Rainer Gemulla, Wolfgang Lehner

Deferred Maintenance of Disk-Based Random Samples

Erstveröffentlichung in / First published in:

Advances in Database Technology - EDBT 2006. München, 26.-31.03.2006. Springer, S. 423-441. ISBN 978-3-540-32961-9.

DOI: https://doi.org/10.1007/11687238_27

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

Deferred Maintenance of Disk-Based Random Samples

Rainer Gemulla and Wolfgang Lehner

Dresden University of Technology, 01099 Dresden, Germany
{gemulla, lehner}@inf.tu-dresden.de

Abstract. Random sampling is a well-known technique for approximate processing of large datasets. We introduce a set of algorithms for incremental maintenance of large random samples on secondary storage. We show that the sample maintenance cost can be reduced by refreshing the sample in a deferred manner. We introduce a novel type of log file which follows the intuition that only a "sample" of the operations on the base data has to be considered to maintain a random sample in a statistically correct way. Additionally, we develop a deferred refresh algorithm which updates the sample by using fast sequential disk access only, and which does not require any main memory. We conducted an extensive set of experiments and found, that our algorithms reduce maintenance cost by several orders of magnitude.

1 Introduction

Random samples are widely used as versatile synopses for large datasets. Such synopses are a must or at least desirable in most real-world scenarios. On the one hand, the complete dataset may not be accessible. For example, the dataset produced by a data stream is unbounded in size, and it is often too expensive to keep track of all the data elements which ever entered the system. Thus, a synopsis with a bounded size, i.e., independent of the dataset size, allows for inference of statistical properties of the dataset at the cost of some precision. On the other hand, the effort to process the complete dataset may be unacceptably high, e.g., when the dataset is very large or when the complexity of the algorithms exceeds the available resources. The latter case is ubiquitous in data warehouse systems which typically contain a huge amount of data subject to complex data mining algorithms.

Within the last decade, random sampling has been proposed as an adequate technique to summarize large datasets. Most applications require uniform samples to derive precise results and error bounds, i.e., each sample of the same size is equally likely to be produced. There exists a variety of alternative synopses for certain scenarios, but uniform random sampling bears the advantage of application neutrality. Whenever it is not known in advance which estimates will be computed on the synopsis, a uniform random sample is a good choice.

Random samples may be computed on-the-fly in certain scenarios. However, this is typically expensive—if not impossible—to perform [1]. Alternatively, one

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

may materialize the sample and update it if the underlying dataset changes. Since synopsis maintenance is no "free" operation, i.e., it has a performance impact on the processing of updates to the dataset, the cost for maintenance should be as small as possible. In the database community, research has shown that it is more efficient to decouple the update of a materialized view from operations on the underlying dataset [2]. This approach is typically referred to as *deferred refresh*.

Contributions. In this paper, we propose deferred maintenance strategies for disk-based random samples with a bounded size. Our approach is based on the well-known reservoir sampling scheme. We introduce a novel type of log file and show that it is sufficient to keep track of only a "sample" of the operations on the dataset to maintain a statistically correct random sample. Furthermore, we develop an algorithm for deferred refresh, which performs only fast sequential I/O operations, minimizes the number of reads and writes to the sample, and does not require any main memory. Our experiments indicate, that deferred maintenance reduces the maintenance cost by several orders of magnitude.

Assumptions. We assume that the random sample is too large to fit into the main memory and thereby resides on secondary storage. In fact, many estimators based on samples require the sample to be sufficiently large, e.g., even "simple" statistics estimators like the estimation of the number of distinct values do not perform well on undersized samples. The situation gets worse if more complex algorithms are executed on the sample, e.g., association rule mining or clustering algorithms. Moreover, the overall memory consumption increases with the number of samples maintained in-memory.

Concerning the storage system, we assume that sequential access is faster than random access, and that the storage system tries to store data in a sequential sequence of blocks.[1] For example, if the data is stored on a hard disk, sequential access is indeed faster than random access. Most file systems try to arrange data in sequential blocks to make use of this fact, and file system caches allow for "conversion" of random (write) accesses to sequential ones. Again, we assume that the sample is large, and therefore, the effectiveness of the cache is limited.

Throughout the paper, we assume that access to the base data is disallowed at any time. The sample maintenance algorithms "see" only the insertions, updates and deletions executed on the underlying dataset. The internal structure of the dataset is of no interest to the sampling algorithm, so that our approach natively extends to arbitrary settings, e.g., data streams, SQL views or XML repositories. We subsequently assume that the random sample is computed from a dataset R.

Paper Organization. The remainder of the paper is structured as follows: In Section 2, we discuss related work from the sampling, database and data stream community. Section 3 introduces a novel logging scheme which minimizes storage consumption and logging overhead. In Section 4, we propose efficient algorithms

[1] Even if sequential and random access perform similarly, our algorithms reduce the total number of accesses to the storage system. Moreover, if the storage system does not align data in blocks, the performance of our algorithms increases.

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

to refresh the sample by accessing the log file only. In Section 5 we discuss the applicability of our algorithms in the environment of a DBMS. An extensive set of experiments is presented in Section 6. We conclude the paper with Section 7.

2 Related Work

We first present general techniques for bounded-size random sampling, and then discuss specific methods for sampling in a data stream system as well as in a database system.

Uniform sampling. Bounded-size sampling schemes produce uniform samples of a given size M. *Sequential sampling* [3] is one of the most efficient sampling schemes which fall into this category. It accesses exactly M elements of R to compute the sample. Unfortunately, sequential sampling has to know the dataset size in advance, thus, it is not applicable to sample maintenance. However, the well-known *reservoir sampling* scheme [4] is able to maintain a sample of a dataset of unknown size, as long as there are only insertions. The basic idea is to insert the first M elements into the sample. Afterwards, each newly arriving element replaces a random element of the sample with probability $M/(|R|+1)$, or is rejected otherwise. Vitter [4] developed some techniques to efficiently compute the next element to be inserted into the sample. All the algorithms presented in this paper are based on reservoir sampling.

Sampling data streams. Sampling is ubiquitous in data stream management systems for the following two reasons: On the one hand, sampling is used to cope with high system load. If the number of arriving elements is too high to be processed completely, one may "simply" throw away some of the stream elements. This approach often appears in the context of *load shedding* [5, 6]. On the other hand, inference of statistical properties for the whole data stream seen so far is challenging since complete materialization of the stream is not feasible. One solution to this problem is the maintenance of a random sample of the complete data stream, potentially with some bias towards newer elements [7]. The maintenance algorithm has to be efficient, so that it can deal with the high arrival rates found in typical data stream scenarios.

Jermaine et al. introduced the *geometric file* (GF) [7], a technique for disk-based maintenance of samples from a data stream. The technique is based on reservoir sampling and minimizes I/O efforts by decreasing the number of accessed blocks. In fact, the major part of the GF is never read, most updates have block-level granularity and are written sequentially. However, the GF makes use of an in-memory buffer, and its performance depends strongly on the size of this buffer. Since each maintained sample requires its own buffer, the GF does not scale well with the number of samples. The GF is a deferred refresh algorithm since the sample is updated only if the buffer is completely filled. We compare the GF with our algorithms in Section 6.5.

Sampling in databases. Database samples are often tailored to their application, e.g., to represent a given workload [8], to handle data skew [9] or to

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

support joins [10] and groupings [11, 12]. Most of these techniques make use of random sampling and extend it by some means or other [7, 8, 9, 10, 11, 12, 13]. In fact, there are lots of sampling schemes which rely on reservoir sampling. These algorithms can be natively extended to support fast deferred refresh using the techniques presented in this paper. We discuss issues specific to database systems in Section 5.

3 Logging and Refresh

In this paper, we consider the maintenance of a random sample computed from a dataset R. In the following, we assume that a uniform random sample of size M has been computed already (e.g., using reservoir sampling), and that this sample is maintained as the underlying data changes. We distinguish *immediate refresh* strategies, which always keep the sample up-to-date, and *deferred refresh* strategies, which refresh the sample from time to time (e.g., lazily or periodically, see [2]). We say that a maintenance strategy is *incremental* if it never accesses the base data directly, but only the elements which are inserted.[2]

Incremental maintenance strategies consist of two phases: A *log phase* captures the insertions into the dataset, and a *refresh phase* updates the sample using the logged data. This holds for both immediate and deferred refresh strategies. In fact, immediate refresh can be seen as a deferred maintenance strategy which refreshes the sample every time the log has changed. In this section, we introduce several strategies for realizing the log phase in the case of random sampling. We assume that the log file resides on secondary storage, so that no memory is consumed. Additionally, we present naive refresh algorithms which update the sample using the log file.

3.1 Full Logging

The most basic logging strategy is to write all the insertions into the log file. We refer to this approach as *full logging*. Probably the simplest way to refresh the sample using the full log is to apply reservoir sampling subsequently to each of its elements. We denote this approach *naive full refresh*. Clearly, this strategy does not make use of the fact that the log file may contain more information than needed to update a sample, since the sample itself reflects only a portion of the underlying dataset. As will become evident in Section 5, there are more efficient refresh strategies with full logs.

The example in Figure 1 depicts a sample consisting of five elements and the full log file after 45 elements have been inserted. The reservoir sampling algorithm decides for every element whether it is included in the sample or not. In the former case, the element is called a *candidate* and replaces a random element of the sample. In the latter case, the element is ignored. As we proceed through the log, there are more and more candidates selected, and each of these

[2] We preliminarily assume that the dataset is subject to insertions only, and extend our results to updates and deletions in Section 5.

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

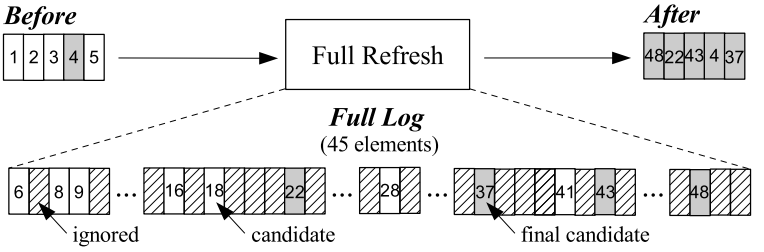

Fig. 1. Deferred sample maintenance using a full log

candidates can potentially overwrite a candidate (within the sample) which has been written earlier during the refresh phase. We say that a candidate is *final* if it is not overwritten within the current refresh operation.

Clearly, the above approach has serious disadvantages:

1. Obviously, most of the elements in the full log are not accepted into the sample and therefore logged unnecessarily. In the example, 11 out of 45 elements are made candidates, while only 4 of them remain in the final sample.
2. Updating the sample relies on random I/O (though the logfile is read sequentially). This property is directly inherited from the reservoir sampling algorithm.
3. The algorithm performs unnecessary I/O operations since the non-final candidates are overwritten by later candidates.

We propose an alternative refresh strategy for full logs in Section 5 which eliminates (2) and (3) above.

3.2 Candidate Logging

The elimination of (1) above is straightforward. The basic idea is that the elements which are ignored by the refresh operation do not have to be included in the log file. Therefore, we push the acceptance test of the reservoir sampling algorithm to the log phase.[3] Instead of logging every element added to the dataset, we decide on-the-fly whether the element is made a candidate or not. Thus, we write an arriving element to the log file with probability $M/(|R|+1)$ or ignore it otherwise. We refer to this logging strategy as *candidate logging* and denote the log file $C = \{c_1, \ldots, c_l\}$. Note that the order of the elements within the log is important since each candidate has been accepted with a different probability.

For example, instead of writing all 45 elements of Figure 1 to the full log, we only need to log the 11 candidates shown in Figure 2. In fact, the smaller the sample size with respect to the current dataset size, the more elements are skipped between two candidates on average. If we insert n elements into R, the expected log file size is given by

[3] We are free to use any other acceptance test. For example, the biased reservoir sampling scheme in [7] is more suitable for data stream sampling.

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

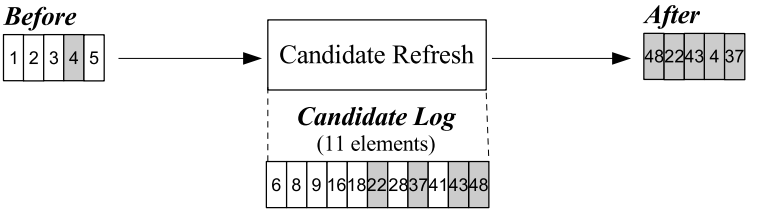

Fig. 2. Deferred sample maintenance using a candidate log

$$E(|C|) = \sum_{i=1}^{n} \frac{M}{|R|+i} \approx M \ln \frac{|R|+n}{|R|}.$$

Here, we used the logarithmic approximation for harmonic numbers. Note that $E(|C|)$ decreases as $|R|$ increases. The refresh algorithm has to be modified to make sure that every element of the candidate log is inserted into the sample. We scan the candidate log sequentially and write each candidate to a random position in the sample. We refer to this algorithm as *naive candidate refresh*. It sequentially reads $|C|$ elements of the log file and randomly writes $|C|$ elements to the sample.

Within the next section, we develop algorithms which reduce the number of read and written elements, and access both the log file and the sample sequentially (thereby eliminating (2) and (3) above).

4 Algorithms for Candidate Refresh

The naive candidate refresh algorithm has the undesirable property that access to the sample is non-sequential. Additionally, candidates written to the sample may be overwritten by subsequent candidates. This is clearly inefficient since it suffices to write out only the last candidate assigned to each element of the sample. The easiest way to circumvent these drawbacks is to precompute the changes to the sample and to write out the final candidates afterwards. Thus, all the algorithms presented in this section consist of a *precomputation phase* and a *write phase*. Using this approach, we can avoid random I/O completely while at the same time reducing the total number of disk accesses. We will present three different algorithms for precomputation, one using an in-memory array, one using an in-memory LIFO-stack, and one using no memory at all.

4.1 Array Refresh

Let A be an integer array of size M with all of its elements set to *empty*. We can use A to determine which elements of the candidate log are going to be included in the final sample. We modify the naive refresh algorithm as follows: Instead of physically reading the candidate log $C = \{c_1, \ldots, c_l\}$, we operate on the indexes $1, \ldots, l$ of the candidates within the log and thereby preliminarily avoid access

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

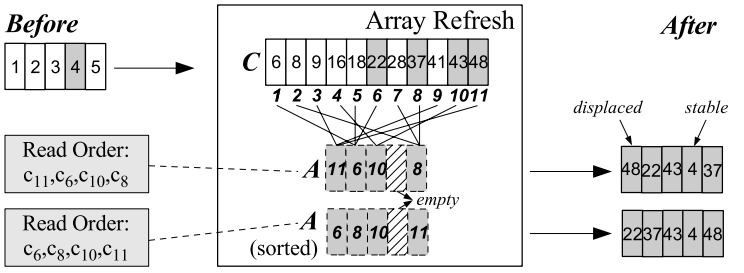

Fig. 3. Array Refresh

Algorithm 1. Array Refresh

Require: sample size M, candidate log C
1: create an in-memory array A with M empty elements
2: **for** $i = 1$ to $|C|$ **do** *// indexes of the candidates*
3: write i to a random element of A
4: **end for**
5: sort non-empty fields of A *// optional*
6: **for** $j = 1$ to M **do** *// indexes of the sample*
7: **if** $A[j]$ is not empty **then**
8: read candidate $A[j]$
9: write candidate to the jth element of the sample
10: **end if**
11: **end for**

to the log file. Furthermore, instead of writing the candidates to the sample, we store their indexes in the respective element of the in-memory array A. This prevents the random I/O of the naive algorithm.

Array A is shown for the example data in Figure 3. For clarity, empty fields are striped and indexes are written in italic and bold letters. The array consists of some empty elements and some elements containing indexes. This information is sufficient to refresh the sample in a sequential scan. Let $j = 1, \ldots, M$ denote the current position within the sample. We look up the jth value in A (denoted $A[j]$) and check whether it contains an index or not. In the former case, we write the candidate with the index $A[j]$ to the current element of the sample. We refer to sample elements which are overwritten during the refresh as *displaced elements*. In the latter case, $A[j]$ is empty and we leave the current element of the sample as it is (we do not read it actually). These elements are denoted *stable*. Note that we do not know which elements of the sample are stable and which are displaced until we have finished the precomputation phase.

The *Array Refresh* algorithm is summarized in Algorithm 1, and an example is shown in Figure 3. Access to the sample is now sequential, but access to the log file is not. However, since the order of the elements within the sample is of no interest, we may sort array A right after the preprocessing phase. Care

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

must be taken that the sort algorithm does not move empty elements to another position. These elements are linked with stable elements which in turn should be distributed randomly. Using the sorted array, access to the log file is sequential.

To analyze the I/O effort of the Array Refresh algorithm, we define a random variable Ψ_j which evaluates to 1 if the jth element of the sample is displaced and to 0 otherwise ($1 \leq j \leq M$). Clearly, the probability that an element is displaced is independent of its position within the sample:

$$P(\Psi_j = 1) = 1 - \left(1 - \frac{1}{M}\right)^{|C|}$$

In the example, each element is displaced with a probability of roughly 91%. Let $\Psi = \sum \Psi_j$ describe the total number of displaced elements, which corresponds to the number of elements read from the candidate log and subsequently written to the sample. By linearity of the expected value we get

$$E(\Psi) = M \left(1 - \left(1 - \frac{1}{M}\right)^{|C|}\right)$$

This evaluates to 4.57 in the example (Ψ itself equals 4). The Array Refresh algorithm performs Ψ sequential reads from the log file and Ψ sequential writes to the sample with $\Psi \leq \min(M, |C|)$. Therefore, Array Refresh performs better than the naive refresh algorithm. However, array A consumes a lot of memory and sorting A is an expensive operation. The next algorithm reduces the memory consumption from M to Ψ indexes and does not require a sort operation.

4.2 Stack Refresh

The *Stack Refresh* algorithm is based on the observation that the probability of overwriting a candidate by subsequent candidates is decreasing during the processing of the candidate log. For example, the first candidate may be overwritten by all the other candidates, while the last one is never overwritten. Again, we precompute the indexes of the candidates which are going to be written to the sample. A stack is used as internal data structure in order to avoid sorting.

The candidate indexes are processed in reverse order, that is, from $|C|$ to 1. For each index i, we decide whether it is part of the sample or overwritten by one of the indexes *already processed*. The latter is the case if i falls onto a position in the sample which is already occupied by one of the candidates. For example, suppose we process the candidate log as shown in Figure 3 but in reverse order. Candidate index 11 occupies sample position 1. Therefore, candidate indexes 9 and 3 – which also try to occupy position 1 – are both overwritten by 11. Therefore, only 11, 10, 8 and 6 are final in the example.

During the precomputation phase, each index i is selected with probability $p_k = (M-k)/M$ with k being the number of indexes selected already. Obviously, p_k remains constant as long as no index selected. The random variable X_k

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

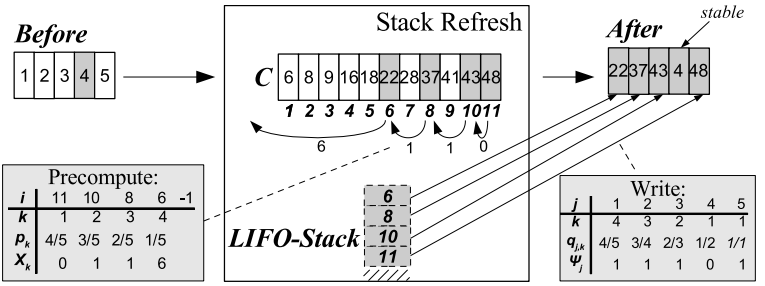

Fig. 4. Stack Refresh

describes how many indexes we have to skip until the next one is selected. X_k is geometrically distributed:

$$P(X_k = x) = P(\text{skip } x \text{ elements, select } (x+1)\text{th element})$$
$$= (1-p_k)^x p_k = \left(\frac{k}{M}\right)^x \left(\frac{M-k}{M}\right)$$

To summarize: We select the first index $|C|$. Afterwards, we generate X_1, skip X_1 indexes, and select the next one. This process is repeated using X_2, X_3, and so on. The algorithm stops as soon as M indexes have been selected or if there are no more candidates ($i < 1$). As can be seen in Figure 4, the indexes are selected in *descending order*. Therefore, we use a LIFO-stack to keep track of the selected indexes and to reverse their order.

In contrast to the Array Refresh algorithm, we do not maintain the information on which index falls onto which position. In other words, we do not precompute the set of stable and displaced elements. After the precomputation phase has finished, the stack only contains the indexes of the candidates which have to be written to the sample. We have to decide which of the corresponding candidates have to be written to which position of the sample, and which elements of the sample remain stable.

If the stack contains k indexes and the sample has size M, there are $M-k$ stable elements. In the example, the 4 selected indexes have to be distributed among the 5 elements of the sample. Therefore, only a single element remains stable. To refresh the sample, we scan it sequentially and decide for each position whether it remains stable or is overwritten by a candidate from the stack.[4] Let $j = 1, ..., M$ be the current position within the sample and k be the current stack size. Then, position j is overwritten with probability:

$$q_{j,k} = \frac{k}{M-j+1} = \frac{\text{remaining indexes}}{\text{remaining sample elements}}$$

In summary, with probability $q_{j,k}$ we pop the uppermost index from the stack, read the corresponding candidate from the log file, and write it to the current

[4] This can be done efficiently using the sequential sampling scheme introduced in [3].

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

Algorithm 2. Stack Refresh

Require: sample size M, candidate log C
1: $k \leftarrow 0$; $i \leftarrow |C|$ *// no. of selected indexes; current index*
2: **repeat**
3: PUSH(i); $k \leftarrow k + 1$ *// select the current index*
4: $p_k \leftarrow \frac{M-k}{M}$ *// selection probability for the next index*
5: $X_k \leftarrow$ NEXTGEOMETRIC(p_k) *// generate X_k*
6: $i \leftarrow i - X_k - 1$ *// skip X_k indexes*
7: **until** $i < 1 \vee k = M$
8: **for** $j = 1$ to M **do** *// indexes of the sample*
9: $q_{j,k} \leftarrow \frac{k}{M-j+1}$ *// probability that current element is displaced*
10: **with probability** $q_{j,k}$ **do**
11: $i \leftarrow$POP()
12: read the candidate with index i
13: write the candidate to the jth element of the sample
14: $k \leftarrow k - 1$ *// decrease no. of remaining candidates*
15: **end**
16: **end for**

position of the sample. In the case of Figure 4, this happens for the first, second, third and fifth element of the sample. The fourth element is stable and therefore not overwritten by a candidate. In this case, we advance to the next position without touching the stack. Algorithm 2 summarizes the complete process.

The Stack Refresh algorithm processes the sample as well as the candidate log sequentially. It needs less memory than Array Refresh since only Ψ indexes are stored in memory. The sort operation is avoided by using a stack as the central data structure. Again, the Stack Refresh algorithm performs Ψ sequential reads from the log file and Ψ sequential writes to the sample. The next algorithm improves Stack Refresh by avoiding any memory consumption.

4.3 Nomem Refresh

The Stack Refresh algorithm needs to store the selected indexes in memory for two reasons: First, the order of the generated indexes is descending. If we had not used the stack, access to the candidate log would be in reverse order and therefore less efficient. Second and more important, even if we accepted reverse scanning, we cannot avoid using the stack in general. In order to determine whether the current element of the sample is stable or not, we have to know the number of remaining indexes (see $q_{j,k}$) which is equal to the stack size. Unfortunately, we do not get this information before the precomputation phase has finished, but then we do not know which candidate indexes have been selected unless we store them in memory or are able to compute exactly the same indexes again. We show how to modify the precomputation approach in such a way that in-memory data structures are avoided if a pseudo-random number generator (PRNG) is used.

PRNGs are ubiquitous in current computer systems, e.g., each call to NEXTGEOMETRIC() in Algorithm 2 is implemented by using such a PRNG.

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

Algorithm 3. Nomem Refresh

Require: sample size M, candidate log C
1: store state of the geometric PRNG
2: compute $X = \sum(X_k + 1)$ with $k = M - 1, \ldots, 1$
3: restore state of the geometric PRNG
4: $i \leftarrow |C| - X$ *// determine first index*
5: $k \leftarrow M - 1$
6: **while** $i < 1$ **do** *// ignore negative indexes*
7: $i \leftarrow i + X_k + 1$
8: $k \leftarrow k - 1$
9: **end while**
10: **for** $j = 1$ to M **do** *// indexes of the sample ($k + 1$ candidates left)*
11: **with probability** $q_{j,k+1} = \frac{k+1}{M-j+1}$ **do** *// current element is displaced*
12: read the candidate with index i
13: write the candidate to the jth element of the sample
14: $i \leftarrow i + X_k + 1$
15: $k \leftarrow k - 1$
16: **end**
17: **end for**

A PRNG computes a sequence of numbers which appears to be random. However, the generated numbers depend only on an internal state. After a random number has been computed, the PRNG advances to the next state by using a certain algorithm. This state transition is deterministic. The central idea of the Nomem Refresh algorithm is to store the state of the PRNG before generating the sequence of selected indexes and to reset it afterwards to allow the generation of the same sequence again. Therefore, there is no need to buffer the indexes in memory. The memory consumption of the PRNG state is negligible ranging from 1 to 1000 words for common generators [14].

Reconsider the random variable X_k of the Stack Refresh algorithm. It denotes how many elements of the candidate log are skipped before the next one is selected. Since the X_k are independent of each other, it does not matter in which order they are generated. The Stack Refresh algorithm selects the candidate indexes in the following order (ignoring indexes smaller than 1):

$$|C|, \quad |C| - \sum_{k=1}^{1}(X_k + 1), \quad |C| - \sum_{k=1}^{2}(X_k + 1), \quad \ldots, \quad |C| - \sum_{k=1}^{M-1}(X_k + 1)$$

To generate this sequence in reverse order, we have to compute the quantity $X = \sum_{k=1}^{M-1}(X_k + 1)$ to determine the first index (with $k = M - 1, \ldots, 1$). Then, we subsequently add $X_k + 1$ to determine the next index. Therefore, each of the X_k is accessed twice. As already stated, we avoid buffering of the X_k by resetting the PRNG after the computation of X. The whole procedure is summarized in Algorithm 3. For brevity, we omit details of the generation of X_k since it is identical to Algorithm 2.

As illustrated in Figure 5, the Nomem Refresh algorithm selects the indexes in the following order (ignoring indexes smaller than 1):

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

$$|C| - X, \quad |C| - X + \sum_{k=M-1}^{M-1} (X_k + 1), \quad |C| - X + \sum_{k=M-2}^{M-1} (X_k + 1), \quad \ldots, |C|$$

Since this sequence is strictly increasing, the candidate log is accessed sequentially. There is no need for any in-memory data structure any longer. The algorithm requires slightly more processing power than Stack Refresh, since twice as many samples from the geometric distribution are computed.

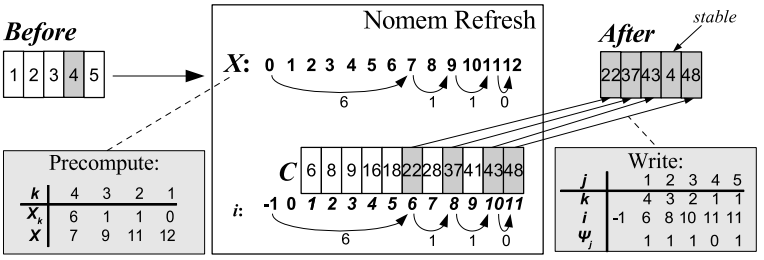

Fig. 5. Nomem Refresh

5 Deferred Sample Maintenance in a DBMS

Even though the candidate log file is smaller than the full log, there are situations in which full logging is the preferred technique. For example, the transaction log of a database system may already contain all the information we need. Alternatively, if we maintain a materialized view on the very same dataset the sample is built on, and if this view is refreshed using deferred maintenance too, the full log is typically maintained by the database system to incrementally refresh the materialized view, e.g., IBM DB2 makes use of a staging table and the Oracle RDBMS uses a materialized view log. Fortunately, we can apply the candidate refresh algorithms on a full log by using the same idea as used for the Nomem Refresh algorithm.

Each of the candidate refresh algorithms requires the size of the candidate log as its input for precomputing the final sample. If a full log is maintained, one does not know in advance how many tuples will be candidates and how many will be skipped. However, Vitter [4] defined a random variable describing the number of tuples skipped between two subsequent candidates. Thus, we store the state of the PRNG and compute the indexes of the candidates in advance (without actually storing them). Using this procedure, we can precalculate how many tuples of the full log are candidates. Then, we reset the random number generator and run an arbitrary candidate refresh algorithm. Every time the candidate log is accessed, we calculate the index of the respective candidate by computing Vitter's skips again and access the respective tuple of the full log. This procedure is nearly as efficient as if a candidate log were used. The only difference is that the tuples selected for the sample are further apart from each other, so that the number of blocks read from disk increases.

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

Another problem arising in the context of a DBMS is that there are updates and deletions. We show how our refresh algorithms can be extended to support these operations as well. First, we store all updates in a separate log file and apply all these updates *after* each refresh of the sample. The situation becomes more difficult if some elements are removed from the dataset. In this case, it is not possible to maintain a candidate log since insertions after a deletion are included in the sample with a different probability than assumed during candidate logging. Thus, we use a full log file if there are deletions. If we assume (or make sure) that the insertions and deletions are disjunctive, we first conduct all the deletions and afterwards process the full log using the techniques presented in this paper (using a potentially smaller sample size). We currently investigate how a reservoir sample can be maintained so that deletions are supported as well.

6 Experiments

We implemented the various refresh algorithms and conducted a set of experiments to evaluate their performance. We distinguish between online, offline and total cost of maintaining the sample. The online cost is the processing cost of arriving insertions. The offline cost mirrors the cost for refreshing the sample. The total cost is the sum of online and offline cost. This distinction is helpful since it captures different application areas. For example, in a streaming system, the online cost is important since it expresses the processing time for each operation within the sample operator. The refresh may be conducted by an independent system which has access to the log file, thereby not affecting online processing. In a DBMS, both logging and refresh are typically conducted by the very same system, so that the total cost is more important than the online cost. For clarity, we arrange the figures for online and total cost side by side so that they can be compared easily. Note that most of the plots have logarithmic axes.

Experimental results. We found that using a candidate log is significantly faster than refreshing the sample immediately or using a full log. When it comes to sample refresh, we found that the refresh algorithms using precomputation outperform the naive ones, and that the computational overhead of Nomem Refresh is negligible. The more operations occur between two consecutive refresh operations, the more is gained by using advanced refresh techniques. Our algorithms scale well, since the sample size has only a linear effect on the refresh costs. In comparison to the geometric file, our techniques are more efficient if the GF is not allowed to consume large amounts of memory for its internal buffer.

6.1 Experimental Setup

The experiments were conducted on an Athlon AMD XP 3000+ system running Linux with 2GB of main memory and an IDE hard drive with 7,200 RPM. We first measured the access times per block using a $1.6GB$ on-disk sample (with a cache of $100MB$). Our hard disk is formatted with the ext3 filesystem. It has a block size of 4096 bytes, and we assumed that each element occupies 32 bytes,

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

i.e., each block contains 128 elements. We found that a sequential read/write takes about $0.094ms$ per block, a random read $8.45ms$, and a random write $5.50ms$ (due to asynchronous writes). Now, for each algorithm, we counted the number of sequential/random reads and writes on a block-level basis. We then weighted these numbers with the access times above. This strategy allows for quantifying the cost of the single phases independently, while at the same time enabling us to run a large variety of different experiments.

All the algorithms have been implemented using the Java programming language and Sun's JDK version 1.5.0_03. For full refresh, we used the techniques described in Section 5. Unless stated otherwise, each experiment was run at least one hundred times and results were averaged. We assumed that the sample is refreshed periodically.

6.2 Online Cost

We first evaluated the online I/O cost of sample maintenance. We used a sample size of one $1M$ and inserted $100M$ elements into a dataset with initial size $1M$. Figure 6 shows the cumulated cost over time without any intermediate refreshes. Obviously, immediate refresh is far more expensive than writing to a log file. However, if the dataset size gets really large, immediate refresh is cheaper than writing to the full log, since the fraction of the candidate elements decreases over time. Candidate logging is the most efficient technique and is by several orders of magnitude faster than immediate refresh.

Next, we measured the online impact induced by different sample sizes (Figure 8). We used the same setting as in the former experiment, but plotted the cumulated cost after $100M$ operations. Clearly, the maintenance cost of the full log is independent of the actual sample size, while the cost for immediate refresh and candidate logging increases with an increasing sample size, since more candidates are generated if the sample is larger. However, candidate logging is always faster than full logging. In fact, the cost of writing the full log is an upper bound to the cost of writing the candidate log.

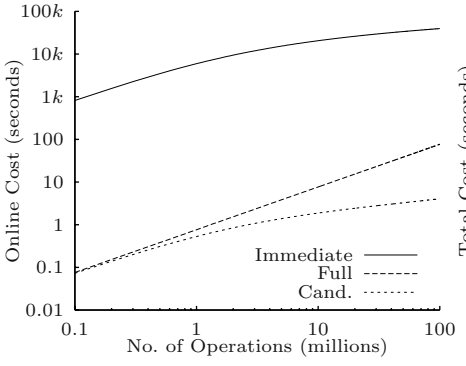

Fig. 6. Online cost over time

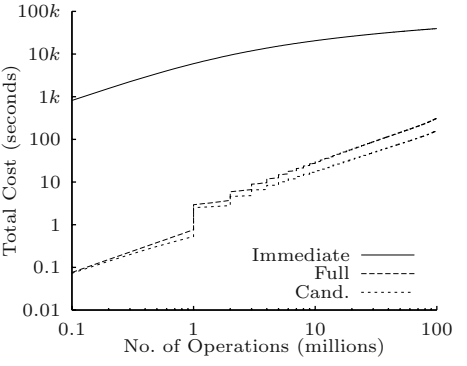

Fig. 7. Total cost over time

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

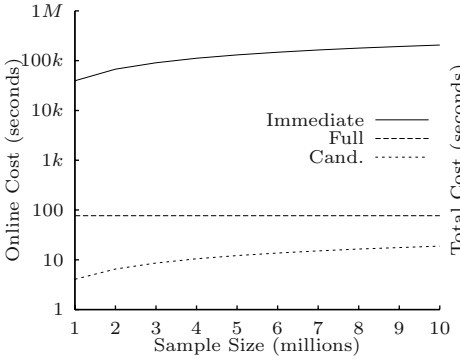

Fig. 8. Online cost and sample sizes

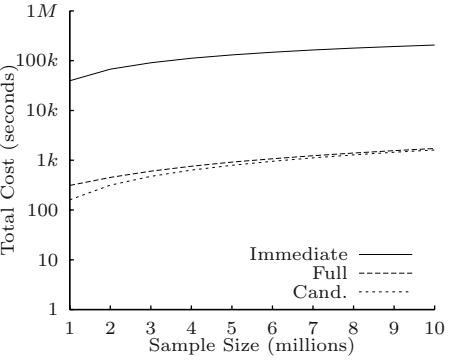

Fig. 9. Total cost and sample sizes

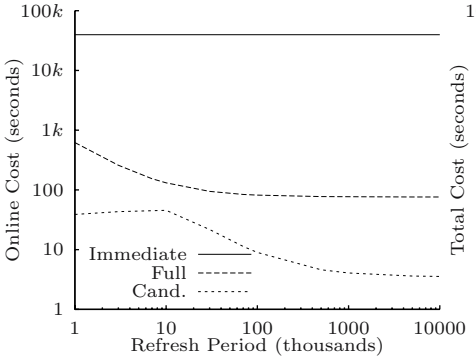

Fig. 10. Online cost and refresh period

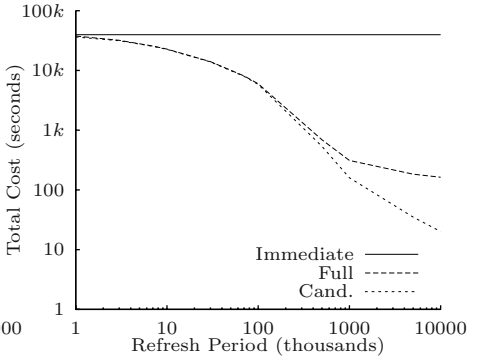

Fig. 11. Total cost and refresh period

Furthermore, we compared the online cost for different refresh periods (Figure 10). We used the same experimental setting as in the former experiments. The cost for maintaining the sample directly is independent of the refresh period (always 1). However, both candidate logging and full logging re-use the log file after a refresh so that one random I/O is performed to move from the current position to the beginning of the log file (otherwise, the costs are independent of the refresh period, too). Thus, with an increasing refresh period, these random I/Os occur less frequently and the cost drops. Again, candidate logging is faster than full logging. Note that if the refresh period is less than $10k$, the candidate log often consists of only a single block, which is the minimum.

6.3 Total Cost

We ran the same experiments as above again but now measured the total I/O cost (including refresh). Note that Array, Stack and Nomem Refresh have equal I/O cost. We refreshed the sample after every $1M$ insertions. As can be seen in Figure 7, deferred refresh is significantly faster than immediate refresh. The costs for full and candidate refresh are almost the same since we used the algorithm

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

described in Section 5 for full refresh. However, the costs for writing the log file are different, so that the candidate techniques are faster than the techniques using a full log. The I/O cost of the first few refreshes is magnified due to the log-log-plot.

Figure 9 illustrates that the total costs for maintaining the sample are increasing as the sample size increases. Again, deferred refresh significantly outperforms immediate refresh. The costs of full maintenance and candidate maintenance are almost equal if the sample is really large. However, we performed 100 million operations in every case. If the number of operations were larger, this effect would vanish.

As can be seen in Figure 11, deferred refresh is faster than immediate refresh if refreshes are not extremely frequent. Since the total costs are governed by the refresh cost, full and candidate maintenance strategies perform equally if the refresh period is short. However, the larger the refresh period gets, the more effort is saved by using a candidate log. Thus, the candidate strategies become more efficient than full refresh in this case.

6.4 Memory Consumption and Computational Cost

In this experiment, we measured CPU cost and memory consumption for the different implementations of deferred refresh. Even though the disk access pattern is the same for Array, Stack and Nomem Refresh, their CPU and memory costs are different. For the experiments, we used a sample size of $1M$ elements. We inserted elements until the number of candidates reached a certain size. Then, we refreshed the sample and measured the memory consumption and CPU cost. Note that computation and I/O are typically performed in parallel.

Figure 12 plots the consumed memory in dependency of the number of candidates. Array Refresh always maintains an array that has as many elements as the sample. However, the elements of the array are only 4 bytes long (index size), while the sample elements are usually larger. The Stack Refresh algorithm requires more and more memory as the number of candidates in the log file increases. Note that the figure includes extreme cases, e.g., in which the number of candidates is more than twice the sample size. Thus, the memory consumption of Stack Refresh is small in most cases. However, Nomem Refresh does not consume any memory. We plotted the size of the in-memory buffer of the geometric file for expository reasons. The number of candidates in a geometric file can only grow as large as its internal buffer. Thus, if we want to delay the refresh to, say, $100,000$ (final) candidates, the buffer of the geometric file has to be as large as 10% of the sample.

In Figure 13, we plot the CPU time for a refresh in the same experimental setting. Clearly, Stack Refresh is the fastest method. For small candidate logs, Array Refresh is more efficient than Nomem Refresh, while the opposite is true for large log files (due to the sort operation of Array Refresh). Nomem Refresh has to compute $2M$ random numbers to select the final candidates. To minimize the total CPU time, we propose the following strategy: If the expected number of final candidates ($E(\Psi)$) is small (say, $< 4k$), we use the Stack Refresh algorithm. Otherwise, we use Nomem Refresh to save main memory.

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

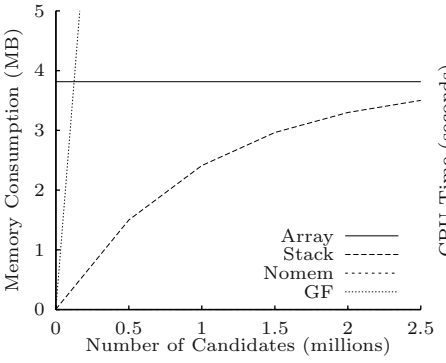

Fig. 12. Memory consumption

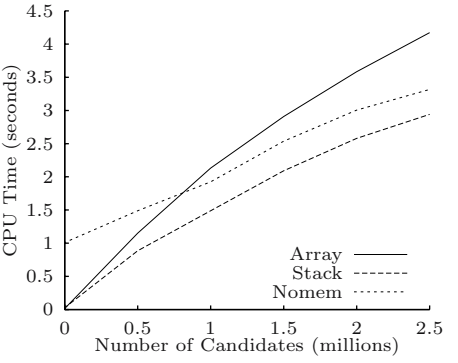

Fig. 13. Computational cost

6.5 Comparison to the Geometric File

The GF [7] is the only algorithm known to the authors which targets deferred maintenance of a disk-based sample. In this section, we briefly point out the differences between our algorithms presented and the GF. First, the GF buffers arriving insertions in main memory. In contrast to our algorithms, the buffer is accessed randomly and therefore cannot be serialized to disk without losing performance. Additionally, the GF keeps a part of the sample in memory to optimize the I/O cost, i.e., the on-disk part of the sample is not uniform. This may be problematic in the case of system failures.

Using the GF, one is not able to conduct a refresh at an arbitrary time. In fact, the sample is only refreshed if the buffer reaches its full size. Consequently, one may either control the desired buffer size or the frequency of refresh operations, but not both. To compare our refresh algorithms to the GF,[5] we proceeded as follows: First, we refreshed the sample every time the GF issued a refresh. Thus, the number of refreshes conducted by the GF and by our techniques is equal. Second, we assumed that our algorithms may use the same amount of in-memory buffer as the GF. We used this buffer to store a part of the sample in memory, thereby reducing the number of disk accesses. In fact, if we store 5% of the sample in memory, we expect that the refresh cost drops by 5%.

Again, we set the sample size to $1M$ elements and inserted $100M$ elements. We measured the total cost for different buffer sizes. The results are shown in Figure 14. Clearly, the larger the buffer, the less cost is incurred by the algorithms, since the cumulative number of refreshes is decreasing. If the buffer is less than 3% of the sample size, both full and candidate refresh are faster than the GF. If we increase the buffer to up to 4% of the sample size, the GF is faster than full refresh but slower than candidate refresh. If the buffer is larger than 4% of the sample size, the geometric file is the most efficient algorithm. Thus, the optimal strategy depends on the amount of memory we are willing to sacrifice, and on the desired flexibility of deciding on refresh periods.

[5] We used block-aligned segments and set $\beta = 32k$.

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

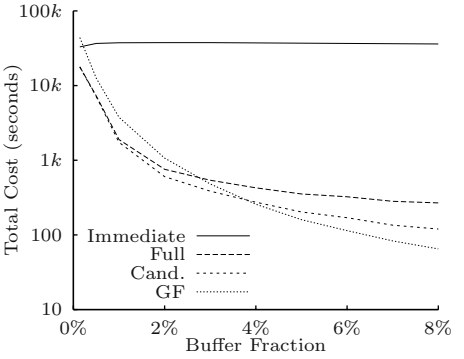

Fig. 14. GF buffer size & total cost

7 Summary

We developed a set of algorithms which allow for deferred maintenance of random samples of an arbitrary dataset. We introduced a novel type of log file which minimizes the amount of data used to track changes on the underlying dataset. We showed that such a log file imposes far less overhead in processing arriving operations than traditional log files and immediate sample maintenance. Furthermore, we developed different strategies to efficiently process the log file in order to update the sample. We optimized our algorithms so that they rely on fast sequential disk access only, while the number of read and write operations is minimized. Additionally, we showed how main memory consumption can be avoided at the cost of some CPU time. Finally, we conducted a set of experiments indicating that our algorithms are more efficient than any known algorithm using a small amount of in-memory data structures only.

Acknowledgement. This work has been supported by the German Research Society (DFG) under LE 1416/3-1. We like to thank the anonymous reviewers, S. Schmidt, and P. Rösch for their helpful comments on a previous version of the paper.

References

1. Haas, P., König, C.: A Bi-Level Bernoulli Scheme for Database Sampling. In: Proc. ACM SIGMOD. (2004) 275–286
2. Gupta, A., Mumick, I.S., eds.: Materialized Views: Techniques, Implementations, and Applications. MIT Press (1999)
3. Vitter, J.S.: Faster Methods for Random Sampling. Commun. ACM **27** (1984) 703–718
4. Vitter, J.S.: Random Sampling with a Reservoir. ACM TOMS **11** (1985) 37–57
5. Haas, P.J.: Data Stream Sampling: Basic Techniques and Results. In: Data Stream Management: Processing High Speed Data Streams, Springer (2006) (to appear).

Final edited form was published in "Advances in Database Technology - EDBT 2006. München 2006", S. 423-441,
ISBN 978-3-540-32961-9
https://doi.org/10.1007/11687238_27

6. Tatbul, N., Çetintemel, U., Zdonik, S.B., Cherniack, M., Stonebraker, M.: Load Shedding in a Data Stream Manager. In: Proc. VLDB. (2003) 309–320
7. Jermaine, C., Pol, A., Arumugam, S.: Online Maintenance of Very Large Random Samples. In: Proc. ACM SIGMOD. (2004) 299–310
8. Ganti, V., Lee, M.L., Ramakrishnan, R.: ICICLES: Self-Tuning Samples for Approximate Query Answering. In: The VLDB Journal. (2000) 176–187
9. Chaudhuri, S., Das, G., Datar, M., Narasayya, R.M.V.R.: Overcoming Limitations of Sampling for Aggregation Queries. In: Proc. ICDE. (2001) 534–544
10. Acharya, S., Gibbons, P.B., Poosala, V., Ramaswamy, S.: Join Synopses for Approximate Query Answering. In: Proc. ACM SIGMOD. (1999) 275–286
11. Acharya, S., Gibbons, P.B., Poosala, V.: Congressional Samples for Approximate Answering of Group-By Queries. In: Proc. ACM SIGMOD. (2000) 487–498
12. Babcock, B., Chaudhuri, S., Das, G.: Dynamic Sample Selection for Approximate Query Processing. In: Proc. ACM SIGMOD. (2003) 539–550
13. Olken, F., Rotem, D.: Maintenance of Materialized Views of Sampling Queries. In: Proc. ICDE. (1992)
14. Matsumoto, M., Nishimura, T.: Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. ACM TOMACS **8** (1998) 3–30