Online Maintenance of Very Large Random Samples

Christopher Jermaine, Abhijit Pol, Subramanian Arumugam
Department of Computer and Information Sciences and Engineering
University of Florida
Gainesville, FL, USA, 32611
{cjermain, apol, sa2}@cise.ufl.edu

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. Our algorithms are also suitable for biased or unequal probability sampling.

1 Introduction

Despite the variety of alternatives for approximate query processing (including several references listed in this paper [8][9][10][14][29]), sampling is still one of the most powerful methods for building a one-pass synopsis of a data set 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 [7][25]).
- 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][2][4][5][6][12][13][16][18][19][26] 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 [22][24], who also offer the definitive survey of related work through the early 1990's [23]. However, this work is relevant mostly for sampling from data stored in a database, and is not suitable for emerging applications such as stream-based data management. Furthermore, the implicit assumption in most existing work is that a "sample" is a small, in-memory data structure. This is not always true. For many applications, very large samples containing billions of records can be required to provide acceptable accuracy.

Fortunately, modern storage hardware gives us the capacity to cheaply store very large samples that should suffice for even difficult and emerging applications, such as futuristic "smart dust" environments where billions of tiny sensors produce billions of observations per second that must be joined, cross-correlated, and so on. Currently, a terabyte of commodity hard disk storage costs only around $1,000. Given current trends, we should see storage costs of $1,000 per petabyte by the year 2020. To put this into context, a petabyte is enough storage to store a sample containing 10 trillion, 100-byte records.

While modern storage hardware might be up to the task of actually storing large samples, physically getting the sample on a disk and maintaining it in the face of an intense stream of new data is another matter. Current techniques suitable for maintaining samples from a data stream are based on *reservoir sampling* [11][20][27]. 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 $N >> |B|$?*

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-

SIGMOD 2004, June 13-18, 2004, Paris, France.

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.

(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. 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 less than $(\omega/|B|)\log_2|B|$ random disk head movements for each newly sampled record. The multiplier ω can be made very small (down to 20 or so in practice) by making use of a small amount 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, 5, and 6, we describe the geometric file. In Section 7, we describe how the geometric file can be used for *biased sampling*, where certain records are considered more important than others. Biased sampling can be useful in many situations; for example, in sensor data management, queries might refer to recent sensor readings far more frequently than older ones. In Section 8, we detail 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 than ε with a probability δ (see Cochran for a nice introduction to sampling [7]). The value δ 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) [24]. 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 student's 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 [16] 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* [11][20][27]. To maintain a sample R of size $N = |R|$, 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 ∞ do:
(3)    Wait for a new record r to appear in the stream
(4)    With probability |R| / i
(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 [27]. After a certain number of records have been seen, the algorithm can "wake 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. Currently, a terabyte of storage requires as few as five disks, giving us a random I/O capacity of only around 500 disk head movements per second. This means we can sample only 250 records per second at 10ms per random I/O with one terabyte of storage. To put this in perspective, it would take more than *one year* to sample enough 100 byte records to fill that terabyte.

- *The massive rebuild extension*. As an alternative, we could make use of all of our available main memory to buffer new samples. When the buffer fills, we simply scan the entire reservoir and replace a random subset of the existing records with the new, buffered samples. The modified algorithm is sketched below. B refers to the buffer, and $|B|$ refers to the total buffer capacity.

Algorithm 2: Reservoir Sampling with a Buffer

```
(1)  For int i = 1 to ∞ do:
(2)     Wait for a new record r to appear in the stream
(3)     If i < |R| // until the reservoir fills, no buffering
(4)        Add r directly to R and continue
(5)     Else
(6)        With probability |R| / i
(7)           With probability Count (B) / |R|
(8)              // new samples can rewrite buffered samples
(9)              Replace a random record in B with r
(10)          Else
(11)             Add r to B
(12)          If Count (B) == |B|
(13)             Scan the reservoir R and empty B in one pass
(14)             B = ∅
```

With a large disk block size and main memory, we can expect that most disk blocks will receive at least one new sample, so we might as well rely on fast, sequential I/O to update the entire file. The drawback of this approach is that 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 after 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, and requires taking the sample off-line.

4 The Geometric File

The three alternatives described for maintaining a large sample all have drawbacks. In this section, we discuss a fourth algorithm and an associated data organization called the *geometric file*.

4.1 Intuitive Description

The geometric file makes use of a main memory buffer to store new samples as they are captured from the data stream. The samples collected one at time in the buffer are viewed as a single *subsample* or a *stratum* [7]. Thus, the records in a subsample are a non-random subset of the records in the file, since they were sampled during the sample time period from the data stream.

To maintain the correctness of the reservoir sampling algorithm that is the basis for the geometric file, each new subsample that is added to the reservoir must overwrite a true, random subset of the records in the reservoir. Because a subsample is a non-random sample of the records from the stream, a new subsample must overwrite a random sample of the records stored in *each* of the existing subsamples currently archived in the reservoir. The key observation behind the geometric file is that the decay of a subsample as it loses its records to new subsamples can be characterized with reasonable accuracy using a *geometric series* (hence the name *geometric file*). As new subsamples are added to the file 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 new subsample to the overall size of the reservoir. Thus, the size of a subsample decays approximately in an exponential manner as new subsamples are added to the reservoir.

This exponential decay is used to great advantage in the geometric file. Each subsample is partitioned into a set of *segments* of exponentially decreasing size. These segments are sized so that every time a new subsample is added to disk, we expect that each existing subsample loses *exactly the set of records contained in its largest segment*. As a result, each subsample loses one segment every time the buffer is emptied. Since segments are stored as contiguous runs of blocks on disk, this leads to fast, sequential writes and few (if any) reads to process additional samples.

4.2 Characterizing Subsample Decay

To describe the geometric file in detail, we begin with an analogy between samples in a subsample and water in a bathtub. Say we have a tub which contains a certain amount of water (e.g., 10 gallons). Imagine that we decide to empty some fraction of the water from the tub so that the fraction of the original water that remains is α (in our example, let $\alpha = 0.8$). We then measure how much water was removed, and find that we have taken out n units ($n = 2$ gallons in our example). After this water is removed from the tub, we again remove the same fraction $1 - \alpha$ of the remaining water. Obviously, this time we will remove $n\alpha = 1.6$ gallons of water. We next remove the same fraction $1 - \alpha$ of the remaining water, this time removing $n\alpha^2 = 1.2$ gallons. This process is repeated over and over until the tub has less than β gallons of water remaining.

Three questions that are very relevant to our problem of maintaining very large samples from a data stream are:

- How much water have we emptied after m iterations?
- What would be the sum of all of the water measurements taken if we completely emptied the tub?

•How many times must we remove water from the tub before there are β gallons left?

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

Observation 1: Given a decay rate $\alpha < 1$, it holds that $\sum_{i=0}^{m} n\alpha^i = \frac{n\alpha^{m+1} - n}{\alpha - 1}$ for any $n \in \mathbf{R}$, $m \in \mathbf{Z}+$.

Observation 2: Given a decay rate $\alpha < 1$, it holds that $\sum_{i=0}^{\infty} n\alpha^i = \frac{n}{1-\alpha}$ for any $n \in \mathbf{R}$.

Observation 3: Given a decay rate $\alpha < 1$, define $f(j)$ as $\sum_{i=j}^{\infty} n\alpha^i$

From Observation 1, it follows that the largest j such that $f(j) \geq \beta$ is $j = \left\lceil \frac{\log\beta - \log n + \log(1-\alpha)}{\log\alpha} \right\rceil$.

These observations are relevant to our problem of characterizing subsample decay. Consider some arbitrary subsample S of our large, disk-based sample R (so $S \subset R$). Imagine that R is maintained using a reservoir sampling algorithm in conjunction with a main memory buffer B, where $\frac{|B|}{|R|} = 1 - \alpha$. Recall that the way that reservoir sampling works is that new samples from the data stream are chosen to overwrite random samples currently in the reservoir. If we fill a buffer with new samples, and then merge those new samples with R by overwriting a random subset of the existing samples in R, then the subsample S will expectedly lose $1-\alpha|S|$ of its own records.[1]

We can roughly describe the expected decay of S after repeated buffer flushings using the three observations stated above. If $|S| = \frac{n}{\alpha - 1}$ for some number n, then it follows from Observation 2 that the ith merge from the buffer into R will expectedly remove $n\alpha^{i-1}$ records from what remains of S. Furthermore, the expected number of merges required until S has only β samples left is

$$\left\lceil \frac{\log\beta - \log n + \log(1-\alpha)}{\log\alpha} \right\rceil$$

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 $\left\lceil \frac{\log\beta - \log n + \log(1-\alpha)}{\log\alpha} \right\rceil$ "segments" of exponentially decreasing size, plus a special, final group of segments of total size β, then the ith buffer flush into R will expectedly remove exactly one segment from S. The process is shown in Figure 1.

4.3 Basic 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 that are loaded into our disk-based sample in a single emptying of the buffer. Since we know that the number of records that remain in S will expectedly decay over time as depicted in Figure 1, we can organize our large, disk based sample as a set of decaying subsamples. 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 n samples to the most recent buffer flush. In general, the ith largest subsample was created by the ith 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.

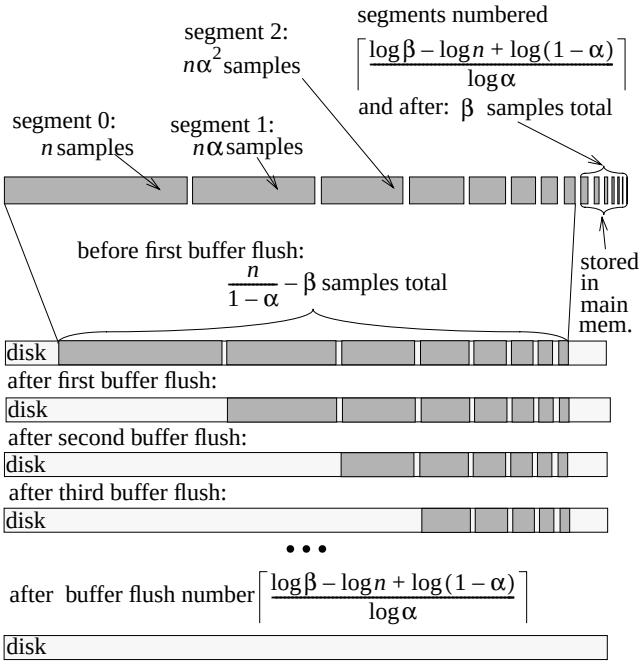


Figure 1: Decay of a subsample after multiple buffer flushes.

Using this file organization to dynamically maintain a random sample from a data stream is then straightforward. Every time that the buffer fills, it is loaded into the reservoir. To do this, first randomize the ordering of the sampled records in the buffer. Then divide the records from the buffer into segments of size n, $n\alpha$, $n\alpha^2$, $n\alpha^3$, and so on. All segments of size $n\alpha^i$ or smaller for $i \geq \left\lceil \frac{\log\beta - \log n + \log(1-\alpha)}{\log\alpha} \right\rceil$ are put into a single group of total size β which is stored in main memory. After the records from the buffer have been partitioned into segments, those segments are used to overwrite the largest on-disk segment of each of the existing subsamples in R. The reason that the large group of very small segments of total size β is buffered in main memory is that overwriting a segment requires at least one random disk head movement, which is costly. By storing the very small segments in main memory, we can reduce the number of disk head movements with little additional main-memory storage cost. The overall process of adding a new subsample to a geometric file is depicted in Figure 3. As new subsamples are added to the file, the size of each existing subsample decays geometrically.

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.*

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).

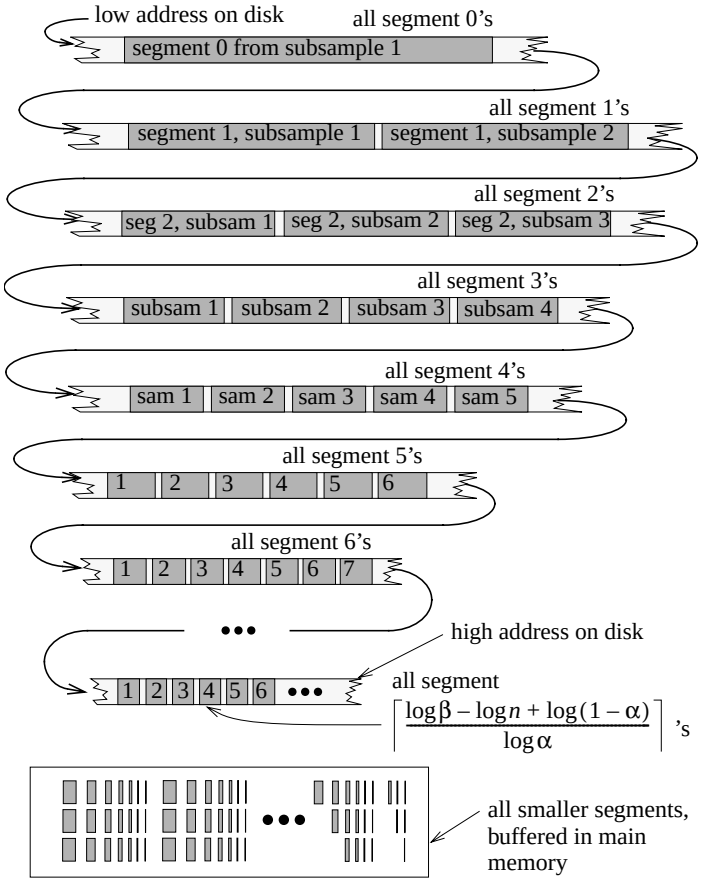


Figure 2: Basic structure of the Geometric File.

- Each buffer flush requires only $\left\lceil \frac{\log\beta - \log n + \log(1-\alpha)}{\log\alpha} \right\rceil$ 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 What If the Unexpected Happens?

Unfortunately, the process is actually not quite so simple. All of the above analysis is based on a notion of what is *expected*. In reality, sampling is a random process, and variance will occur.

In the geometric file, this variance is observed when new samples from the buffer are added to disk as a new subsample. When a segment from the new subsample S_{new} overwrites some of the records from an existing subsample S, the algorithm described thus far replaces a *fixed number* of the samples from S. In reality, we need to randomly choose records from the reservoir to replace. While the number of records replaced in S will *expectedly* be proportional to the current size of S, this is not guaranteed.

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)).

Algorithm 3: Partitioning the buffer into segments

```
(1) For each record r in the buffer B, do:
(2)     Randomly choose a victim subsample S_i such that
(3)         Pr[choosing i] = SubsampleSize[i] / |R|
(4)     SubsampleSize[i]--;
(5)     SubsampleSize[newSubsample]++
(6)     Add r to segment i in the new subsample
```

Thus, when we add a new subsample to disk via a buffer flush, we need to perform a *randomized* partitioning of the buffer into segments, described above by Algorithm 3.

4.4.1 Handling Variance In Subsample Size

The difficulty is that by introducing this additional randomization, we now have no hard guarantees on the sizes of the various segments in a given subsample. We *expect* that the various segments will be correctly sized, but there will be some variance. To handle the variance that has been introduced, we will associate a Last-In-First-Out (LIFO) stack with each of the subsamples held on disk. The stack associated with a subsample will hold any of a subsample's records that cannot fit in the subsample's segments due to anomalies experienced during the insertion process.

Making use of this additional storage is fairly straightforward. Imagine that records from a new subsample S_{new} are sent to overwrite a segment from an existing subsample S. Then, there are two possible cases (the cases are illustrated in Figure 5).

- **Case 1**: *The amount of data in the new segment from S_{new}* ***is smaller than*** *the amount of data in the existing segment from S by some number of records ε.* (Shown in Figure 5 (a)). In this case, ε records are popped off of S_{new}'s stack to reflect the additional records that must be added to the segment. Furthermore, ε records from S's segment are moved to S's stack to reflect the fact that these records were not removed from R (Figure 5 (c)).

- **Case 2**: *The amount of data in the new segment from S_{new}* ***exceeds*** *the amount of data in the existing segment from S by some number of records ε.* (Shown in Figure 5 (b)). In this case, ε records are popped off of S's stack to reflect the fact that S has lost some additional records. Furthermore, ε records from S_{new}'s segment are moved to S_{new}'s stack, since we do not have enough space to write these records to disk in their correct position (Figure 5 (d)).

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; overflow or underflow can be handled efficiently by adding or removing records directly.

4.4.2 Bounding the Variance

Because the LIFO 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 exactly how much overflow we can expect from a given subsample, which will bound the growth of the 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, necessitating a reorganization. This would also introduce significant additional complexity into the geometric file, since the

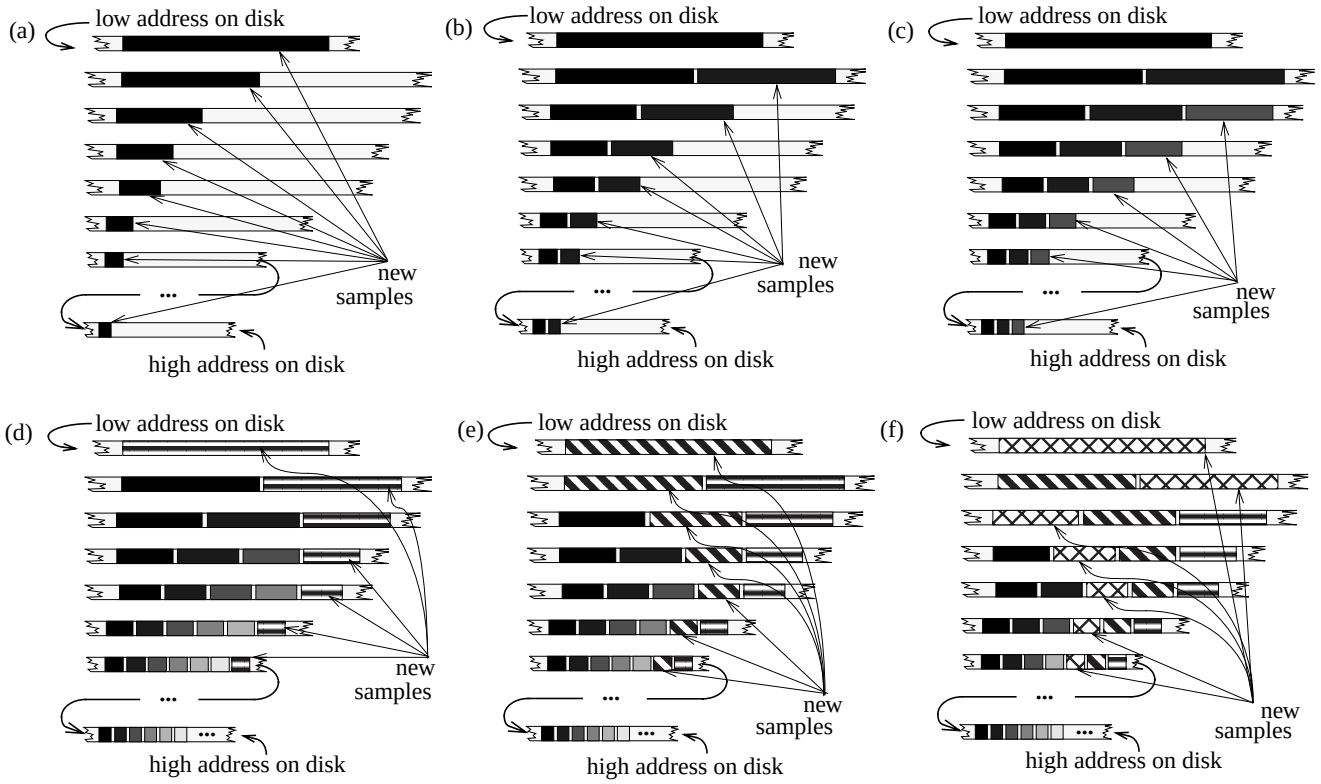


Figure 3: Building a geometric file. First, the file is filled with the initial data produced by the stream (a through c). To add the first records to the file, the buffer is allowed to completely fill with samples. The buffered records are then randomly grouped into segments, and the segments are written to disk to form the largest initial subsample (a). For the second initial subsample, the buffer is only allowed to fill to α of its capacity before being written out (b). For the third initial subsample, the buffer fills to α^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).

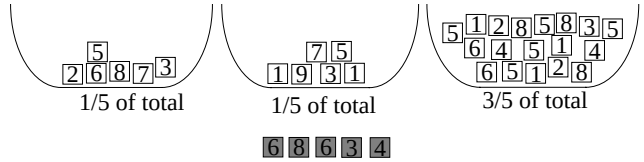
assumption has to be that additional samples will continue to arrive from the data stream, meaning that the reorganization would have to be performed concurrently with new data insertion.

To avoid this, we observe that if the stack associated with a subsample 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.

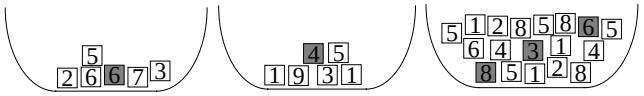
To bound this difference, we first note that after adding b new samples into the reservoir, the probability that any existing sample in the reservoir has been overwritten by a new sample is

$$1 - \left(1 - \frac{1}{|R|}\right)^b$$

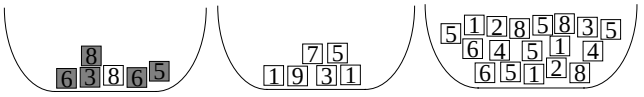
During the addition of new records to the reservoir, we can view a subsample S of initial size $|B|$ as a set of $|B|$ identical, independent Bernoulli trials (coin flips). The ith trial determines whether the ith sample was removed from S. Given this model, the number of samples remaining in S after b new samples have been added to the reservoir is binomially distributed with $|B|$ trials and $P = \Pr[s \in S \text{ remains}] = \left(1 - \frac{1}{|R|}\right)^b$. 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 $\mu = bP$ and standard deviation $\sigma = \sqrt{|B|P(1-P)}$ [21]. Simple arithmetic implies that the greatest variance is achieved when a subsample has expectedly 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 *none* of them overflow their respective stacks. If the buffer is flushed to disk 100,000 times, 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%.



(a) Five new samples randomly replace existing samples which are grouped into three buckets



(b) Most likely outcome: new samples distributed proportionally



(c) Possible (though unlikely) outcome: new samples all distributed to smallest bucket

Figure 4: Distributing new records to existing subsamples.

5 Choosing Parameter Values

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 (considering the cost to write the segment and to subsequently adjust 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 1 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 $\left\lceil \frac{\log 320 - \log 10^5 + \log(1-0.99)}{\log 0.99} \right\rceil = 1029$ 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. 1028 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:

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

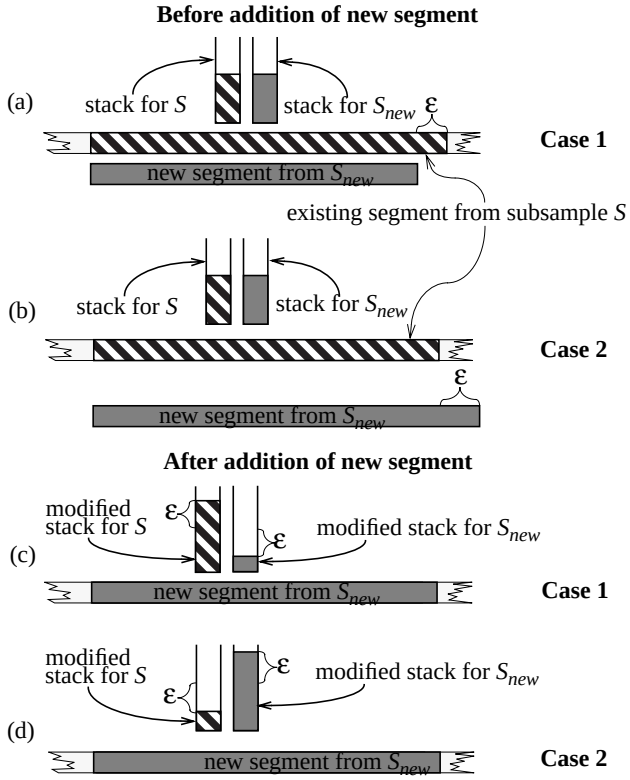


Figure 5: Effect on stack of overwriting an existing segment.

Proof: From Observation 1, we know that to ensure that the largest subsample is of size $|B|$, then $\frac{n}{1-\alpha} = |B|$. Furthermore, we also know that since the *i*th largest existing subsample has lost its largest (i - 1) segments, it follows that the size of the *i*th subsample is $\sum_{j=i-1}^{\infty} \left(\frac{n}{1-\alpha}\right)\alpha^j$. Then the size of all subsamples is then $\sum_{i=0}^{\infty}\sum_{j=i-1}^{\infty} \left(\frac{n}{1-\alpha}\right)\alpha^j$. Again using observation 1, this expression can be simplified to $\frac{n}{(1-\alpha)^2}$. Since $n = (1-\alpha)|B|$, the total size of all subsamples is $\frac{|B|}{(1-\alpha)}$, and $(1-\alpha) = \frac{|B|}{|R|}$. ■

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 $\left\lceil \frac{\log 10^4 - \log 10^5 + \log(1-0.99)}{\log 0.99} \right\rceil$ 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 demonstrated in the last section, the value of α can have a significant effect on geometric file performance. If α = 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 α =.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 $|R|/|B|$.

However, there is a way to improve the situation drastically. All that we have to do is to 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.

The algorithms described previously must be changed slightly to maintain the *m* files at once. When the *j*th new subsample is added to the reservoir, it is partitioned into segments exactly as described by algorithm 3, and all subsamples from all geometric files can be partially or fully overwritten by samples from the new subsample. However, to write the new subsample to disk, the segments created by algorithm 3 are first *consolidated* ($\alpha' < \alpha$ implies that we need to create larger segments). Consolidation is performed by grouping the new subsample's first *m* segments to form the subsample's first consolidated segment; the next *m* segments are grouped to form the second consolidated segment, and so on. The consolidated segments are then used to overwrite the largest on-disk segments *of only the subsamples stored in the* (*j* mod *m*)*th geometric file*.

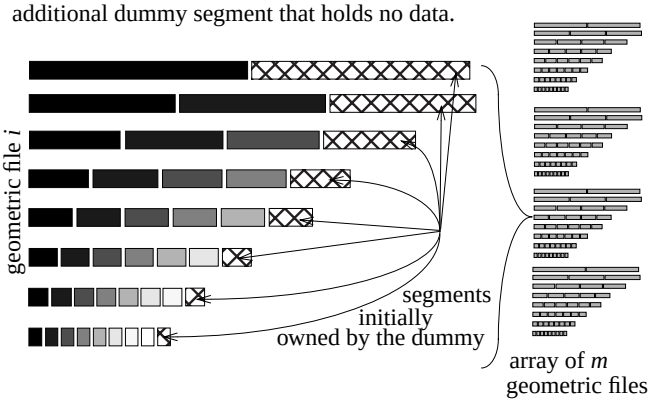
In this way, adding a new subsample only affects a single geometric file, and files are written in round-robin order. The reason that we can use $\alpha' < \alpha$ is that when a subsample has its largest on-disk segment overwritten with a consolidated segment from a new subsample, it is giving up its own storage space in lieu of storage space from other subsamples stored in the other geometric files. Thus, rather than having to write to every subsample in every geometric file, we only write to the subsamples in *one* of the files.

The problem with this modified algorithm is that when the subsample prematurely gives up this unnaturally large segment all at once, it is losing samples before it should. The effect is that a subsample may have fewer samples stored on disk than it needs to have to preserve the correctness of the reservoir sampling algorithm.

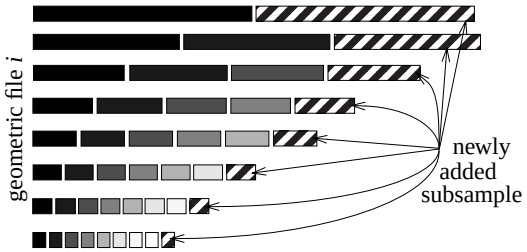
To remedy this, in each geometric file we allocate enough space to hold a complete, empty subsample. This subsample is referred to as the *dummy*. The dummy never decays in size, and never stores its own samples. Rather, it is used as a buffer that allows us to sidestep the problem of a subsample decaying too quickly.

When a new subsample is added to a geometric file, it is written only to the segments owned by the dummy. The LIFO stacks associated with each existing subsample are treated as if the new subsample had overwritten the existing subsamples directly, though they can be updated lazily, as the subsample's particular geometric file is processed. After the new subsample has been written, each existing subsample then "gives" the dummy its largest segment. Because the data in that segment will not be over-written until the next time that this particular geometric file is written to, it will be protected. Thus, 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 dummy. These samples can easily be ignored during query processing. The only additional cost is that each of the geo-

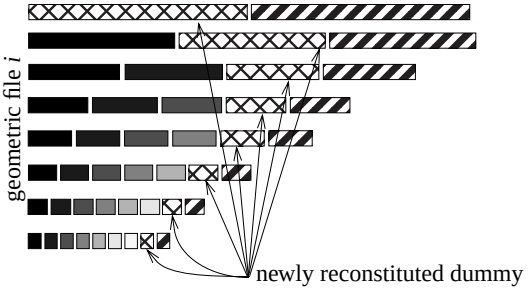
(a) Initial configuration. Each of the *m* geometric files has an additional dummy segment that holds no data.



(b) The *j*th new subsample is added by overwriting the dummy in the *i* = (*j* mod *m*)th geometric file



(c) Existing subsamples give their largest segment to reconstitute the dummy. The data in these segements are protected until the next time the dummy is overwritten.



(d) The next *m* -1 buffer flushes write new subsamples to the other *m* - 1 geometric files, using the same process. The *m*th buffer flush again overwrites the dummy in the *i*th geometric file, and the process is repeated from step (c).

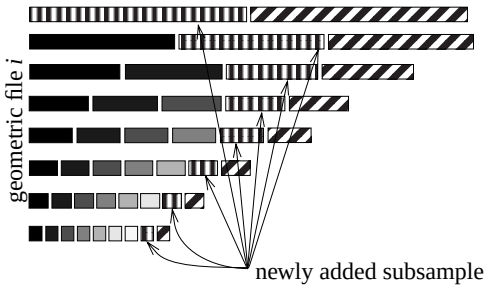


Figure 6: Speeding up the processing of new samples using multiple geometric files.

Algorithm 4: Biased Reservoir Sampling

(1) Set *totalWeight* to 0
(2) For int i = 1 to ∞ do:
(3) Wait for a new record *r* to appear in the stream
(4) *totalWeight* += *f*(*r*)
(5) If $i \leq |R|$, add *r* directly to *R*
(6) Otherwise, with probability $\frac{|R| \times f(r)}{totalWeight}$
(7) Remove a randomly selected record from *R*
(8) Add *r* to *S*

metric files on disk must have $|B|$ additional units of storage allocated. The use of a dummy subsample is illustrated in Figure 6.

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, and around four random movements at 10ms each are required per subsample segment. We omit the mathematics for brevity, but using multiple geometric files, the number of segments required to write a new subsample to disk is $\omega(\log_2|B| - \varepsilon)$ for $\omega = (\log_{0.5}\alpha')^{-1}$, $\varepsilon = \log_2\beta + \log_2(1-\alpha')$ (hence the result given in the introduction). Recall that the total storage required by all geometric files is $|R|\frac{(1-\alpha')}{(1-\alpha)}$. 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 will need less than 100 segments per 1GB buffer flush. At 4 seeks per segment, this is only *4 seconds* of random disk head movements to write 1GB of new samples to disk.

7 Biased Sampling With a Geometric File

Thus far, we have described how the geometric file can be used to efficiently maintain a very large sample from a data stream. We have assumed that each record produced by the stream has an equal probability of being sampled. A powerful technique with many applications in data management is *biased* or *unequal probability* or *stratified* random sampling [7]. Biased sampling is based on the simple observation that the relative importance of all records in a data set is typically not uniform, in which case the random sample should over-represent the more important records.

Many applications of biased sampling are described in the data management literature [4][5][6][12][13]. One example in which biased sampling can be useful over streaming data is when one wishes to take into account the time at which a record was produced. For example, consider the problem of sensor data management. Most queries will be over recent sensor readings, so it makes sense to over-represent the most recent data in the sample.

7.1 A Definition of Biased Sampling

To begin with, we assume the existence of a user-defined weighting function *f* which takes as an argument a record *r*, and returns a real number greater than 0 that describes the record's utility in subsequent query processing. *f* is defined by the user and is application-specific. In this paper, we do not consider how to define *f*, as this has been the topic of much previous research [4][5][6][12][13]. Rather, we assume the existence of such a function, and our goal is to develop tools to support its use in online sampling from a data stream. Given *f*, we define a *biased sample* as follows:

Definition 1: A *biased sample R* of the *i* records produced by a data stream is a random sample of all of the stream's records such that the probability of sampling the *j*th record from the stream is $\Pr[r_j \in R] = \frac{|R| f(r_j)}{\sum_{k=0}^{i} f(r_k)}$

The probability of selecting a record from the stream is proportional to the value of the utility function *f* applied to the record. This is a fairly simple and yet powerful definition of biased sampling, and is general enough to support many applications.

7.2 Biased Sampling With A Reservoir

We begin our description of biased sampling using the geometric file by first presenting Algorithm 4, a slightly modified version of the reservoir sampling algorithm, suitable for biased sampling.

The difference between this algorithm and algorithm 1 is that records are sampled from the stream with probability proportional to the weight function *f*. The algorithm is a useful tool for biased sampling from a stream, as described by the following lemma.

Lemma 2. Let *R* be a biased sample of the records from a stream. Let *R′* be the modified version of *R* that results from one execution of steps (3) - (8) of algorithm 4. If $\frac{|R| f(r_i)}{totalWeight} \leq 1$, then *R′* will be a biased sample of the records from the stream.

Proof: We need to prove that for each record *r* from the stream, $\Pr[r \in R'] = \frac{|R| f(r)}{\sum_{j=0}^{i} f(r_j)}$. For the new record r_i, this fact follows directly from step (6). Thus, we must prove this fact for r_k, $k < i$. Since *R* is correct, we know that for $k < i$, $\Pr[r_k \in R] = \frac{|R| f(r_k)}{\sum_{j=0}^{i-1} f(r_j)}$. In step (6), there are 2 cases; either the new record r_i is chosen for the reservoir, or it is not. If r_i is *not* chosen, then r_k remains in the reservoir for $k < i$. If r_i *is* chosen, then r_k remains in the reservoir if r_k is not selected for expulsion from the reservoir (the chance of this happening if r_i is chosen is (|R| - 1) / |R|). Thus, the probability that a record r_k is in *R′* is:

$$\Pr[r_k \in R'] = \Pr[r_k \in R]\left(\Pr[r_i \in R']\left(\frac{|R|-1}{|R|}\right) + 1 - \Pr[r_i \in R']\right)$$

$$= \Pr[r_k \in R]\left(\frac{|R|\Pr[r_i \in R'] - \Pr[r_i \in R']}{|R|} + \frac{|R| - |R|\Pr[r_i \in R']}{|R|}\right)$$

$$= \Pr[r_k \in R]\frac{(|R| - \Pr[r_i \in R'])}{|R|}$$

$$= \Pr[r_k \in R]\left(1 - \frac{f(r_i)}{\sum_{j=0}^{i} f(r_j)}\right) = \Pr[r_k \in R]\left(\frac{\sum_{j=0}^{i-1} f(r_j)}{\sum_{j=0}^{i} f(r_j)}\right)$$

$$= \left(\frac{|R| f(r_k)}{\sum_{j=0}^{i-1} f(r_j)}\right)\left(\frac{\sum_{j=0}^{i-1} f(r_j)}{\sum_{j=0}^{i} f(r_j)}\right) = \frac{|R| f(r_k)}{\sum_{j=0}^{i} f(r_j)}$$

This is exactly the desired probability, and the lemma is proven. ■

7.3 Biased Sampling With the Geometric File

Algorithm 4 and Lemma 2 suggest a simple modification to the geometric file to allow biased sampling. To maintain a biased sample, we could simply maintain a running sum of the total weight of

all samples produced by the stream, and then select each incoming record from the stream with probability $(|R| f(r_i))/ totalWeight$.

However, there is a problem with this solution. Algorithm 4 is only correct so long as $(|R| f(r_i))/ totalWeight$ never exceeds one. If this value *does* exceed one, then the probability calculations used to prove Lemma 2 are meaningless and the correctness of the algorithm is not preserved. Unfortunately, we may see such meaningless probabilities with high frequency early on as the reservoir is initially filled with samples and the value of *totalWeight* is relatively small. After some time, the situation will improve as the number of records produced by the stream is very large and *totalWeight* grows accordingly, making it unlikely that any single record will have $(|R| f(r_i))/$ totalWeight > 1.

To handle those early cases where $(|R| f(r_i))/ totalWeight$ exceeds one, we will require a few algorithmic modifications. Our goal is try to ensure that the weight of each record r is exactly $f(r)$. However, we will not be able to guarantee this in the case of streaming data. We describe what we will be able to guarantee in the context of the *true weight* of a record:

Definition 2: The value t is the *true weight* of a record r if and only if $\Pr[r \in R] = \frac{|R|t}{totalWeight}$.

What we will be able to guarantee is then twofold:

(1) First, we will be able to guarantee that the true weight of record r_j will be exactly $f(r_j)$ if $(|R| f(r_i))/ totalWeight \leq 1$ for $i > j$.

(2) We can also guarantee that we can compute the *true weight* for a given record to unbias any estimate made using our sample.

In other words, our biased sample can still be used to produce unbiased estimates that are correct on expectation [7], but the sample might not be biased exactly as specified by the user-defined function f, if the value of $f(r)$ tends to fluctuate wildly. While this may seem like a drawback, the number of records not sampled according to f will usually be small. Furthermore, since the function used to measure the utility of a sample in biased sampling is usually the result of an approximate answer to a difficult optimization problem [5] or the application of a heuristic [13], having a small deviation from that function might not be of much concern.

7.3.1 Auxiliary Information

To use the geometric file for biased sampling, it is vital that we be able to compute the true weight of any given record. To allow this, we will require that the following auxiliary information be stored:

- Each record r will have its *effective weight* $r.weight$ stored along with it in the geometric file on disk. Once *totalWeight* becomes large, we can expect that for each new record r, $r.weight = f(r)$. However, for the initial records from the data stream, these two values will not necessarily be the same.
- Each subsample S_i will have a *weight multiplier* M_i associated with it. Again, for subsamples containing records produced by the data stream after *totalWeight* becomes large, M_i will typically be one. For efficiency, M_i can be buffered in main memory. Along with the effective weight, the weight multiplier can give us the *true weight* for a given record, which will be $M(r) \times r.weight$.

7.3.2 Algorithmic Changes

Given that we need to store this auxiliary information, the algorithms for sampling from a data stream using the geometric file will require three changes to support biased sampling. These modifications are described now:

- **During start-up**. To begin with, the reservoir is filled with the first $|R|$ records from the stream. For each of these initial records, $r.weight$ is set to one. Let *totalWeight* be the sum of $f(r)$ over the first $|R|$ records. When the reservoir is finished filling, M_i is set to *totalWeight* / $|R|$ for every one of the initial subsamples. In this way, the true weight of each of the first $|R|$ records produced by the data stream is set to be the mean value of $f(r)$ for the first $|R|$ records. Giving the first $|R|$ records a uniform true weight is a necessary evil, since they will all be overwritten by subsequent buffer flushes with equal probability.
- **As subsequent records are produced by the data stream.** Just as suggested by Algorithm 4, additional records produced by the stream are added to the buffer with probability $(|R| f(r))/ totalWeight$, so that at least initially, the true weight of the ith record is exactly $f(r_i)$. The interesting case is when $\frac{|R| f(r_i)}{totalWeight} > 1$ when the ith record is produced by the data stream. In this case, we must scale the true weight of every existing record up so that $\frac{|R| f(r_i)}{totalWeight} = 1$. To accomplish this, we do the following:
  (1) For each on-disk subsample, M_j is set to be $\frac{|R| M_j f(r_i)}{totalWeight}$.
  (2) For each sampled record still in the buffer, $r_j.weight$ is set to $\frac{|R| \times r_j.weight \times f(r_i)}{totalWeight}$
  (3) Finally, *totalWeight* is set to $|R| f(r_i)$.
- **As the buffer fills**. When the buffer fills and the jth subsample is to be created and written to disk, M_j is set to 1.

Finally, we argue that the above algorithm is correct, in the sense that the sample maintained by this algorithm is a correct, biased sample for some easily computable weighting function $f' \approx f$, where the true weight of record r sample is $M(r) \times r.weight$.

Lemma 3. Using the above algorithm, after i records have been produced by the stream, we are guaranteed that $\Pr[r \in R] = \frac{|R| M(r) r.weight}{\sum_{j=0}^{i} M(r_j) r_j.weight}$ for every record r produced by the data stream.

Proof outline. The key reason for the correctness of the algorithm is that when a new record is encountered with $\frac{|R| f(r_i)}{totalWeight} > 1$, the true weight of all of the existing samples is "bumped up" so as to maintain the relative probabilities of being sampled for all samples produced thus far by the data stream. ■

8 Benchmarking

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. 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

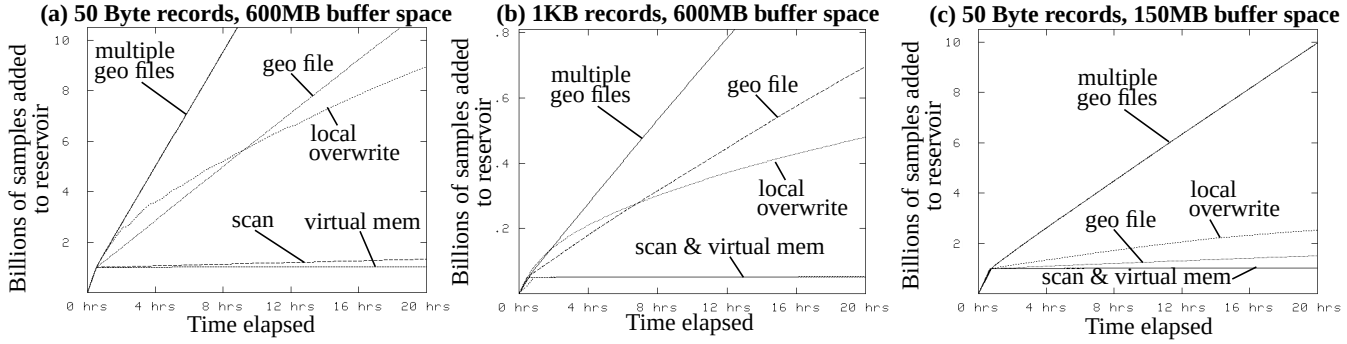


Figure 7: Results of benchmarking experiments.

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. The following three experiments were performed:

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 produced (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 is plotted as a function of time in Figure 7 (a). These lines can be seen as representing the maximum sustained processing capabilities of each of the five options. In each experiment, it was assumed that every record produced by the stream was sampled, so the samples were biased towards the more recent records produced by the stream. This is probably realistic in many cases. For unbiased sampling, fewer samples would be taken from the stream by line (4) of Algorithm 1, and so the performance of each of the five methods would all be scaled up by the same factor, as each method would need to add only some fraction of the stream's records to the reservoir.

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 7 (b). Thus, we test the effect of record size on the five options.

Experiment 3: This experiment is identical to Experiment 1, except that the amount of buffer memory 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 7 (c). This experiment tests the effect of a constrained amount of main memory.

Discussion

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 7 (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). There *is* something of a decline in performance 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 2 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 additional processing occurs due to the inefficiency of the two options.

9 Related Work

Sampling has a very long history in the data management literature, and research continues unabated today [1][2][4][5][6][12][13][16][18][19][26]. 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 [22][24]) 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 [23]. 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 [11][20]. In his well-known paper [27], 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 [28], for example).

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 [8][9][14]); even some work on sampling from a data stream [3]. 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 [17] (which resulted in the development of *online aggregation* [18] and *ripple joins* [16]). 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 online sample of arbitrary size with an amortized cost of less than $(\omega/|B|)\log_2|B|$ random disk head movements for each newly sampled record. The multiplier ω can be made very small (down to 20 or so in practice) by making use of a small amount of additional disk space. We have considered using the geometric file for biased or unequal probability sampling.

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] S. Acharya, P.B. Gibbons, V. Poosala: Congressional Samples for Approximate Answering of Group-By Queries. *SIGMOD* 2000: 487-498

[2] S. Acharya, P.B. Gibbons, V. Poosala, S. Ramaswamy: Join Synopses for Approximate Query Answering. *SIGMOD* 1999: 275-286

[3] B. Babcock, M. Datar, R. Motwani. Sampling From a Moving Window Over Streaming Data, *SODA* 2002

[4] B. Babcock, S. Chaudhuri, G. Das: Dynamic sample selection for Approximate Query Processing. *SIGMOD* 2003: 539-550

[5] S. Chaudhuri, G. Das, V.R. Narasayya: A Robust, Optimization-Based Approach for Approximate Answering of Aggregate Queries. *SIGMOD* 2001

[6] S. Chaudhuri, G. Das, M. Datar, R. Motwani, V.R. Narasayya: Overcoming Limitations of Sampling for Aggregation Queries. *ICDE* 2001: 534-542

[7] W. Cochran. *Sampling Techniques*. Wiley & Sons, 1977

[8] A. Das, J. Gehrke, M. Riedewald: Approximate Join Processing Over Data Streams. *SIGMOD* 2003: 40-51

[9] A. Dobra, M.N. Garofalakis, J. Gehrke, R. Rastogi: Processing complex aggregate queries over data streams. *SIGMOD* 2002: 61-72

[10] D. Gunopulos, G. Kollios, V.J. Tsotras, C. Domeniconi: Approximating Multi-Dimensional Aggregate Range Queries over Real Attributes. *SIGMOD* 2000: 463-474

[11] C. Fan, M. Muller, I. Rezucha: Development of sampling plans by using sequential (item by item) techniques and digital computers. *J. Amer. Stat. Assoc.* 57: 387-402 (1962)

[12] S. Ganguly, P.B. Gibbons, Y. Matias, A. Silberschatz: Bifocal Sampling for Skew-Resistant Join Size Estimation. *SIGMOD* 1996: 271-281

[13] V. Ganti, M.-L. Lee, R. Ramakrishnan: ICICLES: Self-Tuning Samples for Approximate Query Answering. *VLDB* 2000: 176-187

[14] J. Gehrke, F. Korn, D. Srivastava: On Computing Correlated Aggregates Over Continual Data Streams. *SIGMOD* 2001

[15] P.B. Gibbons, Y. Matias, V. Poosala: Fast incremental maintenance of approximate histograms. *TODS* 27(3): 261-298 (2002)

[16] P.J. Haas, J.M. Hellerstein: Ripple Joins for Online Aggregation. *SIGMOD* 1999: 287-298

[17] J.M. Hellerstein, R. Avnur, V. Raman: Informix under CONTROL: Online Query Processing. *Data Mining and Knowledge Discovery* 4(4): 281-314 (2000)

[18] J.M. Hellerstein, P.J. Haas, H.J. Wang: Online Aggregation. *SIGMOD* 1997: 171-182

[19] C. Jermaine: Robust Estimation With Sampling and Approximate Pre-Aggregation. *VLDB* 2003: 886-897

[20] T. Joens: A note on sampling from a tape file. *Comm. ACM*: 5, 343 (1964)

[21] N.L. Johnson and S. Kotz: *Discrete Distributions*. Houghton Mifflin, 1969.

[22] F. Olken, D. Rotem: Random Sampling from B+ Trees. *VLDB* 1989: 269-277

[23] F. Olken, D. Rotem: Random Sampling from Database Files: A Survey. *SSDBM* 1990: 92-111

[24] F. Olken, D. Rotem, P. Xu: Random Sampling from Hash Files. *SIGMOD* 1990: 375-386

[25] J. Shao: *Mathematical Statistics*. Springer-Verlag, 1999

[26] H. Toivonen: Sampling Large Databases for Association Rules. *VLDB* 1996: 134-145

[27] J.S. Vitter. Random sampling with a reservoir. *ACM Trans. on Math. Soft.*, 11(1): (1985)

[28] J.S. Vitter: An Efficient Algorithm for Sequential Random Sampling. *ACM Trans. on Math. Soft.*, 13(1): 58-67 (1987)

[29] J.S. Vitter, M. Wang: Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets. *SIGMOD* 1999: 193-204