Title

PHYSICAL DATABASE SUPPORT FOR SCIENTIFIC AND STATISTICAL DATABASE MANAGEMENT

Author

Olken, F.

Publication Date

1986-05-01

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

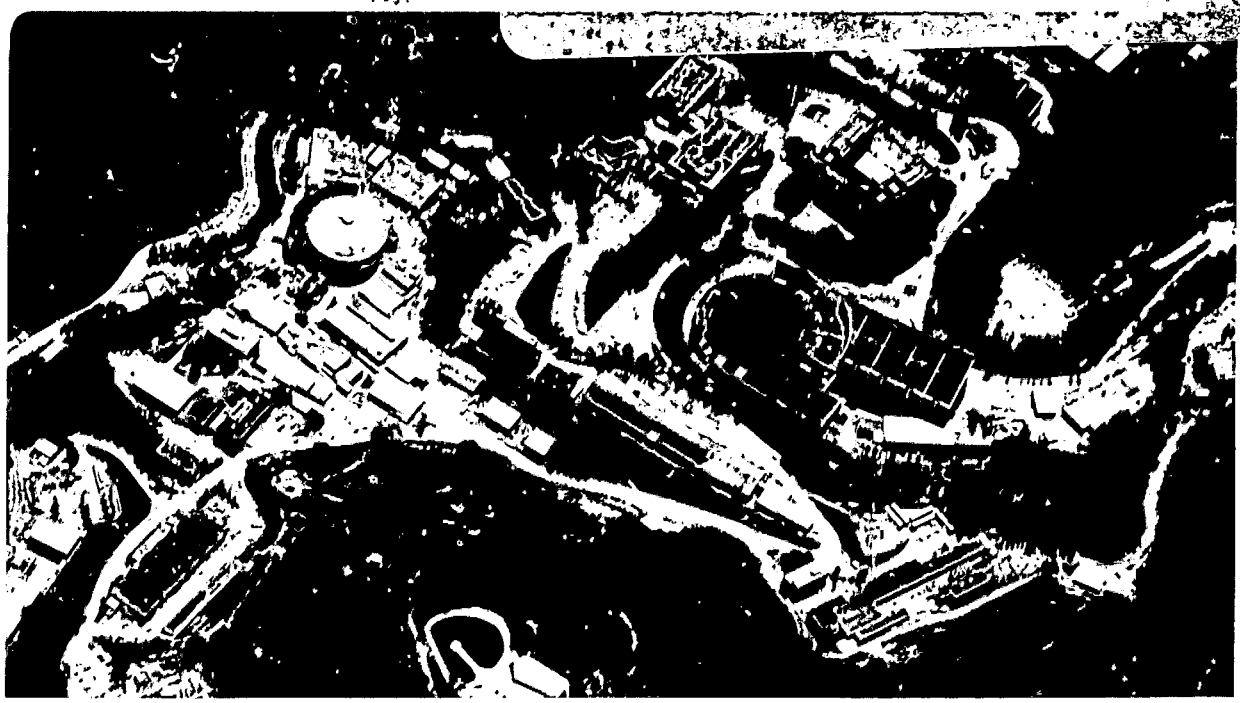
Computing Division

To be presented at the 3rd International Workshop
on Statistical and Scientific Database Management,
Luxembourg, Grand-Duchy of Luxembourg,
July 22-24, 1986

PHYSICAL DATABASE SUPPORT FOR SCIENTIFIC AND STATISTICAL DATABASE MANAGEMENT

F. Olken

May 1986



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

DISCLAIMER

LBL-19940

Physical Database Support for Scientific and Statistical Database Management

Frank Olken

Computer Science Research Department
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

May, 1986

Physical Database Support for Scientific and Statistical Database Management *

Frank Olken
Computer Science Research Dept.
Lawrence Berkeley Laboratory
Berkeley, CA 94720

May, 1986

Abstract

In this paper we survey the various physical database techniques that can be used to implement scientific and statistical database management systems. We consider techniques for storing the data, and algorithms for query processing. We discuss file structures, access methods, compression methods, buffering strategies, and algorithms for aggregation, transposition, and sampling. We conclude with some thoughts on areas for further research.

1 Introduction

This paper is a survey of physical database implementation techniques which are specially suited to the implementation of database management systems (DBMS) for scientific and statistical databases (SSDB).

The paper is concerned with efficient techniques for *storing* and *querying* the data. Unless otherwise specified we shall assume that the data is stored on magnetic disks.

Our criterion for *storage efficiency* is simply the amount of space required to store the data. *Query efficiency* may either refer to CPU time requirements or I/O time requirements. Usually I/O time requirements are the dominant consideration. In practical terms this usually amounts to minimizing the number of disk seeks.

The paper is organized into the following sections:

Our discussion of sampling is somewhat more extensive than other sections because of the lack of recent surveys of the area and because the topic is of special interest to the author.

2 SSDB Characterization

2.1 Factors Affecting DBMS Implementation

The implementation techniques chosen to implement a DBMS depend on several factors: type of data, type of queries, storage media, and computing environment. In this paper we shall be largely concerned with the impact of the special types of

*Issued as tech report LBL–19940 This work was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy, under contract number DE-AC03-76SF00098.

data and queries which arise in scientific and statistical database applications. We shall make conventional assumptions concerning storage media (magnetic disks with random access memory buffers) and computing environment (a single Von Neumann machine).

2.2 Type of Data

Commercial data are typically text and numeric data. The data are dynamic (i.e., updates are frequent). Hence commercial DBMSs are often preoccupied with issues of concurrency control.

In contrast statistical and scientific data are comprised primarily of numeric and discrete categorical data (i.e., data drawn from finite sets of (mostly unordered) categories such as race, sex, state of residence, type of material tested, experimental treatment, etc.) Summary data (e.g., tables of population by sex, race, age strata) frequently exists, usually in the form of multi-dimensional arrays. Statistical and scientific data are usually fairly static, and often sparse. Thus statistical and scientific data lend themselves to certain file organizations and data compression techniques which would not be practical for commercial data, which exhibit high update traffic. Concurrency control is not a critical issue. For further information see [Sho82,SOW84].

2.3 Type of Queries

Queries may be classified into several types: exact match, partial match, range, partial range, nearest neighbor, aggregate, and sampling.

Exact match queries require that all of the key attributes of a record exactly match those specified in the query. *Partial match* queries specify only some subset of the key attributes of a data record. *Range* queries specify finite intervals for all of the key attributes of a data record. *Partial range* queries specify finite intervals for *some* of the key attributes of a data record. *Nearest neighbor* queries specify a point, and a distance function. They seek the data records which are *nearest* to the specified point. *Aggregate* queries require the computation of some aggregate statistic (such as SUM, COUNT, MEAN, MEDIAN) from some specified set of data. *Sampling* queries require the generation of a random sample from some specified set of data.

In business applications *exact match* queries are the most common, e.g., retrieve the bank balance for a specific account number. Most bibliographic data base queries are *partial match* queries, e.g., find all book with "statistical" and "computing" in the title. In contrast scientific and statistical data base systems must support query types not often used in commercial or bibliographic DBMSs: *multi-dimensional range*, *multi-dimensional nearest neighbor*, *aggregation*, and *sampling* queries. The nearest neighbor queries arise from statistical operations such as clustering, classification, kernel density estimation, matched case generation for case control studies, and scientific applications such as particle track reconstruction. Range queries arise from applications such as density estimation, histogramming, and the construction of summary (contingency) tables.

3 File Design

3.1 Record-wise

Conventional DBMSs store each *record (tuple)* in a single (logically) contiguous block of storage.[1] We call this file organization *record-wise*. This is efficient for transactions which examine (or modify) many fields from a small number of records. Such transactions are commonplace in commercial database applications. Similar reference patterns arise in SSDBs when performing random sampling. Howeverr, the record-wise organization is less efficient for some types of queries found in SSDB applications, as discussed below.

3.2 Transposed files

Several SDBMSs (e.g., RAPID) store data in a *transposed file* format, also referred to as *vertically partitioned* or *attribute partitioned*. Here each *column* (containing the values of certain attribute (field) for all records) is stored in a logically contiguous block of storage.[2] Often each *column* is stored as a separate file.

This organization is good when transactions tend to reference a small number of fields in a large proportion of the records. Such reference patterns are commonplace when performing aggregations and more complex statistical analyses (cross tabulations, regression, etc.) on large multi-purpose survey data such as census data. It is a poor organization for sampling data records, as it may require one disk access for each field of each record sampled.

[1] The file system is usually responsible for mapping logically contiguous files onto (possibly noncontiguous) physical disk blocks. Hence a record larger than a disk block may not be stored on physically contiguous disk blocks.

[2] Again, physical disk blocks may not be contiguous.

For further information on the use of transposed file organizations see [Sve79], [THC79], [WFS75], [Tan83], [BT81]. Copeland, [CK85], discusses the advantages and performance of a fully transposed file organization (with tuple surrogates included in each partition). Batory, [Bat79], discusses efficient methods of searching transposed files. Khoshafian, [Kho84,KBD85], discusses the implementation and performance of various statistical operations (e.g., $X'X$, QR decomposition, and Singular Value Factorization) on both conventional and transposed file organizations.

Wong, [WLO*85], has taken transposed files to their ultimate limits, proposing to transpose the files at the bit column level. Various domain encoding techniques, combined with data compression, offer the possibility of major I/O improvements for counting queries and highly selective retrieval queries. The method has very poor performance for updates and sampling and hence is appropriate only for archival databases.

3.3 Vertically Clustered Files

An intermediate file design is to *cluster* several columns together, instead of completely transposing the file. *Vertical Clusters* are formed from columns (attributes) which tend to be referenced together. Such files are a compromise between record-wise and fully transposed file organisations, and can be tuned (by choice of the clustering of attributes) to the proposed application. The disadvantage to this organisation, as compared with fully transposed files, is that uneeded attributes are sometimes retrieved along with desired attributes, reducing I/O efficiency. The design and performance analysis of such *clustered* files are discussed in [MS77], [HN79], [MS84].

3.4 Multi-dimensional Access Methods

Multi-dimensional access methods (MDAMs) are intended to support multi-key retrieval especially partial match, range, partial range, and nearest neighbor queries. Access methods for supporting *partial match* retrievals are discussed in [Riv76]. We shall be concerned here with access methods which support *range, partial range* and *nearest neighbor queries* because continuous numeric data which can be ordered are more common in SSDBs. Most data structures which support partial range queries can also be used for partial match queries.

The topic has generated burgeoning interest in recent years, driven largely by problems in computational geometry [PS85] arising in computer aided design and geographic information systems [DBG*85]. There have been several extensive surveys [Sam84,LP82] and taxonomies [KBCV85] of multi-dimensional access methods recently, as well as a chapter in [PS85, Chapter 2]. An extensive discussion of the use of MDAM for spatial query processing is to be found in [Ore85b]. Hence we will not attempt a comprehensive survey here, but rather limit ourselves to presenting several illustrative techniques.

There are several possible criteria for evaluating the effectiveness of MDAMs. One criterion is *symmetry* with respect to queries: does the MDAM have *symmetric* performance to queries along different dimensions? A second criterion is how well the MDAM performs with *non-uniform, uncorrelated* data. A third criterion is how well the MDAM performs with *correlated* data. Other considerations include: cost of performing updates to the data structure, storage utilization, how well the data structure can be tuned to accomodate non-uniform query distributions.

We will consider multi-dimensional B-trees, quad-trees, kd-trees, kd-tries, and some kinds of grid files. The basic idea of the various MDAMs is to place data which is close together in multi-dimensional logical (data) space, close together in physical space (i.e., colocated on the same disk page). They do this by partitioning the data space. The tree-based MDAMs (MDB-trees, KD-trees, etc.) generate *nested partitions*, i.e., the multi-dimensional space is recursively decomposed into successively smaller partitions, which are nested inside of the larger (outer) partitions. Instead of nested partitions , grid file type MDAMs generate orthogonal partitions of the key space, e.g., partitioning a 2D key space in a manner similar to a tile floor (but the partition widths are allowed to vary).

3.5 Multi-dimensional B-trees

The simplest approach to constructing a MDAM is simply to concatenate the keys to form a composite key. This is basically the approach taken with *multi-dimensional B-trees (MDBTs)*. A top level B-tree is constructed on the first key (dimension). Each leaf of this tree points to a B-tree on the key (dimension), and so forth. Obviously this approach does not provide very *symmetric* access along the different dimensions. Rather it favors retrievals along the first (top) dimension. Searches on the first key immediately narrow to a single subtree at the top level of the MDBT, whereas searches on just the second key (dimension) must examine every one of subtrees at the top level (first key).

Note that the MDBT partitions the data space into nested partitions.

3.6 Quadtrees

Quadtrees (QTs) also partition the data space into nested partitions. For the case of two dimensions, the root partitions the data space into 4 quadrants by specifying the (x,y) coordinates of a point at the origin of the quadrants. Internal nodes at lower levels of the tree recursively partition the quadrants into successively finer sub-quadrants. Quadtrees provide fairly *symmetric* access along different dimensions, i.e., uni-dimensional (partial range) queries have approximately the same efficiency on various dimensions, but they are difficult to keep balanced when updated [OvL82]. They adapt well to *non-uniform* data, but not to *correlated* data. The notion can be extended to higher dimensions (3 dimensional quadtrees are sometimes known as octtrees).

3.7 KD-trees

KD-trees (KDTs) also partition the data space into nested partitions. They differ from quadtrees in that each internal node paritions its subspace into only two halves. Nodes at successive levels of the tree *discriminate* on different dimensions. The discriminating dimensions are chosen cyclically, so that the kd-tree has fairly *symmetric* performance. Like the quadtree, the kd-tree adapts well to *non-uniform* data, but is difficult to keep balanced [OvL82]. Its advantage over the quadtree is that it works well with *correlated* data.

Robinson [Rob81] has proposed a B-tree-like analog of kd-trees. The high fanout is attractive for disk resident indices.

Generalized kd-trees (GKDTs) were proposed by Fushimi [FKN*85]. In GKDTs the discriminating dimensions are not chosen cyclically, but are tailored to the data and query distributions. Since updates do not preserve the optimality of the tree, GKDTs are presently only useful for static data.

3.8 KD-tries

Tries (radix search trees) can also be extended to multi-dimensional data [Ore82] by cyclically choosing the discriminators from the different dimensions. Again a nested sequence of partitions is generated. However, here the partition boundaries are no longer arbitrary (data dependent), but must be chosen according to the radix. Tries have fairly *symmetric* access, reasonable average performance and can be easily updated (no rebalancing is necessary). However, they are not as adaptable to non-uniform data as kd-trees (balancing is not possible).

The KD-tries appear to be more attractive for performing joins, precisely because of their inflexibility with respect to partition boundaries. Two separate KD-tries on over the same types of domains will have *consistent* partitions. Partitions of the two tries will either be entirely disjoint or one partition (region) will be entirely contained in the other. This greatly facilitates performing partitioned joins, as employed in [DKO*84], [DG85], [KTM83], [Bra84]. The use of related ideas for spatial join-like query processing is discussed in detail in [Ore85b].

3.9 Orthogonal Partitioning

A second class of MDAMs paritition the data space orthogonally, that is the data space is partitioned by hyperplanes which cut across the entire data space. One can envision each axis as being partitioned by a scale, and the entire data space being partitioned by the cartesian cross products of the scales, much like graph paper.

The orthogonal partitioning MDAMs (OPMDAMs) differ in how the axis partitions are chosen, the type of directory employed and how they handle overflow.

The simplest OPMDAMs, called *fixed cellular* designs [BF79], employ fixed partitions of each axis. Obviously they are not very adaptable to nonuniform data.

Grid files [NHS84] allow arbitrarily placed axis scales. A directory is employed which may map several grid blocks onto a single disk page. As the file grows the combined grid blocks are unpacked onto several disk pages. Eventually a single grid block occupying an entire page overflows, requiring that one of the intervals on an axis scale be split, a *directory split*. Disk pages and even axis intervals can be merged during deletions. *Grid files* appear well suited to non-uniform data, updatable, but ineffecient in terms of space for correlated data.

Multi-paging [Mer84] is similar to *grid files* except that it allows grid blocks to overflow onto overflow pages, providing possibly better space utilization and simpler directory maintenance at the expense of worse access time.

Multi-level grid files [KW85] employ a KD-trie index for the directory of a grid file. They provide better space efficiency for the directory and facilitate directory splitting.

3.10 Summary

In general multi-dimensional access methods with completely variable partition boundaries, such as KD-trees, seem to offer better storage and retrieval efficiencies than methods, which choose partition boundaries from a fixed set of possibilities, such as KD-tries. However, the KD-tries appear to be more attractive for performing joins, precisely because of their inflexibility. MDB-trees are clearly inferior in terms of symmetry of performance. KD-trees seem to offer better performance than quadtrees or grid files for correlated data.

It is worth noting that if the MDAM is to be used as a *partial sum hierarchy* to support aggregate queries over ranges (as described in Section 8.1) then large radix tries become very inefficient for updating. Trees are to be preferred.

Finally, Orenstein's proposal, [Ore85b], to employ trie-encodings of multi-dimensional keys with conventional B-trees for spatial query processing appears quite promising.

4 Data Compression

In this section we discuss the purposes of employing data compression in SSDBs, and some popular data compression methods used in SSDBs. For a more extensive treatment of the subject the reader should consult [Bas85].

4.1 Purpose

Data compression serves several purposes. The obvious justification is that data compression reduces the amount of storage required to hold the data. In addition to the direct savings in storage costs there are several indirect benefits.

By reducing the storage requirements we often reduce the time required to move the data from disk to RAM. This occurs when transferring multi-block segments of data. Single block reads are unaffected. SSDBMSs (Scientific and Statistical Database Management Systems) are more likely to transfer multi-block segments (than commercial DBMSs) and hence benefit more.

A second effect comes from improving the effective fanout of B-tree indices. Halving the storage required per key will effectively double the fanout of the B-tree, thereby often reducing the height of the tree and thus the number of disk accesses needed to search the index.

There may even be gains in cpu time for query processing, which tends to be highly correlated with the number of disk pages read.[3] Such cpu savings are especially likely if the data can be processed in compressed form [WL86]. Decompression tends to increase cpu time requirements. *Index coding* (discussed below) reduces cpu requirements by substituting fixed length codes (which are easy to manipulate) for variable length fields.

4.2 *ad hoc* Data Compression Methods

We will discuss primarily *ad hoc* methods, because they have thus far proven more practical for SSDBMSs. *Information theoretic* methods will be discussed briefly below.

Summary data tables are commonplace in SSDBMSs. Such tables often record measured statistics (e.g. population counts) over a cross product of categorical variables (e.g. race, sex and age). Instead of explicitly storing the values of the categorical attributes one can implicitly store them in the location of the data via *array linearization*. Such methods are commonly used in programming languages such as FORTRAN to store arrays [HS77].

The most commonly used *linearizations* are multi-dimensional versions of the raster scan pattern used in television tubes [HS77]. Such array linearization techniques have been employed to store summary tables in [OOM85].

However alternative *linearizations* are possible employing *space filling curves*. Such linearizations do a better job of preserving spatial locality, i.e., points which are close together in logical coordinates are usually close in physical (linear) coordinates. Unfortunately the best space filling linearization functions are more difficult to compute. However, one such ordering, the z-ordering, is fairly easily computed by interleaving the bits from several keys. See [OM84], [OS83], [Ore85a], [Ore85b].

4.2.1 Index Coding

Index coding [Bat83] consists of representing long (possibly variable length) attributes, such as state of residence, by means of short fixed length numeric codes. The length of the numeric codes is dictated solely by the number of distinct attribute values which need to be encoded. By assigning the numeric codes in alphabetical order, range queries and prefix partial match queries are possible. Such coding is referred to as *order preserving codes*.

[3] Because of cpu requirments to initiate i/o and copy data between buffers.

Batory [Bat83] has suggested that the codes be initially chosen with a few extra low order zero bits. If additional attributes values are added later (new states), they can be inserted into the code table in between the original codes, while preserving the alphabetical ordering. Eventually, of course, one runs out of codes, but the day of reckoning can be postponed.

4.2.2 Bit Vector Encoding

Bit vector encoding is a method of compressing zero (null) elements. It consists of storing a bit vector in which the one's indicate nonzero (non-null) data elements, and the zero's indicate zero (null) elements.

Example:

Original Data - n items: 1, 0, 0, 0, 5, 3, 0, 0, 2, 0, 0

Bit Vector: 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0

Compressed Data - m items: 1, 5, 3, 2

Bit vector encoding has linear access time, $O(n)$. It requires storage linear in original data size, i.e., $O(n)$ bits in the bit vector.

4.2.3 Run Length Encoding

Another method of compressing nulls (zeros) is run-length encoding (RLE). It consists of replacing consecutive runs of zero data elements, with counts of the run lengths. Lengths of the sequences of nonzero elements are also required. RLE requires total storage proportional to number of nonzero items plus the number of runs. Run lengths may either be stored together with compressed data or separately.

Access time is either linear in the number of runs (if the run lengths are stored separately) or linear in the number of runs and size of compressed data (if the run lengths are stored with the compressed data).

The method can also be used to compress consecutive runs of any repeated data values, in which case the repeated data value must also be stored.

Shown below is an example in which the run lengths have been stored separately from the compressed data.

Example:

Original Data - n items: 1, 0, 0, 0, 5, 3, 0, 0, 2, 0, 0

Run Lengths (Nonzero, Zero) - r runs: (1,3), (2,2), (1,2)

Compressed Data - m items: 1, 5, 3, 2

4.2.4 Header Compression

Header Compression consists of a combination of run length encoding with a B-tree index which is used to obtain efficient random access to the data, avoiding the need to sequentially decompress the data. The run lengths are stored separately from the compressed data, and a partial sum hierarchy (similar to a B^+-tree) is built atop the run lengths to provide an index. The B^+-tree index provides efficient random access. Access time is logarithmic in the number of runs r, $O(\log r)$. The index requires space $O(r)$. Total storage is proportional to number of nonzero items plus the number of runs. See [ES80,EOS81] for further details.

4.3 Information Theoretic Methods

Information theoretic methods of data compression (e.g. Huffman coding) have not enjoyed wide popularity in SSDBMSs. One reason is that they are more expensive to decompress than the simple *ad hoc* methods discussed above. The other reason is the adaptive versions of the information theoretic compression techniques usually require sequential decompression, which is not a problem for serial communications lines or sequential files, but it is impractical for random access databases.

See [Wel84] (and the references therein) for a discussion of an adaptive information theoretic compression method.

5 Buffering

Buffer pool management policies for DBMSs have evolved over the last decade from very simple, fairly autonomous global LRU policies, to policies which are intimately tailored to the query evaluation plan developed by the query optimizer and the access methods employed. This evolution has culminated in recent work [KBD85] suggesting that entire statistical computations should be closely integrated with the buffer pool management policies of the DBMS.

In this section we trace the evolution of DBMS buffer management policies from their crude beginnings to the present proposals, which are closely tailored to SSDB applications.

Traditionally, buffer management policies were derived from classic virtual memory policies. Such poli-

cies are based on simple stochastic models of memory reference patterns, since most programs have no explicit knowledge of their reference behavior. Hence one of the simplest and most popular buffer management strategies is global Least Recently Used (LRU), i.e., evict the page which has been least recently referenced. For DBMSs this policy is often poor.

In DBMSs, the query optimizer often has detailed information on the expected storage reference pattern, derived from the query execution plan and the database schema. Furthermore, the query optimizer may be able to change the preferred query execution strategy to accommodate restricted buffer allocations [DKO*84]. In particular Sacco and Skolnick [SS82] observed that DBMS often generate cyclic reference patterns to pages in the inner loop of a nested loop join. If the buffer pool is not large enough to contain the inner loop, then LRU policy will cause thrashing, every page reference generating a fault. In such a case, MRU (Most Recently Used) policy would perform better. Sacco and Skolnick argued that the buffer manager should be informed of such loops and should assure that sufficient buffers were allocated to contain the inner loop. This was referred to as the *Hot Set policy.*

Chou [Cho85] has gone further in integrating the query optimizer and buffer manager. He argued that buffer management policies should be separately tailored to the anticipated reference behavior of each file during the query evaluation, e.g. sequential scan, looping, index tree traversal, etc. He has shown that if the query optimizer informs the buffer manager of its anticipated reference behavior for each file, the buffer manager can do a better job. This policy is called *Query Locality Set policy.*

Khoshafian [Kho84,KBD85] has pursued similar close integration of query optimization and buffer management for SSDB applications. He has investigated several statistical matrix operations ($X'X$, QR decomposition, and Singular Value Factorization) and shown that significant performance gains can be had by close integration of the numerical algorithms, file organization and buffer management.

Copeland, [CKSV86], has proposed an *attribute* based buffering approach which entails the buffering of the transposed columns of relations, instead of pages containing full tuples. The attribute based buffering approach offers some of the advantages of transposed file organizations for SSDB applications (as described in Section 3.2), without necessarily requiring that the entire database first be reorganized (transposed). However, processing updates would be cumbersome. Hence the technique is best suited to archival SSDB datasets. The technique may entail buffer management of variable sized objects, a problem which has been addressed in the literature on automatic file migration (e.g., [Olk83] and references therein).

6 Data Editing Support

SSDBs are frequently composed of a large, static base file (i.e., the original data set) against which a variety of relatively small editing changes are made. The *edits* are made to correct input errors, impute values to missing data, or delete outliers (anomalous data).

6.1 Differential Files

Differential files techniques [SL76] appear to offer a very attractive means of supporting this pattern of update activity. Instead of applying the updates directly to the base file, one keeps a separate file of the changes called a *differential file.* The differential update file is organized as a hash table. Whenever we process a query against the updated file, we check each record retrieved against the differential file to determine if there is a more recent version, or if the record has been deleted. We must also keep a separate file of additions, which must also be searched for each query. Indexing and searching the addition file can complicate the design of the data management system. However, often the differential files are small and can be kept in memory, so that checking them does not generate I/O activity.

Copeland, [CK85], has noted that differential files can be more storage efficient when transposed file organizations are used.

6.2 Bloom Filters

To avoid a disk seek to the hashed differential file for each record retrieved in a query we can construct a *Bloom filter* [Blo70] on the record ID's in main memory. A Bloom filter is constructed by logically or'ing together several one bit wide hash tables. Each hash table is constructed with a different hash function (overflows are ignored). Supppose that each hash table is 6% full. If we logically or together 8 such hash tables, the result will be approximately half full. For each record ID in a query we probe the Bloom filter 8 times, each time using a different hash function. The probability that all 8 probes will find the hash table entry set, given that the record ID is not in the differential file, is 2^{-8} or about 0.5%. Hence we will only have to check the differential file

for those records which are in it, plus about 0.5% of records which have not been changed.

7 Exploratory Data Analysis Support

Exploratory data analysis (EDA) often entails the examination of many different subsets of the data. In this section we discuss methods of storing and processing these subsets. Often these subsets are very similar, i.e., they may be produced by successively removing outliers, or adding variables (columns). In relational database systems such subsets are referred to as *views*.

EDA generates two problems for the DBMS:

1. Excessive space requirements to store subsets.
2. Sequences of similar queries, which are inefficient if processed independently.

Several techniques have been developed to address these problems. Storage requirements for subsets of the data can be reduced by using: query modification, record ID lists, or differential record ID lists. Views (subsets) can be implemented by storing the query which generated the subset and rerunning it whenever the subset is needed. A more elaborate version of this technique call *query modification* [Sto75]. consists inserting the text of the original query in place of the view (subset) name whenever it is subsequently used and evaluating the composite query.

Record ID lists (a.k.a. tuple ID lists) are simply lists of record identifiers of records contained in the subset. Since the record IDs are usually much shorter than the actual records, this saves space at the expense of having to re-retrieve the records whenever the subset is needed. However, it is more efficient than simple views, since no query processing is required and the tuple ID lists can be directly used in subsequent query processing.

Since many of these subsets are similar (differing perhaps by the removal of a few outliers) they can be represented as *differential* record ID lists. Here only the differences from a *base* relation are stored. The *differential* record ID lists must be combined with the *base* record ID lists to recreate the subset when needed as described above in Section 6.

The problem of efficiently processing a sequence of similar queries has been addressed by Finkelstein [Fin82]. He proposed techniques of indexing and recognizing common subexpressions across multiple queries. By saving and suitably indexing the intermediate results of query evaluations, the need to reevaluate the common subexpressions can be avoided.

Automatic cache management policies similar to those used for automatic file migration [Olk83,Smi81] might be employed to automatically decide when to convert subsets from one representation (instantiation, tuple ID list, query) to another (more compact) representation as the expected usage rate declined. Of course, instead of changing the reprsentation of the subset one could simply automatically migrate the subset to cheaper, slower storage [Olk83,Smi81].

8 Aggregation

By aggregation we mean the computation of aggregate statistics such as (SUM, COUNT, MEAN, MIN, MAX, etc.) over groups of data items (e.g. employees grouped by department). Such computations are commonplace in SSDBs.

The classic algorithm consists of sorting the data items according to their group, and then calculating the desired statistic for each group.

One can often do much better, if the statistic can be computed in fixed space for each group (e.g. SUM, but not MEDIAN). Usually the number of groups is fairly small. We can thus build a table of partially computed statistics, one for each group, in memory. The table is usually organized as a hash table on the group identifier so that it can be accessed in constant time. We then make one pass across the data, hashing the group identifier for each tuple, and updating the appropriate entry in the table of statistics. After reading the data, we usually must sort the statistics table, before writing it out.

If the statistics table is too large to fit in main memory, we can partition the data by hashing on the group identifier, and then process parititions one at a time. See [DKO*84,DG85] [KTM83], [Bra84].

For detailed discussions of query optimization of queries which include aggregation see [Eps79,Klu82]. Nested correlated SQL-like queries with aggregation are treated in [Kim82]. Such nested queries involve parametric aggregates in the inner query, where the parameter is a tuple variable from the outer query. The naive approach to processing such queries, variously called *nested iteration* or *tuple substition* may entail repeated (redundant) evaluations of the inner aggregates. Therefore Kim, [Kim82], suggests evaluating the inner aggregate once, for all groups. This optimization is often performed in the syntatic analysis phase without any semantic information. While this optimization is usually desirable, there are cases in which the naive algorithm

is faster (e.g., when the outer predicate is very selective). Kiessling [Kie85] discusses semantic problems arising from Kim's [Kim82] optimizations and proposes tactics, called *dynamic filters*, for improving the efficiency of this type of query processing.

8.1 Fast Summary Statistics for Range Queries

Often we will want to obtain summary statistics (aggregates such as SUM, COUNT, AVG, MIN, MAX) over a range of data values, e.g., find the total income of all employees between 40 and 50 years old.

Such queries can be answered by combining a tree index to the data with a *partial sum tree*. In the example above we might construct a binary tree index to the employee data on the age attribute. In addition, each internal node of the tree would contain the sum of the incomes of all the emplyees contained in the subtree rooted at that node.

We can thus determine the sum of incomes of employees who are at least 40 and under 50 years old by traversing the binary tree twice, once to find the youngest employee at least 40 years old, and once to find the oldest employee under 50. As we traverse the tree, we combine the partial sums appropriately, adding partial sums of included subtrees and subtracting partial sums of excluded subtrees, to give the desired sum over the specified range.

For a one dimensional index we can calculate the sum for any range query in logarithmic time. We can also update the tree in (average) logarithmic time. The basic idea has been known for some time and used to maintain rank information in trees. See [Knu73,BK75,WE80]. The idea can be extended to B-trees, and to multi-dimensional trees. See [FV82,Fre81] for a discussion of the computational complexity of answering aggregate queries over multi-dimensional ranges and citations to data structure papers.

9 Transposition

9.1 Uses

Transposition is a commonly used operation in SSDBMSs. One application is the conversion of files between regular (record structured) and transposed versions. The second application concerns the restructuring of summary data tables [OOM85]. Often the user will want to transpose such table. For example mortality data may be tabulated by race, by sex, by age, by county. A user might prefer the data tabulated by county, then by race, by sex, by age. Or gasoline consumption may be tabulated first by state, and then year and type; whereas some user prefers instead a tabulation first by type, and then by state and year.

Transposition of an array of data can be thought of as changing the array linearization function, and then sorting the data according to the new location tags. For sparse data this is in fact the preferred algorithm:

1. Tag each data item with its new location.
2. Sort the data according to the location tags.
3. Strip the location tags, and build compressed form.

For dense data sets (all data elements present, although some may be zero) we dispense with the tags and the comparisons involved in the sorting. Instead, we merely perform the data movements which would occur if we sorted the tagged data items, in effect performing an *implicit sort*. The decisions as to where to move the data items are embedded in the structure of the transposition program, rather than being explicitly determined by comparisons.

The naive transposition algorithm, which exchanges diagonally opposite elements, can thus be seen to correspond to an $O(n)$ address calculation sort, (where n is the total number of data elements in the array). If the dataset exceeds the size of main memory this algorithm may generate excessive (i.e., n) numbers of page faults.

Comparison based sorts (and thus the derivative transposition algorithms) typically require $O(n \log n)$ computer time. However, they only require $O((n/p)\log(n/p))$ page faults, where p is the page size. Although the asymptotic complexity is worse than for the naive algorithm, in practice it is better because $(1/p)\log(n/p)$ is always much less than one for realistic values of n and p.

One can do even better by combining the two types of algorithms. Observe that a disk is a *page addressable* random access memory. We partition the array to be transposed into blocks of p pages which move together in the transposition. The transposition is accomplished by transposing each block of p pages via a sort based algorithm in time $O(p \log p)$, i.e., time $O((n/p)\log p)$ for the entire array. Moving the blocks around on disk can then be done in one pass using the naive algorithm, with $O(n/p)$ page faults.

The original paper on tranposition of dense tables was [Flo72]. Floyd's results were generalized in [TUS83a,TUS83b,Wie83]. For sparse data, see [WL86], which discusses the tranposition of compressed relations.

10 Sampling

Random sampling is a very common operation in SSDBs. It is widely agreed that it should be included in the DBMS for efficiency reasons. Present practice usually dictates that samples of relational queries are obtained by first computing the complete result of the relational query, exporting the result out of DBMS to the statistical analysis package, and then extracting a comparatively small sample. Inclusion of sampling operators in the DBMS permits a reduction in the amount of data retrieved. Also the DBMS may be able to employ existing indices. The result of such optimizations, is that samples of relational queries should be obtainable in time proportional to the *sample size*, rather than the size of the full result of the relational query.

Random samples may be required either of existing files, or of the results of relational queries. Sampling from relational queries is beyond the scope of this paper. It is the subject a paper [OR86b] which considers how sampling can be efficiently integrated into the query evaluation in a relational database system.

In this paper we provide brief tutorials on the fundamental methods of generating various types of random samples. These methods can be used directly to sample from existing flat files on disk, or from the full results of relational queries as they are generated. These methods also form the basis of the techniques in [OR86a] used to sample from structured (B-trees, grid files, etc.) and to embed sampling operations within the relational query evaluation.

10.1 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 or sequential, 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.

One characteristic concerns the sample size determination. We shall be primarily concerned with *fixed size* samples, where the sample size has been specified by the user. Alternatively, the user may specify that a certain *fraction* of the records should be sampled. This is called *binomial sampling*, because the sample size distribution is binomial. It is also discussed below in Section 10.2.

Other possiblities for sample size determination include: *multi-stage sampling*, and *sequential sampling*. In each of theses cases the ultimate sample size is determined dynamically from the statistical properties of the data set. See [Coc77].

Another way of characterizing the sample is whether it is drawn with or without replacement. Samples drawn with replacement are usually easier to obtain (as we shall see). However, samples drawn without replacement generally provide more information for a given sample size.

The sample may be drawn from a population of known or unknown size. Sampling from unknown population sizes arises when sampling sequentially from tape files or from files as they are generated.

The inclusion probabilities for individual records may be uniform (an *unweighted or simple random sample (SRS)*) or they may be *weighted* according some attribute of the record.

Access to the data may be random (if it is stored on disk or RAM), or sequential (if the data is on tape or is being generated as the output of a query).

10.2 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. This idea has been developed by Vitter [Vit84,Vit85] for sequential sampling of fixed size samples as discussed below in Sections 10.7 and 10.8.

10.3 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.*

10.4 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 we can use it as above. If no index exists we can use *acceptance/rejection* [Rub81] sampling as described in Section 10.6.1 on the pages with

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

If the page is accepted we select a record on the page at random. This assures uniform selection probabilities for all records.

10.5 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.

10.6 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.

We 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.

10.6.1 Acceptance/Rejection Sampling

Suppose that we wish 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}.

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

$$p_j = \frac{w_j}{w_{max}} \tag{1}$$

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

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

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.

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

10.6.3 Alias Method

Another method of weighted sampling is the *alias* method proposed by Walker [Wal77]. This method is similar to acceptance/rejection methods, except that if a random record number is rejected, then an *alias* is supplied in its place. Thus the time to obtain a sample of size s is $O(s)$. A table of adjusted sampling weights, and aliases is required. 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.

10.7 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.

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 we can skip records in zero time (e.g., fixed size records on disk) then the total running time will be $O(s)$, otherwise we may be forced to read every record in time $O(n)$.

10.8 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 algorithms then proceed sequentially through the file, updating the reservoir, so that it always contains a simple random sample.

Again, the best 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.

11 Summary

In this paper we have attempted to survey what is presently known about physical data management for scientific and statistical databases. We have discussed:

- The characteristics of SSDBs which differentiate them from conventional commercial databases: the low update activity, the tendency for queries to reference few attributes and many tuples, and the frequent use of range, aggregate, and sampling queries.
- The use of *tranposed* file organizations, their advantages for typical statistical queries, and their disadvantages for conventional and sampling queries.

- *Ad hoc* data compression methods such as index encoding, run-length encoding, and header compression, which are well suited to SSDB applications.

- Several of the papers on buffering strategies, which showed that closer integration of buffering with the DBMS and perhaps statistical operators offers significant performance gains.

- The use of differential files, Bloom filters, common subexpression indexing, and subset representation caching to deal with the update and subset management problems generated in data editing and exploratory data analysis.

- algorithms for dynamically computing aggregate queries, and indexing strategies such as partial sum trees for improving the efficiency of linear aggregate query processing over ranges.

- Several methods of performing transposition on both sparse and dense data.

- The fundamental random sampling techniques, which form the basis of more elaborate methods of sampling from relational queries.

12 Research Agenda

Much work remains to be done in physical database support for scientific and statistical data management.

It is clear that the performance of various file designs is very sensitive to the type of queries. For example, transposed files are well suited for queries involving aggregation and statistical operations, but not for sampling. Linear space multi-dimensional access methods appear to do well for full range queries, but poorly for uni-dimensional queries. Further work, and perhaps some experience with real SSDBMSs is needed to provide better guidance to designers of SSDBs.

In particular, more work is needed on characterizing the performance of multi-dimensional access methods with non-uniform correlated data and queries. There have been a few simulations and virtually no analytical work in this area.

The work on integrating statistical algorithms and buffering could be extended to other important scientific and statistical algorithms. Better I/O models of access method performance, which incorporate caching, file allocation on disks, and detailed disk motions would be useful for query optimizers.

Better proposals for access methods suited to write once optical disks are needed. Query processing algorithms which work directly on compressed data have only begun to be explored. Finally, there does not seem to have been much work on modelling, storing, or retrieving statistical models.

Acknowledgements

The author would like to thank his colleagues Janet Lafleur, Harry Wong, and Doron Rotem, and especially Arie Shoshani for their encouragement and comments. Thanks are also due to the referees of earlier versions of this paper for their comments.

References

[Bas85] M.A. Bassiouni. Data compression in scientific and statistical databases. *IEEE Transactions on Software Engineering*, SE-11(10):1047–1057, October 1985.

[Bat79] D.S. Batory. On searching tranposed files. *ACM Transactions on Database Systems*, 4(4):531–544, December 1979.

[Bat83] D.S. Batory. Index coding: a compression technique for large statistical databases. In *Proceedings of the Second International Workshop on Statistical Database Management*, pages 306–314, September 1983.

[BF79] J.L. Bentley and J.H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, December 1979.

[BK75] B.T. Bennett and V.J. Kruskal. Lru stack processing. *IBM Journal of Research and Development*, 19(4):353–357, July 1975.

[Blo70] B.H. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[Bra84] Kjeil Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of the 10th International Conference on Very Large Databases (VLDB)*, pages 323–333, Singapore, August 1984.

[BT81] R.A. Burnett and J.J. Thomas. Data management support for statistical data

editing and subset selection. In *Proceedings of the First LBL Workshop on Statistical Database Management*, pages 88–102, December 1981.

[Cho85] Hong-Tai Chou. *Buffer Management of Database Systems.* PhD thesis, Univ. of Wisconsin, Madison, May 1985.

[CK85] George F. Copeland and Setrag Khoshafian. A decomposition storage model. In *ACM SIGMOD International Conference on the Management of Data*, pages 268–279, 1985.

[CKSV86] G. Copeland, S. Khoshafian, M. Smith, and P. Valduriez. Buffering schemes for permanent data. In *IEEE International Conference on Data Engineering*, IEEE Computer Society, Los Angeles, Calif., February 1986.

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

[DBG*85] Umeshawar Dayal, Alejandro Buchman, David Goldhirsch, Sandra Heiler, Frank A. Manola, Jack A. Orenstein, and Arnon S. Rosenthal. *PROBE - A Research Project in Knowledge-Oriented Database Systems: Preliminary Analysis.* Technical Report CCA-85-03, Computer Coroporation of America (CCA), Cambridge, Mass., July 1985.

[DG85] David J. DeWitt and Robert Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 151–164, VLDB Endowment, Stockholm, Sweden, 1985.

[DKO*84] D.J. DeWitt, R.H. Katz, F. Olken, L.D. Shapiro, M. Stonebraker, and D. Wood. Implementation technique for large main memory database management systems. In *1984 ACM-SGIMOD International Conference on the Management of Data*, pages 1–8, ACM, Boston, 1984.

[EN82] Jarmo Ernvall and Olli Nevalainen. An algorithm for unbiased random sampling. *The Computer Journal*, 25(1), 1982.

[EOS81] S. Eggers, F. Olken, and A. Shoshani. A compression technique for large statistical databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 424–434, 1981.

[Eps79] R. Epstein. *Techniques for Processing of Aggregates in Relational Database Systems.* Technical Report UCB/ERL M79/8, University of California, Berkeley, February 1979.

[ES80] S. Eggers and A. Shoshani. Efficient access of compressed data. In *Proceedings of the International Conference on Very Large Databases*, pages 205–211, 1980.

[Fin82] S. Finkelstein. Common expression analysis in database applications. In *ACM SIGMOD International Conference on the Management of Data*, pages 235–245, 1982.

[FKN*85] Shinya Fushimi, Masaru Kitsuregawa, Masaya Nakayama, Hidehiko Tanaka, and Tohru Moto-oka. Algorithm and performance evaluation of adaptive multidimensional clustering technique. In *ACM SIGMOD International Conference on the Management of Data*, pages 308–318, ACM, Austin, Texas, May 1985.

[Flo72] R.W. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 105–109, Plenum Press, 1972.

[Fre81] Michael L. Fredman. A lower bound on the complexity of orthogonal range queries. *Journal of the ACM*, 28(4):696–705, October 1981.

[FV82] F.L. Fredman and D.J. Volpen. The complexity of partial match retrieval in a dynamic setting. *Journal of Algorithms*, :68–78, 1982.

[HN79] M. Hammer and B. Niamir. A heuristic approach to attribute partitioning. In *ACM SIGMOD International Conference on the Management of Data*, pages 93–101, 1979.

[HS77] H. Horowitz and S. Sahni. *Fundamentals of Data Structures.* Computer Science Press, Inc., Potomac, Md., 1977.

[KBCV85] Setrag Khoshafian, Jay Banerjee, George Copeland, and Patrick Valduriez. *A Performance-directed Taxonomy for Single-key and Multi-key File Structures.* Technical Report, Microelectronics and Computer Technology Corp. (MCC), Austin, Texas, June 1985.

[KBD85] Setrag Khoshafian, Douglas M. Bates, and David J. Dewitt. Efficient support of statistical operations. *IEEE Transactions on Software Engineering*, SE-11(10):1058–1070, October 1985.

[Kho84] Setrag Khoshafian. *A Building Blocks Approach to Statistical Databases.* PhD thesis, Univ. of Wisconsin, Madison, May 1984.

[Kie85] Werner Kiessling. On semantic reefs and efficient processing of correlation queries with aggregates. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 241–249, VLDB Endowment, Stocklholm, Sweden, 1985.

[Kim82] Won Kim. On optimizing an sql-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.

[Klu82] Anthony Klug. Access path selection in the "abe" statistical query facility. In *ACM SIGMOD International Conference on the Management of Data*, pages 161–173, ACM, Orlando, Florida, June 1982.

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

[KTM83] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to database machine and its architecture. *New Generation Computing*, (1):62–74, 1983.

[KW85] Ravi Krishnamurthy and Kyu-Young Whang. *Multilevel Grid Files.* Technical Report, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., June 1985.

[LP82] D.T. Lee and Franco P. Preparata. Computational geometry – a survey. *IEEE Transactions on Computers*, C-33(12):1072–1101, December 1982.

[Mer84] T.H. Merrett. *Relational Information Systems.* Reston, 1984.

[MS77] Salvatore T. March and Dennis Severance. The determination of efficient record segmentations and blocking factors for shared data files. *ACM Transactions on Database Systems*, 2(3):279–296, September 1977.

[MS84] Salvatore T. March and Gary D. Scudder. On the selection of of efficient record segmentations and backup strategies for shared databases. *ACM Transactions on Database Systems*, 9(3):409–429, September 1984.

[NHS84] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: an adaptable, symmetric multkey structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[Olk83] Frank Olken. Hopt: a myopic version of the stochopt automatic file migration policy. In *1983 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 39–43, ACM, Minneapolis, Minn., August 1983.

[OM84] Jack A. Orenstein and T.H. Merrett. A class of data structures for associative searching. In *Third SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, 1984.

[OOM85] Gultekin Ozsoyoyoglu, Meral Ozsoyoyoglu, and Francisco Mata. A language and physical organization technique for summary tables. In *ACM SIGMOD International Conference on the Management of Data*, pages 3–16, 1985.

[OR86a] F. Olken and D. Rotem. Simple random sampling from relational databases. In *Proceedings of the International Conference on Very Large Databases*, VLDB Endowment, August 1986. condensed version of LBL-20707.

[OR86b] F. Olken and D. Rotem. *Simple Random Sampling from Relational Databases.* Technical Report LBL-20707, Lawrence Berkeley Laboratory, February 1986.

[Ore82] Jack A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, June 1982.

[Ore85a] Jack A. Orenstein. *Spatial Query Processing in an Object-Oriented Database System.* Technical Report, Computer Coroporation of America (CCA), Cambridge, Mass., 1985.

[Ore85b] Jack A. Orenstein. *Spatial Query Processing in PROBE.* Technical Report, Computer Coroporation of America (CCA), Cambridge, Mass., December 1985.

[OS83] M. Ouksel and P. Scheuermann. Storage mappings for multidimensional linear dynamic hashing. In *Second SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, 1983.

[OvL82] Mark H. Overmars and Jan van Leeuwen. Dynamic multi-dimensional data structures based on quad- and k-d trees. *Acta Informatica*, 17(3):267–285, 1982.

[PS85] Franco P. Peparata and Michael Ian Shamos. *Computational Geometry, An Introduction.* Springer-Verlag, New York, 1985.

[Riv76] Ronald L. Rivest. Partial match retrieval algorithms. *SIAM Journal of Computing*, 5(1):19–50, March 1976.

[Rob81] J.T. Robinson. The k-d-b tree: a search structure for large multidimensional dynamic indexes. In *ACM SIGMOD International Conference on the Management of Data*, pages 10–18, 1981.

[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.

[Sho82] A. Shoshani. Statistical databases: characteristics, problems, and some solutions. In *Proceedings of the 8th International Conference on Very Large Databases (VLDB)*, pages 208–222, 1982.

[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.

[Smi81] Alan Jay Smith. Long term file migration: development and evaluation of algorithms. *Communications of the ACM*, 24(8):521–532, August 1981.

[SOW84] A. Shoshani, A. Olken, and H.K.T. Wong. Characteristics of scientific databases. In *Proceedings of the 10th International Conference on Very Large Databases (VLDB)*, pages 147–160, 1984.

[SS82] Giovanni Sacco and Mario Skolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceedings of the 8th International Conference on Very Large Databases (VLDB)*, pages 257–262, Mexico City, September 1982.

[Sto75] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *ACM SIGMOD International Conference on the Management of Data*, pages 65-78, 1975.

[Sve79] P. Svensson. On search performance for conjunctive queries in compressed, fully transposed ordered files. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 155–163, 1979.

[Tan83] Y. Tanaka. A data-stream database machine with large capacity. In D.K. Hsiao, editor, *Advanced Database Machine Architectures*, Prentice Hall, 1983.

[THC79] M. J. Turner, R. Hammond, and F. Cotton. A dbms for large statistical databases. In *Proceedings of the 5th International Conference on Very Large Databases (VLDB)*, pages 319–327, 1979.

[TUS83a] T. Tsuda, A. Urano, and T. Sato. Transposition of large tabular data structures with applications to physical database organization, part i. *Acta Informatica*, 19:13–33, 1983.

[TUS83b] T. Tsuda, A. Urano, and T. Sato. Transposition of large tabular data structures with applications to physical database organization, part ii. *Acta Informatica*, 19:167–182, 1983.

[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.

[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.

[Wel84] Terry A. Welch. A technique for high performance data compression. *Computer*, 17(6):8–19, June 1984.

[WFS75] Gio Weiderhold, J.F. Fries, and Weyl S. Structured organization of clinical databases. In *Proceedings of the National Computer Conference*, AFIPS Press, May 1975.

[Wie83] Gio Wiederhold. *Database Design*. McGraw Hill, second edition, 1983.

[WL86] H.K.T. Wong and J.Z. Li. Transposition algorithms for very large compressed databases. In *Proceedings of the International Conference on Very Large Databases*, VLDB Endowment, August 1986. (condensed version of LBL-21157D).

[WLO*85] Harry K.T. Wong, Hsiu-Fen Liu, Frank Olken, Doron Rotem, and Linda Wong. Bit transposed files. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 448–457, Stockholm,Sweden, August 1985.

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