Title
Random Sampling from Hash Files

Authors
Olken, Frank
Rotem, D.
Xu, P.

Publication Date
1989-12-01

LBL-28566

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

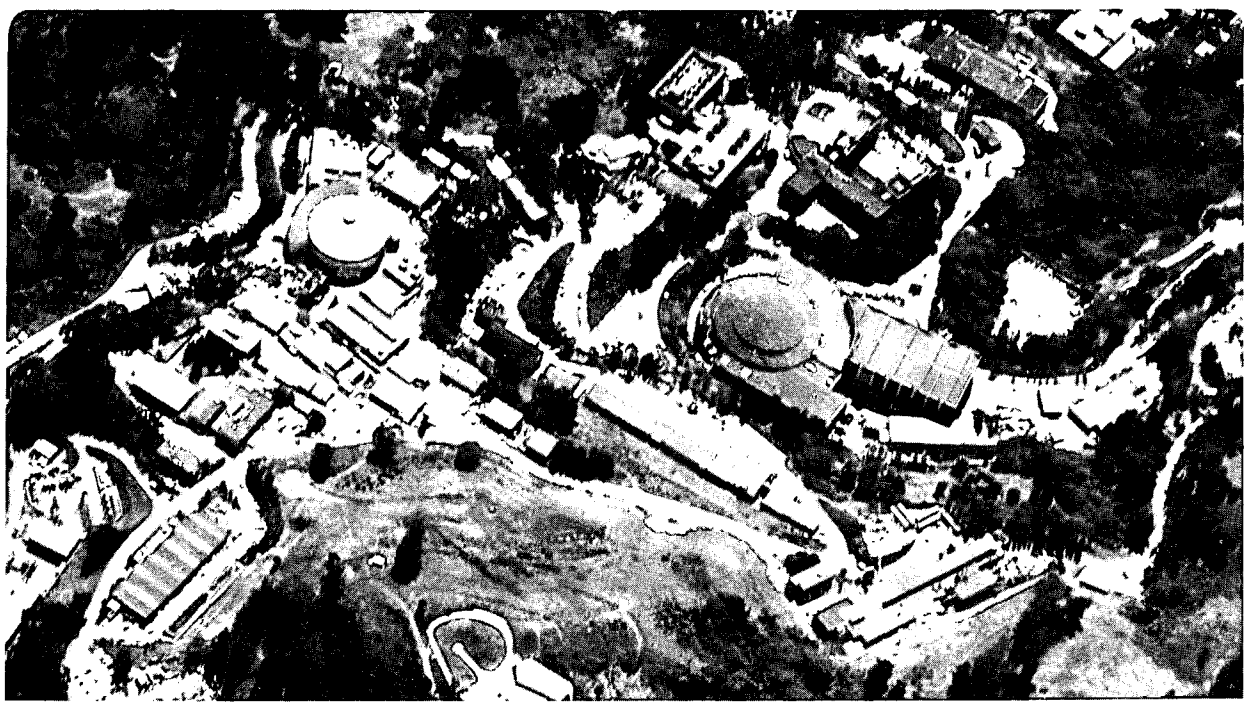
Information and Computing Sciences Division

To be presented at SIGMOD 1990, Atlantic City, NJ, May 23–25, 1990, and to be published in the Proceedings

Random Sampling from Hash Files

F. Olken, D. Rotem, and P. Xu

December 1989



Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

DISCLAIMER

LBL-28566

RANDOM SAMPLING FROM HASH FILES

Frank Olken & Doron Rotem

Computing Science Research & Development
Information & Computing Sciences Division
Lawrence Berkeley Laboratory
1 Cyclotron Road
Berkeley, California 94720

and

Ping Xu

Computer Science Department
San Francisco State University
San Francisco, CA

December 1989

This work was supported by the Office of Health and Environmental Research Program and the Director, Office of Energy Research, Applied Mathematical Sciences Research Program, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

Random Sampling from Hash Files *

Frank Olken and Doron Rotem
Computer Science Research & Development Dept.
Information and Computing Sciences Div.
Lawrence Berkeley Laboratory
1 Cyclotron Road, Berkeley, CA 94720

Ping Xu
Computer Science Dept.
San Francisco State University
San Francisco, CA

December 1989

Abstract

In this paper we discuss simple random sampling from hash files on secondary storage. We consider both iterative and batch sampling algorithms from both static and dynamic hashing methods. The static methods considered are open addressing hash files and hash files with separate overflow chains. The dynamic hashing methods considered are Linear Hash files [Lit80] and Extendible Hash files [FNPS79]. We give the cost of sampling in terms of the cost of successfully searching a hash file and show how to exploit features of the dynamic hashing methods to improve sampling efficiency.

1 Introduction

In this paper we discuss simple random sampling from hash files on secondary storage. We consider both iterative and batch sampling algorithms from static and dynamic hash files. This is a continuation of our research on sampling from databases [OR86, OR89].

The main contribution of this paper is to show that one can introduce simple random sampling of hash files without substantial modification to the data structures or substantial increase in normal costs of accessing or updating the hash files. We provide detailed cost formulae, supporting simulations, and we show the relationship of sampling costs to the cost of searching the same data structures.

* This work was partially supported by the Director, Office of Energy Research, Office of Basic Energy Sciences, Applied Mathematical Sciences Division of the U.S. Department of Energy under Contract DE-AC03-76SF00098.
Authors electronic mail addresses: olken@lbl.gov, rotem@csam.lbl.gov

We first consider static hash files of two types: open addressing (any method which rehashes bucket overflow into the primary area) and separate overflow chaining (in which each primary bucket has a separate chain of overflow pages). See [Knu73] for a detailed exposition and analysis of these hashing methods.

The many dynamic hashing methods can be classified according to whether or not they employ some sort of directory. We consider one method from each class: Linear Hashing by Litwin [Lit80] (no directory) and Extendible Hashing by Fagin [FNPS79] (directory).

For each hash file we consider iterative methods, which repeatedly extract a sample of size one until they accumulate a sample of the requisite size. We then consider batch sampling methods, which are modelled on batch retrieval methods, treating batch sampling of open addressing hash files in detail. Batch sampling avoids rereading of the same page twice, which can occur in iterative sampling. We also discuss the use of sequential scan sampling methods.

For both of the dynamic hashing methods we consider both naive sampling methods and more sophisticated methods which exploit the structure of the dynamic hash file, i.e., two-file method for Linear Hashing and double acceptance/rejection sampling for Extendible Hashing. We show that the more sophisticated methods have better performance.

We commence with a discussion of the motivation for this work.

1.1 Why sample?

Obtaining information, whether from a database or the real world requires time and effort. Often, we do not need exact answers to our questions. In some cases we can obtain the approximate answer we need from a random sample of a population, in lieu of examining the entire population. This may provide substantial savings in time and expense. In a database context, the savings may arise not just in terms of the cost of retrieving the records from the database, but also from sample "post-processing" costs.

For large administrative and scientific databases retrieval costs can be significant. For example, social security and tax record databases contain tens of millions of records. High energy physics experimental datasets often contain hundreds of gigabytes of data. Even if the datasets are smaller, real-time computing systems may impose harsh time constraints which preclude extensive retrievals (see [HOT89]).

Even if retrieval costs were negligible, sampling would still be important in order to reduce sample post-processing costs. Some of these costs may arise from extensive computations on each

record (e.g., in physics an event record may contain 100KB of data, and require 100M instructions to process).

Yet even if computing were free, sampling would still be important for those applications which require physical inspection of the real world objects which correspond to the sampled database records. The most ubiquitous application concerns the use of sampling to audit financial databases [Ark84, LTA79], which must be done annually for most such databases. Other common applications include inspection and/or testing of components for quality control [Mon85, LWW84], and medical examinations of sampled patients for epidemiological studies.

Random sampling is typically used to support statistical analysis of a dataset, either to estimate parameters of interest [HOT88, HOT89] or for hypothesis testing. Cochran [Coc77] provides a classic treatment of the statistical methodology. Applications include control systems, scientific investigations, product quality control, and policy analyses. For example, one might want a sample of bank interest records to check against tax records to estimate tax fraud rates.

We should note that the accuracy of estimates from samples is typically a function of size of the sample, with little dependence on the population size. Hence sampling is most advantageous when sampling from large populations, as would be found in very large administrative and scientific databases.

1.2 Why put sampling in DBMS?

The fact that users want to sample data from databases does not necessarily imply that it should be included in the DBMS. Conceivably, sampling could be performed outside the database on the results of a query, as it is now done.

However, we expect that putting sampling operators within the DBMS can yield major gains in efficiency for many sampling queries, at negligible cost to normal operations. The efficiency gains arise from the reduction in the amount of data data to be retrieved for sampling queries, and by exploiting indices and access methods used in the DBMS. Instead of completely processing a database query and then sampling the result, we can, in effect, interchange the sampling and query operators, so that we sample prior to query evaluation. In one earlier paper [OR86] we discussed query processing algorithms which permit sampling the results of single relational operators. In a second paper [OR89] we discussed the sampling from base relations organized as B^+ trees (the most common access method in use). In this paper we turn to sampling from hash files, probably the second most common type of access method.

Sampling can also be used in the DBMS to provide estimates of the answers of aggregate queries, in applications where such estimates may be adequate (e.g. policy analysis), and where the cost in time or money to fully evaluate the query may be excessive. In his dissertation [Mor80], Morgenstein discussed this issue of estimation procedures for various aggregate queries (e.g., count) while mentioning sampling procedures only briefly. More recently, Hou [HOT88] has discussed the construction of statistical estimators for arbitrary relational expressions for COUNT aggregates. He also envisions their application to real-time applications [HOT89].

Sampling may also be used to estimate database parameters used by the query optimizer to choose query evaluation plans. In [Wil84], Willard discusses the determination of the asymptotically optimal sample size for estimating the selectivity of a selection query. In [LN89], Lipton and Naughton discuss the use of sampling to estimate the size of transitive closures.

Finally, sampling has been proposed [Den80] as a means of providing security for individual data, while permitting access to statistical aggregates.

1.3 Why sample hash files?

Given that we have decided to put sampling operators into our DBMS, why should we concern ourselves with sampling from hash files? The answer is that sampling is akin to a selection operator, in that the query optimizer will attempt to move down in the query evaluation plan graph toward the base relations [OR86]. Hence we will often find ourselves applying sampling to base relations.

Hash files are very common primary access methods for base relations in administrative and transaction processing database systems, e.g. bank records may hashed on account number, payroll records hashed on social security number. Second, hash files are used as join indices to support join processing and transitive closure computations [Lu87], etc.

1.4 Organization of Paper

The remainder of the paper is organized as follows. In Section 2 we review acceptance/rejection sampling. In Section 3 we discuss sampling from open addressing hash files, In Section 4 we treat sampling from separately chained overflow hash files, In Section 5 we examine sampling from Linear Hash [Lit80] files. In Section 6 we present sampling from Extendible Hash [FNPS79] files. Batch and sequential scan sampling methods are discussed in Section 7. We present our experimental results (simulations) in Section 8. Finally, Section 9 contains our conclusions.

2 Basic Sampling Techniques

2.1 Simple Random Sampling

In this paper we shall be concerned with *fixed size simple random samples. Fixed size* indicates that the target sample size has been specified by the user. *Simple random sample* indicates that we want uniform inclusion probabilities for each record.

We shall assume that the sample size is small compared to the number of records in the file and ignore the costs of removing duplicates (and increasing the sample size) to convert from samples without replacement to samples with replacement. Unless otherwise noted, whenever we refer to a sample we mean a *simple random sample with replacement.*

Throughout the paper, we measure the cost of algorithms in terms of the number of disk blocks read.

2.2 Acceptance/Rejection Sampling

Acceptance/rejection (A/R) sampling forms the basic sampling strategy used throughout this paper. In this paper we will use it to compensate for algorithm or data structure induced variations in sample inclusion probabilities so as to finally obtain a *simple random sample*, i.e., one with uniform inclusion probabilities. We briefly describe this classic sampling technique here for those in the database community who may be unfamiliar with it.

Suppose that we wish to draw a weighted random sample of size 1 from a file of n records, with inclusion probability for record r_j proportional to the weight w_j. The maximum of the w_j is denoted by w^*.

We can do this by generating a uniformly distributed random integer, j, between 1 and n, and then accepting the sampled record r_j with probability p_j:

$$p_j = \frac{w_j}{w^*} \tag{1}$$

The acceptance test is performed by generating another uniform random variate, u_j, between 0 and 1 and accepting r_j if $u_j < p_j$. If r_j is rejected, we repeat the process until some r_j is accepted.

The reason for dividing w_j by w^* is to assure that we have a proper probability (i.e., $p_j \leq 1$). If we do not know w^* we can use instead a bound Ω such that $\forall j, \Omega > w_j$. It is well known that the number of iterations required to accept a record r_j will be geometrically distributed with a mean of $(E[p_j])^{-1}$. Hence using Ω in lieu of w^* results in a less efficient algorithm.

α_i	acceptance probability of record i
α	$= E(\alpha_k)$ = expected acceptance probability of record
b_i	bucket occupancy for bucket i
$\bar{b}$	$= n/m$ = average hash bucket occupancy (records)
b^*	maximum hash bucket occupancy (records)
c_i	occupancy (records) of $i'th$ directory cell (EXH)
d_i	occupancy (records) of $i'th$ data page (EXH)
h_i	chain length for bucket i (pages)
$\bar{h}$	average chain length (pages)
h^*	maximum chain length (pages)
$C_{method}(s)$	expected cost of retrieving sample of size s, via specified method
m	number of buckets in the file
n	number of records in file
p_k	probability of inclusion of record k
s	number of records desired in sample
s'	inflated sample size (to compensate for acceptance/rejection)

Table 1: Notation used in the paper.

We rely heavily on Acceptance/rejection sampling because it is well-suited to situations where the weights are frequently updated (since it does not require any auxiliary indices).

2.3 Notation

Throughout the paper, for a variable x, we will use $\bar{x}$ to denote the average of x and x^* to denote the maximum of x, for any quantity x.

3 Open Addressing Hash Files

In this section we discuss how to sample from open addressing hash files [Knu73], i.e., those hash files in which overflow records are rehashed into the primary file. We discuss iterative methods, which repeatedly extract a sample of size one until they have accumulated sufficient size sample.

OA	Open addressing [Knu73]
SO	Separate Overflow Chaining [Knu73]
LI	Linear Hash files [Lit80]
EX	Extendible Hash files [FNPS79]

Table 2: Hash File abbreviations

Batch methods, which are based on batch retrieval, are discussed later, in Section 7.

For our analysis we adopt the *uniform hashing* model [Knu73, pp. 527-528], which assumes that the hashing functions randomize the placement of records in buckets.

For the purpose of sampling, an open addressing hash file may be viewed simply as a variably blocked file, irrespective of the particular hash function used to place the records into pages. Thus these sampling methods are generally applicable to any type of file for which the number of records per page varies. This may arise either because the individual record sizes vary (with a fixed block size), or because updates to the file have resulted in variable numbers of records/block, or because hashing has been used to place records in blocks.

3.1 Iterative Algorithm

Given a hash (variably blocked) file, which contains n records, stored in m contiguous buckets on disk, where the $i'th$ bucket contains b_i records. We denote by b^* the maximum of the b_i's. In Figure 1 we describe an acceptance/rejection algorithm, ARHASH, for obtaining a single random sample from such a file. This procedure must be repeated s times to obtain a sample of size s.

The next lemma shows that algorithm ARHASH gives every record the same inclusion probability.

Lemma 1 *Each record has an equal probability of $\frac{1}{n}$ of being chosen into the sample by algorithm ARHASH.*

Proof: A specific record will be accepted during a single iteration of the **while** loop if its bucket has been chosen and it is selected within that bucket. This event occurs with probability p where:

$$p = \frac{1}{m}\frac{1}{b^*} \tag{2}$$

```
procedure ARHASH ;
            /* This procedure samples one record
            from a variably blocked file */
comment accepted is a boolean variable which
          indicates when a sample was accepted.
Set accepted to false;
while accepted=false do
      /* generate a random no. between 1 and m */
      r := RAND(1,m);
      /* generate a random no. between 1 and b_max */
      j := RAND(1,b*);
      Read bucket r;
      if j ≤ b_r then
            accept the j'th record of bucket r into the sample
            and set accepted to true ;
endwhile.
```

Figure 1: Algorithm for sampling from a hash file (variably blocked)

Since this probability is the same for all records in the file we conclude that the probability P of accepting *some* record in the first execution of the loop is:

$$P = \frac{n}{m \times b^*} \tag{3}$$

We denote by Q the probability of rejecting a bucket in this iteration, i.e., $Q = 1 - P$. A specific record, will be accepted during the $i'th$ execution of the loop if no record is accepted for the first $i - 1$ executions of the loop followed by an acceptance on the $i'th$ execution. The probability of this event is $Q^{i-1}p$. Summing for i between 1 and infinity we get

$$p\frac{1}{1-Q} = \frac{p}{P} = \frac{1}{n} \tag{4}$$

as required. □

Lemma 2 *The expected number of disk accesses to obtain one random sample is*

$$\frac{m \times b^*}{n} = b^*/\bar{b} \tag{5}$$

Proof: As was shown in the previous lemma, the probability of accepting some record in each execution of the loop is P. Therefore the number of reads until a record is accepted is a random variable with geometric distribution with parameter P. The expected value for this random variable is $\frac{1}{P} = \frac{m \times b^*}{n}$ as required. □

4 Separate Overflow Chain Hash Files

In this section we consider sampling from hash files which have separate overflow chains for each bucket in the primary file area [Knu73, pp. 535]. We use *bucket* to refer to the hash partition function, and *primary (overflow) page(s)* to refer to the primary (overflow chain) page(s) of a bucket.

4.1 Iterative Algorithm

The iterative algorithm selects a bucket at random, then does acceptance/rejection test with acceptance probability, $\alpha_i = b_i/b^*$ as in the open addressing hash file. If the bucket is accepted we must then sample one record from the bucket, this may require reading some of the overflow pages. We repeat this until we obtain the desired sample size.

Let d^* = the maximum number of records per page and b^* = the maximum number of records per hash bucket.

Theorem 1 *Consider a hash file with chained overflow (separate or common), which stores a count of the records in a bucket in the primary page for the bucket, Let $S(H,1)$ = the expected cost of a successful search of hash file H for a single record. The expected cost of a simple random sample of size one from hash table H is $C(H,1)$:*

$$C(H,1) = (\frac{b^*}{\overline{b}} - 1) + S(H,1) \tag{6}$$

Proof: The first term, $(\frac{b^*}{\overline{b}} - 1)$ is the cost of the rejected buckets. It is the expectation of a geometric distribution with success probability equal to the average acceptance probability, $\overline{b}/b^*$ minus the cost of the first page read in a successful search.

Once we have accepted the bucket, $S(H,1)$ gives us the expected search cost within the bucket. It is equal to the expected cost of a successful search because accepted records have been chosen at random (uniformly) from the entire file. □

Corollary 1 *The expected cost of iterative sampling of size 1 from separate overflow hash files is:*

$$C_{SOI}(H,1) = (\frac{b^*}{\overline{b}} - 1) + S_{SOI}(H,1) \tag{7}$$

where $S_{SOI}(H,1)$ is the expected cost a successful search of a hash file with separate overflow chaining:

$$S_{SOI}(H,1) = 1 + (1/\bar{b}) \sum_{k=1}^{\infty} k \sum_{j=1}^{d^*} \frac{(k-1)d^*}{2} + jP(kd^* + j, \bar{b}) \tag{8}$$

where $P(i,\lambda)$ is the Poisson distribution:

$$P(i,\lambda) = \frac{e^{-\lambda}\lambda^i}{i!} \tag{9}$$

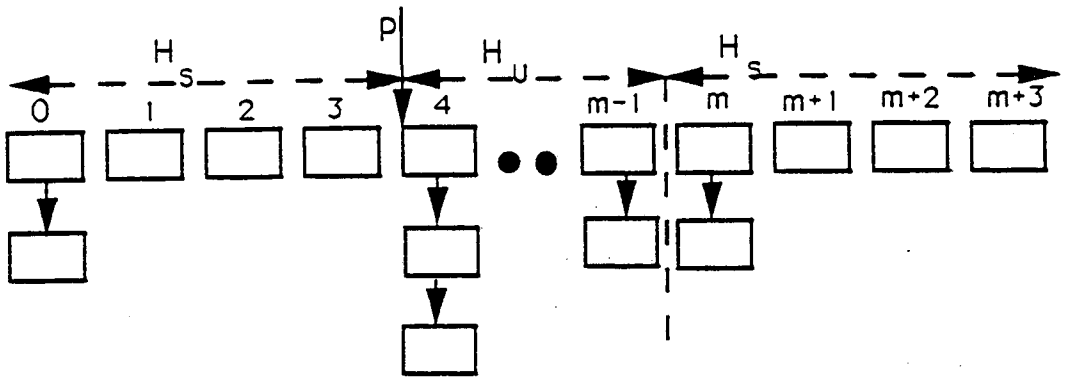
Proof: Result follows from Theorem 1 and from the derivation of $S(H)$ given by [Lar82]. Note that our notation differs from Larson. □

5 Linear Hashing

In this section we consider sampling from Linear Hash files [Lit80]. We first give a brief overview of the method. Linear hash files are based on static separately chained overflow hash files. Initially the file has m buckets in the primary area numbered from 0 to $m-1$. For simplicity, assume that a key k is hashed into a bucket using the hashing function $h_1(k) = k \bmod m$. As the loading of the file increases we gradually split buckets in order from bucket 0 to $m-1$. The decision whether to perform a split is based on a *split criterion* set by the designer which is evaluated after each key insertion. Splitting of buckets continues as long as the *split criterion* is *true*. An example of such a *split criterion* is "maximum length of an overflow chain exceeds 3 pages".

Now we describe a single split operation. The split pointer P initially points to bucket 0 and is incremented by 1 after every split so that it always points at the next bucket to be split. The split operation of bucket i consists of creating a new bucket numbered $i+m$ (appending it to the end of the file) and rehashing all the records in the original bucket i into either bucket i or $i+m$ using a new hash function $h_2(k) = k \bmod (2m)$. When all the original buckets have been split, the file has doubled in size. After each doubling of the file, the pointer P is reset to point at bucket 0 and the two hashing functions used are set to: $h_1(k) = k \bmod (2^j m)$ and $h_2(k) = k \bmod (2^{j+1}m)$, where j is the number of file doublings which have occurred.

This naturally leads us to model a Linear Hash file as two distinct separate overflow chain hash files, H_s and H_u where H_s is comprised of the split buckets, and H_u comprised of the unsplit buckets. These two hash files differ in the number of buckets, and the average bucket loading. Let m_s and m_u denote the number of buckets in H_s and H_u respectively and let n_s and n_u denote the number of records in these two files. See Figure 2.



A linear hash file with in which the first 4 buckets have been split
The dashed arrows indicate the bucket region of each subfile

Figure 2: Drawing of Linear Hash File

We have two ways of sampling from the linear hash file. In the 1-file method we treat the entire hash file as a single variably blocked file. In the 2-file method we sample from the two subfiles, H_s and H_u separately, taking advantage of their different structure.

5.1 One-file Method

The 1-file method is essentially ARHASH algorithm for variably blocked files applied to hash buckets. Upon accepting a bucket, we must select a single record from the bucket at random. This may entail additional accesses to overflow pages. As for ARHASH, the 1-file method requires that we maintain b^*, the maximum bucket occupancy.

Theorem 2 *The expected cost of iterative sampling a sample of size 1 from a Linear Hash file using the one-file method is:*

$$C_{LHI_1}(H,1) = (\frac{b^*}{\bar{b}} - 1) + (\frac{n_u}{n})S_{SOI}(H_u,1) + (\frac{n_s}{n})S_{SOI}(H_s,1) \quad (10)$$

where $S_{SOI}(H,1)$ is the expected cost of a successful search of a hash file with separate overflow chains, given in Equation 8.

Proof: The first term $(\frac{b^*}{b} - 1)$ is simply the expected number of rejected buckets. The second and third terms are the weighted average of the cost of searching in the split and unsplit hash files for accepted records. □

5.2 Two-file Method

The 2-file method requires that we maintain the counters n_s and n_u, m the number of buckets, $b_s{}^*$, the maximum bucket occupancy of split buckets, $b_u{}^*$, the maximum bucket occupancy of unsplit buckets, and finally the pointer P whose value partitions the split and unsplit buckets and hence determines m_s and m_u.

To obtain a sample of size 1 with the iterative 2-file method we randomly choose one of the files H_s or H_u with probability $\frac{n_s}{n}$ and $\frac{n_u}{n}$ respectively, and then proceed to sample from that file.

Theorem 3 *The expected cost of iterative sampling a sample of size 1, from a Linear Hash file using the two-file method is:*

$$C_{LHI_2}(H,1) = (\frac{n_u}{n})C_{SOI}(H_u,1) + (\frac{n_s}{n})C_{SOI}(H_s,1)] \tag{11}$$

Proof: The cost is a weighted average of the cost of sampling from the split and unsplit sub-files, where the weights are the probability of choosing the corresponding subfile. Thus, the first term is the probability of selecting the sub-file of unsplit buckets times the cost of iteratively sampling from a separate overflow chained hash file with corresponding number of blocks equal to the number of unsplit buckets, and population equal to the number of records in the unsplit portion of the file. The second term accounts for the sub-file of unsplit buckets. □

Substituting for C_{SOI} from Corollary 1 yields:

$$C_{LHI_2}(H,1) = (\frac{n_u}{n})\left((\frac{b_u{}^*}{\overline{b_u}} - 1) + S_{SOI}(H_u,1)\right) \tag{12}$$

$$+(\frac{n_s}{n})\left((\frac{b_s{}^*}{\overline{b_s}} - 1) + S_{SOI}(H_s,1)\right) \tag{13}$$

Theorem 4 *The expected cost of iterative sampling a linear hash file with the 2-file method is always less than or equal to the expected cost of sampling with the 1-file method.*

$$C_{LHI_1}(H,1) \geq C_{LHI_2}(H,1) \tag{14}$$

Proof: Subtracting the two cost formulae gives:

$$C_{LHI_1} - C_{LHI_2} = (\frac{b^*}{\overline{b}}) - \left(\frac{n_u}{n}\frac{b_u{}^*}{\overline{b_u}} + \frac{n_s}{n}\frac{b_s{}^*}{\overline{b_s}}\right) \tag{15}$$

Substituting $\overline{b_u} = n_u/m_u$, $\overline{b_s} = n_s/m_s$, $\overline{b} = n/m$,

$$C_{LHI_1} - C_{LHI_2} = \left(\frac{m}{n} - \left(\frac{m_u}{n}\frac{b_u^*}{b^*} + \frac{m_s}{n}\frac{b_s^*}{b^*}\right)\right) b^* \tag{16}$$

Since $b_u^*/b^* \leq 1$ and $b_s^*/b^* \leq 1$ we have the following bound:

$$C_{LHI_1} - C_{LHI_2} \geq \left(\frac{m}{n} - \frac{(m_u + m_s)}{n}\right) b^* \tag{17}$$

$$\geq (m/n - m/n)b^* \tag{18}$$

$$\geq 0 \tag{19}$$

$$C_{LHI_1} \geq C_{LHI_2} \tag{20}$$

Q.E.D. □

As we have seen, the difference in performance of the 1-file and 2-file methods arises from excessive rejections by the 1-file method due to large difference in the bucket occupancy between the two files.

6 Extendible Hashing

In this section we consider Extendible Hash (EX) tables as described by Fagin, et al. in [FNPS79]. In order to make this presentation self-contained we provide here a brief overview of the method while introducing our notation for the parameters which are relevant for sampling.

An Extendible Hash file consists of data pages in which the records are stored, and a directory D which is an array of pointers such that each entry $D[j]$ contains a pointer to a data page, which we denote as p_j. Since more than one directory entry may point to a data page, the data page may have multiple names in our notation. Depending on its size, the directory may be either memory or disk resident. The size of D is controlled by a parameter called *directory depth* denoted by dd which is set initially by the designer to some value and is incremented (or decremented) dynamically as the file grows or shrinks. The number of entries in the directory D is set to 2^{dd}. A record is inserted (and searched) by applying a hash function h to its key k such that $h(k)$ is a number between 0 and $2^{dd} - 1$ and then following the pointer in $D[h(k)]$ to the required page.

Each data page p_i (i.e., page pointed at by directory entry $D[i]$) contains in addition to its records two counters, d_i and pd_i. The first counts how many records reside on the page and the

second is called *page depth* and its significance will be explained below. Initially when the file is empty, all directory entries point to a single empty page in which both these counters are set to 0.

When page p_i becomes full it is split by moving some of its keys to a newly created page called its twin page. The idea is to always keep on the same page all the records with keys k which agree on their first (most significant) pd_i binary digits in $h(k)$. For this reason, each time a page p_i is split, the value of pd_i is incremented by 1 and this new value is also assigned as the page depth of the twin page. The records moved to the twin page are exactly those whose key k has a 1 in the $pd_i'th$ most significant binary digit of $h(k)$.

This movement of records must also be reflected in the directory D so that exactly half of the directory entries containing pointers to page p_i are set to point to the twin page. These are all $D[x]$ pointing to page p_i ($D[x] = D[i]$) such that the $pd_i'th$ binary digit of x is equal to 1.

As the file become more heavily loaded the data pages are repeatedly split so that eventually data pages are pointed at by a single directory entry. When such a page overflows, we are forced to double the size of the directory. This is done by incrementing dd (directory depth) and splitting each previous entry into two entries by copying the pointer in it to both copies.

For the purpose of sampling we are interested in one additional quantity, namely, the number of entries which point at page p_i. We denote this quantity by g_i. The value of g_i can be easily computed from the previously defined counters as follows:

$$g_i = 2^{dd-pd_i} \tag{21}$$

The reason for this is that initially $g_i = 2^{dd}$, and each time a page is split its page depth is increased by 1 and the number of entries pointing at it is reduced by a half.

When we sample from Extendible Hash files we need to access data pages via the directory D and therefore always start by picking a random directory entry $D[j]$. For simplicity we will assume here that the directory is in memory, otherwise our costs have to be adjusted for disk accesses to the directory.

We now examine two ways of proceeding with acceptance/rejection sampling: double A/R page sampling, and A/R cell sampling.

6.1 Double A/R page sampling

As mentioned above, we start by picking a random directory entry $D[j]$. As usual, we want to sample from a page p_j with probability proportional to the number of records on it, d_j. However,

as we noted earlier, a single page may be pointed at by many directory entries so that we would oversample from pages which are pointed at by many entries. For that reason we accept a page p_j with probability proportional to d_j but inversely proportional to g_j (the number of directory cells which point at it). We therefore accept page p_j with probability α_j:

$$\alpha_j = \frac{(d_j/g_j)}{(d_j/g_j)^*} \tag{22}$$

The denominator is the maximum of the ratio in the numerator taken over all pages, it appears in this expression to assure that α_j is a probability, i.e., $\alpha_j \leq 1$. If the page p_j is accepted, then sample ANY record on that page at random.

Lemma 3 *Let $C_{EXTIDP}(H,1)$ = expected cost of sampling one record from an extensible hash file with double A/R page sampling.*

$$C_{EXTIDP}(H,1) = (E[\alpha_j])^{-1} \tag{23}$$

Proof: This result follows from Lemma 1. Note that to determine d_j we must retrieve the data page, since we assume no modification to directory. □ Note that we take the expectation with respect to j, the directory entry index,i.e., this expectation is the sum of the quantities α_j each weighted by the fraction of the directory entries pointing at page p_j.

6.2 A/R cell sampling

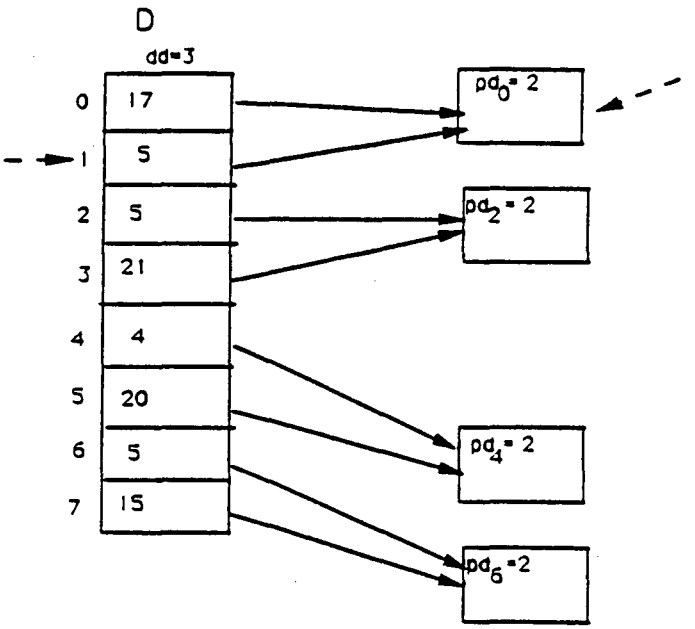
Here we view the directory D as an open addressing hash table with 2^{dd} buckets. Each bucket corresponds to an directory entry. Let us denote by c_j the number of records which hash into directory entry $D[j]$. As before, we randomly pick a directory entry $D[j]$ and accept the page it is pointing at with probability β_j:

$$\beta_j = (c_j/c^*) \tag{24}$$

If the page is accepted, then sample at random one record on the data page which hashed into $D[j]$. This is easily determined from the binary representation of the keys on the page.

Lemma 4 *Let $C_{EXTICS}(H,1)$ = expected cost of sampling one record from an extensible hash file with A/R cell sampling.*

$$C_{EXTICS}(H,1) = (E[\beta_j])^{-1} \tag{25}$$



Extendible hashing with maximum page capacity 26.
Dashed arrows show the cell and page selected in the example

Figure 3: Drawing of Extendible Hash File

Proof: This result follows from Lemma 1. Note again that to determine c_j we must retrieve the data page, since we assume no modification to directory. □

As an example, consider Figure 3. The numbers in the directory entries indicate the number of records hashed into them. The d_is can be computed as the sum of the numbers in the entries pointing at them. In this case $c^* = 21$ and $(d_i/g_i)^*$ is 13. If entry 1 is randomly selected, its page will be accepted with probability 11/13 according to double A/R page sampling but only with probability 5/21 according to the cell A/R method.

Theorem 5 *The cost of double A/R page sampling is always less than or equal to the cost of A/R sampling.*

$$C_{EXTIDP}(H,1) \leq C_{EXTICS}(H,1) \tag{26}$$

Proof: Observe that

$$d_j = \sum_{\forall i, D[i]=D[j]} c_i \tag{27}$$

$$d^* = \max_j d_j \tag{28}$$

Note that: $\forall j, (d_j/g_j) \leq c^*$, hence $(d_j/g_j)^* \leq c^*$. Thus:

$$E[\alpha_j] = \frac{E[(d_j/g_j)]}{(d_j/g_j)^*} \tag{29}$$

$$\geq \frac{E[(d_j/g_j)]}{c^*} \tag{30}$$

$$\tag{31}$$

$$E[\alpha_j] \geq \frac{E[E_{\forall i, D[i]=p_j}[c_j]]}{c^*} = \frac{E[c_j]}{c^*} \tag{32}$$

$$E[\beta_j] = E[c_j]/c^* \tag{33}$$

Hence:

$$E[\alpha_j] \geq E[\beta_j] \tag{34}$$

From the two lemmas we have:

$$C_{EXTIDP}(H,1) \leq C_{EXTICS}(H,1) \tag{35}$$

Q.E.D. □

It follows from the above analysis (and also from our simulations) that Double A/R sampling of Extendible Hash files will be most advantageous when the file is lightly loaded and many directory entries point to the same page. This will occur every time the directory size doubles. As the file becomes heavily loaded, so that each directory entry points to a distinct page both methods will yield identical performance.

7 Batch and Sequential Algorithms

7.1 Batch Algorithms

In this subsection we consider batch sampling from hash files. Our work is based on batch retrieval algorithms. The basic premise is to batch accesses to secondary storage so as to avoid rereading disk pages, as might occur with the iterative algorithms. Batch sampling can be applied to any of the hash files discussed above. However, for expository purposes we will present batch sampling for open addressing hash files.

We begin our discussion by observing that because of rejections in A/R sampling, we will need an inflated gross sample size, s', so that after acceptance/rejection we are left with a desired net sample size s. We recall from our discussion of acceptance/rejection sampling, and sampling from open addressing hash files, that the expected size of the gross sample required for a sample of size s is:

$$s' = \frac{b^*}{\bar{b}} s \tag{36}$$

For *one-pass batch sampling*, we will see that the net sample size will be a binomial random variable, $t \sim B(s', \alpha)$ where $\alpha = E[acceptance\,probability]$. Since the resulting net sample size may be less than the target sample size, additional passes may be needed to increase the sample size to the target level. For open addressing hash files, we have: $\alpha = \bar{b}/b^*$. Hence simple batch sampling is *binomial sampling*, returning a variable size sample, rather than a fixed size sample. For a simple random sample, the sample size can be readily adjusted by either randomly discarding records, or by augmenting the sample via additional iterative or batch sampling (called *multi-pass batch sampling*). Since we assume that the sample fits in memory, discarding excess records requires no additional I/O. However, it is often more efficient to simply further inflate the gross sample size to reduce the chance that the net sample size is inadequate.

Batch methods are typically useful when the gross sample size s' is a signficant fraction of the number of blocks of the file m. If $s' \ll m$, then there is little likelihood of rereading a page while sampling, so there is no point in employing a batch algorithm (it could actually be inferior).

Recall that A/R sampling of open addressing hash files has 3 phases: selection of a bucket at random, followed by an acceptance/rejection test, and finally retrieval of a sample record from the accepted bucket. The batch algorithm has 3 similar phases:

1. Instead of selecting the buckets one at a time we do them all at once. We note that if we randomly toss balls into urns repeatedly, the resulting occupancy distribution for the urns is multinomial. Thus we generate a multinomial random vector $x' \sim M(s', m)$ which determines how the gross sample is allocated among the buckets. The gross sample allocated to bucket i is denoted as x'_i.

2. Now, instead of performing acceptance/rejection tests one at at a time, we do all of the A/R tests for a single bucket at once. Since each A/R test prodcues a Bernoulli random variable, with parameter b_i/b^*, the sum of x'_i tests will be a binomial random variable, the number of records accepted from bucket i. Thus for each bucket i, we generate a binomial random variable $y_i \sim B(x'_i, b_i/b^*)$.

3. If $y_i > 0$ then we sample y_i records from the page, otherwise we do not read the page.

The expected cost of this sampling method is simply the expected number of elements of the multinomial vector x' which are nonzero.

$$E(C_{OAB}(s', m)) = m(1 - (1 - \frac{1}{m})^{s'}) \quad (37)$$

Proof: This is a classical result on occupancy statistics, see [JK77, pg. 144]. This result is also well known in the database literature [Car75] as the expected number of blocks retrieved from a file to retrieve s' records. Note that we are sampling with replacement here. □

We conclude our discussion of batch sampling by noting that its regime of utility for hash file sampling is smaller that it was for B^+ tree sampling [OR89], because in B^+ trees we often need to reread pages near the root, even if data pages are being read only once for iterative sampling.

7.2 Sequential Scan Sampling

Basically, batch sampling saves us from rereading pages while extracting the sample. In order for it to be useful, there must be a significant probability of rereading pages, i.e., allocating more than one element of the gross sample to the same page (bucket). However, if this probability is substantial, then we expect to read nearly all the pages of the file.

Hence, an alternative to the batch sampling described above is to sequentially scan the file and use a sequential sampling methods such as Vitter's [Vit85] (see below) to extract a random sample. In this application, sequential scan sampling requires that we read every page (bucket) of the file, in order to determine the number of records on it (and perhaps sample from them).

Simple batch sampling will typically outperform sequential scan sampling. However, if our purpose is to obtain a sample of records from a hash file which satisify some predicate of unknown selectivity, then we may be unable to reliably estimate the gross sample size required. At this point sequential scan (reservoir [Vit85]) sampling becomes the method of choice. Reservoir sequential sampling methods are used for sampling from files of unknown size (here because of the unknown predicate selectivity). They construct a reservoir of candidate elements of the sample (initially the first elements of the file), which they randomly replace as they sequentially read the file. At all times the reservoir contains a simple random sample without replacement. If necessary, this can easily be converted to a simple random sample with replacement.

8 Experimental Results

In this section we present experimental results (from simulations) concerning the performance of the various hash file sampling methods discussed above. More extensive results can be found in

[Xu89].

We present here simulation results concerning the sampling methods for Linear Hashing, both iterative and batch sampling methods. We also report results for iterative sampling from Extendible Hash files. Throughout this section we report sampling cost per element of the sample.

These results were obtained by constructing memory resident versions of the hash files, loading them with keys having a uniform random distribution, and then randomly sampling from the data structures. Additional records were then loaded into the data structures and the sampling repeated with higher load factors. Thus results for various load factors were not independent experiments. As expected the load factor for the hash files is a key performance parameter.

8.1 Linear Hashing

In Figure 4 we show the performance of iterative sampling methods from Linear Hash files. In this experiment the we split pages whenever the bucket chain length exceeds 3 pages (counting the primary page of the bucket as 1). The page capacity is 50 records and we have 97 pages initially in the file. The reader can clearly see that the 2-file method provides consistent performance, and for some file loadings substantially outperforms the 1-file sampling method. Note the cyclical nature of the 1-file method performance, which reflects the cyclic variation in the fraction of the disk pages which have been split.

In Figure 5 we show a similar experiment in which the page splitting criterion is to split pages whenever the load factor of the primary storage area exceeds 3. This criterion produces higher sampling costs and more variance, because it does not constrain the maximum chain length as tightly as the first criterion. The 2-file method continues to outperform the 1-file method.

8.2 Batch Sampling from Linear Hash Files

In Figures 6 and 7 we compare iterative and batch sampling from Linear Hash files via the 1-file method for the same splitting criteria shown in Figures 4 and 5. The sample size is either 3,000 or 5,000. The reader can see that batch sampling outperforms iterative sampling consistently, and that the unit sampling cost decreases with larger sample sizes.

8.3 Iterative Sampling from Extendible Hash Files

In Figure 8 we compare the two methods of iteratively sampling from Extendible Hash Files: double acceptance/rejection sampling of data pages vs. cell A/R sampling, The reader can see that

double A/R sampling outperforms cell A/R sampling for lightly loaded files. For heavily loaded files, the two methods present identical performance. The loading at issue here is the loading of cells relative to capacity of data pages.

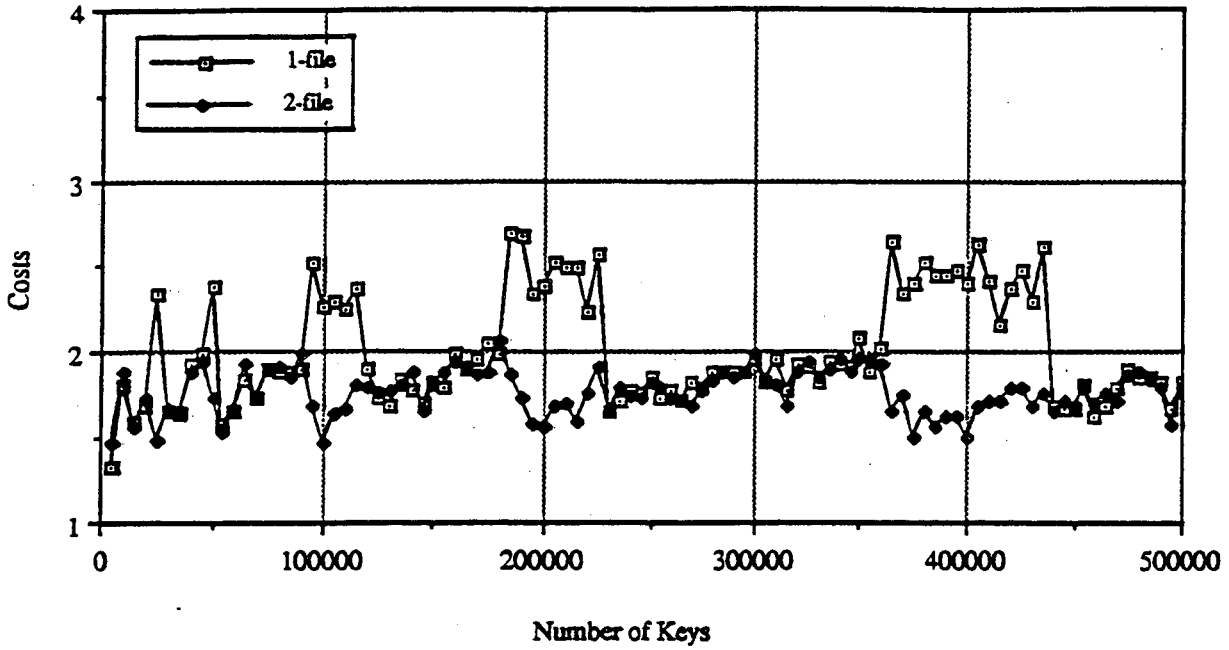


Figure 4. The comparison of the costs of the A/R sampling from 1-file and from 2-file methods (Criterion is "chain length ≥ 3, m = 97, page capacity = 50)

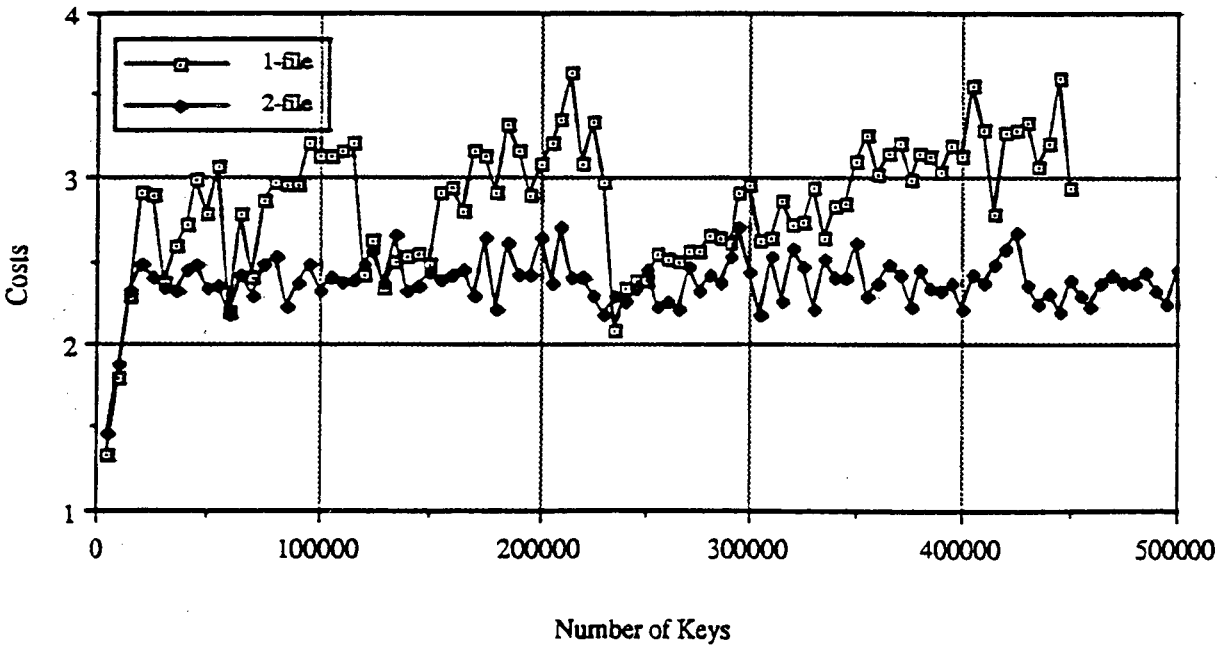


Figure 5. The comparison of the costs of the A/R sampling from 1-file and from 2-file methods (Criterion is "total number of records / capacity of primary area ≥ 3", m = 97, page capacity = 50)

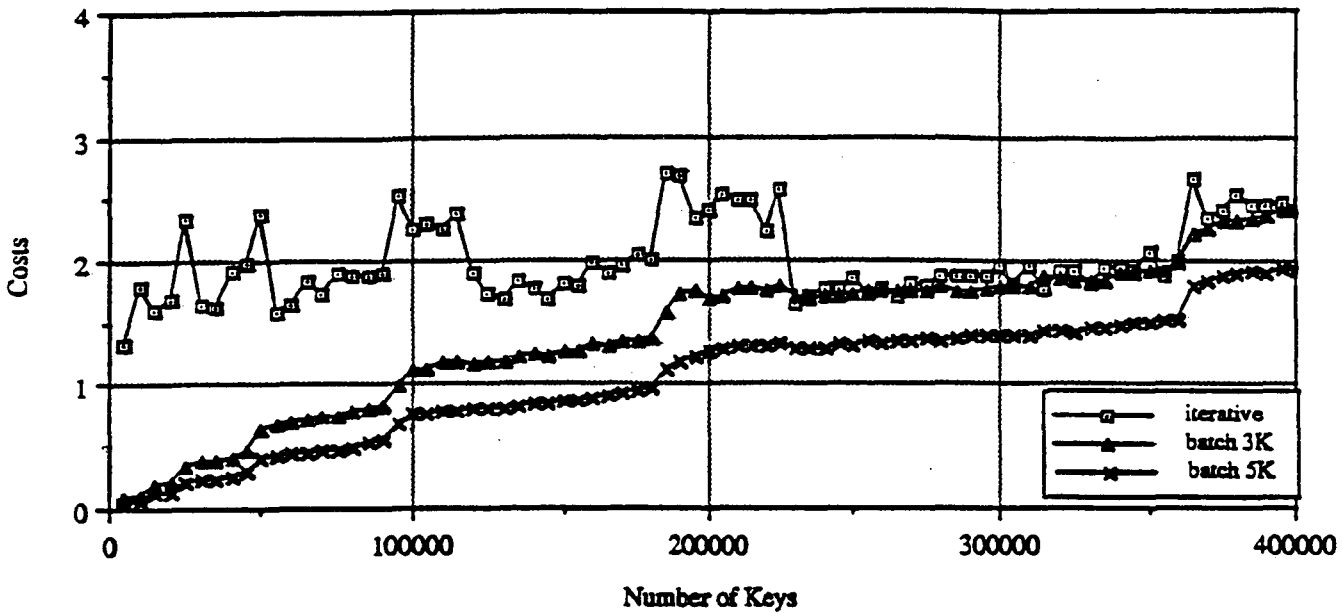


Figure 6. The comparison of the costs of the iterative and batch A/R sampling methods (Criterion is "chain length ≥ 3, m = 97, page capacity = 50)

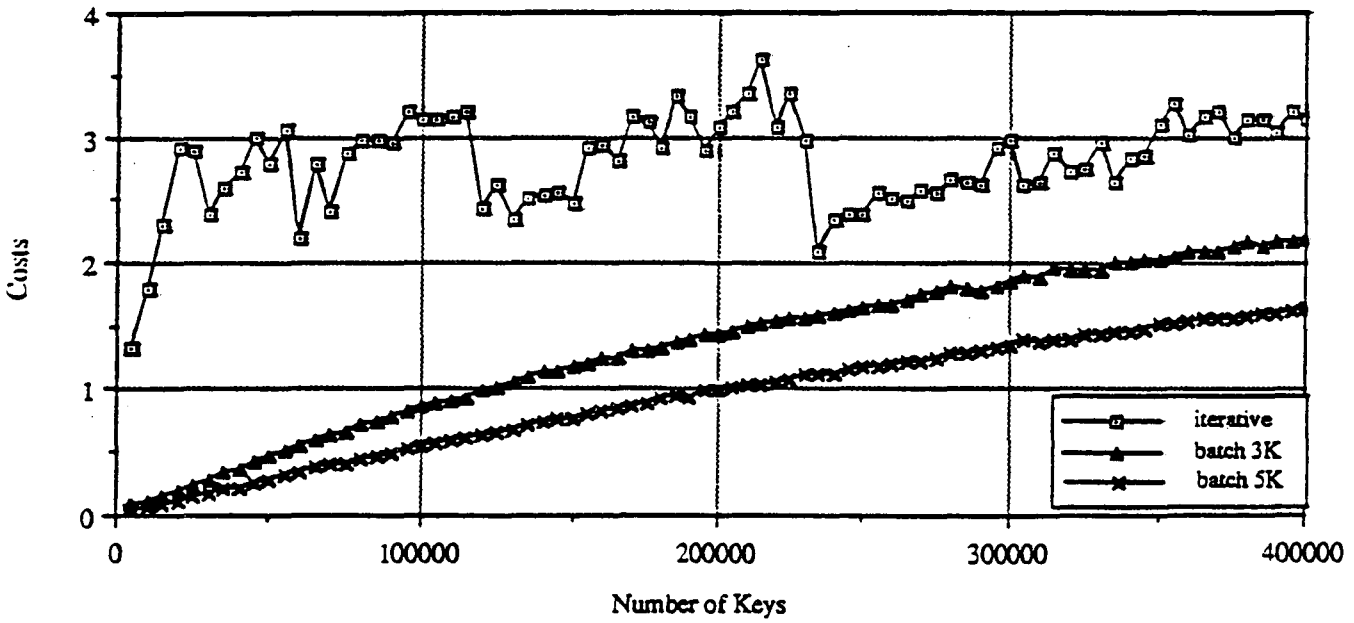


Figure 7. The comparison of the costs of the iterative and the batch A/R sampling methods (Criterion is "total number of records / capacity of primary area ≥ 3", m = 97, page capacity = 50)

9 Conclusions

We have shown how to to retrieve simple random samples from various types of hash tables without substantially altering the underlying hash table access methods or their normal performance. These methods are based on acceptance/rejection sampling, and provide a simple, inexpensive way to add sampling to relational database management systems. These methods are especially suited to systems which are only infrequently sampled, e.g., for auditing. For systems subject to heavy sampling query loads, adding auxiliary information to existing data structures or additional indices could improve sampling performance.

We have shown that sampling methods which exploit the structure of dynamic hash files have better performance than naive sampling algorithms. Thus the 2-file sampling method dominates 1-file sampling method for Linear Hashing, and double A/R sampling of data pages dominates cell A/R sampling for Extendible Hashing. These more sophisticated sampling methods are especially useful for lightly loaded hash files.

Batch sampling methods (which avoid rereading pages) will dominate iterative methods when the sampling fraction (of buckets) is large (≈ 1), i.e., gross sample size approximates (or exceeds) the number of buckets.

In such circumstances, sequential scan sampling (reservoir methods) will be preferred to batch sampling if we must also evaluate a predicate of unknown selectivity, because the predicate selectivity is necessary to determine the gross sample size for batch sampling.

Acknowledgements

The authors would like to thank Tekin and Meral Ozsoyoglu for their continuing encouragement. Jack Morgenstein first introduced us to the problem.

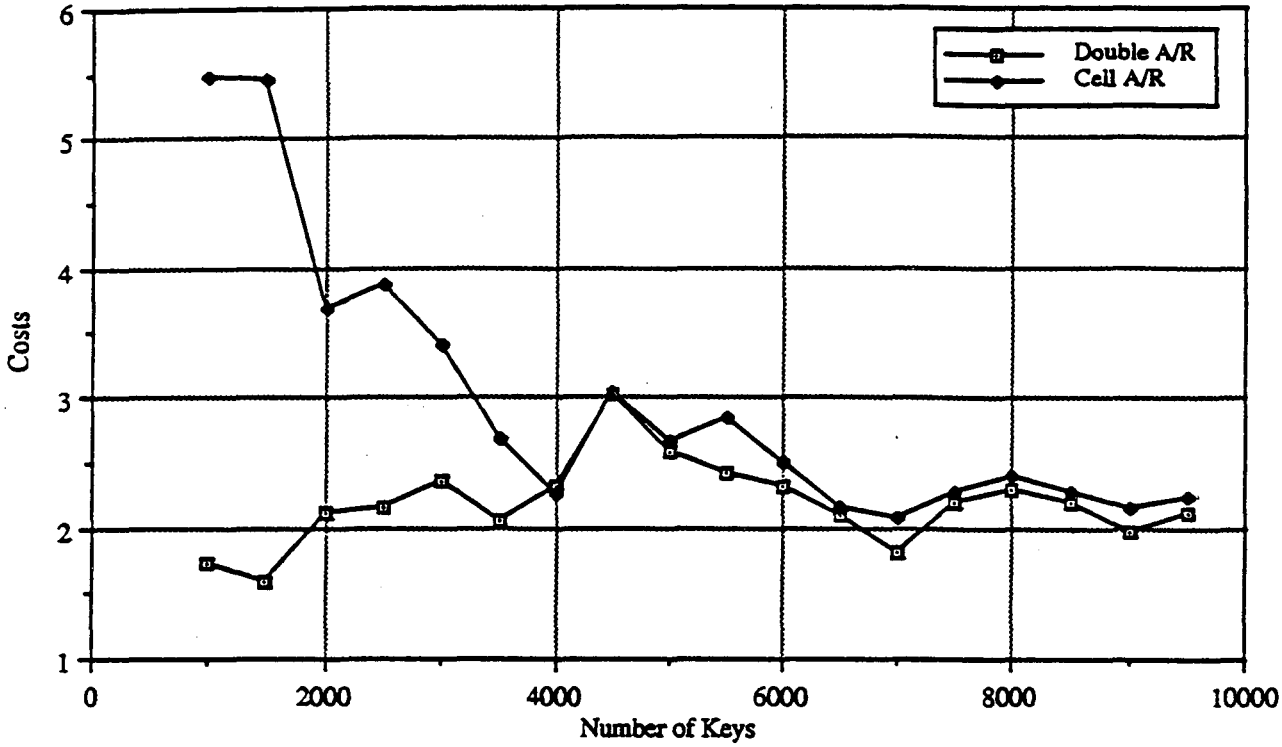


Figure 8. Extendible hashing (node capacity = 20, directory size = 1024)

References

[Ark84] Herbert Arkin. *Handbook of Sampling for Auditing and Accounting.* McGraw-Hill, 1984.

[Car75] A.F. Cardenas. Analysis and performance of inverted database structures. *Communications of the ACM*, 18(5):253–263, May 1975.

[Coc77] William G. Cochran. *Sampling Techniques.* Wiley, 1977.

[Den80] Dorothy E. Denning. Secure statistical databases with random sample queries. *ACM Transactions on Database Systems*, 5(3):291–35, Sept. 1980.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, Sept. 1979.

[HOT88] Wen-Chi Hou, Gultekin Ozsoyoglu, and Baldeo K. Taneja. Statistical estimators for relational algebra expressions. In *Proceedings of the Seventh ACM Conference on Principles of Database Systems*, pages 288–293, March 1988.

[HOT89] Wen-Chi Hou, Gultekin Ozsoyoglu, and Baldeo K. Taneja. Processing aggregate relational queries with hard time constraints. In *ACM SIGMOD International Conference on the Management of Data*, pages 68–77, June 1989.

[JK77] Norman L. Johnson and Samuel Kotz. *Urn Models and Their Application.* John Wiley and Sons, 1977.

[Knu73] Donald Ervin Knuth. *The Art of Computer Programming: Vol. 3, Sorting and Searching.* Addison-Wesley, 1973.

[Lar82] Per-Ake Larson. Performance analysis of linear hashing with partial expansions. *ACM Transactions on Database Systems*, 7(4):566–587, Dec. 1982.

[Lit80] W. Litwin. Linear hashing: a new tool for file and table addressing. In *Proceedings of the Sixth International Conference on Very Large Databases (VLDB)*, pages 212–223, 1980.

[LN89] Richard J. Lipton and Jeffrey F. Naughton. Estimating the size of generalized transitive closures. In *Proceedings of the 15th International Conference on Very Large Databases (VLDB)*, pages 165–171, August 1989.

[LTA79] Donald A. Leslie, Albert D. Teitlebaum, and Rodney J. Anderson. *Dollar Unit Sampling.* Copp Clark Pitmanan, 1979.

[Lu87] Hongjun Lu. New strategies forcomputing the transitive closure of a database relation. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB)*, pages 267–274, September 1987.

[LWW84] H.-J. Lenz, G.B. Wetherill, and P.-Th. Wilrich, editors. *Frontiers in Statistical Quality Control 2.* Physica-Verlag, Wurzburg, Germany, 1984.

[Mon85] Douglas C. Montogmery. *Introduction to Statistical Quality Control.* Wiley, 1985.

[Mor80] Jacob Morgenstein. *Computer Based Management Information Systems Embodying Answer Accuracy as a User Parameter.* PhD thesis, Univ. of California, Berkeley, December 1980.

[OR86] Frank Olken and Doron Rotem. Simple random sampling from relational databases. In *Proceedings of the Twelfth International Conference on Very Large Databases (VLDB)*, pages 160–169, August 1986.

[OR89] Frank Olken and Doron Rotem. Random sampling from b^+ trees. In *Proceedings of the 15th International Conference on Very Large Databases (VLDB)*, August 1989.

[Vit85] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, March 1985.

[Wil84] Dan Willard. Sampling algorithms for differential batch retrieval problems (extended abstract). In *Proceedings ICALP-84.* Springer-Verlag, 1984.

[Xu89] Ping Xu. Sampling from b^+ trees and hash files. M.s. thesis, San Francisco State Univ., 1989.

LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
INFORMATION RESOURCES DEPARTMENT
BERKELEY, CALIFORNIA 94720