# Random Sampling from Databases

by

Frank Olken

B.S. (University of California at Berkeley) 1973
M.S. (University of California at Berkeley) 1981

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in Charge:

Professor Michael Stonebraker, Chair
Professor Alan J. Smith
Professor Leo Breiman
Professor Arie Segev

1993

The dissertation of Frank Olken is approved:

---

Chair Date

---

Date

---

Date

---

Date

University of California at Berkeley

1993

**Random Sampling from Databases**

**Abstract**

**Random Sampling from Databases**

by

Frank Olken

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Michael Stonebraker, Chair

In this thesis I describe efficient methods of answering random sampling queries of relational databases, i.e., retrieving random samples of the results of relational queries.

I begin with a discussion of the motivation for including sampling operators in the database management system (DBMS). Uses include auditing, estimation (e.g., approximate answers to aggregate queries), and query optimization.

The second chapter contains a review of the basic file sampling methods used in the thesis: acceptance/rejection sampling, reservoir sampling, and partial sum (ranked) tree sampling. I describe their usage for sampling from variably blocked files, and sampling from results as they are generated. Related literature on sampling from databases is reviewed.

In Chapter Three I show how acceptance/rejection sampling of $B^+$ trees can be employed to obtain simple random samples of $B^+$ tree files without auxiliary data structures. Iterative and batch algorithms are described and evaluated.

The fourth chapter covers sampling from hash files: open addressing hash files, separately chained overflow hash files, linear hash files, and extendible hash files. I describe both iterative and batch algorithms, and characterize their performance.

I describe and analyze algorithms for sampling from relational operators in Chapter Five: selection, intersection, union, projection, set difference, and join. Methods of sampling from complex relational expressions, including select-project-join queries, are also described.

In Chapter Six I describe the maintenance of materialized sample views. Here I combine sampling techniques with methods of maintaining conventional materialized views. I consider views defined by simple queries consisting of single relational operators.

The penultimate chapter covers sampling from spatial databases. I develop algorithms for obtaining uniformly distributed samples of points which satisfy a spatial predicate represented as a union of polygons in the database. Sampling algorithms from both quadtrees and R-trees are described, including spatial reservoir sampling algorithms.

I conclude with a summary of the thesis and an agenda for future work.

# Contents

# List of Tables

# List of Figures

# Preface

This thesis has its origins in the 1980 dissertation of Jack Morgenstein [Mor80] and in discussions at the Second Workshop on Statistical Data Management hosted by Lawrence Berkeley Laboratory and held in Palo Alto, California in 1983. (This workshop series eventually became the Statistical and Scientific Data Management Conferences, hereafter referred to as SSDBM). Morgenstein's thesis was an early effort to address the issues of sampling from databases. He was primarily interested in providing approximate answers to aggregate (SUM, COUNT) queries quickly. From conversations with Jack and reading his thesis, I was aware of his work and realized that much more remained to be done. At the Second SSDBM there was much debate about which statistical and graphics functions should be included in a statistical data management system. However, there was considerable agreement that sampling operators should be included.

I began to work on sampling from databases in early 1984 with Doron Rotem, addressing issues of sampling from relational operators first. Some early results were presented as part of a talk I gave at the Interface Conference (Interface between Statistics and Computer Science) in April of 1984. I gave a more extensive presentation at the SIAM Conference Frontiers in Computational Statistics in October of 1984. The first paper [OR86] appeared in the Kyoto VLDB Conference in August of 1986. I went on to look at sampling from $B^+$ trees[OR89], hash files [ORX90], the maintenance of materialized sample views [OR92a], and sampling from spatial databases [OR92b, OR93].

The emphasis of this work has been on the algorithms to obtain random samples (usually simple random samples) of database queries, rather than on estimation techniques. The emphasis on algorithms reflects my perception that the algorithmic issues had not been adequately addressed, whereas there was already an extensive statistical literature on related estimation problems.

Also, Neil Rowe persuaded me that database statistical abstracts (collections of summary statistics on database partitions) and a specialized inference system could generally provide approximate answers to many routine aggregate queries much faster than sampling. For many purposes (especially physical inspection of the "real world" objects corresponding to the sampled database records) samples of database records are essential, and estimation from database statistical abstracts is not an alternative.

# Acknowledgements


## People

First, I must thank Doron Rotem, my colleague in the data management group at LBL who has effectively served as my de facto thesis advisor for this research. He has provided continuing encouragement, criticism, a source of suggestions for topics, and extensive editorial comments. Without Doron's continuing interest and encouragement I would probably never have completed the thesis.

I also wish to thank the members of my thesis committee: the chairman Prof. Mike Stonebraker (UCB EECS), Prof. Leo Breiman (UCB Statistics), Prof. Arie Segev (UCB Bus. Admin.), and Prof. Alan Jay Smith (UCB EECS), for reading the dissertation and making many useful comments. The organization of the thesis was greatly improved due to the suggestions of Prof. Stonebraker. Prof. Stonebraker's comments also prompted me to extend the results on sampling from relational operators to more complex relational expressions.

Prof. Alan Jay Smith (UCB EECS) supervised my earlier efforts to write a dissertation on performance evaluation of storage hierarchy management policies. It was at his recommendation that I took classes in statistics, stochastic processes and queueing theory at UCB which spurred my interest in statistics and provided me with the statistical background to undertake this work. Alan also taught me to see storage hierarchy management and design decisions as micro-economic decisions, a lesson I continue to profit from.

Ping Xu performed simulation studies of the algorithms for sampling from $B^+$ trees and hash files [Xu89] and permitted me to use some of her figures in my thesis. Ed Reiss helped with xeroxing references and reformatting some bibliographic citations. Carol Backhus (LBL Library) provided me with indispensable assistance in bibliographic searching.

This work was inspired initially by Jack Morgenstein's thesis [Mor80].

Arie Shoshani (Data Management Group Leader at LBL) and Carl Quong (then Computer Science Research Department Head at LBL) provided support in the initial years of the thesis. Subsequently, after I joined the Genome Computing Group, Sylvia Spengler (Deputy Director of the LBL Human Genome Center) provided continuing encouragement to finish the thesis. She also read and commented on an early draft of the thesis. William E. Johnston (formerly Group Leader for the LBL Human Genome Computing Group) permitted me to spend some time on thesis. Jasper Rine (Director, LBL Human Genome Center) provided support in the penultimate months of the dissertation. The LBL Particle Data Group permitted me the use of their printer for printing the thesis.

Prof. Eugene Lawler (UCB EECS, Vice Chair for Graduate Affairs) encouraged me to

finish with news that bureaucratic obstacles to completion would likely prove surmountable. Kathryn Crabtree (UCB EECS) helped to shephard my readmission through the bureaucracy.

Tekin and Meral Ozsoyoglu, Betty Salzberg, Don Batory, Jeff Naughton, Ronnie Hibshoosh. Shalom Tsur, Laura and Peter Haas, Witold Litwin, Victor Markowitz, Ralph Merkle, Deborah Hopkins, Dalit Naor, Prof. Marty Graham, and Cassandra Smith all provided me with encouragement to finish.

Kevin Seppi (IBM), Srini Seshadri (IIT), and Gennady Antoshenkov (DEC) supplied me with copies of their papers (and theses) on sampling. Charles Parrish (LLNL), R.W. Mensing (LLNL), and Richard Carlson (LLNL) helped by explaining nuclear materials inventory sampling to me, and supplied me with their papers about these systems.

Prof. Eugene Wong (UCB EECS) first suggested the use of materialized sample views to me. Mark Durst (LBL) helped me understand epidemiological applications and resampling techniques. Prof. David Aldous (UCB Statistics) and Sridar Seshadri (UCB Haas Business School) provided advice on some statistical matters. Carl Blumstein (UC UERG) explained the use of surveys of energy consumption and directed me to the DOE/EIA Commercial Building Energy Consumption Survey. Ms. Stephanie Pedersen (LBL) explained LBL internal audit practicies to me.

Susan Eggers and Harry K.T. Wong (both then at the LBL Data Management Group) helped to recruit me to the LBL Data Management Group.

Marty Gelbaum (LBL Computer Center) installed and supported Tex, LaTex, dvips, and the LaTex macros which I used to typeset the text, patiently responding the my frequent questions and complaints.

## Funding Sources

This work was primarily supported by the Director, Office of Energy Research, Office of Basic Energy Sciences, Applied Mathematical Sciences Division and by the Director, Office of Health and Environmental Research (via the Human Genome Program Office and the LBL Human Genome Center) both of the U.S. Department of Energy under Contract DE-AC03-76SF00098.

Partial support of this work was also provided by the National Science Foundation under grants MIP-8713274, MIP-9116578 and CCR-9117028, by NASA under Grant NCC 2-550, and by the State of California under the MICRO program.

# Chapter 1

# Introduction

In this chapter I introduce the subject of this dissertation, random sampling of the results of queries from relational databases. I provide some examples of random sampling queries and discuss how such queries spawn the problems addressed in this thesis. Brief definitions of some database terms and statistical terms related to sampling are included.

The remainder of the chapter is devoted to discussion of the motivation for including random sampling query facilities in database management systems. A number of major applications of random sampling from databases are described.

In Chapter 2 I will discuss basic sampling techniques, and related literature on sampling from databases.

## 1.1 Random Sampling Queries

A *random sampling query* returns a random sample of the results of a relational retrieval query. More generally, the sampling operator might appear anywhere in a nested SQL query where a SELECT could appear.

For example, given two relations EMPLOYEE(EMP_ID, NAME, ADDR, SALARY) and DOCTORS(EMP_ID, SPECIALTY), a query such as:

```
SAMPLE 100 OF
SELECT EMP_ID, ADDR
FROM EMPLOYEE WHERE SALARY > $50K
```

will produce a random sample of size 100 from all employees who make over $50K.

Another query:

```
SELECT AVG(SALARY),MAX(SALARY)
FROM EMPLOYEE
WHERE EMP_ID IN (SAMPLE 100 OF
        SELECT EMP_ID
        FROM DOCTORS
        WHERE SPECIALTY = "SURGEON")
```

will produce the average and the maximum salaries computed from a random sample of 100 surgeons.

Note that for these queries the user has explicitly specified the desired sample size. Throughout the thesis I will generally assume that the user has specified the desired sample size *exactly*, after reckoning that additional sample elements are too expensive to process (inspect) and that smaller samples are large enough to permit sufficiently accurate estimation or hypothesis testing. Since postprocessing (usually inspection) costs concerning the sample usually dwarf the cost of sampling from the database the assumption of external sample size specification is plausible.

There are other applications of sampling from databases, e.g., aggregate statistic (AVG, COUNT) estimation or internal use by the query optimizer, in which the sample size would not be specified externally. In such contexts the sample size may be determined during the course of the sampling, e.g., sequential (a.k.a. adaptive) sampling processes. Such cases are treated in Chapter 2.

Having seen examples of sampling queries, I now begin to examine the central question of this thesis: how does one efficiently answer such sampling queries?

## 1.2 Issues of Sample Query Processing

This thesis is concerned with the question of how to efficiently process the sorts of random sampling queries described above.

Sampling can be thought of as a funny kind of selection operator. Thus, as with selection, one would like to push the sampling operator down the query processing plan tree (a.k.a. query tree) as far as possible toward the leaves (reading the base relations).

Such a strategy spawns a set of problems which are addressed in this thesis:

- How does one push the sampling operator down the query tree? What modifications of the relational operators are needed? What other information must be passed up the query tree? What are the constraints on pushing down the sampling operators?
- How does one implement sampling of base relations stored according to various access methods: variably blocked records, $B^+$ trees, hash files, etc.
- How does one implement on-the-fly sampling of intermediate results, of unknown size?
- How does one maintain materialized views of sampling queries? i.e., if one computes and stores a view defined by a sampling query, then how does one update it as the underlying relations are updated?
- For spatial databases, how does one obtain a uniform sample from the region specified by the union of polygons stored in a spatial data structure?

Before proceeding further, I pause to define some of the terms I will need throughout the dissertation.

## 1.3 Database Terminology

I assume that the reader has some familiarity with the idea of a relational database (see [Ull88, Dat90]). The following definitions are included for those readers who are less familiar with database terminology.

- A *selection* operator selects a subset of tuples (records) from a relation (table) which satisfy some logical predicate, e.g., all persons whose salary exceeds \$100K.
- *Predicate selectivity* refers to the number (or proportion) of records which satisfy a particular selection predicate.
- The *projection* operator, $\pi_{A,B}(R)$, selects a subset of attributes (columns) $A, B$ of a relation $R$ removing duplicate rows remaining after the columnar restriction.
- The *natural join*, (a.k.a *equi-join*) operator, written $R \bowtie S$, is used to merge relations. It computes the cross-product of two relations, and then selects the elements of the cross-product which satisfy an equality-predicate on specified attributes. For example, one might perform an equi-join on the student ID number between a relation containing student enrollment records (i.e., student ID, course no., section no.) and a student name relation (i.e., student ID, student name). Unless otherwise specified *join* will always refer to the *natural join*.
- The semijoin of relation $R$ by relation $S$, written $R \ltimes S$, is the natural join of $R$ and $S$ projected onto the attributes of $R$. In plain English these are the records in $R$ which match (on the join attribute) some record in $S$, e.g., match on name or social security number.
- An attribute of a relation is a *key*, if its value uniquely identifies each tuple in the relation, e.g., bank account number.
- A *view* is a derived relation specified by a query against the *base relations* or previously defined *views* of the database. Views can be used in subsequent queries wherever a relation may appear.
- A *materialized view* is a view which has been instantiated (computed according to its definition). Such materialized views must be maintained (updated or recomputed) when the underlying base relations are modified.
- Finally, a *select-project-join (SPJ) query* is one which consists of selects, projects, and joins.

## 1.4 Sampling Terminology

For those readers unfamiliar with sampling terminology I include a few definitions here.

- The *population* to be sampled is assumed to be a set of records (tuples).
- A *fixed size random sample* is one in which the sample size is a specified constant.

- A *binomial random sample* is one in which the sample size is a *binomial random variable*. Such samples are generated by sequentially scanning a set of records and including each record with a fixed probability, $\alpha$.

- A *simple random sample without replacement (SRSWOR)* is a subset of the elements of a population, for which each element of the population is equally likely to be included in the sample. No duplicates are allowed. SRSWOR are naturally generated by sampling algorithms which sequentially scan a file. They are usually more informative than a simple random sample with replacement (duplicates allowed) of the same size.

- A *simple random sample with replacement (SRSWR)* can be generated by repeatedly taking a simple random sample of size 1 and concatenating the samples together. Each element of the population is equally likely to be to be the first, second, .... element of the sample. Duplicates are allowed. SRSWR are naturally generated by iterative, and batch sampling algorithms.

  SRSWR samples are used to implement sampling from joins, since tuples (records) from the first of the joined relations may be joined with more than one tuple from the second of the joined relations.

- A *stratified random sample* is obtained by partitioning the population, e.g., by sex, then taking a SRS (usually WOR) of specified size from each strata. Typically, the sample sizes are allocated to the strata in proportion to the population size of the strata.

- A *weighted random sample* is one in which the inclusion probabilities for each element of the population are not uniform.

- A *probability proportional to size (PPS or $\pi PS$) sample* is a weighted random sample without replacement in which the probability of inclusion of each element of the population is proportional to the *size* (e.g., mass, or dollar value) of the element.

- A *dollar unit sample (DUS) (a.k.a. monetary unit sample (MUS))* is a weighted random sample generated by iteratively taking a sample of size 1 with inclusion probabilities proportional to the *sizes* (typically monetary value) of the elements. It is similar to a PPS sample except that duplicates are allowed. Such samples are simpler to generate than true PPS samples, but less informative.

- *Clustered samples* are generated by first sampling a *cluster unit* (e.g., a household or disk page) and then sampling several elements within the cluster unit (e.g., all the records on a disk page). Such samples are typically cheaper to obtain than simple random samples, but more complex to analyze.

- *A systematic sample* is one obtained by taking every $k'th$ element of a file. (The starting point is chosen at random.)

Sampling procedures may be:

- *Single stage* - just choose a sample of specific size.

- *Two stage* - first choose a small sample, use the information from the initial sample to decide how large the second sample needs to be, e.g., to obtain a desired level of accuracy of estimation.
- *Sequential sampling* - Sample from the file iteratively, after each sample element is obtained decide whether it is necessary to continue sample. Also known as *adaptive sampling*.
- *Group sequential sampling* - similar to *sequential sampling* except that one decides whether to continue after groups of sample elements are obtained.

Sampling algorithms may be classified as:

- *Iterative* - Those algorithms which loop, generates one sample element at a time. Typically this will generate a SRSWR.
- *Batch* - Those algorithms which generate a group of sample elements at a time. Such algorithms (analogous to batch searching) avoid redundant rereading of disk pages which may arise with iterative algorithms.
- *Sequential (scan)* - This is a class of sampling algorithms which sequentially scan (possibly skipping) a file to generate a sample. Typically this will generate a SRSWOR. Some of these algorithms require that the file size is known, some do not.
- *Reservoir* - A subclass of sequential scan sampling algorithms which are used to sample from files of unknown size. Such algorithms might be used to sample on-the-fly from the results of a query as the results are generated (at which time one does not yet know the size of the result file).

## 1.5 Motivation

### 1.5.1 Why sample?

Random sampling is used on those occasions when processing the entire dataset is unnecessary and too expensive in terms of response time or resource usage. The savings generated by sampling may arise either from reductions in the cost of retrieving the data from the database management system (DBMS) or from subsequent "post-processing" of the sample.

Retrieval costs are significant when dealing with large administrative or scientific databases. Post-processing of the sample may involve expensive statistical computations or further physical examination of the real world entities described by the sample. Examples of the latter include physical inspection and/or testing of components for quality control [Mon85, LWW84], physical audits of financial records [Ark84, LTA79], and medical examinations of sampled patients for epidemiological studies. Most major financial databases are subject to annual audits, which typically entail random sampling of records from the database for corroboration.

Sampling is useful for applications which are attempting to estimate some aggregate property of a set of records, such as the total number of records which satisfy some predicate.

Thus random sampling is typically used to support statistical analysis of a dataset, either to estimate parameters of interest [HOT88, HOT89, HO91] or for hypothesis testing. See [Coc77] for a classic treatment of the statistical methodology. Applications include scientific investigations such as high energy particle physics experiments. Other applications include quality control and public policy analyses. For example, one might sample a join of welfare recipient records with tax returns or social security records in order to estimate welfare fraud rates. Or one might use a sample of IRS records to estimate the net revenue generated by proposed tax legislation.

### 1.5.2 Why put sampling into a DBMS?

Thus it is clear that sampling from databases is useful in a number of settings. This is not sufficient to conclude that sampling should be included in the DBMS. Conceivably, one could add this functionality by means of an add-on package, outside the DBMS. This is the current practice for audit sampling software.

However, I believe that inclusion of sampling operators within the DBMS is worthwhile, for the following reasons:

- to efficiently process queries which seek random samples,
- to efficiently (and quickly) provide approximate answers to aggregate queries,
- to estimate parameters for query optimizers,
- to provide privacy and security for individual data in statistical databases.

I believe that one should put sampling operators into the DBMS primarily for reasons of efficiency. By embedding the sampling within the query processing, one can reduce the amount of data which must be retrieved in order to answer sampling queries and one can effectively exploit indices created by the DBMS. Instead of first processing the relational query and then sampling from the result, one can, in effect, interchange the sampling and query operators. Thus, one pushes the sampling operator down in the query processing plan tree, as with selection. For some relational operators (e.g., projection, join) one must take additional measures to assure that the resulting sample is still a simple random sample. The necessary techniques are discussed in Chapter 6.

Sampling can also be used in the DBMS to efficiently estimate the answers to 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). Sampling procedures were only briefly discussed. More recently, Hou, et al. [HOT88, HO91] have discussed the construction of statistical estimators for arbitrary relational expressions for COUNT aggregates. They also envision the application of their methods to real-time applications [HOT89].

Sampling may also be used to estimate the database parameters used by the query optimizer in choosing query evaluation plans. This is discussed more fully later in Chapter 2.

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

### 1.5.3 Why don't DBMSs support sampling?

Given the variety of uses for sampling from databases, one might well wonder why facilities for random sampling queries have not yet been included in DB management systems. There are two reasons: lack of knowledge about sampling in the DB community and the recent emergence of the technological trends which motivate the inclusion of sampling in DBMSs. Below I consider the reasons in more detail:

- Lack of knowledge - Prior to 1986 there was little research in the database community on how to implement sampling of a relational database. Since that time several researchers have explicated many of the technical issues and advantages.

- Effort - The algorithms described here require some modifications to access methods. New code must be written to implement the sampling algorithms. In some cases, e.g., partial sum tree (a.k.a. ranked) tree sampling algorithms, the data structures and update code must be modified.

- Complexity - Optimizing queries involving sampling increases the complexity of the query optimizer. If the query optimizer is rule based, the modifications should be manageable. For hard coded query optimizers, the problem is more difficult.

- Lack of appreciation - Until recently, the database community did not appreciate the utility of sampling, both for use within the DBMS and for users. It has taken time and work by a number investigators to explicate the utility of sampling for query optimization, approximate answers to aggregate queries, load balancing in parallel database machine joins, auditing, data analysis, etc.

- Growth of database size - The sample size required for estimation and hypothesis testing is largely independent of the underlying database size. Hence, sampling becomes more useful as databases grow in size. Sampling was less important in earlier years, as databases were smaller.

- Main memory database systems - Sampling is especially attractive in a main memory database system - which have an effective block size of one. Such main memory resident database systems have only recently become practical as memory prices have declined.

- Growth of data analysis - As computers, disks, and DBMSs have become cheaper, faster, and more sophisticated (and users have become better accustomed to their use) one sees many more attempts to do interactive data analyses of large databases, and more efforts to keep large historical databases online for data analysis purposes. Sampling can facilitate this work.

- Growth of nonparametric data analysis - The past decade has seen considerable development of nonparametric data analyses. Typically these analyses are extremely computationally intensive and have thus have only recently become economically feasible. Many of these techniques either directly employ sampling, or would be used on sample datasets (to reduce the computational costs).

- Auditors' ignorance - Auditors realize that they have difficulties sampling databases for audit purposes, but they have not realized that they could demand that database management systems support random sampling queries. Once such sampling facilities begin to appear, internal auditors will likely demand their inclusion in the specifications for new DBMS procurements (much as crash recovery is now a standard specification).

- Auditors' conflict of interest - Many large accounting firms sell autonomous audit sampling software packages to other accounting firms and (especially) to internal audit organizations. If sampling facilities were included in DBMSs, external audit sampling packages could be much simpler (perhaps unnecessary), resulting in revenue losses to those accounting firms which market them.

- Lack of statistical training and interest - Most database researchers have had relatively little training or interest in statistics, which has slowed the adoption of sampling techniques in the database community.

I turn next to the rationale for including support for spatial sampling in DBMSs.

### 1.5.4 Why provide spatial sampling?

Spatial databases have become increasingly important over recent years. The largest applications are geographic information systems (GIS), and computer aided design (CAD).

Statistical analyses are frequently made of large databases for scientific and decision support purposes. Often approximate answers will suffice. Random sampling offers an efficient method of obtaining such approximate answers of aggregate queries [HOT88, HOT89, Coc77]. I (and others) have discussed this previously for conventional databases (see Chapter 2). Random sampling has three common justifications: reduced database retrieval costs (or time), reduced "postprocessing" computational costs, or reduced cost of physical inspections. I consider each rationale in turn.

In spatial databases (e.g., GIS applications) one expects sampling to be even more important than in conventional databases. There are two reasons: spatial databases (i.e., GIS) are typically very large, and they tend to be heavily used for analysis purposes. Indeed analysis is often a key motivation for the creation of many GIS's, e.g., for urban planning, crop forecasting, environmental monitoring. Such analyses often involve either aggregate queries, statistical analyses or graphical displays. In spatial contexts, common aggregate queries might involve the estimation of areas, volumes, total crop yields, forest inventories [dV86], or total mineral or petroleum reservoir sizes.

For example, consider a geographic information system (GIS) which stores information on land use for each parcel, one might wish to estimate the area devoted to agriculture within Alameda county. This involves both estimation of an aggregate query and evaluation of a spatial predicate (within Alameda county). One could do this by taking random samples of points within Alameda county and checking for the land use. Total land devoted to agriculture in Alameda county could then be estimated by the fraction of points falling on agricultural parcels multiplied by the area of Alameda county (assumed to be known).

Sampling is also used to reduce dataset size prior to graphical display of the data, both to reduce computational effort, and because of resolution limits of displays.

Finally, 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 (or points in the case of a spatial DB). Spatial applications include agronomy, forestry [dV86], pest management, geology, ecology, and environmental monitoring, all of which entail physical inspection of the sites designated by the database records.

The accuracy of parameter estimates from samples is typically primarily a function of size of the sample [Coc77], with little dependence on the population size. Hence sampling is most advantageous when sampling from large populations, as would be found in very large GIS databases.

I turn next to the rationale for including support for random sample views in DBMSs.

### 1.5.5 Why provide sample views?

A sample view is a derived relation whose contents are defined by a random sampling query. It can be used wherever a relation could occur in a retrieval query. There are several reasons for providing sample views:

- **Consistency:** Relational DBMSs typically permit views of any other retrieval query. Hence one would also like to be able to specify views of sampling queries.
- **Repetitive analyses:** Often analyses will be repeated periodically against the samples of current state of the DB. A sample view facilitates such studies.
- **Multi-query optimization:** Sample views provide a simple means of extracting common subexpressions from several sampling queries, thereby facilitating multi-query optimization.
- **Complex query formulation:** It has been widely observed that users often find it easier to formulate (and diagnose) complex queries as a sequence of simpler queries. The intermediate results may be specified as a view. Again such facilities are desirable also for sampling queries.
- **Security:** Sample queries have been shown to provide some security against compromise of sensitive data in statistical databases [DS80] via aggregate queries. Sample views provide a means of implementing Denning's proposal.

### 1.5.6 Why materialized sample views?

Materialized sample views are those sample views which have been instantiated, and hence must be maintained (updated or recomputed) when the underlying relations are updated.

Even if one has decided to provide sample views in his (her) DBMS, it is not self-evident that one should support the maintenance of *materialized* sample views. The rationale is the traditional one: materialized views may be more efficient than virtual views (query modification) for supporting queries against the views [Han87b]. In general, materialized views are to be preferred when queries against the views are common, and updates of the

underlying relations relatively infrequent [Han87b]. For sample views, the case for materialized views is even stronger, since sampling fractions from very large databases are typically quite small, a few percent or less. Thus most sample queries are effectively highly selective queries. Since most updates will therefore not alter the sample view, materialized sample views (MSVs) are superior to virtual sample views (VSVs) at larger update frequencies than conventional views.

Another application of MSVs consists of correlated sample snapshots. It is well known that correlated samples (with many common individuals) are useful for measuring changes over time. Such so-called panel surveys, or longitudinal studies are done for unemployment estimation, epidemiological studies, opinion studies, etc. Thus work on sample views can be readily adapted to sample snapshots, cf. the work of Adiba & Lindsay [AL80] and Lindsay et al. [LHMP86].

## 1.6 Applications

I now discuss a number of applications for which sampling from databases is necessary. Readers who are already convinced of the utility of database sampling can skip to the second chapter, which reviews basic techniques and related literature on sampling from files and databases.

### 1.6.1 Financial Audit Sampling

One very important and ubiquitous use of sampling from databases is financial audit sampling. Auditors have used random sampling to verify accounting records for almost 60 years. Kinney [Kin86] recounts some of the history of sampling, citing Carman's 1933 paper [Car33] (which he reprints) as the earliest paper on the subject. The theory and practice of audit sampling has been the subject of many papers and a number of monographs [AL81], [Ark84], [Bai81], [IPA75], [Kin86],[GC86],[Guy81], [HRA62],[Rob78] [SS86], [LTA79], [Van76],[Van50],[Van50],[VL88], [AIoCPA83],[GC86],[McR74],[NL75] from the early 1950's onward. Most of these monographs concern such statistical issues as sample design, determination of sample size, and estimation [NL75]. A discussion of open research issues in audit sampling can be found in [ALS88]. Audit sampling is now so well established that is treated in virtually all auditing texts and has been subject of AICPA auditing standards (see [AIoCPA92]). For a recent survey and annotated bibliography of various sampling designs and statistical methods for auditing see [Gea89].

Perhaps the most familiar use of financial audit sampling to most readers is the U.S. Internal Revenue Service [Hin87], which uses (weighted) random sampling of income tax returns to audit taxpayer compliance with income tax laws and thereby deter tax evasion.

The implications for database management systems (DBMSs) are clear. The commercial market for DBMSs is largely concerned with recording various financial transactions (inventory, sales, payroll, accounts payable, accounts receivable, securities markets, etc.). The effective enforcement of financial controls is a major design consideration in constructing financial transaction processing systems. Adequate records must be preserved to facilitate auditing and prosecution of thieves, embezzlers, etc.

For publicly held corporations and public institutions such financial databases are subject to annual audits on behalf of investors, taxpayers and the IRS. Often, institutions will conduct more frequent, ongoing internal audits to attempt to identify and deter fraud, theft and graft. In the course of such audits, the auditors will extract a random sample of the records from the database and attempt to corroborate the recorded information (e.g., by physically inspecting inventories, locating employees on the payroll, examining receipts and purchase orders, etc.). At LBL, for example, the internal audit unit is currently comprised of includes 3 full-time auditors (plus a secretary) (about 0.1% of the total staff) at a cost of $350K (about 0.15% of the total LBL budget or about 10% of the ADP budget) [Ped93]. Other UC campuses and DOE labs have 5 to 10 internal auditors. Note that, for LBL, the audit expenses greatly exceed annual DBMS license costs.

When auditing banks, savings and loan institutions, and other lenders, regulators and commercial auditors are not only concerned with fraud, but also with assessing the risk of loan defaults. Here, they would sample loans from a database containing the bank's loan portfolio, and then investigate whether the borrower was meeting the loan payments and would be able to continue to do so. Bad loan losses from insolvent Savings and Loan institutions have run into the hundreds of billions of dollars. One accounting firm has been fined $400M for inadequate audits of savings and loan institutions [BB92].

This thesis is largely concerned with *simple random sampling (SRS)*. However, current audit practice typically uses:

- *stratified sampling* - where strata are defined by transaction size (in dollars), with sample sizes allocated proportional to total dollar value of each strata,
- *Probability Proportional to Size (PPS)* - sampling without replacement based on the monetary amount of the transaction, or
- *Dollar Unit Sampling (DUS)* (see [LTA79]) also known as *Monetary Unit Sampling (MUS)* - essentially PPS sampling, but with replacement.

The object of all of these methods is to increase the probability of sampling large value transactions. Two of the more popular audit sampling strategies are stratified sampling and dollar unit sampling. Both methods favor large transaction and are straightforward to implement.

Extending our results to stratified sampling is quite simple, inasmuch as stratified sampling employs SRS within each strata. PPS sampling is more difficult (because it is sampling without replacement). Most of the algorithms described in this thesis can be readily adapted to DUS sampling. See, for example, the discussion in Chapter 2 of Wong & Easton's algorithm for weighted sampling and the algorithms for spatial sampling in Chapter 5. I have focused on simple random sampling because it is simpler, and it facilitates the exposition of the algorithms.

What then is the current practice for auditing financial databases? Often, the auditor will extract an entire relation and then apply a sequential (scan) algorithm to the resulting file (see Chapter 2). Alternatively, if he(she) knows that the keys to the relation are reasonably dense (e.g., purchase order numbers, check numbers, etc.) he(she) could generate a random set of keys, load the set as a temporary relation and perform a (semi-)join with the

target relation to obtain a simple random sample. These methods are cumbersome, inefficient in computing resources, and time-consuming for the auditors. Bailey, et al. complain in a 1988 auditing research review [BGH88] that:

> ... there still are unresolved problems when using GASs [General Audit Software] in advanced computing environments. For instance, it may not be possible to ... access complex data structures directly. In such cases, an intermediate flat file is created, raising issues of completeness and correctness of original data representation in the intermediate file. *Another limitation is the cumbersome handling of audit queries.* [emphasis added]

Auditors' jobs would be simpler and faster if the DBMS directly supported sampling.

### 1.6.2 Fissile Materials Auditing

Financial auditing is not the only type of auditing for which random sampling is useful. Auditing nuclear materials stockpiles to detect theft or covert diversion to nuclear weapons programs is another application of audit sampling of direct interest to the U.S. Department of Energy (DOE). Specifically, DOE Order 5633.3A requires that the accountable nuclear material holdings at all DOE contractors be inventoried at bi-monthly, semi-annual and annual intervals. The order specifically sanctions the use of random sampling inspection plans as part of the nuclear materials inventory verification process.

Good [GGH79] advocates the use of monetary unit sampling for auditing nuclear materials inventories, where the monetary unit is defined as grams of fissionable material. Nuclear materials handling facilities typically maintain detailed computerized records of the location of all nuclear materials to track losses (for safety, pollution, arms control) and deter theft. Hence, inspection would likely commence with a monetary unit or stratified sample from the fissile materials inventory database.

Parrish and Mensing [Par90] describe just such a sampling plan and its implementation in conjunction with a nuclear materials inventory database system at Lawrence Livermore National Laboratory. The system employs an INGRES DBMS and comprises approximately 100 MB. Since, INGRES does not support sampling, sampling is performed by:

1. Run COUNT queries grouped by stratification attributes (mass, form).
2. Extract the records for each strata.
3. Perform simple random sampling within each strata by a user written program.

The sampling is stratified according the mass of fissionable materials in each container and form of the material (some forms are more attractive targets of theft). The database is actually a temporal database, recording the complete history of the nuclear materials inventory at Livermore. It is an online database, updated (manually) as nuclear materials are moved or processed. Further details of the system are classified.

Los Alamos (LANL) has a similar system for nuclear materials inventory control and auditing [Car92]. In addition to DOE national laboratories and contractors, the nuclear fuel and nuclear power industry have similar requirements for nuclear materials inventory control

and auditing, supervised by the U.S. Nuclear Regulatory Commission and the International Atomic Energy Commission [BB88, Jae88, IAE80].

Random inspections of weapons storage sites and ships (to verify weapons counts), missiles (to verify MIRV limits), etc. are important techniques for arms control treaty verification and to detect weapons thefts. Inspection targets will be selected from databases of weapons storage locations, etc.

### 1.6.3 Epidemiological Applications

Another important application of sampling from databases is epidemiology, for disease studies [CDH87, BAG87], clinical epidemiology [ZsSVB89, ZsvSS+91], and clinical trials. Sampling is used to select patients for further detailed study (either therapeutic, or observational). The cost (and sometimes risk) of these studies often precludes the use of entire populations.

Here the most common type of sampling is case control (a.k.a. matched) sampling. A (typically stratified) random sample of patients, taken from a medical records (or personnel) database [ZsvSS+91], is chosen as the "cases", stratified by race, sex, and age. For each *case* a small (usually 1 to 5) random set of *controls* is found in the database, which closely match the characteristics of the cases (e.g., race, sex, age, weight, blood pressure, childbearing history). The design and analysis of such studies is described in two monographs [Sch82, BD80].

If the database is fairly small, one will want to do the matching as closely as possible. As Rosenbaum [Ros89]. describes, such truly matched samples can be quite expensive to compute. However, for large medical records databases (e.g., HMO's, large employers), it is often sufficient to obtain the matching (control) cases by simple random sampling within the corresponding (narrowly defined) strata.

For disease studies, the cases are distinguished from the controls by suspect *prognostic variables (attributes)*, e.g., smoking, exposure to suspected carcinogens, etc. Matched samples permit one to reduce the confounding impact of auxiliary variables (race, sex, age, etc.) on the statistical analysis and thereby permit more sensitive analysis of the effect of the prognostic variable(s).

For clinical trials, the cases are distinguished from the controls by therapeutic variables - e.g., the use of a new drug or surgical technique. Matched pairs of patients are randomly assigned to the two therapies.

Increasingly, large employers are deploying health surveillance systems to monitor the health of employees, to provide early warnings of occupationally related health problems. This is especially true of hazardous materials industries, such as the nuclear, chemical and mining industries. The U.S. Dept. of Energy has begun to create such a health surveillance system of all of its present and former contractor employees. Support for stratified random sampling, and case control sampling would be very desirable for such health surveillance databases.

### 1.6.4 Exploratory Data Analysis & Graphics

Exploratory data analysis (EDA) of statistical or scientific data is another area where database sampling would be useful. Typical these analyses are done interactively on random

samples of the dataset. Because of desired interactive usage, performance (e.g, of the sampling method) is a major concern. Often during the course of the statistical analyses one will want to analyze different logical subsets of the data.

There are several reasons for using a sample data set:

- Smaller data sets permit faster statistical computations. This is particularly an issue when doing nonparametric statistics, e.g., CART [BFOS84], which are frequently computationally intensive. Thus interactive analyses may be impractical on larger data sets.

- Smaller data sets facilitate graphical display of data. One common way of visualizing data is via a point cloud. The human eye picks up depth cues from motion, so that rotating the point cloud (in real time) permits the analyst to visualize the data in 3D space. Sampling of the data set reduces the computational requirements for the real time rotation. If the dataset is too dense, the cloud would become opaque, and the method would be useless. In the case of high dimensional data it is common to display and rotate multiple 3D projections of the data simultaneously. This increases the incentive to sample the data set.

- Resampling techniques [Efr82] are a class of sampling-based statistical techniques which includes *bootstrapping*, *jacknifing*, and *cross validation*. These methods involve repeated sampling of datasets to to construct non-parametric confidence intervals for various estimators (e.g., of the mean).

### 1.6.5 Statistical Quality Control

Random sampling has long been used in quality control studies to assess the quality of manufactured goods. A random sample of the manufactured items is obtained, tested (often destructively). From the results of testing the sample, inferences are drawn about the quality of the entire production lot.

Such random sample quality control inspections are performed both by producers of (say) integrated circuits and by consumers (e.g., computer manufacturers) of ICs. The topic has been the subject of numerous articles, monographs, standards, etc. The American Society for Quality Control and its journal are largely concerned with the use of random sampling for quality control.

The connection between quality control and database sampling arises from computer integrated manufacturing, especially batch manufacturing such as ICs. Here, one tracks the manufacturing of parts in a database system, recording the process conditions (temperature, pressure, chemical doping concentrations), machines used, operators, etc. Assessing the quality of the resulting ICs, and tuning the process to increase yields of successful ICs are important activities for IC manufacturers (also disk drive manufacturers). In such computer integrated manufacturing settings, selection of random samples of components for testing will be done by taking a random sample of the database which records the component manufacturing. For process improvement studies, one may well want to select samples from specific machines, or conditions. Alternatively, case control samples may be used to study proposed changes in the IC process to improve yields.

Such quality control sampling is not limited to manufacturing. Large service organizations (banks, airlines, hotels, hospitals, HMO's, government agencies (e.g., IRS), et al.) increasingly sample their transaction databases and check with customers to evaluate service quality. Hospitals and HMOs conduct random audits of patient cases to assess the level of patient care. Researchers conducting large scale molecular biology experiments, such as those the Human Genome Project, also use random sampling methods for quality control.

### 1.6.6 Polling, Marketing Research

The simplest types of telephone polling and market research simply dial phone numbers at random.

However, polling or market research studies (either telephone or postal surveys) directed at specific groups (e.g., Republicans, environmentalists, investors, homeowners, owners of specific types of automobiles, etc.) increasingly generate their samples of voters/consumers from databases recording political affiliation, organizational membership, contributions, home ownership, automobile registrations, etc. [Lar92].

At least three firms, Marketing Systems Group of Philadelphia, Pennsylvania, SDR Inc. of Atlanta, Georgia and Survey Sampling Inc. of Fairfield, Connecticut specialize in generating random samples of phone numbers of selected demographic or geographic classes of households. These firms will generate samples where the probability of inclusion of a phone number is a function of the demographic or geographic characteristics of the telephone exchange [Sta93]. They can also generate random samples from various mailing lists, etc. [Sta93].

The market research industry is estimated at $1 Billion [Sta93]. It is comprised of database vendors, software vendors, sampling service bureaus (which provide individual samples), research service bureaus (which conduct the polling) market research firms (which design and analyze the surveys). Only Marketing Systems Group sells sampling software (with databases) for random digit dialing surveys [Sta93].

The increasing use of laser scanner-based cash registers for inventory control also permits the construction of large databases of purchases, which can be sampled to study joint product purchasing behavior, or the impact of promotional advertising. If checking account or credit card numbers are recorded, it is possible to link together individual purchasing activities over time [Lar92].

Similarly, credit and debit card usage histories, and checking account histories, can also be used to generate enormous databases which can be sampled for marketing analyses [Lar92]. Credit card payment history databases can be sampled for statistical analyses to estimate parameters for credit scoring.

### 1.6.7 Censuses, Official Surveys

An important use of database sampling arises in conducting official surveys of individuals, businesses, households, etc. Typically, a *frame* (i.e., a database) of the candidate population is constructed from various types of administrative records (building permits, tax returns, licenses, social security records, etc.). Random samples of the frame are then used to direct the survey interviewers (or phone or postal surveys).

One important use of such surveys is to assess the coverage of enumerative censuses, e.g., the decennial U.S. census. In the U.S. this is called the *the post enumeration survey*. Surveyed individuals are matched against the census records to determine if they were counted the first time round. The count of those who were not is used to estimate the undercount.

Various cities, counties, states, etc. have attempted to use their own administrative records to estimate local undercounts.

Another important use of database sampling is for official surveys which are used to directly estimate various social statistics. Once again frames (registries of sampling units - persons, households, firms) are constructed from various administrative records databases, and then sampled. The selected units are then interviewed. Important examples in the U.S. include the monthly Current Population Survey (about 60K households) and the Bureau of Labor Statistics' Establishment Survey (of firms) both of which are used to estimate employment and unemployment statistics for the U.S. The CPS is also used for a wide variety of social statistics. Other annual U.S. surveys include Crime Victimization Surveys, American Housing Surveys, and National Center for Health Statistics Surveys [Mai91]. Aside from agency publications, many details about the construction of these surveys can be found in the annual American Statistical Association Conference Proceedings, Section on Survey Statistics and the Annual U.S. Census Bureau Research Conference Proceeedings.

Of particular interest to the U.S. Dept. of Energy is the triennial Commercial Buildings Energy Consumption Survey (CBECS) [BG91]. This survey is concerned with estimating total U.S. energy consumption in commercial buildings, broken down by end use (lighting, heating, etc.) and fuel source.

The CBECS survey employs a four-stage cluster sample design, with both stratified and PPS sampling of clusters and buildings. Sampling units, and large building samples are taken from existing databases. Sampling of individual (small) buildings is done from enumerated lists of buildings within the final sampling units. The building manager of each sampled building was interviewed and asked to answer a detailed questionnaire concerning the building and its energy consumption.

Note that the large building sample used by CBECS involves sampling from the union of several lists of buildings, and employs algorithmic ideas similar to those developed in this thesis for one-pass spatial reservoir sampling.

Another example of the use of survey sampling for energy consumption studies is the study by Blumstein, et al. [BBM85] concerning commercial energy use in San Diego, CA. This study was use PPS sampling from the projection (onto *buildings*) of *utility account* data. (A single building might have multiple utility accounts).

Finally, it should be noted that the IRS conducts a variety of statistical analyses of samples of tax records, both to estimate various economic statistics, and to study proposed changes in tax laws.

### 1.6.8 Statistical Database Security and Privacy

Protecting the privacy of data about individuals contained in statistical databases has been a key problem for the U.S. Census Bureau and other statistical database generators. It is also an increasingly important issue with statistical databases of purchasing behavior

which are used for marketing analyses [Lar92]. For a recent survey article on the topic see Guthrie [Gea89]. The discussion below is based on [Den80, Gea89] which contain extensive bibliographies.

A variety of methods of disclosure control have been attempted, but most have succumbed to various sophisticated attacks. Such attacks are often facilitated by the provision of sophisticated query facilities in modern statistical databases. If users can supply enough information to identify an individual, they can determine the value of confidential attributes by asking highly specific COUNT queries which will either return 0 or 1. In particular, methods which simply refuse to answer small cardinality COUNT queries can be defeated by *trackers* [DDS79, DS80], queries which pad their response counts to sufficient size to circumvent the restrictions on small COUNT queries. The padding introduced by the *trackers* can be removed, to reveal the desired counts and individual data. Such trackers could, in principle, be defeated by refusing to process queries whose intersections (with previous queries) were too small. However, no practical method is known to do this [Den80].

Attention has thus focused on methods which avoid providing precisely correct answers to aggregate (COUNT, SUM, STD. DEV.) queries. Such methods include:

- Rounding the query results,
- Perturbing the query results with additive or multiplicative noise,
- Perturbing the data values with additive or multiplicative noise (preferably keeping the noise values separately in the database to facilitate internal operational uses)
- Swapping data fields between "similar" records
- Aggregating attribute categories
- Averaging data values across groups of records
- Random Sample Queries

Dorothy Denning [Den80] described how random sampling queries could be used to provide security and privacy for statistical databases. Instead of answering queries against the database, the queries are run against a sample of the database. By introducing sampling errors, random sampling queries effectively preclude the precise computations used by various "trackers" to deduce information about individuals.

However, by rerunning the same sampling query many times and averaging the results, trackers might eventually obtain sufficient precise statistics to deduce private information. This can be defeated by performing the sampling in such a fashion that rerunning a query will yield an identical sample. This can be done if the pseudorandom numbers which are used to decide if a record is included in the sample are functions of the query and database record, rather than a product of a pseudorandom number generator. For example one might compute a hash function on all of the record attributes to generate a pseudorandom number to determine inclusion of the record.

Denning provides an extensive discussion of various aspects and potential threats to the system. Small cardinality queries are still a problem and would require suppression. This was confirmed in a later study by Duncan [Dun91]. Denning's method is quite attractive

because it can readily be integrated with existing statistical analysis techniques, it is fast, and it does not require significant additional storage.

In a later paper, Palley [PS87] argues that regression techniques can be used to recover approximate estimates of confidential parameters. He tested his methods by estimating income statistics from the U.S. Census sample microdata file. His estimates had an $r^2 = 0.5$ correlation with the true income statistics. While his method is not nearly as accurate as the previous trackers (which could give exact answers), it is still disquieting. He found the method could be used against Denning's random sample queries.

However, for regression attacks to work, it must be possible to construct a reasonably reliable regression model for the confidential attribute. Clearly, income statistics are among the most vulnerable, since they correlate with education, size of house, number of bathrooms, etc. Other confidential attributes, such as medical status, are presumably less vulnerable to such attacks.

Ironically, the implementation of Palley's method relies heavily on randomly generated SUM, COUNT, and STD. DEV. queries over the independent random variables. Large numbers of queries (e.g., 300) were required. Cheon, in a subsequent paper [Che91], improved the performance of the method with small samples.

The U.S. Census Bureau routinely uses random samples (1 and 5 percent sample microdata files) for disclosure suppression in distributing micro-data (individual level data) files. The low sampling fractions discourage efforts to identify individuals.

## 1.7 Organization of Thesis

The remainder of the thesis is organized into 6 more chapters.

- Chapter 2 contains a review of basic sampling techniques, and a discussion of related literature on sampling from databases. In this chapter I explain acceptance/rejection sampling and reservoir sampling which I will use throughout the thesis. I also discuss the use of partial sum trees for weighted sampling and their adaptation to B-trees. The literature survey is largely concerned with the development of various sampling techniques and estimators for estimating the size of query results.

- Chapter 3 is concerned with sampling from $B^+$ tree files, I describe a class of algorithms based on acceptance/rejection sampling. Such methods are needed to overcome the non-uniform fan-outs of $B^+$ trees and to assure uniform inclusion probabilities for each record.

- Chapter 4 covers sampling from hash files. I discuss sampling from Open Address Hash Files, Separately Overflow Chained Hash Files, Linear Hash Files and Extendible hash files. I again use acceptance/rejection sampling to overcome the effects of non-uniform bucket loading.

- Chapter 5 is concerned with spatial sampling. Specifically, I consider the problem of obtaining a uniform spatial point sample from a region which is specified as the union of a set of polygons. The polygons need not be disjoint. Sampling from quadtrees and R-trees is discussed.

- Chapter 6 describes sampling from relational operators. Here I use acceptance/rejection sampling to facilitate the interchange of sampling and relational operators. The A/R sampling compensates for the non-uniformities of inclusion probabilities introduced by the relational operators. Selection, intersection, set difference, projection and join operators are considered. I also briefly consider sampling from complex relational expressions: select-project-join (SPJ) expressions, cascaded select/intersect/join/difference operators, and complex unions.

- Chapter 7 is concerned with techniques for the maintenance of materialized sample views. Sample view materialization would be used for samples which are repeatedly analyzed and for longitudinal panel survey studies. The techniques described attempt to maximize the reuse of previously obtained sample elements and still return a simple random sample of the current view. The techniques combine classic view update techniques with database sampling algorithms.

# Chapter 2

# Literature Survey

## 2.1 Introduction

In this chapter I survey the literature on related work on techniques for sampling from files and databases. I review previous work on basic techniques for sampling from single files which either already exist or are being generated in their entirety. I then turn to to related work on sampling from databases: to answer aggregate queries and for query optimization purposes. This literature survey sets the stage for the presentation of my work on implementing sampling queries, which commences in the following chapter.

## 2.2 A Review of Sampling from Files

Over the last 20 years there has been considerable work done on developing basic techniques for sampling from a single flat file (usually with fixed blocking). I employ some of these techniques for query sampling. In Table 3 I list the major results, with citations to the relevant algorithms.

| Type of sampling | Citation | Expected Disk Accesses |
|---|---|---|
| SRSWR | [Yao77] | $O(s)$ |
| SRSWR, variable blocking | [ORX90, OR90] | $O(s(b_{max}/b_{avg}))$ |
| SRSWOR | [EN82] | $O(s)$ |
| Weighted RS | [WE80] | $O(s \log n)$ |
| Sequential RS, known population size | [FMR62] | $O(n/b_{avg})$ |
| | [Vit84] | $O(s)$ |
| Sequential RS, unknown population size | | $O(n/b_{avg})$ |
| | [Vit85] | $O(s(1+\log(n/s))$ |

Table 2.1: Basic Sampling Techniques from a single file. Assume each sample taken from a distinct disk page, i.e., $s \ll (n/b_{avg})$ For Vitter's algorithms assume random disk I/O.

## 2.3 Types of Sampling

There are a variety of types of sampling which may be performed. The various types of sampling can be classified according to the manner in which the sample size is determined, whether the sample is drawn with or without replacement, whether access pattern is random (e.g. from disk or RAM) or sequential (e.g., from tape or on-the-fly from intermediate results), whether or not the size of the population from which the sample is drawn is known, and whether or not each record has a uniform inclusion probability (*simple random sampling* vs. *weighted random sampling*).

## 2.4 Binomial sampling

Binomial sampling is often provided (e.g. in the SIR DBMS) because it can be implemented very easily. One merely sequentially scans the file, generating a random number uniformly distributed between zero and one for each record. If the random number is less than the sampling fraction, the corresponding record is included in the sample. The algorithm runs in time linear with the file size, i.e., $O(n)$, where $n$ is the number of records in the base file.

Alternatively, if the population size, $n$, is known one can generate the sample size from a binomial distribution, and then apply the algorithms for generating fixed size sample discussed below.

A third possibility is to generate the random intervals between successive samples, using a geometric distribution. The records in the intervals are skipped, and the records at the end of each interval are included in the sample. This does not require knowledge of the population size. Analogous methods have been developed by Vitter [Vit84, Vit85] for sequential sampling of *fixed size samples.*

## 2.5 SRSWR from disk

The simplest type of fixed size sampling consists of simple random samples (i.e., unweighted) with replacement (SRSWR) drawn from a file of known size stored on disk as fixed size records (i.e., fixed blocking).

The sample of size $s$ can be obtained by generating uniformly distributed random numbers between 1 and $N$, the number of records in the relation, and reading (random access) the corresponding records. This requires $O(s)$ cpu and disk time. The algorithm can be improved by sorting the random record numbers before retrieving the records. This will reduce the seek time, *assuming that the disk file is allocated approximately monotonically and that the user retains control of the disk arm.*

## 2.6 SRSWOR from disk

Simple random sampling without replacement (SRSWOR) can be done by sampling with replacement and checking a hash table comprised of the records already sampled for

duplicates. If duplicates are found, additional samples are taken. This approach works well if the sample size is a small fraction of the total population.

A better approach [EN82] consists of building a hash table of sampled record numbers, together with substitute record numbers. Each time a record is sampled, its number is inserted in the hash table along with the number of the last unsampled record in the relation. Subsequent record numbers are drawn uniformly from a truncated range 1 to $N - k$ (after the $kth$ record has been sampled). The advantage of this approach is that fewer random record numbers need to be generated.

## 2.7 Weighted Random Sample

Weighted random samples, in which the inclusion probability is proportional to some parameter of the item sampled (e.g. size), are often sought.

I discuss briefly the three major methods of obtaining weighted random samples: *acceptance/rejection* [Rub81], partial sum trees, and the alias method. The three methods vary in sampling efficiency and update efficiency, with acceptance/rejection providing the worst sampling efficiency and the easiest updating, while the alias method provides the most efficient sampling and most difficult updates. Partial sum trees provide intermediate performance on both updates and sampling.

### 2.7.1 Acceptance/Rejection Sampling

Suppose that one wishes to draw a weighted random sample of size 1 from a file of $N$ records, denoted $r_j$, with inclusion probability for record $r_j$ proportional to the weight $w_j$. The maximum of the $w_j$ is denoted $w_{max}$.

One 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 $p_j$:

$$p_j = \frac{w_j}{w_{max}} \tag{2.1}$$

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, one repeats the process until some $j$ is accepted.

The reason for dividing $w_j$ by $w_{max}$ is to assure that one has a proper probability (i.e., $p_j \leq 1$). If one does not know $w_{max}$ one can use instead a bound $\Omega$ such that $\forall j, \Omega > w_j$,.

One can view this acceptance/rejection procedure as a Bernoulli process with success probability $E[p_j]$. The Bernoulli process has been extensively studied. It is well known that the number of trials (here iterations) until the first success (accepted record) has a geometric distribution with a mean of $(E[p_j])^{-1}$. (Note that I count the last (successful) trial.) Hence, using $\Omega$ in lieu of $w_{max}$ results in a less efficient algorithm. For sampling with replacement, it is also well known from the theory of Bernoulli processes that the number of iterations required to accept $s$ records with be a negative binomial distribution with mean $s(E[p_j])^{-1}$.

Acceptance/rejection sampling is well suited to sampling with *ad hoc* weights or when the weights are being frequently updated. Other methods, such as the partial sum tree method

discussed below, require preprocessing the entire table of weights. Acceptance/rejection sampling has long been used for generation of non-uniform pseudorandom number generation [Rub81].

### 2.7.2 Partial Sum Trees

Wong and Easton [WE80] proposed to use binary partial sum trees to expedite weighted sampling.

As above, consider the file of $N$ records, in which each record $r_j$ has inclusion probability $w_j$ in a sample of size 1. Binary partial sum trees are simply binary trees with $N$ leaves, each containing one record $r_j$ and its weight $w_j$. Each internal node contains the sum of the weights of all the data nodes (i.e., leaves) in its subtree. Each record, $r_j$, can be thought to span an interval $[\sum_1^{j-1} w_j, \sum_1^j w_j)$, of length $w_j$.

A sample of size 1 is obtained by generating a uniform random number, $u$, which ranges between 0 to $W$, where $W = \sum_1^N w_j$. The partial sum tree is then traversed from root to leaf to identify the record which spans the location $u$.

The height of the tree is $O(\log N)$, where N is the number of records. Hence the time to obtain a sample of size $s$ is $O(s \log N)$. The tree can also be updated in time $O(\log N)$ should the record weights be modified, or if sampling without replacement is desired.

Partial sum trees can be constructed in the form of B-trees, in order to minimize disk accesses by increasing the tree fanout (and hence the radix of the log). Alternatively, a partial sum tree may be embedded into a B-tree index on some domain.

Partial sum tree sampling may well outperform acceptance/rejection sampling. Essentially, it is another index, specially suited to sampling. However, it is practical only when the weights are known beforehand. Like any other index, it increases the cost of updates.

However, I believe that updates will greatly outnumber sampling queries in most applications. Hence acceptance/rejection methods will be preferred in most applications.

### 2.7.3 Alias Method

Another method of weighted sampling is the *alias* method proposed by Walker [Wal77]. This method is somewhat similar to acceptance/rejection methods. A record is randomly chosen via a uniform distribution and then an acceptance/rejection test is performed, using an adjusted weight. However, if a record is rejected, then a precomputed *aliased record* is supplied in its place. Thus the time to obtain a sample of size $s$ is is simply proportional to the sample size, i.e., $O(s)$. Before one can commence sampling one must compute a table of adjusted sampling weights and aliases for each record. The table is of size $O(n)$, the population size. The algorithm given by Walker to construct the table requires time $O(n^2)$. However, better data structures and search algorithms can reduce this to $O(n \log n)$.

Walker does not indicate any method of updating the alias and adjusted weight tables. Hence this method would only be useful for static databases.

## 2.8 Sequential sampling, known population size

Sequential sampling of a population of known size arises when sampling from a tape file of known size. Disk files of known sizes may also be sampled sequentially, either to reduce the disk seeks generated by random accessing, or because the file is sorted and the user also wants the sample to be sorted in the same fashion, e.g., for a report.

One algorithm for obtaining a sample of size $s$ from $n$ records includes the $k'th$ record ($k = 1, 2, 3, ....n$) with probability

$$p_k = \frac{s - u}{n - k + 1} \tag{2.2}$$

where $u$ is the number of records collected into the sample thus far, i.e., $u = 0, 1, 2, 3, ...s$. This algorithm and its proof were published by Fan, et al., [FMR62] in 1962. It requires time O(n), i.e., proportional to the size of the file. Fan, et al.'s paper contains several other sampling algorithms. The algorithm also appears in both the first and second edition's of Knuth's *Semi-Numerical Algorithms* [Knu69, Knu81].

If the file is on a random access device such as disk then the fastest algorithm is due to Vitter [Vit84]. His algorithm generates the random intervals between successive records which are to be included in the sample. Hence his algorithm requires that only $O(s)$ random numbers be generated, where $s$ is the target sample size. If one can skip records in zero time (e.g., fixed size records on disk) then the total running time will be $O(s)$, otherwise one may be forced to read every record in time $O(n)$.

## 2.9 SRSWR from disk, variable blocking

Variable numbers of records per page (variable blocking) may arise due to variable record size, hash file organizations, or deletions. If an index exists one can use it as above. If no index exists one can use *acceptance/rejection* sampling (discussed above in Section 2.7.1 and in [Rub81]) on the pages with

$$\text{prob(accept this page)} = \frac{\text{no. of records on this page}}{\text{max. no. records per page}} \tag{2.3}$$

If the page is accepted one selects a record on the page at random. This assures uniform selection probabilities for all records. This algorithm for sampling was described in [OR90] and [ORX90] (where it was applied to hash files). As above (in Section 2.7.1) the expected number of iterations until one accepts a record will be:

$$\frac{\text{max. no. of records per page}}{E[\text{no. of records per page}]} \tag{2.4}$$

If the actual maximum number of records per page is unknown one can always use an upper bound (with attendant loss of efficiency).

DeWitt, et al., [DNSS92] adapted this sampling method for use percentile estimation and called it *extent map sampling*. In DeWitt et al.'s paper the file structure is represented via an extent list (i.e., a list of contiguous regions on disk allocated to consecutive logical disk blocks, as used in DEC/VMS operating system), so that the i'th block can be located by an in-memory search of the extent list. DeWitt et al. used all the tuples on the retrieved disk block, *page level sampling*. Otherwise the algorithms are identical.

## 2.10 Sequential sampling, unknown population size

Sequential sampling of a population of unknown size arises when sampling a tape file of unknown size, sampling the output of a query as it is generated (to avoid writing the entire output to disk), or online sampling of transaction streams.

The algorithms for sequential sampling of a population of unknown size are known as *reservoir algorithms*, because they create a *reservoir* of size $s$ (the desired sample size) of candidate sample records. In all of these algorithms the reservoir is initially filled with the first $s$ records read. The reservoir algorithms then proceed sequentially through the file, updating the reservoir, so that it always contains a simple random sample. Thus the $k'th$ record encountered ($k > n$) is included in the reservoir with probability $s/k$. If it is decided to include the $k'th$, a record in the reservoir is chosen randomly to be replaced.

The algorithm described above appears in the second edition (1981) of of *Semi-numerical Algorithms* [Knu81] by Donald Knuth, who attributes the algorithm (without citation) to Alan Waterman. Mcleod and Bellhouse describe the same reservoir sampling algorithm (and prove its correctness) again in [MB83] in 1983.

An earlier reservoir (less efficient) algorithm appeared in the first edition of Knuth's *Semi-Numerical Algorithms* [Knu69]. The earlier algorithm was first published by Fan, et al. [FMR62] in 1962. This algorithm generate a uniform random variate for each record, and always keeps the records with the $s$ smallest random variates in the reservoir.

The fastest reservoir algorithm for random access files is by Vitter [Vit85], who has extended his work on known population size samples to the case of unknown population sizes. As before, Vitter generates the random intervals of records to be skipped. Hence he examines only those records which get put into the reservoir. The running time for his algorithm is $O(s(1 + \log(n/s))$, assuming that skipping can be done in zero time, i.e., that one has random access to the file.

## 2.11 Database Abstracts for Estimating Query Sizes

Before commencing a discussion of the use of sampling for selectivity and query size estimation, I briefly consider alternative approaches to estimating predicate selectivity and query result sizes based on keeping various statistics about the database.

Predicate selectivity, the number of records satisfying a selection predicate, is a key parameter used by the query optimizer to choose between scanning a relation to test a selection predicate, or using a secondary index to evaluate the selection predicate.

### 2.11.1 Anti-Sampling

Neil Rowe [Row83, Row85, Row88] has perhaps been the most outspoken advocate of anti-sampling. In his doctoral dissertation and subsequent papers he proposed keeping a database statistical abstract, comprised of various count, sum, and other statistics defined over various partitions of the database. He then developed a set of techniques for combining the various statistics to construct bounds on other statistics or portions of the database.

The required statistical tables can be quite modest if only uni-dimensional histograms, etc. (for simple predicate selectivity estimation) are required. If more complex predicates

are anticipated, then one would want to construct multi-dimensional histograms, and other statistics, which consume more space. Practically speaking, it appears unlikely that one would keep more than histograms of dimensionality higher than two. If the requisite statistics for a relation are stored together, they could be read in a single disk read. The required computations are quite fast. Hence, Rowe argued that his approach was much faster than sampling, although the statistics would have to be updated during database updates (raising issues of concurrency control hot spots). I am inclined to agree with Rowe that for many simple (single predicate) queries, database abstracts will be more efficient that sampling. Current commercial databases (e.g., INGRES) largely rely on such histograms for selectivity estimation.

The use of histograms for selectivity estimation was first adopted for the INGRES project. Histograms of the distribution individual attributes in each relation were computed. Initially, the bins used for the histograms were chosen to equi-width, i.e., the range of the attribute was partitioned into equal intervals. Equi-depth histograms are determined by choosing the attribute partitioning so that each histogram bin has the same count, i.e., the bin boundaries are uniformly spaced quantiles of the empirical attribute distribution. Subsequently, Piatetsky-Shapiro in [PSC84] argued that equi-depth histograms (i.e., bins), rather than the then common equi-width histogram bins gave better control over errors in estimation selectivities. He used sampling of the relations to generate the histograms. Muralikrishna & DeWitt [MD88] described how to construct equi-depth multi-dimensional histograms using a variant of the R-tree, which they call an H-tree. They also used sampling of relations to generate the histograms.

### 2.11.2 Partial Sum (Ranked) Trees

It has long been known [Knu73], that one could store the number of leaves of subtrees in the internal nodes of a tree in order to determine the rank of a leaf in logarithmic time, i.e., by summing counts of all the left branches passed on the way down the tree to the target leaf. This can be used as a *selectivity index* to rapidly determine the selectivity of range predicates (for which a B-tree index exists). The method can be readily extended to computing any desired additive statistic (SUM, COUNT) over a range in an ordered tree. This has been described earlier in this chapter for weighted sampling by Wong and Easton.

Variations of the technique have been rediscovered by several authors. Bennett and Kruskal use to implement efficient LRU stack processing algorithms in [BK75]. Stonebraker [SSL+83] used the technique to implement ordered relations to support text editing of documents with a relational DBMS. Srivastava [SL88] used the method with $B^+$ trees to compute a variety of aggregate statistics (SUM, COUNT), order statistics, and sampling (stratified, random, systematic). Ghosh [Gho88] employed the use of partial sum *tries*. The EXODUS project at Univ. of Wisconsin [CDRS86] uses the technique in an access method to support insertion and deletion from sequences. Most recently, Antoshenkov [Ant92], has described an approach which computes bounded approximations of rank statistics. By allowing some slop in the counts stored in each node, Antoshenkov avoids the need to propagate changes in the counts all the way up the tree on each modification to the tree (insert/delete).

## 2.12 Sampling for Estimation

One common use of sampling is to estimate the size of relational query results. This may be done to give a user an approximate answer (e.g., for a COUNT query), or it may be used to estimate selectivities or intermediate result sizes for query optimization purposes. Note that if one wants the information for query optimization, then one typically prefers to have the block selectivity estimate (rather than record selectivity). The block selectivity estimate is a better predictor of I/O time. This point is discussed further in the next section.

### 2.12.1 Cluster Sampling

Hou and Ozsoyoglu wrote several papers [HOT88, HOT89, HO91] on estimating the results of COUNT queries. Their motivation was real time control systems [HOT89].

Their approach involved cluster sampling (where disk pages of relations constituted the clusters). Having read the disk pages, they argued that one might as well use all of the information on each disk page. Of course, the sample elements are not necessarily independent. One can correct for this in the computation of the estimator, and its variance.

Cluster sampling is worthwhile when there is significant variability of records on the page in terms of the attributes sought (intra-cluster variability). If the relation is ordered on the attributes being sampled, then there will be little intra-cluster variability and clustered sampling may offer very little improvement over simple random sampling. If however, the relation is ordered on some attribute irrelevant to the desired attribute (e.g., name vs. salary) then there will be great intra-cluster variability and clustered sampling will be quite worthwhile.

Special estimators of the total tuple count were created by Hou & Ozsoyoglu [HOD91] to account for the effects of relational operators. They discuss estimators for complex aggregate (e.g., COUNT) relational queries.

Their methods are attractive if the relations are not ordered with respect to attributes of interest, and either the relations are of moderate size ($10^6$ records or less) or no indices are available for the relations.

### 2.12.2 Sequential Sampling

Lipton & Naughton in [LNS90, LN90] revived the sequential sampling (which they called *adaptive sampling*) approach of Wald [Wal47] to selectivity estimation. Here sequentiality is statistical, i.e., one decides after each element sampled whether to continue sampling (in contrast to the usage in DBMS, where sequential algorithms usually refer to sequentially scanning the relation). It is well known that sequential sampling algorithms outperform conventional single-stage sampling algorithms in terms of the number of sample points required, since they can adjust the sample size to the population parameters.

Note, however, that sequential sampling policies effectively require iterative sampling algorithms - rather than the more efficient batch or sequential scan algorithms. However, Lipton & Naughton did not consider the efficiencies to be garnered from "batch" (or sequential scan) sampling. Batch sampling algorithms are more efficient because they avoid rereading pages (esp. in the upper part of a tree index). Sequential scan sampling algorithms may be more efficient due to reduced seek time of sequential vs. random disk reads.

While such efficiencies may be insignificant for hashed files, they are potentially significant (e.g., a factor of 3-4) for $B^+$ tree files.

In a subsequent paper, Haas & Swami [HS92a, HS92b] developed improved stopping rules for sequential sampling of selectivity estimation. Haas & Swami first observed that Lipton, et al. were using apriori bounds for the mean and variance of the population in their stopping rule. Haas & Swami therefore suggested estimating the mean and variance for the stopping rule from the sample taken thus far. They prove that the their sampling procedure is asymptotically efficient, i.e., it uses the minimum sample size for the required accuracy. Note that they use a hybrid accuracy constraint: absolute accuracy for small values, and relative accuracy for large values. Haas & Swami also suggest the use of "batch" sampling on the grounds that traversing the access manager interface for each sample is inefficient; they ignore the savings in I/O costs attainable with batch sampling (akin to batch searching), Their experimental results suggest that this sampling procedure is comparable to the two stage sampling method [HOD91] discussed below and better than Lipton, et al.'s procedure.

Haas & Swami then analyze the use of stratified sampling, with uniform sample sizes allocated to each strata. Such an allocation of sample sizes is appropriate if the strata have equal variances. (They suggest checking this assumption at the conclusion of the sampling and resort to additional sampling of large variance strata.) Their experimental results suggest that this algorithm offers slightly to much better coverage (depending on intra-strata homogeneity) than their earlier algorithm and consistently beats Lipton et al.'s procedure.

Haas & Swami also incorporate some improvements in the sequential sampling procedure to correct for undercoverage, the tendency of the sequential sampling algorithms to report confidence intervals which include a smaller proportion of the sample distribution than nominally specified.

### 2.12.3 Two Stage Sampling

In [HOD91] Hou, et al. describe the use of two stage sampling to estimate COUNT query results. The first stage sample is used to estimate the required sample size for the second stage sample. For a specific sampling design one would expect that two stage sampling is intermediate between single stage and sequential sampling in terms of expected sample size for a given level of accuracy. It is attractive in the database setting, as it permits the use of batch sampling algorithms from the base relations (see the discussion in Chapters 3, 4. Chapters 4,3.

Hou, et al., find it attractive because it is readily adapted to their earlier estimation procedure, which effectively computes the relational query on the cross product of the samples from the base (relational) files. They found that the combination of two stage sampling, with their estimator, outperformed the sequential sampling algorithm of Lipton and Naughton [LNS90, LN90].

Hou, et al. also suggest the use of simple random sampling without replacement (SRSWOR) in place of simple random sampling with replacement (SRSWR). That SRSWOR is more efficient than SRSWR has long been known in the statistical literature. Hou, et al. also suggest the use of systematic sampling (SS), presumably with a random starting point. They observe that systematic sampling will outperform SRS when the relation is

sorted (a common practice), but would be worse if there are periodic fluctuations in the value of the desired attribute as a function of the ordinal position the record. Hence, unless aperiodicity is known apriori (and recorded in the DB catalog) or one are prepared to compute the spectra of his(her) databases, it is difficult to see how this strategy could be safely employed.

Hou, et al. note that their methods do not require the use of indices over the relations, in contrast to the methods of Lipton, et al. and many of the methods described in this thesis. This is true, but they pay a large penalty, e.g., for intersections, in sample size when sampling large relations with small intersections, or small join selectivities. Their experiments do not consider such large relations with small intersections or join selectivities. They consider a join selectivity of about $10^{-3}$.

### 2.12.4 Transitive Closure

Lipton & Naughton [LN89] showed how to use sampling to estimate the size of a transitive closure. Essentially, they sample from the base relation, and then compute the size of reachability set for each element in the sample. The estimated size of the transitive closure is then the sum of sizes of the reachability sets of the sample elements times the inverse of the sampling proportion.

### 2.12.5 Parallel Sampling

Seshadri & Naughton [NS90, Ses92] discuss the use of stratified and clustered sampling for parallel sampling on a multi-processor to estimate selectivities [SN91]. They show that simple random sampling is asymptotically inefficient in a parallel environment (as the the number of processors grows) as it leads to heavily skewed workloads - everyone waits for the processor with the largest number of samples. They cite a theorem of Gonnet that the limit of maximum path length for hash overflow (as $n/k \rightarrow \infty$) is $\log n / \log \log n$, where $n$ is the number of records and $k$ is the number of buckets. They then propose the use of stratified sampling (stratifying over the processors) to achieve perfect load balancing by allocating uniform sample sizes to each strata (processor-disk) pair.

They go on to argue (along the lines of Hou and Ozsoyoglu) in favor of clustered sampling for selectivity estimation, with the disk page being the cluster unit. (See discussion above.)

#### 2.12.5.1 Percentile Estimation

Seshadri & Naughton [NS90, Ses92] discuss the use of random sampling to estimate percentiles of the data distribution. Estimation of percentiles is important in many applications such as parallel sorting, and some types of join computations on parallel machines, e.g., band (approximate) joins or inequality joins. The goal is to minimize *skew* (load imbalance) across the parallel processors, so as to minimize the completion of the join (sorting) operation. For example, sorting (on $k$ processors) can be done by estimating the $k-1$ percentiles of the file, called "splitters", and then letting each processor sort independently all keys which fall between two successive "splitters". Percentile estimation is done by taking a random sample of a sufficient size, and then taking the sample percentiles as estimators.

The goal is to provide a guaranteed bound over the skew among the processors while taking a minimal sample size.

DeWitt, et al., [DNS91a], describe the use of percentile estimation via sampling for use in partitioned band-join algorithms. Band-joins are an extension of equi-joins in which an error tolerance is allowed in the equality test, i.e., the difference between the join attributes of tuples from two join relations must be less than some error bound.

Similarly, the same authors [DNS91b] describe the successful implementation a partitioned external sorting algorithm where the partitioning is based on a splitting vector (quantiles) estimated via sampling. They also provide an extensive discussion of the literature on the use of sample quantiles in various types of partitioned (e.g., distributive) sorting algorithms (both main-memory and external). The asymptotic analysis of distributive sorting algorithms is also discussed in [Dev86].

### 2.12.6 Estimating the Size of Projections

Estimating the size of projections is a classical topic in database research and has a long history in the statistical literature, where the problem is known as "estimating the number of species (classes)". Bunge and Fitzpatrick provide a recent survey of results on the problem and an extensive bibliography (over 100 citations) in [BF93]. Here I discuss only one recent paper from the database literature.

Naughton and Seshadri [NS90] describe the estimation of the size of projections via sampling. Approaches to the problem vary depending on whether there is an index available on some or all of the projection attributes:

1. Do the projection, then count.

2. Scan the relation, doing probabilistic counting [WVZT90, FM85, Fla85, FM83, ASW87].

3. Sample the base relation; for each element of the sample find the number of the records in the base relation which match the sampled record on the projection attributes, call this $|x_i|$. Estimate the size of the projection as the estimated average contribution of each element of the base relation to the projection size times the size of the base relation, i.e.,

$$\widehat{|\pi_R(|)} = (\frac{|R|}{s}) \sum_{i=1}^{s} |x_i|^{-1} \tag{2.5}$$

   where $s$ is the sample size, and $|R|$ is the cardinality of relation $R$. This estimation procedure would require that the relation is sorted/hashed/indexed on the projection attributes.

4. Sample the base relation, count duplicates (over the projection attributes) of tuples in the sample. Estimate the size of the projection using Goodman's estimator [Goo49], etc. This method does not require that the base relation be indexed. However, this method is very unreliable when the sample size is too small to encompass a significant number of duplicates. For example, suppose one has a base relation of $10^8$ records, whose projection (on some particular projection domain) is $10^7$ records. Then a simple random sample of 100 records will probably contain no duplicates. But such a sample

would probably not contain any duplicates whether the projection size was $10^6$, $10^7$, or $10^8$ records. Hence, this method is not an accurate method of estimating projection sizes for large projections when the sample size is small. Note that Hou and Ozsoyoglu [HOD91] also use Goodman's estimator for the size of projections.

Naughton and Seshadri propose yet another estimator, a variant of the third algorithm described above. They assume that the projection, $\pi_{A,B}(R)$ is on two attributes $A$ and $B$. The also assume that the base relation, $R$ is sorted/hashed/indexed on $A$. They sample a tuple $x \in A$. They then determine the cardinalities $\pi_i$ of the tuples of $R$ which match on $A$, and $l_i$ the number of tuples of $R$ which match on $A, B$. Their estimator is thus $(1/s)\sum_{i=1}^{s} \pi_i/l_i$, where $s$ is the sample size.

Note that this method implicitly assumes that relation $R$ is at least indexed/sorted/hashed on $A$, but not necessarily on $A, B$. If not, one must do a scan of the relation. Thus it is intermediate in its indexing requirements.

Naughton and Seshadri show that their estimator of the size of a projection will outperform Goodman's estimator (esp. for small samples). This is hardly surprising. Note that this method commences with a simple random sample of the base relation using the methods described in this thesis.

### 2.12.7 Estimating the Size of Joins

Seshadri (in Chapter 2 of his thesis [Ses92]) considers the problem of estimating the size of joins (or equivalently join selectivity) via sampling. He first considers several variations of the cross product sampling approach discusses (above) by Hou and Ozsoyoglu.

He considers:

- tuple independent sampling, SRS sampling of pairs of tuples from the two relations $R, S$.
- tuple cross product sampling, the cross product of SRS samples of tuples from each relation $R, S$,
- page independent sampling, cluster (disk page) sampling of pairs of pages from the two relations $R, S$,
- page cross product sampling, the cross product of cluster (disk page) samples from each relation $R, S$.

Seshadri found that cluster (disk page) sampling was always at least as good as the corresponding tuple level sampling. He also found that cross product sampling was always at least as good as the corresponding independent sampling. He was unable to establish an order among independent cluster (page) sampling and tuple cross product sampling. Clearly, clustered (page level) cross product sampling is the preferred algorithm - *when no index is present.*

Seshadri goes to describe and analyze an *index based join selectivity estimation algorithm.* His algorithm is related to the simple random sampling join sampling algorithm discussed here in Chapter 6. Basically, he samples from one relation, say $R$, and then looks

up in the index the number, $m_i$, of matching tuples in $S$. The estimator of the join size is simply:

$$|R|(1/s)\sum_{i=1}^{s} m_i \tag{2.6}$$

Seshadri shows that this estimation algorithm outperforms the previous algorithms.

## 2.13 Block Selectivity Estimation

In the previous section I discussed papers which were concerned with the use of sampling to estimate record selectivity or intermediate result size (in records). However, for query optimization purposes, it is not record selectivity which is needed, but rather block selectivity, because the disk reads are done in units of disk blocks.

Random record placement models [Car75, Yao77] derive the block selectivity estimates from the record selectivity estimates, under the assumption that the records are randomly placed with respect to the blocks. This assumption is appropriate for estimating the block selectivity of simple random sampling queries. Several authors have extended these results to various types of tree structured (e.g., $B^+$ tree) files [SG76, Pal85, LDJ89, LM90] under the rubric of "batch searching" of files.

Christodoulakis [Chr84b, Chr84a] discusses block selectivity estimation where records may be clustered together, e.g., if the file is sorted on the attribute used in a selection predicate. Note that block selectivity can also be estimated via sampling techniques used to estimate the size of projections (onto the block ID attribute).

If blocks are referenced randomly, or if we lose control of the disk drive between successive block reads then one could model expected I/O time as simply a constant times the number of blocks accessed. However, if the blocks to be read are spatially clustered (within and among disk cylinders), *and one can hang onto to the disk long enough to read nearby blocks*, then I/O time models should at least account for cylinder selectivity, and perhaps cylinder proximity. The advent of workstations with local disks makes this a plausible scenario. It is also plausible for large IBM mainframes in which the number of disks exceeds the degree of multiprogramming. Note that one should be able to use the same techniques for estimating cylinder selectivity from block selectivity as have been used for estimating block selectivity from record selectivity.

## 2.14 Sampling for Query Optimization

Query optimizers often must choose from among several possible query processing strategies. The most efficient query plan is typically a function of the sizes of various intermediate query results, and the selectivities of various partial query predicates.

Sampling can be used estimate these selectivities and the sizes of the results. The alternative is typically to maintain histograms (i.e., summary statistics) to assist in estimating query result sizes and predicate selectivities. As noted above, reality would require that we estimate block and cylinder selectivities, but here I largely confine my remarks to record selectivities.

### 2.14.1 Frequentist Approach

Willard [Wil84, Wil91] considered the problem of determining the asymptotically optimal sample size required to estimate predicate selectivities for query optimization of scan operations. Specifically, he was trying to decide which of two predicates to evaluate first when scanning a relation for a conjunctive query. He showed that the optimal (minimax) sample size to minimize the maximum combined cost of sampling and evaluating the query was $O(n^{2/3})$. This result holds for any loss function linear in the estimated parameter and the relation size, $n$.

Thus for a relation with $10^6$ records, the optimal sample size would be approximately $10^4$ records. However, if the blocking factor (records/disk page) was 50 (implying the file occupies $2.0 \times 10^4$ disk pages) then one would still be reading almost half the pages of the file for a simple random sample. Clearly, cluster sampling is more attractive if feasible. In a main memory database, the blocking factor is effectively one, so the simple random sampling is much more practical.

Willard discounts Bayesian approaches to the problem, arguing that one frequently does not have sufficient information to construct a prior distribution.

### 2.14.2 Decision Theoretic Approach

One can see the precursor of a decision theoretic approach in the work of Chu [Chu89] in estimating block selectivities for access path selection. He first calculates bounds on the block selectivities and considers the query optimization problem - index access vs. complete file scan. If the bounds are sufficient to settle the question of which query strategy to choose, he does not attempt more accurate estimates.

Kevin Seppi [Sep90, SM92] considered the problem of selectivity estimation systematically from the perspective of a decision theoretic framework. Given a prior probability distribution on the parameters of interest (e.g., predicate selectivity), Seppi uses sampling and Bayes rule to compute a posterior probability distribution. The Bayesian posterior probability distribution can be incorporated into a decision rule, along with expected query processing costs (as a function of estimated parameters).

Thus Seppi can estimate the value of additional sampling in terms of improving the query optimization decision. There is no point in additional sampling to refine parameter estimates beyond the accuracy needed to decide between alternative query processing strategies.

Seppi's approach lends itself naturally to sequential sampling techniques and to adaptive query optimization. It also facilitates the inclusion of other sources of information on database parameters (e.g., previous queries, histograms, moments).

### 2.14.3 Applications

DeWitt, et al., [DNSS92] use sampling determine the amount of skewness of the data distribution in two relation, and then to choose among several parallel join processing strategies.

Wolfson, et al., [WZB+93] is concerned with choosing (at run time) among strategies for computing the single source reachability queries. In such queries the relational database

is being used to store a directed graph, and the query asks how many nodes can be reached from a specified node.

Wolfson, et al., use sampling both to estimate the number of nodes in the graph, and the average out-degree of each node. These estimates, together with information on the parallel machine architecture are used to estimate the query processing cost of the various strategies and then choose among them.

### 2.14.4 Commercial Implementations

Antoshenkov [Ant93] describes the use of random sampling for selectivity estimation in the Rdb/VMS query optimizer. This is the only report I know of concerning the use of sampling for query optimization in a commercial DBMS. Antoshenkov apparently only uses sampling to estimate the selectivity of range selection predicates. The selectivity estimation is embedded in an adaptive (dynamic) query optimizer, which reconsiders query strategies after completing the computation of intermediate results.

## 2.15 Related Work on View Materialization

Here I review related work on view materialization, which I will use later in the chapter on maintenance of materialized sample views. Work on view materialization covers two subjects: *policy* (when to update the materialized view) and *mechanism* (how to update the materialized view).

### 2.15.1 Update Mechanisms

Buneman and Clemson [BC79] and Blakeley et al. [BLT86] are concerned with *screening tests* for detecting irrelevant tuples. These are tuples (records) whose insertion or deletion can be shown not to affect the contents of a view, regardless of the state of the database (as determined by a satisfiability computation on the selection predicate after substitution of the tuple attributes [BLT86]). Buneman and Clemson [BC79] also discuss partial evaluation strategies which partially recompute view definitions in an attempt to exclude update tuples whose irrelevance to the view depends on the database content. Blakeley et al. [BCL86] continue their discussion of screening procedures by introducing the notion of *autonomously computable updates*, i.e., updates whose relevance to a view can be determined solely from the update and the contents of the view. Screening test are clearly worthwhile and can also be applied to the maintenance of materialized sample views.

Blakeley et al. [BLT86] also discuss algebraic *differential* view update procedures. Here each updated base relation is represented as:

$$R' = R \cup \{t_{inserted}\} - \{t_{deleted}\} \tag{2.7}$$

and queries are relational algebra expressions. Note that this sort of differential representation of a relation was previously used for the implementation of hypothetical relations by Woodfill & Stonebraker [WS83], and by Agrawal & Dewitt [AD83]. Blakeley et al. expand the relational expressions - exploiting distributivity of the query operators over set union and difference (where possible) - grouping invariant terms and variable terms separately

(both with respect to the updates $\{t_{inserted}\}$ and $\{t_{deleted}\}$). I will use similar ideas to address maintenance of materialized sample views.

In particular, Blakeley et al. [BLT86] discuss the problem of maintaining view with duplicates. Thus if the view contains a projection one must record the count of duplicates for each tuple in the projection, so that one can tell when a deletion from the base relation causes a deletion from the projection view. Alternatively, all views can be required to include keys of each of the base relations. Similar problems are encountered with with sampled projection views.

Hanson [Han87b, Han87a] reviews the differential view update methods developed by Blakeley (described above), the hypothetical relations implementation work of Woodfill & Stonebraker [WS83] and of Agrawal & Dewitt [AD83], and the differential file techniques of Severance [SL76]. Hanson also discusses the implementation of views containing aggregate queries (SUM, COUNT, etc.). As has been remarked, this work employs similar differential view update techniques for the update of materialized sample views. Hanson [Han87a] also considers a RETE network approach to materialized view maintenance. RETE networks are a data flow type computation, used in artificial intelligence to determine when to fire specific rules in a rule-based expert systems. He then considers (at length) the relative efficiency of various view update policies (immediate or deferred) and mechanisms (algebraic differential view update, query modification, RETE network differential view update) under various workload scenarios.

Ceri and Widom [CW91] discuss a system supporting maintenance of materialized views via a rule system in STARBURST. They explain how to automatically derive the necessary rules for incremental view maintenance employing differential update methods. Ceri and Widom permit singly nested queries, but restrict their attention to views which do not include duplicates or aggregate functions. However, they do permit projection operations in the view definitions. In such cases they may require recomputation of the entire view upon deletion of tuples (if the projection domains do not include keys of the base tables).

Srinivasan and Carey [SC92] develops a *compensation-based* approach to processing long-running queries in the presence of update traffic. Their method avoids the use of two phase locking of the tables being queried. Hence, they avoid the delays and possible deadlocks which may arise from the locking. Instead they run the queries without locks and compensate for updates that occur while processing the query. The approach resembles the differential view update techniques described by Blakeley and Hanson in that it exploits the distributivity of relational operators. It differs from previous work in that differential view updating is integrated inside the various relational operator algorithms, e.g., sort/merge joins, hash joins, etc. This is akin to my results on the integration of sampling with relational operators.

Segev & Park [SP89] discuss mechanisms for updating of distributed materialized views. Their work consists of two parts: duplicate tuple elimination and an improved tuple screening test. Segev & Park are concerned with batch processing of updates against materialized views via the differential relation techniques developed by Blakeley, Hanson, et al. as described above. Thus duplicate tuple elimination is concerned with eliminating all but the first and last updates to a particular tuple, as the intervening updates are irrelevant - being overwritten by later updates. Segev and Park also discuss the use multiple query optimization techniques to efficiently implement screening tests (to discriminate between relevant

and irrelevant update tuples) for multiple views. Segev and Park's work can be applied to the problem of materialized sample views.

### 2.15.2 Update Policies

Buneman and Clemons [BC79] address implementation of immediate update propagation policies. Adiba & Lindsay [AL80] and Lindsay et al. [LHMP86] are concerned with periodic view update propagation, e.g., for accounting reporting . Hanson [Han87b, Han87a] compares the efficiency of immediate and on-demand view update policies under various workload scenarios.

In [SF91] Segev & Fang discuss *currency-based* update policies for distributed materialized views. Such a currency constraint enforces a maximum time bound between the state of the base relations of a database and the state of each view. Segev & Park find the optimal update policy for a set of materialized views from the class of update policies which:

- are a hybrid of periodic updates and on-demand updates subject to a currency constraint, and
- which allow update propagation via other views (not just directly from the base relation), where permissible view implications are pre-specified (i.e., from the view definitions and fixed currency-constraints).

In [SF90] Segev & Fang extend their earlier results to allow update propagation via other views, where permissible view implications are not pre-specified, but determined (in part) by the currency-constraints computed by view update policy optimizer. This work on update policies could be extended to the maintenance of materialized sample views.

# Chapter 3

# Random Sampling from $B^+$ trees

## 3.1 Introduction

This chapter begins the heart of this thesis, the novel techniques which I have developed to implement random sampling queries. I start by addressing the most basic question: how does one sample from base relations stored in the most popular access method, a $B^+$ tree file.

Recall that a $B^+$ tree is a B-tree in which all of data records are kept at the leaf nodes, and internal nodes contain only keys used to partition the records. The uniform height of the $B^+$ tree (all leaves are the same distance from the root) and the requirement that all data records be stored at the leaves facilitates the design and analysis of the sampling algorithms.

Virtually all database systems used to record financial transactions (accounting systems, inventory control systems, bank records, etc.) are subject to annual audit, usually involving random sampling of the records for corroboration. Yet commercial database management systems do not support queries to retrieve a random sample of some portion of the database. One reason is that previous proposals to support retrieval of random samples from databases have required the modification of standard access methods, and the maintenance of additional information in the indices. In particular, earlier authors have described sampling from modified $B^+$ trees, *called ranked $B^+$ trees*, which incorporate information which permits the computation of the rank of a record.

In this chapter I discuss methods of sampling from regular $B^+$ trees, employing acceptance/rejection (A/R) methods. These new algorithms do not require any modification of the standard $B^+$ tree structures, nor do they require maintenance of any additional fields in the $B^+$ trees. Hence these new sampling methods can be more easily retrofit to existing DBMSs. While not quite as efficient as earlier proposals, it should be suitable for applications which only need sampling infrequently, e.g. for auditing. For $B^+$ tree files I discuss both iterative and batch sampling methods.

Whereas $B^+$ trees are uniform height trees, simplifying the sampling algorithm, Rosenbaum [Ros91] has recently described a similar algorithm for uniform random sampling from arbitrary trees, i.e., when not all leaves are the same distance from the root.

### 3.1.1 Notation

The notation used in this chapter is summarized for easy reference in Table 3.1. Notation is fully explained in text where is first used. Here I mention some general conventions. The desired sample size is denoted as $s$, whereas $s'$ denotes the inflated sample size (i.e., to compensate for losses due to acceptance/rejection sampling). The number of records in the file is denoted $n$. I denote the fan-out of an $B^+$ tree node as $f_i$, the average fan-out as $f_{avg}$, and maximum fan-out as $f_{max}$. The average number of records per page for a variably blocked files will be $b_{avg}$, the maximum will be $b_{max}$. The height of the standard $B^+$ tree is denoted as $h$. For the ranked $B^+$ tree I denote the height as $h'$. (As I shall show, this is usually the same or possibly one more than $h$.) Acceptance probabilities of records are denoted $\alpha_k$, and the expectation as $\alpha$. Similarly, the acceptance probability of a node is denoted $\beta_k$, and the expectation as $\beta$. The cost (in disk page accesses) of retrieving a sample of size $s$ by a particular method is denoted as $C_{method}(s)$. In computing this cost I typically assume that there is a sufficient cache to hold one entire root-to-leaf path, so that the root node (at least) is always in the cache. For batch sampling, I will need to know the expected number of blocks referenced when retrieving $k$ records (at random) from a file containing $m$ blocks, this will be denoted $Y(k, m)$ (Cardenas's function).

Abbreviations of algorithms used in this chapter are summarized in Table 3.2. I use the suffix "I" to indicate iterative algorithms, which select sample one record at a time. Such algorithms naturally return simple random samples with replacement (duplicates) (SRSWR) but can be modified to return simple random sample random samples without replacement by removing duplicates. I use the suffix "B" to denote batch algorithms, which are the sampling analog of batch search algorithms in trees. Such algorithms naturally return simple random samples without replacement (no duplicates, denoted SRSWOR), but can be modified to return simple random samples without replacement by synthetically generating duplicates.

## 3.2 Iterative Sampling from a $B^+$ tree

### 3.2.1 A/R Sampling from a $B^+$ tree

In this section I explain the use of acceptance/rejection sampling to sample from a $B^+$ tree without requiring the storage of any additional information in the $B^+$ tree nodes. Although this new method has a higher retrieval cost than earlier methods based on ranked $B^+$ trees, it does not require any modification of existing access methods, nor any additional update costs. Hence this method will be preferred over earlier methods for applications where updates dominate sampling retrievals.

For expository reasons I commence with a discussion of the *naive method* (a random walk from root to leaf, followed by an acceptance/rejection test). Subsequently, I show that a modification of this method, known as *early abort*, dominates the *naive method*. In Section 3.3 I consider *batch* versions of each algorithm, which in turn dominate the original *iterative* algorithms discussed in this section.

Assume that the buffer pool is sufficiently large to cache one entire path (from root to leaf) of the $B^+$ tree. For simplicity of analysis, I neglect the minor effect of caching beyond

| | |
|---|---|
| $\alpha_k$ | acceptance probability of record $k$ |
| $\alpha$ | $= E(\alpha_k)$ = expected acceptance probability of record |
| $\beta$ | $= (f_{avg}/f_{max})$ = average acceptance prob. at of a node |
| $b_{avg}$ | average blocking factor for variably blocked file |
| $b_{max}$ | maximum blocking factor for variably blocked file |
| $C_{method}(s)$ | cost of retrieving sample of size $s$, via specified method |
| $f_i$ | fan-out from internal node $i$ of $B^+$ tree, or<br>number of records in leaf node $i$ of $B^+$ tree |
| $f_{avg}$ | average fan-out from internal node of $B^+$ tree, or<br>average number of records in leaf node of $B^+$ tree |
| $f_{max}$ | maximum fan-out from internal node of $B^+$ tree, or<br>maximum number of records in leaf node of $B^+$ tree |
| $h$ | height of $B^+$ tree(count root as height 1) |
| $h'$ | height of ranked $B^+$ tree(usually same as $h$ ) |
| $i = 0$ | denotes root node of $B^+$ tree |
| $n$ | number of records in file |
| $p_k$ | probability of inclusion of record $k$ |
| $path_k$ | path (node identifiers) from root to leaf containing record $k$ |
| $\pi$ | expected length of path in early abort algorithm |
| $s$ | number of records desired in sample |
| $s'$ | inflated sample size (to compensate for acceptance/rejection) |
| $thisnode$ | pointer to a $B^+$ tree node (internal or leaf) |
| $Y(k, m)$ | Cardenas's function for expected number of blocks<br>referenced when retrieving $k$ records from $m$ block file |
| $w_k$ | probability of sampling of record $k$ on a simple random walk from root to leaf of $B^+$ tree |

Table 3.1: Notation used in Chapter 3 on Sampling from $B^+$ trees.

| | |
|---|---|
| NI | Naive Iterative algorithm for $B^+$ tree files |
| EAI | Early Abort Iterative algorithm for $B^+$ tree files |
| RI | Iterative algorithm for ranked $B^+$ tree files |
| NB | Naive Batch algorithm for $B^+$ tree files |
| EAB | Early Abort Batch algorithm for $B^+$ tree files |
| RB | Batch algorithm for ranked $B^+$ tree files |

Table 3.2: Algorithm abbreviations

the root page (for these iterative algorithms). This will not alter the relative performance of the various iterative algorithms.

### 3.2.2 The Problem

As with other file structures the problem is to produce uniform inclusion probabilities for the target data records. Simply choosing a random edge from each internal node will not suffice, because nodes reached from internal nodes with low fanout will be more likely to be sampled than those reached from nodes with high fanout.

### 3.2.3 Naive Iterative method

Basically, the naive method consists of performing acceptance/rejection sampling on complete random paths through the tree (from root to leaf). The acceptance/rejection sampling is used to correct the inclusion probability of each sampled path, so that every record (stored in the leaves of the $B^+$ tree) has the same inclusion probability. I discuss this method primarily for expository reasons, since it is dominated by the *early abort method* (described in Sect. 3.2.4).

In this method one selects a random path from the root to a record in a leaf (i.e., at each internal node one chooses a branch at random (equi-probably), (see Figure 3.1) at the leaf one selects a record at random (equi-probably)). Upon reaching the leaf one performs an acceptance/rejection test to decide whether to keep this path. The acceptance probability is calculated as one traverses the path from root to leaf as the product of the ratios of actual fan-out to maximum fan-out at each node (except the root). One is, in effect, sampling from a full multi-way tree, discarding paths which do not actually exist.

Thus in Figure 3.1 I illustrate this algorithm. I show a $B^+$ tree with the internal nodes drawn as ovals. The dots in the internal nodes signify the keys. The solid arrows represent pointers in the internal nodes to children. The root is at the top of the figure. The leaves are drawn as boxes at the bottom of the figure. The annotated dotted lines describe a single iteration of the naive sampling algorithm along a path through the $B^+$ tree, as described above. One chooses a pointer at random at each internal node. Upon reaching a leaf, one performs acceptance/rejection sampling of the records with with acceptance probability given by Equation 3.3.

Denote by $B$ a $B^+$ tree of order $m$ with height $h$ (there exist $h$ nodes on any path from root to leaf, including the root and leaf). Let $f_i$ denote the fan-out of node $i$, and $f_{max}$ denote the maximum fan-out of any node. (For simplicity, I shall assume that the max. no. records/leaf is the same as the maximum fan-out of internal nodes.) This is the number of branches from an internal node, and the number of records in a leaf node. Designate the root node to be node zero.

**Lemma 1** *The naive algorithm generates a simple random sample. The inclusion probability $p_k$ for record $r_k$ contained in leaf $j$ for a single (root-to-leaf) path traversal is:*

$$p_k = {f_0}^{-1} {f_{max}}^{-h+1} \tag{3.1}$$

*where $f_{max} = 2m + 1$ is the maximum fanout of an internal node (and for simplicity also the maximum number of records on a leaf node).*

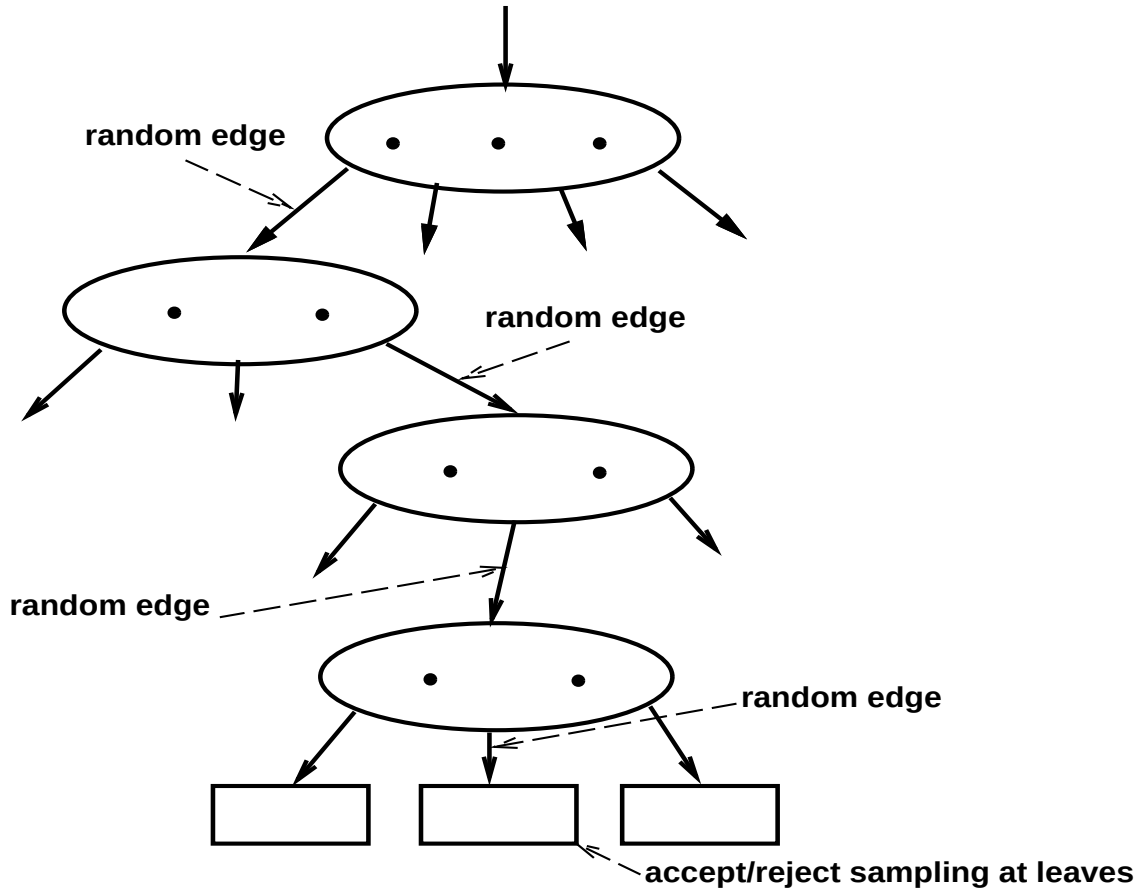



Figure 3.1: Example of Naive Iterative Method of Sampling from $B^+$ tree

```
function ARBTREEP1();

comment This procedure will return a single sample from a a B+ tree.
begin
      temp ;= nil ;
      while temp = nil do
            temp := ARBTREEPROBE();
      endwhile
      return(temp);
end

procedure  ARBTREEPROBE();

comment This procedure will return at most a single sample from a
          a B+ tree.
begin
      p := 1.0; /* initialize acceptance probability */
      thisnode := root;
      while thisnode ≠ leaf do
            if thisnode ≠ root then
              /* update acceptance probability */
              p := p * (f_thisnode / f_max);
            endif
            /* Choose a subtree at random */
            /* RAND() generates a random number between 0 and 1 */
            thisnode := thisnode.nodeptr[RAND() * f_thisnode];
      endwhile
      /* update acceptance probability at leaf*/
      p := p * (f_thisnode / f_max);
      /* do acceptance/rejection sampling */
      if RAND() < p then
        return (one random record from thisnode)
      else return(nil);
end
```

Figure 3.2: Code for naive A/R sampling from $B^+$ tree

**Proof:** Let $w_k = p(\text{sampling record } k \text{ on a random walk})$. Then

$$w_k = \prod_{i \in path_k} f_i^{-1} \tag{3.2}$$

where $path_k$ refers to the path from root to leaf node containing the record $k$. Let the acceptance probability for record $k$ be $\alpha_k$, defined:

$$\alpha_k = \prod_{i \in path_k, i \neq 0} (f_i / f_{max}) \tag{3.3}$$

Note that the product here excludes the root node, because it is common to all paths, hence it does not introduce any non-uniformity into the inclusion probabilities. Recall that all paths in the tree have the same height, $h$. Hence:

$$p_k = \alpha_k w_k = f_0^{-1} \prod_{i \in path_k, i \neq 0} (1 / f_{max}) \tag{3.4}$$

$$p_k = {f_0}^{-1} {f_{max}}^{-h+1} \tag{3.5}$$

□

**Theorem 1** *The expected cost of the naive method for a simple random sample with replacement of size $s$ from a $B^+$ tree is approximately:*

$$E[C_{NI}(s)] \approx (\beta^{-1})^{h-1} s(h-1) + 1 \tag{3.6}$$

**Proof:** Caching is generally only effective for the root page, hence the $(h-1)$ factor, rather than $h$. The last term simply accounts for initially reading the root page. Hence:

$$E[C_{NI}(s)] \approx s'(h-1) + 1 \tag{3.7}$$

where $s'$ is the gross sample size necessary to produce a net sample of size $s$ after acceptance/rejection sampling. By assuming that all the fan-outs are equal to $f_{avg}$ one obtains:

$$E[s] \approx s' \beta^{h-1} \tag{3.8}$$

Inverting this equation I shall assume (ignoring stochastic variation) that the required gross sample size, $s'$ is:

$$s' = (\beta^{-1})^{h-1} s \tag{3.9}$$

Substituting this value of $s'$ into Eqn. 3.7 yields the desired theorem, Eqn. 3.6. □

### 3.2.4 Early abort iterative method

The *early abort method* of sampling from a $B^+$ tree, derives from the *naive* method. Both are based on acceptance/rejection sampling of random paths in the tree. The difference is that the naive method traverses complete paths from root to leaf before deciding on acceptance or rejection while the *early abort method* performs an acceptance/rejection test at each node (except the root).

If the node is rejected, then one can abort searching this path, permitting early abortions of a path. One can in effect reject a leaf node while part way down the path to it, without requiring that one retrieve the entire path. Hence, this method clearly dominates the naive method in expectation.

One way of thinking about the algorithm is to imagine that we are sampling random paths from the full multi-way tree. As soon as one goes down a branch which is not present in the actual (partially full) tree, one aborts that path.

Thus in Figure 3.3 I show an example of the the early abort iterative $B^+$ tree sampling algorithm. The $B^+$ tree with the root at the top. Internal nodes are shown as ovals, with dots signifying the keys, solid arrows representing pointers to child nodes/leaves, and boxes representing leaves. I assume that the maximum permitted fan-out is 5. The annotated dotted arrows describe the course of the algorithm. The algorithm commences by selecting a pointer from the root page at random. Here I show that the leftmost pointer was chosen. This pointer is followed to the second level node. One then performs an acceptance/rejection test with acceptance probability equal to the ratio of the fan-out to the max. fan-out, i.e., 4/5. This is done by generating a Bernoulli random variable with success probability of 0.8. I assume that this succeeds. One then again choose a pointer at random, here the rightmost pointer. One follows this pointer to a third level node. Again one performs an acceptance/rejection test, now with acceptance probability of 3/5. Here I suppose that this acceptance/rejection test has failed, and that the algorithm immediately stop pursuing this path in the tree, resuming at the root of the tree. This process is repeated until enough records are accepted to yield the desired sample size. Note that upon reaching a leaf one would perform an acceptance/rejection test with acceptance probability equal to the the ratio of the number of records in this leaf to max. no. of records in any leaf.

At each node (except the root) along a path from root to leaf one performs an acceptance/rejection test with acceptance probability for node $i$ denoted as $\beta_i$:

$$\beta_i = f_i / f_{max} \tag{3.10}$$

Recall

$$\beta = f_{avg} / f_{max} \tag{3.11}$$

The root node is accepted unconditionally, i.e., $\beta_0 = 1$.

The code for this algorithm is shown in Figure 3.4.

**Lemma 2** *For a single (root-to-leaf) path traversal, the early abort algorithm generates a simple random sample with inclusion probability $p_k$ for record $r_k$, where:*

$$p_k = {f_0}^{-1} {f_{max}}^{-h+1} \tag{3.12}$$

**Proof:** Observe that the probability, $b_k$, of accepting a record $k$ (in a leaf node) is simply the product of the $\beta_i$ along the path:

$$b_k = \prod_{i \in path_k, i \neq 0} \beta_i = \alpha_k \tag{3.13}$$

i.e., the same as for the *naive* algorithm. Hence, my result follows from the proof of the naive algorithm. □

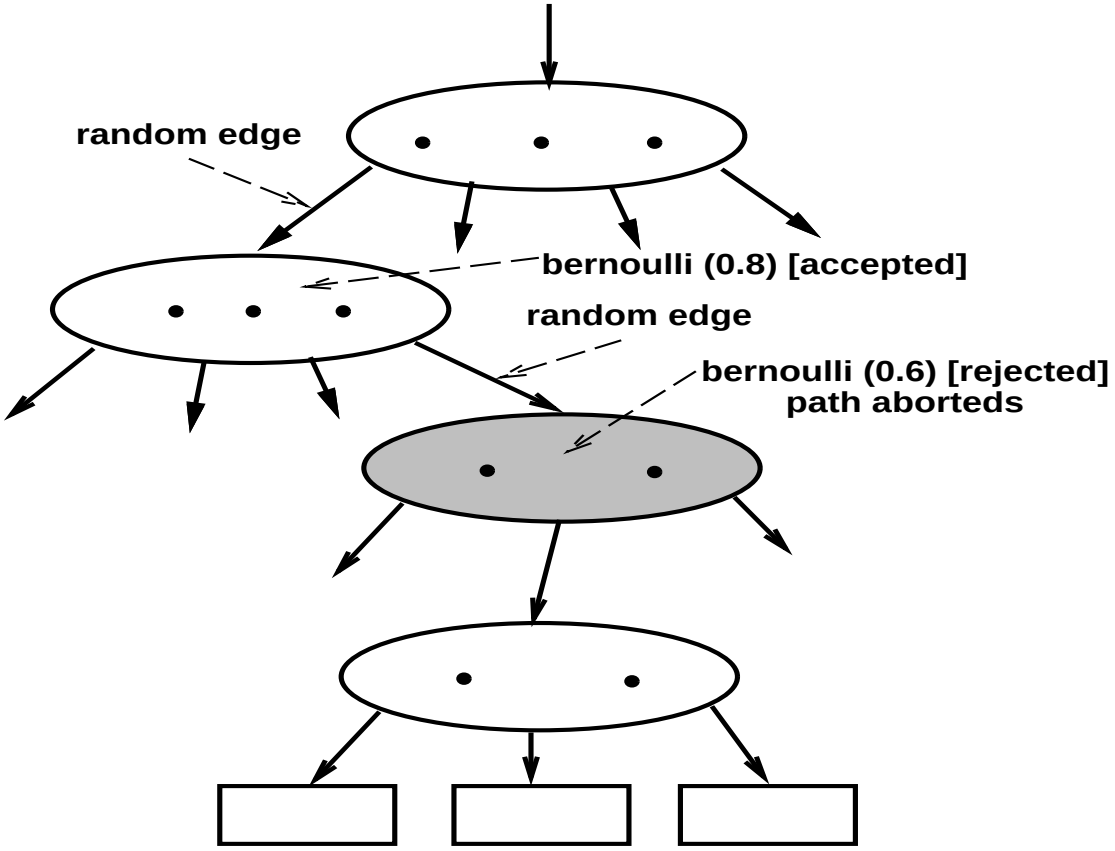



Figure 3.3: Example of Early Abort Iterative Method of Sampling from $B^+$ tree

```
function EARLY1();

comment This procedure will return a single sample from a a B+ tree;
begin
      temp ;= nil ;
      while temp = nil do
            temp := EARLYPROBE();
      endwhile
      return(temp);
end

procedure  EARLYPROBE();

comment This procedure will return at most a single sample from a
a B+ tree
begin
      thisnode := root;
      while thisnode ≠ leaf do
            if thisnode ≠ root then
              /* do acceptance/rejection sampling on internal node*/
              /* RAND() generates a random number between 0 and 1 */
              if RAND() ≥ (f_thisnode/f_max); then
                return(nil)
              endif
            endif
            /* pick a branch at random */
            thisnode := thisnode.nodeptr[RAND() * f_thisnode];
      endwhile
      /* do acceptance/rejection sampling on leaf node*/
      if RAND() < (f_thisnode/f_max) then
        return (one random record from thisnode)
      else return(nil);
end
```

Figure 3.4: Code for early abort A/R sampling from $B^+$ tree

In effect, the early abort algorithm searches the same paths as the naive algorithm, but it aborts the search at the first rejection. Thus for some paths, it accesses fewer pages. Hence its expected cost will be strictly less than that of the naive algorithm, unless all of the pages are full.

**Theorem 2** *The expected cost of the early abort iterative method for a simple random sample with replacement of size s from a $B^+$ tree is approximately:*

$$E[C_{EAI}(s)] \approx (\beta^{-1})^{h-1} s \frac{(\beta^{h-1} - 1)}{\beta - 1} + 1 \tag{3.14}$$

**Proof:** Here I have again assumed that all the fan-outs are equal to $f_{avg}$. As before, the gross sample size must be increased by a factor of $(\beta^{-1})^{h-1}$ to account for the losses due to A/R sampling.

Recall that for the naive algorithm the length of each path examined is $h$ (with a one-path cache it will be approx. $h-1$). What is the expected length of a path searched in the early abort algorithm? Let $\pi_j$ denote the path length for attempt $j$ (assuming the root is cached) and $\pi$ denote its expectation $E[\pi_j]$. I ignore the root (because it is cached). Acceptance/rejection sampling at each node could cause an early abort - but this happens after the node has been read in - so one subtracts one more from the exponent on the expected acceptance probability $\beta = (f_{avg}/f_{max})$ of a node. This gives:

$$\pi = E[\pi_j] = \sum_{i=1}^{i=h-1} P(\pi_j \geq i) = \sum_{i=1}^{i=h-1} \beta^{i-1} \tag{3.15}$$

Summing one obtains:

$$\pi = \frac{(\beta^{h-1} - 1)}{\beta - 1} \tag{3.16}$$

The total cost is given by:

$$E[C_{EAI}(s)] \approx s'\pi + 1 \tag{3.17}$$

where the last term consists of reading the root and where again one obtains:

$$s' = s(\beta^{-1})^{h-1} \tag{3.18}$$

is the adjusted gross sample size to compensate for the attrition due to acceptance/rejection sampling. Substituting the expressions for $s'$ from Eqn. 3.18 and $\pi$ from Eqn. 3.16 and into Eqn. 3.17 yields Eqn. 3.14 of the theorem. □

Observe that from my definition of $\pi$ that it must be less than $h-1$, hence one obtains:

**Theorem 3** *The expected cost of the early abort iterative method is strictly less that the expected cost of the naive iterative method.*

**Proof:** Both methods must explore the same expected number of paths, as noted above the expected path length for early abort is less than that for the naive algorithm. □

### 3.2.5 Sampling from a ranked $B^+$ tree

In this section I discuss how to extract a fixed size simple random sample from a *ranked* $B^+$ tree file. A *ranked* $B^+$ *tree* file is one whose nodes have have been augmented with information which permits one to find the $j'th$ record in the file. I include this method as a benchmark against which to compare the proposed new algorithms.

Suppose that one wishes to sample from a file containing $n$ records. One generates a uniformly distributed random number, $j$, between 1 and $n$, and then sample the $j'th$ record. To do this one must be able to identify the $j'th$ record. If the access method to the file is a $B^+$ tree, then one must be able to find the $j'th$ ranked record in the file. Hence, one must store information in the tree which allows one to calculate the rank of each record.

This idea is discussed in [Knu73]. Similar ideas are used in [BK75] and [WE80]. Essentially one stores in each node of the tree a count (partial sum) of the number of leaves in that subtree. In a binary tree [Knu73] the rank of each leaf can be calculated by suitable sums and differences of the count fields of all the nodes on the path from root to leaf. In a $B^+$ tree one promotes the count fields one level in the tree so that each node stores not only the total count of leaves in its subtree, but also the counts for each child (alongside its key). Hence, while a rank access to the tree must still examine on average half the entries in each $B^+$ tree page, the number of disk pages which must be fetched is only equal to the height of the tree. This matter is discussed in [SL88], and (for tries) in [Gho86].

The use of a ranked $B^+$ tree for sampling is straightforward and can be attributed to Wong and Easton [WE80]. Others have also published on the topic, e.g., Srivastava [SL88] and Ghosh (for tries) [Gho86]. For this algorithm, which I call the ranked iterative algorithm (denoted RI), one simply generates a random number, $j$, between 1 and $n$ (the total number of records in the file) and then access the $B^+$ tree via the rank fields to retrieve the $j'th$ record. If there is no caching of disk pages, then each random probe retrieves a complete path from root to leaf, consisting of $h'$ pages where $h'$ is the height of the ranked $B^+$ tree.

**Theorem 4** *The expected number of disk pages accessed for a simple random sample with replacement of size $s$ from a ranked $B^+$ tree via the ranked iterative algorithm RI is:*

$$E(C_{RI}(s)) \approx s(h' - 1) + 1 \tag{3.19}$$

*where $h'$ is the height of the ranked $B^+$ tree, i.e., $h' = h/(1 + log_{f_{avg}}(2/3))$, assuming that storing the rank field reduces the fan-out of internal nodes by $1/3$.*

**Proof:** If one assumes that each entry in a ranked $B^+$ tree node is comprised of a key, a pointer, and a rank field, each of equal size then the average fan-out of the ranked $B^+$ tree will be $2/3$ the average fan-out of the standard $B^+$ tree. (Also for the maximum fan-out.) Thus one obtains:

$$h \approx \left\lceil \log_{f_{avg}}(n) \right\rceil \tag{3.20}$$

$$h' \approx \left\lceil \log_{(2/3)f_{avg}}(n) \right\rceil \tag{3.21}$$

Hence:

$$h' \approx h/(1 + \log_{f_{avg}}(2/3)) \tag{3.22}$$

In practice, $B^+$ trees are not usually very deep, no more than 4 or 5 levels, hence the lower fan-out of the ranked trees will usually add no more than one level to the height of the tree, often the ranked tree will be the same height as the unranked tree, i.e., $h' = h$.

The first path traversed from root to leaf required $h'$ page I/Os (the height of the ranked $B^+$ tree). The subsequent $s-1$ paths traversed may rereference the same pages (by chance). If we assume that the cache holds one path, then the probability of rereferencing a page at level $l$ of the tree is approximately $f^{-l}$, where $f$ is the average fan-out, since there are approximately $f^l$ nodes (leaves) at level $l$. Hence:

$$E(C_{RI}(s) = h' + (s-1) \sum_{l=0}^{l=h'-1} (f^l - 1)/f^l \tag{3.23}$$

$$E(C_{RI}(s) = h' + (s-1)(h' - \sum_{l=0}^{l=h'-1} 1/f^l) \tag{3.24}$$

$$E(C_{RI}(s) = h' + (s-1)(h' - (f^{-l} - 1)/(f^{-1} - 1)) \tag{3.25}$$

$$E(C_{RI}(s) \approx h' + (s-1)(h' - 1) \tag{3.26}$$

$$E(C_{RI}(s) \approx s(h' - 1) + 1 \tag{3.27}$$

In effect, only the root page is rereferenced in the cache. □

The performance of the various iterative algorithms (ranked, naive, early abort) is summarized in Figure 3.5. Clearly, the ranked $B^+$ tree sampling algorithm outperforms the early abort algorithm, which in turn outperforms the naive algorithm. Note that I have assumed that the ranked $B^+$ tree is the same height as a regular $B^+$ tree.

In practice, the ranked $B^+$ tree will have a height no more than one greater, than the corresponding $B^+$ tree. For typical size B-trees (i.e., height 4) one would therefore expect that the ranked B-tree sampling algorithm will always outperform the acceptance/rejection B-tree algorithms.

However, the ranked B-tree algorithm requires that one makes serious modifications to the B-tree access method code, and that one maintains the rank information in the B-tree at all times.

As discussed earlier, duplicate removal can be performed in $O(s)$ memory. By checking online for duplicate random numbers, before fetching each record, one can obtain a SRS without replacement in the same number of disk accesses.

## 3.3 Batch Sampling from $B^+ - trees$

In this section I consider batch methods of sampling from $B^+$ trees. Such methods are intended to reduce or eliminate the rereading of disk blocks incurred by iterative sampling algorithms (assuming that the buffer pool holds only one entire path through the $B^+$ tree). Batch sampling algorithms process the entire sample as one batch in a manner very similar to batch searching. Batch sampling from ranked $B^+$ trees completely eliminates rereading disk blocks. Batch sampling of regular $B^+$ trees via acceptance/rejection methods returns a sample of random (i.e., binomial) size. Hence, it may occasionally be necessary to repeat

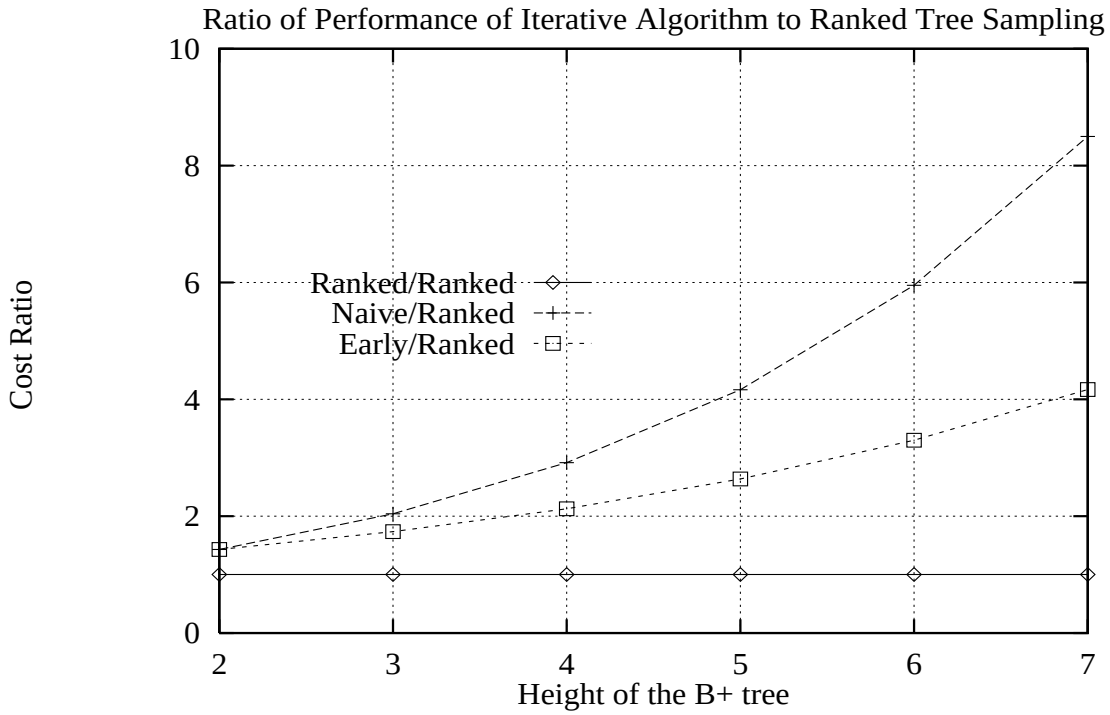



Figure 3.5: Iterative Algorithm Performance Graph

the batch sampling process to obtain a sufficiently large sample. Generally, this can be avoided by using an inflated estimate of the gross sample size required and then randomly discarding excess sample elements.

### 3.3.1 Standard $B^+$ trees

#### 3.3.1.1 Naive Batch

The naive batch sampling algorithm for standard $B^+$ trees simply applies the naive iterative algorithm in parallel. Instead of sampling paths through the tree one at a time, one processes a batch of paths at once. Naive batch sampling from a $B^+$ tree is very similar to batch searching of a $B^+$ tree. The advantage is that one can eliminate the rereading of disk pages which can occur with the iterative algorithms.

This effect is most useful at higher levels of the $B^+$ tree, at the leaves one would typically not expect more that one sampled record per data page. In effect, batch sampling conducts a depth first search of the $B^+$ tree to find the sampled records, pursuing the same paths through the tree as the naive $B^+$ tree sampling algorithm, but reordering the manner of examining the paths to avoid rereading pages.

Suppose that one has an estimate $s'$ of the required gross sample size needed to produce a net sample of size $s$. One starts at the root with a gross sample size of $s'$ and proceed with a depth first search of the tree.

At each internal node of the tree one allocates the incoming portion of the sample to the various subtrees by generating a equiprobable multinomial random vector. A equiprobable multinomial random vector from a population of size $s$ in $k$ cells is a vector $V = (v_1, v_2, ..., v_k)$ of length $k$, which records the number of balls $v_i$ which fall in cell $i$, when $s$ balls are thrown in the the cells at random (equiprobably). (More generally the probability of balls falling into each cell could vary, but one does not need this.) The uniform multinomial vector can be generated in two ways: by generating a random branch for each of $s$ balls and incrementing the corresponding cell count, or alternatively by generating a Poisson random variable for each cell, and then adjusting the resulting variable with the first method, so that their sum is correct [BB84].

Only those branches with nonzero sample sizes allocated to them are pursued. Note that I am not doing any acceptance/rejection sampling at the internal nodes, only distributing the gross sample among the children.

Upon reaching a leaf node one performs acceptance/rejection sampling on the portion of the gross sample allocated to this leaf to obtain the net sample size from this leaf. This one does by generating a binomial random variable, $x_k \sim B(s_k, \alpha_k)$ with parameters, $s_k =$ gross sample size allocated to this leaf, and $\alpha_k =$acceptance probability for this path $k$ (as in naive iterative algorithm).

Having determined the net sample size for a particular leaf, one extracts a simple random sample with replacement of this size from the records on the leaf (this is trivial).

One then continues with the depth first search of the tree until complete. The resulting sample may be the wrong size (because of acceptance/rejection). If the resulting sample is too large, one discards the excess (chosen at random). If the resulting sample is too small, one repeats the process (adding to the sample) until one has enough. For many purposes, the exact sample size may not matter and these corrections would be unnecessary. However,

throughout the thesis I have assumed here that the sample size is specified exogenously in the query, e.g., by a statistician, to meet requirements for reliability of hypothesis testing or estimation accuracy. If the sample is to be used for physical inspection, e.g., in auditing, quality control, or epidemiology, then excessive sample size can be expensive.

I show an example of the execution of the naive batch algorithm in Figure 3.6. Again I show a $B^+$ tree with the root at the top, internal nodes shown as ovals, keys as dots, pointers as solid arrows, and leaves as boxes. In this example one begins at the root with a gross sample size of 100. The gross sample is allocated among the 4 subtrees emanating from the root by computing a uniform multinomial random vector (of length 4). Here I show the vector as (18,33,20,29), i.e., the numbers next to the solid arrows (representing pointers). One continues, as if performing a depth first search. Thus the leftmost node at the second level has been allocated 18 elements of the gross sample size. These are apportioned according to a uniform multinomial random vector of length 3 (because there are only 3 pointers emanating from this node). The multinomial random vector here is (7,9,2). At the third level I have shown a node with a gross sample size of 2, which is allocated among the child nodes by a uniform multinomial random vector of length 3, viz. (0,2,0). At the fourth level I have shown a node with a gross sample size of 2, which is allocated among the 3 leaves by a uniform multinomial random vector of length 3, viz. (0,1,1). At each leaf, one generates a binomial random variable, $x_i \sim B(n_i, p_i)$, where $n_i$ is the gross sample size allocated to the leaf and $p_i$ is the acceptance probability, i.e., $b_i/b_{max}$, the ratio of the number of records in leaf $i$, to the max. number of records in any leaf.

Code for the naive batch sampling algorithm is shown in Figure 3.7.

**Lemma 3 Cardenas's Lemma** *The expected number of disk blocks referenced, $d$, when sampling $k$ records (with replacement) from a file of $m$ equal size blocks is given by:*

$$d = m(1 - (1 - (1/m)^k)) \tag{3.28}$$

*Denote this function of $m$ and $k$ as $Y(k, m)$.*

**Proof:** See [Yao77]. □

For the naive batch method, the effort to retrieve a gross sample of size $s'$, is the same as the effort to perform batch searching on a $B^+$ tree, with the same batch size.

**Theorem 5** *The expected cost in I/O to retrieve a sample of size $s$ via the naive batch sampling method is approximately:*

$$E(C_{NB}(s)) \approx 1 + \sum_{j=1}^{j=h-1} Y(s', f_0 f^{j-1}) \tag{3.29}$$

*where $f = f_{avg}$ and and $s' = s(\beta^{-1})^{h-1}$ (inflated gross sample size) as before.*

**Proof:** As before, one uses a gross sample size of $s'$ to compensate for the losses due to acceptance rejection. This theorem is a derived by applying Cardenas's Lemma (Lemma 3) for each level of the tree. It is assumed that all nodes have fan-out $f = f_{avg}$, except the root which has fan-out $f_0$.

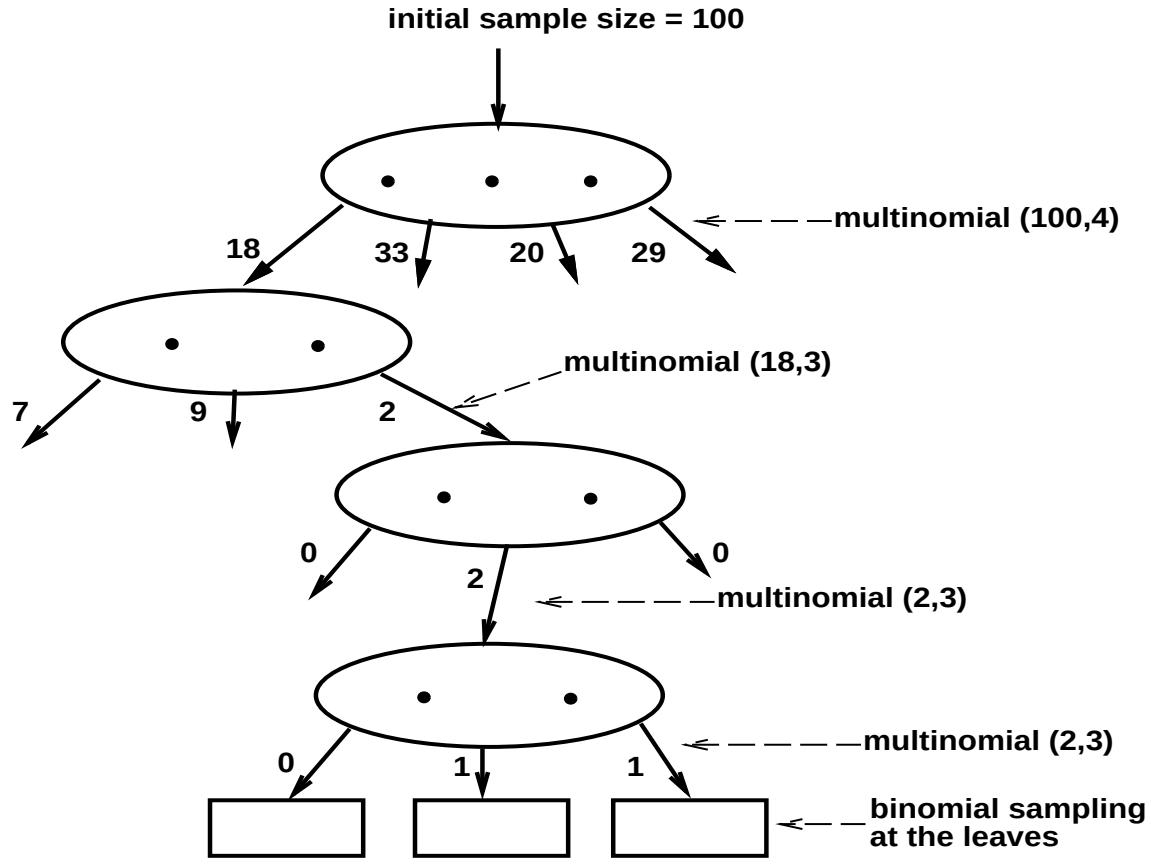



Figure 3.6: Example of Naive Batch Method of Sampling from $B^+$ tree

```
function BATCHBTREE(s , p, thisnode);
        /* This procedure will return a simple random sample */
        /* with replacement of size at most s from a B+ tree*/
        /* p is the acceptance probability (initially set to 1.0)*/
        /* thisnode points to the (sub)tree */
declare integer x(f_max) ;
        /* x is a vector to hold a multinomial sample */
        /* f_max is the max. fan-out per node */
begin
    k := s
    if thisnode = root then
      sample := nil /* initialize sample */
      p := 1.0      /* initialize acceptance probability */
    else /* this is not the root */
        /* so adjust acceptance probability */
        p := p * (f_thisnode / f_max);
    endif
    if thisnode ≠ leaf then
        /* generate a multinomial sample of size k from f_thisnode bins */
        x := MULTINOMIAL (k, f_thisnode)
        /* Recursively sample from each branch, according to the */
        /* multinomial sample */
        for i = 1 to f_thisnode do
            if x_i ≠ 0 then
                    Append (sample, BATCHBTREE(x_i, p, thisnode.nodeptr[i]));
            endif
        endfor
    else /* this is a leaf */
        k := BINOMIAL (k, p) ; /* do acceptance rejection sampling */
        if k > 0 then
                /* obtain a simple random sample with replacement of size k */
                /* from this page, append it to the sample being created */
                Append (sample, SRSWRONPAGE (k, thisnode));
        endif
    endif
    return(sample);
end
```

Figure 3.7: Code for naive batch A/R sampling from $B^+$ tree

```
function BATCHBTREEX(s , p, thisnode);
        /* This procedure will return a simple random sample */
        /* with replacement of size at exactly s from a B+ tree*/
        /* It simply calls BATCHBTREE until it gets a large enough sample */
        /* h is the height of the B+ tree */
        /* s is the sample size */
        /* p is the acceptance probability (initially set to 1.0) */
        /* thisnode points to the (sub)tree */
sample := nil ;
j := k ;
while j > 0 do
      Append (sample, BATCHBTREE(inflate(j, h), p, rootnode));
      j := k - count(sample);
endwhile
if count(sample) > s then
  /* delete excess elements from sample */
  delete (sample, count(sample) - s );
  endif
  return(sample); end;

function inflate (s, h);
        /* This function returns an inflated gross sample size */
        /* to compensate for the effects of the acceptance/rejection */
        inflate := fudgefactor * (f_max/f_avg)^(h-1) ; end;
```

Figure 3.8: Code for naive exact batch A/R sampling from $B^+$ tree

Let us number the levels 0, 1, 2, ..., h-1 from root to leaf, and let $j$ denote the level number.

For level 0, one has $s'$ records, 1 block (the root). For level 1, one still has $s'$ records, and $f_0$ blocks. For level 2, one has $s'$ records, and $f_0 f$ blocks. Thus for level $j$, $j > 1$, one has $s'$ records and $f_0 f^{j-1}$ blocks, hence by Cardenas's Lemma the number of blocks accessed at level $j$ will be $Y(s', f_0 f^{j-1})$. Summing yields the theorem. □

#### 3.3.1.2 Early Abort Batch Method

The early abort batch method is simply the batch analog of the early abort iterative method. It is similar to the naive batch method, differing only in that the acceptance/rejection sampling is performed (by computing binomial samples) at each node (except the root) as one searches from root to leaf. Recall that the naive batch algorithm only does an acceptance/rejection test at the leaf. As was the case with the iterative methods, the early abort batch method dominates the naive batch method, because it avoids ever having to reread any disk pages.

Thus one commences a depth first search at the root with a gross sample size of $s'$. At the root, and each subsequent internal node, one allocates the incoming sample to the various branches by means of a multinomial random vector. Only those branches with nonzero sample sizes allocated to them are pursued.

At each level beyond the root, one performs acceptance/rejection sampling of the incoming sample by generating a binomial random variable, $x_i \sim B(s_i, \beta)$ with parameters $s_i$, (the incoming sample size), and acceptance probability $\beta_i$ (as in the iterative early abort algorithm). The resulting net sample size for the node is then allocated the branches via a multinomial random vector. Only those branches with nonzero sample sizes allocated to them are pursued.

I show an example of the execution of the naive batch algorithm in Figure 3.9. Again I have shown a $B^+$ tree with the root at the top, internal nodes shown as ovals, keys as dots, pointers as solid arrows, and leaves as boxes. In this example one begins at the root with a gross sample size of 100. The gross sample is allocated among the 4 subtrees emanating from the root by computing a uniform multinomial random vector (of length 4). Here I show the vector as (18,33,20,29), i.e., the numbers next to the solid arrows (representing pointers). One continues, as if performing a depth first search. Thus the leftmost node at the second level has been allocated 18 elements of the gross sample size. Because this node has only 3 pointers emanating from it (vs. a maximum of 5), one computes the accepted gross sample size for this node as binomial random variable distributed as $B(5, 3/5)$, here assumed to be zero. At this point the one abandons this search path, backtracks to the most recent unexplored pointer with a nonzero gross sample size allocated and continues.

At each leaf, one generates a binomial random variable, $x_i \sim B(n_i, p_i)$, where $n_i$ is the gross sample size allocated to the leaf and $p_i$ is the acceptance probability, i.e., $b_i/b_{max}$, the ratio of the number of records in leaf $i$, to the max. number of records in any leaf. One then extracts a simple random sample with replacement of size $x_i$ of the records on the page.

Inadequate net sample sizes are dealt as described above for the naive batch sampling algorithm. Code is shown in Figure 3.10.

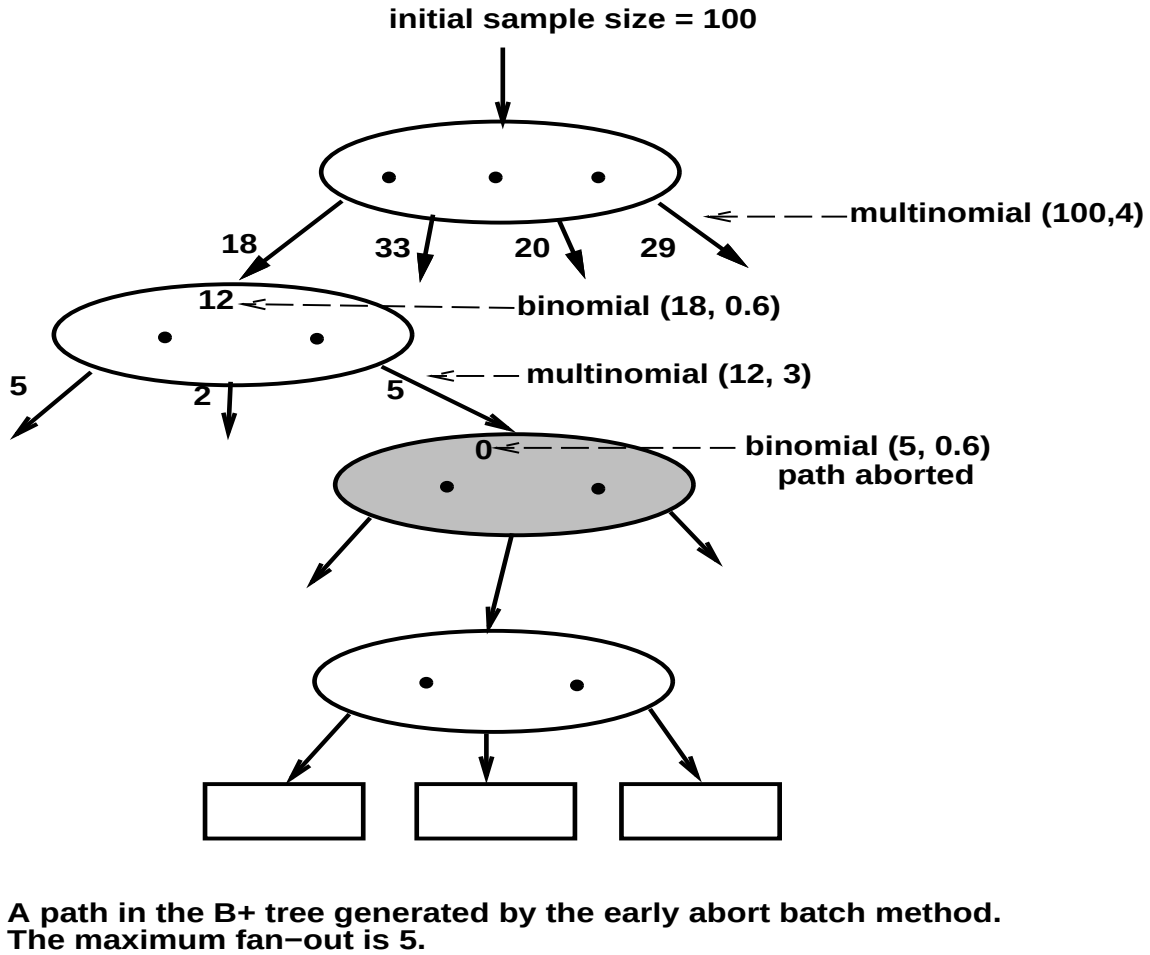



Figure 3.9: Example of Early Abort Batch Method of Sampling from $B^+$ tree

```
function EBATCHBTREE(k , thisnode);
        /* This procedure will return a simple random sample */
        /* with replacement of size at most k from a B+ tree*/
        /* k is the sample size */
        /* thisnode points to the (sub)tree */
declare integer x(f_max) ;
        /* x is a vector to hold a multinomial sample */
        /* f_max is the max. fan-out per node */
begin
    if thisnode = root then
      sample := nil /* initialize sample */
      else /* this is not the root */
           /* so compute new sample size, in effect doing batch */
           /* acceptance/rejection sampling */
           p := (f_thisnode/f_max);
           k := BINOMIAL (k, p) ;
      endif
      if thisnode ≠ leaf then
        /* generate a multinomial sample of size k from f_thisnode bins */
        x := MULTINOMIAL (k, f_thisnode)
        /* Recursively sample from each branch, according to the */
        /* multinomial sample */
        for i = 1 to f_thisnode do
            if x_i ≠ 0 then
                    Append (sample, BATCHBTREE(x_i,thisnode.nodeptr[i]));
            endif
        endfor
        else /* this is a leaf */
             if k > 0 then
                    /* obtain a simple random sample with replacement of size k */
                    /* from this page, append it to the sample being created */
                    Append (sample, SRSWRONPAGE (k, thisnode));
             endif
        endif
        return(sample);
      end
```

Figure 3.10: Code for early abort batch A/R sampling from $B^+$ tree

```
function EBATCHBTREEX(s , p, thisnode);
        /* This procedure will return a simple random sample */
        /* with replacement of size at exactly s from a B+ tree*/
        /* It simply calls EBATCHBTREE until it gets a large enough sample */
        /* h is the height of the B+ tree */
        /* s is the sample size */
        /* p is the acceptance probability (initially set to 1.0) */
        /* thisnode points to the (sub)tree */
sample := nil ;
j := k ;
while j > 0 do
      Append (sample, EBATCHBTREE(inflate(j, h), p, rootnode));
      j := k - count(sample);
endwhile
if count(sample) > s then
  /* delete excess elements from sample */
  delete (sample, count(sample) - s );
  endif
  return(sample);
end;

function inflate (s, h);
        /* This function returns an inflated gross sample size */
        /* to compensate for the effects of the acceptance/rejection */
        inflate := fudgefactor * (f_max/f_avg)^(h-1) ;
        return(inflate);
end;
```

Figure 3.11: Code for exact early abort batch A/R sampling from $B^+$ tree

For the early abort batch method, the analysis is more complicated. The search paths which are aborted early do not generate as many page accesses. If one numbers the levels from 0 to $h$, denoting by *level i* the node at distance $i$ from the root, then at level $i$, $i > 0$, one will access $k_i$ records from ${f_0 f_{avg}}^{i-1}$ pages, where $k_i$ is approximately a binomial random variable with expectation $s' \cdot \beta^{i-1}$, where I have ignored the variation in $f_i$, replacing it with $f_{avg}$.

**Theorem 6** *The expected I/O cost to retrieve a simple random sample of size $s'$ via the early abort batch sampling method is approximately:*

$$E(C_{EAB}(s)) \approx 1 + \sum_{j=1}^{j=h-1} Y(s'\beta^{j-1}, f_0 f^{j-1}) \tag{3.30}$$

*where $f = f_{avg}$ and $s' = s(\beta^{-1})^{h-1}$ (inflated gross sample size) as before, and Y(k,m) is Cardenas's function defined above.*

**Proof:** As before, one uses a gross sample size of $s'$ to compensate for the losses due to acceptance/rejection. This theorem is derived by applying Cardenas's Lemma (Lemma 3) for each level of the tree. It is assumed that all nodes have fan-out $f = f_{avg}$, except the root which has fan-out $f_0$.

Let us again number the levels 0, 1, 2, ..., h-1 from root to leaf, and let $j$ denote the level number.

For level 0, one has $s'$ records, 1 block (the root). For level 1, one still has $s'$ records, and $f_0$ blocks. For level 2, one has $s'\beta$ records (because one has done acceptance/rejection at level 1), and $f_0 f$ blocks. Thus for level $j$, $j > 1$, one has $s'\beta^{j-1}$ records and $f_0 f^{j-1}$ blocks, hence by Cardenas's Lemma the number of blocks accessed at level $j$ will be $Y(s'\beta^{j-1}, f_0 f^{j-1})$. Summing yields the theorem. □

### 3.3.2 Ranked $B^+$ trees

Finally, consider batch sampling of ranked $B^+$ trees. One simply generates a simple random sample of the ranks, sort it, and then perform a batch search of the ranked $B^+$ tree file. Alternatively one could use Vitter's algorithm [Vit84] to generate the sequential skips required to determine the ranks of the sampled records. (Note that Vitter's algorithm generates a simple random sample without replacement, rather than with replacement. This is easily corrected.) I am only concerned here with I/O costs and I assume that the sample of ranks can easily fit in memory.

Assume that one has a cache large enough to hold a complete path through the tree from root to leaf, so that reexamining pages along this path required to retrieve the sample is costless (in terms of disk I/O). Then batch sampling is equivalent to batch searching of a $B^+$ tree, a classic problem treated by [Pal85] (among others). Essentially, the number of pages to be retrieved is simply the number of distinct pages in the union of all paths to sampled pages.

The only difficulty is that storing the rank information reduces the fan-out of the internal nodes, thereby increasing the search path length. Assume that typical nodes in a standard $B^+$ tree contain pairs of (key, pointer), whereas ranked $B^+$ trees contain triples

(key, pointer, partial rank sum). If one assumes that each key, pointer and rank occupy the same space, then the maximum fan-out of a ranked $B^+$ tree will be approximately 2/3 that of a normal $B^+$ tree with the same blocksize. As noted in the discussion of the ranked iterative algorithm, this will typically make no difference in the height of the $B^+$ tree.

**Theorem 7** *The expected I/O cost to retrieve a simple random sample of size s from a $B^+$ tree via the ranked batch sampling method is approximately:*

$$E(C_{NB}(s)) \approx 1 + \sum_{j=1}^{j=h-1} Y(s, f_0 f^{j-1}) \tag{3.31}$$

*where $f = f_{avg}$, $f_0$ is the fan-out of the root, and $Y(k,m)$ is Cardenas's function defined above.*

**Proof:** The proof is identical to that of Theorem 5, except that one does not need to inflate the sample size since one is not doing acceptance/rejection sampling. □

### 3.3.3 Comparisons

It is clear that the batch algorithms will dominate the respective iterative algorithms, because they avoid rereading disk blocks which are used in more than one sample path. The extent of the saving will depend on the proportion of the $B^+$ tree which is read. However, even if there is no saving in terms of reading leaves, one would expect significant savings at upper levels of the $B^+$ tree (close to the root).

I have the following result on early abort batch vs. naive batch algorithms:

**Theorem 8** *The early abort batch outperforms naive batch.*

$$E[C_{EAB}(s)] \leq E[C_{NB}(s)] \tag{3.32}$$

**Proof:** This result follows from the cost functions and the fact that Cardenas's functions $Y(k,m)$ is monotone increasing in $k$ (the sample size). □

The relationship between the performance of the ranked batch algorithm and the early abort batch algorithm is more subtle. Because the ranked batch algorithm may increase the height of the $B^+$ tree it is possible that RB will perform worse than EAB. However, more typically RB will not increase the height of the $B^+$ tree and will out-perform EAB.

How big is the difference between $E[C_{RB}(s)]$ and $E[C_{EAB}(s)]$? It is clearly less than a factor of $(\beta^{-1})^{h-1}$, the factor by which one inflates the gross sample size of EAB to compensate for the attrition due to acceptance/rejection sampling. For random $B^+$ trees $\beta$ is known to be 0.7. Hence for a random $B^+$ tree of height 5, the methods differ by a factor of no more than 4.

## 3.4 Conclusions

The most important results of this chapter concern algorithms to retrieve simple random samples of $B^+$ trees, without any additional data structures or indices. The new methods are based on acceptance/rejection sampling, and provide a simple, inexpensive way to add

sampling to a relational database systems. They are appropriate for systems which only infrequently need to support sampling, e.g., for auditing.

I began by considering iterative acceptance/rejection algorithms. It is clear that the early abort iterative algorithm is always preferred to the naive algorithm. For realistic $B^+$ trees, the ranked tree iterative sampling algorithm will always outperform the early abort iterative acceptance/rejection sampling algorithm. However, ranked $B^+$ trees require additional maintenance during insertions/deletions. Hence, they may be unacceptable in heavy update traffic environments.

I then considered batch sampling algorithms, the sampling analogs of batch search algorithms.. The batch early abort algorithm again outperforms naive batch sampling. As in the case of batch searching, batch sampling is more efficient than iterative sampling. The story for sampling ranked trees is much the same as before, i.e., ranked sampling is faster but requires more update maintenance.

I published a preliminary version of this chapter in 1986 [OR86]. Subsequently, Antoshenkov [Ant92], described an improved sampling algorithm for compressed $B^+$ trees, which is a hybrid of the acceptance/rejection and ranked $B^+$ tree algorithms described in this chapter. Compressed $B^+$ trees use front-compression on the keys (and variable size records in the leaves). The larger variability in fan-outs causes the the acceptance/rejection algorithms described in this chapter to become much less efficient.

Antoshenkov maintains bounded approximations of rank statistics, which he calls pseudo-ranks. By allowing some slop in the counts stored in each node, Antoshenkov avoids the need to propagate changes in the counts all the way up the tree on each modification to the tree (insert/delete). He is thus able to dramatically reduce the cost of updating the rank statistics. Note that although the root page (and perhaps the next level) of the tree is cached, modifications would still require writing the page(s) to disk. He then uses acceptance/rejection sampling to correct for inaccuracies in the approximate rank statistics. By carefully controlling the allowable errors in the rank statistics he can make the acceptance/rejection sampling quite efficient. For compressed $B^+$ trees Antoshenkov's algorithm is clearly preferable.

Finally, it may be the case that one wishes to apply a selection predicate to the records sampled from the $B^+$ tree. Hence, it may be very difficult to accurately estimate the required gross sample size needed for the batch sampling algorithms. In such cases, a two stage sampling algorithm may be desirable. The first stage of which is used to estimate the predicate selectivity needed to compute the gross sample size for a second batch sampling operation.

# Chapter 4

# Sampling from Hash Files

## 4.1 Introduction

In this chapter I continue my discussion of sampling from base relations, examining techniques for simple random sampling from hash files on secondary storage. If $B^+$ trees are the most popular access method used in DBMSs, then hash files are surely the second most popular. I consider both iterative and batch sampling algorithms from static and dynamic hash files.

The main contribution of this chapter 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. I provide detailed cost formulae, supporting simulations, and I show the relationship of sampling costs to the cost of searching the same data structures.

I 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. I consider one method from each class: Linear Hashing by Litwin [Lit80] (no directory) and Extendible Hashing by Fagin [FNPS79] (directory).

For each hash file I consider iterative methods, which repeatedly extract a sample of size one until they accumulate a sample of the requisite size. I 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. I also discuss the use of sequential scan sampling methods.

For both of the dynamic hashing methods I 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. I show that the more sophisticated methods have better performance.

### 4.1.1 Organization of Chapter

The remainder of the chapter is organized as follows. In Section 4.2 I discuss sampling from open addressing hash files, In Section 4.3 I treat sampling from separately chained overflow hash files, In Section 4.4 I examine sampling from Linear Hash [Lit80] files. In Section 4.5 I present sampling from Extendible Hash [FNPS79] files. Batch and sequential scan sampling methods are discussed in Section 4.6. I present experimental results (simulations) in Section 4.7. Finally, Section 4.8 contains conclusions.

The notation used in this chapter is explained in Table 4.1. Abbreviations for algorithms are given in Table 4.2.

### 4.1.2 Notation

Throughout the chapter, for a variable $x$, I will use $\bar{x}$ to denote the average of $x$ and $x_{max}$ to denote the maximum of $x$, for any quantity $x$. Other notation used in this chapter is summarized in Table 4.1. Individual notations are explained in the text where they are first used. Here I mention some of the notational conventions. The desired sample size is denoted as $s$, whereas $s'$ denotes the inflated sample size (i.e., to compensate for losses due to acceptance/rejection sampling). The number of records in the file is denoted $n$. The number of hash buckets in the file is $m$. I denote the fan-out of an $B^+$ tree node as $f_i$, the average fan-out as $f_{avg}$, and maximum fan-out as $f_{max}$. The occupancy (in records) of hash bucket $i$ is denoted $b_i$, the average will be $\bar{b}$, and the (actual) maximum will be $b_{max}$. The maximum possible bucket occupancy will be denoted as $b_\Omega$. Overflow chain length for hash bucket $i$ is denoted as $h_i$. Acceptance probabilities of records are denoted $\alpha_k$, and the expectation as $\alpha$. The expected cost (in disk page accesses) of retrieving a sample of size $s$ by a method $M$ is denoted as $C_M(s)$. For extendible hash files, $c_i$ will denote the occupancy (in records) of $i'th$ directory cell, while $d_i$ will denote the occupancy (in records) of the $i'th$ data page. For batch sampling, one will need to know the expected number of blocks referenced when retrieving $k$ records (at random) from a file containing $m$ blocks, this will be denoted $Y(k, m)$ (Cardenas's function).

Abbreviations used to refer to the various types of hash files in this chapter are summarized in Table 4.2. Algorithms are then named by appending either an "I" or a "B" suffix. I use the suffix "I" to indicate iterative algorithms, which select sample one record at a time. Such algorithms naturally return simple random samples with replacement (duplicates) (SRSWR) but can be modified to return simple random sample random samples without replacement by removing duplicates. I use the suffix "B" to denote batch algorithms, which are the sampling analog of batch search algorithms in trees. Such algorithms naturally return simple random samples without replacement (no duplicateds, denoted SRSWOR), but can be modified to return simple random samples without replacement by synthetically generating duplicates.

## 4.2 Open Addressing Hash Files

In this section I 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. Database

| | |
|---|---|
| $\alpha_i$ | acceptance probability of record $i$ |
| $\alpha$ | $= E(\alpha_k)$ = expected acceptance probability of record |
| $b_i$ | bucket occupancy for bucket $i$ |
| $b$ | $= n/m$ = average hash bucket occupancy (records) |
| $b_{max}$ | max. hash bucket occupancy (records) |
| $b_\Omega$ | max. hash bucket capacity |
| $c_i$ | occupancy (records) of $i'th$ directory cell (Extendible Hash Files) |
| $d_i$ | occupancy (records) of $i'th$ data page (Extendible Hash Files) |
| $h_i$ | chain length for bucket $i$ (pages) |
| $\overline{h}$ | average chain length (pages) |
| $h_{max}$ | maximum chain length (pages) |
| $C_M(s)$ | expected cost of retrieving sample of size $s$, via method $M$ |
| $m$ | 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 4.1: Notation used in Chapter 4 on Sampling from Hash Files.

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

Table 4.2: Hash File abbreviations

management systems do not typically use this method of hashing. I include this hash method, because it permits me to explain the basic ideas of sampling from hash files in a simple setting. Subsequently, I discuss more complex (and realistic) hash access methods.

This algorithm for sampling from hash (or any variably blocked file) was described in [OR90] and [ORX90] (where it was applied to hash files). DeWitt et al. [DNSS92] subsequently adapted this sampling method for use percentile estimation in variably blocked files and called it *extent map sampling*. In DeWitt et al.'s paper the file structure is represented via an extent list (i.e., a list of contiguous regions on disk allocated to consecutive logical disk blocks, as used in DEC/VMS operating system), so that the $i'th$ block can be located by an in-memory search of the extent list. DeWitt et al. used a form of cluster sampling, using all of the tuples contained on each disk page read. Otherwise, the algorithms are the same.

I discuss iterative methods, which repeatedly extract a sample of size one until they have accumulated sufficient size sample. Batch methods, which are based on batch retrieval, are discussed later, in Section 4.6.

For this analysis I adopt the *uniform hashing* model [Knu73, pp. 527-528], which assumes that the hashing functions randomize the placement of records in buckets. This is a crucial, but conventional, assumption of the analyses throughout this chapter.

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.

### 4.2.1 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. denote by $b_{max}$ the maximum of the $b_i$'s. In Figure 4.1 I 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 algorithm uniformly chooses a block (bucket, or disk page), $i$, at random, and then does acceptance/rejection test on the block with acceptance probability, $\beta_i$, proportional to the number of records in the bucket:

$$\beta_i = b_i / b_{max} \tag{4.1}$$

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

**Lemma 4** *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

```
procedure ARHASH ;
        /* This procedure samples one record from a variably blocked file, */
        /* such as an open addressed hash file. */
        /* */
        /* It uses acceptance/rejection sampling to compensate for */
        /* variable bucket (page) loading, so as to assure uniform */
        /* inclusion probabilities for each record. */

comment accepted is a boolean variable which indicates when a sample was accepted.
Set accepted to false;
while accepted=false do
    /* generate a random bucket (page) no. between 1 and m */
    r := RAND(1, m);
    /* generate a random record no. (in the bucket) between 1 and b_max */
    j := RAND(1, b_max);
    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 4.1: Algorithm for sampling from a hash file (variably blocked)

probability $p$ where:

$$p = \frac{1}{m}\frac{1}{b_{max}} \tag{4.2}$$

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

$$P = \frac{n}{m \times b_{max}} \tag{4.3}$$

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 one gets

$$p\frac{1}{1-Q} = \frac{p}{P} = \frac{1}{n} \tag{4.4}$$

as required. □

**Lemma 5** *The expected number of disk accesses to obtain one random sample is*

$$E[C_{ARHASH}(1)] = \frac{m \times b_{max}}{n} = b_{max}/\bar{b} \tag{4.5}$$

**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 $\frac{1}{P} = \frac{m \times b_{max}}{n}$ as required. □

For heavily loaded hash tables $b_{max}$ is effectively bounded by the page capacity, $b_\Omega$. Hence, the efficiency of the sampling method will be inversely proportional to the storage utilization (occupancy ratio), $\bar{b}/b_\Omega$.

## 4.3 Separate Overflow Chain Hash Files

In this section I consider sampling from hash files which have separate overflow chains for each bucket in the primary file area [Knu73, pp. 535]. I 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.

### 4.3.1 Iterative Algorithm

The iterative algorithm selects a bucket at random, then does acceptance/rejection test with acceptance probability, $\alpha_i = b_i/b_{max}$ as in the open addressing hash file. If the bucket is accepted one must then sample one record from the bucket, this may require reading some of the overflow pages. One repeats this until one obtains the desired sample size.

Let $d_{max}$ = the maximum number of records per page and $b_{max}$ = the maximum number of records per hash bucket.

**Theorem 9** *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 $C(H,1)$:*

$$C(H,1) = (\frac{b_{max}}{\overline{b}} - 1) + S(H,1) \tag{4.6}$$

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

Once one has 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:*

$$C_{SOI}(H,1) = (\frac{b_{max}}{\overline{b}} - 1) + S_{SOI}(H,1) \tag{4.7}$$

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

$$S_{SOI}(H,1) = 1 + (1/\overline{b}) \sum_{k=1}^{\infty} k \sum_{j=1}^{d_{max}} \frac{(k-1)d_{max}}{2} + jP(kd_{max} + j, \overline{b}) \tag{4.8}$$

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

$$P(i,\lambda) = \frac{e^{-\lambda}\lambda^i}{i!} \tag{4.9}$$

**Proof:** Result follows from Theorem 9 and from the derivation of $S(H,1)$ given by [Lar82]. Note that this notation differs from Larson's. □

## 4.4 Linear Hashing

In this section I consider sampling from Linear Hash files [Lit80]. I 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 one gradually splits 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 I 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

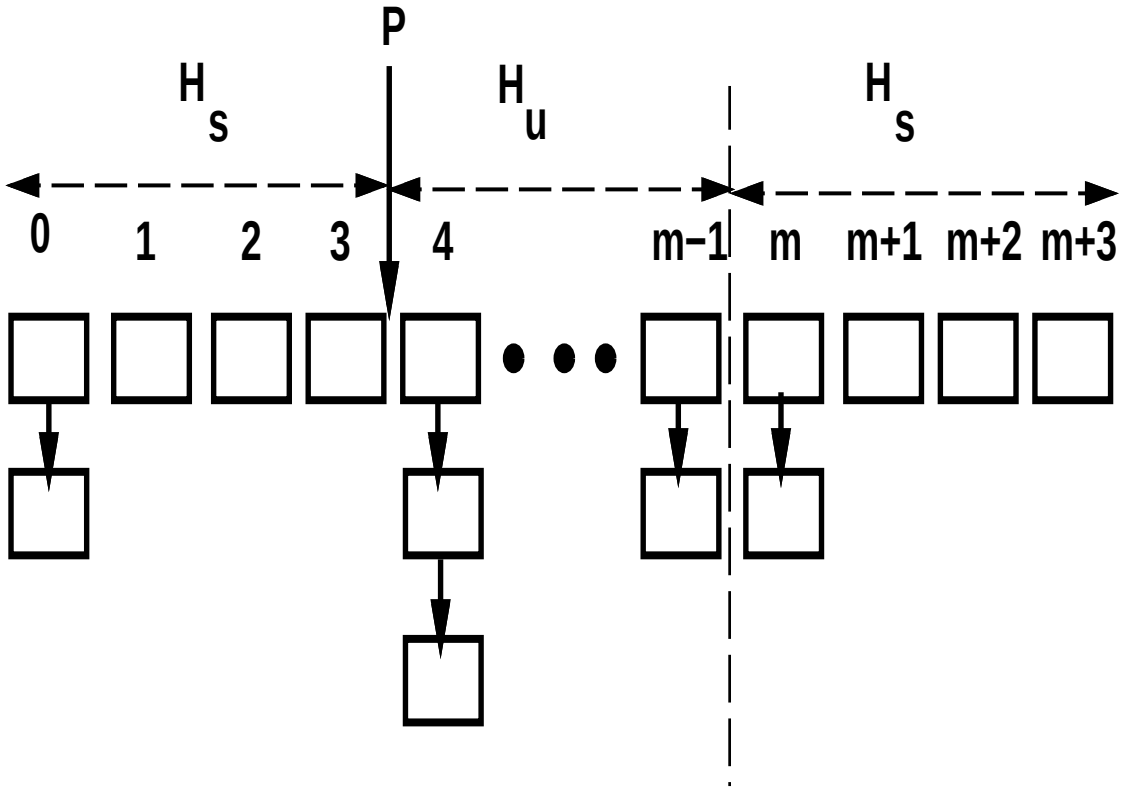



Figure 4.2: Drawing of Linear Hash File

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 4.2.

One has two ways of sampling from the linear hash file. In the 1-file method one treats

the entire hash file as a single variably blocked file. In the 2-file method one samples from the two subfiles, $H_s$ and $H_u$ separately, taking advantage of their different structure.

### 4.4.1 One-file Method

The 1-file method is essentially ARHASH algorithm for variably blocked files applied to hash buckets. Upon accepting a bucket, one 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 one maintains $b_{max}$, the maximum bucket occupancy.

**Theorem 10** *The expected cost of iterative sampling a sample of size 1 from a Linear Hash file using the one-file method is:*

$$\begin{aligned} C_{LHI_1}(H,1) &= (\frac{b_{max}}{\overline{b}} - 1) + (\frac{n_u}{n})S_{SOI}(H_u,1) \\ &\quad +(\frac{n_s}{n})S_{SOI}(H_s,1) \end{aligned} \tag{4.10}$$

*where $S_{SOI}(H,1)$ is the expected cost of a successful search of a hash file with separate overflow chains, given in Equation 4.8.*

**Proof:** The first term $(\frac{b_{max}}{\overline{b}} - 1)$ 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. □

### 4.4.2 Two-file Method

The 2-file method requires that one maintains the counters $n_s$ and $n_u$, $m$ the number of buckets, $b_{smax}$, the maximum bucket occupancy of split buckets, $b_{umax}$, 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 one randomly chooses 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 11** *The expected cost of iterative sampling a sample of size 1, from a Linear Hash file using the two-file method is:*

$$C_{LHI_2}(H,1) = (\frac{n_u}{n})C_{SOI}(H_u,1) + (\frac{n_s}{n})C_{SOI}(H_s,1) \tag{4.11}$$

**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:

$$\begin{aligned} C_{LHI_2}(H,1) &= (\frac{n_u}{n})\left((\frac{b_{umax}}{\overline{b_u}} - 1) + S_{SOI}(H_u,1)\right) \\ &\quad +(\frac{n_s}{n})\left((\frac{b_{smax}}{\overline{b_s}} - 1) + S_{SOI}(H_s,1)\right) \end{aligned} \tag{4.12}$$

**Theorem 12** *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.*

$$C_{LHI_1}(H,1) \geq C_{LHI_2}(H,1) \tag{4.13}$$

**Proof:** Subtracting the two cost formulae gives:

$$C_{LHI_1} - C_{LHI_2} = (\frac{b_{max}}{\overline{b}}) - \left(\frac{n_u}{n}\frac{b_{umax}}{\overline{b_u}} + \frac{n_s}{n}\frac{b_{smax}}{\overline{b_s}}\right) \tag{4.14}$$

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

$$C_{LHI_1} - C_{LHI_2} = \left(\frac{m}{n} - \left(\frac{m_u}{n}\frac{b_{umax}}{b_{max}} + \frac{m_s}{n}\frac{b_{smax}}{b_{max}}\right)\right) b_{max} \tag{4.15}$$

Since $b_{umax}/b_{max} \leq 1$ and $b_{smax}/b_{max} \leq 1$ one obtains the following bound:

$$\begin{aligned} C_{LHI_1} - C_{LHI_2} &\geq \left(\frac{m}{n} - \frac{(m_u + m_s)}{n}\right) b_{max} && (4.16) \\ &\geq (m/n - m/n) b_{max} && (4.17) \\ &\geq 0 && (4.18) \end{aligned}$$

$$C_{LHI_1} \geq C_{LHI_2} \tag{4.19}$$

Q.E.D. □

Thus the difference in performance of the 1-file and 2-file methods arises from excessive rejections by the 1-file method due to large differences in the bucket occupancies between the two files.

## 4.5 Extendible Hashing

In this section I consider Extendible Hash (EX) tables as described by Fagin, et al. in [FNPS79]. In order to make this presentation self-contained I provide a brief review of extendible hashing while introducing my 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 is denoted as $p_j$. Since more than one directory entry may point to a data page, the data page may have multiple names in my 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, one is 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 one is interested in one additional quantity, namely, the number of directory entries which point at page $p_i$. Denote this quantity by $g_i$. The value of $g_i$ can be easily computed from the previously defined counters as follows:

$$g_i = 2^{dd-pd_i} \tag{4.20}$$

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

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

### 4.5.1 Double A/R page sampling

As mentioned above, one starts by picking a random directory entry $D[j]$. As usual, one wants to sample from a page $p_j$ with probability proportional to the number of records on it, $d_j$. However, as I 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 one accepts 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). Recall (from Eqn. 4.20) that $g_i$ can be computed as $g_i = 2^{dd-pd_i}$. Therefore accept page $p_j$ with probability $\alpha_j$:

$$\alpha_j = \frac{(d_j/g_j)}{(d_j/g_j)_{max}} \tag{4.21}$$

The denominator is the maximum of the ratio in the numerator taken over all pages, it appears in this expression to assure that $\alpha_j$ 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 6** *Let $C_{EXTIDP}(H,1)$ = expected cost of sampling one record from an extensible hash file with double A/R page sampling.*

$$C_{EXTIDP}(H,1) = (E[\alpha_j])^{-1} \tag{4.22}$$

**Proof:** This result follows from Lemma 5. Note that to determine $d_j$ one must retrieve the data page, since one assumes no modification to directory. □

Note that one takes the expectation with respect to $j$, the directory entry index, i.e., this expectation is the sum of the quantities $\alpha_j$ each weighted by the fraction of the directory entries pointing at page $p_j$.

### 4.5.2 A/R cell sampling

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

$$\beta_j = (c_j/c_{max}) \tag{4.23}$$

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

**Lemma 7** *Let $C_{EXTICS}(H,1)$ = expected cost of sampling one record from an extensible hash file with A/R cell sampling.*

$$C_{EXTICS}(H,1) = (E[\beta_j])^{-1} \tag{4.24}$$

**Proof:** This result follows from Lemma 5. Note again that to determine $c_j$ one must retrieve the data page, since one assumes no modification to directory is permitted. □

As an example, consider Figure 4.3. The numbers in the directory entries indicate the number of records hashed into them. The number of records stored on each page, the $d_i's$, can be readily maintained on the data pages or determined by simpling counting the records

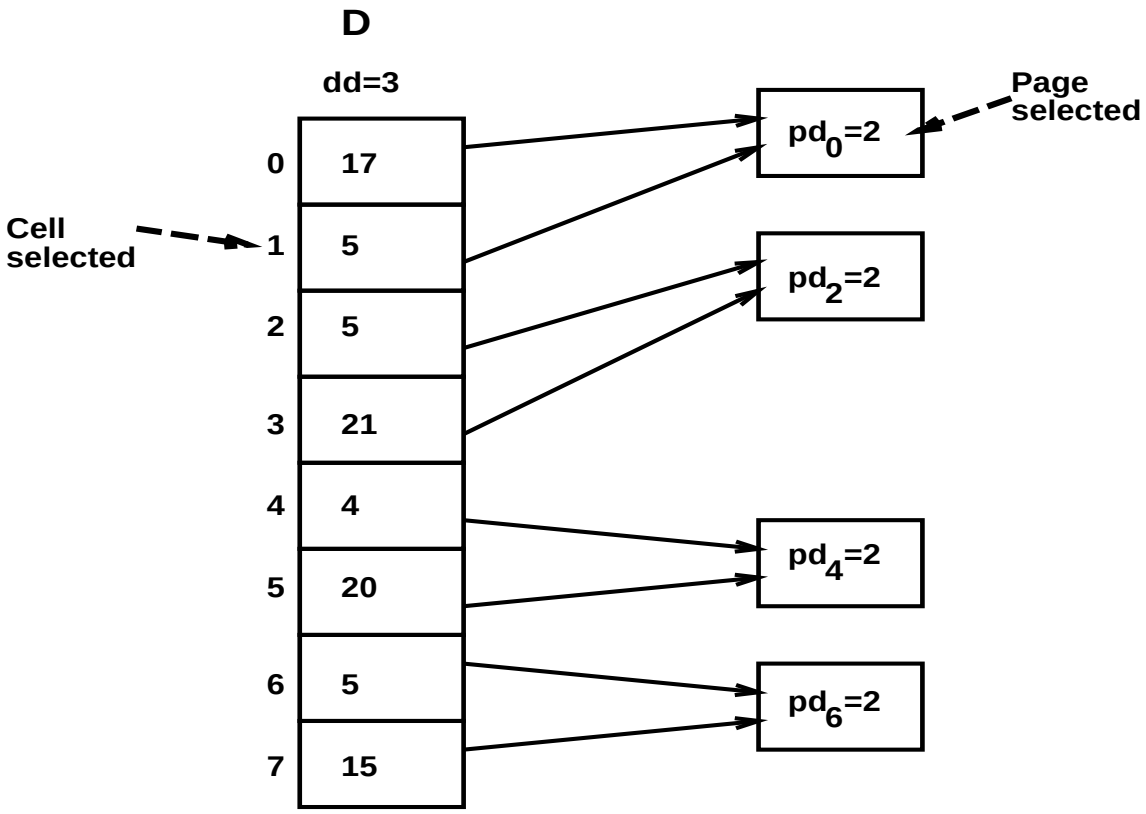



Figure 4.3: Drawing of Extended Hash File

on the candidate sample page. In this case $c_{max} = 21$ and $(d_i/g_i)_{max}$ 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 13** *The cost of double A/R page sampling is always less than or equal to the cost of A/R sampling.*

$$C_{EXTIDP}(H,1) \leq C_{EXTICS}(H,1) \tag{4.25}$$

**Proof:** From Eqn. 4.21 we obtain:

$$E[\alpha_j] = E\left[\frac{(d_j/g_j)}{(d_j/g_j)_{max}}\right] \tag{4.26}$$

$$E[\alpha_j] = \frac{E[(d_j/g_j)]}{(d_j/g_j)_{max}} \tag{4.27}$$

Note that: $\forall j, (d_j/g_j) \leq c_{max}$, hence Thus:

$$E[\alpha_j] \geq \frac{E[(d_j/g_j)]}{c_{max}} \tag{4.28}$$

Observe that:

$$d_j = \sum_{\forall i, D[i]=D[j]} c_i \tag{4.29}$$

Substituting for $d_j/g_j$ yields:

$$E[(d_j/g_j)] = E[E_{\forall i, D[i]=p_j}[c_j]] \tag{4.30}$$

Hence:

$$E[\alpha_j] \geq \frac{E[E_{\forall i, D[i]=p_j}[c_j]]}{c_{max}} = \frac{E[c_j]}{c_{max}} \tag{4.31}$$

Recall that:

$$E[\beta_j] = \frac{E[c_j]}{c_{max}} \tag{4.32}$$

Hence:

$$E[\alpha_j] \geq E[\beta_j] \tag{4.33}$$

From the two lemmas one obtains:

$$C_{EXTIDP}(H,1) \leq C_{EXTICS}(H,1) \tag{4.34}$$

Q.E.D. □

It follows from the above analysis 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.

## 4.6 Batch and Sequential Algorithms

### 4.6.1 Batch Algorithms

In this subsection I consider batch sampling from hash files. This 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. For expository purposes I will present batch sampling for open addressing hash files.

Observe that because of rejections in A/R sampling, one will need an inflated gross sample size, $s'$, so that after acceptance/rejection one is left with a desired net sample size $s$. From the earlier discussion of acceptance/rejection sampling, and sampling from open addressing hash files, one finds that the expected size of the gross sample required for a sample of size $s$ is:

$$s' \approx \frac{b_{max}}{\bar{b}} s \tag{4.35}$$

For *one-pass batch sampling*, 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, one obtains: $\alpha = \bar{b}/b_{max}$. 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 one assumes 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 significant 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, if one overestimates the gross sample size required).

Recall that A/R sampling of open addressing hash files has 3 phases: selection of a random bucket to be sampled from, 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 one selects them all at once. Note that if one randomly tosses balls into urns repeatedly, the resulting occupancy distribution for the urns is equiprobable multinomial. Thus one generates a equiprobable 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, one does all of the A/R tests for a single bucket at once. Since each A/R test produces a Bernoulli random variable, with parameter $b_i/b_{max}$, the sum of $x'_i$ tests will be a binomial

random variable, the number of records accepted from bucket $i$. Thus for each bucket $i$, one generates a binomial random variable $y_i \sim B(x'_i, b_i/b_{max})$.

3. If $y_i > 0$ then one samples $y_i$ records from the bucket, otherwise one proceeds to the next bucket.

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

$$E(C_{OAB}(s', m)) = m(1 - (1 - \frac{1}{m})^{s'})) \tag{4.36}$$

**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 one is sampling with replacement here. □

Alternatively, one can use a Poisson approximation to estimate the number of blocks read.

$$prob(reading\, block_i) \approx 1 - e^{-s'/m} \tag{4.37}$$

Hence, the expected number of blocks read:

$$E[blocks\, read] \approx m(1 - e^{-s'/m}) \tag{4.38}$$

For comparison, recall that the iterative algorithm reads $s'$ blocks.

I conclude this 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 one often needs to reread pages near the root, even if data pages are being read only once for iterative sampling.

### 4.6.2 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 one expects to read most of the pages of the file.

Hence, an alternative to the batch sampling described above is to sequentially scan the file and use a sequential scan (i.e., reservoir) sampling methods such as that described in [Knu81] or [MB83]. These methods were introduced in Section 2.10. In this application, sequential scan sampling requires that one read every page (bucket) of the file, in order to determine the number of records on it (and perhaps sample from them).

Given an accurate estimate of the required gross sample size, batch sampling of a hash file will outperform sequential scan sampling, because the sequential scan algorithm must read every page of the hash file, while the batch sampling algorithm need only read (once) pages which possibly contain sample records - some pages are read, but not sampled because they fail the acceptance/rejection test. At worst, batch sampling might have to read all of the pages of the hash file.

However, if one's purpose is to obtain a sample of records from a hash file which satisfy some selection predicate of unknown selectivity, then one may be unable to reliably estimate the gross sample size required for batch sampling. Failure to estimate a sufficiently large gross sample size will result in too small a net sample. Then the batch sampling process will have to be repeated to obtain the additional sample elements required. Hence, one may find oneself reading some disk pages a second time. At this point sequential scan (i.e., reservoir) sampling becomes attractive, because it will read each disk page in the file only once.

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.

## 4.7 Experimental Results

In this section I present experimental results from simulations concerning the performance of the several of the hash file sampling methods discussed above.

The simulations were based on my algorithms, as described above. These experiments were conducted by Ping Xu, who was then a master's student of Dr. Doron Rotem, and are reproduced here with her permission. Dr. Rotem was also supervising my dissertation at the time. Much more extensive experimental results can be found in Ping Xu's master's thesis [Xu89].

I present here simulation results concerning the both iterative and batch sampling methods for Linear Hashing. I also report results for iterative sampling from Extendible Hash files. These simulation results confirm the analytical results of this chapter and illustrate the behavior the of algorithms. Throughout this section I report sampling cost as the number of disk accesses 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. The uniform random key distribution has been classically used in studies of the performance of hash algorithms. For good hash functions, any smooth key distribution should produce similar results. Key distributions with hot spots (i.e., many persons named John Smith) would produce worse results.

Memory resident versions of the data structures were employing to reduce the running of time the simulations. In order to further reduce the running times for the experiments and to reduce variance of the measurements, successively larger load factor experiments were generated by adding records to the previously generated (lower load factor) hash files and resuming sampling, rather than starting each load factor experiment from an empty hash file. Hence the results for various load factors were not fully independent experiments. As expected the load factor for the hash files is a key performance parameter.

### 4.7.1 Linear Hashing

Figure 4.4 shows the performance of iterative sampling methods from Linear Hash files. In this experiment pages were split whenever the bucket chain length exceeded 3 pages (counting the primary page of the bucket as 1). Page capacity, 50, and number of pages initially in the hash file, 97, were chosen, in part, to facilitate running the experiments in a timely fashion, in main memory of the workstation then in use. I expect larger files would exhibit similar behavior.

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. As buckets begin to be split, the cost of the sampling increases sharply, because the split buckets now have only 50 percent loading of the unsplit buckets. Recall that buckets split from left to right, so that if bucket 50 is forced to split (because it has exceeded the maximum permitted chain length) so will all lower numbered buckets. Since the bucket loading is from a uniform distribution, the first bucket to split will (on average) be the middle bucket. Hence, the sharp increase in sampling costs, which persists until the last bucket splits. Once all of the buckets pages are split, they all have similar occupancy ratios, hence the acceptance/rejection sampling becomes more efficient. Notice in Figure 4.4 that the width and spacing of the regions where 1-File sampling is expensive are doubling in size, corresponding to the doubling in the size of the linear hash file, with each successive round of page splitting.

Figure 4.5 shows a similar experiment in which the page splitting criterion was to split pages whenever the load factor of the primary storage area exceeded 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. The sampling costs grow more gradually, because of the total load factor bucket splitting criterion. (Splitting buckets increases the size of the primary region, bringing the file back below the splitting threshold.) When the last bucket in a round splits, there is a sharp improvement in the sampling efficiency, because the maximum bucket capacity has just dropped almost in half.

### 4.7.2 Batch Sampling from Linear Hash Files

Figures 4.6 and 4.7 compare iterative and batch sampling from Linear Hash files via the 1-file method for the same two splitting criteria shown in Figure 4.4 and Figure 4.5. The batch 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 batch sizes.

In Figure 4.6 note the step function behavior of the batch sampling costs, reflecting the cyclic splitting of the buckets in the linear hash file. With less than 300K keys, the batch sizes are sufficient to force the algorithm to read almost every disk page. Hence, the cost of sampling in this region primarily reflects the size of the hash file, rather than the inefficiencies of acceptance/rejection sampling. However, once the hash file exceeds 350K keys, one is no longer sampling almost every disk page, and the cost of batch sampling for 3K batch samples is nearly the same as for iterative sampling. The cost of 5K batch

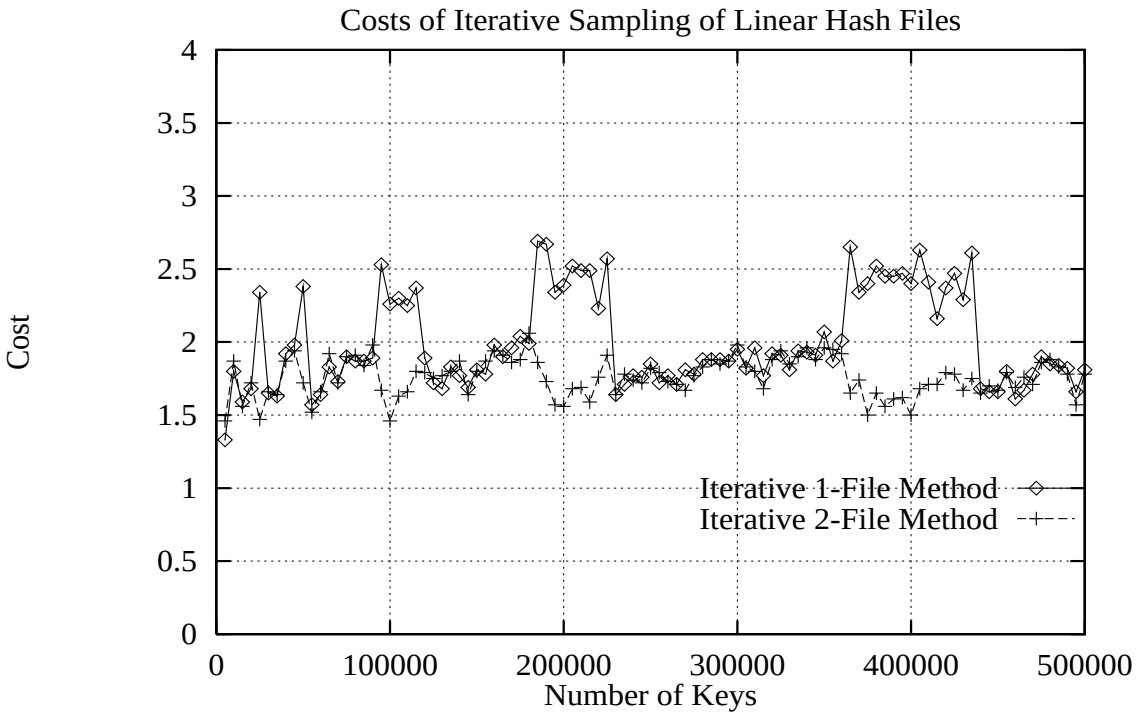



Figure 4.4: The comparison of the costs of the iterative A/R sampling from 1-file and 2-file methods for sampling from Linear Hash Files. (Page splitting criterion is: chain length $\geq 3$, initial number of buckets = 97, page capacity = 50)

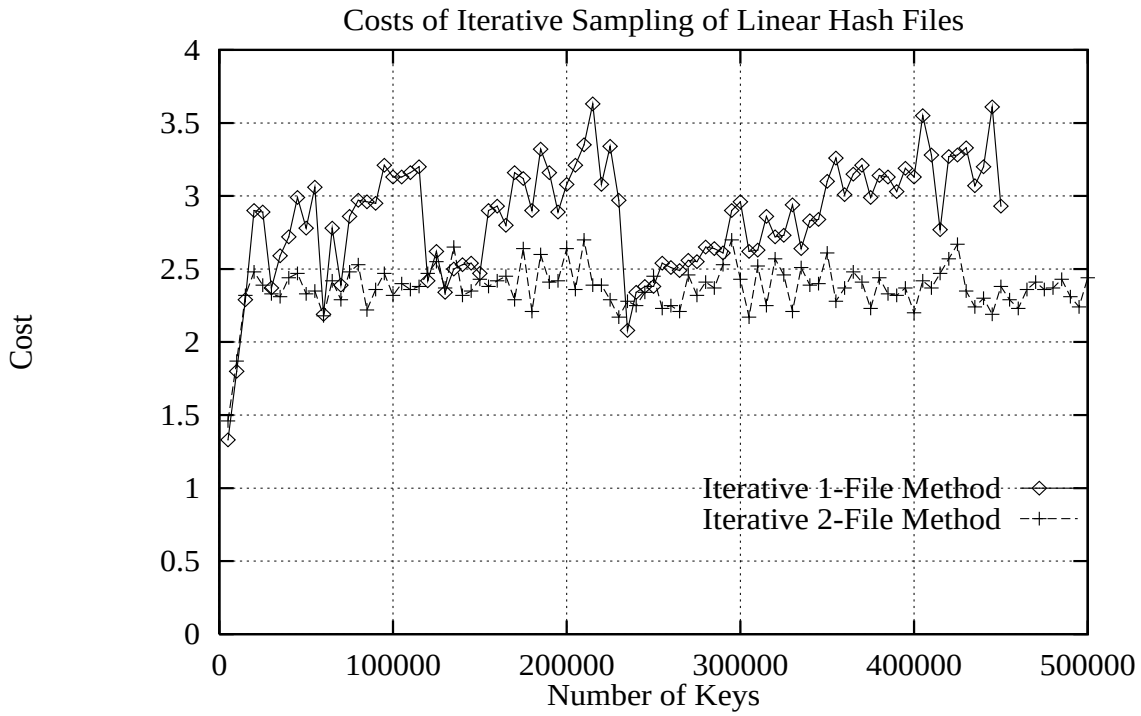



Figure 4.5: The comparison of the costs of the iterative A/R sampling from 1-file and 2-file methods for sampling from Linear Linear Hash Files. (Page splitting criterion is: (total no. of records/capacity of primary area) $\geq 3$, m= 97, page capacity = 50)

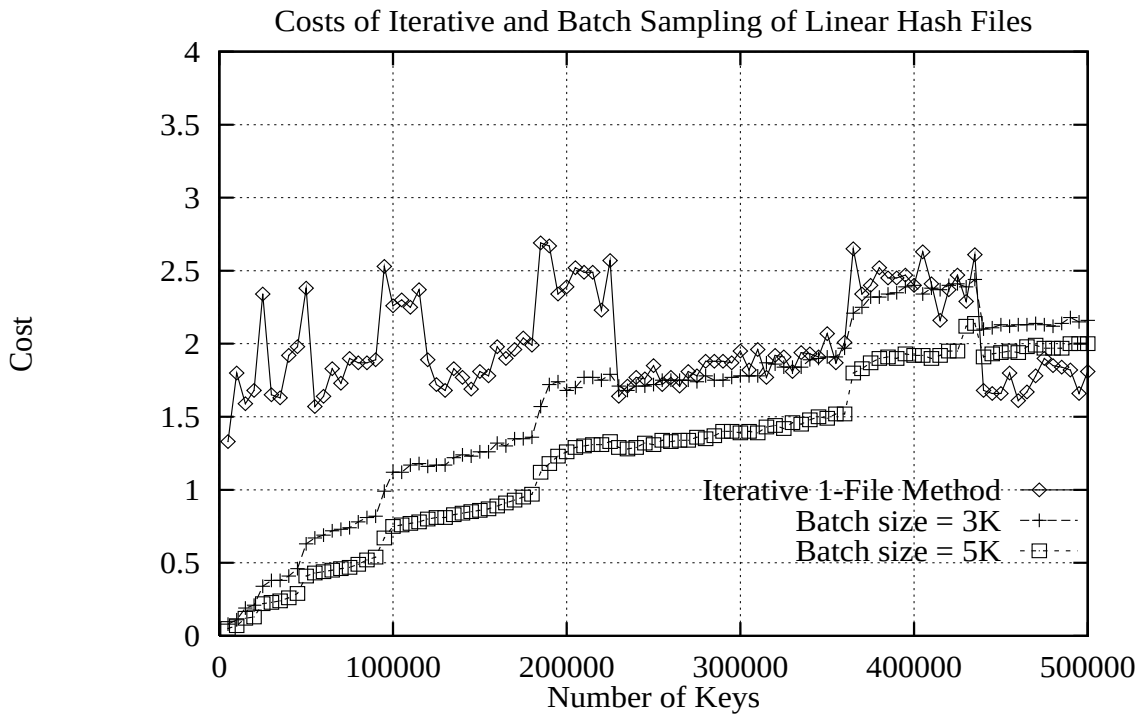



Figure 4.6: Comparison of costs of iterative and batch sampling of Linear Hash Files. (Page splitting criterion is: chain length $\geq 3$, initial number of buckets = 97, page capacity = 50)

sampling is still somewhat less than iterative sampling, but is coming closer, and showing a more pronounced cyclic behavior due to acceptance/rejection sampling.

In contrast consider Figure 4.7. Here the growth in sampling costs is much smoother, especially for batch sampling algorithms, reflecting the smoother growth of the linear hash file due to total load factor splitting criterion. The cyclic cost behavior of the batch sampling does not become evident until the hash file exceeds 400K keys. The more conservative splitting criterion has apparently slightly postponed a major split cycle.

### 4.7.3 Iterative Sampling from Extendible Hash Files

Figure 4.8 compares 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 lightly loaded files, there is variation in the number of directory cells which point to a given disk page. In such circumstances, double A/R sampling is more efficient.

For heavily loaded files, the two methods present essentially identical performance. When the Extended Hash Files are heavily loaded, each disk page is pointed at by one

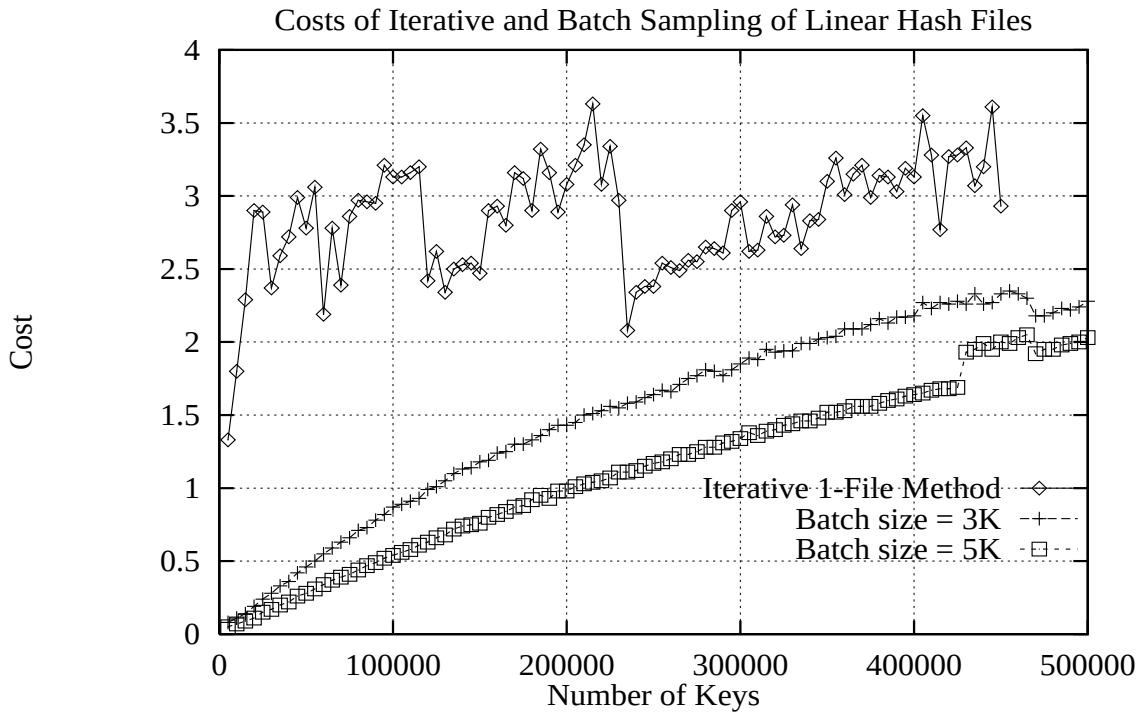



Figure 4.7: Comparison of of the costs of iterative and batch sampling of Linear Hash Files. (Page splitting criterion is: (total no. of records/capacity of primary area) $\geq 3$, initial number of buckets = 97, page capacity = 50)

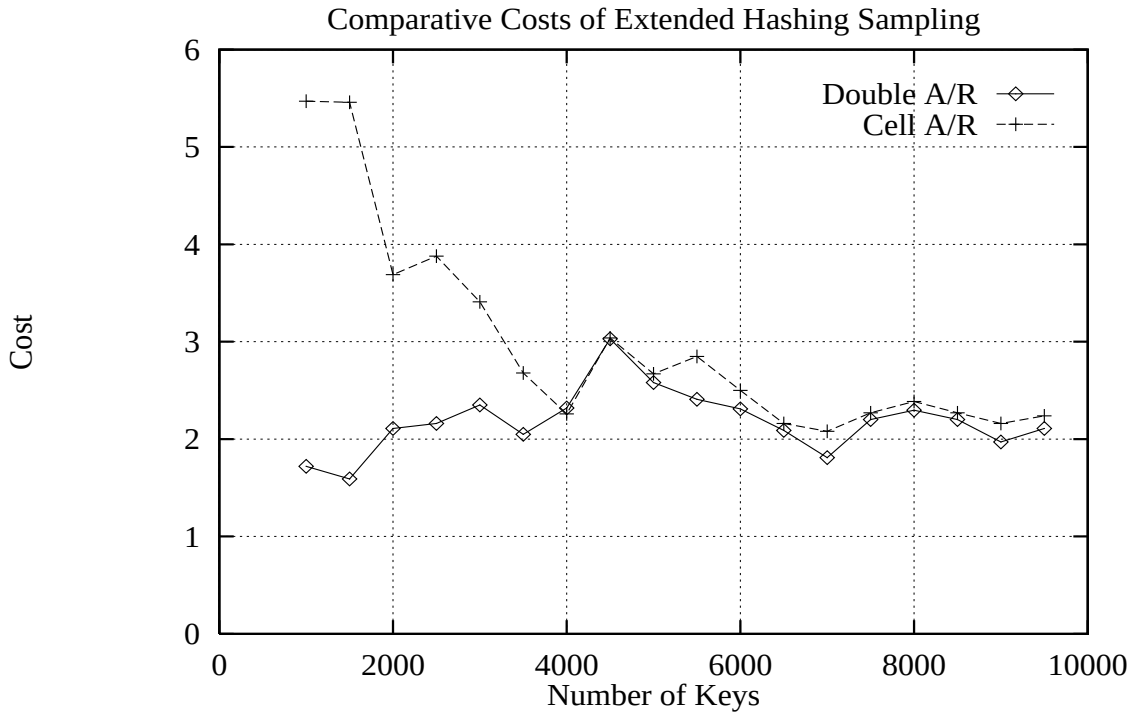



Figure 4.8: Cost of Iterative Sampling from Extendible Hash Files (node capacity = 20, directory size = 1024)

directory cell. Because there is essentially no variation in the number of directory cells which point to a disk page, double A/R sampling and cell A/R sampling are essentially the same algorithm, hence identical performance.

## 4.8 Conclusions

I 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.

For expository reasons, I began the chapter with a discussion of sampling from open addressing hash (OAH) files, although such files are not commonly used in DBMSs. The acceptance/rejection sampling algorithm, ARHASH, required for OAH files is the basis for

the more complex hash file sampling algorithms. It can also be used for sampling from any variably blocked file. DeWitt, et al. [DNSS92] have (subsequent to my publication [ORX90]) described a variant of this algorithm called *extent based sampling* for sampling variably blocked files which are comprised of a small number of contiguous disk extents.

I 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.

As the gross sample size required approaches the number of hash buckets, iterative sampling algorithms will reread some hash buckets. Batch sampling methods do not reread any pages, hence will outperform the iterative methods. However, to perform batch sampling, one must estimate the required gross sample size. This is straightforward, unless one has a selection predicate of unknown selectivity. Poor estimates of the required gross sample size will either result in unnecessary bucket reads, or require a second iteration of the batch sampling algorithm to complete the sample.

Hence, if one is uncertain of the predicate selectivity, but expect to read nearly all of the hash file then sequential scan sampling (reservoir methods) will be preferred to batch sampling, since reservoir sampling algorithms always read the entire hash file exactly once.

# Chapter 5

# Spatial Sampling

## 5.1 Introduction

### 5.1.1 The Problem

In this chapter I am concerned with sampling from spatial databases. Specifically: *given a spatial predicate (target region) specified as the union of a set of polygons, one seeks to generate a random sample of points uniformly distributed over the target region.* I shall also assume one knows the specifications of a rectangular bounding box which encloses the target region.

Such problems arise commonly in geographic information systems (GIS), demographic databases, etc., as discussed earlier in Section 1.5.4.

This raises two unique problems:

- One is typically sampling points in a continuous $n$-dimensional real space, rather than from explicitly stored finite sets,
- One must sample from specialized spatial data structures, such as quadtrees or R-trees.

Assume that the spatial predicate has already been realized as the union of a set of polygons organized into some data structure. Hence, sampling from a spatial predicate can be reduced to sampling from the specific spatial data structure. In doing so, I pass over the problem of sampling from a subset of a spatial data structure which would needlessly complicate this exposition.

A given realization of a spatial predicate (set of polygons organized via some data structure) may be characterized in terms of two parameters:

- *coverage* = the percentage of the area of the bounding box which satisfies the predicate.
- *expected stabbing number* = the average number of polygons which overlap a point in the target region.

I will characterize the choice of preferred spatial sampling algorithms in terms of these parameters.

There are two basic strategies to spatial sampling:

- *Sample First:* First generate a sample point uniformly from the bounding box, then check the spatial predicate. I will show that such methods are efficient when the coverage is high (close to one).
- *Predicate First:* Choose a polygon at random with probability proportional to area, then choose a point within the polygon, calculate the stabbing number, and accept the point with probability inversely proportional to its stabbing number. I will show that these methods are efficient when the stabbing number is low (close to one), and the coverage is low (much less than one).

I discuss sampling from quadtrees and R-trees, two of the most popular data structures which illustrate the major issues involved.

## 5.2 The Model

For convenience I will consider a 2-dimensional database. These results can be extended to higher dimensional databases.

### 5.2.1 Coverage and Stabbing Numbers

In Figure 5.1 I illustrate the definition of the *coverage* and *stabbing number* parameters. Estimating the coverage and average stabbing number is important for the query optimizer in choosing the best sampling strategy. I will explain these parameters and show how to compute them for a simple probabilistic model of a database of randomly placed squares of fixed size. Later, I will discuss how these parameters impact the sampling algorithms and their performance.

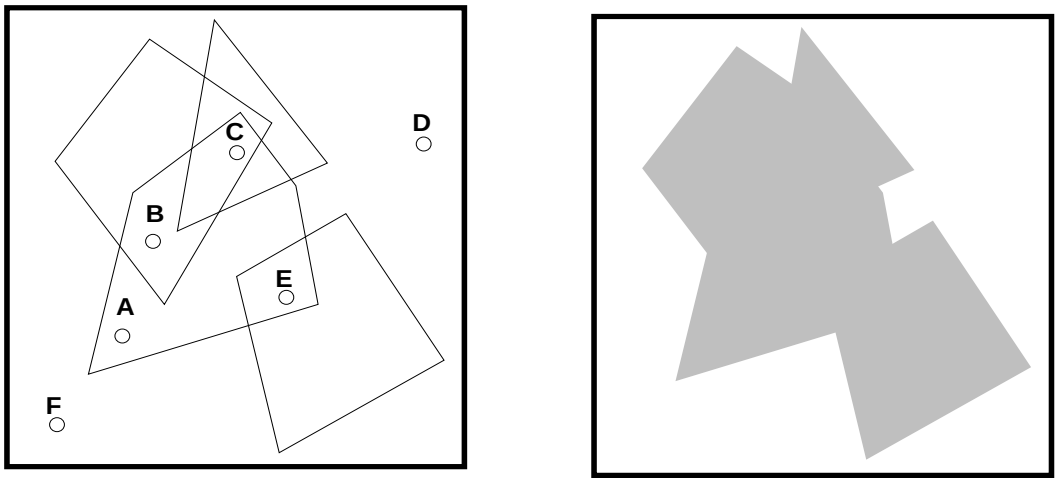
Consider a set of polygons, $\mathbb{P}$, contained in a spatial database (SDB). Define a *bounding box*, $\mathbb{B}$, for the SDB as a rectangle which encloses all of the objects contained in the SDB. $\mathbb{B}$ may either be specified by the database administrator, or automatically maintained by the DBMS. As I shall show, the efficiency of several of the sampling methods is inversely proportional to the area of $\mathbb{B}$, so it is desirable to choose $\mathbb{B}$ as small as possible.

Define the *coverage* with respect to a predicate $p$, $C_p$, as the fraction of the area of $\mathbb{B}$ which satisfies the spatial predicate. In this chapter I shall assume (for expository purposes) that the spatial predicate is specified by inclusion in $\mathbb{P}$. Thus the coverage would be the area of the union of all polygons in $\mathbb{P}$ divided by the total area of the bounding box, $\mathbb{B}$, of the database.

The *stabbing number*, $\tau(x)$ of a point $x$ is the number of polygons in $\mathbb{P}$ which cover it. If the polygons are pairwise disjoint the stabbing number of any point in $\mathbb{P}$ is simply one.

### 5.2.2 Poisson model

To illustrate the relationship between polygon area, polygon density, coverage, and average stabbing number, I will now consider a simple probabilistic model of the polygons in the data base. This model has been extensively studied in the stochastic geometry literature [Hal88, Ser82, Ald89], where is it is known variously as the *mosaic process* or *boolean model*. This simple model has been chosen to facilitate the exposition.




Polygons Union of polygons

Figure 5.1: Examples of coverage and stabbing numbers. The lefthand figure shows the polygons whose union determines the spatial predicate. The coverage is the ratio of the shaded area (union of smaller polygons) shown in the righthand figure to the area of the bounding box. The stabbing numbers of points A,B,C,D,E,F are respectively 1,2,3,0,2,0.

---

Assume that the polygons are generated by first generating the the "centers" of the polygons according to a Poisson spatial point process with point density $\rho$. Each such "center" point is the center of a polygon, whose shape (and size) are drawn from a second independent probability distribution. It will suffice for our purposes that the polygon shape distribution be such that and that the radii of minimal enclosing circles be bounded, and hence the areas of the polygons have finite first and second moments. No other assumptions about the polygon shapes are needed.

Under these assumptions (and ignoring edge effects) one can show [Hal88, Ser82, Ald89] that the stabbing number of any point $x$ in $\mathbb{B}$ will simply be a Poisson random variable with parameter $\lambda = \rho A$, where $A$ is the expected area of a polygon.

Hence, $C$, the coverage is:

$$C = 1 - e^{-\rho A} \tag{5.1}$$

and the expected stabbing number for a point $x$ in $\mathbb{B}$ is:

$$E[\tau] = p(\text{covered})E[\tau|\text{covered}] \tag{5.2}$$

But $E[\tau] = \rho A$ and $p(covered) = C = 1 - e^{-\rho A}$, hence the expected stabbing number for a covered point, $x_c$ in $\mathbb{P}$ is:

$$E[\tau_c] = E[\tau|\text{covered}] = \rho A/(1 - e^{-\rho A}) \tag{5.3}$$

The reader will note that in this Poisson model I have not constrained the polygons to lie entirely within the bounding box $\mathbb{B}$, so that these results are only appropriate if the polygons are small compared to $\mathbb{B}$ (a reasonable assumption for typical GIS applications).

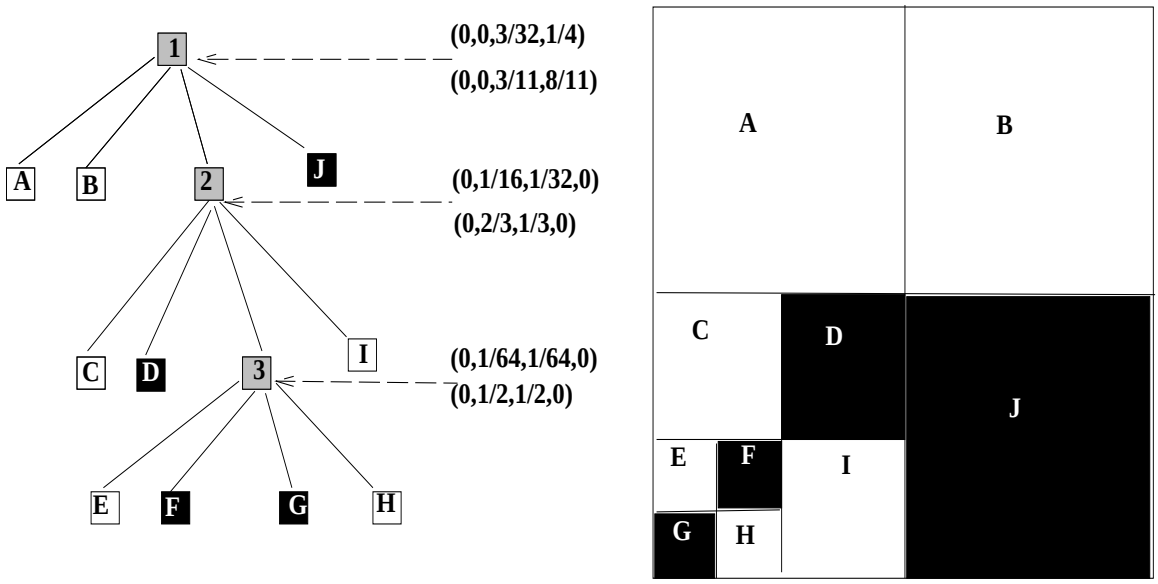



Figure 5.2: Example of a region quadtree. Quadtree is shown on left. Implied spatial partitioning is shown to the right. For each internal node, $(\vec{a})_i$ is shown above, $(\vec{b})_i$ below.

Having seen the origin and relationship (for a simple spatial probabilistic model) of the coverage and average stabbing number parameters, I now turn to an examination of sampling algorithms for two data structures: quadtrees and R-trees.

## 5.3 Quadtrees

I now begin the discussion of spatial sampling algorithms in terms of a region quadtree, in which the spatial predicate is represented as the regions covered with black pixels. A region quadtree is illustrated in shown in Figure 5.2. Quadtrees [Sam84, Sam89b] are actually actually a type of trie, i.e., they recursively partition the unit square into quadrants (e.g., until the quadrant has homogeneously colored pixels).

Quadtrees are both a popular data structure in GIS applications, and facilitate the exposition, as all of the regions are disjoint.

Consider a region quadtree [Sam84, Sam89b] defined over a $n \times n$ array of pixels (where $n = 2^m$ (for some integer $m$)), with the regions being labeled black and white if all of the pixels in a region are 1 or 0 respectively, as shown in Figure 5.2. Suppose that one wishes to obtain a uniform spatial sample of $s$ pixels from the black regions stored in the quadtree.

Recently, Rosenbaum [Ros91] has described a similar algorithm for uniform random sampling of leaves of arbitrary trees. As the reader will see, my algorithm samples leaves in proportion to their area - in order to sample pixels uniformly. For quadtrees, this is actually simpler.

### 5.3.1 Sample First

As noted above, one has one's choice of a Sample First algorithm or a Query First algorithm. The Sample First algorithm consists of loop in which one generates a random pixel location $(x, y)$, and then perform a point location query on the quadtree. If the pixel is black, it is accepted into the sample. Otherwise one loops until a black pixel is accepted. This process is repeated until the desired sample size is obtained.

### 5.3.2 Query First Quadtree Algorithm

The Query First Quadtree (QFQT) algorithm begins at the root. At each node one chooses a branch at random. If the branch is a white leaf, one returns to the root and repeat the sampling procedure. If the branch is a black leaf, one chooses a pixel at random from the region and return. If the branch points to another internal node, one applies this procedure recursively.

Equivalently, this can be seen as choosing a random pixel from the $n^2$ pixels of the quadtree, and then conducting a point location search, repeating the procedure if one discovers the chosen pixel is white. Thus, for the region quadtree, sample first and query first sampling strategies are equivalent (e.g., in terms of number of nodes visited). The reason for the equivalence is that each quadtree node uniformly decomposes its spatial region into four equal-size disjoint quadrants. Also, we have assumed here that the "polygons" are specified as quadtree regions. Other data structures (e.g., R-trees) either do not uniformly partition their spatial regions, or do not partition the space into disjoint regions, or allow variable shape polygons to be stored in the leafs.

Define:

- E[d] = the expected distance to a leaf, assuming that the leaves are sampled with probability proportional to area, i.e.,

$$E[d] = \sum_{l \in leaves} 4^{-d(l)} d(l) \tag{5.4}$$

  where $d(l)$ is the distance from the root to the leaf $l$.

- $p_b$ = the probability of choosing a black pixel when sampling uniformly from space. Note that this is the coverage, $C$.

- $p_w$ for white pixels similarly, note $p_w = 1 - p_b$.

In the example shown in Figure 5.2 $E[d] = 21/16$ and $p_b = 11/32$.

**Theorem 14** *The expected cost, $E[T]$, of obtaining a single sample from a quadtree, $Q$, is given by:*

$$E[T] = E[d]/p_b \tag{5.5}$$

**Proof:** Let $E[d_b]$ = the expected distance to a black leaf, when doing uniform spatial sampling, and $E[d_w]$ for white leaves. Then observe that one can compute $E[T]$ in terms

of sampling either a white pixel or black pixel. If one samples a white pixel, one must resample. This gives the following recurrence equation:

$$E[T] = p_b \cdot E[d_b] + p_w \cdot (E[d_w] + E[T]) \tag{5.6}$$

Rearranging gives:

$$(1 - p_w) \cdot E[T] = p_b \cdot E[d_b] + p_w \cdot (E[d_w]) \tag{5.7}$$

Since the left hand side here is simply $E[d]$, and $(1 - p_w) = p_b$ one has:

$$p_b \cdot E[T] = E[d] \tag{5.8}$$

from which the theorem follows directly. □

For a quadtree defined on an $n \times n$ array of pixels, one can show that $E[T]$ may be $O(n^2)$, i.e., as bad as a sequential scan of the pixels (to obtain a single sample pixel). Below, I describe a partial area quadtree, which is always more efficient, but requires additional maintenance (during updates). I also describe a spatial reservoir algorithm, which is more efficient when $p_b \ll 1$ and the sample size $s$ is comparable to the number of pages in the file.

### 5.3.3 Partial Area Quadtree Algorithms

Sampling can be performed more efficiently by constructing (and maintaining) a *partial area quadtree index (PAQT)*. This is based on the work of Wong and Easton [WE80] on weighted random sampling.

In each internal node $i$ one stores a vector $\vec{a_i} = (a_{i,1}, a_{i,2}, a_{i,3}, a_{i,4})$ where $a_{i,j}$ = the total black area in the leaves of the $j'th$ subtree of the $i'th$ node. In the PAQT algorithm one performs a random walk on the tree from root to leaf. At each internal node $i$ one chooses a branch $j$ with probability $\beta_{i,j} = a_{i,j} / \sum_{k=1}^{4} a_{i,k}$. Thus one defines $\vec{\beta_i} = (\beta_{i,1}, \beta_{i,2}, \beta_{i,3}, \beta_{i,4})$. Note that $\beta_{i,j}$ is zero for white quadrants. Both $\vec{a_i}$ and $\vec{\beta}$ are shown in the example shown in Figure 5.2.

The advantage of the PAQT method is that (unlike QFQT) each random walk on the tree returns a sample pixel. The expected cost of retrieving a sample of size $s$ is thus $sE[d_b]$, the sample size times the expected distance to a black leaf.

### 5.3.4 Spatial Reservoir Quadtree Algorithm

In this section I adapt the classic reservoir sampling algorithm [MB83, Knu81, Vit85] described in Chapter 2 to support spatial sampling from disjoint polygons. Reservoir sampling algorithms permits us to sequentially sample a file of records of unknown size, e.g., intermediate query results. For finite population sampling, the reservoir is initially filled with the $s$ records, where $s$ is the desired sample size. Successive records are added to the sample reservoir with probability $s/k$ for the the $k'th$ record, replacing randomly selected records in the reservoir. Vitter's improvement permits us to skip over records which would not be included in the reservoir, e.g., for the case of random access files.

Adapting the reservoir sampling algorithm to spatial sampling from the quadtree involves two changes:

- sequential traversal of the quadtree leaves,
- "batch" updating of the sample reservoir, upon encountering each new leaf.

One first must contrive to search all of the leaves sequentially. If the leaves are linked together this is trivial, otherwise any method of systematically walking through the tree will suffice (e.g., preorder, postorder, inorder).

One begins the algorithm by walking through the tree until one encounters the first black leaf. Then one fills the sample reservoir with a random sample of $s$ pixels distributed uniformly over the first black leaf. Here $s$ is the target sample size.

One then proceeds to walk through tree looking for black leaves. Whenever one encounters a new black leaf, one randomly replaces some of the sample pixels from the reservoir with new sample pixels chosen uniformly from the newly encountered leaf. The number of sample pixels to be replaced is determined by generating a Binomial random variable, $x \sim B(s, (w_k/W_k))$, where $w_k$ is the area of the $k'th$ black leaf, $W_k = \sum_{i=1}^{k} w_k$, and $x$ is the number of pixels to be replaced. The sample pixels to be replaced are chosen (uniformly) randomly from those in the reservoir. The replacement sample pixels are chosen uniformly from the region enclosed by the new black leaf.

A detailed description of the algorithm is given in Figure 5.3.

**Example:** Let us assume that quadtree shown in Figure 5.2 represents an array of $16 \times 16$ pixels. Suppose that one wants to sample 10 random black pixels. One traverses the leaves from left to right, encountering the black leaves D, F, G, and J with respective areas of 16, 4, 4, 64 pixels. One initially samples 10 random pixels from D and form the putative sample $S_1$. Upon visiting F one forms $S_2$ comprised of 10 pixels by sampling 10 pixels from F and $S_1$ with probabilities $4/20 = 1/5$ and $4/5$ respectively. Upon visiting G one forms $S_3$ comprised of 10 pixels by sampling 10 pixels from G and $S_2$ with probabilities $4/24 = 1/6$ and $5/6$ respectively. Finally, Upon visiting J one forms $S_4$ comprised of 10 pixels by sampling 10 pixels from G and $S_3$ with probabilities $64/88 = 8/11$ and $3/11$ respectively.

**Theorem 15** *The quadtree spatial reservoir algorithm will generate a uniformly distributed random sample of pixels over the union of the regions covered by the leaves.*

**Proof:** The proof is by induction on, $k$, the number of black leaves encountered. I will show that for each $k$, $S_k$ (the sample at stage $k$) is a random sample of size $s$ from the first $k$ leaves encountered. Let us denote by $w_k$ the number of pixels (area) in the $k'th$ black leaf encountered.

For $k = 1$, $S_1$ is by definition a random sample of size $s$. Assume that the theorem is correct for $k = m$, i.e., that $S_m$ is a random sample of size $s$ from the $W_m = \sum_{i=1}^{m} w_i$ black pixels of the first $m$ leaves encountered.

For $k = m + 1$, one samples from the $m + 1'st$ leaf with probability $w_{m+1}/W_{m+1}$, hence a random pixel is drawn from it with probability:

$$p_{entering} = s \cdot w_{m+1}/W_{m+1} \cdot 1/w_{m+1} = s/W_{m+1} \tag{5.9}$$

A pixel from the first $m$ leaves appears in the sample $S_{m+1}$ if it is in $S_m$ and it is not replaced by any of the new pixels entering the sample from the $m + 1'st$ leaf. The probability of this

```
procedure SPRQT(root):
    /* This procedure samples points uniformly form the (black) leaves of a quadtree. */
    /* We assume that the black quadtree leaves have been chained together, and that */
    /* successive leaves can be obtained with a procedure nextleaf(leaf_ptr);        */
begin
    /* Initialize the reservoir to contain points in the first leaf.                 */
    for i := 1 to s do
        q := first(quadtree leaf);
        Generate sample_point[i] ~ Uniform(q);
    endfor
    /* Initialize the area seen.                                                     */
    totalarea := area of quadtree_leaf[i];
    /* Process each leaf                                                             */
    for i := 2 to n_quadtree_leaves do
        begin
            /* Get the next black quadtree leaf                                      */
            q := nextleaf(q);
            /* Determine its area.                                                   */
            a := area(quadtree_leaf[q]);
            totalarea = totalarea + a;
            /* Determine number of sample points in reservoir to replace.           */
            Generate m ~ Binomial(s, a/totalarea);
            /* Generate new sample points.                                           */
            R := SRSWOR of size m from the integers 1 to s;
            /* Replace m sample points in reservoir.                                 */
            for each j ∈ R do
                Generate sample_point[j] ~ Uniform(q);
            endfor
        end
    endfor
    end
```

Figure 5.3: Algorithm for Spatial Reservoir Sampling from a Quadtree

is:

$$p_{remaining} = s/W_m \cdot (1 - w_{m+1}/W_{m+1}) = s/W_{m+1} \tag{5.10}$$

The first term follows by induction, and the second from the definition of the selection process. The theorem is proved since at stage $m+1$ I have shown that $p_{entering} = p_{remaining}$, hence every pixel seen thus far has the same inclusion probability. □

This algorithm obviously requires that one visits every node in the quadtree (unless the leaves are linked). Assuming a depth first search and a cache which can hold the deepest path from root to leaf, the number of pages read will simply be the number of pages in the quadtree.

One could also ask what is the total number of random sample pixels, $s_g$, which must be generated in order to generate a net sample size of $s$. This is given by the following formula:

$$E[s_g] = s \cdot \sum_{i=1}^{n} \frac{w_i}{W_i} \tag{5.11}$$

$$\tag{5.12}$$

where $W_k = \sum_{i=1}^{k} w_k$. Replacing $w_j$ by $w_{max}$ in the numerator and $w_{min}$ in the denominator give an upper bound.

$$E[s_g] \leq s \cdot \sum_{i=1}^{n} \frac{w_{max}}{i \cdot w_{min}} \tag{5.13}$$

$$\leq s \cdot \frac{w_{max}}{w_{min}} \sum_{i=1}^{n} i^{-1} \tag{5.14}$$

$$\leq s \cdot \frac{w_{max}}{w_{min}} H_n \tag{5.15}$$

where $H_n$ is the $n'th$ Harmonic number (approximately $\ln n$). Similarly,

$$E[s_g] \geq s \cdot \frac{w_{min}}{w_{max}} H_n \tag{5.16}$$

Thus one finds that:

**Theorem 16** *The expected gross sample size, $s_g$ is bounded by:*

$$s \cdot \frac{w_{min}}{w_{max}} H_n \leq s_g \leq s \cdot \frac{w_{max}}{w_{min}} H_n \tag{5.17}$$

Table 5.1 summarizes the costs of various sampling algorithms from quadtrees.

## 5.4 R-trees

An R-tree [Gut84, Sam89a] is a spatial data structure analogous to a $B^+$ tree used for storing polygons, i.e., it is a uniform height tree. The polygons are represented in the tree by their minimal bounding rectangles. The bounding rectangles must be *iso-oriented*, i.e., not rotated, but rather aligned with the coordinate axes. The root of the R-tree is the minimal

| Algorithm | Expected Cost Nodes visited |
|---|---|
| Sample First | $sC^{-1}E[d]$ |
| Query First Quadtree | $sC^{-1}E[d]$ |
| Partial Area Quadtree | $sE[d_b]$ |
| Reservoir | $O(n_b)$ |

Table 5.1: Summary of Costs of Sampling from Quadtree. $C$ = coverage, $d$ = distance to leaf, $d_b$ = distance to black leaf, $n_b$ = number of black leaves.

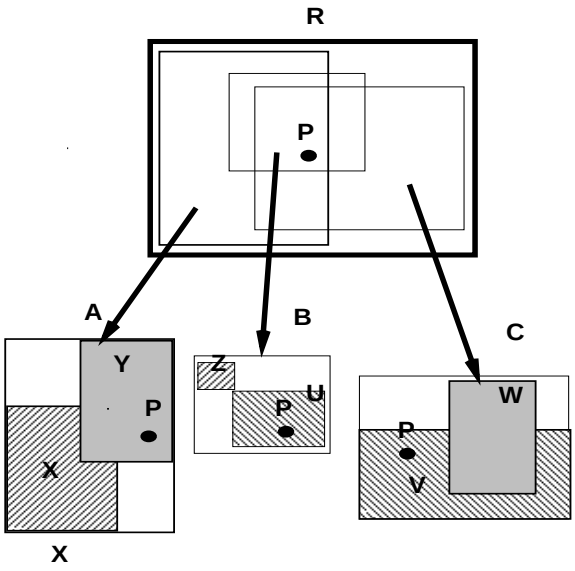



Figure 5.4: Example of R-tree. The point P is included in three polygons, Y, U, and V, hence has a stabbing number of 3.

bounding rectangle, $\mathbb{B}$, which encloses all objects in the database. Each node in the tree corresponds to the minimal bounding rectangle for all of the objects in its subtree. Thus the R-tree is another recursive spatial decomposition. However, the partitioning (nodes) are NOT required to be disjoint. See the example in Figure 5.4

### 5.4.1 Acceptance/Rejection R-tree Algorithms

Such a non-disjoint decomposition has important implications for the sampling algorithms. In order to uniformly sample a point $x$ in $\mathbb{P}$ one can not simply choose a random

path through the R-tree. One must compensate for the fact that some points are contained in more than one leaf, and therefore would be included in the sample with a higher probability.

To do this one must be able to determine the stabbing number for a point. This is a classic problem in computational geometry and a variety of data structures and algorithms have been proposed for its solution in various settings (See [Sam89a].) In the case of an R-tree, determining the stabbing number of a point $x$ is a normal part of the point location query processing. Starting at the root, one must follow every branch from a node which might cover the sample point $x$. This can account for a significant portion of the cost of naive acceptance/rejection sampling from R-trees. Hence, I will also describe early abort algorithms, which avoid much of this cost.

The Acceptance/Rejection R-tree (ARRT) algorithm then consists of picking a path through the tree at random, and then employing acceptance/rejection sampling to compensate for:

1. variable fan-out, $f_j$, at each node $j$ (except the root) on the path from the root to leaf containing polygon $i$ (note that for the leaves "fan-out" is the number of polygons in the leaf),

2. variable polygon area, $A_i$,

3. stabbing number, $\tau(x)$, of the sample point, $x$. $\tau(x)$ can be computed as part of of the sampling procedure.

Thus, the acceptance probability for a point $x$ in polygon $i$ would be:

$$p_{x,i} = (A_i/A_{max})(\tau(x))^{-1} \prod_{\text{j on path to i}} (f_j/f_{max}) \tag{5.18}$$

where $f_{max}$ is the maximum fan-out, $f_j$, over all nodes (leaves) in the tree. Note that for simplicity we assume that the number of polygons/leaf has the same distribution as the fan-out. As in the case of the $B^+$ tree (Chapter 3) one can construct an *early abort algorithm* by performing an acceptance/rejection test at each internal node (or leaf) along a search path (except for the root) with acceptance probability $= f_j/f_{max}$, where $f_j$ is the fan-out (number of polygons for the leaf node). Then perform one last acceptance/rejection test at the leaf with acceptance probability: $p_{x,i} = (A_i/A_{max})(\tau(x))^{-1}$.

Recall from Chapter 2 that for A/R algorithms, that the expected number of sample points needed per sample accepted is equal to the inverse of the expected acceptance probability. Hence, here it would be proportional to the expected stabbing number for covered points.

It is also worth observing that one can perform an early abort version of the acceptance test due to the stabbing number, i.e., performing an acceptance/rejection test as one encounters each additional polygon (beyond the first) covering $x$, with acceptance probabilities $= 1/2, 2/3, 3/4, \ldots, (\tau(x)-1)/\tau(x)$, where $\tau(x)$ is the stabbing number for point $x$. It is trivial to see that the compound acceptance probability is $\prod_{k=2}^{\tau(x)}((k-1)/k)) = 1/\tau(x)$. Unfortunately, one can only perform such an acceptance/rejection test when one encounters a polygon which actually covers the candidate sample points. One must still examine

branches of the tree whose bounding boxes indicate that they might cover the candidate sample, even though searching these subtrees subsequently proves fruitless. The next theorem shows that the early abort algorithm always dominates the naive algorithm in its cost.

The Early Abort A/R R-tree Algorithm is illustrated in Figure 5.4. Here, one randomly selects a polygon, e.g., Y, generate a random point in the polygon, e.g. P. One then searches the R-tree for other covering polygons, i.e., U and V. Upon discovering that U covers P, one attempts to reject P with probability 1/2, if not rejected, one continues the search for covering polygons. Upon encountering polygon V, one attempts to reject P with probability 1/3. Thus the combined acceptance proability for P is 1/3, i.e., the inverse of the stabbing number, as required.

**Theorem 17** *The early abort R-tree acceptance/rejection sampling algorithm will never visit more nodes than the naive R-tree acceptance/rejection sampling.*

**Proof:** The early abort algorithm performs the same stabbing number computation as the naive algorithm unless it decides to reject a point after discovering a covering polygon. Thus the early abort algorithm will never do more work than the naive algorithm, and whenever the stabbing number is greater than one, it will do better (on average).

Specifically, I show below that the expected number of covering polygons visited, $E[P]$, will be $H_{\tau(x)}$ the $\tau(x)'th$ Harmonic number, which is approximately $\log \tau(x)$ [Vit85].

$$E[P] = \sum_{k=1}^{\tau(x)} p(x \geq k) \tag{5.19}$$

but $p(x \geq k) = 1/k$, hence:

$$E[P] = \sum_{k=1}^{\tau(x)} k^{-1} = H_{\tau(x)} \tag{5.20}$$

Note that this does not include the unsuccessful search effort for covering polygons. □

### 5.4.2 Partial Area R-tree (PART) Algorithms

One can construct an algorithm, Partial Area R-tree Algorithm (PART), which employs a Partial Area R-tree, analogous to the Partial Area Quadtree, in which the area of a node is simply the sum of areas of polygons stored in the subtree rooted at that node.

The reader will recall that the Partial Area Quadtree Algorithm is the spatial analog of the Partial Sum Tree (or Ranked Tree) algorithms for sampling uniformly from the leaves of a tree.

The Partial Sum Tree (Ranked Tree) algorithms stored in each internal node the partial sums of the number of leaves contained in the subtree rooted at that internal node. This permitted the sampling algorithm to locate the $k'th$ ranked leaf in time proportional to the average tree height. Hence the ranked tree sampling algorithms simply generate a random record number, and then retrieves the corresponding leaf.

The Partial Area Quadtree Algorithms was the spatial analog, storing at each internal node the sum of the areas of the black leaves of the subtree rooted at that internal node.

Wong and Easton's algorithm [WE80] for weighted random sampling was then used to determine the black leaf from which to sample.

To adapt this algorithm to R-trees it is necessary to:

- Modify the R-tree to store at each internal node the partial sum of the areas of all of the polygons stored in the leaves of the subtree.
- Correct (via acceptance/rejection sampling) for the possible overlap of polygons stored in the R-tree.

The first phase of PART is similar to the Partial Area Quadtree Algorithm, i.e., an adaptation of Wong and Easton's algorithm. However, to correct for overlapping polygons, the first phase of PART would be followed by acceptance/rejection tests for each candidate sample point $x$ where the acceptance probability would simply be:

$$p_{x,i} = \tau(x)^{-1} \tag{5.21}$$

The efficiency will be correspondingly improved over naive acceptance/rejection. As above, one can construct an early abort version of the acceptance/rejection testing due to the stabbing number.

The expected cost (disk pages read) of the early abort partial area R-tree algorithm is:

$$E[C_{PART}(s)] \approx O(\, s\bar{\tau}_c \, \log \bar{\tau}_c \log N\,) \tag{5.22}$$

where $s$ is the target sample size, $\bar{\tau}_c$ is the expected stabbing number of a covered point, and $N$ is the number of polygons. Here $\log N$ reflects the height of tree, $\bar{\tau}_c$ reflects work done to A/R rejections due to stabbing number, $\log \bar{\tau}_c$ reflects the work done to compute the stabbing number for a point (given early abort).

This algorithm (although developed independently) is somewhat similar to Antoshenkov's pseudo-ranked $B^+$ tree sampling algorithm [Ant92]. Both algorithms combine partial sum trees and acceptance/rejection sampling. Note that we could also use Antoshenkov's idea of keeping approximate partial sums to reduce the effort required to maintain the partial area tree, while still obtaining most of the efficiency gains.

### 5.4.3 Spatial Reservoir R-tree Algorithm

If one were to use a naive spatial reservoir algorithm (as I did for quadtrees) for R-trees one would oversample points in the intersection of multiple polygons (in proportion to their stabbing numbers). One will correct for this oversampling by only including a sample point if it was sampled from the "topmost" polygon which covers the point. The ordering of the polygons is immaterial, so one simply orders them as one encounters them in a sequential scan of the leaves of the R-tree. This assures us of a uniform sample from the union of the polygons.

To check whether a point was sampled from the "topmost" (i.e., last covering) polygon, one could check each (current) point in the sample reservoir against each polygon as it is encountered. However, it will typically be more efficient to maintain a k-d tree (or quadtree) of the points in the reservoir, which can be queried with a range query from the bounding box of each polygon subsequently encountered, before conducting the point inclusion tests

```
procedure OPSPRRT(s ):
    /* This procedure computes a simple random sample of s pts from an R-tree.    */
begin
    total_sample := empty set
    sample_size_req := s;
    while sample_size_req > 0  do
        begin
            reservoir_size := sample_size_req * est_avg_stabbing_number;
            SPRRT1P(sample, reservoir_size, sample_size_gen);
            total_sample := total_sample ∪ sample;
            sample_size_req := sample_size_req - sample_size_gen ;
        end
    endwhile
end
```

Figure 5.5: Algorithm for One Pass Spatial Reservoir Sampling from an R-tree

against subsequent polygons. (A k-d tree is a tree which recursively partitions a region into quadrants (for the 2-dimensional case) in contrast to a quadtree which is actually a trie. See [Sam89b].) I expect that savings in point-inclusion-in-polygon tests will more than compensate for maintaining the k-d tree. I assume that the sample reservoir and accompanying k-d tree can be kept in main memory, since the size of these data structures is proportional gross sample size required, approximately $s \cdot \bar{\tau}_c$, i.e., the product of the target sample size and the expected stabbing number for covered points. Note that this is larger than the reservoir required for conventional reservoir sampling, or for reservoir sampling from quadtrees where the expected stabbing number for covered points is one. The spatial sample reservoir need not employ a k-d tree, any point spatial data structure could be used.

The algorithm is described in Figure 5.5 and Figure 5.6.

I illustrate the algorithm with an example shown in Figure 5.7. Here the target sample size is 3, and I have set the reservoir size to 4. The respective areas of the polygons are 100, 150, 75 for polygons A, B, C. One starts by sampling points 1, 2, 3, and 4 from polygon A. One inserts them into the k-d tree used to store the sample reservoir (not shown). The area of B is 150, the total area of A and B is thus 250. Hence, one generates a binomial random variate with parameters 4 (the reservoir size) and 0.6 (= 150/250 = the ratio of the new polygon area to the total polygon area seen). Let us suppose this binomial random variate was 2. Then one chooses 2 points from the reservoir at random to replace. Suppose these are points 2 and 3. One deletes them from the k-d tree. Now one checks the remaining points in the reservoir to see if they are covered by B. Since point 4 is covered by B, it is marked "not in sample". One now generates replacements for points 2 and 3 in the reservoir. These are points 5 and 6. The reservoir now contains points 1,4, 5, 6 with point 4 marked not in sample. One now considers polygon C, generating a binomial random variate

```
procedure SPRRT1P(sample, reservoirsize,
    generatedsamplesize);
    declare array of pts: sample[reservoirsize];
    declare array of integers frompolygon[reservoirsize];
    declare array of boolean insample[reservoirsize];

    begin
        /* Initialize the reservoir to contain pts in the first polygon.                 */
        /* Note: "U" is a routine which generates a uniform random sample from a polygon. */
        for i := 1 to s do
            Generate sample[i] ~ U(polygon[1]);
            Insert sample[i] into KDTREE;
            insample[j] = true;
        endfor
        totalarea := area of polygon[1];
        /* Process each polygon.                                                         */
        for i := 2 to n_rtree_leaves do
            begin
                a := area of polygon[i];
                totalarea = totalarea + a;
                Generate m ~ Binomial(s, a/total_area);
                R := SRSWOR of size m from the integers 1 to s;
                /* Delete the sample pts being replaced in reservoir.                    */
                for each j ∈ R do
                    Delete sample[j] from KDTREE;
                endfor
                /* Mark sample pts covered by this this polygon as deleted.              */
                P := pts_in_polygon(polygon[i], KDTREE);
                for each j ∈ P do
                    insample[j] = false;
                endfor
                /* Generate new sample pts uniformly within this polygon.                */
                for each j ∈ R do
                    Generate sample[j] ~ U(polygon[i]);
                    Insert sample[j] into KDTREE;
                    insample[t] = true;
                endfor
            end
        endfor
    end
```

Figure 5.6: Cont. of Algorithm for One Pass Spatial Reservoir Sampling from an R-tree

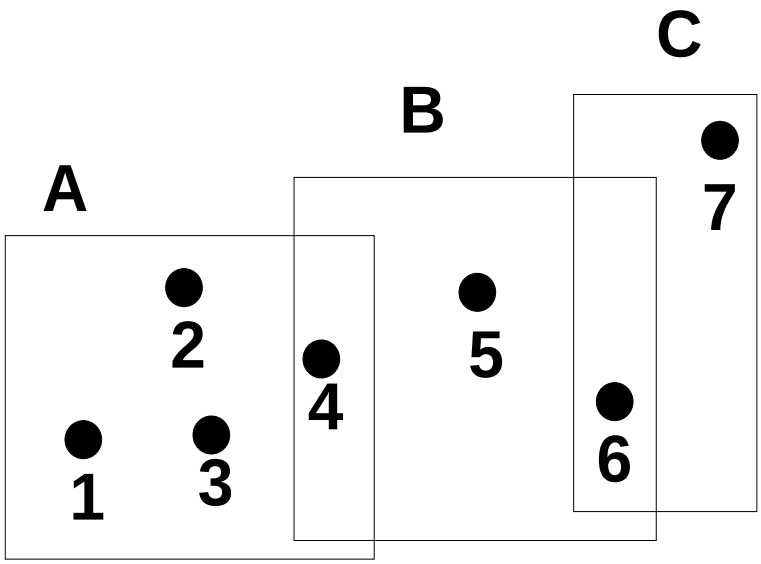



Figure 5.7: Example of One Pass Spatial Reservoir Algorithm.

with parameters 4 and 75/325. Suppose that it is 1. One then randomly selects one of the 4 points in the reservoir to replace. Suppose that point is 4. One deletes 4 from the k-d tree, leaving 1, 5, 6. One checks the remaining points to see if they are covered by polygon C. Point 6 is covered, hence it is marked "not in sample". One now replaces the point, 4, which one just deleted from the k-d tree by generating a random point in polygon C, i.e., point 7, which one inserts into the k-d tree. The reservoir now contains points 1, 5, 6, 7, with point 6 marked "not in sample". Hence the final sample is 1, 5, 7 as desired.

### 5.4.4 Performance Summary

Table 5.2 summarizes the costs of various sampling algorithms from R-trees. Note that the *Reservoir* algorithm has cost linear in the size of the database, but independent of the target sample size. All of the other algorithms have costs linear in the target sample size.

The expected cost of obtaining a sample of size $s$ via the *Sample First* algorithm, $C_{SF}(S)$, is proportional to the expected search cost for single point (which one reckons as $O(\log N)$ under the assumption that polygons are small and the $N$ is the number of polygons), and inversely proportional to the *coverage* (to account for sample points not in the target region). Thus:

$$C_{SF}(s) = O(\, s\frac{\log N}{C}) \tag{5.23}$$

The expected cost of a sample of size $s$ obtained via the A/R Tree algorithm will be:

$$C_{ART}(s) = O(\, s\frac{A_{max}}{E[A]}\left(\frac{f_{max}}{f_{avg}}\right)^{\log N - 1} \bar{\tau}_c^2 \log N) \tag{5.24}$$

| **Algorithm** | **Expected I/O Cost Nodes visited** |
|---|---|
| Sample First | $O(\ s\frac{\log N}{C}\ )$ |
| A/R Tree | $O(\ s\frac{A_{max}}{E[A]}\left(\frac{f_{max}}{f_{avg}}\right)^{\log N-1}\bar{\tau}_c^2 \log N\ )$ |
| Early abort A/R tree | $O(\ s\frac{A_{max}}{E[A]}\left(\frac{f_{max}}{f_{avg}}\right)^{\log N-1}\bar{\tau}_c \log \bar{\tau}_c \log N\ )$ |
| Partial area R-tree (early abort) | $O(\ s\bar{\tau}_c \log \bar{\tau}_c \log N\ )$ |
| One pass Reservoir | $O(N)$ |

Table 5.2: Summary of Costs of Sampling from R-trees. $A$ = area of polygon, $A_{max}$ = max. area of any polygon, $\bar{\tau}_c$ = expected stabbing number of covered points, $N$ = number of polygons, $C$ = coverage, $f_{avg}$ = average fan-out, $f_{max}$ = maximum fan-out.

where $\frac{A_{max}}{E[A]}$ = the ratio of the maximum polygon area to average polygon area, due to acceptance/rejection of varying polygon areas, $\left(\frac{f_{max}}{f_{avg}}\right)^{\log N-1}$ = the ratio of the maximum fan-out to average fan-out of the R-tree raised to the power equal the height of the R-tree minus 1, due to acceptance/rejection of varying fan-out, no A/R at the root node, $\log N$ = the expected search cost for a single point ($\log N$), $\bar{\tau}_c^2$ = the square of the expected stabbing number (once to account for the inefficiences of acceptance/rejection and once to account for computing the stabbing number of a point by searching for all enclosing polygons). This is likely to be pessimistic.

The bound on the expected cost, $C_{EAART}(s)$, of a sample of size $s$, obtained via the *Early Abort A/R Tree* algorithm is the same as for the naive *A/R Tree* algorithm, except that one replaces one expected stabbing number factor by its log, to account for the early abort efficiencies. (We have interchanged the expectation and log operators here, so we actually have an upper bound, since log is concave.) Hence:

$$C_{ART}(s) \approx O(\ s\frac{A_{max}}{E[A]}\left(\frac{f_{max}}{f_{avg}}\right)^{\log N-1}\bar{\tau}_c \log \bar{\tau}_c \log N\ ) \tag{5.25}$$

See the graphs in Figure 5.8 and Figure 5.9 for a comparison of the sampling costs. Early Abort Algorithm always outperforms the naive Acceptance/Rejection R-Tree Algorithm. Partial Area R-Tree Algorithm is still better (but requires increased update maintenance costs). The Sample First Algorithm improves with increasing coverage, and is the best algorithm for high expected stabbing number scenarios. As the sample size increases, the reservoir algorithm becomes increasingly competitive.

## 5.5 Query Optimization

My recommendations for choosing a spatial sampling algorithm are given below and summarized in Table 5.3.

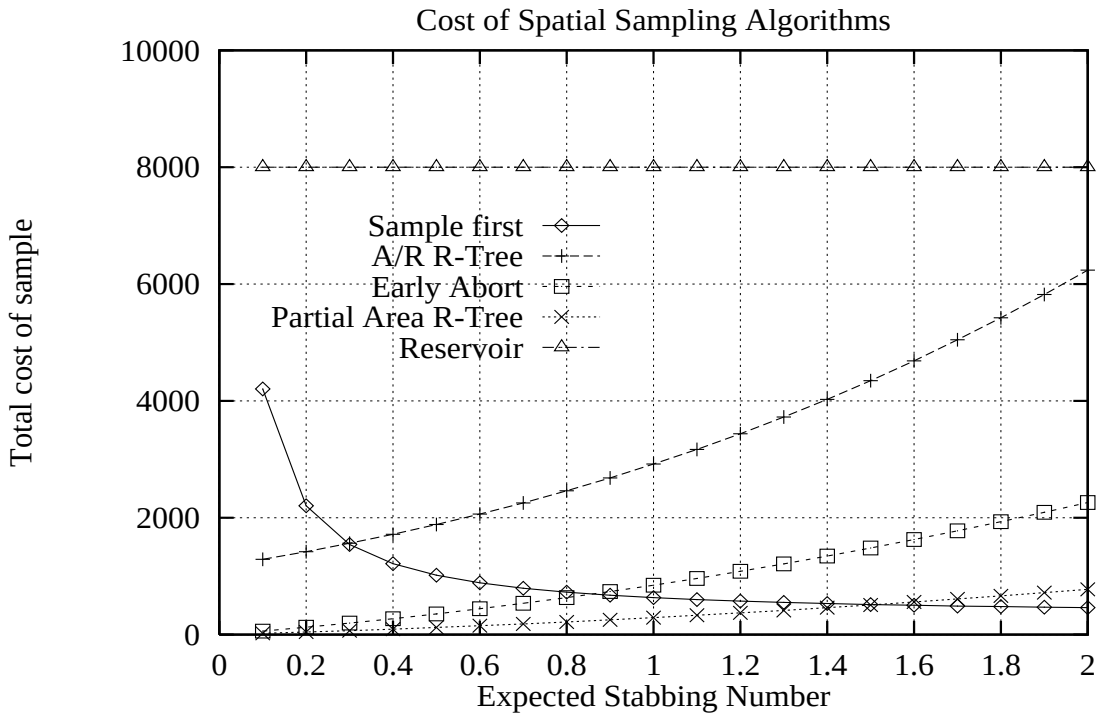



Figure 5.8: Graph of expected costs of R-tree sampling, assuming Poisson model. X-axis is $E[\tau]$, the expected stabbing number of any point in the bounding box, which is varied by adjusting the area of the polygons. Y-axis is the cost in nodes (pages) visited. Assume $s$ = 100 = target sample size, $N$ = 160,000 = (expected) number of polygons, average fan-out = average block size = 20, $\left(\frac{f_{max}}{f_{avg}}\right) \approx 1.4$. In this graph we assume all polygons are equal size, determined by $E[\tau]$ and $N$.

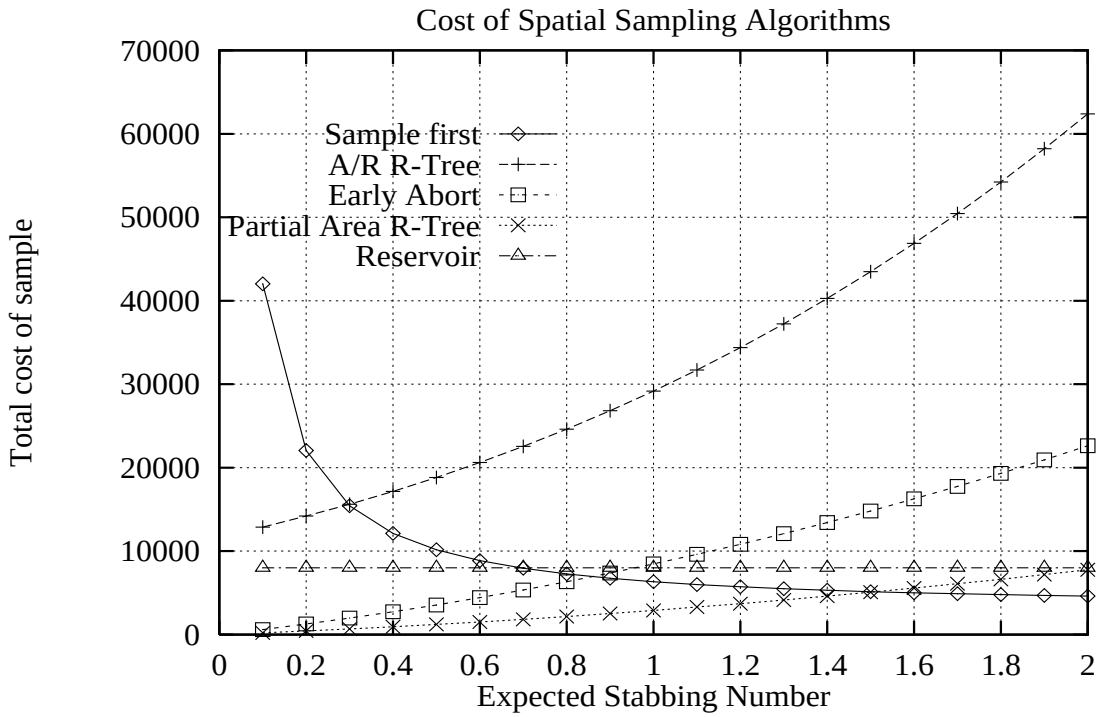



Figure 5.9: Graph of expected costs of R-tree sampling, assuming Poisson model. X-axis is $E[\tau]$, the expected stabbing number of any point in the bounding box, which is varied by adjusting the area of the polygons. Y-axis is the cost in nodes (pages) visited. Assume $s$ = 1,000 = target sample size, $N$ = 160,000 = (expected) number of polygons, average fan-out = average block size = 20, $\left(\frac{f_{max}}{f_{avg}}\right) \approx 1.4$. In this graph we assume all polygons are equal size, determined by $E[\tau]$ and $N$.

| Coverage $0 \leq c \leq 1$ | Sample Density $s/n_{pages}$ | Avg. Stabbing Number | Polygon Area Variability | Update Frequency | Algorithm |
|---|---|---|---|---|---|
| - | $\geq 1$ | - | - | - | Reservoir |
| $\approx 1$ | $\ll 1$ | - | - | - | Sample first |
| $\ll 1$ | $\ll 1$ | $\approx 1$ | low | - | A/R Tree |
| $\ll 1$ | $\ll 1$ | $\approx 1$ | - | low | Hybrid Partial Area Tree |

Table 5.3: Table of Preferred Spatial Sampling Algorithms. (Note: "-" denotes "don't care")

.

- If the sample size approaches (or exceeds) the number of leaves then employ a reservoir algorithm (esp. if the leaves are chained).
- If the coverage is high (close to 1), use a Sample First Algorithm, as this avoids the need to calculate stabbing numbers.
- If the database is relatively static, and sampling queries are frequent, then build a partial area tree, and use a Partial Area R-tree algorithm.
- If the coverage is low, use a Acceptance/Rejection Tree algorithm, as this will not sample points which are not covered.
- For large samples which necessitate reading much of the file, use a batch algorithm, which avoids rereading pages.
- Among algorithms which involve acceptance/rejection, early abort algorithms are always preferable, and most useful for large expected stabbing numbers.

## 5.6 Extensions

Several useful extensions of the algorithms described in this chapter are possible. These include:

- Batch algorithms as described in earlier in the chapter on sampling from $B^+$ trees. These algorithms would do a breadth first search of the quadtree or R-tree, allocating the gross sample via multinomial distributions at each node. For the R-tree one would still need acceptance/rejection sampling to compensate for variable polygon size and overlapping polygons. If one can accurately estimate the required gross sample size, i.e., there are not additional selection predicates, then batch sampling will outperform the iterative algorithms.
- Hybrid algorithms, e.g., a mixture of sample first and query first over various regions of varying coverage. Sample first would be used over regions of dense coverage, query first over regions of low coverage. Although more complex, hybrid algorithms should outperform the corresponding component algorithm, in cases of non-uniform polygon distribution.

- Adaptive (switching) algorithms, which switch sampling algorithms, after accumulating sufficient information to conclude, for example, that a reservoir algorithm will complete faster than an iterative algorithm. Thus one would attempt to estimate the coverage parameter will sampling iteratively. If the estimated coverage was too high one could switch to a sequential scan (i.e., reservoir) algorithm. Similarly, one could imagine algorithms which switched between query first, and sample first. Seppi [SM92] describes a general approach to adaptive query optimization based on a decision theoretic approach. Adaptive algorithms should outperform the corresponding non-adaptive algorithms.

### 5.6.1 Probability Proportional to Density (PPD)

Throughout this chapter I have been concerned with obtaining uniform spatial samples from a region specified as the union of a set of polygons. Before concluding, I turn briefly to a simple case of non-uniform spatial sampling which frequently arises when generating samples of non-uniform spatial processes, where we have aggregate statistics over polygons (census tracts, police precincts, fire districts) for the rates of occurrences.

*Probability proportional to density (PPD)* sampling is the spatial analog of *probability proportional to size (PPS)* sampling over finite sets. One is given a collection of disjoint polygons (census tracts, etc.) each of which contains a known *population* (persons, number of fires, crimes, etc.). Assume that the specified population (fires, crimes, etc.) of each polygon is uniformly distributed within the polygon. One wishes to generate a spatial sample where the inclusion probability of a point is proportional to the *population density* (of persons, crimes, fires) in the enclosing polygon. Such samples are useful for studying models of dispatching of various emergency vehicles, transportation systems, retail marketing, etc.

The algorithms (e.g., for R-trees) which I have previously described for uniform sampling can be easily modified to accommodate the PPD sampling. Simply substitute the population size of each polygon for the area in our calculations for how to allocate the sample across the various polygons. Typically, the polygons (census tracts) are disjoint so that problems of overlapping polygons are avoided.

## 5.7 Conclusions

In this chapter I formulated and classified a variety of problems in random sampling from spatial databases. I have also reviewed several important applications of these techniques in agronomy, forestry, environmental monitoring, planning, etc.

Two major parameters of the database, coverage and expected stabbing number, shape the performance of the sampling algorithms. I have shown how these parameters may be estimated from a simple probabilistic model of a spatial database.

The substantive results of the paper have concerned techniques for uniform spatial point sampling of regions which are defined as the union of polygons stored in various spatial data structures (quadtrees and R-trees).

I have shown how to sample random points from quadtrees how to extend reservoir sampling from finite sets to spatial sampling from quadtrees. I have also shown how to compensate for the non-disjoint partitioning of data space by R-trees by incorporating

stabbing number information into the acceptance probability. An early abort algorithm has been constructed which avoids the need to perform the full stabbing number computation. I have described a one-pass reservoir spatial sampling algorithm which accounts for overlapping polygons.

I have characterized the expected performance of the various spatial sampling algorithms and discussed the circumstances under which various algorithms are likely to prove appropriate.

Much work remains to be done in improving the analyses and implementing and testing the proposed algorithms.

Finally, one observes that data structures which generate non-disjoint spatial partitions (e.g., R-trees) require sampling algorithms which are both more complex and less efficient.

# Chapter 6

# Sampling from Relational Operators

## 6.1 Introduction

In this chapter I will show how to sample the output of individual relational operators such as selection, projection, intersection, union, difference, and join. These sampling techniques form the basic building blocks for sampling from more complex composite queries. The techniques entail a synthesis of the basic file sampling techniques and algorithms for implementing relational operators. I discuss only simple random sampling. I also briefly discuss sampling from more complex relational expressions.

In order to facilitate the exposition, I treat the simpler relational operators first, leaving the more difficult results concerning joins for the later sections of the chapter. The final sections concern sampling from complex relational expressions.

My cost measure is the number of disk pages read, denoted as $D$. Usually one will be interested in the expected number of disk pages read, $E(D)$.

Most of this chapter is written in terms of iterative sampling algorithms, which are analogous to classic tuple-substitution techniques for the evaluation of relational queries. These iterative algorithms have the advantage that they generate the exact sample size required. Iterative algorithms are also readily amenable to incorporation within sequential (adaptive) sampling procedures, in which the sample size is determined after looking at some of the sample. For convenience of exposition, most of the algorithms, e.g., for sampling from joins, are described as iterative algorithms.

However, for every iterative algorithm, there exist corresponding bottom-up (batch) algorithms for sampling. These are analogous to the traditional bottom-up query evaluation strategies widely used in relational DBMSs today because they are typically more efficient than the iterative tuple-substitution query processing strategies.

The problem with bottom-up algorithms applied to sampling queries is that they may not generate the desired sample size. In such cases one may have to repeat the sampling process to obtain the additional sample records needed. Commonly, one would attempt to avoid such problems in bottom-up sampling query evaluation by hedging on the target sample size (using a larger sample), and randomly discarding any excess records. The closing

sections of the chapter, concerning complex relational expressions, are written in terms of the conventional bottom-up query processing strategies. Bottom-up (batch) sampling query evaluation strategies are not as readily incorporated into sequential (adaptive) sampling procedures. However, they could be used in *group sequential sampling procedures.*

## 6.2 Notation

The sampling operator will be denoted as $\psi$. Sampling method and size will be denoted by subscripts. Hence, $\psi_{SRS,100}(R)$ denotes a simple random sample from relation $R$ of size 100. Similarly, $\psi_{SRS,100}(R)$ denotes a simple random sample without replacement of size 100 from relation $R$. If the sampling method is not indicated, then one should assume that simple random sampling without replacement (SRSWOR) is intended.

More complex sampling schemes will be described via the iteration operators:

- **With Replacement Iterator**, denoted WR(s,$< expr >$), which indicates that $< expr >$ (a sampling expression) is to be repeatedly evaluated until a sample with replacement of size $s$ obtained, i.e., no duplicate checking is performing
- **Without Replacement Iterator**, denoted WOR(s,$< expr >$), which indicates that $< expr >$ (a sampling expression) is to be repeatedly evaluated until a sample without replacement of size $s$ obtained, i.e., duplicates are removed.

**Definition 1** *Two sampling schemes, $A(R)$ and $B(R)$ of relation $R$ are said to be equivalent, denoted by $A \Leftrightarrow B$, if, for every possible instance $r$ of relation $R$ they generate exactly the same size samples, and the inclusion probability for each element of the population is the same in both schemes. Note that the samples are not necessarily identical.*

When discussing batch (bottom-up) sampling algorithms we will relax the requirement that sample sizes are *exactly* the same.

**Definition 2** *$MIX(\alpha, < expr_1 >, < expr_2 >)$ denotes a random mixture of two sampling schemes. It indicates that one samples according to sampling expression $< expr_1 >$ with probability $\alpha$, and with probability $1 - \alpha$ one samples according to sampling expression $< expr_2 >$.*

$MIX$ is used to implement sampling from unions.

**Definition 3** *$ACCEPT(\alpha, < expr >)$ indicates that one accepts the sample element generated according to sampling expression $< expr >$ with probability $\alpha$.*

$ACCEPT$ is used to implement sampling from projections and joins.

## 6.3 Selection

Denote the selection of records satisfying predicate *pred* from relation $R$ by $\sigma_{pred}(R)$ The number of records in relation $R$ is $n$. The fraction of records of relation $A$ which

satisfies predicate $pred$ is $\rho_{pred}$. Hence $n\rho_{pred}$ is the number of records in relation $R$ which satisfy the selection predicate.

Selection is unique in that it correctly commutes with the sampling operator, i.e., selecting from a simple random sample generates a simple random sample of a selection.

**Theorem 18** *Sampling and selection operators commute (up to sample size), i.e., we can obtain a sample of a selection by repeatedly sampling individual records and evaluating the selection predicate on each sampled record until we obtain target sample size.*

$$\psi_s(\sigma_{pred}(R)) \Leftrightarrow WOR(s, \sigma_{pred}(\psi_1(R)))$$

**Proof:** For records which do not satisfy the predicate, the inclusion probability is obviously zero on both sides.

In the sampling scheme on the lefthand side the inclusion probability, $p$, for any record $r$ which satisfies the selection predicate is:

$$p = s/(n\rho_{pred}) \tag{6.1}$$

i.e., all such records have equal inclusion probabilities.

On the righthand side, one repeatedly sample one record from $R$, evaluate the selection predicate, and then retain it if it satisfies the selection predicate and is not a duplicate. This continues until one has a sample size $s$.

Since the selection operator does not alter the inclusion probabilities of those records which satisfy the selection predicate, they remain equi-probable. From the definition of the $WOR$ iterator, one is assured that the sample size is $s$ distinct records. □

Techniques for sampling from selections may be classified according as to whether they use an index, or scan the entire relation. The first class can be further classified according to whether the index contains rank information, which permits random access to the $j'th$ ranked record. Assume that the index is a single attribute record-level index constructed as a $B^+$ tree. Except as noted, assume that the predicate can be fully resolved by the index. Based on this classification schema, one has the following algorithms:

- KSKIPI: sample sequentially via random accesses in the index pages, i.e., skipping over records not included in the sample reservoir [Vit85],
- RAI: random access sample via the index, iterating until desired sample size is obtained,
- SCANI: sequentially sample via the index, i.e., scan every relevant index page,
- RA: random access sample, iterating until the desired sample size is obtained,
- SCAN: sample sequentially scanning every page of the relation,

In order to generate random accesses via the index, one must assume that the index includes rank information as discussed in Section 5.3.

The first method, sequential sampling via random access skips (KSKIPI) can be expedited [Vit84] if the population size (number of tuples which qualify on the predicate) is

known, i.e., computable from the rank information in the index. In this case the expected number of disk accesses is given by:

$$E(D_{KSKIPI}) \approx (s(1 + \log_f(\frac{n\rho_{pred}}{sf}))) \tag{6.2}$$

Here $f$ is the average fan-out of each node in the $B^+$ tree index. The log term is due to average height in the tree one must backtrack for each skip. Assume one additional access for each element of the sample to actually retrieve the sampled record.

Again assuming rank information in the index, the second method, random probes of the subtree of the index selected by the predicate (RAI), has an expected cost of:

$$E(D_{RAI}) \approx (s(1 + \log_f(\frac{n\rho_{pred}}{f}))). \tag{6.3}$$

Clearly, KSKIPI is always more efficient than RAI for simple predicates. However, there may be occasions in which multi-attribute predicates are specified for which only a single index is available. This precludes the use of KSKIPI, because one doesn't know the size or the identity of the population satisfying the multi-attribute predicate. However, one can continue to use RAI on one index, and evaluate the multi-attribute predicate on each record sampled.

The third method, sequentially sampling via the index consists of finding the pages of the index which point to records which satisfy the predicate, and then sequentially scanning and sampling each such index page, assuming that successive index pages are chained. The sequential sampling would be done with a reservoir method such as [Vit85], which does not require a known population size. This method would be used when the index does not contain the rank information needed for RAI or KSKIPI. It has an expected cost of:

$$E(D_{SCANI}) \approx \log_f(\frac{n}{f}) + \frac{n\rho_{pred}}{f} + s \tag{6.4}$$

The fourth method, direct random access sampling (RA), does not require any index. For a relation with a fixed blocking factor the number disk accesses required to obtain $s$ distinct records is a negative binomial distribution whose mean is given by:

$$E(D_{RA}) = \frac{s}{\rho_{pred}} \tag{6.5}$$

assuming that $s \ll n\rho_{pred}$. (See Section 2.7.1. The advantage of this method is that it does not require an index. If $\rho_{pred}$ is close to 1 this method avoids superfluous accesses to the index. If $\rho_{pred}$ is very small the SCAN method is to be preferred.

The fifth method, SCAN, consists of simply scanning the entire relation to perform the selection, with a pipelined sequential sampling of the result. The number of page accesses is simply the size of the relation:

$$E(D_{SCAN}) = \frac{n}{b_R} \tag{6.6}$$

Here $b_R$ is the blocking factor for relation R.

## 6.4 Projection

For simplicity I only consider projection on a single domain. Similar results hold for projection on multiple domains. Similarly, for expository purpose I treat only sampling with replacement. As shown earlier extensions to sampling without replacement are straightforward. Denote the projection of relation $R$ onto domain $A$ as $\pi_A(R)$.

Let $A$ be an attribute defined on the domain $a_1, a_2, ..., a_m$. The set $R.a_i$ includes all the tuples in $R$ with value $a_i$ on the attribute $A$.

**Definition 4** *The minimum frequency of the attribute $A$ in relation $R$, denoted as $|R.a|_{min}$, is the minimum cardinality in relation $R$ of any projection domain value $a_i$.*

**Theorem 19** *If $A$ is not a (candidate) key of $R$ then, then projection and sampling operators do not commute. The projection of a simple random sample will not yield a simple random sample of the projection (unless the projection domains include a key of the base relation).*

$$\psi_s(\pi_A(R)) \not\Leftrightarrow \pi_A(\psi_s(R))$$

**Proof:** Consider the following counterexample: Relation $R$ is comprised of two domains $< A, B >$. Instance $r$ of relation $R$ is:

$$r = \{(1,2),(1,3),(2,4)\} \tag{6.7}$$

Then the projection $\pi_A(r) = \{(1),(2)\}$ Now

$$prob[(1) \in \psi_1(\pi_A(R))] = 1/2 \tag{6.8}$$
$$prob[(1) \in \pi_A(\psi_1(R))] = 1/3 \tag{6.9}$$

Hence

$$prob[(1) \in \psi_1(\pi_A(R))] \neq prob[(1) \in \pi_A(\psi_1(R))] \tag{6.10}$$

Projection does not generally commute with sampling because the projection operator removes duplicates. Hence, interchanging projection and sampling will produce uneven inclusion probabilities. □

However, if the attribute $A$ is a key of the relation $R$, then there will be no duplicate values of $A$ in $R$, hence projection and sampling can be exchanged with impunity.

**Theorem 20** *One can obtain a simple random sample of a projection, by performing acceptance/rejection sampling from the base relation, where the acceptance probability of a record is inversely proportional to the number of records in the base relation with the same projection attributes.*

$$\psi_{SRSWR,s}(\pi_A(R)) \Leftrightarrow$$

$$\pi_A(WR(s, ACCEPT(\frac{|(R.a)|_{min}}{|(R.a_i)|}, \psi_{SRSWR,1}(R))))$$

*where $|(R.a_i)|$ is the multiplicity of projection domain value $R.a_i$ in relation $R$, i.e., the number of tuples in $R$ which match value $a_i$ on the projection domain.*

**Proof:** On the righthand side one samples with replacement from relation $R$ first. Hence, each value $a_i$ in the projection domain $A$ would have an inclusion probability of $|(R.a_i)|/|R|$. Since one wants uniform inclusion probabilities on the projected domain values, one employs acceptance/rejection sampling to correct the inclusion probabilities. The acceptance probability for a tuple with value $a_i$ in the projection domain $A$ is given as:

$$p_i = \frac{|(R.a)|_{min}}{|(R.a_i)|} \tag{6.11}$$

Hence, for each sample element accepted the probability of including a particular value $a_i$ of the projection domain $A$ is:

$$p = \frac{|(R.a)|_{min}}{|R|} \tag{6.12}$$

i.e., one now has an equi-probable sample of the projection. One repeats the acceptance/rejection sample until one has accumulated $s$ records in one's sample. □

Hence the expected cost is:

$$E(D) \approx s\frac{|(R.a)|_{avg}}{|(R.a)|_{min}} \tag{6.13}$$

assuming $s \ll |\pi_A(R)|$, where $|(R.a)|_{avg} = |R|/m$ is the average cardinality of attribute $A$ over all attribute values $a_i$ present in the relation. Here I have assumed that relation $R$ is hashed on the projection domain so that records may be retrieved in a single access.

In order for the above algorithm to work one must be able to readily determine the cardinality (number of duplicates) of each projected tuple. This requires that the relation to be projected must be either sorted, indexed or hashed on the projection domain. Also one must either know $|(R.a)|_{min}$ or replace it with a lower bound of 1, at the expense of reduced efficiency.

Suppose, however, the relation to be projected is not "indexed" on the projection domains. Then one has 3 choices:

- *Project first*: Do the projection first, then sample. This requires scanning the entire base relation. Main memory versions of projection require space proportional to the size of the projection.

- *Semi-join*: Estimate the required gross sample size, $s_g$, required. Construct a sample with replacement of this size from the base relation $R$, call this sample $T$. Note that this is a multi-set (bag). Compute the semijoin of the base relation $R$ with the sample $T$ over the projection attributes, i.e., find all records in $R$ which match records in $T$ on the projection attributes. Now perform a count query, grouping on the projection attributes, to obtain the cardinalities of matching records in $R$ for each record in $T$. Use these cardinalities to acceptance/rejection sampling of the elements of $T$. Adjust the resulting sample (e.g., by randomly discarding excess elements) to obtain the exact sample size desired. Note that while this algorithm also requires scanning the entire base relation, the main memory space requirements are only proportional to $s_g$, the gross sample size required. This algorithm is also amenable to parallelization, by partitioning the file on some hash function of the projection attributes.

- *Reservoir projection sampling*: Scan the base relation using a reservoir sampling algorithm. Also check each tuple in the base relation to see if it's projection is already in the reservoir. If so, mark the tuple in the reservoir. Only unmarked tuples will be accepted at the end of the relation scan. Thus one only accepts projected tuples if they were the last tuple in the base relation with the specified projection. This circumvents the problem of oversampling duplicate tuples. The advantage of this method is reduction in main memory space requirements, which are only proportional to the required reservoir size, i.e., the gross sample size. This algorithm should outperform the semi-join algorithm discussed above, since it does not require the first sampling phase of the semi-join algorithm. However, it is not clear how to efficiently parallelize this algorithm. The reservoir projection algorithm is analogous to the spatial reservoir R-tree algorithm discussed in Chapter 5.

## 6.5 Intersection

Denote the intersection of two distinct relations $R$ and $T$ as $R \cap T$.

While it is possible to distribute sampling over intersection and still preserve uniform inclusion probabilities, the resulting computation is so inefficient that it is rarely worthwhile.

**Theorem 21** *A simple random sample of the intersection of two relations, $R$, and $S$ can be obtained by sampling from one of the relations checking for inclusion in the other relation.*

$$\psi_s(R \cap T) \Leftrightarrow WOR(s, \psi_1(R) \cap T) \tag{6.14}$$

$$\Leftrightarrow WOR(s, R \cap \psi_1(T)) \tag{6.15}$$

**Proof:** Consider the first case. From the lefthand side one finds that the inclusion proability for tuples in $R \cap T$ is $s/|R \cap T|$, zero otherwise. For the righthand side the inclusion probability for all tuples in the intersection of $R$ and $T$ is $s/|R \cap T|$, zero otherwise. In either case one has a simple random sample. □

One thus has one's choice of which relation to sample from and which relation to do the intersection with. Typically, if only relation $R$ has an index, then one would sample from $T$ and then intersect with $R$ using its index, since the alternative would require scanning all of $T$ in order to perform the intersection.

If both $R$ and $T$ have indices one must consider the relative costs of the two options based on the size of the relations, the type of index (hash, $B^+$ tree, primary or secondary), and the blocking factors for each relation.

If neither $R$ nor $T$ have indices, then one would sample from the larger relation, so that the intersection scan can be performed on the smaller relation.

**Definition 5** *Define the intersection selectivities $\rho_R, \rho_T$ as:*

$$\rho_R = \frac{|R \cap T|}{|R|}, \quad \rho_T = \frac{|R \cap T|}{|T|} \tag{6.16}$$

*i.e., $\rho_R$ is the probability that an element of $R$ is in the intersection of $R$ and $T$, and $\rho_T$ is the probability that an element of $T$ is in the intersection of $R$ and $T$,*

Then the cost in disk accesses of sampling from $R$ and then checking for inclusion in $T$, assuming $T$ has a $B^+$ tree index is:

$$E(D_R) \approx \frac{s}{\rho_R}(1 + \log_{f_{TI}}(\frac{|T|}{f_{TI}})) \tag{6.17}$$

where $f_{TI}$ is the the average fan-out of the $B^+$ tree index to relation $T$. We have assumed here that $R$ is stored with fixed blocking, so that each record sampled requires one disk access. The factor $1/\rho_R$ is due to the attrition of candidate sample elements which are not in the intersection of $R$ and $T$. The factor $(\log_{f_{TI}}(\frac{|T|}{f_{TI}}))$ accounts for the effort required to check each element of $R$ by looking up in relation $T$. Again assume that $s \ll |R \cap T|$, i.e., neglect the extra cost of sampling without replacement.

An analogous formula for $E(D_T)$ can be written if one samples from relation $T$ and check the intersection in relation $R$. The choice of which file to sample from can be made by comparing the values of the two cost formulas.

Subsequent to the original publication of a preliminary version of this chapter [OR86] Hou and Ozsoyoglu proposed an alternative procedure [HOT88] for estimating the size of a set intersection which entailed sampling from each operand and then constructing the intersection of the two random samples. Their approach may be inefficient for very large databases. Suppose each sample is a small fraction of the base relations, $R$ and $S$ (say 0.1 percent or $10^4$ records out of $10^7$ records in each base relation) and that actually $R = S$ so that $R \cap S = R = S$, i.e., $|R \cap S| = 10^7$ Then the expected size of the intersection of the two samples (i.e., 10 records or less) would be $10^{-6}$ times the size of the full intersection. This may well prove too small to provide reliable estimates of size of the intersection. In such cases, my method of simple random sampling from intersections (Chapter 6), which entails sampling from one operand and checking for inclusion in the intersection against the second operand, will likely prove preferable (assuming that the necessary index is at hand). Similar problems arise with their estimation of the size of joins (especially, if they are on keys). Again, in these cases my methods (as discussed in this chapter) will be preferable *if the requisite index is at hand.* Hou and Ozsoyoglu incorporated my methods into the later version [HO91] of their paper.

## 6.6 Difference

Denote the difference of two relations $R$ and $T$ as $R - T$.

**Theorem 22** *Sampling operators do not distribute over set difference, i.e., the simple random sample of the difference $R - T$ of two relation $R$ and $T$ can not be obtained by taking the difference of simple random samples of $R$ and $T$.*

$$\psi_k(R - T) \not\Leftrightarrow \psi_k(R) - \psi_k(T) \tag{6.18}$$

**Proof:** Distributing sampling over difference operators fails because elements in $R \cap T$ but not in $\psi_k(T)$ may be erroneously included in $\psi_k(R) - \psi_k(T)$. □

**Theorem 23** *A simple random sample of the intersection $R - T$ of two relations, $R$ and $T$, can be obtained by taking a simple random sample of $R$ and then checking that it is not contained in $T$.*

$$\psi_s(R - T) \Leftrightarrow WOR(s, \psi_1(R) - T) \tag{6.19}$$

*i.e., sampling from the difference of two relations is equivalent to sampling from the first relation and then taking the difference.*

**Proof:** Clearly the sampling scheme on the righthand size will produce a sample without replacement of the desired size. It remains to be shown that the sample is from $R - T$ and that each element in $R - T$ has an equal inclusion probability. Since $\psi_1(R) \in R$ it follows that $\psi_1(R) - T \in (R - T)$. Since $\psi_1(R)$ has uniform inclusion probabilities over all elements of $R$ and set differencing with T does not alter the inclusion probabilities of records in $R - T$, it follows that the righthand side sampling scheme has uniform inclusion probabilities for records in $R - T$. □

Thus sampling from relation differences is very similar to sampling from relation intersections. One samples from $R$ and then check that the tuple is not in $T$. Hence the expected cost assuming $T$ has a $B^+$ tree index is approximately:

$$E(D) \approx \frac{s}{(1 - \rho_R)}(1 + \log_{f_{TI}}(\frac{|T|}{f_{TI}})) \tag{6.20}$$

where $s$ is the target sample size, $\rho_R$ is the intersection selectivity defined previously as: $\rho_R = \frac{|R \cap T|}{|R|}$. Here $1 - \rho_R$ is the probability that an element of $R$ is not contained in $T$. Hence, (if sampling with replacement) the number of iterations required to retrieve each element of the sample will be geometric with mean $1/(1 - \rho_R)$. The $\log_{f_{TI}}(\frac{|T|}{f_{TI}})$ factor is simply the expected cost of searching the B-tree in which relation $T$ is stored.

Again assume that $s \ll |R - T|$, i.e., neglecting the extra cost of sampling without replacement.

## 6.7 Union

Denote the union of two distinct relations $R$ and $T$ as $R \cup T$.

**Theorem 24** *The union of simple random samples from two relations $R$ and $T$ does not yield a simple random sample of the union of the two relations.*

$$For\ any\ S_1, S_2 : \psi_s(R \cup T) \not\Leftrightarrow \psi_{S_1}(R) \cup \psi_{S_2}(T) \tag{6.21}$$

**Proof:** Interchanging sampling and union fails because all elements on lefthand side have identical inclusion probabilities of $s/|R \cup T|$, whereas the righthand side inclusion probabilities for elements in the intersection $R \cap T$ are $\frac{S_1}{|R|} + \frac{S_2}{|T|} - \frac{S_1 S_2}{|R||T|}$ whereas the inclusion probability for elements in $R - T$ is $S_1/|R|$ and the inclusion probability for elements in $T - R$ is $S_2/|T|$. Hence elements in $R \cap T$ do not have same inclusion probability as elements in $(R \cup T) - (R \cap T)$. □

```
begin
i := RAND(1, |R| + |T|);
If i ≤ |R|
   then get record i from R.
   else
        begin
                j := i − |R|;
                Get record j from T.
                Check if record j is in R.
                If so, discard record j, otherwise retain it.
        end
endif
end
```

Figure 6.1: Code for Union Operator

The correct treatment of sampling from unions requires that one sample elements of intersection only once. Observe that:

$$R \cup T = R \cup (T - R) \tag{6.22}$$

**Theorem 25** *A simple random sample of the union of two relations $R$ and $T$ can be obtained by iteratively attempting to either: sample from one relation $R$, or sampling from the second relation $T$ and testing that the element is not in $R$. The attempts are allocated in proportion to the sizes of the relations $R$ and $T$. Since union is a commutative operator, either relation may be used as the primary relation. (Here I show the result for sampling without replacement, it also holds for sampling with replacement).*

$$\psi_{SRSWOR}(R \cup T) \Leftrightarrow WOR(s, MIX(\frac{|R|}{|R| + |T|}, \psi_1(R), (\psi_1(T) - R)) \tag{6.23}$$

*Recall that $MIX(\alpha, < expr_1 >, < expr_2 >)$ indicates that one samples according to $< expr_1 >$ with probability $\alpha$, and with probability $1 - \alpha$ one samples according to $< expr_2 >$. Also recall that $WOR(s, < expr >)$ indicates that one evaluates the sampling expression $< expr >$ repeatedly (removing duplicates) until one has a sample of size $s$.*

**Proof:** Recall that the problem with naively interchanging sampling and union is that elements of the intersection were sampled twice as often. This algorithm uniformly samples from the concatenation of $R$ and $T$ and then discards those samples from $T$ which lie in the intersection. The result is that elements in the intersection are sampled with the same probability as elements which are not in the intersection, yielding a uniform random sample. □

Then to generate the a single sample of $R \cup T$ we repeat the algorithm in Figure 6.7 until a sample is accepted:

Assuming $B^+$ tree indices, and $s \ll |R \cup T|$, one finds the expected number of iterations of the above algorithm to obtain a single sample is:

$$E(l_R) = \frac{(|R| + |T|)}{(|R \cup T|)} \tag{6.24}$$

For each iteration, one samples $T$ (at a cost of one disk access), and the check the $B^+$ tree index to $R$. Thus each iteration has a cost of:

$$(1 + \frac{|T|}{|R| + |T|} \log_{f_{RI}}(\frac{|R|}{f_{RI}})) \tag{6.25}$$

Hence the total cost is:

$$E[D_{union}] = \frac{(|R| + |T|)}{(|R \cup T|)}(1 + \frac{|T|}{|R| + |T|} \log_{f_{RI}}(\frac{|R|}{f_{RI}})) \tag{6.26}$$

**CAVEAT:** The alert reader will notice that this algorithm requires the availability of the cardinalities of the operand relations $R$ and $T$. If $R$ and $T$ are base relations, this is trivial. However, if $R$ and $T$ are the results of relational expressions, then their cardinalities may prove awkward to obtain without actually computing the the results.

A bottom-up batch version of the algorithm can be written simply as:

$$\psi_{SRSWR}(R \cup T) \Longleftrightarrow \psi_{SRSWR}(R) \cup T \tag{6.27}$$

or

$$\psi_{SRSWR}(R \cup T) \Longleftrightarrow \psi_{SRSWR}(T) \cup R \tag{6.28}$$

Note that one may not get exactly the sample size one needs.

## 6.8 Join

Given two relations $R$ and $T$, let the relation $W$ be the result of their equijoin, i.e., $W = R \underset{R.x=T.x}{\rhd\lhd} T$. In this section I describe algorithms for sampling from $W$. For reasons of efficiency one wishes to avoid computing the full join. For simplicity of exposition I discuss only sampling with replacement in this section. Conversion to sampling without replacement is straightforward.

Sampling from $W$ can be done in different ways depending on the initial structure of the relations $R$ and $S$. Some important factors in determining the sampling method are:

1. Is the join attribute a key in one or more of the joined relations?
2. Are the relations $R$ or $T$ indexed or hashed on the join attribute ?
3. Is the join selectivity factor large?

In this section I will cover several of the basic join sampling methods and evaluate them with respect to their efficiency. I commence by describing the notation and providing some needed definitions.

Denote the semi-join of relation R with relation S over domains A of R and domain B of S as $R \underset{A=B}{\ltimes} S$. Let $X$ be an attribute defined on the domain $x_1, x_2, ..., x_m$. The set $R.x_i$ includes all the tuples in $R$ with value $x_i$ on the attribute $X$. The join selectivity factor $\rho(R \underset{R.x=T.x}{\bowtie} T)$ of relations $R$ and $T$ over the attribute $X$ is defined as

$$\rho(R \underset{R.x=T.x}{\bowtie} T) = \frac{\sum_{i=1}^{m} |R.x_i||T.x_i|}{|R||T|} \tag{6.29}$$

where m is the number of distinct values of the join domain. When the context is clear I will simply denote this by $\rho$.

### 6.8.1 Join without keys

First, I will deal with the case that the join attribute $X$ is not a key in any of the relations $R$ or $T$. Assume that relation $T$ is "indexed" on the join attribute $X$, and that the modal frequency of the attribute $X$ in relation $T$, as defined below, is also known.

**Definition 6** *The modal frequency of the attribute $X$ in relation $T$, denoted as $|T.x|_{max}$, is the maximum cardinality in relation $T$ of any join domain value $x_i$, i.e.,*

$$|T.x|_{max} = \max_{all\ i} |T.x_i| \tag{6.30}$$

**Theorem 26** *A simple random sample of size $s$ of the equi-join of two relations $R$ and $T$ can be obtained by iteratively:*

1. *sampling a single element from $R$*
2. *joining this element to $T$, yielding $V$*
3. *sampling a single element from the join result*
4. *accepting the record with acceptance probability proportional to the cardinality of $V$*

*This repeated until a sample of size $s$ is obtained (including duplicates).*

*In algebraic notation the algorithm is written:*

$$\psi_{SRSWR,s}(R \underset{R.x=T.x}{\bowtie} T) \Longleftrightarrow WR(s, ACCEPT(\frac{|T.x_i|}{|T.x|_{max}}, \psi_1(\psi_1(R) \underset{R.x=T.x}{\bowtie} T))) \tag{6.31}$$

*where $\psi_{SRSWR,s}(U)$ denotes a simple random sample of size $s$ from relational expression $U$, and $|T.x_i|$ denotes the cardinality in relation $T$ of join domain value $x_i$ resulting from the sample $\psi_1(R)$. Recall that $WR(s, < expr >)$ indicates that one should iteratively evaluate the sampling expression $< expr >$ until one has accumulated a sample of size $s$ (including possible duplicates).*

**Proof:** Clearly the righthand side sampling scheme will produce a sample of the requisite size from $R \underset{R.x=T.x}{\bowtie} T$. What must be shown is that it will have uniform inclusion probabilities.

**comment** This version of the algorithm
samples $R$ first;
**For** $j := 1$ **to** $s$ **do**
set *accept* to $false$
**While** *accept* $= false$
**begin**
Choose a random record $r$ from $R$.
Assume $r[X] = x_i$.
Find the cardinality of $T.x_i$.
Accept $r$ into the sample with probability
$\alpha_i = \frac{|T.x|_i}{|T.x|_{max}}$
In case record $r$ is accepted, choose randomly a tuple $t$
of $T.x_i$ and join $r$ with it.
Store the result $r \underset{R.x=T.x}{\rhd\lhd} t$ in the sample file.
set *accept* to $true$.
**end**
**Endwhile**
**Endfor**

Figure 6.2: Algorithm - $RAJOIN_R$

Each iteration of the above algorithm begins by sampling a tuple from $R$. Each tuple in $R$ has inclusion probability $|R|^{-1}$. The sampled tuple, $\psi_1(R)$, is then joined with $T$ and a random sample of the result taken, denoted $\psi_1(\psi_1(R \underset{R.x=T.x}{\rhd\lhd} T))$. Clearly each member of $(\psi_1(R) \underset{R.x=T.x}{\rhd\lhd} T)$ has the inclusion probability $(|T.x_i)|)^{-1}$ as defined above. This single sample is then accepted with probability $\frac{|T.x_i|}{|T.x|_{max}}$. Hence inclusion probability of any member of $R \underset{R.x=T.x}{\rhd\lhd} T$ is given for a single iteration by the product:

$$p = |R|^{-1}(|T.x_i|)^{-1}\frac{|T.x_i|}{|T.x|_{max}} \tag{6.32}$$

$$= \frac{1}{|R||T.x|_{max}} \tag{6.33}$$

i.e., one has uniform inclusion probabilities for each iteration. By induction, this is true for the full algorithm. □

In practice one uses an equivalent, but somewhat faster algorithm which is shown in Figure 6.2. Here, the instead of performing the join first and then sampling and doing the acceptance/rejection test, one determines the number of records from $T$ which would join with the sample element of $R$, performs the acceptance/rejection test, and then (if one passes the A/R test) samples an element from the matching records of $T$ to join with our sample element of $R$.

Note that one does not actually construct the full $\psi_1(R) \underset{R.x=T.x}{\rhd\lhd} T$, since one only needs a sample of size 1 from it.

The efficiency of this method is established in the following lemma.

**Lemma 8** *The expected number of times that the while loop in $RAJOIN_R$ will be performed until a sample element is accepted, $E(l_R)$, is:*

$$E(l_R) = \frac{|T.x|_{max}}{\rho|T|} \tag{6.34}$$

**Proof:** As explained earlier in Chapter 2, the number of times one must evaluate an acceptance/rejection sampling step in order to obtain a single element is a geometric random variable with expectation equal to the inverse of the expected acceptance probability, $E[\alpha_i]$. For the join algorithm, the acceptance probability, $\alpha_i$ is:

$$\alpha_i = \frac{|T.x|_i}{|T.x|_{max}} \tag{6.35}$$

Hence the expectation of the acceptance probability, $E[\alpha_i]$ is:

$$E[\alpha_i] = \sum_{all\ i\ in\ R} \frac{|R.x|_i}{|R|} \frac{|T.x|_i}{|T.x|_{max}} \tag{6.36}$$

Recall that the definition of $\rho$ is:

$$\rho(R \underset{R.x=T.x}{\rhd\lhd} T) = \frac{\sum_{i=1}^{m} |R.x_i||T.x_i|}{|R||T|} \tag{6.37}$$

where m is the number of distinct values of the join domain. Hence rearranging the formula for the expectation of the acceptance probability and substituting for the join selectivity, $\rho$, yields:

$$E[\alpha_i] = \frac{\rho|T|}{|T.x|_{max}} \tag{6.38}$$

since $l_R$ is a geometric random variable with mean proportional to the inverse of the acceptance probability:

$$E(l_R) = E[\alpha_i]^{-1} \tag{6.39}$$

substituting the equation for $E[\alpha_i]$ yields the theorem:

$$E(l_R) = \frac{|T.x|_{max}}{\rho|T|} \tag{6.40}$$

□

Hence the total efficiency (disk accesses) of the algorithm is:

$$E(D_{RAJOIN_R}) \approx sE(l_R)(1 + \log_{f_{TI}}(\frac{|T|}{f_{TI}})) + s \tag{6.41}$$

$$\approx s\frac{|T.x|_{max}}{\rho|T|}(1 + \log_{f_{TI}}(\frac{|T|}{f_{TI}})) + s \tag{6.42}$$

where $f_{TI}$ is the average fan-out for the $B^+$ tree index for $T$. Here the log factor is the time to search the $B^+$ tree index of $T$ for each sample. The last $s$ term in each equation represents the cost of finally retrieving the sampled records from $T$.

If there is an index on $R$ then an analogous algorithm $RAJOIN_T$ can be constructed by simply exchanging the roles of $R$ and $T$ in the above algorithm and cost analysis. If both $R$ and $T$ are indexed, either algorithm could be used. If the join selectivity, $\rho$, is very small, neither algorithm is recommended. Instead, it may be preferable to compute the full join and then sample sequentially the output of the join using [Vit85] as it is generated.

### 6.8.2 Join with key

Suppose that the join domain $X$ is a key of relation $T$ and that relation $T$ is indexed on $X$. In this case, the acceptance/rejection sampling is unnecessary, if one first samples from relation $R$.

**Theorem 27** *If the join domain $X$ is a key (or candidate key) of relation $T$, then we can obtain a simple random sample from a join of $R$ and $T$ by iteratively sampling from $R$ and joining the sample tuple with $T$ until we have obtained the desired sample size.*

$$\psi_{SRSWR,s}(R \underset{R.x=T.x}{\triangleright\!\!\triangleleft} T) \Longleftrightarrow WR(s,(\psi_1(R)) \underset{R.x=T.x}{\triangleright\!\!\triangleleft} T) \tag{6.43}$$

**Proof:** Because the join domain $X$ is a key (or candidate key) of $T$ we can be certain that $T$ contains at most one matching tuple for each tuple in $R$, hence the join acts effectively as a selection on $R$. The result then follows from earlier results on sampling from selections. □

## 6.9 Multiple Joins

I now turn briefly to the question of how to sample from queries which contain multiple joins, intersections, set difference and selection operations. Since selection operations safely interchange with sampling operators the selection operator can be evaluated whenever it is most convenient.

I now define a modified equi-join operator, called the A/R join operator and denoted $\overset{AR}{\triangleright\!\!\triangleleft}$. The A/R join operator takes two arguments, a bag (multi-set), and a relation and performs the combined join and acceptance/rejection sampling, as described above in the section on join sampling. Hence one can describe (a bottom-up batch version) of the previous iterative join algorithm as:

**Theorem 28** *A simple random sample with replacement of a join of $R$ and $T$ can be computed by performing an A/R join of a simple random sample with replacement of $R$ with relation $T$.*

$$\psi_{SRSWR}(R \triangleright\!\!\triangleleft T) \Longleftrightarrow \psi_{SRSWR}(R) \overset{AR}{\triangleright\!\!\triangleleft} T \tag{6.44}$$

*Note that if one employs a batch version of the algorithm, then additional iterations may be necessary to generate a full-size sample.*

Using this description of the join sampling algorithm it is easy to see that for multiple joins one can simply push the sample operator down one path in a join-tree (to the base relation), replacing the join operators with A/R join operators as one goes.

Thus one can transform a join expression as follows:

$$\psi_{SRSWR}(R \bowtie (S \bowtie T)) \Longleftrightarrow \psi_{SRSWR}(R) \overset{AR}{\bowtie} (S \bowtie T) \tag{6.45}$$

or as:

$$\psi_{SRSWR}(R \bowtie (S \bowtie T)) \Longleftrightarrow R \overset{AR}{\bowtie} (\psi_{SRSWR}(S) \overset{AR}{\bowtie} T) \tag{6.46}$$

or as:

$$\psi_{SRSWR}(R \bowtie (S \bowtie T)) \Longleftrightarrow R \overset{AR}{\bowtie} (S \overset{AR}{\bowtie} \psi_{SRSWR}(T)) \tag{6.47}$$

Thus cascaded joins do not pose a major problem for sampling, except that we must somehow obtain a bound for the modal frequency of the join attribute. Similar results hold for set intersection and set difference and for combinations of selections, joins, intersections, and difference operators. Combinations including projection and union are more complex.

Consider the following example.

$$Z = \psi((P \bowtie (Q \bowtie (R \cap T))) - W) \tag{6.48}$$

where the sampling operator is a simple random sample with replacement. One first interchanges the sampling and set difference operators, to get:

$$Z = \psi(P \bowtie (Q \bowtie (R \cap T))) - W \tag{6.49}$$

One then push the sampling operator through first join operator, to get either:

$$Z = (\psi(P) \overset{AR}{\bowtie} (Q \bowtie (R \cap T)) - W \tag{6.50}$$

or

$$Z = (P \overset{AR}{\bowtie} \psi(Q \bowtie (R \cap T))) - W \tag{6.51}$$

Continuing from the latter version, one generates two more alternatives by pushing the sampling operator down one level further:

$$Z = (P \overset{AR}{\bowtie} (\psi(Q) \overset{AR}{\bowtie} (R \cap T)) - W \tag{6.52}$$

or

$$Z = (P \overset{AR}{\bowtie} (Q \overset{AR}{\bowtie} \psi(R \cap T))) - W \tag{6.53}$$

From the second version one pushes the sampling through the intersection operator to obtain two more equivalent expressions:

$$Z = (P \overset{AR}{\bowtie} (Q \overset{AR}{\bowtie} (\psi(R) \cap T))) - W \tag{6.54}$$

or

$$Z = (P \overset{AR}{\bowtie} (Q \overset{AR}{\bowtie} (R \cap \psi(T)))) - W \tag{6.55}$$

All of the above expressions are equivalent, i.e., they will generate simple random samples of the original relational expression - although for the bottom-up algorithms the sample size may vary. For each expression, one may implement either a bottom-up algorithm, or an iterative (tuple-substitution style) version.

## 6.10 Sampling Select-Project-Join Queries

In this section I consider briefly how to extend our results to support sampling from the more complex select-project-join queries which have been widely studied. Such queries are traditionally written as a equi-join query, followed by projection, followed by selection. Such queries are very common in practice.

First recall that selection and sampling operators commute, so that one can perform the selection operator whenever it is convenient. Often query optimizers will attempt to push selection down in the query processing plan (toward the leaves, where the base relations are being read) to reduce the cardinality of the operands as quickly as possible. However, sampling operators will often provide even greater reductions in the cardinality of the operands. Hence, it may be best to perform sampling before selection. I will henceforth ignore the selection operator and simply address sample-project-join queries.

The problem with sampling from project-join queries is that one can not simply sample from the join and then take the projection. Recall that projection removes duplicates as well as restricting the columns retrieved. As I described earlier in the discussion of sampling from projection operators, one must account for the impact of the duplication elimination on inclusion probabilities.

**Theorem 29** *One can compute a sample of the project-join query (onto to attributes $\{z\}$), $\psi_{SRSWR}(\pi_z(R \bowtie T))$, by:*

1. *Compute a sample of the join: $S = \psi_{SRSWR}(R \bowtie T)$*
2. *Compute the projections of the join sample: $U = \pi_{attr_z(R)}(S)$, $V = \pi_{attr_z(T)}(T)$, where $attr_z(R)$ are simply the subset of the final projection attributes which appear in relation $R$, (similarly for $T$).*
3. *Compute the reduced join operands via semi-joins of the sample projections with the base relations: $R' = R \ltimes U$, $T' = T \ltimes V$.*
4. *Compute the join of the reduced operands, to obtain all the elements of the join of the base relations whose projections are equivalent to the projections of the join sample: $W = R' \bowtie T'$*
5. *Perform a COUNT GROUPED BY the projection attributes query to obtain the sizes of the projection equivalence classes: $W' = \pi^c{}_z(W)$*
6. *Augment the join sample elements with the counts (of the corresponding projection equivalence classes): $S' = S \bowtie W'$.*
7. *Perform an acceptance/rejection test for each element of the augmented join sample with acceptance probability for the $i$'th tuple equal to $(S'.count_i)^{-1}$, the inverse of the size of the $i$'th projection equivalence class, to yield the accepted relation $S''$.*
8. *Finally, discard unnecessary columns from the result (retaining any duplicates): $Y = columns_z(S'')$*

**Proof:** This is basically the acceptance/rejection algorithm discussed earlier for sampling from a projection. The steps 2-6 simply compute the cardinalities of elements of the join which have the same projection domain values as the join sample elements. Step 7 does the acceptance/rejection required for the projection. Finally, step 8 discards unneeded columns. □

The practicality of this algorithm hinges on the availability of the requisite indices on the base relations, small values for the counts (projection equivalence class cardinalities), and reasonable expected values for the average acceptance probability.

## 6.11 Domain Sampling from Complex Unions

Now I consider the problem of complex unions, i.e., sampling from the union of two relational expressions. The reader will recall that our algorithm for sampling from unions (discussed above) required the cardinalities of the both operands. If the operands are arbitrary relational expressions this would require that one compute the expressions to determine their sizes. If one is willing to accept approximately correct samples, then could estimate the cardinalities of the operands.

There is another approach which is sometimes practical. I call it *domain sampling*. It was originally conceived as a way to support sampling operations on DBMSs which did not provide any sampling support.

The idea is to sample from the underlying common domain of the keys of the operands, and then join the putative sample with the union of the operands, using the union as a selection predicate.

Hence:

$$\psi_{SRSWR}(R \cup T) \Longleftrightarrow \psi_{SRSWR}(key\_domain(R)) \rhd\lhd (R \cup T) \tag{6.56}$$

where I have assumed that $R$ and $T$ have the same $key\_domain$. In effect one is treating $R \cup T$ as a selection applied (via the join) to the sample from key domain. Since I take a simple random sample from the key domain and I have earlier shown that one can interchange selection with sampling operators (up to sample size), the result will be a simple random sample from the union. I also need the observation that joining with the complete key domain is a no-op, i.e.,

$$key\_domain(R) \rhd\lhd (R \cup T) = R \cup T \tag{6.57}$$

Here the join also restores the non-key attributes.

Distributing the join over the union one obtains:

$$\psi_{SRSWR}((R \cup T)) \Longleftrightarrow (S \rhd\lhd R) \cup (S \rhd\lhd T) \tag{6.58}$$

where $S = \psi_{SRSWR}(key\_domain(R))$. This is algorithm one would use.

The gross sample size required is (approximately) proportional to the inverse of density of the keys in the union, i.e.,

$$s_g \approx s\frac{|key\_domain(R)|}{|R \cup T|} \tag{6.59}$$

where $s_g$ is the gross sample size, $s$ is the target sample size, and $|X|$ is the cardinality of the relation $X$.

The advantage of this algorithm is that it does not require any knowledge of the operand cardinalities, $|R|$ and $|T|$. Hence it can be applied to complex unions where $R$ and $T$ are relational expressions (with a common key domain). It obviously requires that key domain of the operands be a finite set, e.g., a bounded set of integers. While typical key domains are unbounded sets of positive integers one can always substitute the range (mininum,maximum) of keys actually used. The range limits are usually easily maintained or determined.

Note that this approach is not limited to unions, it can be applied to any relational expression. In particular, it could also be an attractive approach to sampling from projections.

## 6.12 Conclusions

In this chapter I have shown how to compute simple random samples from simple relational queries specified by single relational algebra operators: selection, intersection, union, difference, projection, and join. I have used acceptance/rejection techniques to compensate for the effects of the relational algebra operators on inclusion probabilities.

Specifically, I have shown that sampling and selection operators trivially commute. A sample of a projection can be obtained by performing acceptance/rejection sampling of the base relation, with acceptance probability proportional to number of matching records, i.e., matching on the projection domain.

For binary operators, one pushes the sampling operator down one side of the subtree. Thus a sample of the intersection of two relations is computed by intersecting a sample of the first relation and the entire second relation. A sample of set difference is obtained by computing the difference of a sample of the subtrahend and the entire minuend relation. A sample of the union of two relations is obtained by sampling uniformly from the concatenation of the two relations and discarding those elements sampled from the second relation which are in the intersection.

Sampling from the join of two relations is done by sampling from one relation and then doing an acceptance/rejection sampling from the join of the sample tuple with the second relation with acceptance probability proportional to the number of matching records in the second relation.

For all of the commutative operators (intersection, union, and join) one can exchange the roles of the two relations, which side one decides to push the sampling operator down. The query optimizer must choose, based on the availability of indices, conditional inclusion probabilities and join selectivities.

Sampling from cascaded joins is straightforward. One merely pushes the sampling operator down some path in the query plan tree, replacing the join operators encountered with A/R join operators (which perform the needed acceptance/rejection tests). Intermixed cascades of intersections, set differences, joins, and selections can be treated similarly.

Complex union queries are more difficult because the standard algorithm requires that the cardinalities both of the operands be known. However, if the keys are taken from a finite set,e.g., part serial numbers, then one can use *domain sampling* to sample from the union without knowing the cardinality of the operand relations. Domain sampling here entails sampling from the underlying key domain and then joining with union (actually

one distributes the join over the union). Hence, the *domain sampling* algorithm can be used when the "operands" are complex relational expressions, which one does not wish to instantiate. The efficiency is dependent on the proportion of the key domain set which are keys in the result. Domain sampling can be applied to any relational expression.

I have also shown how to sample from a select-project-join (SPJ) query (selections are not an issue). The complex algorithm is appropriate when both operands are indexed and the size of projection equivalence classes is small.

There exist both iterative (tuple-substitution) and bottom-up (batch) versions of each of the sampling query evaluation strategies described. The iterative versions offer exact sample sizes are easy adaptation to sequential (adaptive) sampling procedures, while the bottom-up algorithms often offer better performance when the target sample size is specified externally.

Finally, I note that, because of projection and union operators, it will not always be practical to push sampling operators all the way down the query execution plan tree to the base relations. There will be occasions when it will be most efficient to partially (or even fully) compute the relational expression, and then sample the result. On such occasions one would likely employ one of the reservoir sampling algorithms to permit sampling on-the-fly as the relational result is generated.

# Chapter 7

# Maintaining Materialized Sample Views

## 7.1 Introduction

A *view* is a derived relation whose contents are specified by a relational query against base relations (or previously defined views) of the database. Subsequent to definition *views* can be used in queries wherever a relation may appear. They are widely used in relational database systems for access control, and to present customized schemas for various users.

Views may be either:

- *virtual* - implemented by query modification (i.e., macro-expansion of the queries containing references to views) [Sto75], or
- *materialized* - implemented by instantiation, storage, and update of the view relation.

Materialized views must be maintained as updates are made to the base relations. (I am not concerned here with the issue of updates directly to views.)

*Database snapshots* [AL80, LHMP86] consist of a record of the state of a portion of a database at a particular instant in time. Recording periodic snapshots presents problems very similar to updating materialized views. Periodic snapshots are used to support various analysis and reporting applications.

Define a *sample view* to be a view specified as a sampling query in a manner analogous to conventional view definitions for relational queries. In this chapter I discuss the maintenance of *materialized sample views* (MSVs) in the presence of insertions, deletions, and updates to the underlying base relations. For expository purposes I confine myself to simple (single relational operator) relational queries.

Maintenance of materialized views presents two kinds of problems:

- *policy* - when to update the materialized view [SF91]?
  - incrementally, as each update occurs, (a.k.a. immediate [Han87b] or ASAP)
  - periodically [AL80, LHMP86, SF91],
  - on demand, i.e., when queries are posed against the view (a.k.a. deferred [BC79, Han87b, SF91]);

  - at random times [SF91],
  - hybrid policies [SF91],
  - currency-based [SF90] - enforce a maximum time lag between the state of the base relations and the view.

- *mechanism* - how to update the materialized view?
  - screening tests - which update tuples are relevant to each materialized view [BC79, BLT86, BCL86, Han87b, SP89],
  - view rematerialization - completely recompute the materialized view [BC79],
  - differential update - only compute the changes to the materialized view [BLT86, Han87b].

In this chapter I shall be concerned with only with *mechanism* questions concerning materialized sample views.

The basic idea of this chapter is to reuse the maximal portion of the original sample when constructing the updated sample. These results are thus a synthesis of classical techniques for updating materialized views [SI84, BLT86, TB88, Han87b, CW91] with the algorithms I have described for sampling from relations and queries. Devroye [Dev91] discusses a similar idea of sample reuse *(coupled samples)* in a simulation setting. Sample reuse is the basis of *panel surveys* which are widely used for longitudinal studies.

Throughout this chapter I shall assume that we are computing *simple random samples (SRS)*, i.e., each member of the population has the same inclusion probability. For expository reasons, unless otherwise noted, I will compute samples with replacement (SRSWR). These results can be easily extended to *simple random sampling without replacement (SRSWOR)*, by removing duplicates and increasing the sample size.

### 7.1.1 Organization of Chapter

The remainder of the chapter is organized as follows. In Section 7.2 I discuss maintenance of SRS views against selection queries against single relations stored as variable blocked files. I then discuss sampling from relational operators, treating in turn the maintenance of samples of projection (Section 7.3) and join (Section 7.4). Finally, Section 7.5 contains my conclusions and directions for future work.

### 7.1.2 Notation

Table 7.1 summarizes the notation and abbreviations used in this chapter. The notation is explained further when it is first used. Throughout the chapter, for a variable $x$, I will use $\bar{x}$ to denote the average of $x$, $x_{max}$ to denote the maximum of $x$, and $x'$ to denote the value of the variable $x$ after updating. Individual notations are explained in the text when they are first used.

Here I mention some general conventions. The desired sample size is denoted as $s$, whereas $s_g$ denotes the inflated sample size (i.e., to compensate for losses due to acceptance/rejection sampling). The number of records in the file is denoted $n$. The number of blocks in the file is $m$. Block $i$ is denoted as $B_i$. The block occupancy (in records)

| | |
|---|---|
| $\alpha_i$ | acceptance probability of record $i$ |
| $b_i$ | block occupancy for block $i$ |
| $B_i$ | $i'th$ block of the file |
| $b$ | $= n/m =$ average block occupancy (records) |
| $b_{max}$ | max. block occupancy (records) |
| $C_M(s)$ | expected cost of retrieving sample of size $s$, via method $M$ |
| $m$ | number of blocks in the file |
| $n$ | number of records in file |
| $s$ | number of records desired in sample |
| $s_g$ | gross sample size (to compensate for acceptance/rejection) |
| $Q$ | previous contents of the relational query being sampled |
| hline $\|Q\|$ | no. of tuples prev. contained in the relational query being sampled |
| hline $\Delta_I$ | tuples added to relational query being sampled |
| hline $\|\Delta_I\|$ | number of tuples added to relational query being sampled |
| hline CAR | correlated acceptance/rejection algorithm |
| NCAR | naive correlated acceptance/rejection algorithm |
| 2FS | two-file sampling method |
| RES | reservoir sampling method |
| SRSWR | simple random sample with replacement |
| SRSWOR | simple random sample without replacement |
| A/R | acceptance/rejection sampling |
| $\psi_k(F)$ | sample of size $k$ from file $F$ |
| $\mu_i(A)$ | multiplicity of value $i$ in relation $A$ |

Table 7.1: Notation used in Chapter 7 on Maintaining Sample Views

of block $B_i$ is denoted $b_i$, the average will be $\bar{b}$, and the (actual) maximum will be $b_{max}$. The maximum possible bucket occupancy will be denoted as $b_\Omega$. Overflow chain length for hash bucket $i$ is denoted as $h_i$. Acceptance probabilities of records are denoted $\alpha_k$, and the expectation as $\alpha$. The expected cost (in disk page accesses) of retrieving a sample of size $s$ by a method $M$ is denoted as $C_M(s)$. For batch sampling, one will need to know the expected number of blocks referenced when retrieving $k$ records (at random) from a file containing $m$ blocks, this will be denoted $Y(k, m)$ (Cardenas's function).

I will need to refer the previous value of the query result being sampled for the view, call this $Q$, and let $|Q|$ be its size (number of tuples). Similarly, $\Delta_I$ are the tuples being added to query result being sampled, and $|\Delta_I|$ is their number.

As usual, SRSWR denotes simple random sampling with replacement (duplicates allowed), and SRSWOR denotes simple random sampling without replacement (duplicates removed). The reservoir sampling algorithm is denoted as RES, and the two file sampling method is denoted as 2FS. NCAR denotes the naive correlated acceptance/rejections algorithm for maintaining a materialized view, while CAR denotes the more sophisticated version of the algorithm. $\psi_k(F)$ indicates a simple random sample of size $k$ from file $F$. I will also need the multiplicity of a particular value $i$ in a relation $A$ which I will denote as $\mu_i(A)$.

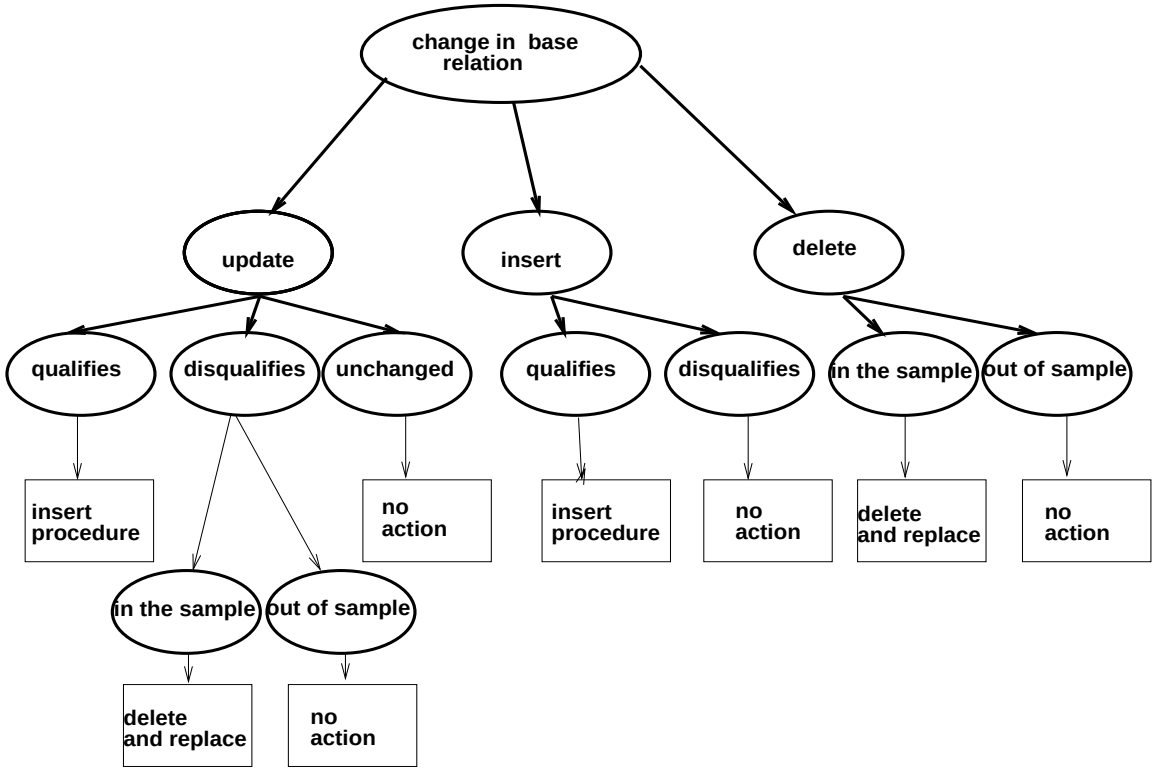



Figure 7.1: Decision tree for sampling strategies.

## 7.2 Updating a Sample Selection View

Figure 7.1 summarizes the strategy for updating sample views of a single relation selection query.

As can be seen in the figure, there are three operations which can be performed to the base relation which could affect a sample view: insert, delete, update.
I treat tuple insertion first. There are two cases:

1. The inserted tuple qualifies for the query predicate. In this case one needs to adjust the sample in a manner described below.

2. The inserted tuple does not qualify for the query predicate. In this case one need do nothing.

I then treat tuple deletion. There are two cases:

1. The deleted tuple was in the sample. In this case one needs to replace the deleted tuple in the sample by simply taking an additional SRSWR of size 1.

2. The deleted tuple was not in the sample. In this case one need do nothing.

I now treat tuple updates. There are three possible cases:

1. The updated tuple now qualifies for the query predicate, where it did not before. This is equivalent to an insertion into the base relation.

2. The updated tuple no longer qualifies for the query predicate. This is equivalent to a deletion.

3. The the status of the query predicate for this tuple is unchanged. No action is necessary, except to update the tuple if it is included in the sample.

### 7.2.1 View Insertion Algorithms

I now describe in detail three algorithms to update a sample view of a of a single relation selection query, where the inserted tuples are known to qualify for the selection predicate (as noted above, nonqualifying tuples require no action).

If the inserted record becomes part of query result being sampled, then one needs to reevaluate one's sample, since it is conceivable, that some record(s) would be replaced in the sample by the newly inserted records.

Three algorithms are available to do this. I describe them briefly here.

1. **Reservoir Sampling**: a method usually used for SRSWOR as described earlier. One begins reservoir sampling from $|Q|+1$, where $|Q|$ is the size of the original query result, from which one constructed the original sample with which one initializes the reservoir. Obviously, this method requires knowledge of the size of the original query result being sampled. While this is usually available when sampling from base relations, it may not be available when sampling from selection (or other types of) queries. When feasible, this algorithm requires I/O time $O(|\Delta_I|)$, where $\Delta_I$ is the set of qualifying inserted tuples and $|\Delta_I|$ is their number. (Assuming that the original sample is in memory.).

2. **Two-file Method**. Here one also assumes that one knows $|Q|$ and $|\Delta_I|$. One allocates the sample to the two partitions $Q$ and $\Delta_I$ according to a binomial distribution as described earlier. One then obtains a SRSWR of specified size from each partition; for samples from partition $Q$ one can reuse one's earlier sample. This method assumes random access to $\Delta_I$. The cost for fixed blocked random access file the expected I/O time is $O(s*|\Delta_I|/(|Q|+|\Delta_I|))$. I have implicitly assumed that $Q$ and $\Delta_I$ are disjoint.

3. **Correlated Acceptance-Rejection (CAR)** The two methods above, both require knowledge of $|Q|$ and $|\Delta_I|$. If there is no index on the selection predicate, and one did not employ reservoir sampling for the original sample, then $|Q|$ will be unknown.

   This method assumes only that one knows the number of blocks $m$ in the original file, the original net sample size $s$, and the number of sample probes which were rejected. It employs acceptance/rejection techniques, effectively rerunning the original A/R sampling over the augmented file. It requires knowledge of $b_{max}$, the maximum number of records/block (or similar cardinality information for more elaborate queries). Both the naive and standard version of CAR are described below. CAR is shown in Figure 7.2.

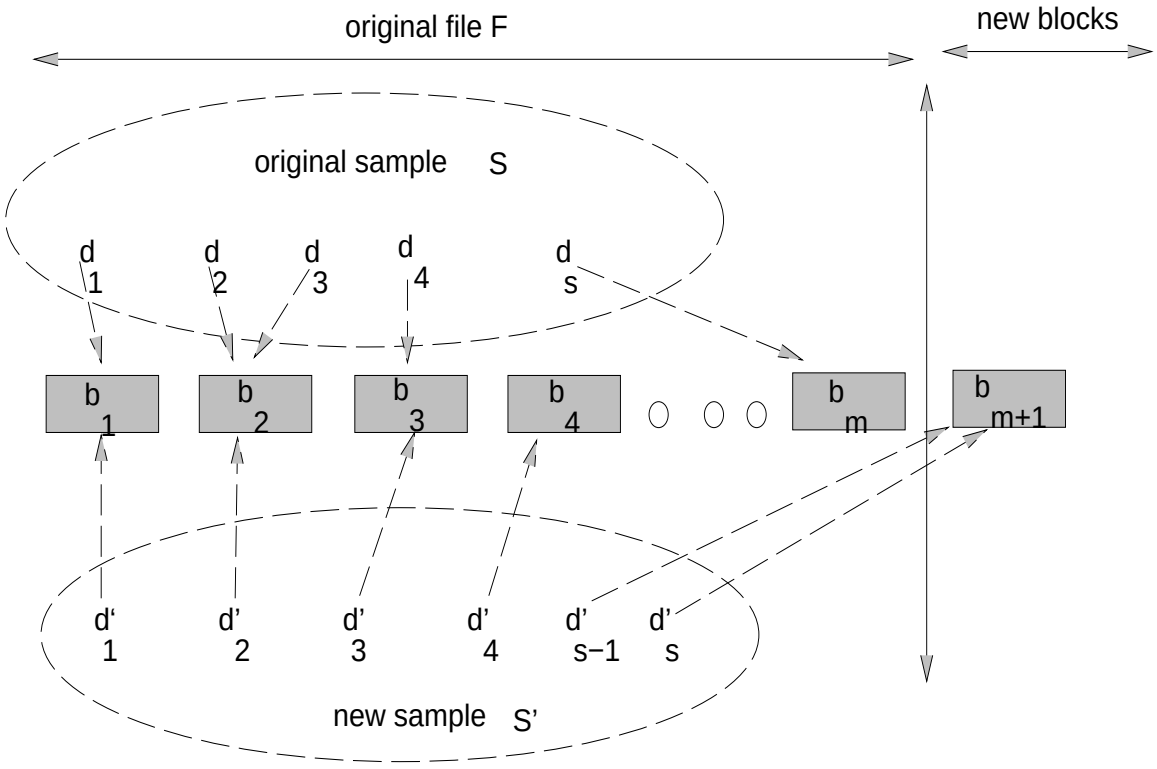



Figure 7.2: Correlated Acceptance-Rejection Sampling

### 7.2.2 Correlated Acceptance/Rejection

Here I discuss a method of reusing our earlier A/R sampling to construct a new A/R sample from a file to which insertions have been appended.

Assume that the original file contained $m$ blocks, each with $b_i$ records and that $b_{max}$ was the maximum number of records per block. In constructing the initial sample one used a gross sample size of $s_g$, and produced a net sample of size $s$.

Now assume that $k$ records have been appended to the file, forming one new block, so that one now has $m+1$ blocks, with

$$b_{max}{}' = max(b_{max}, k) \tag{7.1}$$

i.e., $b_{max}{}'$ is the new max. number of records/block. See Figure 7.2.

Now one can construct a SRSWR from the augmented file via acceptance/rejection by first selecting a block at random from the $m+1$ blocks, and then performing A/R sampling with acceptance probability equal to $b_i/b_{max}{}'$.

Equivalently:

1. one can choose the block of the next sample element to be considered for A/R by first choosing a partition (either the first $m$ blocks, or the last block with prob. $m/m+1, 1/m+1$ respectively);
2. if one winds up in the first partition, then one selects a block at random as before;
3. lastly, one performs the A/R test, with prob. $b_i/b_{max}{}'$;

However, for those sample attempts allocated to the original partition, steps 2 and 3 are identical to the A/R sampling one conducted for the original sample, except that the $b_{max}{}'$ may be larger than $b_{max}$ was. Hence, for those sample attempts allocated to the original partition one can effectively rerun the original A/R sampling, reusing the original sample. However, each accepted element of the reused sample must be subjected to an additional acceptance/rejection test to account for the larger $b_{max}{}'$. The detailed algorithm is described in Figure 7.3.

**Theorem 30** *The naive CAR (NCAR) algorithm will produce a SRSWR from the augmented file* $F' = F \cup B_{m+1}$.

**Proof:** One must show that each record in $F'$ has equal probability of inclusion into the sample. The probability of selecting a particular record from block $B_i$ of $F$ in a single draw, $\gamma_i$ is:

$$\gamma_i = \frac{m}{(m+1)} m^{-1} \alpha_i \frac{b_{max}}{b_{max}{}'} (b_i)^{-1} \tag{7.2}$$

where $\alpha_i = b_i/b_{max}$ was the acceptance probability of a block from block $b_i$ into the original sample.

Hence:

$$\gamma_i = \frac{1}{(m+1)} b_{max}{}' \tag{7.3}$$

Let $F' = F \cup B_{m+1}$ = updated file; $E = \{e_k\}$ = original gross sample; $S$ = original sample; $S'$ will be the updated sample; $\psi_k(F)$ = SRSWR of size $k$ from file $F$; $b_{max}$ = maximum block occupancy for file $F$; $b_{max}{}'$ = maximum block occupancy for the augmented file $F'$.

The procedure NCAR (naive CAR):

```
procedure NCAR;
begin
S' <-- ∅;
J <-- {j : e_j was accepted into S}
i <-- 0;
while |S'| < s and i < |E| do
        Select from F with prob. m/m + 1,
        from augmenting block B_{m+1} otherwise;
        if (selection from F)
        then
              i <-- i + 1;
              if i ∈ J then
                 accept e_i with p = b_max/b_max';
                 if (accepted) then S' <-- S' ∪ e_i endif
              endif
        else if (selection from augmenting block B_{m+1})
             then
                   accept a <-- ψ_1(B_{m+1}) with p = b_{m+1}/b_max'.
                   if (accepted) then S' <-- S' ∪ a endif
             endif;
        endif;
endwhile
/* If the resulting sample size is too small, finish */
/* with a SRS from the augmented file. */
if |S'| < s then S' <-- S' ∪ ψ_{s-|S'|}(F');
return;
end;
```

Figure 7.3: Naive CAR algorithm.

Hence:

$$E[\gamma_i] = \frac{1}{(m+1)} {b_{max}}' \tag{7.4}$$

which is what one wants.

One now must show that the records from the augmenting block, $B_m + 1$ have the same inclusion probabilities as records from $F$.

For the last block, $B_{m+1}$, one has the probability of selecting a particular record from block $B_{m+1}$ of $F$ in a single draw, $\gamma$ is:

$$\gamma = \frac{1}{(m+1)} \frac{b_{m+1}}{{b_{max}}'} \frac{1}{b_{m+1}} = E[\gamma_i] \tag{7.5}$$

□

Note: A more sophisticated version, Correlated Acceptance/Rejection (CAR), of the Naive Correlated Acceptance/Rejection (NCAR) algorithm does not require that one keep the indices of the original accepted sample elements in the original gross sample. In NCAR one reuses elements of the original gross sample in the same order they were originally generated. Hence, one must either keep a vector of length $s_g = |E|$, the original gross sample size, or at least a list of the indices of the accepted elements of the original gross sample.

However, all of the elements of $E$ are exchangeable, so that one can reuse any permutation of the original gross sample. Thus in CAR, whenever an element of $E$, the original gross sample, is to be reused one takes a SRSWOR from $E$. For each such element, one needs two pieces of information: whether it was accepted, and if so what record was sampled. This can done by acceptance/rejection sampling with acceptance probability for the $j'th$ attempted sample from $E = (s_r)/(|E| - j - 1)$, where $s_r$ = the number of remaining elements of the original (accepted) sample. If accepted, a record is chosen (at random) from the remaining members of the original (accepted) sample and $s_r$ is reduced by one.

The advantage of CAR is the reduced storage requirement, since one only needs $|E|$, $b_{max}$ and the accepted elements of the original sample. As in the case of NCAR, a second acceptance/rejection test with acceptance probability of $b_{max}/b'_{max}$ is still needed for reused sample elements.

### 7.2.3 Analysis of Naive CAR (NCAR)

The computational cost of the naive CAR algorithm, cost (NCAR), is proportional to the number of times one has to perform the while loop until a sample of the required size $s$ is constructed. In order to compute this, one will first compute the probability $p$ of accepting a sample each time one goes through the loop. The cost of the algorithm is clearly $O(s/p)$. It is easy to see that:

$$p = (\frac{m}{m+1})(\frac{b_{max}}{b'_{max}}) + (\frac{1}{m+1})(\frac{b_{m+1}}{b'_{max}}) \tag{7.6}$$

The first term arises from acceptance rejection in $F$ and the second from acceptance rejection in $B_{m+1}$. In Figure 7.4 is a plot the value of $p$ with increasing $b_{m+1}$. In this figure one assumes $m = 100$, and $b_{max} = 20, 25, 30$. One can see that $p$ grows with $b_{m+1}$ and achieves its maximum value, 1, when

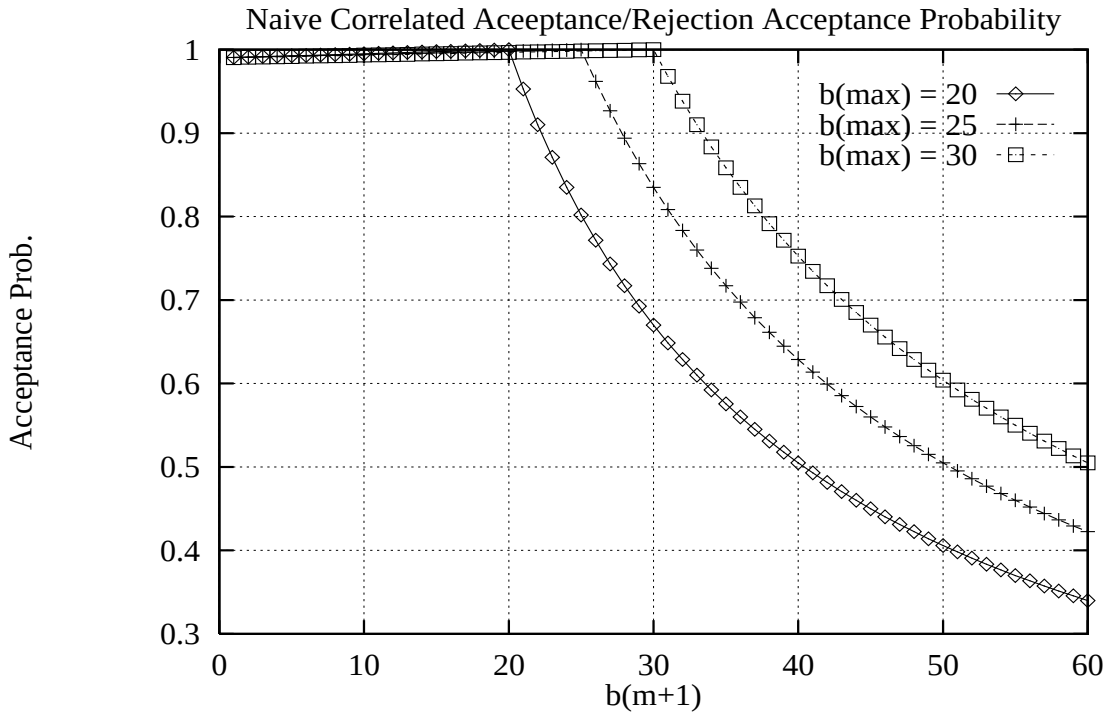



Figure 7.4: Graph of acceptance probability of naive CAR algorithm.

$$b_{m+1} = b_{max} = b'_{max} \tag{7.7}$$

and then decreases when $b_{m+1} > b_{max}$. This can be generalized to the case that $l$ new blocks are inserted. In this case

$$p = (\frac{m}{m+l})(\frac{b_{max}}{b'_{max}}) + (\frac{l}{m+l})(\frac{\bar{b}_l}{b'_{max}}) \tag{7.8}$$

where $b'_{max}$ is the overall maximum blocksize in the augmented file and $\bar{b}_l$ is the average blocksize computed over the $l$ new blocks.

In the Figure 7.5 one has a plot of the cost/sample element with increasing $b_{m+1}$.

## 7.3 Updating a Sample Projection View

Maintaining sample views for projection queries is similar to that of maintaining sample views of selection queries. However, typically, one does not know the cardinality of the projection query result. Hence, one generally cannot employ reservoir techniques.

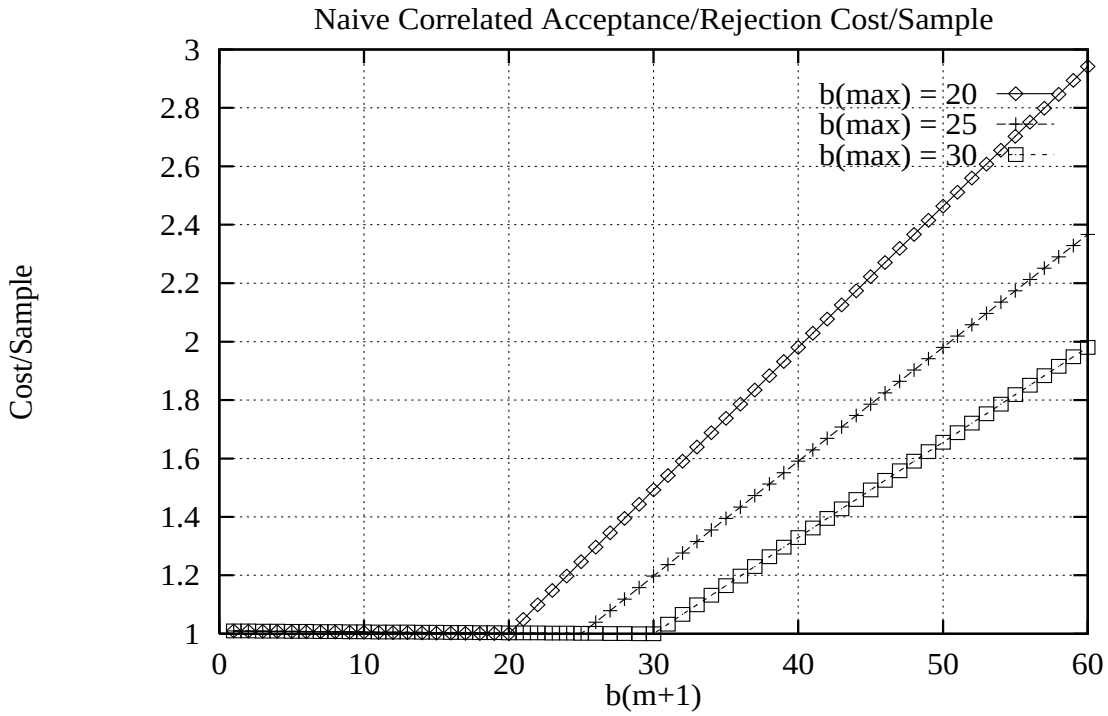



Figure 7.5: Graph of cost of naive CAR algorithm, measured as I/O's per sample element.

One can divide this problem into two cases, those modifications of the base relation which alter the result of the projection query, and those which do not. To do this efficiently typically requires the existence of an index to the base relation on the projection attributes.

For those deletions, insertions, and updates to the base relation which would leave the projection unchanged, one need do nothing. One's sample remains a simple random sample of the projection query on the base relations.

For deletions from the base relation which do not affect elements of the sample projection view, one need do nothing - one will still have a simple random sample. If the deletion were to remove an element of the sample projection view from the projection, then one must remove that element from the sample and obtain a replacement element by resampling the projection (with the methods described in Chapter 6).

In the case where one inserts records to the base relation which alter the result of projection query, then one will need to update the sample view. Observe that this is similar to the update of a sample selection view and the CAR algorithm (described above) can be applied with minor modifications (to account for the projection operator - see Chapter 6).

## 7.4 Updating a Sample Join View

Updating a sample join view is by far the most complicated case for MSV. To facilitate this discussion I review some of the results from Chapter 6 on techniques of sampling from join queries.

Assume that one joins two relations $A$ and $B$ over some join attribute which takes its values from the domain $1, 2, ..., M$. Also assume that the value $i$ for the join attribute appears in $A$ with multiplicity $\mu_i(A)$ and in $B$ with multiplicity $\mu_i(B)$. Let $\mu_{max}(A)$ and $\mu_{max}(B)$ denote the maximum of these multiplicities in the relations $A$ and $B$ respectively.

As I showed in Chapter 6, there are at least two different ways in which a single sample from the results of a join query can be obtained. One can use either of two methods:

**Method 1**: Draw a random tuple, $A_j$, from $A$. If the value of the join attribute for this sample is $i$, one accepts it with probability $\mu_i(B)/\mu_{max}(B)$. If one accepts it, one then joins $A_j$ with some random element of $B \ltimes A_j$.

Equivalently, one can reverse the roles of $A$ and $B$, i.e.:

**Method 2**: Draw a random tuple, $B_j$, from $B$. Assuming a join attribute value $i$, one accepts $B_j$ with probability $\mu_i(A)/\mu_{max}(A)$. If one accepts it, one joins $B_j$ with some random element of $A \ltimes B_j$.

In both cases one repeats the sampling procedure until a sample of size $s$ is obtained. The method of choice depends on join selectivities as well as availability of indices on $A$ and $B$ in order to check the required multiplicities.

Assume that a random sample of size $s$ was obtained using Method 1, (initial sample from $A$) and $k$ new tuples are inserted into a new block of $A$, $\Delta_A$. This is a relatively simple case since one can use the CAR method described earlier with minor modifications. Basically an element which was rejected from the original sample will still be rejected after this insertion. On the other hand, it is possible that elements in the original sample may be replaced by new ones coming from the new block.

One proceeds to sample as follows. One first decides whether to sample from the old relation, $A$, or the new block, $\Delta_A$, exactly in the manner described in the CAR algorithm.

If one decides to sample from the old file, we accept a member of the old sample as described previously. In case one decides to draw a sample from the new block with join attribute value $i$, one accepts it with probability $\mu_i(B)/\mu_{max}(B)$.

Now examine the case where the sample was obtained using Method 1 and $k$ new tuples are inserted into $B$ yielding $B'$. Here it is possible that a sample which was previously rejected should be accepted after the insertions because multiplicities in $B$ may be increased due to the inserted tuples. Since in the CAR method one does not keep track of rejected elements but only the cardinality of the gross sample size (or the index set of accepted records) one cannot update the sample properly.

As one can see from the discussion above, one will need to store the rejected elements of the gross sample from $A$ in addition to the original net sample in order to efficiently update sample views of $A \bowtie B$. The expected amount of storage needed will thus be inversely proportional to the acceptance rate for the sampling method chosen.

Essentially, I propose to rerun the original sampling from $A$, with the same random numbers, but with adjusted acceptance/rejection probabilities. This would nominally require that one store the entire original gross sample, together with the random number used to accept/reject each element. One then reruns the sampling, testing the original random numbers against the new acceptance probabilities: $\mu_i(B')/\mu_{max}(B')$. Observe that if the acceptance probability for a sample element has increased, then previously accepted elements will again be accepted. Previously accepted elements whose acceptance probability has decreased will be retained with probability equal to the ratio of the new and old acceptance probabilities.

Of course a sample from $A$ is not sufficient, one must also complete the join. For newly accepted elements of the sample, one proceeds as before in Method 1. For retained elements, $A_j$, from the original sample, one must decide whether one can keep the same matching element from $B$ (now $B'$). For each retained $A_j$, this amounts to maintaining a materialized sample view of size one from the relation $B' \ltimes A_j$. This is essentially the same as a sample from a selection query on $B'$ where the predicate is $B'.x = A_j.x$, and $x$ is the join attribute. Hence, one can use my previous results on maintenance of materialized views of sample selection queries for each semi-join query.

One can halt when one obtains a sufficiently large sample. If one exhausts the original gross sample without obtaining the desired net sample size, one resumes sampling from the table $A$.

It is easy to see that this will generate the desired SRS from $A \bowtie B'$.

A small improvement is possible. One need not store the entire tuples from $A$ for originally rejected elements of the gross sample. It will suffice to store the keys from which one can retrieve the tuples if needed.

## 7.5 Conclusions

Because sample views are typically quite small compared to the underlying base relations, materialized base relations will be an attractive strategy for supporting sample views under heavy retrieval traffic. Hence it is important to develop efficient methods of maintaining materialized sample views when the base relations are updated.

| Type of Update | Type of Sample View | | | | | | |
|---|---|---|---|---|---|---|---|
| | FB File $\psi(F)$ | VB File $\psi(F)$ | Select $\psi(\sigma_f(A))$ | Project $\psi(\pi_a(A))$ | Intersect $\psi(A \cap B)$ | Union $\psi(A \cup B)$ | Join $\psi(A \bowtie B)$ |
| Delete ($r \notin \Psi$) | no-op | no-op | no-op | no-op | no-op | no-op | no-op |
| Delete ($r \in \Psi$) | SRS | SRS | SRS | SRS | SRS | SRS | SRS |
| Insert ($r \notin Q$) | NA | NA | no-op | no-op | no-op | no-op | no-op |
| Insert ($r \in Q$) $\lvert Q \rvert$ known | RES,2FS | 2FS,RES | 2FS,RES | 2FS,RES | 2FS,RES | 2FS,RES | 2FS,RES |
| Insert ($r \in Q$) $\lvert Q \rvert$ unknown | CAR | CAR | CAR | CAR | CAR | CAR | CAR |
| Update ($r \notin Q \longrightarrow r \notin Q$) | NA | NA | no-op | | | | |
| Update ($r \notin Q \longrightarrow r \in Q$) | NA | NA | same as insert $r \in Q$ | | | | |
| Update ($r \in \Psi \longrightarrow r \in \Psi$) | update sample | | | | | | |
| Update ($r \in Q \longrightarrow r \notin Q$) | same as delete $r \in Q$ | | | | | | |

Table 7.2: Various results in this chapter. $\Psi$ (used as a noun) denotes the sample, $Q$ denotes the query result which was sampled, $f$ denotes the selection predicate, $a$ denotes the projection attributes, NA denotes *not applicable*, no-op denotes do nothing, FB denotes fixed blocking, VB denotes variable blocking, 2FS denotes 2 Files Sampling strategy, RES denotes Reservoir Sampling strategy (note RES requires knowledge of the cardinality of $Q$).

In this chapter I have shown how to efficiently maintain materialized sample views. These methods are based on classic methods of materialized view maintenance combined with either reservoir sampling, two-file sampling, or correlated acceptance-rejection sampling. These methods are especially suited to systems for which queries against the sample views are common.

Correlated Acceptance/Rejection is particularly attractive when one does not know the cardinality of the original relational query which was sampled, but one does know the number of blocks used to store the base relation. An example would be if the original sample view involved sampling from a selection query from a single relation with a non-indexed selection predicate and the view was created by interchanging the sampling and selection operators, i.e., sampling first.

I have discussed maintaining materialized sample views from base relations, projections and joins. Table 7.2 summarizes the results of this chapter, and some related results.

For certain purposes - e.g., auditing - one may not want to maintain a sample view, unless one could be certain of its security. If the purpose of obtaining a sample is merely to compute some simple aggregate statistics, then one may be better off maintaining a statistical database abstract consisting of statistics over various database partitions, as described by Rowe [Row85]. Statistical summaries of the database tend to be more concise and accurate than samples. Samples, however, are more versatile.

# Chapter 8

# Conclusions

## 8.1 Summary

In thesis I have described novel algorithms for implementing random sampling queries of relational databases and analyzed their performance.

Like selection, sampling produces an output which is much smaller than its input. Hence, there is considerable interest in pushing sampling operations down in the query processing plan tree toward the leaves (reading of base relations). This has generated much of the agenda of this thesis:

- basic sampling techniques from files
- sampling from various access methods used to store base relations: $B^+$ trees, hash files, spatial data structures (quadtrees and R-trees)
- sampling from relational operators,
- maintenance of materialized sample views.

I began by discussing the motivation for including sampling operators in database management systems. Applications of sampling in databases include: financial auditing, statistical database security, query optimization, quality control, epidemiology, nuclear materials inventory audits, etc.

Present practice with regard to sampling queries is to implement the sampling function outside the database, e.g., using reservoir sampling techniques. I have shown that there are substantial gains to be had in efficiency by embedding the sampling operators within the DBMS. Gains from supporting sampling within the DBMS include:

- improved efficiency in answering sampling queries,
- improved efficiency in providing approximate answers to aggregate queries,
- use of sampling to estimate parameters used by query optimizers, and to permit dynamic (adaptive) query optimization.

I have reviewed basic sampling methods used to construct the database sampling algorithms in this thesis: The most important were:

- Acceptance/Rejection (A/R) sampling - which I have used throughout this thesis to sample from variably blocked files, hash files, $B^+$ tree files, etc.
- Sequential (reservoir) sampling - for sampling from files of unknown size via sequential scans (e.g. sampling as a relation or intermediate result is created).
- Wong & Easton's weighted sampling from ranked trees - which forms the basis of algorithms for sampling from ranked $B^+$ trees.

I have discussed the related literature on sampling from databases, especially the extensive efforts to use sampling to estimate intermediate result sizes, and predicate/join/projection selectivities for use by query optimizers in choosing query processing strategies. A number of authors (e.g., Hou, Ozsoyoglu, Naughton, Seshadri) have argued that various types of clustered sampling techniques can be used advantageously for result size (or selectivity) estimation, by clever estimator design. The work of Dorothy Denning on the use of random sampling to provide statistical database security was reviewed.

For $B^+$ trees I have shown how to use acceptance/rejection sampling sampling to sample from $B^+$ trees without requiring the additional maintenance tree maintenance required by techniques which sample from ranked $B^+$ trees. I considered both naive and (the preferred) early abort algorithms. I also showed how to construct batch sampling algorithms for $B^+$ trees, analogous to batch searching algorithms. Recently, Antoshenkov used a hybrid of my algorithm and the classic sampling from ranked $B^+$ trees to construct an algorithm with modest update costs and much more efficient sampling behavior.

I began the discussion of sampling from hash files by considering an acceptance/rejection algorithm for open address hashing (not widely used in DBMSs). This algorithm can be used for any variably blocked file, and has been subsequently adapted by Dewitt, et al. for extent-based files.

For linear hashed files both one-file and two-file sampling methods, based on acceptance/rejection were considered. Two-file sampling methods proved more efficient. For extended hash files I showed that double A/R sampling was more efficient than cell A/R sampling. Batch sampling methods were yet more efficient.

Turning to sampling from spatial access methods, I have shown how two parameters, *coverage* (the proportion of the region covered by polygons) and expected *stabbing number* (the number of polygons covering a point) shape the choice of spatial sampling methods. High coverage dictates the generation of random points in space, followed by point inclusion queries of the spatial data structure, whereas low coverage favors directly sampling from the spatial data structure. I have shown how my acceptance/rejection $B^+$ tree sampling techniques can be extended to quadtrees (which are not uniform height) and R-trees (which are not disjoint). Similarly, ranked $B^+$ tree sampling techniques can be extended to quadtrees and R-trees to yield efficient sampling algorithms (at the price of increased maintenance). I found that R-trees impose serious performance penalties for sampling operations. More efficient batch sampling algorithms and reservoir sampling algorithms were briefly discussed for cases in which a major fraction of the file must be read.

Because the result of a sampling operator are typically so much smaller than its input(s), one wants to push the sampling operator down the query processing plan tree toward the base relations. However, sampling and relational operators do not (generally) simply

commute (distribute). I have shown how to obtain simple random samples from simple relational queries consisting of individual relational operators: such as select, project, join, intersect, union.

Selection commutes with sampling operators (up to sample size), whereas the other relational operators require modification (e.g., acceptance/rejection sampling) to assure uniform inclusion probabilities. For the binary relational operators, the sampling operator is generally pushed down one branch of the operator tree (e.g., intersect, join, difference), and an acceptance/rejection test may be added to the relational operator (project,join). I have shown how to sample from intermixed cascades of select, join, intersect, and set difference operators.

Complex unions, i.e., unions of arbitrary relational expressions, are difficult to sample from, because the standard algorithms requires the cardinality of each operand. I have described a *domain sampling* method which permits the sampling from complex unions without knowledge of the operand cardinalities. It requires that the underlying key domain be a finite set, preferably densely populated with actual keys.

I have also described an algorithm for sampling from select-project-join (SPJ) queries. The algorithm augments elements of a sample of the join with the sizes of the corresponding projection equivalence classes of the underlying join. These projection equivalence class sizes are then used in an acceptance/rejection algorithm to correct for the impact of the projection operator on the inclusion probabilities.

The various algorithms described for sampling from relational expressions come in both iterative (tuple-substitution) and bottom-up (batch) versions. The iterative algorithms offering exact sample sizes and easy adaptability to sequential (adaptive) sampling procedures, while the bottom-up algorithms offer better performance in many cases. Even so, it may still be the case that sampling from complex relational expression will require partial (or occasionally even full) evaluation of the underlying relational expression prior to sampling.

Finally, I have shown how to maintain materialized sample views while the base relations are updated. This was done by combining sampling algorithms (reservoir sampling, two-file sampling) with classical methods for updating materialized views to maintain materialized sample views with minimal effort. Such methods are well suited to support heavily used sample views.

To conclude, I have described the basic algorithms needed to efficiently support simple random sampling queries of relational databases and analyzed their performance.

## 8.2 Future Work

Most of my work has been on simple random sampling, however extensions to stratified random sampling are straightforward. While I have touched on weighted random sampling, there is much more work to be done in this area.

It now appears that one of the major applications of sampling in DBMSs will be to estimate various parameters needed for query optimization: predicate and join selectivities, and the cardinalities of intermediate results. This area has begun to be explored in the work of Ozsoyoglu, Naughton, Seshadri, Antoshenkov and Seppi. I expect that Seppi's work on decision theoretic approaches to the use of sampling in query optimization appears will be prove to be the most suitable basis for further developments in this area.

*The most important work to be done at this point would be to construct a prototype implementation of a system for answering sampling queries.* This will require significant effort, at modifying access routines, writing sampling codes, and modifying a query optimizer. Incorporating sampling operators increases the complexity of the query optimization problem, suggesting that rule-based query optimizers may prove an attractive implementation strategy.

# Bibliography


[AD83] Rakesh Agrawal and David J. Dewitt. Updating Hypothetical Databases. *Information Processing Letters*, 16:145–146, April 1983.

[AIoCPA83] Statistical Sampling Subcommittee American Institute of Certified Public Accountants. *Audit sampling.* American Institute of Certified Public Accountants, 1983.

[AIoCPA92] Auditing Standards Board American Institute of Certified Public Accountants. *Codification of statements on auditing standards.* published for the American Institute of Certified Public Accountants by Commerce Clearing House, 1992.

[AL80] M. Adiba and B. Lindsay. Database Snapshots. In C. Zaniolo, editor, *Proceedings of the 6th International Conference on Very Large Databases (VLDB)*, pages 53–60, May 1980.

[AL81] Alvin A. Arens and James K. Loebbecke. *Applications of statistical sampling to auditing.* Prentice-Hall, 1981.

[Ald89] David Aldous. *Probablity Approximations via the Poisson Clumping Heuristic.* Springer-Verlag, 1989.

[ALS88] Abraham D. Akresh, Jame K. Loebbecke, and William R. Scott. Audit Approaches and Techniques (Chapter 2). In A. Rashad Abdel-khalik and Ira Solomon, editors, *Research Opportunities in Auditing, The Second Decade*, pages 13–54. American Accounting Associaton: Auditing Section, 1988.

[Ant92] Gennady Antoshenkov. Random Sampling from Pseudo-Ranked $B^+$ trees. In *Proceedings of the 19th International Conference on Very Large Databases (VLDB)*, pages 375–382, August 1992.

[Ant93] Gennady Antoshenkov. Dynamic Query Optimization in Rdb/VMS. In *Proceedings of the IEEE Data Engineering Conference*, pages 538–547, April 1993.

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

[ASW87] M. Astrahan, M. Schkolnick, and K.-Y. Whang. Approximating the number of unique values of an attribute without sorting. *Information Systems*, 12(1):11–15, 1987.

[BAG87] J.A. Baldwin, E.D. Acheson, and W.J. Graham, editors. *Textbook of medical record linkage*. Oxford University Press, 1987.

[Bai81] Andrew D. Jr. Bailey. *Statistical auditing : review, concepts, and problems*. Harcourt Brace Jovanovich, 1981.

[BB84] B. Brown, Morton and Judith Bromberg. An Efficient Two-Stage Procedure for Generating Random Variates from the Multinomial Distribution. *The Americal Statistician*, 38(3):216–218, August 1984.

[BB88] W.M. Bowen and C.A. Bennett. Statistical Methods for Nuclear Material Management. Technical Report PNL-5849, NUREG/CR-4604, ISBN 0-87079-588-0, Battelle Pacific Northwest Labs, Dec. 1988.

[BB92] Kenneth H. Bacon and Lee Berton. Ernst to pay $400 million over audit of 4 big thrifts; firm settles U.S. charges it inadequately audited institutions that failed. (Ernst and Young). *Wall Street Journal*, page A3, Nov. 24 1992.

[BBM85] Carl Blumstein, Susan Buller, and C. Bart McGuire. Analysis of the San Diego Gas and Electric Company's 1982 Commercial Energy Use Survey. Technical Report UER-144, Universitywide Energy Research Group, University of California, Feb. 1985.

[BC79] O. Peter Buneman and Eric K. Clemons. Efficiently Monitoring Relational Databases. *ACM Transactions on Database Systems*, 4(3):368–382, Sept. 1979.

[BCL86] Jose A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. In *Proceedings of the 12th International Conference on Very Large Databases (VLDB)*, pages 457–466, August 1986.

[BD80] N.E. Brelsow and N.E. Day. *Statistical Methods in Cancer Research*, volume Volume 1, The Analysis of Case Control Studies. World Health Org., Intl. Agency for Research on Cancer, 1980. (distributed by Oxford Univ. Press).

[BF93] J. Bunge and M. Fitzpatrick. Estimating the Number of Species: A Review. *Journal of the American Statistical Association*, 88(421):364–373, March 1993.

[BFOS84] L. Breiman, J.H. Freidman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Wadsworth & Brooks/Cole, Pacific Grove, CA, 1984.

[BG91] Eugene M. Burns and Miriam L. Goldberg. Commercial Buildings Characteristics 1989. Technical Report DOE/EIA-0246(89), U.S. Dept. of Energy, Energy Information Administration, July 1991.

[BGH88] Andre D. Jr. Bailey, Lynford E. Graham, and James V. Hansen. Technological Development and EDP (Chapter 3). In A. Rashad Abdel-khalik and Ira Solomon, editors, *Research Opportunities in Auditing, The Second Decade*, pages 13–54. American Accounting Associaton: Auditing Section, 1988.

[BK75] B.T. Bennett and V.J. Kruskal. LRU Stack Processing. *IBM Journal of Research and Development*, 19(4):353–357, July 1975.

[BLT86] Jose A. Blakeley, Per-Åke Larson, and Frank W. Tompa. Efficiently Updating Materialized Views. In C. Zaniolo, editor, *ACM SIGMOD International Conference on the Management of Data*, pages 61–71, May 1986.

[Car33] Lewis A. Carman. The Efficacy of Tests. *The American Accountant*, pages 360–366, Dec. 1933.

[Car75] A.F. Cardenas. Analysis and Performance of Inverted Database Structures. *Communications of the ACM*, 18(5):253–263, May 1975.

[Car92] Richard Carlson. Private communication. (Responsible for the Los Alamos Nuclear Material Control System), Sept. 1992.

[CDH87] Fredrick A. Connell, Paula Diehr, and L. Gary Hart. *The Use of Large Databases in Health Care Studies*, volume 8, pages 51–74. Annual Reviews Inc., 1987.

[CDRS86] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugne J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th International Conference on Very Large Databases (VLDB)*. Morgan Kaufmann, 1986.

[Che91] P.R. Cheon, M.J.; Philipoom. Using Regression to Compromise Statistical Databases: a Modification of the Attribute Correlation Modeling Approach. *Journal of Database Administration*, 2(2):15–21, Spring 1991.

[Chr84a] Stavros Christodoulakis. Estimating Block Selectivities. *Information Systems*, 9(1):69–79, 1984.

[Chr84b] Stavros Christodoulakis. Implications of Certain Assumptions Database Performance Evaluation. *ACM Transactions on Database Systems*, 9(2):163–186, June 1984.

[Chu89] Pai-Cheng Chu. Database Access Path Selection: A Two Step Approach. *Information Systems*, 14(5):385–392, 1989.

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

[CW91] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In G. Lohman, A. Sernadas, and R. Camps, editors, *Proceedings of the 17th International Conference on Very Large Databases (VLDB)*, pages 577–589, September 1991.

[Dat90] Christopher J. Date. *An Introduction to Database Systems*. Addison-Wesley, 5th edition edition, 1990.

[DDS79] D.E. Denning, P.J. Denning, and M.D. Schwartz. The Tracker: A Threat to Statistical Database Security. *ACM Transactions on Database Systems*, 4(1):79–96, March 1979.

[Den80] Dorothy E. Denning. Secure Statistical Databases with Random Sample Queries. *ACM Transactions on Database Systems*, 5(3):291–315, Sept. 1980.

[Dev86] Luc Devroye. *Lecture Notes on Bucket Algorithms.* Birkhäuser, Boston, 1986.

[Dev91] Luc Devroye. Coupled Samples in Simulation. *Operations Research*, 38(1):115–126, January-February 1991.

[DNS91a] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. A Comparison of Non-equijoin Algorithms. In *Proceedings of the 18th International Conference on Very Large Databases (VLDB)*, pages 443–452, August 1991.

[DNS91b] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. Parallel Sorting on a Shared-Nothing Architecture Using Probabilistic Splitting. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, pages 280–291, December 1991.

[DNSS92] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *Proceedings of the 19th International Conference on Very Large Databases (VLDB)*, pages 27–40, August 1992.

[DS80] D.E. Denning and J. Schlorer. A Fast Procedure for Finding a Tracker in a Statistical Database. *ACM Transactions on Database Systems*, 5(1):88–102, March 1980.

[Dun91] S. Duncan, G.T.; Mukherjee. Microdata Disclosure Limitation in Statistical Databases: Query Size and Random Sample Query Control. In *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 278–87. IEEE Computer Society Press, 20-22 May 1991.

[dV86] Pieter G. de Vries. *Sampling Theory for Forest Inventory.* Springer-Verlag, 1986.

[Efr82] Bradley Efron. *The Jackknife, the Bootstrap and Other Resampling Plans.* Society for Industrial and Applied Mathematics, 1982.

[EN82] Jarmo Ernvall and Olli Nevalainen. An Algorithm for Unbiased Random Sampling. *The Computer Journal*, 25(1):45–47, 1982.

[Fla85] P. Flajolet. Approximate counting: a detailed analysis. *BIT*, 25(1):113–34, 1985.

[FM83] P. Flajolet and G.N. Martin. Probabilistic counting. In *24th Annual Symposium on Foundations of Computer Science*, pages 76–82, 1983.

[FM85] P. Flajolet and G.N. Martin. Probabilistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31(2):182–209, Oct. 1985.

[FMR62] C.T. Fan, M.E. Muller, and I. Rezucha. Development of Sampling Plans by Using Sequential (Item by Item) Selection Techniques and Digital Computers. *Journal of the American Statistical Association*, 57:387–402, June 1962.

[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.

[GC86] Dan M. Guy and D.R. Carmichael. *Audit sampling : an introduction to statistical sampling in auditing.* Wiley, 1986.

[Gea89] Donald Guthrie and et al. Statistical Models and Analysis on Auditing, Panel on Nonstandard Mixture of Distrubitions. *Statistical Science*, 4(1):2–33, 1989.

[GGH79] P.T. Good, J. Griffith, and A.G. Hamlin. A statistical approach to the verification of large stocks of fissile material in diverse forms. In *Nuclear Safeguards Technology 1978*, pages 361–9. IAEA Vienna, Austria, 1979.

[Gho86] S. Ghosh. SIAM: Statistics Information Access Method. In *Proceedings of the Third International Workshop on Statistical and Scientific Database Management*, pages 286–293. EUROSTAT, Luxembourg, 1986.

[Gho88] Sakti P. Ghosh. SIAM: Statistics Information Access Method. *Information Systems*, 13(4):359–368, 1988.

[Goo49] L.A. Goodman. On the Estimation of the Number of Classes in a Population. *Annals of Mathematical Statistics*, 20:572–579, 1949.

[Gut84] A. Guttman. R-trees: A Dynamic Indes Structure for Spatial Searching. In *ACM SIGMOD International Conference on the Management of Data*, pages 47–57, June 1984.

[Guy81] Dan M. Guy. *An introduction to statistical sampling in auditing.* Wiley, 1981.

[Hal88] Peter Hall. *Introduction to the Theory of Coverage Processes.* John Wiley & Sons, 1988.

[Han87a] Eric N. Hanson. *Efficient Support for Rules and Derived Objects in Relational Database Systems.* PhD thesis, U.C. Berkeley, August 1987.

[Han87b] Eric N. Hanson. A Performance Analysis of View Materializations Strategies. In *ACM SIGMOD International Conference on the Management of Data*, pages 440–453, May 1987.

[Hin87] John P. Hiniker. The selection of returns for audit by the IRS. In *American Statistical Association Proceedings of the Survey Research Methods*, pages 294–299. American Statistical Association, Alexandria, VA, 1987.

[HO91] Wen-Chi Hou and Gultekin Ozsoyoglu. Statistical Estimators for Aggregate Relational Algebra Queries. *ACM Transactions on Database Systems*, 16(4):600–654, Dec. 1991.

[HOD91] Wen-Chi Hou, Gultekin Ozsoyoglu, and Erdogan Dogdu. Error-Constrained COUNT Query Evaluation in Relational Databases. In *ACM SIGMOD International Conference on the Management of Data*, pages 278–287, June 1991.

[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.

[HRA62] Henry P. Hill, Joseph L. Roth, and Herbert Arkin. *Sampling in auditing; a simplified guide and statistical tables.* Ronald Press Co., 1962.

[HS92a] Peter J. Haas and Arun N. Swami. Sequential Sampling Procedures for Query Size Estimation. Technical Report RJ 8558, IBM Alamaden, January 1992.

[HS92b] Peter J. Haas and Arun N. Swami. Sequential Sampling Procedures for Query Size Estimation. In *ACM SIGMOD International Conference on the Management of Data*, pages 341–350, June 1992.

[IAE80] IAEA. *IAEA Technical Safeguards Technical Manual, Part F, Statistical Concepts and Techniques*, second revised edition, 1980. Available from NTIS.

[IPA75] *Studies on statistical methodology in auditing, Proceedings of the 10th Conference on Accounting Research.* Institute of Professional Accounting, Graduate School of Business, University of Chicago, 1975.

[Jae88] J.L. Jaech. Training Manual on Statistical Methods for Nuclear Material Management. Technical Report PNL-5855, Battelle Pacific Northwest Labs, April 1988.

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

[Kin86] William R. Kinney, editor. *Fifty years of statistical auditing.* Garland Publishers, 1986.

[Knu69] Donald Ervin Knuth. *The Art of Computer Programming: Vol. 2, Seminumerical Algorithms.* Addison-Wesley, 1969.

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

[Knu81] Donald Ervin Knuth. *The Art of Computer Programming: Vol. 2, Seminumerical Algorithms.* Addison-Wesley, 2nd edition, 1981.

[Lar82] Per-Ake Larson. Performance Analysis of Linear Hashing with Partial Expansions. *ACM Transactions on Database Systems*, 7(4):566–587, Dec. 1982.

[Lar92] Erik Larson. *The Naked Consumer : How Our Private Lives Become Public Commodities*. H. Holt, 1992.

[LDJ89] Sheau-Dong Lang, J.R. Driscoll, and J.H. Jou. A unified analysis of batched searching of sequential and tree-structured files. *ACM Transactions on Database Systems*, 14(4):604–18, Dec. 1989.

[LHMP86] B. Lindsay, L. Hass, C. Mohan, and P. Pirahesh, H. Wilms. A Snapshot Differential Refresh Algorithm. In C. Zanioilo, editor, *ACM SIGMOD International Conference on the Management of Data*, pages 53–60, June 1986.

[Lit80] W. Litwin. Linear hashing: a New Tool for File and Table Addressing. In *Proceedings of the 6th International Conference on Very Large Databases (VLDB)*, pages 212–223, 1980.

[LM90] S.D. Lang and Y. Manolopoulos. Efficient expressions for completely and partly unsuccessful batched search of tree-structured files. *IEEE Transactions on Software Engineering*, 16(12):1433–5, Dec. 1990.

[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.

[LN90] R. Lipton and J.F. Naughton. Query Size Estimation Through Adaptive Sampling. In *Proceedings of the ACM Conference on Principles of Database Systems*, pages 40–46, April 1990.

[LNS90] R. Lipton, J.F. Naughton, and D.A. Schneider. Practical Selectivity Estimation Through Adaptive Sampling. In *ACM SIGMOD International Conference on the Management of Data*, pages 1–11, May 1990.

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

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

[Mai91] Mark H. Maier. *The Data Game, Controversies in Social Science Statistics*. M.E. Sharpe, 1991.

[MB83] A.I. McLeod and D.R. Bellhouse. A Convenient Algorithm for Drawing a Simple Random Sample. *Applied Statistics*, 32(2):182–184, 1983.

[McR74] T. W. McRae. *Statistical sampling for audit and control*. Wiley, 1974.

[MD88] M. Muralikrishan and David J. DeWitt. Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries. In *ACM SIGMOD International Conference on the Management of Data*, pages 28–36, 1988.

[Mon85] Douglas C. Montgomery. *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.

[NL75] John Neter and James K. Loebbecke. *Behavior of Major Statistical Estimators in Sampling Accounting Populations: An Empirical Study.* American Institute of Certified Public Accountants, 1975.

[NS90] Jeffrey F. Naughton and S. Seshadri. On Estimating the Size of Projections. In S. Abiteboul and P.C. Kanellakis, editors, *Proceeedings of ICDT 90, Third International Conference on Database Theory*, pages 499–513. Springer-Verlag, Dec. 1990.

[OR86] Frank Olken and Doron Rotem. Simple Random Sampling from Relational Databases. In *Proceedings of the 12th 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)*, pages 269–277. Morgan Kaufman, August 1989.

[OR90] Frank Olken and Doron Rotem. Random Sampling from Database Files: A Survey. In Z. Michalewicz, editor, *Statistical and Scientific Database Management, Proceedings of the Fifth International Conference*, pages 92–111. Springer-Verlag, April 1990.

[OR92a] Frank Olken and Doron Rotem. Maintenance of Materialized Views of Sampling Queries. In *Proceedings of the IEEE Data Engineering Conference*, pages 632–641, Feb. 1992.

[OR92b] Frank Olken and Doron Rotem. Sampling from Spatial Databases. Technical Report LBL-32024, Lawrence Berkeley Laboratory, July 1992.

[OR93] Frank Olken and Doron Rotem. Sampling from Spatial Databases. In *Proceedings of the IEEE Data Engineering Conference*, pages 199–208, April 1993.

[ORX90] Frank Olken, Doron Rotem, and Ping Xu. Random Sampling from Hash Files. In *ACM SIGMOD International Conference on the Management of Data*, pages 375–386, May 1990.

[Pal85] P. Palvia. Expressions for Batched Searching of Sequential and Hierarchical Files. *ACM Transactions on Database Systems*, 10(1):97–106, March 1985.

[Par90] R.W. Parrish, C.P.; Mensing. A nuclear material sampling plan. In *Proceedings of the Institute of Nuclear Materials Management (INMM) Conference*, July 1990.

[Ped93] Stephanie Pedersen. Private communication. Ms. Pedersen is head of the LBL internal audit department., March 1993.

[PS87] Micahel A. Palley and Jeffrey S. Simonoff. The Use of Regression Methodology for the Compromise of Confidential Information in Statistical Databases. *ACM Transactions on Database Systems*, 12(4):593–608, Dec. 1987.

[PSC84] Gregory Piatetsky-Shapiro and Charles Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *ACM SIGMOD International Conference on the Management of Data*, pages 256–275, June 1984.

[Rob78] Donald M. Roberts. *Statistical auditing.* American Institute of Certified Public Accountants, 1978.

[Ros89] Paul R. Rosenbaum. Optimal Matching for Observational Studies. *Journal of the American Statistical Association*, pages 1024–1032, Dec. 1989.

[Ros91] Paul R. Rosenbaum. Sampling the Leaves of a Tree with Equal Probabilities. *submitted to Journal of the American Statistical Association*, Dec. 1991.

[Row83] Neil Rowe. *Rule-based Statistical Calculations on a Database Abstract.* PhD thesis, Stanford, May 1983.

[Row85] Neil Rowe. Anti-Sampling for Estimation: An Overview. *IEEE Transactions on Software Engineering*, SE-11(10):1081–1091, October 1985.

[Row88] Neil Rowe. Absolute Bounds on Set Intersection and Union Sizes. *IEEE Transactions on Software Engineering*, SE-14(7):1033–1047, July 1988.

[Rub81] Reuven Y. Rubinstein. *Simulation and the Monte Carlo Method.* John Wiley and Sons, 1981.

[Sam84] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 6(2):187–260, June 1984.

[Sam89a] H. Samet. *Applications of Spatial Data Structures.* Addison-Wesley, 1989.

[Sam89b] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1989.

[SC92] V. Srinivasan and Michael J. Carey. Compensation-Based On-Line Query Processing. In Michael Stonebraker, editor, *ACM SIGMOD International Conference on the Management of Data*, pages 331–340, June 1992.

[Sch82] James J. Schlesselman. *Case-Control Studies - Design, Conduct, Analysis.* Oxford Univ. Press, 1982.

[Sep90] Kevin D. Seppi. *A Bayesian Approach to Selected Database Issues.* PhD thesis, Univ. of Texas at Austin, August 1990.

[Ser82] J. Serra. *Image Analysis and Mathematical Morphology.* Academic Press, 1982.

[Ses92] S. Seshadri. *Probabilistic Methods in Query Processing.* PhD thesis, Univ. of Wisconsin, 1992.

[SF90] Arie Segev and Weiping Fang. Currency-based Updates to Distributed Materialized Views. In *Proceedings of the IEEE Data Engineering Conference*, pages 512–520, Feb. 1990.

[SF91] Arie Segev and Weiping Fang. Optimal Update Policies for Distributed Materialized Views. *Management Science*, 37(7):851–870, July 1991.

[SG76] B. Shneiderman and V. Goodman. Batch Searching of Sequential and Hierarchical Files. *ACM Transactions on Database Systems*, 1(3):268–275, Sept. 1976.

[SI84] Oded Shmueli and Alon Itai. Maintenance of Views. In *ACM SIGMOD International Conference on the Management of Data*, pages 244–255, June 1984.

[SL76] D.G. Severance and G.M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Transactions on Database Systems*, 1(3):256–267, September 1976.

[SL88] J. Srivastava and V.L. Lum. A Tree Based Access Method (TBSAM) for Fast Processing of Aggregate Queries. In *Proceedings of the 4th International Conference on Data Engineering*, pages 504–510. IEEE Computer Society, 1988.

[SM92] Kevin D. Seppi and Carl J. Morris. A Bayesian Approach to Database Query Optimization. *ORSA Journal of Computing*, accepted 1992.

[SN91] S. Seshadri and Jeffrey Naughton. Sampling Issues in Parallel Database Systems. In *Proceedings of the Conference on Extending Database Technology*, pages 328–343. Springer-Verlag, March 1991.

[SP89] Arie Segev and Jooseok Park. Updating Distributed Materialized Views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.

[SS86] Kenneth W. Stringer and Trevor R. Stewart. *Statistical techniques for analytical review in auditing.* Wiley, 1986.

[SSL+83] Michael Stonebraker, Heidi Stettner, Nadene Lynn, Joseph Kalash, and Antonin Guttman. Document Processing in a Relational Database System. *ACM Transactions on Office Information Systems*, 1(2):143–158, April 1983.

[Sta93] Amy Starr. Private communication. A staff member at Marketing Systems Group, Jan. 1993.

[Sto75] Michael Stonebraker. Implementation of Integrity Constraints and Views by Query Modification. In *ACM SIGMOD International Conference on the Management of Data*, pages 65–78, 1975.

[TB88] F.W. Tompa and J.A. Blakely. Maintaining Materialized Views Without Accessing Base Data. *Information Systems*, 13(4):393–406, 1988.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume vol. 1. Computer Science Press, 1988.

[Van50] Lawrence Lee Vance. *Scientific method for auditing; applications of statistical sampling theory to auditing procedure*. University of California Press, 1950.

[Van76] Robert W. Vanasse. *Statistical sampling for auditing and accounting decisions: a simulation*. McGraw-Hill, 2nd edition, 1976.

[Vit84] Jeffrey Scott Vitter. Faster Methods of Random Sampling. *Communications of the ACM*, 27(7):703–718, July 1984.

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

[VL88] Miklos A. Vasarhelyi and Thomas W. Lin. *Advanced auditing : fundamentals of EDP and statistical audit technology*. Addison-Wesley, 1988.

[Wal47] Abraham Wald. *Sequential Analysis*. John Wiley & Sons, 1947.

[Wal77] Alastair J. Walker. An Efficient Method for Generating Discrete Random Variables with General Distributions. *ACM Transactions on Mathematical Software*, 3(3):253–256, September 1977.

[WE80] C.K. Wong and M.C. Easton. An Efficient Method for Weighted Sampling without Replacement. *SIAM Journal on Computing*, 9(1):111–113, February 1980.

[Wil84] Dan Willard. Sampling Algorithms for Differential Batch Retrieval Problems (Extended Abstract). In *Proceedings ICALP-84*. Springer-Verlag, 1984.

[Wil91] Dan E. Willard. Optimal Sample Cost Residues for Differential Database Batch Query Problems. *Journal of the ACM*, 38(1):104–119, January 1991.

[WS83] John Woodfill and Michael Stonebraker. An Implementation of Hypothetical Relations. In *Proceedings of the 9th International Conference on Very Large Databases (VLDB)*, pages 157–166, 1983.

[WVZT90] Kyu-young Whang, B.T. Vander-Zanden, and H.M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 15(2):208–29, June 1990.

[WZB+93] Ouri Wolfson, Weining Zhang, Harish Butani, Akira Kawaguchi, and Kui Mok. A Methodology for Evaluating Parall Graph Algorithms and It Application to Single Source Reachability. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, pages 243–250, Jan. 1993.

[Xu89] Ping Xu. Sampling from $B^+$ Trees and Hash Files. Master's thesis, San Francisco State Univ., Dec. 1989.

[Yao77] S. Bing Yao. Approximating the Number of Accesses in Database Organizations. *Communications of the ACM*, 20(4):260–261, April 1977.

[ZsSVB89] J.H.M. Zwetsloot-schonk, P. Snitker, J.P. Vanderbroucke, and A.R. Bakker. Using hospital informations system for clinical epidemiolgical research. *Medical Informatics*, 14(1):53–62, 1989.

[ZsvSS+91] J.H.M. Zwetsloot-schonk, W.A.H.J. van Stiphout, P. Snitker, L.A. van Es, and J.P. Vanderbroucke. How to approach a hospital informations system as a sampling frame. *Medical Informatics*, 16(3):287–298, 1991.