

Abhijit Pol · Christopher Jermaine · Subramanian Arumugam

Maintaining Very Large Random Samples Using the Geometric File ^{*}

Received: date / Accepted: date

Abstract Random sampling is one of the most fundamental data management tools available. However, most current research involving sampling considers the problem of how to use a sample, and not how to compute one. The implicit assumption is that a “sample” is a small data structure that is easily maintained as new data are encountered, even though simple statistical arguments demonstrate that very large samples of gigabytes or terabytes in size can be necessary to provide high accuracy. No existing work tackles the problem of maintaining very large, disk-based samples from a data management perspective, and no techniques now exist for maintaining very large samples in an online manner from streaming data. In this paper, we present online algorithms for maintaining on-disk samples that are gigabytes or terabytes in size. The algorithms are designed for streaming data, or for any environment where a large sample must be maintained online in a single pass through a data set. The algorithms meet the strict requirement that the sample always be a true, statistically random sample (without replacement) of all of the data processed thus far. We also present algorithms to retrieve small size random sample from large disk-based sample which may be used

for various purposes including statistical analyses by a DBMS.

1 Introduction

Despite the variety of alternatives for approximate query processing (including several references listed at the end of this paper [13,14,20,18,39]), sampling is still one of the most powerful methods for building a one-pass synopsis of a data set, especially in a streaming environment where the assumption is that there is too much data to store all of it permanently. Sampling’s many benefits include:

- Sampling is the most widely-studied and best understood approximation technique currently available. Sampling has been studied for hundreds of years, and many fundamental results describe the utility of random samples (such as the Central Limit Theorem, Chernoff, Hoeffding and Chebyshev bounds [9,35]).
- Sampling is the most versatile approximation technique available. Most data processing algorithms can be used on a random sample of a data set rather than the original data with little or no modification. For example, almost any data mining algorithm for building a decision tree classifier can be run directly on a sample.
- Sampling is the most widely-used approximation technique. Sampling is common in data mining, statistics, and machine learning. The sheer number of recent papers from ICDE, VLDB, and SIGMOD [1,3,5,8,7,16,17,19,22,24,25,36] that use samples testify to sampling’s popularity as a data management tool.

Given the obvious importance of random sampling, it is perhaps surprising that there has been very little work in the data management community on *how to actually perform random sampling*. The most well-known papers in this area are due to Olken and Rotem [32,31], who also offer the definitive survey of related work

* A version of this work appeared in the proceedings of SIGMOD 2004, Paris, France, under the title, “*Online Maintenance of Very Large Random Samples*.”

Abhijit Pol
Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611.
Tel.: +1-352-8714729
Fax: +1-352-3921220
E-mail: apolt@cise.ufl.edu

Christopher Jermaine
Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611.
E-mail: cjermain@cise.ufl.edu

Subramanian Arumugam
Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611.
E-mail: sa2@cise.ufl.edu

through the early 1990's [33]. However, this work is relevant mostly for sampling from data stored in a database, and implicitly assumes that a "sample" is a small data structure that is easily stored in main memory.

Such assumptions are sometimes overly restrictive. Consider the problem of approximate query processing. Recent work has suggested the possibility of maintaining a sample of a large database and then executing analytic queries over the sample rather than the original data as a way to speed up processing [2, 21]. Given the most recent TPC-H benchmark results¹, it is clear that processing standard report-style queries over a large, multi-terabyte data warehouse may take hours or days. In such a situation, maintaining a fully materialized random sample of the data (or "sample view" [30]) may be desirable. In order to save time and/or computer resources, queries can then be evaluated over the sample rather than the original data, as long as the user can tolerate some carefully controlled inaccuracy in the query results.

This particular application has two specific requirements that are addressed by our work. First, it may be necessary to use quite a large sample in order to achieve acceptable accuracy; perhaps on the order of gigabytes in size. This is especially true if the sample will be used to answer selective queries or aggregates over attributes with high variance (see Section 2). Second, whatever the required sample size, it is often independent of the size of the database, since estimation accuracy depends primarily on sample size². In other words, the required sample size will generally not grow as the database size increases, as long as other factors such as query selectivity remain relatively constant. Thus, this application requires that we be able to maintain a large, disk-based, fixed-size random sample of the archived data, even as new data are added to the warehouse. This is precisely the problem we tackle in the paper.

For another example of a case where existing sampling methods can fall short, consider stream-based data management tasks, such as network monitoring (for an example of such an application, we point to the Gigascope project from AT&T Laboratories [11, 12, 10]). Given the tremendous amount of data transported over today's computer networks, the only conceivable way to facilitate ad-hoc, after-the-fact query processing over the set of packets that have passed through a network router is to build some sort of statistical model for those packets. The most obvious choice would be to produce a very large, statistically random sample of the packets that have passed through the router. Again, maintaining such a sample is precisely the problem we tackle in this paper. While other researchers have tackled the problem of

maintaining an online sample targeted towards more recent data [6], no existing methods have considered how to handle very large samples that exceed the available main memory.

If one accepts the notion that being able to maintain a very large (but fixed size) random sample from a data stream is an important problem, it is reasonable to ask: Is maintaining such a sample difficult or costly using modern algorithms and hardware? Fortunately, modern storage hardware gives us the capacity to inexpensively store very large samples that should suffice for even difficult and emerging applications. A terabyte of commodity hard disk storage now costs less than \$1,000. Given current trends, we should see storage costs of \$1,000 per petabyte by the year 2020. However, even given such large storage capacities, it turns out that maintaining a large sample is difficult using current technology. The problem is not purchasing the hardware to store the sample; rather, the problem is actually getting the samples onto disk, so as to guarantee the statistical randomness of the sample, in the face of data streams that may exceed tens of gigabytes per minute in the case of a network monitoring application.

Current techniques suitable for maintaining samples from a data stream are based on *reservoir sampling* [15, 28]. Reservoir sampling algorithms can be used to dynamically maintain a fixed-size sample of N records from a stream, so that at any given instant, the N records in the sample constitute a true random sample of all of the records that have been produced by the stream. However, as we will discuss in this paper, the problem is that existing reservoir techniques are suitable *only when the sample is small enough to fit into main memory*.

Given that there are limited techniques for maintaining very large samples, the problem addressed in this paper is as follows:

Given a main memory buffer B large enough to hold $|B|$ records, can we develop efficient algorithms for dynamically maintaining a massive random sample containing exactly N records from a data stream, where $N \gg |B|$?

Key design goals for the algorithms we develop are:

1. The algorithms must be suitable for streaming data, or any similar environment where a large sample must be maintained on-line in a single pass through a data set, with the strict requirement that the sample always be a true, statistically random sample of fixed size N (without replacement) from all of the data produced by the stream thus far.
2. When maintaining the sample, the fraction of I/O time devoted to reads should be close to zero. Ideally, there would never be a need to read a block of samples from disk simply to add one new sample and subsequently write the block out again.

¹ <http://www.tpc.org>

² The unimportance of database size for certain queries is due to the fact that the bias and variance of many sampling-based estimators are related far more to sample size than to the sampling fraction (see Cochran [9] for a thorough treatment of finite population random sampling).

3. The fraction I/O of time spent performing random I/Os should also be close to zero. Costly random disk seeks should be few and far between. Almost all I/O should be sequential.
4. Finally, the amount of data written to disk should be bounded by the total size of all of the records that are ever sampled.

Our Contributions

In this paper, we describe a new data organization called the *geometric file* for maintaining a very large, disk-based sample from a data stream (where “sample” refers to a simple random sample without replacement from the underlying data set). The geometric file meets each of the requirements listed above. With memory large enough to buffer $|B| > 1$ records, the geometric file can be used to maintain an online sample of arbitrary size with an amortized cost of $O(\omega \times \log|B|/|B|)$ random disk head movements for each newly sampled record (see Section 6.2). The multiplier ω can be made arbitrarily small by making use of additional disk space. A rigorous benchmark of the geometric file demonstrates its superiority over the obvious alternatives.

Paper Organization

The rest of the paper is organized as follows. In Section 2, we give an explanation as to why very large sample sizes can be mandatory in common situations. Section 3 describes three initial alternatives for maintaining very large, disk-based samples in a streaming environment. In Sections 4 and 5 we describe the geometric file. Section 6 describes how multiple geometric files can be maintained all-at-once to achieve considerable speed up with the same target sample size. In Section 7 we discuss algorithms to retrieve small size random sample from large disk-based sample. In Section 8 we present some benchmarking results. Section 9 discusses related work, and the paper is concluded in Section 10.

2 Sampling: Sometimes a Little Is Not Enough

One advantage of random sampling is that samples usually offer statistical guarantees on the estimates they are used to produce. Typically, a sample can be used to produce an estimate for a query result that is guaranteed to have error less ε than with a probability δ (see Cochran for a nice introduction to sampling [9]). The δ value is known as the *confidence* of the estimate.

Very large samples are often required to provide accurate estimates with suitably high confidence. The need for very large samples can be easily explained in the context of the *Central Limit Theorem* (CLT) [31]. The CLT

implies that if we use a random sample of size N to estimate the mean μ of a set of numbers, the error of our estimate is usually normally distributed with mean zero and variance σ^2/N , where σ^2 is the variance of the set over which we are performing our estimation. Since the “spread” of a normally distributed random variable is proportional to the square root of the variance (also known as the *standard deviation*), the error observed when using a random sample is governed by two factors:

1. The error is *inversely* proportional to the square root of the sample size.
2. The error is *directly* proportional to the standard deviation of the set over which we are estimating the mean over.

The significance of this observation is that the sample size required to produce an accurate estimate can vary tremendously in practice, and grows quadratically with increasing standard deviation. For example, say that we use a random sample of 100 students at a university to estimate the average students age. Imagine that the average age is 20 with a standard deviation of 2 years. According to the CLT, our sample-based estimate will be accurate to within 2.5% with confidence of around 98%, giving us an accurate guess as to the correct answer with only 100 sampled students.

Now, consider a second scenario. We want to use a second random sample to estimate the average net worth of households in the United States, which is around \$140,000, with a standard deviation of at least \$5,000,000. Because the standard deviation is so large, a quick calculation shows we will need more than *12 million* samples to achieve the same statistical guarantees as in the first case.

Required sample sizes can be far larger when standard database operations like relational selection and join are considered, because these operations can effectively magnify the variance of our estimate. For example, the work on ripple joins [22] provides an excellent example of how variance can be magnified by sampling over the relational join operator.

3 Very Large Samples In a Single Pass

This Section gives some background on maintaining very large samples in a streaming environment. We begin by discussing a classical algorithm for maintaining an online sample from a data stream, and then discuss a few adaptations for disk-based samples.

3.1 Reservoir Sampling

The classic algorithm for maintaining an online random sample of a data stream is known as *reservoir sampling* [15,28]. To maintain a reservoir sample R of target size $|R|$, the following loop is used:

Algorithm 1 Reservoir Sampling

```

1: Add first  $|R|$  items from the stream directly to  $R$ 
2: for int  $i = |R| + 1$  to  $\infty$  do
3:   Wait for a new record  $r$  to appear in the stream
4:   with probability  $|R|/i$  do
5:     Remove a randomly selected record from  $R$ 
6:     Add  $r$  to  $R$ 

```

A key benefit of the reservoir algorithm is that after each execution of the for loop, it can be shown that the set R is a true, uniform random sample (without replacement) of the first i records from the stream. Thus, at all times, the algorithm maintains an unbiased snapshot of all of the data produced by the stream. The name “reservoir sampling” is an apt one. The sample R serves as a reservoir that buffers certain records from the data stream. New records appearing in the stream may be trapped by the reservoir, whose limited capacity then forces an existing record to exit the reservoir.

Reservoir sampling can be very efficient, with time complexity less than linear in the size of the stream. Variations on the algorithm allow it to “go to sleep” for a period of time during which it only counts the number of records that have passed by [37]. After a certain number of records have been seen, the algorithm “wakes up” and capture the next record from the stream.

3.2 Reservoirs For Very Large Samples

Reservoir sampling is very efficient if the sample is small enough to be stored in main memory. However, efficiency is difficult if a large sample must be stored on disk. Obvious extensions of the reservoir algorithm to on-disk samples all have serious drawbacks:

- *The virtual memory extension.* The most obvious adaptation for very large sample sizes is to simply treat the reservoir as if it were stored in virtual memory. The problem with this solution is that every new sample that is added to the reservoir will overwrite a random, existing record on disk, and so it will require two random disk I/Os: one to read in the block where the record will be written, and one to re-write it with the new sample. This means we can sample only on the order of 50 records per second at 10ms per random I/O per disk. Currently, a terabyte of storage requires as few as five disks, giving us a sampling rate of only $5 \times 50 = 250$ records per second. To put this in perspective, it would take months to sample enough 100 byte records to fill that terabyte.
- *The massive rebuild extension.* As an alternative, when new samples are selected from the stream, they are not added to the on-disk reservoir immediately. Rather, we make use of all of our available main memory to buffer new samples. At all times, the records stored in the buffer B logically represent a set samples that *should* have been used to replace on-disk samples in

order to preserve the correctness of the reservoir algorithm, but that have not yet been moved to disk for performance reasons. When the buffer B fills, we simply scan the entire reservoir R , and replace a random subset of the existing records with the new, buffered samples. The modified algorithm is given as Algorithm 2. $Count(B)$ refers to the current number of records in B . Note that since the records contained in B logically represent records in the reservoir that have not yet been added to disk, a newly-sampled record can either be assigned to replace an on-disk record, or it can be assigned to replace a buffered record (this is decided in Step (7) of the algorithm).

Algorithm 2 Reservoir Sampling with a Buffer

```

1: for int  $i = 1$  to  $\infty$  do
2:   Wait for a new record  $r$  to appear in the stream
3:   if  $i \leq |R|$  then
4:     Add  $r$  directly to  $R$  and continue
5:   else
6:     with probability  $|R|/i$  do
7:       with probability  $Count(B)/|R|$  do
8:         //new samples can overwrite buffered samples
9:         Replace a random record in  $B$  with  $r$ 
10:      else do
11:        Add  $r$  to  $B$ 
12:      if  $Count(B) == |B|$  then
13:        Scan the reservoir  $R$  and empty  $B$  in one pass
14:         $B = \emptyset$ 

```

In a realistic scenario, the ratio of the number of disk blocks to the number of records buffered in main memory may approach or even exceed one. For example, a 1 TB database with 128 KB blocks will have 7.8 million blocks; and for such a relatively large database it is realistic to expect that we have access to enough memory to buffer millions records. As the number of buffered records per block meets or exceeds one, most or all of the blocks on disk will contain a record that has been randomly selected for replacement by line (9) of Algorithm 2, and so all of the database blocks must be updated. Thus, it makes sense to rely on fast, sequential I/O to update the entire file in a single pass. The drawback of this approach is that every time that the buffer fills, we are effectively rebuilding the *entire* reservoir to process a set of buffered records that are a small fraction of the existing reservoir size.

- *The localized overwrite extension.* We will do better if we enforce a requirement that all samples are stored in a random order on disk. If data are clustered randomly, then we can simply write the buffer sequentially to disk at any arbitrary position. Because of the random clustering, we can guarantee that wherever the buffer is written to disk, the new samples will overwrite a random subset of the records in the reservoir and preserve the correctness of the algorithm. The problem with this solution is that af-

ter the buffered samples are added, *the data are no longer clustered randomly and so a randomized overwrite cannot be used a second time*. The data are now clustered by insertion time, since the buffered samples were the most recently seen in the data stream, and were written to a single position on disk. Any subsequent buffer flush will need to overwrite portions of *both* the new and the old records to preserve the algorithm’s correctness, requiring an additional random disk head movement. With each subsequent flush, maintaining randomness will become more costly, as data become more and more clustered by insertion time. Eventually, this solution will deteriorate, unless we periodically re-randomize the entire reservoir. Unfortunately, re-randomizing the entire reservoir is as costly as performing an external-memory sort of the entire file containing samples, and requires taking the sample off-line.

4 The Geometric File

The three extensions to Algorithm 1 can be used to maintain a large, on-disk sample, but all of them have drawbacks. In this section, we discuss a fourth algorithm and an associated data organization called the *geometric file* to address these pitfalls. The geometric file is best seen as an extension of the massive rebuild option given as Algorithm 2. Just like Algorithm 2, the geometric file makes use of a main-memory buffer that allows new samples selected by the reservoir algorithm to be added to the on-disk reservoir in a lazy fashion. However, the key difference between Algorithm 2 and the algorithms used by the geometric file is that the geometric file makes use of a far more efficient algorithm for merging those new samples into the reservoir.

4.1 Intuitive Description

Except for Step (13) of Algorithm 2, the basic algorithm employed by the geometric file is not much different. As far as Step (13) is concerned, the difference between the geometric file and the massive rebuild extension is that the geometric file empties the buffer more efficiently, in order to avoid scanning or periodically re-randomizing the entire reservoir.

To accomplish this, the entire sample in main memory that is flushed into the reservoir is viewed as a single *subsample* or a *stratum* [9], and the reservoir itself is viewed as a collection of subsamples, each formed via a single buffer flush. Since the records in a subsample are non-random subset of the records in the reservoir (they are sampled from the stream during a specific time period), each new subsample needs to overwrite a true, random subset of the records in the reservoir in order to

maintain the correctness of the reservoir sampling algorithm. If this can be done efficiently, we can avoid rebuilding the entire reservoir in order to process a buffer flush.

At first glance, it may seem difficult to achieve the desired efficiency. The buffered records that must be added to the reservoir will typically overwrite a subset of the records stored in *each* of the existing subsamples during a buffer flush. Though we may be able to avoid rebuilding the entire file, the fact that the buffer must overwrite a subset of each on-disk subsample presents a challenge when trying to maintain acceptable performance, because this naturally leads to fragmentation (see the discussion of the localized overwrite extension in Section 3.2). For example, if there are 100 on-disk subsamples, the buffer must be split 100 ways in order to write to a portion of each of the 100 on-disk subsamples. This fragmented buffer then becomes a new subsample, and subsequent buffer flushes that need to replace a random portion of this subsample must somehow efficiently overwrite a random subset of the subsample’s fragmented data.

The geometric file uses a careful, on-disk data organization in order to avoid such fragmentation. The key observation behind the geometric file is that the number of records of a subsample that are replaced with records from buffered sample can be characterized with reasonable accuracy using a geometric series (hence the name *geometric file*). As buffered samples are added to the reservoir via buffer flushes, we observe that *each existing subsample loses approximately the same fraction of its remaining records every time*, where the fraction of records lost is governed by the ratio of the size of a buffered sample to the overall size of the reservoir. By “loses”, we mean that the subsample has some of its records replaced in the reservoir with records from a subsequent subsample. Thus, the size of a subsample decays approximately in an exponential manner as buffered samples are added to the reservoir.

This exponential decay is used to great advantage in the geometric file, because it suggests a way to organize the data in order to avoid problems with fragmentation. Each subsample is partitioned into a set of *segments* of exponentially decreasing size. These segments are sized so that every time a buffered sample is added to the reservoir, we expect that each existing subsample loses exactly the set of records contained in its largest remaining segment. As a result, each subsample loses one segment to the newly-created subsample every time the buffer is emptied, and a geometric file can be organized into a fixed and unchanging set of segments that are stored as contiguous runs of blocks on disk. Because the set of segments is fixed beforehand, fragmentation and update performance are not problematic: in order to replace records in an existing subsample with the records from a new buffer flush, a simple, efficient, sequential overwrite

of the existing subsample’s largest segment generally suffices.

4.2 Characterizing Subsample Decay

To describe the geometric file in detail, we begin with an analogy between the samples in a subsample S that are lost over time, and radioactive decay. Imagine that we have 100 grams of Uranium at an initial point of time ($U_0 = 100$), and a decay rate $(1 - \alpha) = 0.1$ with a retention rate of α . On day one, the mass of Uranium decays to $U_0 \times \alpha = 90$ grams, because the Uranium loses $U_0 \times (1 - \alpha) = 10$ grams of its mass. We define $n = U_0 \times (1 - \alpha)$ to be the mass of Uranium lost on the very first day, giving $n = 10$ for our example.

On day two, (with $U_1 = 90$) the Uranium further decays to $U_1 \times \alpha = 81$ grams, this time losing $U_1 \times (1 - \alpha) = U_0 \times \alpha \times (1 - \alpha) = n \times \alpha = 9$ grams of its mass. On day three, it further decays by $n \times \alpha^2 = 7.2$ grams, and so on. The decay process is allowed to continue until we have less than β grams of Uranium remaining.

Continuing with the Uranium analogy, three questions that are relevant to our problem of maintaining very large samples from a data stream are:

- What is the amount of Uranium lost on any given i th day?
- How can the initial mass of Uranium, 100 grams, can be expressed in terms of n and α ?
- How many days it will take for us before we are left with β grams or less of Uranium?

These questions can be answered using the following three simple observations related to geometric series:

Observation 1: Given a retention rate $\alpha < 1$ and n to be the first term of a geometric series, the i th term is given by $n \times \alpha^{i-1}$ for any $n \in \mathbb{R}$.

Observation 2: Given a retention rate $\alpha < 1$, it holds that $\sum_{i=1}^{\infty} n \times \alpha^{i-1} = \frac{n}{1-\alpha}$ for any $n \in \mathbb{R}$.

Observation 3: Given a retention rate $\alpha < 1$, define $f(j)$ as $\frac{n}{1-\alpha} \times \alpha^j$. From Observation 2, it follows that the largest j such that $f(j) \geq \beta$ is $j = \lfloor \frac{\log \beta - \log n + \log(1-\alpha)}{\log \alpha} \rfloor$. We denote this floor by Ψ .

To relate this back to the task of reservoir sampling, imagine that our large, disk-based reservoir sample R is maintained using a reservoir sampling algorithm in conjunction with a main memory buffer B (as in Algorithm 2). Recall that the way reservoir sampling works is that new samples from the data stream are chosen to overwrite random samples currently in the reservoir. The buffer temporarily stores these new samples, delaying the overwrite of a random set of records that are already stored on disk. Once the buffer is full, all new

samples are merged with the R by overwriting a random subset of the existing samples in R .

Consider some arbitrary subsample S of R (so $S \subset R$), with capacity $|S|$. Since the buffer B represents the samples that have already over-written the equal number of records of R , a buffer flush overwrites exactly $|B|$ samples of R . Thus, on expectation the merge will overwrite $\frac{|S| \times |B|}{|R|}$ samples of S . If we define $\frac{|B|}{|R|} = 1 - \alpha$, then on expectation, S should lose $|S| \times (1 - \alpha)$ of its own records due to the buffer flush¹. We refer this loss as *subsample decay*.

We can roughly describe the expected decay of S after repeated buffer merges using the three observations stated before. If the subsample retention rate $\alpha = 1 - \frac{|B|}{|R|}$, then:

- From Observation 1, it follows that the i th buffer merge, on expectation, removes $n \times \alpha^{i-1}$ samples from what remains of S .
- From Observation 2, it follows that the initial size of a subsample $|S| = \frac{n}{1-\alpha}$.
- From Observation 3, it follows that the expected number of merges required until S has β or less samples left is Ψ .

The net result of this is that it is possible to characterize the expected decay of any arbitrary subset of the records in our disk-based sample as new records are added to the sample through multiple emptyings of the buffer. If we view S as being composed of Ψ on-disk “segments” of exponentially decreasing size, plus a special, a single group of final segments of total size β that are buffered in main memory (subsequently referred to as the “beta segment”), then the i th buffer flush into R will on expectation overwrite exactly one on-disk segment from S . S loses an additional segment with every buffer flush until the subsample has only its beta segment remaining. At the point that only the subsample’s beta segment remains, the samples contained therein can be replaced directly. The reason that the beta segment is buffered in main memory is that overwriting a segment requires at least one random disk head movement, which is costly. By storing the beta segment in main memory, we can reduce the number of disk head movements with little main-memory storage cost. The process is depicted in Figure 1.

4.3 Geometric File Organization

This decay process suggests a file organization for efficiently maintaining very large random samples from a data stream. Let a subsample S be the set of records

¹ Actually, this is only a fairly tight approximation to the expected rate of decay. It is not an exact characterization because these expressions treat the emptying of the buffer into the reservoir as a single, atomic event, rather than a set of individual record additions (See Section 4.4).

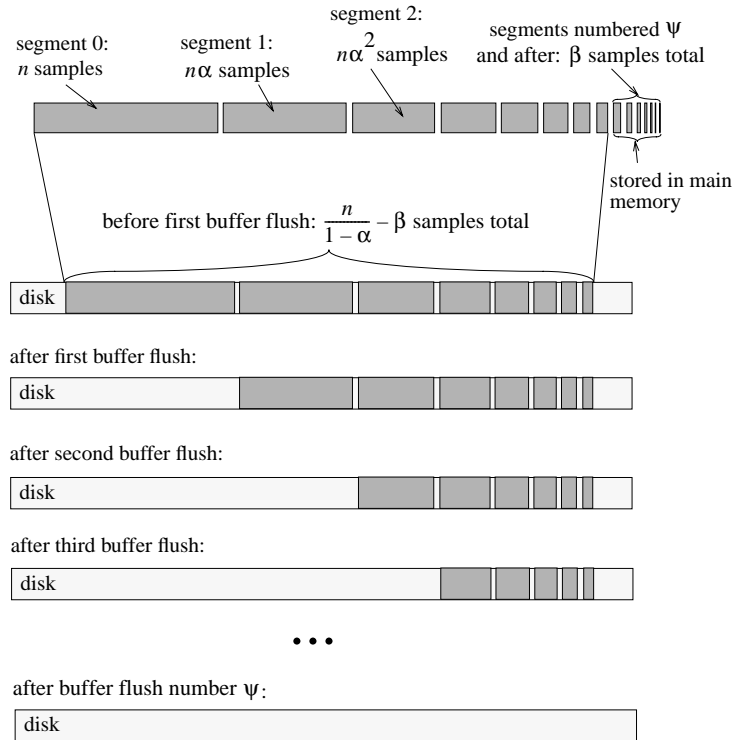


Fig. 1 Decay of a subsample after multiple buffer flushes.

that are loaded into our disk-based reservoir sample R in a single emptying of the buffer. Since we know that the number of records that remain in S will on expectation decay over time as depicted in Figure 1, we can organize our large, disk based sample as a set of decaying subsamples. At any point of time, the *largest* subsample was created by the most recent flushing of the buffer into R , and has not yet lost any segments. The *second largest* subsample was created by the second most recent buffer flush; it lost its largest segment in the most recent buffer flush. In general, the i th largest subsample was created by the i th most recent buffer flush, and it has had $i - 1$ segments removed by subsequent buffer flushes. The overall file organization is depicted in Figure 2.

4.4 Reservoir Sampling With a Geometric File

Given this organization, processing a buffer flush becomes an easy task. The overall reservoir sampling algorithm for the geometric file organization is given as Algorithm 3. The terms n , α , and Ψ carry the meaning discussed in Section 4.2. This process described by Algorithm 3 is depicted graphically in Figure 3.

This file organization has several significant benefits for use in maintaining a very large sample from a data stream:

- Performing a buffer flush requires *absolutely no reads from disk*.
- Each buffer flush requires only Ψ random disk head movements; all other disk I/Os are sequential writes. To add the new samples from the buffer into the geometric file to create a new subsample S , we need only seek to the position that will be occupied by each of S 's on-disk segments.
- Even if segments are not block-aligned, only the first and last block in each over-written segment must be read and then re-written (to preserve the records from adjacent segments).

4.4.1 Introducing the Required Randomness

One issue that needs to be addressed is the partitioning the buffer into segments in Algorithm 3 Step (21). In order to maintain the algorithm's correctness, when the buffer is flushed to disk it must overwrite a truly random subset of the records on disk. Thus, when performing the flush, we need to randomly choose records from the reservoir to replace. This implies that the on-disk subsamples (which are expectedly of size $\frac{n}{1-\alpha}$, $\frac{n\alpha}{1-\alpha}$, $\frac{n\alpha^2}{1-\alpha}$, and so on) will lose around $n, n\alpha, n\alpha^2$ records, and so on, respectively. However, while the number of records replaced in a subsample S will *on expectation* be proportional to the

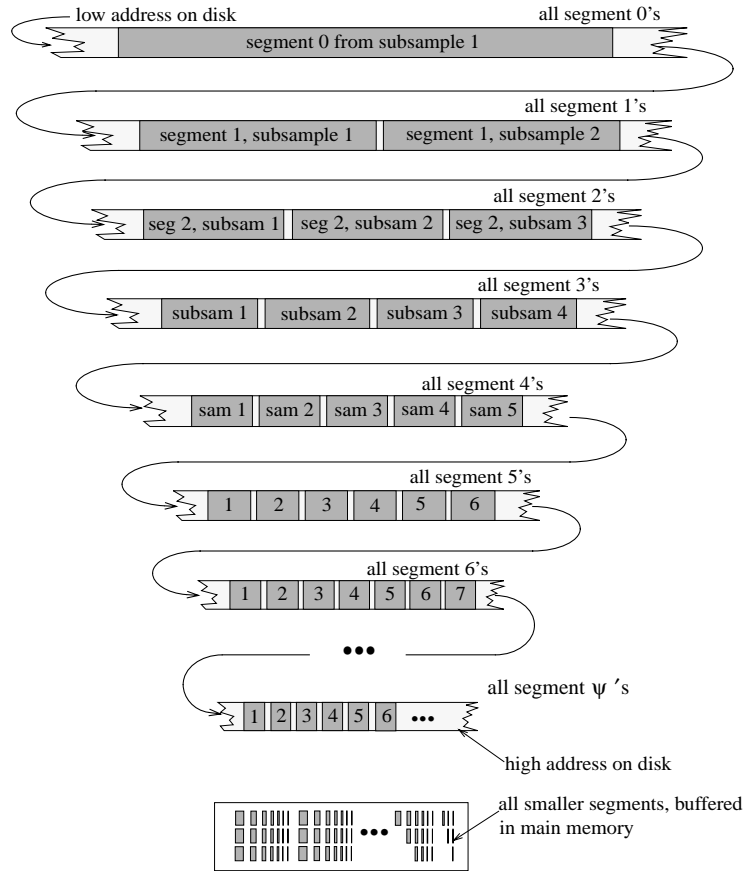


Fig. 2 Basic structure of the Geometric File.

size of S (and hence equal to the size of S 's largest on-disk segment) this replacement must be performed in a randomized fashion. The situation can be illustrated as follows. Say we have a set of numbers, divided into three buckets, as shown in Figure 4. Now, we want to add five additional numbers to our set, by randomly replacing five existing numbers. While we do expect numbers to be replaced in a way that is proportional to bucket size (Figure 4 (b)), this is not always what will happen (Figure 4 (c)).

In order to correctly introduce this variance into the geometric file, we need to add a few additional steps to Algorithm 3. Before we add a new subsample to disk via a buffer flush in Step (21), we first perform a logical, randomized partitioning of the buffer into segments, described by Algorithm 4. In Algorithm 4, each newly-sampled record is randomly assigned to replace a sample from an existing, on-disk subsample so that the probability of each subsample losing a record is proportional to its size. The result of Algorithm 4 is an array of M_i values, where M_i tells Step (21) of Algorithm 3 how many records should be assigned to overwrite the i th on-disk subsample.

4.4.2 Handling the Variance

Of course, there is no guarantee that $M_1 = n, M_2 = n\alpha, M_3 = n\alpha^2$, and so on, so there is no guarantee that Algorithm 3 will overwrite exactly the number of records contained in each subsample's largest segment. To handle this problem, we associate a stack (or buffer¹) with each of the subsamples. The stack associated with a subsample will buffer any of a subsample's records that logically should not have been over-written during buffer flush into the subsample (because M_i for some buffer flush for that subsample was smaller than expected), but whose space had to be claimed by the buffer flush in order to write a new subsample to disk. If the size of the stack is positive, it means that the corresponding subsample is larger than expected because it has had fewer of its records over-written than expected. We also allow a negative stack size. This simply means that some of the subsample's records *should* have been over-written but were not, because an M_i value for that subsample was

¹ We use the term "stack" rather than "buffer" to clearly differentiate the extra storage associated with each subsample from the buffer B .

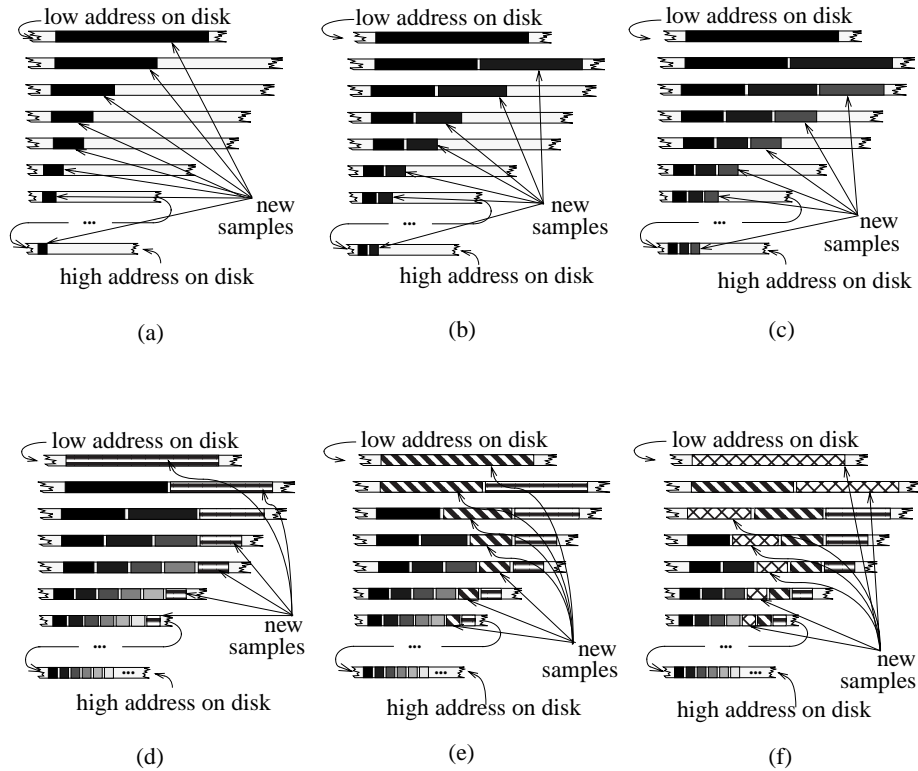


Fig. 3 Building a geometric file. First, the file is filled with the initial data produced by the stream (a through c). To add the first records to the file, the buffer is allowed to fill with samples. The buffered records are then randomly grouped into segments, and the segments are written to disk to form the largest initial subsample (a). For the second initial subsample, the buffer is only allowed to fill to $|B|\alpha$ of its capacity before being written out (b). For the third initial subsample, the buffer fills to $|B|\alpha^2$ of its capacity before it is written (c). This is repeated until the reservoir has completely filled (as was shown in Figure 2). At this point, new samples must overwrite existing ones. To facilitate this, the buffer is again allowed to fill to capacity. Records are then randomly grouped into segments of appropriate size, and those segments overwrite the largest segment of each existing subsample (d). This process is then repeated indefinitely, as long as the stream produces new records (e and f).

larger than expected. A stack size of $-k$ means that k of the subsample's on-disk records logically are not part of the reservoir (even though they are physically present on disk), and should be ignored during query processing.

Making use of the set of stacks is fairly straightforward. Imagine that $n\alpha^{(i-1)}$ of a buffer's records are sent to overwrite a segment from an existing subsample S_i , but according to Algorithm 4, M_i should have been. Then, there are two possible cases:

- Case 1: M_i is smaller than $n\alpha^{(i-1)}$ by some number of records ε . In this case, ε records are removed from the segment that is about to be over-written and pushed onto S_i 's stack in order to buffer them. This is necessary because these records logically should not be over-written by the records that are going to be added to the disk, but they will be.
- Case 2: M_i is larger than $n\alpha^{(i-1)}$ by some number of records ε . In this case, ε records are popped off of S_i 's stack to reflect the additional records that should have been removed from S_i , but were not.

These stack operations are performed just prior to Step (23) in Algorithm 3. Note that since the final group of segments from a subsample of total size β are buffered in main memory, their maintenance does not require any stack operations. Once a subsample has lost all of its on-disk samples, overwrites of records in this set can be handled by simply replacing the records directly.

4.4.3 Bounding the Variance

Because the stacks associated with each subsample will be used with high frequency as insertions are processed, each stack must be maintained with extreme efficiency. Writes should be entirely sequential, with no random disk head movements. To assure this efficiency and avoid any sort of online reorganization, it is desirable to pre-allocate space for each of the stacks on disk.

To pre-allocate space for these stacks, we need to characterize how much overflow we can expect from a given subsample, which will bound the growth of the

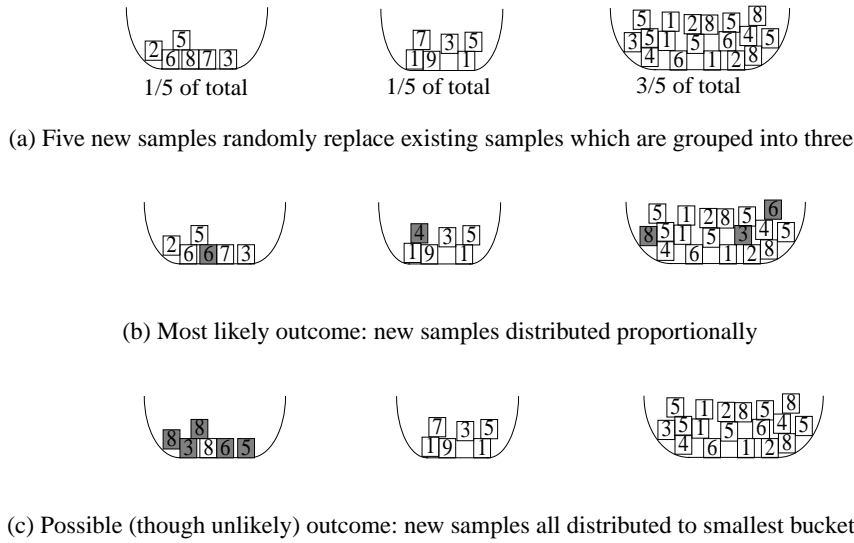


Fig. 4 Distributing new records to existing subsamples.

Algorithm 3 Reservoir Sampling with a Geometric File

```

1: Set  $numSubsamples = 0$ 
2: for int  $i = 1$  to  $\infty$  do
3:   Wait for a new record  $r$  to appear in the stream
4:   if  $i \leq |R|$  then
5:     Add  $r$  to  $B$ 
6:     if  $Count(B) == |B|\alpha^{numSubsamples}$  then
7:       Randomize the ordering of the records in  $B$ 
8:       Set  $n = Count(B) \times (1 - \alpha)$ 
9:       Partition  $B$  into segments of size  $n, n\alpha, n\alpha^2$ , and
10:      so on
11:      Flush the first  $\Psi$  segments to the disk
12:      Store the group of remaining segments in main
13:      memory
14:       $numSubsamples ++$ 
15:       $B = \emptyset$ 
16:   else
17:     with probability  $|R|/i$  do
18:       with probability  $Count(B)/|R|$  do
19:         Replace a random record in  $B$  with  $r$ 
20:       else do
21:         Add  $r$  to  $B$ 
22:       if  $Count(B) == |B|$  then
23:         Partition the buffer into segments of size
24:          $n, n\alpha, n\alpha^2$ , and so on (see Section 4.4.1)
25:         for each segment  $sg_j$  from  $B$  do
26:           Overwrite the largest segment of  $j$ th largest
27:           subsample of  $R$  with  $sg_j$ 
28:          $B = \emptyset$ 

```

Algorithm 4 Randomized Segmentation of the buffer

```

1: for each subsample  $i$  in the reservoir  $R$  do
2:   Set  $N_i =$  Number of records in  $S_i$ 
3:   Set  $M_i = 0$ 
4: for each record  $r$  in the buffer  $B$  do
5:   Randomly choose a victim subsample  $S_i$  such that
6:    $\Pr[choosing\ S_i] = N_i / \sum_{s_j} N_j$ 
7:    $N_i --$ ;  $M_i ++$ 

```

subsample's stack. It is important to have a good characterization of the expected stack growth. If we allocate *too much* space for the stacks, then we allocate disk space for storage that is never used. If we allocate *too little* space, then the top of one stack may grow up into the base of another. If a stack does overflow, it can be handled by buffering the additional records temporarily in memory or moving the stack to a new location on disk until the stack can again fit in its allocated space. This is not a catastrophic event, but it increases the disk I/O associated with stack maintenance and leads to fragmentation, and so it is an event that we would like to render very rare.

To avoid this, we observe that if the stack associated with a sub-sample S contains any samples at a given moment, then S has had fewer of its own samples removed than expected. Thus, our problem of bounding the growth of S 's stack is equivalent to bounding the difference between the expected and the observed number of samples that S loses as $|B|$ new samples are added to the reservoir, over all possible values for $|B|$.

To bound this difference, we first note that after adding $|B|$ new samples into the reservoir, the probability that any existing sample in the reservoir has been over-written by a new sample is $1 - \left(1 - \frac{1}{|R|}\right)^{|B|}$. During the addition of new records to the reservoir, we can view a subsample S of initial size $|B|$ as a set of $|B|$ identical, independent Bernoulli trials (coin flips). The i th trial determines whether the i th sample was removed from S . Given this model, the number of samples remaining in S after $|B|$ new samples have been added to the reservoir is binomially distributed with $|B|$ trials and $P = \Pr[s \in S\ remains] = 1 - \left(1 - \frac{1}{|R|}\right)^{|B|}$. Since we are interested in

characterizing the variance in the number of samples removed from S primarily when $|B|P$ is large, the binomial distribution can be approximated with very high accuracy using a normal distribution with mean $\mu = |B|P$ and standard deviation $\sigma = \sqrt{|B|P(1-P)}$ [29]. Simple arithmetic implies that the greatest variance is achieved when a subsample has on expectation lost 50% of its records to new sample ($P = 0.5$); at this point the standard deviation σ is $0.5\sqrt{|B|}$. Since we want to ensure that stack overruns are essentially impossible, we choose a stack size of $3\sqrt{|B|}$. This allows the amount of data remaining in a given subsample to be up to six standard deviations from the norm without a stack overflow, and is not too costly an additional overhead. A quick lookup in a standard table of normal probabilities tells us that this will yield only around a 10^{-9} probability that any given subsample overflows its stack. While achieving such a small probability may seem like overkill, it is important to remember that many thousands of subsamples may be created in all during the life of the geometric file, and we want to ensure that very few of them overflow their respective stacks. If 100,000 on-disk segments are replaced, then using a stack of size $3\sqrt{|B|}$ will yield a very reasonable probability that we experience no overflows of $(1 - 10^{-9})^{100,000}$, or 99.990%. In practice, the actual probability of experiencing no overflows will be even greater. This is due to the fact that the standard deviation in subsample size for most of a subsample's lifespan will be much less than $0.5\sqrt{|B|}$, due to the high percentage of its lifespan that it has an associated P of less than 0.5 as it slowly loses all of its samples.

5 Choosing Parameter Values

Given a specified file size and buffer size, two parameters associated with using the geometric file must be chosen: α , which is the fraction of a subsample's records that remain after the addition of a new subsample, and β , which is the total size of a subsample's segments that are buffered in memory.

5.1 Choosing a Value for Alpha

In general, it is desirable to minimize α . Decreasing α decreases the number of segments used to store each subsample. Fewer segments means fewer random disk head movements are required to write a new subsample to disk, since each segment requires around four disk seeks to write (one to read the location and one to write a new segment, and similarly two more considering the cost of subsequently adjusting the stack of the previous owner).

To illustrate the importance of minimizing α , imagine that we have a 1GB buffer and a stream producing 100B records, and we want to maintain a 1TB sample. Assume that we use an α value of 0.99. Thus, each subsample is

originally 1GB, and $|B| = 10^7$. From Observation 2 we know that $\frac{n}{1-\alpha}$ must be 10^7 , so we must use $n = 10^5$. If we choose $\beta = 320$ (so that β is around the size of one 32KB disk block), then from Observation 3 we will require $\lfloor \frac{\log 320 - \log 10^5 + \log(1-0.99)}{\log 0.99} \rfloor = 1029$ segments to store the entire new subsample.

Now, consider the situation if $\alpha = 0.999$. A similar computation shows that we will now require 10,344 segments to store the same 1GB subsample. This is an order-of-magnitude difference, with significant practical importance. With four disk seeks per segment, 1029 segments might mean that we spend around 40 seconds of disk time in random I/Os (at 10ms each), whereas 10,344 might mean that 400 seconds of disk time is spent on random disk I/Os. This is important when one considers that the time required to write 1GB to a disk sequentially is only around 25 seconds. While minimizing α is vital, it turns out that we do not have the freedom to choose α . In fact, to guarantee that the sum of all existing subsamples is $|R|$, the choice of α is governed by the ratio of $|R|$ to the size of the buffer $|B|$:

Lemma 1.

(The size of a geometric file is $|R|$) \Leftrightarrow $(1 - \alpha) = \frac{|B|}{|R|}$

Proof. In the proof, (and consequently the Lemma) we ignore the fact that $|B| \times \alpha^{i-1}$ may not be integral, we also ignore the storage associated with auxiliary structures such as the stacks and the beta segments. In this case, the geometric file is simply collection of subsamples of decaying size. We know that the largest subsample on disk is created by the most recent buffer flush and has $|B|$ records in it. From Observation 1 the size of the i th subsample of a file is $|B| \times \alpha^{i-1}$. It then follows from Observation 2 that the total size of all subsamples of a geometric file is $\sum_{i=1}^{\infty} |B| \times \alpha^{i-1} = \frac{|B|}{1-\alpha}$, and thus $(1 - \alpha) = \frac{|B|}{|R|}$. \square

We will address this limitation in Section 6.

5.2 Choosing a Value for Beta

It turns out that the choice of β is actually somewhat unimportant, with far less impact than α . For example, if we allocate 32KB for holding our β in-memory samples for each subsample, and $|B|/|R|$ is 0.01, then as described above, adding a new subsample requires that 1029 segments be written, which will require on the order of 1029 seeks. Redoing this calculation with 1MB allocated to buffer samples from each on-disk subsample, the number of on-disk segments is $\lfloor \frac{\log 10^4 - \log 10^5 + \log(1-0.99)}{\log 0.99} \rfloor$ or 687. By increasing the amount of main memory devoted to holding the smallest segments for each subsample by a factor of 32, we are able to reduce the number of disk head movements by less than a factor of two. Thus, we

will not consider optimizing β . Rather, we will fix β to hold a set of samples equivalent to the system block size, and search for a better way to increase performance.

6 Speeding Things Up

As discussed above, the value of α can have a significant effect on geometric file performance. If $\alpha = 0.999$, we can expect to spend up to 95% of our time on random disk head movements. However, if we were instead able to choose $\alpha = 0.9$, then we reduce the number of disk head movements by factor of 100, and we would spend only a tiny fraction of the total processing time on seeks. Unfortunately, as things stand, we are not free to choose α . According to Lemma 1, α is fixed by the ratio $|B|/|R|$. That is, for a fixed desired size of reservoir we need a larger buffer to lower the value of α .

However, there is a way to improve the situation. Given a buffer of fixed capacity $|B|$ and desired sample size $|R|$, we choose a smaller value $\alpha' < \alpha$, and *then maintain more than one geometric file at the same time to achieve a large enough sample*. Specifically, we need to maintain $m = \frac{(1-\alpha')}{(1-\alpha)}$ geometric files at once. These files are identical to what we have described thus far, except that the parameter α' is used to compute the sizes of a subsample's on-disk segments and size of each file is $\frac{|R|}{m}$. The remainder of this Section describes the details of how multiple geometric files are used to achieve greater efficiency.

6.1 Reservoir Sampling with Multiple Geometric Files

The reservoir sampling algorithm with multiple geometric files is similar to the Algorithm 3. Each of the m geometric files is still treated as a set of decaying subsamples, and each subsample is partitioned into a set of segments of exponentially decreasing size, just as is done in Algorithm 3, Steps (5)-(13). The only difference is that as each file is created, the parameter α' is used instead of α in Steps (6), (8)-(9), and each of the m geometric files is filled after one another, in turn. Thus, each subsample of each geometric file will have segments of size $n, n\alpha', n\alpha'^2$ and so on.

Algorithm 5 Randomized Segmentation of the Buffer for Multiple Geometric Files

- 1: **for** each S_{ij} , the i th subsample in j th file **do**
 - 2: Set N_{ij} = Number of records in S_{ij}
 - 3: Set $M_{ij} = 0$
 - 4: **for** each record r in the buffer B **do**
 - 5: Randomly choose a victim subsample S_{ij} such that
 $\Pr[\text{choosing } ij] = N_{ij} / \sum_{s_{kl}} N_{kl}$
 - 6: $N_{ij} - -$; $M_{ij} + +$
-

However, processing additional records from the stream is somewhat different. As more and more records are produced by the stream, new samples are captured and are added to the buffer exactly as in Algorithm 3 Steps (15)-(20) until buffer is full. Once the buffer is full, its record order is then randomized, just as is in a single geometric file. Next the buffer is flushed to disk. This is where the algorithm is modified. Overwriting records on disk with records from the buffer is somewhat different, in two primary ways:

- *Partitioning the buffer*: In Algorithm 4, the buffer is partitioned so that the size of each buffer segment is on expectation proportional to the current size of subsamples in a single file. In case of multiple geometric files, we partition the buffer just like in Algorithm 4; however, we randomly partition the buffer across *all* subsamples from *all* geometric files. The number of buffer segments after the partitioning is the same as the total number of subsamples in the entire reservoir, and the size of each buffer segment is on expectation proportional to the current size of each of the subsamples from one of the geometric files. This allows us to maintain the correctness of the reservoir sampling algorithm. The buffer partitioning steps in case of multiple geometric files are given in Algorithm 5.
- *Merging buffer segments with multiple geometric files*: This step requires quite a different approach compared to Algorithm 3's buffer merge algorithm. We discuss all the intricacies subsequently, but at high-level, the largest segment of each subsample from *only one* geometric file is over-written with samples from the buffer. This allows for considerable speedup, as we discuss in Section 6.2. At first, this would seem to compromise the correctness of the algorithm: logically, the buffered samples must over-write samples from *every one* of the geometric files (in fact, this is precisely why the buffer is partitioned across all geometric files, as described in the previous bullet). However, the correctness can be maintained by making use of some additional buffer space. In Sections 6.1.1 to 6.1.3, we describe in detail an algorithm that is able to maintain the correctness of the sample.

6.1.1 Consolidation And Merging

As stated previously, the process of flushing a buffer to disk once it has been partitioned must be altered. The first step in flushing the buffer to disk is the *consolidation* of the many small buffer segments that result from partitioning the buffer across all files to form larger segments that are then used to over-write segments in only a single geometric file. To form the largest consolidated segment, we group the m buffer segments assigned to the largest subsample from every file. The next largest consolidated segment is formed by grouping the m buffer

segments corresponding to the next largest subsample across every files, and so on.

Once the segments assigned to the various files have been consolidated, the resulting segments are used to overwrite subsamples from a single geometric file using exactly the algorithm from Section 4, subject to the constraint that the j th buffer merge overwrites subsamples from the $(j \bmod m)$ th geometric file.

6.1.2 How Can Correctness Be Maintained?

Logically, samples from the buffer have been partitioned so as to preserve the correctness of the reservoir algorithm: each record has been assigned to a subsample with probability proportional to the subsample's size. However, the fact that these partitions are then consolidated and merged into a single subsample would seem to compromise algorithm correctness, since the subsamples in the $(j \bmod m)$ th geometric file are over-written with too many new samples. Thus, this file physically loses many of its samples before it should. This results in a subsample with fewer samples stored on disk than it should have in order to preserve the correctness of the reservoir sampling algorithm.

Our remedy to this problem is to delay overwriting a subsample's largest segment until the time that all (or most) of the records that will be over-written on disk are invalid, in the sense that they have logically been "over-written" by having records from subsequent buffer flushes assigned to replace them. In order to accomplish this, we note that if we did not perform consolidation and instead replaced a segment from each subsample with exactly those records assigned to overwrite records from that subsample, then on expectation a subsample would lose all of the records in its largest segment after m buffer flushes. Thus, if we somehow delay overwriting the largest segment in each file for m buffer flushes, we could sidestep the problem of losing too many records due to consolidation.

The way to accomplish this is to overwrite subsamples in a lazy manner. We merge the buffer with the $(j \bmod m)$ th geometric file, but we do not overwrite any of the valid samples stored in the file until the *next* time we get to the file. We can achieve this by allocating enough extra space in each geometric file to hold a complete, empty subsample in each geometric file. This subsample is referred to as the *dummy*. The dummy never decays in size, and never stores its own samples. Rather, it is used as a buffer that allows us to sidestep the problem of a subsample decaying too quickly. When a new subsample is added to a geometric file, the new subsample overwrites segments of dummy rather than overwriting largest segment of any existing subsamples. Thus, we have protected segments of subsamples that contain valid data by overwriting dummy's records instead.

When records are merged from the buffer into the dummy, the space previously owned by the dummy is

given up to allow storage of the file's newest subsample. After this flush, the largest segment from each of the subsamples in the file is given up to reconstitute the new dummy. Because the records in (new) dummy's segments will not be over-written until the next time that this particular geometric file is written to, all of the data that is contained within it is protected.

Note that with a dummy subsample, we no longer have a problem with a subsample losing its samples too quickly. Instead, a subsample may have slightly too many samples present on disk at any given time, buffered by the file's dummy. These extra samples can easily be ignored during query processing. The only additional cost we incur with dummy is that each of the geometric files on disk must have $|B|$ additional units of storage allocated. The use of a dummy subsample is illustrated in Figure 5.

6.1.3 Handling the Stacks in Multiple Geometric Files

One final issue that should be considered is maintenance of the stacks associated with each subsamples of the $(j \bmod m)$ th geometric file. Just as in the single file case, the purpose of the stack associated with a subsample is to store samples that are still valid, but whose space must be given up in order to store new samples from the buffer that have been flushed to disk. With multiple geometric files, this does not change. It is possible that when the buffer is written to the dummy subsample in a file, the dummy may still contain valid samples from a subsample in that file. Specifically, one or more of the dummy's segments may contain valid samples from the last subsample to own the segment. In that case, the valid samples are saved to that subsample's stack before the dummy is over-written.

6.2 Speed-up Analysis

The increase in speed achieved using multiple geometric files can be dramatic. The time required to flush a set of new samples to disk as a new subsample is dominated by the need to perform random disk head movements. For each subsample, we need two random movements to overwrite its largest segment (one to read the location and one to write a new segment) and then two more seeks for its stack adjustment; a total of around 40 ms/segment. The number of segments required to write a new subsample to disk in the case of multiple geometric files (and thus the number of random disk head movements required) is given by Lemma 2.

Lemma 2.

Let $\omega = (\log(1/\alpha'))^{-1}$. Multiple geometric files can be used to maintain an online sample of arbitrary size with a cost of $O(\omega \times \log |B| / |B|)$ random disk head movements

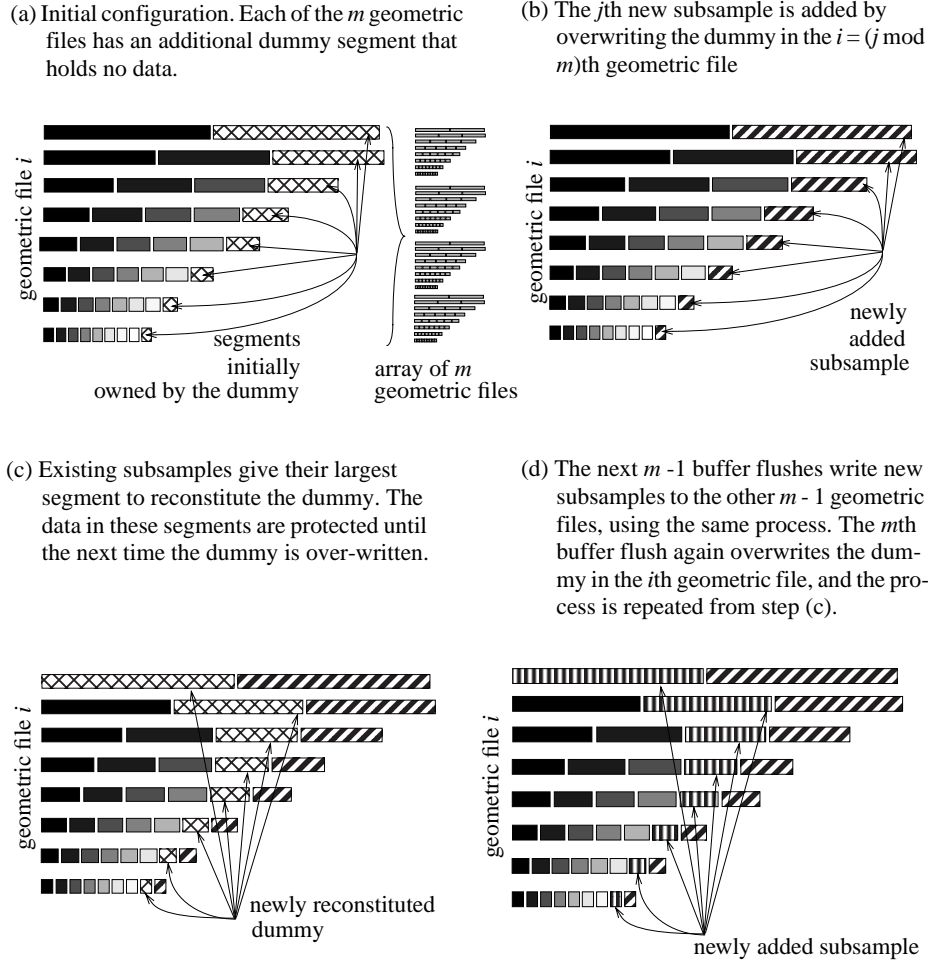


Fig. 5 Speeding up the processing of new samples using multiple geometric files.

for each newly sampled record.

Proof. We know that for every buffer flush, m segments in the buffer are grouped to form a consolidated segment. All such consolidated segments are then used to overwrite the largest on-disk segments of the subsamples stored in a single geometric file. From Observation 3, we know that the number on-disk segments of a subsample (and thus the number of consolidated segments) is $\lfloor \frac{\log \beta - \log n + \log(1 - \alpha')}{\log \alpha'} \rfloor$. Substituting $n = (1 - \alpha') \times |B|$ and simplifying the expression (as well as ignoring the floor) we compute the number of segments to write as $\frac{1}{\log \alpha'} (\log \beta - \log |B|)$. If we let $\omega = (\log(1/\alpha'))^{-1}$ the number of segments can be expressed as $\omega(\log |B| - \log \beta)$. Assuming a constant number c of random seeks per segment written to the disk, the total random disk head movements required per record is $\omega c ((\log |B| - \log \beta)/|B|)$, which is $O(\omega \times \log |B|/|B|)$. \square

In case of multiple geometric files we use additional space for m dummy subsamples. Thus, the total storage re-

quired by all geometric files is $|R| + (m \times |B|)$. If we wish to maintain a 1TB reservoir of 100B samples with 1GB of memory, we can achieve $\alpha' = 0.9$ by using only 1.1TB of disk storage in total. For $\alpha' = 0.9$, we need to write less than 100 segments per 1GB buffer flush. At 40 ms/segment, this is only 4 seconds of random disk head movements to write 1GB of new samples to disk.

7 Sampling the Sample

A geometric file is a simple random sample (without replacement) from a data stream. In this Section we develop techniques which allow a geometric file to *itself* be sampled in order to produce smaller sets of data objects that are themselves random samples (without replacement) from the original data stream. The goal of the algorithms described in this Section is to efficiently support further sampling of a geometric file by making use of its own structure.

7.1 Why Might We Need To Sample From a Geometric File?

In Section 2, we argued that small samples frequently do not provide enough accuracy, especially in the case when the resulting statistical estimator has a very high variance. However, while in the general case a very large sample can be required to answer a difficult query, a huge sample may often contain too much information. For example, reconsider the problem of estimating the average net worth of American households as described in Section 2. In the general case, many millions of samples may be needed to estimate the net worth of the average household accurately (due to a small ratio between the average household's net worth and the standard deviation of this statistic across all American households). However, if the same set of records held information about the size of each household, only a few hundred records would be needed to obtain similar accuracy for an estimate of the average size of an American household, since the ratio of average household size to the standard deviation of sample size across households in the United States is greater than 2. Thus, to estimate the answer to these two queries, vastly different sample sizes are needed.

7.2 Different Sampling Plans for the Geometric File

Since there is no single sample size that is optimal for answering all queries and the required sample size can vary dramatically from query to query, this Section considers the problem of generating a sample of size N from a data stream using an existing geometric file that contains a large sample of records from the stream, where $N \leq R$. We will consider two specific problems. First, we consider the case where N is known beforehand. We will refer to a sample retrieved in this manner as a batch sample. Batch samples of fixed size have been suggested for use in several approximate query processing applications [13, 14, 20, 18, 39]. In general, the drawback of making use of a batch sample is that the accuracy of any estimator which makes use of the sample is fixed at the time that the sample is taken, whereas the benefit of batch sampling is that the sample can be drawn with very high efficiency.

We will also consider the case where N is not known beforehand, and we want to implement an iterative function *GetNext*. Each call to *GetNext* results in an additional sampled record being returned to the caller, and so N consecutive calls to *GetNext* results in a sample of size N . We will refer a sample retrieved in this manner as an *online* or *sequential sample*. The drawback of online sampling compared to batch sampling is that it is generally less efficient to obtain a sample of size N using online methods. However, since the consumer of the sample can call *GetNext* repeatedly until an estimator with enough accuracy is obtained, online sampling is more

flexible than batch sampling. An online sample retrieved from a geometric file can be useful for many applications, including *online aggregation* [22, 24]. In online aggregation, a database system tries to quickly gather enough information so as to approximate answer to an aggregate query. As more and more information is gathered, the approximation quality is improved, and the online sampling procedure is halted when the user is happy with the approximation accuracy.

7.3 Batch Sampling From a Geometric File

7.3.1 A Naive Algorithm

The most obvious way to implement batch sampling is to make use of the reservoir sampling algorithm to raw a sample of size N from a geometric file of size $|R|$ in a single pass. As the following lemma asserts, the resulting sample is also a sample of size N from the original data stream.

Lemma 3.

The reservoir sampling algorithm over a geometric file produces a correct random sample of the stream.

Proof. If S is the batch sample of size N retrieved from a geometric file R of size $|R|$ using the reservoir sampling algorithm, then we know from the correctness of the reservoir sampling algorithm that:

$$\begin{aligned} Pr[S \in R] &= \frac{\# \text{ of subsets of size } N \in R}{\# \text{ of such subsets in a data stream } D} \\ &= \frac{\binom{|R|}{N}}{\binom{|D|}{N}} \end{aligned}$$

Now, imagine that $S \in R$. If we obtain a sample of size N from R using the reservoir algorithm, the probability that we choose precisely S is:

$$Pr[S \text{ sampled from } R | S \in R] = 1 / \binom{|R|}{N}$$

Thus we have:

$$\begin{aligned} Pr[S \text{ sampled from } R] &= Pr[S \text{ sampled from } R | S \in R] \\ &\quad \times Pr[S \in R] \\ &= \frac{1}{\binom{|R|}{N}} \frac{\binom{|R|}{N}}{\binom{|D|}{N}} \\ &= \frac{1}{\binom{|D|}{N}} \end{aligned}$$

This is precisely the probability we would expect if we sampled directly from the stream without replacement. \square

Unfortunately, though it is very simple, the naive algorithm will be inefficient for drawing a small sample from a large geometric file since it requires a full scan of the geometric file to obtain a true random sample for *any* value of N . Since the geometric file may be gigabytes in size, this can be problematic.

7.3.2 A Geometric File Structure Based Algorithm

We can do better if we make use of the structure of a geometric file itself. The intuitive outline of this approach is as follows. To obtain a batch sample of size N , we pre-calculate how many records from each on-disk subsample will be included in the batch sample, and then we read the appropriate number of records sequentially from the various segments of each subsample. The process of choosing the number of records to select from each subsample is analogous to Olken and Rotem’s procedure for choosing the number of records to select from each hash bucket when performing batched sampling from a hashed file [33]. Once the number of sampled records from each segment has been determined, sampling those records can be done with an efficient sequential read since within each on disk segment, all records are store in a randomized order. The key algorithmic issue is how to calculate the contribution of each subsample. Since this contribution is a multivariate hypergeometric random variable, we can use an approach analogous to Algorithm 4, which is used to partition the buffer to form the segments of a subsample. In other words, we can view retrieving N samples from a geometric file analogous to choosing N random records to overwrite when new records are added to the file.

The resulting algorithm can be described as follows. To start with, we partition the sample space of N records into segments of varying size exactly as in Algorithm 4. We refer to these segments of the sample space as *sampling segments*. The sampling segments are then filled with samples from the disk using a series of sequential reads, analogous to the set of writes that are used to add new samples to the geometric file. The largest sampling segment obtains all of its records from the largest subsample, the next largest sampling segment obtains all its record from second largest subsample, and so on.

When using this algorithm, some care needs to be taken when N approaches to the size of a geometric file. Specifically, when all disk segments of a subsample are returned to a corresponding sampling segment, we must also consider the subsample’s in-memory buffered records and any records contained in its stack in order to obtain desired size sample. The detailed algorithm is presented as Algorithm 6.

It is clear that this algorithm obtains the desired batch sample by scanning exactly N records as against the entire scan of the reservoir sampling at the cost of few random disk seeks. Since the sampling process is analo-

Algorithm 6 Batch Sampling a Geometric File

```

1: Set  $NS$  = Number of subsamples in a geometric file
2: for  $i = 1$  to  $NS$  do
3:   Set  $RecsInSubsam[i]$  = Size of  $i$ th subsample
4:   Set  $RecsToRead[i] = 0$ 
5: for  $i = 1$  to  $NS$  do
6:   Choose  $j$  such that  $\Pr[\text{choosing } j] = RecsInSubsam[j] / |R|$ 
7:    $RecsInSubsam[j] --$ 
8:    $RecsToRead[j] ++$ 
9: for  $i = 1$  to  $NS$  do
10:  Append to batchsample  $RecsToRead[i]$  records from the  $i$ th subsample

```

gous to the process of adding more samples to the file, it is just as efficient, requiring $O(\omega \times \log |B|/N)$ random disk head movements for each newly sampled record, as described in Lemma 2.

7.3.3 Batch Sampling Multiple Geometric Files

A geometric file structure based batch sampling algorithm can be extended to allow efficient batch sampling from multiple geometric files in the same way that the insertion algorithm for new samples into the geometric file can be extended to allow insertions into multiple geometric files. The extension is fairly straightforward with additional first step where we determine the number of records to be sampled from each geometric file. Once this number is determined, we execute Algorithm 6 on each file in order to obtain the desired batch sample.

7.4 Online Sampling From a Geometric File

7.4.1 A Naive Algorithm

One straightforward way of supporting online sampling from a geometric file is to implement the iterative function *GetNext* as follows. For every call to *GetNext*, we simply generate a random number i between 1 and size of the file $|R|$, and then return a record at the i th position in the geometric file. Care must be taken to avoid choosing same record of R more than once in order to obtain a correct sample without replacement. For example, to sample N records from R , the numbers 0 through $N - 1$ could be hashed or randomized using a bijective pseudo-random function onto the domain 0 through $|R| - 1$, and the resulting N numbers used to generate the sample. To pick the next record to sample, we simply hash N .

It is easy to see that a naive algorithm will give us a correct online sample of a geometric file. However, we will use one disk seek per call to *GetNext*. Since each random I/O requires around 10 milliseconds, the naive algorithm can only sample around 6,000 records from the geometric file per minute per disk. This performance is unacceptable for most applications.

7.4.2 A Geometric File Structure-Based Algorithm

As in the case of batch sampling algorithm, we can make use of the structure of a geometric file to efficiently support online sampling.

Instead of selecting a random record of a geometric file, we randomly pick a subsample and choose its next available record as a return value of *GetNext*. This is analogous to the classic online sampling algorithm for sampling from a hashed file [33], where first a hash bucket is selected and then a record is chosen. Since the selection of a random record within a subsample is sequential, we may reduce the number of costly disk seeks if we read the subsample in its entirety, and buffer the subsample's records in memory. Using this basic methodology, we now describe how a call to the *GetNext* will be processed:

- We first randomly pick a subsample S_i , with the probability of selecting i proportional to the size of i th subsample.
- Next, we look for buffered records of S_i ; if such records exist, we choose and return the first available record as the return value of *GetNext*. If no buffered records are found, we fetch and buffer a number of blocks of records from subsample S_i ; these records are then buffered. We return the first buffered record as the return value of *GetNext*.

Since the records from each subsample are read and buffered in memory sequentially, we are guaranteed to choose each record of the reservoir at most once, giving us desired random sample without replacement. A proof of this is simple, and analogous to the proof of Lemma 3. However, thus far we have not considered a very important question: How many blocks of a subsample S_i should we fetch at the time of buffer refill? In general there are two extremes that we may consider:

- *Fetch many.* If we fetch a large number of blocks at the time of the buffer refill, we reduce the overall time to sample N records for large N . This is due to the fact that by fetching many blocks using a sequential read, we amortize the seek time over a large number of blocks and at the same time we prepare ourselves for future calls to *GetNext*; once the records are fetched from disk, the response time for subsequent calls to *GetNext* is almost instantaneous (only in-memory computations are required). However, the drawback of this approach is that the more records we fetch sequentially from the disk during a single call to *GetNext*, the longer the response time will be for the particular call to *GetNext* during which we fetch those blocks. This is particularly worrisome if we spend a lot of time to fetch blocks which are never used (which will be the case if the user intends to draw only a relatively small-sized sample.)
- *Fetch few.* If we fetch small number of blocks at the time of buffer refills, we reduce the maximum response time for any given *GetNext* call. However, we

then need more seeks to sample N records. The approach can be problematic if user intends to draw a relatively large sample from the file.

In order to discuss such considerations more concretely, we note that the time required to process *GetNext* call is proportional to the number of blocks fetched on the call, assuming that the cost to perform the required in-memory calculations is minimal. If b blocks are fetched during a particular call, we spend $s + br$ time units on that particular call to *GetNext*, where s is the seek time and r is time required to scan a block. Once these b blocks are fetched we incur zero cost for next bn calls to *GetNext*, where n is the blocking factor (number of records per block). Thus, in the case where blocks are fetched at the first call to *GetNext*, we incur the total cost of $s + br$ to sample bn records, and have a response time of $s + br$ units at the first call to *GetNext*, with all subsequent calls having zero cost.

Now imagine that instead we split b blocks into two chunks of size $b/2$ each, and read a chunk-at-a-time. Thus, the first *GetNext* call will cost us $s + br/2$ time units. Once these $bn/2$ records are used up we read next chunk of blocks. The total cost in this scenario is $2s + br$ with a response time of $s + br/2$ time units once at the starting point and other mid-way through. Note that although the maximum response time on any call to *GetNext* is reduced by half, we required more time to sample bn records. The question then becomes, How do we reconcile response time with overall sampling time to give the user optimal performance?

The systematic approach we take to answering this question is based on minimizing the average square sum of response time over all *GetNext* calls. This idea is similar to the widely utilized sum-square-error or MSE criterion, which tries to keep the average error or "cost" from being too high, but also penalizes particularly poor individual errors or costs. However, one problem we face using this strategy in the context of online sampling is that we do not know before hand the value of N , the number of records to be sampled.

Algorithm 7 *GetNext* for Online Sampling

- 1: Set NS = Number of subsamples in a geometric file
 - 2: **for** $i = 1$ to NS **do**
 - 3: Set $RecsInSubsam[i]$ = Size of i th subsample
 - 4: Set $BufferedSubsamSize[i] = 0$
 - 5: Randomly choose a subsample S_i such that $\Pr[\text{choosing } i] = RecsInSubsam[i] / |R|$
 - 6: $RecsInSubsam[i] --$
 - 7: **if** $BufferedSubsamSize[i] == 0$ **then**
 - 8: Set $numRecs$ to minimum of sf/r and $RecsInSubsam[i]$
 - 9: Read and buffer $numRecs$ records of S_i
 - 10: $BufferedSubsamSize[i] = numRecs$
 - 11: $BufferSubsamSize[i] --$
 - 12: Return the next available buffered record of S_i
-

To address this issue, we use a simple heuristic. Every time we refill a buffer, we look at the number of records already sampled from a subsample and assume that the user will ask for the same number of samples as the algorithm progresses. This gives us the planning horizon for which we can determine the number of blocks to be fetched. We also use the obvious constraint that the total number of samples fetched from the subsample should not exceed the number of records in a subsample. Given this, an analytic solution to the problem of minimizing the average squared cost over all calls to *GetNext* is as follows:

- If there are b number of records per blocks then let N/b be the number of blocks in the planning horizon, and let X be the number of equal size chunks that we read on every buffer refill. Our goal is to determine the value of X and the number of blocks in each chunk.
- We know that the time to read a chunk is proportional to $s + (N/b \times r)/X$, and thus the square sum of response time of all *GetNext* calls is $X(s + (N/b \times r)/X)^2$.
- In order to derive a formula for the value of X that minimizes this, we simply differentiate it with respect to X and then solve for the zero.

$$\begin{aligned} &= \frac{d}{dX}(X(s + (N/b \times r)/X)^2) \\ &= \frac{d}{dX}(Xs^2 + 2Nsr + (N/b \times r)^2/X) \\ &= s^2 - (N/b \times r)^2/X^2 \end{aligned}$$

Setting this to zero, we have $X = Nr/bs$. Thus, we divide N/b blocks into Nr/bs chunks and read bs/r number of blocks from a subsample every time we refill the buffer. It turns out that when this solution is used, the number of blocks read at the time of buffer refill depends on the ratio of the seek time to the block scan time. Since this solution is independent of the planning horizon, we always read bs/r blocks irrespective of the number of records sampled so far. Algorithm 7 gives the detailed online sampling algorithm.

8 Benchmarking

In this Section, we detail two sets of benchmarking experiments. In the first set of experiments, we attempt to measure the ability of the geometric file to process a high-speed stream of data records. In the second set of experiments, we examine the various algorithms for producing smaller samples from a large, disk-based geometric file.

8.1 Processing Insertions

In order to test the relative ability of the geometric file to process a high-speed stream of insertions, we have implemented and benchmarked five alternatives for maintaining a large reservoir on disk: the three alternatives discussed in Section 3, the geometric file, and the framework described in Section 6 for using multiple geometric files at once. In the remainder of this Section, we refer to these alternatives as *the virtual memory*, *scan*, *local overwrite*, *geo file*, and *multiple geo files* options. An α' value of 0.9 was used for the *multiple geo files* option.

All implementation was performed in C++. Benchmarking was performed using a set of Linux workstations, each equipped with 2.4 GHz Intel Xeon Processors, 15,000 RPM, 80GB Seagate SCSI hard disks were used to store each of the reservoirs. Benchmarking of these disks showed a sustained read/write rate of 35-50 MB/second, and an ‘‘across the disk’’ random data access time of around 10ms.

8.1.1 Experiments Performed

The following three experiments were performed:

Insertion Experiment 1: The task in this experiment was to maintain a 50GB reservoir holding a sample of 1 billion, 50B records from a synthetic data stream. Each of the five alternatives was allowed 600MB of buffer memory to work with when maintaining the reservoir. For *the scan*, *local overwrite*, *geo file*, and *multiple geo files* options, 100MB was used as an LRU buffer for disk reads/writes, and 500MB was used to buffer newly sampled records before processing. The *virtual memory* option used all 600MB as an LRU buffer. In the experiment, a continual stream of records was selected to be inserted into the reservoir (as many as each of the five options could handle). The goal was to test how many new records could be added to the reservoir in 20 hours, while at the same time expelling existing records from the reservoir as is required by the reservoir algorithm. The number of new samples processed by each of the five options (that is, the number of records added to disk) is plotted as a function of time in Figure 6 (a). By ‘‘number of samples processed’’ we mean the number of records that are actually inserted into the reservoir, and not the number of records that have passed through the data stream.

Insertion Experiment 2: This experiment is identical to Experiment 1, except that the 50GB sample was composed of 50 million, 1KB records. Results are plotted in Figure 6 (b). Thus, we test the effect of record size on the five options.

Insertion Experiment 3: This experiment is identical to Experiment 1, except that the amount of buffer mem-

ory is reduced to 150MB for each of the five options. The *virtual memory* option used all 150MB for an LRU buffer, and the four other options allocated 100MB to the LRU buffer and 50MB to the buffer for new samples. Results are plotted in Figure 6 (c). This experiment tests the effect of a constrained amount of main memory.

8.1.2 Discussion of Experimental Results

All three experiments suggest that the *multiple geo files* option is superior to the other options. In Experiments 1 and 2, the *multiple geo files* option was able to write new samples to disk almost at the maximum sustained speed of the hard disk, at around 40 MB/sec.

It is worthwhile to point out a few specific findings. Each of the five options writes the first 50GB of data from the stream more or less directly to disk, as the reservoir is large enough to hold all of the data as long as the total is less than 50GB. However, Figure 6 (a) and (b) show that only the *multiple geo files* option does not have much of a decline in performance after the reservoir fills (at least in Experiments 1 and 2). This is why the *scan* and *virtual memory* options plateau after the amount of data inserted reaches 50GB. There is something of a decline in performance in all of the methods once the reservoir fills in Experiment 3 (with restricted buffer memory), but it is far less severe for the *multiple geo files* option than for the other options. Furthermore, this degeneration in performance could probably be reduced by using a smaller value for α' .

As expected, the *local overwrite* option performs very well early on, especially in the first two experiments (see Section 3 for a discussion of why this is expected). Even with limited buffer memory in Experiment 3, it uniformly outperforms a single geometric file. Furthermore, with enough buffer memory in Experiments 1 and 2, the *local overwrite* option is competitive with the *multiple geo files* option early on. However, fragmentation becomes a problem and performance decreases over time. Unless offline re-randomization of the file is possible periodically, this degradation probably precludes long-term use of the *local overwrite* option.

It is interesting that as demonstrated by Experiment 3 (and explained in Section 5) a single geometric file is very sensitive to the ratio of the size of the reservoir to the amount of available memory for buffering new records from the stream. The *geo file* option performs well in Experiments 1 and 2 when this ratio is 100, but rather poorly in Experiment 3 when the ratio is 1000.

Finally, we point out the general unusability of the *scan* and *virtual memory* options. *scan* generally outperformed *virtual memory*, but both generally did poorly. Except in experiment 1 with large memory and small record size, with these two options more than 97% of the processing of records from the stream occurs in the first half hour as the reservoir fills. In the 19.5 hours or so after the reservoir first fills, only a tiny fraction of ad-

ditional processing occurs due to the inefficiency of the two options.

8.2 Sampling from a Geometric File

We have also implemented and benchmarked four sampling techniques to sample geometric files discussed in the Section 7. Specifically, we have compared the naive batch sampling and the online sampling algorithms against a geometric file structure based batch sampling and online sampling algorithms. We have also tested these four techniques with the framework that make use of multiple geometric files. All of the algorithms were implemented on top of the geometric file prototype that was benchmarked in the previous Subsection.

8.2.1 Experiments Performed

To compare the various options, we used the following setup. We first initialize a geometric file by sampling and adding records from a synthesized data stream to the files for a period of several hours. This ensures a realistic scenario for testing: the reservoir in the file that is to be tested has been filled, a reasonable portion of each initial subsample has been over-written, and some of the smaller initial subsamples have been removed from the file, and a number of new subsamples have been create. The parameters used in building the geometric file are the same as those describe in Experiment 2 of the previous Subsection (a 50GB file with 50 million, 1KB records). Given such a file, the following set of experiments were performed:

Sampling Experiment 1: The goal of this experiment was to compare the two options for obtaining a batch sample from a geometric file: the naive algorithm, and then the geometric file structure based algorithm. For both algorithms, we plot the time to perform the sampling as a function of the desired sample size. Figure 7 (a) depicts the plot for a single geometric file; Figure 7 (b) shows an analogous plot for the multiple geometric files option.

Sampling Experiment 2: This experiment is analogous to Sampling Experiment 1, except that online sampling is performed via multiple successive calls to *GetNext*. The number of records sampled with multiple calls to *GetNext* versus the elapsed time is plotted in Figure 7 (c) for both the naive algorithm and the more advanced, geometric file structure based algorithm designed to increase the sampling rate and even out the response times. The analogous plot for multiple geometric file case is shown in Figure 7 (d). We also plot the variance in response times over all calls to *GetNext* as a function of the number of calls to *GetNext* in Figures 7(e) and 7(f) (the first is for a single geometric file; the

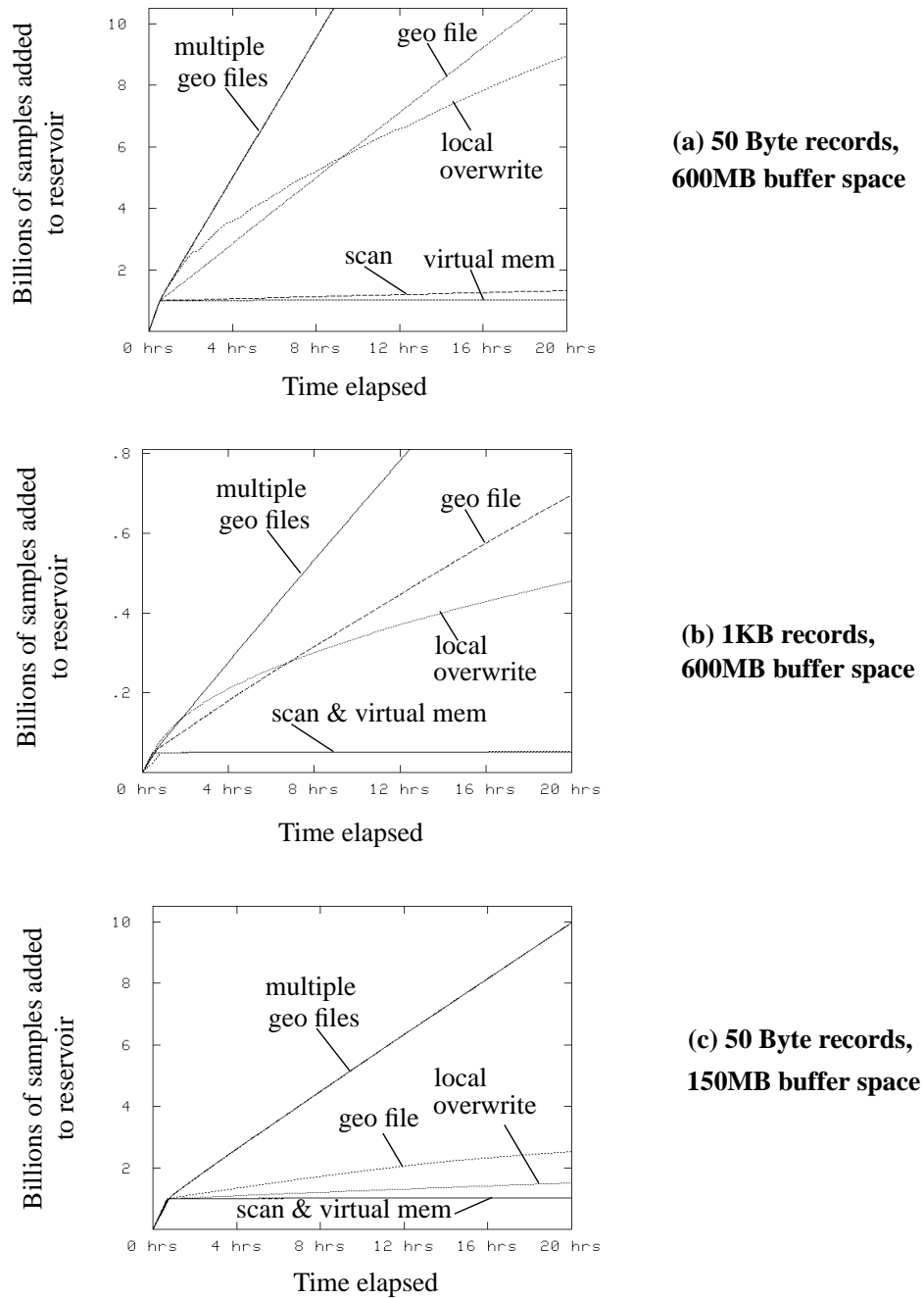


Fig. 6 Results of benchmarking experiments (Processing Insertions).

second is with multiple files). Taken together, these plots show the trade-off between overall processing time and the potential for waiting for a long time in order to obtain a single sample.

8.2.2 Discussion of Experimental Results

Not surprisingly, these results suggest that the geometric file structure based sampling methods are superior over

the more obvious naive algorithms, both in the batch and online case. As expected, the naive batch sampling algorithm took almost constant time to obtain batch sample of any size as it requires scan of the entire geometric file to retrieve any batch sample. The geometric file structure based algorithm can produce a small-size batch sample very fast, and the total sampling time increases linearly with sample size. The time required for the geometric file

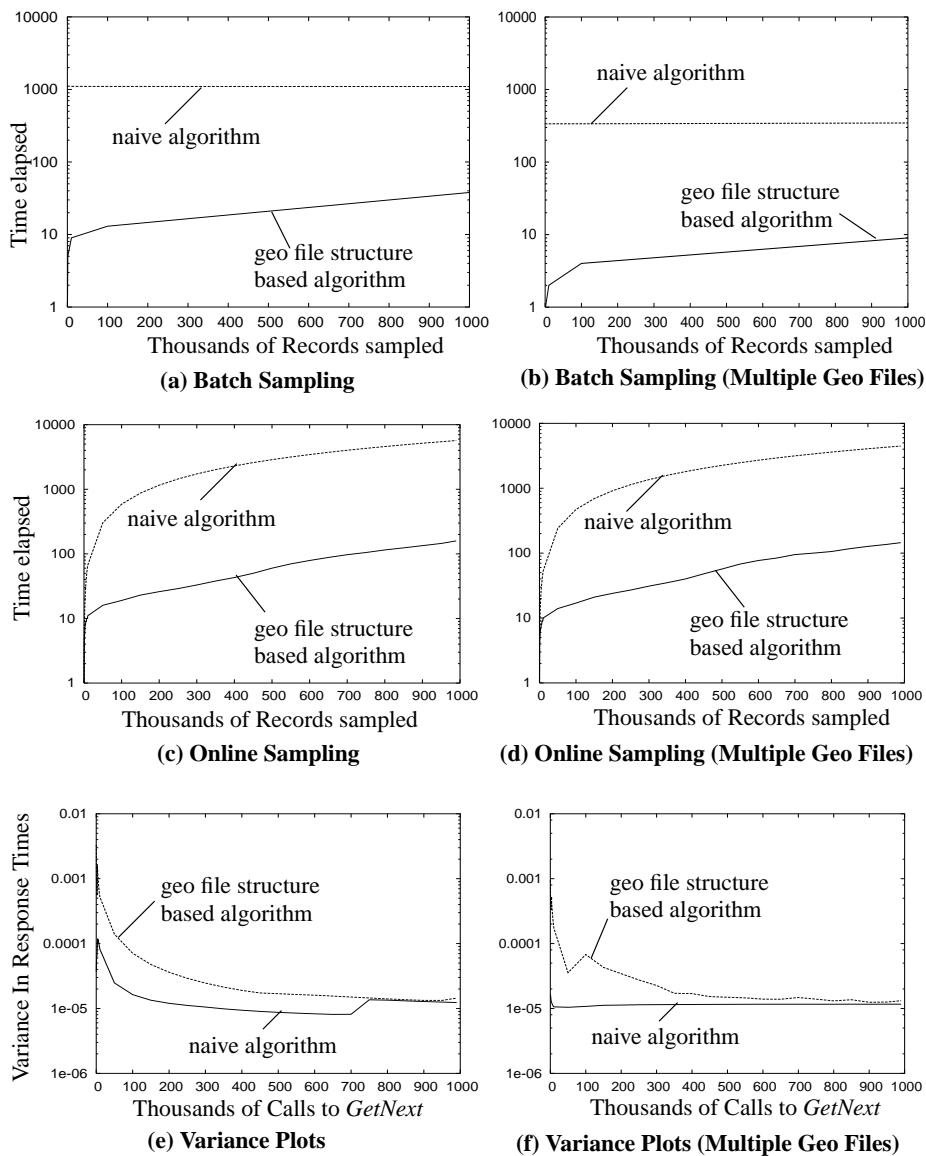


Fig. 7 Results of benchmarking experiments (Sampling from a Geometric file).

structure based algorithm is well below the time required by the naive approach even when 1/10 of file is sampled. In case of online sampling, geometric file structure based algorithm clearly outperformed naive approach and this was not surprising as it must expend one disk seek per sample. For both, batch and online sampling multiple geometric files framework showed results analogous to single geometric file case.

As expected and then demonstrated by variance plots, the variance of online naive approach is smaller than geometric file structure based algorithm. Although with this little larger variance (less than 10 times for 100k samples) in the response times, the structure based approach executed order of magnitude faster (more than 100 times for 100k samples) than the naive approach for any number of

records sampled, justifying our approach of minimizing the average square sum of the response time. In other words, we got enough added speed for a small enough added variance in response time to make the trade-off acceptable. As more and more samples are obtained the variance of structure based algorithm approached variance of the naive algorithm making the trade-off even more reasonable for large intended sample sizes.

Finally, we point out that both the geometric file structure based algorithms, batch and online case, were able to read sample records from disk almost at the maximum sustained speed of the hard disk, at around 45 MB/sec. This is comparable to the rate of a sequential read from disk, the best we can hope for.

9 Related Work

Sampling has a very long history in the data management literature, and research continues unabated today [1, 3, 5, 8, 7, 16, 17, 22, 24, 25, 36]. However, the most previous papers (including the aforementioned references) are concerned with how to *use* a sample, and not with how to actually *store* or *maintain* one. Most of these algorithms could be viewed as potential users of a large sample maintained as a geometric file.

As mentioned in the introduction, a series of papers by Olken and Rotem (including two papers listed in the References section [32, 31]) probably constitute the most well-known body of research detailing how to actually compute samples in a database environment. Olken and Rotem give an excellent survey of work in this area [33]. However, most of this work is very different than ours, in that it is concerned primarily with sampling from an existing database file, where it is assumed that the data to be sampled from are all present on disk and indexed by the database. Single pass sampling is generally not the goal, and when it is, management of the sample itself as a disk-based object is not considered.

The algorithms in this paper are based on *reservoir sampling*, which was first developed in the 1960's [15, 28]. In his well-known paper [37], Vitter extends this early work by describing how to decrease the number of random numbers required to perform the sampling. Vitter's techniques could be used in conjunction with our own, but the focus of existing work on reservoir sampling is again quite different from ours; management of the sample itself is not considered, and the sample is implicitly assumed to be small and in-memory. However, if we remove the requirement that our sample of size N be maintained on-line so that it is always a valid snapshot of the stream and must evolve over time, then sequential sampling techniques related to reservoir sampling that could be used to build (but not maintain) a large, on-disk sample (see Vitter [38], for example).

Several data structures and algorithms have been proposed to speed up index inserts such as LSM Tree [34], Buffer Tree [4], and Y-Tree [26]. These papers consider problem of providing I/O efficient indexing for a database experiencing a very high record insertion rate with which it is almost impossible to insert into traditional B+- tree indexing structure. Any of these methods could be very well used to maintain a large random sample of a data stream as geometric file does. If we consider a reservoir as a bitmap of R entire, a record can be interested at a random bitmap location using above index inserts methods. This technique works because in LSM, Buffer, and Y-Tree inserted record goes to a fix position on disk and remains their until it gets over-written by a newer record. However, none of these methods can come close to the raw write speed of the disk, as the geometric file can [27]. In a sense, the issue is that while the indexing provided by these structures could be used to implement efficient,

disk-based reservoir sampling, it is too heavy-duty a solution. We would end up paying a lot in terms of disk I/O to send a new record to overwrite a specific, existing record chosen at the time the new record is inserted, when all you really need is to have a new record overwrite any random, existing record.

There has been much recent interest in approximate query processing over data streams (a very small subset of these papers is listed in the References section [13, 14, 18]); even some work on sampling from a data stream [6]. This work is very different from our own, in that most existing approximation techniques try to operate in very small space. Instead, our focus is on making use of today's very large and very inexpensive secondary storage to physically store the largest snapshot possible of the stream.

Finally, we mention the U.C. Berkeley CONTROL project [23] (which resulted in the development of *online aggregation* [24] and *ripple joins* [22]). This work *does* address issues associated with randomization and sampling from a data management perspective. However, the assumption underlying the CONTROL project is that all of the data are present and can be archived by the system; online sampling is not considered. Our work is complementary to the CONTROL project in that their algorithms could make use of our samples. For example, a sample maintained as a geometric file could easily be used as input to a ripple join or online aggregation.

10 Conclusions and Future Work

Random sampling is a ubiquitous data management tool, but relatively little research from the data management community has been concerned with how to actually compute and maintain a sample. We have considered the problem of random sampling from a data stream, where the sample to be maintained is very large and must reside on secondary storage. The main contribution of the paper is the development of the *geometric file*. The geometric file can be used to maintain an on-line sample of arbitrary size with an amortized cost of $O(\omega \times \log |B|/|B|)$ random disk head movements for each newly sampled record. The multiplier ω can be made very small by making use of a small amount of additional disk space.

One obvious direction for future work is handling the case where record size is variable. Another problem is efficient index maintenance for the geometric file, so that samples with specific characteristics can be found quickly. This would be useful, for example, for handling a stream that included deletions as well as insertions, or for use during query processing.

References

1. Acharya, S., Gibbons, P., Poosala, V.: Congressional samples for approximate answering of group-by queries. In: ACM SIGMOD International Conference on Management of Data (2000)
2. Acharya, S., Gibbons, P., Poosala, V., Ramaswamy, S.: The aqua approximate query answering system. In: ACM SIGMOD International Conference on Management of Data (1999)
3. Acharya, S., Gibbons, P., Poosala, V., Ramaswamy, S.: Join synopses for approximate query answering. In: ACM SIGMOD International Conference on Management of Data (1999)
4. Arge, L.: The buffer tree: A new technique for optimal i/o-algorithms. In: International Workshop on Algorithms and Data Structures (1995)
5. Babcock, B., Chaudhuri, S., Das, G.: Dynamic sample selection for approximate query processing. In: ACM SIGMOD International Conference on Management of Data (2003)
6. Babcock, B., Datar, M., Motwani, R.: Sampling from a moving window over streaming data. In: ACM-SIAM Symposium on Discrete Algorithms (2002)
7. Chaudhuri, S., Das, G., Datar, M., Motwani, R., Narasayya, V.: Overcoming limitations of sampling for aggregation queries. In: ICDE (2001)
8. Chaudhuri, S., Das, G., Narasayya, V.: A robust, optimization-based approach for approximate answering of aggregate queries. In: ACM SIGMOD International Conference on Management of Data (2001)
9. Cochran, W.: Sampling Techniques. Wiley and Sons (1977)
10. Cranor, C., Gao, Y., Johnson, T., Shkapenyuk, V., Spatscheck, O.: Gigascope high performance network monitoring with an sql interface. In: ACM SIGMOD International Conference on Management of Data (2002)
11. Cranor, C., Johnson, T., Spatscheck, O., Shkapenyuk, V.: Gigascope: A stream database for network applications. In: ACM SIGMOD International Conference on Management of Data (2003)
12. Cranor, C., Johnson, T., Spatscheck, O., Shkapenyuk, V.: The gigascope stream database. In: IEEE Data Engineering Bulletin, pp. 26(1): 27–32 (2003)
13. Das, A., Gehrke, J., Riedewald, M.: Approximate join processing over data streams. In: ACM SIGMOD International Conference on Management of Data (2003)
14. Dobra, A., Garofalakis, M., Gehrke, J., Rastogi, R.: Processing complex aggregate queries over data streams. In: ACM SIGMOD International Conference on Management of Data (2002)
15. Fan, C., Muller, M., Rezucha, I.: Development of sampling plans by using sequential (item by item) techniques and digital computers. In: Journal of American Statistical Association, pp. 57: 387–402 (1962)
16. Ganguly, S., Gibbons, P., Matias, Y., Silberschatz, A.: Bifocal sampling for skew-resistant join size estimation. In: ACM SIGMOD International Conference on Management of Data (1996)
17. Ganti, V., Lee, M.L., Ramakrishnan, R.: Icicles - self-tuning samples for approximate query answering. In: International Conference on Very Large Data Bases (2000)
18. Gehrke, J., Korn, F., Srivastava, D.: On computing correlated aggregates over continual data streams. In: ACM SIGMOD International Conference on Management of Data (2001)
19. Gibbons, P., Matias, Y., Poosala, V.: Fast incremental maintenance of approximate histograms. In: ACM Transactions on Database Systems, pp. 27(3): 261–298 (2002)
20. Gunopulos, D., Kollios, G., Tsotras, V., Domeniconi, C.: Approximating multi-dimensional aggregate range queries over real attributes. In: ACM SIGMOD International Conference on Management of Data (2000)
21. Haas, P.: The need for speed: Speeding up db2 using sampling. In: IDUG Solutions Journal (2003)
22. Haas, P.J., Hellerstein, J.M.: Ripple joins for Online Aggregation. In: ACM SIGMOD International Conference on Management of Data, pp. 287 – 298 (1999)
23. Hellerstein, J., Avnur, R., Raman, V.: Informix under control online query processing. In: Data Mining and Knowledge Discovery, pp. 4(4): 281–314 (2000)
24. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online Aggregation. In: ACM SIGMOD International Conference on Management of Data, pp. 171–182 (1997)
25. Jermaine, C.: Robust estimation with sampling and approximate pre-aggregation. In: International Conference on Very Large Data Bases (2003)
26. Jermaine, C., Datta, A., Omiecinski, E.: A novel index supporting high volume data warehouse insertion. In: International Conference on Very Large Data Bases (1999)
27. Jermaine, C., Omiecinski, E., Yee, W.: The partitioned exponential file for database storage management. In: International Conference on Very Large Data Bases (1999)
28. Joens, T.: A note on sampling from a tape file. In: Communications of the ACM, p. 5: 343 (1964)
29. Johnson, N., Kotz, S.: Discrete Distributions. Houghton Mifflin (1969)
30. Olken, F.: Random Sampling from Databases. In: Ph.D. Dissertation (1993)
31. Olken, F., D. Rotem, P.X.: Random sampling from hash files. In: ACM SIGMOD International Conference on Management of Data (1990)
32. Olken, F., Rotem, D.: Random sampling from b+ trees. In: International Conference on Very Large Data Bases (1989)
33. Olken, F., Rotem, D.: Random sampling from database files - a survey. In: International Working Conference on Scientific and Statistical Database Management (1990)
34. O’Neil, P., Cheng, E., Gawlick, D., O’Neil, E.: The log-structured merge-tree. In: Acta Informatica, pp. 33:351–385 (1996)
35. Shao, J.: Mathematical Statistics. Springer-Verlag (1999)
36. Toivonen, H.: Sampling large databases for association rules. In: International Conference on Very Large Data Bases (1996)
37. Vitter, J.: Random sampling with a reservoir. In: ACM Transactions on Mathematical Software (1985)
38. Vitter, J.: An efficient algorithm for sequential random sampling. In: ACM Transactions on Mathematical Software, pp. 13(1): 58–67 (1987)
39. Vitter, J., Wang, M.: Approximate computation of multidimensional aggregates of sparse data using wavelets. In: ACM SIGMOD International Conference on Management of Data (1999)