Report RJ 10126, IBM Almaden Research Center.

Haas, P. J. and J. M. Hellerstein (1999). Ripple joins for online aggregation. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 287–298.

Haas, P. J. and C. König (2004). A bi-level bernoulli scheme for database sampling. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 275–286.

Haas, P. J., Y. Liu, and L. Stokes (2006). An estimator of the number of species from quadrat sampling. *Biometrics 62*, 135–141.

Haas, P. J., J. F. Naughton, S. Seshadri, and L. Stokes (1995). Sampling-based estimation of the number of distinct values of an attribute. In *Proc. of the 1995 Intl. Conf. on Very Large Data Bases*, pp. 311–322.

Haas, P. J., J. F. Naughton, S. Seshadri, and A. N. Swami (1993). Fixed-precision estimation of join selectivity. In *Proc. of the 1993 ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 190–201.

Haas, P. J., J. F. Naughton, S. Seshadri, and A. N. Swami (1996). Selectivity and cost estimation for joins based on random sampling. *Journal of Computer and System Sciences 52*(3), 550–569.

Haas, P. J., J. F. Naughton, and A. N. Swami (1994). On the relative cost of sampling for join selectivity estimation. In *Proc. of the 1994 ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 14–24.

Haas, P. J. and L. Stokes (1998). Estimating the number of classes in a finite population. *Journal of the American Statistical Association 93*(444), 1475–1487. www.almaden.ibm.com/cs/people/peterh/jasa3rj.pdf.

Haas, P. J. and A. N. Swami (1992). Sequential sampling procedures for query size estimation. In *Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 341–350.

Haas, P. J. and A. N. Swami (1995). Sampling-based selectivity estimation for joins using augmented frequent value statistics. In *Proc. of the 1995 Intl. Conf. on Data Engineering*, pp. 522–531.

Hadjieleftheriou, M., X. Yu, N. Koudas, and D. Srivastava (2008). Selectivity estimation of set similarity selection queries. In *Proc. of the 2008 Intl. Conf. on Very Large Data Bases*. (to appear).

Hanif, M. and K. Brewer (1980). Sampling with unequal probabilities without replacement: A review. *International Statistical Review 48*, 317–335.

Hansen, M. H. and W. N. Hurwitz (1943). On the theory of sampling from finite populations. *Annals of Mathematical Statistics 14*(4), 333–362.

Hellekalek, P. and S. Wegenkittl (2003). Empirical evidence concerning aes. *ACM Transactions on Modeling and Computer Simulation 13*(4), 322–333.

Hellerstein, J. M., P. J. Haas, and H. J. Wang (1997). Online aggregation. In *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 171–182.

Horvitz, D. and D. Thompson (1952). A generalization of sampling without replacement from a finite universe. *Journal of the American Statistical Association 47*(260), 663–685.

Hou, W.-C. and G. Ozsoyoglu (1991). Statistical estimators for aggregate relational algebra queries. *ACM Transactions on Database Systems 16*(4), 600–654.

Hou, W.-C., G. Özsoyoğlu, and E. Dogdu (1991). Error-constrained COUNT query evaluation in relational databases. In *Proc. of the 1991 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 278–287.

Hou, W.-C., G. Ozsoyoglu, and B. K. Taneja (1988). Statistical estimators for relational algebra expressions. In *Proc. of the 1988 ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 276–287.

IDC (2007). *The Expanding Digital Universe*. IDC. http://www.emc.com/digital_universe.

IDC (2008). *The Diverse and Exploding Digital Universe*. IDC. http://www.emc.com/digital_universe.

Ikeji, A. C. and F. Fotouhi (1995). Computation of partial query results with an adaptive stratified sampling technique. In *Proc. of the 1995 Conf. on Information and Knowledge Management*, pp. 145–149.

Ilyas, I. F., V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga (2004). CORDS: automatic discovery of correlations and soft functional dependencies. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 647–658.

Indyk, P. (1999). A small approximately min-wise independent family of hash functions. In *Proc. of the 1999 Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 454–456.

Ioannidis, Y. E. (2003). The history of histograms (abridged). In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pp. 19–30.

Jagadish, H. V., N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel (1998). Optimal histograms with quality guarantees. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pp. 275–286.

Jermaine, C. (2003). Robust estimation with sampling and approximate pre-aggregation. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pp. 886–897.

Jermaine, C., S. Arumugam, A. Pol, and A. Dobra (2007). Scalable approximate query processing with the dbo engine. In *Proc. of the 2007 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 725–736.

Jermaine, C., A. Dobra, S. Arumugam, S. Joshi, S. Joshi, and A. Pol (2005). A disk-based join with probabilistic guarantees. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 563–574.

Jermaine, C., A. Pol, and S. Arumugam (2004). Online maintenance of very large random samples. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 299–310.

Jin, R., L. Glimcher, C. Jermaine, and G. Agrawal (2006). New sampling-based estimators for olap queries. In *Proc. of the 2006 Intl. Conf. on Data Engineering*, pp. 18.

Johnson, N. L., S. Kotz, and A. W. Kemp (1992). *Univariate Discrete Distributions* (2nd ed.). Wiley Series in Probability and Mathematical Statistics. Wiley-Interscience.

Johnson, T., S. Muthukrishnan, and I. Rozenbaum (2005). Sampling algorithms in a stream operator. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1–12.

Jones, T. G. (1962). A note on sampling a tape-file. *Communications of the ACM 5*(6), 343.

Joze-Hkajavi, N. and K. Salem (1998). Two-phase clustering of large datasets. Technical Report CS-98-27, Department of Computer Science, University of Waterloo. http://www.cs.uwaterloo.ca/research/tr/1998/27/CS-98-27.pdf.

Kachitvichyanukul, V. and B. Schmeiser (1985). Computer generation of hypergeometric random variables. *Journal of Statistical Computation and Simulation 22*, 127–145.

Kaufman, L. and P. J. Rousseeuw (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley-Interscience.

Kaushik, R., J. F. Naughton, R. Ramakrishnan, and V. T. Chakravarthy (2005). Synopses for query optimization: A space-complexity perspective. *ACM Transactions on Database Systems 30*(4), 1102–1127.

Kivinen, J. and H. Mannila (1994). The power of sampling in knowledge discovery. In *Proc. of the 1994 ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 77–85.

Knuth, D. E. (1969). *The Art of Computer Programming* (1st ed.), Volume 2: Seminumerical Algorithms. Addison Wesley.

Knuth, D. E. (1981). *The Art of Computer Programming* (2nd ed.), Volume 2: Seminumerical Algorithms. Addison Wesley.

Knuth, D. E. (1997). *The Art of Computer Programming* (3rd ed.), Volume 1: Fundamental Algorithms. Addison Wesley.

Kollios, G., D. Gunopulos, N. Koudas, and S. Berchtold (2001). An efficient approximation scheme for data mining tasks. In *Proc. of the 2001 Intl. Conf. on Data Engineering*, pp. 453–462.

Krishnaiah, P. and C. Rao (Eds.) (1988). *Handbook of Statistics 6: Sampling*. Elsevier Science.

Larson, P.-Å., W. Lehner, J. Zhou, and P. Zabback (2007). Cardinality estimation using sample views with quality assurance. In *Proc. of the 2007 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 175–186.

L'Ecuyer, P. (2006). Uniform random number generation. In S. G. Henderson and B. L. Nelson (Eds.), *Simulation*, pp. 55–81. Elsevier.

Li, K.-H. (1994). Reservoir-sampling algorithms of time complexity o(n(1 + log(n/n))). *ACM Transactions on Mathematical Software 20*(4), 481–493.

Ling, Y. and W. Sun (1995). An evaluation of sampling-based size estimation methods for selections in database systems. In *Proc. of the 1995 Intl. Conf. on Data Engineering*, pp. 532–539.

Lipton, R. J. and J. F. Naughton (1990). Query size estimation by adaptive sampling (extended abstract). In *Proc. of the 1990 ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 40–46.

Lipton, R. J., J. F. Naughton, and D. A. Schneider (1990). Practical selectivity estimation through adaptive sampling. In *Proc. of the 1990 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1–11.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proc. of the 1967 Berkeley Symp. on Mathematical Statistics and Probability*, pp. 281–297.

Manku, G. S. and R. Motwani (2002). Approximate frequency counts over data streams. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, pp. 346–357.

Matsumoto, M. and T. Nishimura (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation 8*(1), 3–30.

McLeod, A. and D. Bellhouse (1983). A convenient algorithm for drawing a simple random sample. *Applied Statistics 32*(2), 182–184.

Menezes, A. J., P. C. van Oorschot, and S. A. Vanstone (1996). *Handbook of Applied Cryptography* (5th ed.). CRC Press.

Mitzenmacher, M. and S. Vadhan (2008). Why simple hash functions work: Exploiting the entropy in a data stream. In *Proc. of the 2008 Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 746–755.

Motwani, R. and P. Raghavan (1995). *Randomized Algorithms*. Cambridge University Press.

Muller, M. E. (1958). The use of computers in inspection procedures. *Communications of the ACM 1*(11), 7–13.

Muralikrishna, M. and D. J. DeWitt (1988). Equi-depth multidimensional histograms. In *Proc. of the 1988 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 28–36.

Muthukrishnan, S. (2005). Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science 1*(2).

Nair, K. A. (1990). An improved algorithm for ordered sequential random sampling. *ACM Transactions on Mathematical Software 16*(3), 269–274.

Nath, S. and P. Gibbons (2008). Online maintenance of very large random samples on flash storage. In *Proc. of the 2008 Intl. Conf. on Very Large Data Bases*. (to appear).

NIST Federal Information Processing Standards (2001). Advanced encryption standard. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

Olken, F. (1993). *Random Sampling from Databases*. Ph. D. thesis, Lawrence Berkeley Laboratory. LBL-32883.

Olken, F. and D. Rotem (1986). Simple random sampling from relational databases. In *Proc. of the 1986 Intl. Conf. on Very Large Data Bases*, pp. 160–169.

Olken, F. and D. Rotem (1989). Random sampling from b+ trees. In *Proc. of the 1989 Intl. Conf. on Very Large Data Bases*, pp. 269–277.

Olken, F. and D. Rotem (1992). Maintenance of materialized views of sampling queries. In *Proc. of the 1992 Intl. Conf. on Data Engineering*, pp. 632–641.

Olken, F., D. Rotem, and P. Xu (1990). Random sampling from hash files. In *Proc. of the 1990 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 375–386.

Palmer, C. R. and C. Faloutsos (2000). Density biased sampling: an improved method for data mining and clustering. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 82–92.

Piatetsky-Shapiro, G. and C. Connell (1984). Accurate estimation of the number of tuples satisfying a condition. In *Proc. of the 1984 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 256–276.

Pinkham, R. S. (1987). An efficient algorithm for drawing a simple random sample. *Applied Statistics 36*(3), 307–372.

Pol, A., C. Jermaine, and S. Arumugam (2008). Maintaining very large random samples using the geometric file. *The VLDB Journal 17*(5), 997–1018.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (1992). *Numerical Recipes in C* (2nd ed.). Cambridge University Press.

Rao, J. (1966). On the comparison of sampling with and without replacement. *Review of the International Statistical Institute 34*(2), 125–138.

Rösch, P., R. Gemulla, and W. Lehner (2008). Designing random sample synopses with outliers. In *Proc. of the 2008 Intl. Conf. on Data Engineering*, pp. 1400–1402.

Särndal, C.-E., B. Swensson, and J. Wretman (1991). *Model Assisted Survey Sampling*. Springer Series in Statistics. Springer Verlag.

SAS Institute Inc. (1998). *SAS Institute Best Practices Paper: Data Mining and the Case for Sampling*. SAS Institute Inc. http://www.sasenterpriseminer.com/documents/SAS-SEMMA.pdf.

Seidel, R. and C. R. Aragon (1996). Randomized search trees. *Algorithmica 16*(4/5), 464–497.

Serfling, R. J. (1980). *Approximation theorems of mathematical statistics*. Wiley Series in Probability and Mathematical Statistics. Wiley.

Seshadri, S. (1992). *Probabilistic methods in query processing*. Ph. D. thesis, University of Wisconsin at Madison.

Strand, M. M. (1979). Estimation of a population total under a "bernoulli sampling" procedure. *The American Statistician 33*(2), 81–84.

Takei, Y., T. Itoh, and T. Shinozaki (2000). Constructing an optimal family of min-wise independent permutations. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E83-A*(4), 747–755.

Tao, Y., X. Lian, D. Papadias, and M. Hadjieleftheriou (2007). Random sampling for continuous streams with arbitrary updates. *IEEE Transactions on Knowledge and Data Engineering 19*(1), 96–110.

Teuhola, J. and O. Nevalainen (1982). Two efficient algorithms for random sampling without replacement. *International Journal of Computer Mathematics 11*, 127–140.

The R Project (2008). The R project for statistical computing. http://www.r-project.org/.

Thompson, S. K. (1992). *Sampling* (1st ed.). Wiley Series in Probability and Mathematical Statistics. Wiley.

Thorup, M. (2000). Even strongly universal hashing is pretty fast. In *Proc. of the 2000 Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 496–497.

Thorup, M. and Y. Zhang (2004). Tabulation based 4-universal hashing with applications to second moment estimation. In *Proc. of the 2004 Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 615–624.

Tillé, Y. (2006). *Sampling Algorithms*. Springer Series in Statistics. Springer.

Toivonen, H. (1996). Sampling large databases for association rules. In *Proc. of the 1996 Intl. Conf. on Very Large Data Bases*, pp. 134–145.

Vitter, J. S. (1984). Faster methods for random sampling. *Communications of the ACM 27*(7), 703–718.

Vitter, J. S. (1985). Random sampling with a reservoir. *ACM Transactions on Mathematical Software 11*(1), 37–57.

Vitter, J. S. (1987). An efficient algorithm for sequential random sampling. *ACM Transactions on Mathematical Software 13*(1), 58–67.

Willard, D. E. (1991). Optimal sample cost residues for differential database batch query problems. *Journal of the ACM 38*(1), 104–119.

Winter, R. (2008). Why are data warehouses growing so fast? *Beye Business Intelligence Network*. http://www.b-eye-network.com/view/7188.

Xu, F., C. Jermaine, and A. Dobra (2008). Confidence bounds for sampling-based group by estimates. *ACM Transactions on Database Systems*. (to appear).

Yao, S. B. (1977). Approximating block accesses in database organizations. *Communications of the ACM 20*(4), 260–261.

Zechner, H. (1997). *Efficient Sampling from Continuous and Discrete Distributions*. Ph. D. thesis, Technical University Graz.

Zhang, T., R. Ramakrishnan, and M. Livny (1996). BIRCH: An efficient data clustering method for very large databases. In *Proc. of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 103–114.

List of Figures

List of Tables

List of Algorithms

Index of Notation

A small number of symbols have a different meaning in different chapters. In the table below, we indicate such overloaded symbols by a preceding star symbol (*) in the where column; the star indicates that the specified meaning is only valid in the specified chapter or section. We do not list variables that have a very limited scope (e.g., a single paragraph or page).

Symbol	Meaning	Where
$\|A\|$	number of items in A (including duplicates)	2.1.2A
$\|D(A)\|$	number of distinct items in A	2.1.2A
$[k]$	set of nonnegative integers less than k: $\{0, \ldots, k-1\}$	6.1
$\binom{N}{n}$	binomial coefficient: $N!/(n!(N-n)!)$	2.1.2A
$n!$	factorial: $n(n-1)(n-2)\cdots 1$, $0! = 1$	2.1.2A
$x \propto y$	x is proportional to y: $x = cy$ for some constant $c \neq 0$	3.3.1
$\cup, \cap, \setminus$	set union, intersection, and difference	3.2
$\uplus, \overset{+}{\cap}, \setminus^{+}$	multiset union, intersection, and difference	3.2
$\mathrm{ARE}[\hat{\theta}]$	average relative error: $\mathrm{E}[\,\lvert\hat{\theta} - \theta\rvert/\theta\,]$	2.1.3A
b	cluster size: number of consecutive transactions of the same type	4.1.3A
B_n	random variable for the number of base-data accesses to retrieve the nth distinct item	3.4.1
$B(k; N, q)$	binomial probability: $\binom{N}{k} q^k (1-q)^{N-k}$	2.1.2A
$B(a, b)$	Beta function: $\int_0^1 t^{a-1}(1-t)^{b-1}\,dt$	6.2.2B
$\mathrm{Bias}[\hat{\theta}]$	expected distance to true value: $\mathrm{E}[\,\hat{\theta}\,] - \theta$	2.1.3A
$\mathrm{Cov}[\,X, Y\,]$	covariance of X and Y: $\mathrm{E}[\,(X - \mathrm{E}[\,X\,])(Y - \mathrm{E}[\,Y\,])\,]$	2.1.3D
c_b	random variable for the number of uncompensated bad deletions (deleted item in sample)	4.1.1A
$c_{\mathrm{b},i}$	random variable for the number of uncompensated bad deletions after ith transaction	4.1.1C
$C_\mathrm{b}(\gamma)$	random variable for the number of bad deletions that result by applying RP on γ	4.2.4A
c_g	random variable for the number of uncompensated good deletions (deleted item not in sample)	4.1.1A
$c_{\mathrm{g},i}$	random variable for the number of uncompensated good deletions after ith transaction	4.1.1C

Symbol	Meaning	Where
$C_{\mathrm{g}}(\gamma)$	random variable for the number of good deletions that result by applying RP on γ	4.2.4A
ϵ	bound on relative error	6.2.2B
e_i	data-stream item of form (i, r_i, t_i)	3.5.4
$\mathrm{E}[X]$	expected value of random variable X	2.1.3A
δ	probability of error	2.1.3C
$1-\delta$	probability of success, condidence level	2.1.3C
Δ	window length parameter of a time-based window	3.5.4
$\Delta_{i,N}$	window length (in time units) of a size-N sequence-based window at time i: $t_i - t_{N-i}$	3.5.4
d	number of uncompensated deletions: $c_{\mathrm{g}} + c_{\mathrm{b}}$	4.1.1A
d_i	number of uncompensated deletions after ith transaction	4.1.1C
$d(\gamma)$	number of uncompensated deletions in γ	4.2.4A
D	number of distinct items in population: $\lvert D(R)\rvert$	5.1.2C
D^+	number of cumulated insertions of new distinct items: $\lvert D(R^+)\rvert$	6.2.1B
$D(A)$	set of distinct items in A	2.1.2A
γ	sequence of transactions: $(\gamma_1, \gamma_2, \ldots)$	3.2
γ_i	ith transaction in sequence: either $+r_j$, $-r_j$ or $r_j \leftarrow r_k$	3.2
h	hash function	*6
H	range of hash function	*6
$\mathscr{H}$	family of hash functions	*6
h	index of stratum	*7.2
H	number of strata	*7.2
H_n	nth harmonic number: $\sum_{i=1}^{n} 1/i$	3.4.1
$H_{n,m}$	partial harmonic number: $\sum_{i=n}^{m} 1/i = H_m - H_{n-1}$	3.4.1
$H(k;\, N, N', M)$	hypergeometric probability: $\binom{N'}{k}\binom{N-N'}{M-k} \,/\, \binom{N}{M}$	4.1.1D
$I_x(a,b)$	regularized incomplete Beta function: $B(a,b)^{-1}\int_0^x t^{a-1}(1-t)^{b-1}dt$	6.2.2B
l_i	smallest possible sample size after ith transaction	4.1.1C
L	lower bound on the sample size	3.3.4B
L	random variable for the number of steps in phase 2 of RPRES	*4.2.1C
μ	population average: τ/N	2.1.3B
M	sample size parameter (expected/desired/maximum sample size)	2.1.2A
M'	sample size parameter after resizing	4.2
$\mathrm{MSE}[\hat{\theta}]$	mean squared error: $\mathrm{Var}[\hat{\theta}] + \mathrm{Bias}[\hat{\theta}]^2$	2.1.3A
n	random variable for the sample size: $\lvert S\rvert$	2.1.2A

Symbol	Meaning	Where
n_i	random variable for the sample size after ith transaction: $\lvert S_i \rvert$	3.5
N	population size: $\lvert R \rvert$	2.1.1
N_i	population size after ith transaction: $\lvert R_i \rvert$	3.2
$N_i(r)$	multiplicity of item r in population after ith transaction	3.5.2
N_h	size of stratum R_h	*7.2
$N_V(r)$	multiplicity of r in multiset V	6.4.1
$N_\Delta(t)$	window size at time t: $\lvert W_\Delta(t) \rvert$	3.5.4
$N(t)$	shortcut for $N_\Delta(t)$	3.5.4
N^+	cumulative number of insertions: $\lvert R^+ \rvert$	3.5.1F
N_i^+	cumulative number of insertions after ith transaction	3.5.1F
N^+	number of arrivals until next stratum expiration	*7.2
$O(f(n))$	Big Oh: set of functions bounded from above by $f(n)$ (asymptotically and up to a constant factor)	
$\Omega(f(n))$	Big Omega: set of functions bounded from below by $f(n)$ (asymptotically and up to a constant factor)	
π_i	first-order inclusion probability of item r_i: $\Pr[r_i \in S]$	2.1.2B
π_{ij}	second-order inclusion probability of items r_i and r_j: $\Pr[r_i \in S, r_j \in S]$	2.1.3B
$P_x(a)$	regularized (lower) incomplete gamma function: $1 - e^{-x}\sum_{i=0}^{a-1} x^i/i!$ for integer $a > 0$	6.2.2C
$\mathcal{P}(A)$	power set of A	2.1.3
$\Pr[A]$	probability of occurrence of A	2.1.2A
$\Pr[A_1, \ldots, A_n]$	probability of joint occurrence of $A_1, \ldots, A_n$	
$\Pr[A \mid B]$	conditional probability of A given B	
q	sampling rate for Bernoulli sampling	2.1.2A
$\mathscr{R}$	universe	3.2
R	population (set or multiset)	2.1.1
R_i	population after processing the first i transactions, $R_0 = \emptyset$	3.2
R_h	a stratum (a partition of R)	*7.2
$R(t)$	population at time t	3.5.4
$R(\gamma)$	population generated by γ	4.2.4A
R^+	augmented population (deleted items retained)	4.2.4A
R^+	set of items that arrive until next stratum expiration	*7.2
$\mathrm{RMSE}[\hat{\theta}]$	root mean squared error: $\sqrt{\mathrm{MSE}[\hat{\theta}]}$	2.1.3A
r_i	an item	2.1.1
$+r$	insertion of item r	3.2
$-r$	deletion of item r	3.2
$r \to r'$	update of item r to item r'	3.2

Symbol	Meaning	Where
$r_{(i)}$	ith order statistic = ith smallest item = i/Nth quantile	2.1.3E
$\mathscr{S}$	set of possible samples	2.1.1
σ^2	population variance: $1/(N-1)\sum_{r_i \in R}(y_i - \mu)^2$	2.1.3B
$\sigma_p(A)$	elements of A that satisfy predicate p (may contain duplicates)	6.2.2F
s^2	sample variance: $1/(n-1)\sum_{r_i \in S}(y_i - \bar{y})^2$	2.1.3B
S	random sample (set or multiset)	2.1.2A
S_i	random sample after processing the first i transactions, $S_0 = \emptyset$	3.2
$S(t)$	random sample at time t	3.5.4
$S(\gamma)$	random sample generated by γ	4.2.4A
S^*	net sample	3.5.1F
S_i*	net sample after ith transaction	3.5.1F
S^+	augmented random sample from R^+	4.2.4A
$\mathrm{SE}[\,X\,]$	standard error of random variable X: $\sqrt{\mathrm{Var}[\,X\,]}$	2.1.3A
$\hat{\mathrm{SE}}[\,X\,]$	estimator of $\mathrm{SE}[\,X\,]$	
t_a	cost of base data access	4.2.2A
t_b	interarrival time between consecutive transactions	4.2.2A
t_i	timestamp of item e_i	3.5.4
$T(d)$	expected total resizing cost with d uncompensated deletions	4.2.2A
$T_1(d)$	expected phase 1 resizing cost with d uncompensated deletions	4.2.2A
$T_2(d)$	expected phase 2 resizing cost with d uncompensated deletions	4.2.2A
τ	population total: $\sum_{r_i \in R} y_i$	2.1.3B
τ_h	total of stratum R_h	7.2
θ	population parameter	2.1.3
$\hat{\theta}$	estimator of θ	2.1.3
$\hat{\theta}(s)$	estimate of θ obtained by applying $\hat{\theta}$ to sample s	2.1.3
U	random variable for the intermediate sample size in phase 1 of RPRES	4.2.1C
U_i	normalized hash value of r_i: $h(r_i)/H$	6.2.2A
$U_{(M)}$	Mth smallest value of $\{\,U_i \mid r_i \in R\,\}$	6.2.2A
u_i	largest possible sample size after ith transaction	4.1.1C
v_i	largest seen sample size after ith transaction	4.1.1C
$\mathrm{Var}[\,X\,]$	variance of random variable X: $\mathrm{E}[\,(X - \mathrm{E}[\,X\,])^2\,] = \mathrm{E}[\,X^2\,] - \mathrm{E}[\,X\,]^2$	2.1.3A
$\hat{\mathrm{Var}}[\,X\,]$	estimator of $\mathrm{Var}[\,X\,]$	
$W_{i,N}$	sequence-based sliding window of size N: $R_i \setminus R_{i-N}$	3.5.4

Symbol	Meaning	Where
$W_\Delta(t)$	sliding window of length Δ at time t: $R(t) \setminus R(t-\Delta)$	3.5.4
$W(t)$	shortcut for $W_\Delta(t)$	3.5.4
X_i	shortcut for $X_i(r)$	5.1.1A
$X_i(r)$	random variable for the multiplicity of item r in sample after ith transaction	3.5.2
$\bar{y}$	sample average: $1/n \sum_{r_i \in S} y_i$	2.1.3B
y_i	numerical value associated with item r_i: $f(r_i)$	2.1.3B
Y_i	shortcut for $Y_i(r)$	5.1.1A
$Y_i(r)$	random variable for the number of net insertions of r after ith transaction, starting the counting process after first sample acceptance of r	5.1.1A
Z	skip counter for Bernoulli sampling	3.4.3B
Z_i	skip counter for reservoir sampling	3.4.3D
$Z_{i,n}$	skip counter for list-sequential sampling	3.4.2

Index of Algorithm Names

The first table describes our general naming scheme for algorithms, where each name consists of zero or more prefixes, a basic name, zero or more suffixes and an optional parameter list. The second table lists all the algorithms names and the position where they are described.

Position	**Symbol**	**Meaning**
Prefix	A	augmented (with counters)
	M	modified (to support updates and/or deletions)
Basic name	BERN	Bernoulli sampling
	BPS	bounded priority sampling
	CAR	correlated acceptance-rejection sampling
	CHAIN	chain sampling
	MIN	min-wise sampling
	PASSIVE	passive sampling
	PS	priority sampling
	RP	random pairing
	RS	reservoir sampling
	SRS	simple random sampling
Suffix	D	for distinct items
	M	for multisets
	P	with purging
	R	with recomputation
	T	with tagging
	W	for sliding windows
	WR	with replacement
	WOR	without replacement
Parameter	d	number of uncompensated deletions
	q	Bernoulli sampling rate
	L	minimum sample size
	M	desired/maximum sample size

Name	**Meaning**	**Where**
ABERN(q)	augmented Bernoulli sampling for multisets	5.1.1
ABERND(q)	augmented Bernoulli sampling for distinct items	5.1.2C
AMIND(M)	augmented min-hash sampling	6.2.1
BERN(q)	Bernoulli sampling	3.4.3A
BERND(q)	Bernoulli sampling for distinct items	3.5.3A
BERNDP(M)	Bernoulli sampling for distinct items with purging	3.5.3C
BERNW(q)	Bernoulli sampling for sliding windows	3.5.4A
BPS(M)	bounded priority sampling	7.1.3
BPSWOR(M)	bounded priority sampling with replacement	7.1.3E
CAR(M)	correlated acceptance/rejection sampling	3.5.1G
CARWOR(M)	correlated acceptance/rejection sampling without replacement	3.5.1H
PASSIVE(M)	chain sampling	3.5.4C
DIS(M)	dynamic inverse sampling for distinct items	3.5.3F
MBERN(q)	modified Bernoulli sampling	3.5.1A
MBERND(q)	modified Bernoulli sampling for distinct items	3.5.3B
MBERNM(q)	modified Bernoulli sampling for multisets	3.5.2A
MBERNP(M)	modified Bernoulli sampling with purging	3.5.1B
MBS	merge-based stratification	7.2.2
MERGE	merging algorithm arbitrary uniform samples	4.3.1
MIN(M)	min-wise sampling	3.4.3E
MIND(M)	min-hash sampling for distinct items	3.5.3D
MINDWR(M)	min-hash sampling for distinct items with replacement	3.5.3E
MRS(M)	modified reservoir sampling	3.5.1C
MRSR(L, M)	modified reservoir sampling with recomputation	3.5.1E
MRST(L, M)	modified reservoir sampling with tagging (and recomputation)	3.5.1F
PASSIVE(M)	passive sampling	3.5.4B
PS(M)	priority sampling	3.5.4D
PSWOR(M)	priority sampling with replacement	3.5.4E
RP(M)	random pairing	4.1.1A
RP(M, d)	random pairing with d uncompensated deletions	4.2.1C
RPMERGE	merging algorithm for random pairing	4.3.2
RPSUB	resizing algorithm for random pairing (downwards)	4.2.4B
RPRES	resizing algorithm for random pairing (upwards)	4.2.1C
RPR(L, M)	random pairing with recomputation	4.1.3B
RS(M)	reservoir sampling	3.4.3C
SRS(M)	simple random sampling	2.1.2A
SRSWR(M)	simple random sampling with replacement	2.1.2A
SUB(M')	basic subsampling algorithm used within RPSUB	4.2.4

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Dissertation selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rainer Gemulla
Dresden, 27. August 2008