The Grid File: An Adaptable, Symmetric Multikey File Structure

J. NIEVERGELT, H. HINTERBERGER
Institut für Informatik, ETH
AND K. C. SEVCIK
University of Toronto

Traditional file structures that provide multikey access to records, for example, inverted files, are extensions of file structures originally designed for single-key access. They manifest various deficiencies in particular for multikey access to highly dynamic files. We study the dynamic aspects of file structures that treat all keys symmetrically, that is, file structures which avoid the distinction between primary and secondary keys. We start from a bitmap approach and treat the problem of file design as one of data compression of a large sparse matrix. This leads to the notions of a *grid partition* of the search space and of a *grid directory*, which are the keys to a dynamic file structure called the *grid file*. This file system adapts gracefully to its contents under insertions and deletions, and thus achieves an upper bound of two disk accesses for single record retrieval; it also handles range queries and partially specified queries efficiently. We discuss in detail the design decisions that led to the grid file, present simulation results of its behavior, and compare it to other multikey access file structures.

Categories and Subject Descriptors: H.2.2 [**Database Management**]: Physical Design; H.3.2 [**Information Storage and Retrieval**]: Information Storage

General Terms: Algorithms, Performance

Keywords and Phrases: File structures, database, dynamic storage allocation, multikey searching, multidimensional data.

1. PROBLEM, SOLUTIONS, PERFORMANCE

A wide selection of file structures is available for managing a collection of records identified by a single key: sequentially allocated files, tree-structured files of many kinds, and hash files. They allow execution of common file operations such as FIND, INSERT, and DELETE, with various degrees of efficiency. Older file structures such as sequential files or conventional forms of hash files were optimized for handling static files, where insertions and deletions are considered to be less important than look-up or modification of existing records. Insertions were usually handled by overflow areas; their growth, however, leads to a

A previous version of this paper appeared in *Trends in Information Processing Systems, Proc 3rd ECI Conference.* Lecture Notes in Computer Science 123.

Authors' addresses: J. Nievergelt and H. Hinterberger, Institut für Informatik, ETH, CH-8092 Zurich, Switzerland; K.C. Sevcik, Computer Systems Research Group, University of Toronto, Toronto, Ontario M5S 1A4 Canada.

progressive degradation of performance, which in practice forces periodic restructuring of the entire file. Modern file structures such as balanced trees or extendible forms of hashing adapt their shape continuously to the varying collection of data they must store, without any degradation of performance. Their discovery was a major advance in the study of data structures.

File processing in today's transaction oriented systems requires file structures that allow efficient access to records, based on the value of any one of several attributes or a combination thereof. The development of file structures that provide multikey access to records repeats the history of single-key structures: earlier schemes, such as inverted files, are extensions of file structures originally designed for single-key access. They do not address the problem of graceful adaptation to highly dynamic files. The design of balanced data structures appears to be significantly more difficult for multidimensional data (each record is identified by several attributes) than for one-dimensional data. This comes as no surprise, since most balanced structures for single-key data rely on a total ordering of the set of key values, and natural total orders of multidimensional data do not exist.

In view of the diversity of file structures for single-key access, one might expect an even greater variety for multikey access. In addition to the traditional inverted file, many other schemes have been proposed: [2, 3, 5, 9, 18, 19, 27, 29, 30, 32] are a representative sample of the techniques known. We review several of these file structures, and their properties, in Section 6: most of them suffer from various deficiencies in a highly dynamic environment. Thus the field is open for improvements, and in this paper we present the grid file as a contribution to the development of balanced multikey file structures.

Consider a file F as a collection of records $R = [a_1, a_2 \ldots, a_k]$, where the a are fields containing attribute values. As an example, consider records with the attribute fields last name, first name, middle initial, year of birth, and social security number, for example [Doe, John, –, 1951, 123456789]. Multikey access means that we reference the records R in file F by using any possible subset of these (key-) fields, as shown in the following examples:

(1) Entire record specified (exact match query, point query)
(2) Doe born in 1951 (a partially specified query)
(3) All records with last name Doe (single-key query)
(4) Social security number 987654321 (presumably unique)
(5) Everybody born between 1940 and 1960 (range or interval query)

Multikey access problems come in two kinds. In information and document retrieval an object (say a book) is characterized by index terms, often chosen from a thesaurus of recommended terms. If we consider each term in the thesaurus to be an attribute, documents become points in a high-dimensional space, but the domain of each attribute is small (perhaps it contains only the two values "relevant" and "irrelevant"). We do not consider this case. We only discuss the other typical case of multikey access, where a record is characterized by a small number of attributes (less than 10), but the domain of each attribute is large and linearly ordered.

For the second case we can specify ranges by expressions r of the form: $l_i \leq a_i \leq u_i$, where l_i and u_i denote lower and upper bounds on attribute value a chosen

from its domain S_i. The point specification $l_i = u_i$ and the "don't care" specification l_i = "smallest value in S_i," u_i = "largest value in S_i," are special cases of range specifications that cover exact matches and partially specified queries. The general form of references we consider is $(r_1, r_2, \ldots, r_k)$, where r_i is a value-range of the ith key-field over the attribute domain S_i. If we abbreviate "don't care" specifications as blanks, we can formulate query (2) of the previous example as (last name = Doe , , , year of birth = 1951), or query (5) as (, , , 1940 ≤ year of birth ≤ 1960,). We consider primarily range queries, but the grid file is applicable to other types of queries as well. For example, [31] applies the related technique of "extendible cells" to closest point problems, and [12] studies the problem of executing relational database queries on the grid file.

We approach the problem of designing a practical multikey access file structure by first considering an extreme solution: the *bitmap representation* of the attribute space, which reserves one bit for each possible record in the space, whether it is present in the file or not. Even though the bitmap representation in its pure form requires impractically large amounts of storage, it points the way to practical solutions based on the idea of data compression.

In a k-dimensional bitmap the combinations of all possible values of k attributes are represented by a bit position in a k-dimensional matrix. The size of the bitmap (number of bit positions) is the product of the cardinalities of the attribute domains. Figure 1 shows a three-dimensional bitmap.

FIND$(r_1, r_2 \ldots, r_k)$ reduces to direct access, INSERT/DELETE requires that a position in the bitmap be set to 1 or 0 respectively, and NEXT in any dimension requires a scan until the next 1 is found. If a sufficiently large memory were available, the bitmap would be the ideal solution to our problem. For realistic applications, however, this bitmap is impossibly large. Fortunately it is sparse (almost all zeros), and hence can be compressed. The sparse matrix compression techniques known in numerical applications are inapplicable, since we need a compression scheme that is compatible with file access operations: FIND, INSERT, and DELETE must be executed efficiently in a compressed bitmap. This we can achieve by introducing a dynamic directory. In maintaining a dynamic partition (directory) on the space of all key-values, we approximate the bit map through compression. This dynamic partitioning is treated in more detail in the next section. The result of this approach is a *symmetric, adaptable* file structure. *Symmetric* means that every key field is treated as the primary key; *adaptable* means that the data structure adapts its shape automatically to the content it must store, so that bucket occupancy and access time are uniform over the entire file, even though the data may be distributed in a highly nonuniform way over the data space.

The efficiency of a file processing system is measured mainly by response times to multikey access requests. The major component of this response time is the time spent in accessing peripheral storage media. In today's systems these storage media are disks, where the maximal amount of data transferred in one access is fixed (a disk block or page). We will therefore assess efficiency in terms of the number of disk accesses. In particular, we aim at file structures that meet the following two principles.

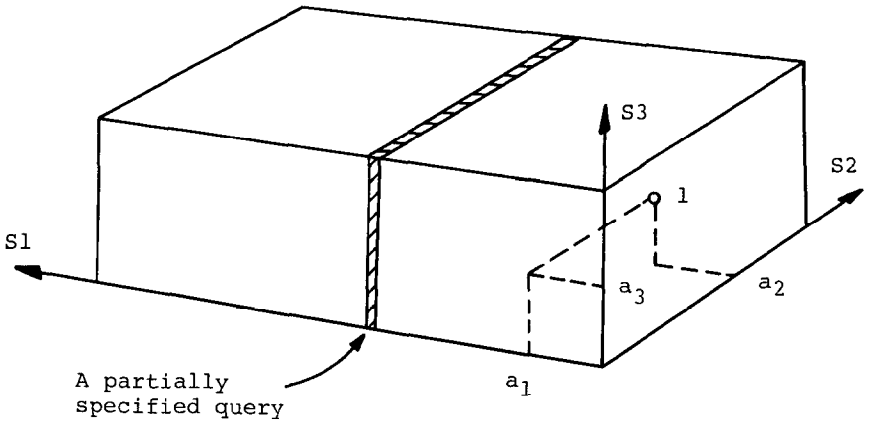


Fig. 1. The three-dimensional bitmap. A "1" indicates the presence of a record with attribute values determined by its position in the map, a "0" indicates absence.

Two-disk-access principle. A fully specified query must retrieve a single record in at most two disk accesses: the first access to the correct portion of the dirctory, the second to the correct data bucket.

Efficient range queries with respect to all attributes. The storage structure should preserve as much as possible the order defined on each attribute domain, so that records that are near in the domain of any attribute are likely to be in the same physical storage block.

Both ideas are as old as disk storage devices: the hope of realizing them led to the traditional index sequential access methods introduced in the late fifties. Since, in a batch processing operation, sequential file access suffices, physical contiguity was dedicated to preserve the order of the primary key alone. And traditional index sequential access techniques find a record specified by a primary key value in one or two disk accesses when the file is newly generated. In practice it turns out, however, that index sequential access techniques, on average, cause more than two disk accesses. This is true for traditional techniques of handling dynamic single-key files by means of chains of overflow buckets. It remained true for balanced trees, which usually require more than two levels for large data collections. The two-disk-access principle for dynamic single-key files was only realized by address computation techniques such as extendible hashing [6]. It was never applied to dynamic multikey files, where each secondary key directory of an inverted file typically introduces an additional disk access.

The reasons why we consider the two principles above important for a modern file system have to do with the expanding spectrum of computer applications:

(1) An interactive system should provide *instantaneous response* to the user's *trivial requests.* In the context of human physiology, *instantaneous* means 1/10th of a second, the limit of resolution of our sense of time. In a typical system 1/10th of a second suffices for a couple of disk accesses, but not for half a dozen. A "trivial request" is often of the form "show me this item," and triggers a fully

specified query. Hence a file system that obeys the two-disk-access principle is a suitable basis for highly interactive applications programs.

(2) Computer-aided design, geographic data processing, and other applications work on geometric objects in two and three dimensions. Complex geometric objects are often decomposed into simple ones, such as rectangles, which can be considered to be points in a higher-dimensional space. Geometric processing generates a lot of intersection and neighborhood queries which translate into range queries along all the dimensions of the space [10].

2. GRID PARTITIONS OF THE SEARCH SPACE

All known searching techniques appear to fall into one of two broad categories: those that *organize the specific set of data* to be stored and those that *organize the embedding space* from which the data is drawn. Comparative search techniques, such as binary search trees, belong to the first category: the boundaries between different regions of the search space are determined by values of data that must be stored, and hence shift around during the life time of a dynamic file. Address computation techniques, including radix trees, belong to the second category: region boundaries are drawn at places that are fixed regardless of the content of the file; adaptation to the variable content of a dynamic file occurs through activation or deactivation of such a boundary. In recent years search techniques that organize the embedding space rather than the specific file content have made significant progress (see [23] for a survey).

Each search technique partitions the search space into subspaces, down to the "level of resolution" of the implementation, typically determined by the bucket capacity. Much can be learned by comparing the partitioning patterns created by different search techniques, regardless of how the partition is implemented. Let us consider three examples, keeping in mind that we only consider the case where the domain of each attribute is large and linearly ordered (for small domains, such as Boolean ones, the space partitioning analogy is not helpful).

Multidimensional trees of various kinds (e.g., [2]) are an example of multikey access techniques that organize the set of data to be stored. The recursively applied divide and conquer principle leads to a partition of the search space as illustrated in Figure 2(a), where region boundaries become progressively shorter. The traditional inverted file, with its asymmetric treatment of primary key and secondary keys, is a hybrid: Values of the primary key determine region boundaries so as to achieve a uniform bucket occupancy, whereas the domain of each secondary key is "partitioned" independently of the data to be stored—to the extreme level of resolution where each value of the secondary key domain is in its own region. Figure 2(b) shows the resulting partition. The grid file presented in this paper is based on grid partitions of the search space illustrated in Figure 2(c). Each region boundary cuts the entire search space in two, but, unlike the inverted file, all dimensions are treated symmetrically. It turns out that the most efficient implementations of grid partitions are obtained by drawing the boundary lines at fixed values of the domain. Hence, grid partitions are an example of techniques that organize the embedding space.

The utility of the different space partitions mentioned above depends on the distribution of data. For the grid partition we assume independent attributes,

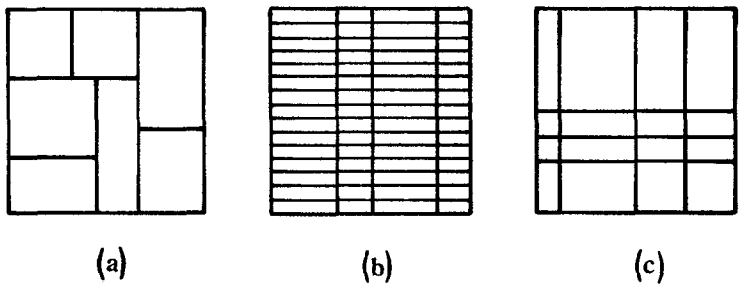


Fig. 2. Space partitions created by different search techniques.

but we do not assume a uniform distribution. Correlated attributes, such as data points that lie on a diagonal, prevent full utilization of a grid partition's discriminatory power. Correlated attributes may mean that attributes are functionally dependent on each other (age and year of birth, for instance), in which case searching becomes more efficient if only some of the attributes are chosen as dimensions of the search space. Or it may mean that we have not chosen the most suitable attributes for searching, and that a transformation of a coordinate system may result in independent attributes (see [10] for an example in the context of a geometric database).

Assuming independent attributes, the grid partition of the search space is obviously well suited for range and partially specified queries. It exhibits some striking advantages over the other types of partitions shown in Figure 2, such as systematic region boundaries and economy of representation (one boundary line of Figure 2(c) does the work of many in Figure 2(a)).

We introduce the following terminology and notation for the three-dimensional case: generalization to k dimensions is obvious. On the record space $S = X \times Y \times Z$ we obtain a grid partition $P = U \times V \times W$ by imposing intervals $U = (u_0, u_1, \ldots, u_l)$, $V = (v_0, v_1, \ldots, v_m)$, $W = (w_0, w_1, \ldots, w_n)$ on each axis and dividing the record space into blocks, which we call grid blocks, as shown in Figure 3.

During the operation of a file system the underlying partition of the search space needs to be modified in response to insertions and deletions. For the grid partition we introduce operations that refine the granularity by splitting an interval, and render it coarser by merging two adjacent intervals.

Partition modification. The grid partition $P = U \times V \times W$ is modified by altering only one of its components at a time. A one-dimensional partition is modified either by splitting one of its intervals in two, or by merging two adjacent intervals into one. Figure 3 shows this for the partition V. Notice that the intervals "below" the one being split or the two being merged retain their index (v in Figure 3), while the indices of the intervals "above" the point of splitting or merging are shifted by +1 or −1, respectively ($v_2 \leftrightarrow v_3$ in Figure 3).

In order to obtain a file system, we will need other operations that relate grid blocks and records to each other, such as: find the grid block in which a given record lies, or list all records in a given grid block. The regularity of the grid partition makes the implementation of such operations straightforward: they are reduced to the separate maintenance of one-dimensional partitions. Thus the

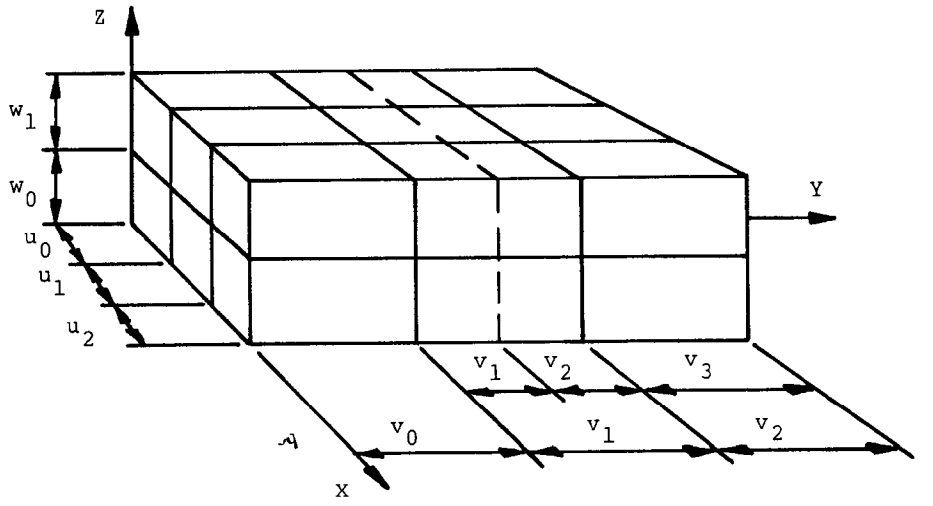


Fig. 3. A three-dimensional record space $X \times Y \times Z$, with a grid partition $P = U \times V \times W$. The picture shows the effect of refining P by splitting interval v_1.

operations that modify the partition emerge as crucial, in the sense that they impose severe constraints on an efficient representation.

3. THE GRID FILE

A file structure designed to manage a disk allocates storage in units of fixed size, called disk blocks, pages, or buckets, depending on the level of description. We use *bucket* for a storage unit that contains records. We assume an unlimited number of them, each one of capacity c records. The case of very small bucket capacities, such as $c = 1$, is not of great interest; other file structures should be designed to handle it. We are interested in the practical range of, say, $10 < c < 1000$, where an entry in the grid directory is tiny compared to a bucket, and where undesirable probabilistic effects such as the "birthday paradox" (high probability that two records cause a bucket to overflow) are less likely. Differences in access time to different buckets are ignored, hence the time required for a file operation can be measured by the number of disk accesses.

The data structure used to organize records within a bucket is of minor importance for the file system as a whole. Often the simplest possible structure, sequential allocation of records within a bucket, is suitable. The structure used to organize the set of buckets, on the other hand, is the heart of a file system. For the grid file, the problem reduces to defining the correspondence betwen grid blocks and buckets: This assignment of grid blocks to buckets is the task of the grid directory, to be described in Section 3.1. In order to obtain an efficient file structure, constraints on access time, update time, and on memory utilization must be met. In particular, we aim at

- the two-disk-access principle for point queries,
- efficient processing of range queries in large linearly ordered domains,

- splitting and merging of grid blocks to involve only two buckets,
- maintaining a reasonable lower bound on average bucket occupancy.

The first three points determine processing efficiency; they are discussed in Sections 3.2 and 3.3, respectively. The fourth point, on memory utilization, is discussed by means of simulation results in Section 5.

3.1 The Grid Directory: Function and Structure

In order to obtain a file system based on the grid partitions described in Section 2, we must superpose a bucket management system onto these partitions. In our case the design of a bucket management system involves three parts:

- defining a class of legal assignments of grid blocks to buckets,
- choosing a data structure for a directory that represents the current assignment,
- finding efficient algorithms to update the directory when the assignment changes.

In this section we discuss the first two points, concerned with function and structure of the directory, respectively.

The purpose of the grid directory is to maintain the dynamic correspondence between grid blocks in the record space and data buckets. Hence we must define the class of legal assignments of grid blocks to buckets before we can design a data structure. Reasons of efficiency dictate that only a subset of all possible assignments of grid blocks to buckets be allowed, characterized by the constraint that bucket regions must be convex.

The two-disk-access principle implies that all the records in one grid block must be stored in the same bucket. Unfortunately, we cannot insist on the converse: if each grid block had its own data bucket, bucket occupancy could be arbitrarily low. Hence it must be possible for several grid blocks to share a bucket: we call the set of all grid blocks assigned to the same bucket B (or equivalently, the space spanned by these grid blocks) the *region* of B. The shape of bucket regions clearly affects the speed of at least the following two operations:

- range queries, and
- updates following a modification of the grid partition.

Given our emphasis on efficient processing of range queries, and given the earlier decision to base the file system on grid partitions of the record space, there appears to be no other choice than to insist that bucket regions have the shape of a box (i.e., a k-dimensional rectangle). We call such an assignment of grid blocks to buckets *convex*. Figure 4 shows a typical convex assignment of grid blocks to buckets. Each grid block points to a bucket. Several grid blocks may share a bucket, as long as the union of these grid blocks forms a rectangular box in the space of records. The regions of buckets are pairwise disjoint, together they span the space of records.

In order to represent and maintain the dynamic correspondence between grid blocks in the record space and data buckets, we introduce the *grid directory*: a data structure that supports operations needed to update the convex assignments

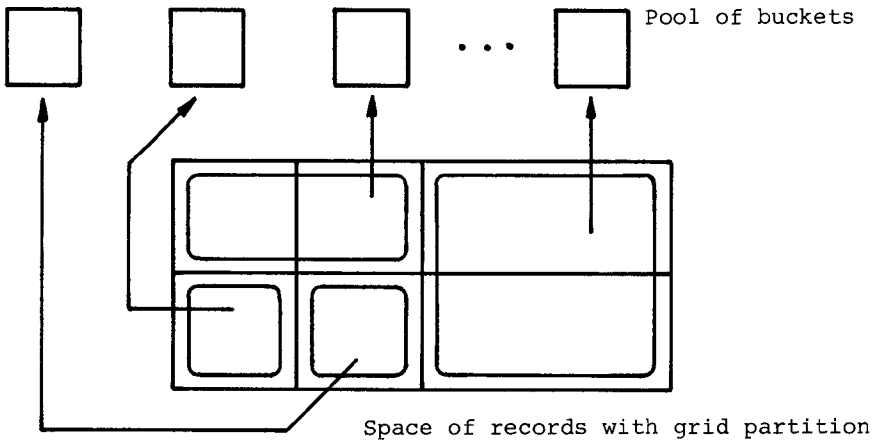


Fig. 4. A convex assignment of grid blocks to buckets.

defined above when bucket overflow or underflow makes it necessary. The choice of just how much to specify about this data structure and how much to leave for later implementation decisions is discussed in Section 4.1. We define the grid directory at a fairly abstract level, fixing only those decisions that appear essential. Different implementations are discussed in Section 4.

A grid directory consists of two parts: first, a dynamic k-dimensional array called the *grid array*; its elements (pointers to data buckets) are in one-to-one correspondence with the grid blocks of the partition; and second, k one-dimensional arrays called *linear scales*; each scale defines a partition of a domain S.

For the sake of notational simplicity we present the case $k = 2$, with record space $S = X \times Y$, from which the general case $k > 2$ is easily inferred.

A grid directory G for a two-dimensional space is characterized by

- Integers $nx > 0$, $ny > 0$ ("extent" of directory).
- Integers $0 \leq cx < nx$, $0 \leq cy < ny$ ("current element of the directory and current grid block").

It consists of

- a two-dimensional array $G(0 \ldots, nx - 1, 0 \ldots, ny - 1)$ ("grid array") and
- one-dimensional arrays $X(0 \ldots, nx)$, $Y(0 \ldots, ny)$ ("linear scales").

Operations defined on the grid directory consist of

- Direct access: $G(cx, cy)$
- Next in each direction

$$\begin{aligned}
&\text{nextxabove: } cx \leftarrow (cx + 1) \bmod nx \\
&\text{nextxbelow: } cx \leftarrow (cx - 1) \bmod nx \\
&\text{nextyabove: } cy \leftarrow (cy + 1) \bmod ny \\
&\text{nextybelow: } cy \leftarrow (cy - 1) \bmod ny
\end{aligned}$$

- Merge
  mergex: given px, $1 \leq px < nx$, merge px with next$x$below; rename all elements above px;
    adjust X-scale.
  mergey: similar to merge$x$ for any py, $1 \leq py < ny$.
- Split
  split$x$: given px, $0 \leq px \leq nx$, create new element $px + 1$ and rename all cells above px;
    adjust X-scale.
  split$y$: similar to split$x$ for any py, $0 \leq py \leq ny$.

Constraints on the values. The restriction to convex assignments of grid blocks to buckets is expressed by the following constraints on the values of grid directory elements:

if $G(i', j') = G(i'', j'')$ then for all i, j with $i' \leq i \leq i''$ and $j' \leq j \leq j''$ we have $G(i', j') = G(i, j) = G(i'', j'')$.
Some splitting and merging policies (see Sections 4.2 and 4.3) restrict the set of assignments that may arise during operation of the grid file to some subset of all convex assignments.

3.2 Record Access

The description of the grid directory in Section 3.1, abstract as it may be, suffices to justify a key assumption on which the efficiency of the grid file is based: the array G is likely to be large and must be kept on disk, but the linear scales X and Y are small and can be kept in central memory.

This assumption suffices for the two-disk-access principle to hold for fully specified queries, as the following example shows. Consider a record space with attribute "year" with domain $0 \cdots 2000$, and attribute "initial" with domain $a \cdots z$. Assume that the distribution of records in the record space is such as to have caused the following grid partition to emerge.

$$X = (0, 1000, 1500, 1750, 1875, 2000); \quad Y = (a, f, k, p, z).$$

A FIND for a fully specified query $(r_1, r_2 \ldots,)$, such as FIND [1980, w], is executed as shown in Figure 5.

The attribute value 1980 is converted into interval index 5 through a search in scale X, and w is converted into the interval index 4 in scale Y. For realistic granularities of these partitions, these linear scales are stored in central memory; thus the conversion of attribute value to interval index requires no time in our model, where only the number of disk accesses count. The interval indices 5 and 4 provide direct access to the correct element of the grid directory, where the bucket address is located. Even if only part of the grid directory can be read into central memory in one disk access, the correct page (the one that contains the desired bucket address) can easily be computed from the interval indices. Range queries, including the special case of partially specified queries, are also handled efficiently by the grid file. In information retrieval the following notion of

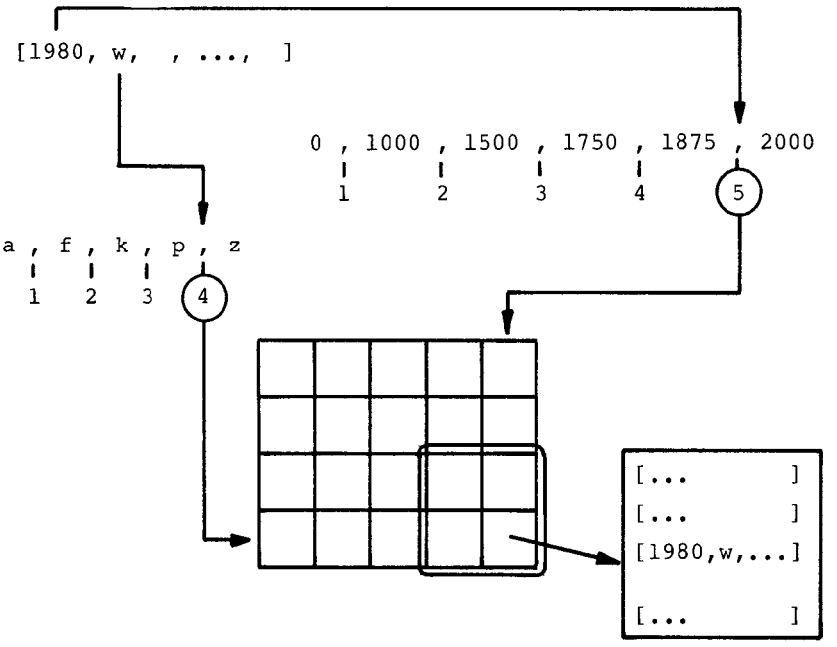


Fig. 5. Retrieval of a single record requires two disk accesses.

"precision" of an answer to a query is well known:

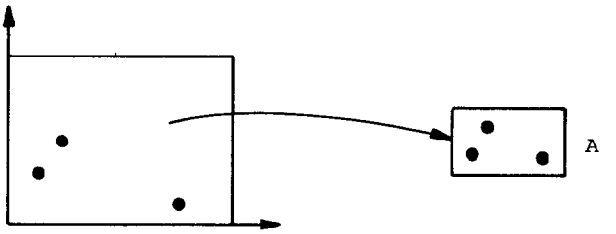
$$\frac{\text{number of records retrieved that meet the query specification}}{\text{total number of records retrieved}}$$

Figure 6 illustrates the fact that the precision of most range queries is high. In particular, precision approaches 1 for queries that retrieve many records (as compared to bucket capacity).

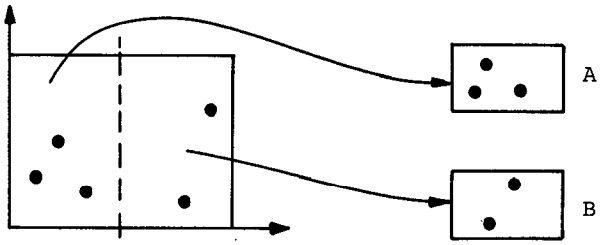
3.3 Dynamics of the Grid File

The dynamic behavior of the grid file is best explained by tracing an example: that is, building up a file under repeated insertions. When deletions occur, merging operations get triggered. In order to simplify the description, we present the two-dimensional case only. Instead of showing the grid directory, whose elements are in one-to-one correspondence with the grid blocks, we draw the bucket pointers as originating directly from the grid blocks.

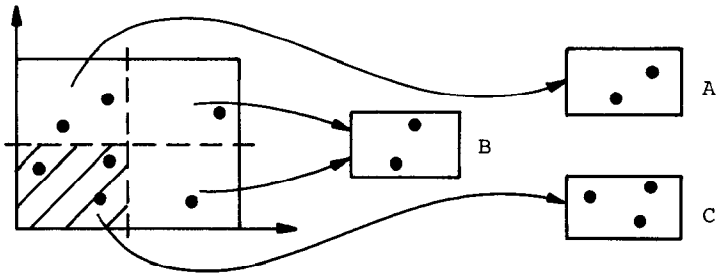
Initially, a single bucket A, of capacity $c = 3$ in our example, is assigned to the entire record space.



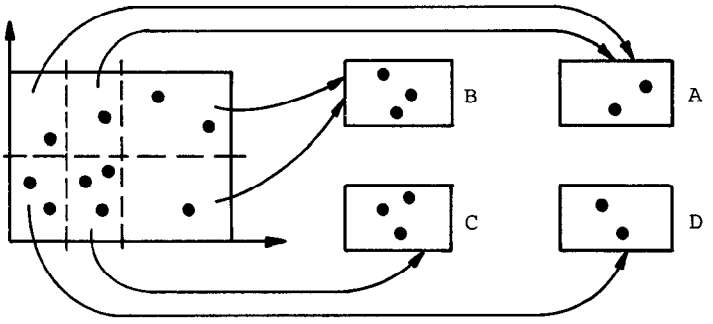
When bucket A overflows, the record space is split, a new bucket B is made available, and those records that lie in one half of the space are moved from the old bucket to the new one.



If bucket A overflows again, its grid block (i.e., the left half of the space) is split according to some splitting policy: we assume the simplest splitting policy of alternating directions. Those records of A that lie in the shaded lower-left grid block of the figure below are moved to a new bucket C. Notice that, as bucket B did not overflow, it is left alone: its region now consists of two grid blocks. For effective memory utilization it is essential that in the process of refining the grid partition we need not necessarily split a bucket when its region is split.



Assuming that records keep arriving in the lower-left corner of the space, bucket C will overflow. This will trigger a further refinement of the grid partition as shown below, and a splitting of bucket C into C and D.



The history of repeated splitting can be represented in the form of a binary tree, which imposes on the set of buckets currently in use (and hence on the set of regions of these buckets) a *twin system*: each bucket and its region have a unique twin from which it is split off. In the picture above, C and D are twins,

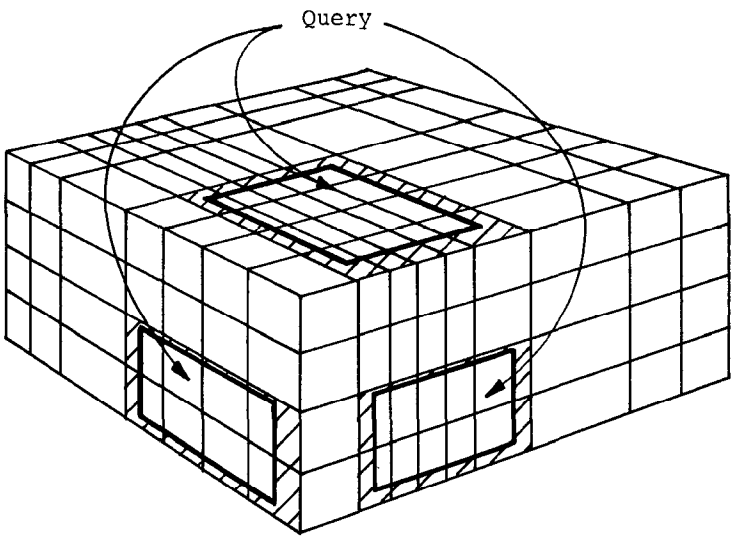


Fig. 6. A range query causes irrelevant records to be retrieved only at the fringes of the answer.

the pair (C, D) is A's twin, and the pair $(A, (C, D))$ is B's twin. Merging, needed to maintain a high occupancy in the presence of deletions, can proceed from the leaves up in the twin system tree. In one-dimensional storage the twin system (also called the "buddy system") created by repeated splits indeed suffices for obtaining a high memory occupancy. In more than one dimension, however, this is not so (in the example above, consider the case where buckets A and B have become almost empty, yet are unable to merge because the pair C and D cannot be merged). Hence, the merging policies to be discussed in Section 4.3 merge buckets that never split from each other.

4. ENVIRONMENT-DEPENDENT ASPECTS

4.1 What to Specify and What to Leave Open

There is a difference between writing a software package for a specific application and designing a general-purpose data structure. In the first case we exploit our knowledge of the intended application (e.g., known size of database, known data distribution, known query frequencies) to obtain an efficient dedicated system. In the second case we postpone all inessential decisions so as to obtain a general design that can be tailored to a specific environment by the implementor.

With the goal of presenting the grid file as a general-purpose file structure suited to multikey access, we have followed the second course of action and specified only those decisions that we consider essential, namely

- *grid partitions* of the search space,
- assignments of grid blocks to buckets that result in *convex* (*box-shaped*) *bucket regions*,
- *grid directory*, consisting of a (possibly large) dynamic array but small linear scales.

Some other important decisions have been left open because we feel that they can be settled in many different ways within the framework set by the decisions above. In this section we discuss the most important open issues, namely

- choice of splitting policy,
- choice of merging policy,
- implementation of the grid directory,
- concurrent access.

4.2 Splitting Policy

Several splitting policies are compatible with the grid file; they result in different refinements of the grid partition. The implementor, or perhaps even the user of a sufficiently general grid file implementation, may choose among them in an attempt to optimize performance on the basis of query frequencies observed in his application.

A refinement of the grid partition gets triggered by the overflow of a bucket, all of whose records lie in a single grid block. Its occurrence is relatively rare: the majority of all overflows involve buckets whose records are distributed over several grid blocks and can be handled by a mere bucket split without any change to the partition. If a partition refinement does occur, there is a choice of dimension (the axis to which the partitioning hyperplane is orthogonal) and location (the point at which the linear scale is partitioned).

The simplest splitting policies choose the dimension according to a fixed schedule, perhaps cyclically. A splitting policy may favor some attribute(s) (in the sense of a linear scale of higher resolution) by splitting the corresponding dimension(s) more often than others. This has the effect of increasing the precision of answers to partially specified queries in which the favored attribute(s) is specified, but others are not.

The location of a split on a linear scale need not necessarily be chosen at the midpoint of the interval, as we have described in Section 3. Little is changed if the splitting point is chosen from a set of values that are convenient for a given application—months or weeks on a time axis, feet or inches on a linear scale used to measure machine parts.

4.3 Merging Policy

Merging occurs at two levels: bucket merging and merging of cross sections in the grid directory. Directory merging is rarely of interest. Although the directory could shrink when two adjacent cross sections have identical values, in most applications, it is unwarranted to reduce directory size as soon as possible, as in a steady-state or in a growing file it will soon grow back to its earlier size. There are only two exceptions: in a shrinking file and in "dynamic weighting of attributes" (to be discussed later) directory merging occurs.

Bucket merging, on the other hand, is an integral part of the grid file. Different policies appear to yield reasonable performance, with a possible trade-off between time and memory utilization. A merging policy is controlled by three decisions: *which pairs* of adjacent buckets *are candidates* for merging their contents into a single bucket; among several eligible pairs, *which one has priority*; and the *merging threshold* that determines at what bucket occupancy merging is actually triggered.

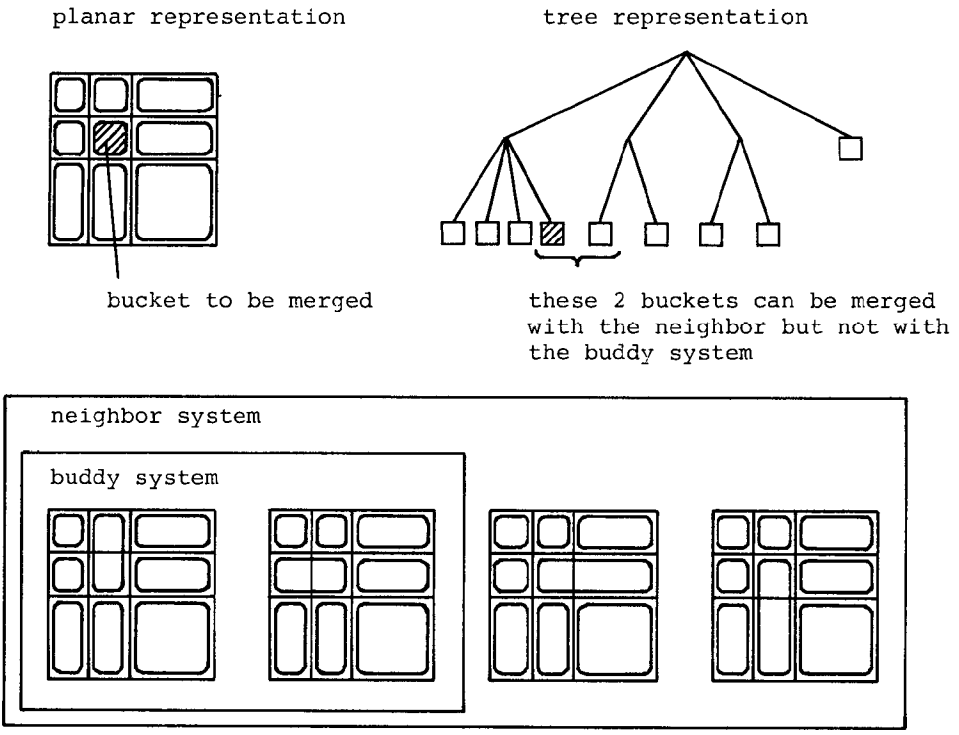


Fig. 7. Possible modifications to the grid directory when merging in the buddy or neighbor systems.

We have used two different systems for determining pairs of buckets that can merge. In the (k-dimensional) "buddy system," a bucket can merge with exactly one adjacent buddy in each of the k dimensions. The assignment of grid blocks to buckets is such that buddies can always merge if the total number of records fits into one bucket. The buddy system is easily implemented, and we recommend it as the standard merging policy. We have also experimented with a more general "neighbor system," in which a bucket can merge with either of its two adjacent neighbors in each of the k dimensions, provided the resulting bucket region is convex. The neighbor system can generate all convex assignments of grid blocks to buckets, whereas the buddy system only generates a subclass that is best described by means of recursive halving and is represented by the "quad tree" in Figure 7. The simulation experiments of Section 5 show that both systems give a reasonable performance.

The second decision, that of which axis to favor when it is possible to merge along several axes, is only relevant when the granularity of the partitions along the different axes happens to be different. For example, if the splitting policy favors some attribute(s) by splitting the corresponding dimension(s) more often than others, the merging policy must not undo this effect by merging more often along these same axes.

The third decision, of setting a merging threshold of p percent, with the interpretation that the contents of two mergeable buckets are actually merged if

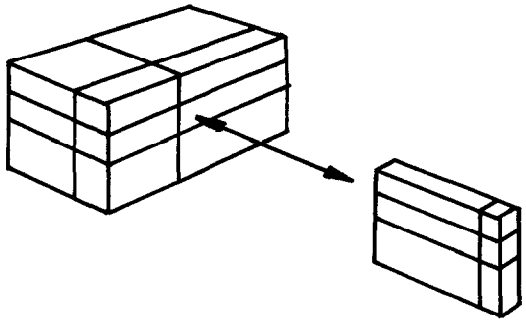


Fig. 8. Insertion and removal of an arbitrary $(k - 1)$-dimensional cross section.

the occupancy of the result bucket is at most p percent, has a clear answer: Thresholds of around 70 percent are reasonable, those above 80 percent lead to poor performance.

4.4 Implementation of the Grid Directory

A grid directory behaves like a k-dimensional array with respect to the operations of *direct access* and *next*, but the insertion and removal of arbitrary $(k - 1)$-dimensional cross sections (corresponding to hyperplanes in the record space) is an unconventional operation that is difficult to reconcile with direct access (see Figure 8). What data structure should be chosen to implement a grid directory?

It is tempting to design exotic data structures that allow fast insertion and removal of arbitrary $(k - 1)$-dimensional cross sections. Let us mention a few and assess their practicality.

Linked lists are prime candidates for representing any structure where insertions and deletions may occur at arbitrary positions. Since they require the traversal of pointer chains to find a desired element, fast access is only guaranteed if these chains reside within the same page or disk block. A grid directory of several dimensions easily extends over several pages, however. Moreover, a list representation of the directory has the disadvantage of introducing a space overhead of k pointers (one or two for each dimension is the most direct way of representing the connectivity among gridblocks), which is significant compared to the normal content of a directory element (typically, a single disk address and a small amount of status information of the bucket located at that address, such as an occupancy number). It is doubtful whether the overhead caused by pointers is justified for an infrequent operation such as a directory split.

If one is willing to incur a significant space overhead, a better idea might be to represent the grid directory by a k-dimensional array whose size is determined solely by the shortest interval in each linear scale, as shown in Figure 9. This technique is the multidimensional counterpart of the directory used in extendible hashing [6]. A refinement of the grid partition causes a change in the structure of the directory only if a shortest interval is split, in which case the directory doubles in size. This data structure anticipates several small structural updates and attempts to replace them by a single large one. The strategy is successful

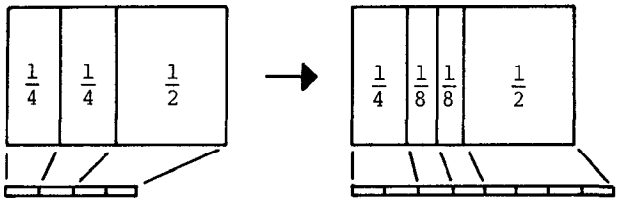


Fig. 9. Representation of the grid directory by an array that occasionally doubles in size.

when data is uniformly distributed, but may lead to an extravagantly large directory when it is not. In extendible hashing the uniformity is provided by a randomizing hash function, even if the data is not uniformly distributed over the record space. Since the grid file is designed to answer range queries efficiently, and randomizing functions destroy order, this approach cannot be used to generate uniformly distributed data. Hence any nonuniformity in the data leads to an oversized directory, and this approach cannot be recommended in general.

An interesting representation of the grid directory that avoids the space overhead of the "doubling array" and yet permits a $(k - 1)$-dimensional update to be written contiguously, rather than being scattered throughout the k-dimensional old array, is described in [12]. A timestamp is attached to the update that allows correct computation of the address of a directory element. Address computation becomes complicated when many such patches are superposed, so the directory must be reorganized periodically. The option of postponing a change to the existing directory, from the moment a partition refinement is needed to some later time, is useful in a concurrent access or real-time environment.

In most applications the split and merge of the directory occur rarely as compared to *direct access* and *next*; thus the conventional allocation of a multidimensional array is sufficient. Split and merge operations may cause a rewrite of the entire directory, and hence take longer than they would in a list, but the old directory can be used to access data while the new one is being written. Moreover, direct access is faster and memory utilization is optimal, thus a larger fraction of the directory may be kept in central memory. We feel these advantages of conventional array allocation outweigh the penalty in processing time.

An attempt to make optimal use of available central memory leads to the scheme of managing the grid directory on disk with a small *resident grid directory* in central memory (Figure 10). Even on a small computer, it is worthwhile to use more auxiliary information in central memory than just linear scales to facilitate access to disk ([8] is an interesting study of how a small amount of internal storage can be used to save disk accesses). The resident grid directory is a scaled down version of the real one, in which the limit of resolution is coarser. An implementation of the grid file used for storing geometric objects uses this fact to answer queries about intersection or distance of objects with few disk accesses [10].

In summary, many ways of implementing the grid directory are possible. Conventional array allocation is the simplest and is adequate, the resident grid directory technique has the best performance when neighborhood relations in all

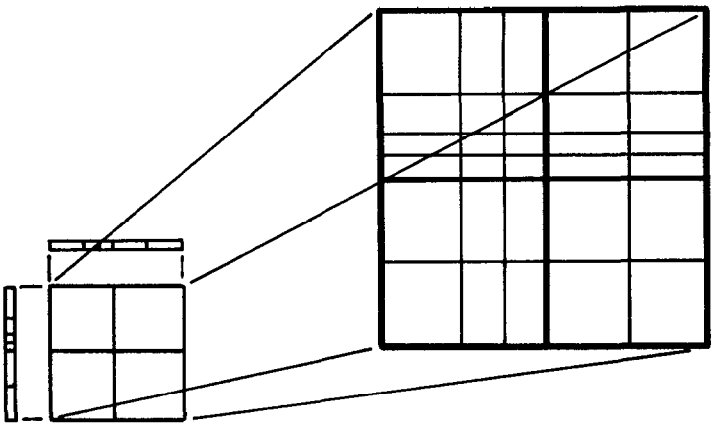


Fig. 10. A small "resident grid directory" manages the large grid directory on disk.

dimensions are important. Regardless of how the grid directory is implemented, it can be supplemented by auxiliary access structures such as directories for attributes not chosen as dimensions of the search space, at the usual cost to be paid in inverted files—directory updates may become a lot more time consuming.

4.5 Extensions: Dynamic Weighting of Attributes, Concurrent Access

Let us mention two possible extensions of the grid file design that appear to be easily incorporated into the structure described so far.

Dynamic weighting of attributes. Transaction environments change. In particular, the query profile that determines access frequencies to individual records in the file may change to such an extent that a new data organization, optimized with respect to the new access frequencies, may enhance the performance of an interactively accessed database. Many multiattribute file organizations have been studied from the point of view of optimal performance of a static file under a fixed access frequency profile (e.g., static multiattribute clustering). Recent efforts aim at extending this clustering to dynamic files, but we are unaware of methods that can adapt to both a dynamic file content and a time-varying query profile.

The grid file permits a dynamic adaptation of its structure to a time-varying access frequency profile by the same technique used to adapt to a varying file content: a dynamic modification of the grid partition. The splitting and merging policies discussed above make it possible to change dynamically the parameters that govern the granularity of the attribute scales. As the granularity of the partition is directly related to the precision of the answers, these parameters can be altered to favor frequent queries.

Adaptation of the splitting and merging parameters can be done automatically by monitoring the query environment and feeding this information into a table-driven grid file. In the extreme, an inactive attribute can be set to a "merge-only" state, whereupon in a dynamic file it will gradually fade away. When its partition

has been reduced to a single interval, the corresponding dimension in the grid directory can be removed or assigned to another attribute.

Concurrent access. An increasing number of applications, such as information or reservation systems, require concurrent access to a file system. Concurrency control is complicated in tree structures because the root is a bottleneck shared by all access paths. If a process has the potential of modifying the data structure near the root (such as an insertion or deletion in a balanced tree), other processes may be slowed down by the observance of locking protocols even if they access disjoint data. The grid file (and other structures based on address computation, see [11] and [23]) has the property that access paths to separate buckets are disjoint, thus allowing simpler concurrency control protocols.

5. SIMULATION EXPERIMENTS

The performance of a file system is determined by two criteria: processing time and memory utilization. The grid file is designed to economize disk accesses, and we must show that this overriding concern for speed is compatible with a reasonable utilization of available space. A theoretical analysis of grid file behavior appears to be difficult for two reasons: many of the techniques developed for analyzing single-key data structures do not directly generalize to their multidimensional counterparts, and the grid file has parameters that are complicated to capture in a mathematical model (such as different splitting and merging policies). For these reasons we resort to simulation.

Our experience with grid file performance is based on three programs, two written in Pascal and one in Modula-2. The first program runs on a DEC-10 under the TOPS-10 operating system. It is a simulation program of 600 Pascal lines, implementing both the buddy and the neighbor systems of splitting and merging. The buddy system requires 150 lines of source code for the splitting and 130 lines for the merging operations. The corresponding figures for the neighbor system are 160 and 220 lines respectively. The second program runs on an APPLE III personal computer under UCSD Pascal. It supports a six-dimensional grid file and consists of approximately 1600 lines of Pascal source code. About 300 lines each are required by SPLIT and MERGE, 150 lines are used up by FIND, INSERT, and DELETE, the rest is devoted to dialog and housekeeping operations. The third grid file program, due to K. Hinrichs, is written in Modula-2 and runs on a Lilith personal computer. It has an interactive graphics interface and is used for storing geometric objects and answering intersection queries [10].

5.1 Objectives and Choice of Simulation Model

The simulation runs described below had the following objectives:

(1) estimation of average bucket occupancy,
(2) estimation of directory size,
(3) visualization of the geometry of bucket regions,
(4) evaluation of splitting and merging policies.

Since the grid file is designed to handle large volumes of data, (1) is by far the most important point. Average bucket utilization need not be close to 100 percent,

but it must be prevented from becoming arbitrarily small under any circumstances. Point (2) is of greater theoretical than practical importance; we don't know the asymptotic growth rate of directory size, but for realistic file sizes the grid directory tends to require only a fraction of the space required by data storage, as an entry in the grid directory ranges from a few bytes (for a disk address) to a few dozen bytes (if additional information is stored, such as a record count or locking information in a concurrent access environment). Point (3) is merely a confirmation of what one expects from the way the grid file was designed, namely that grid partitions and bucket regions adapt their size and shape to data clusters. Point (4) covers space/time trade-offs and is discussed in Sections 5.3 and 5.4, in which we treat steady-state and shrinking files.

Among the many types of loads that may be imposed on a file, the following are particularly suitable as benchmarks for testing and comparing performances:

- the *growing file* (repeated insertions),
- the *steady-state file* (in the long run there are as many insertions as deletions, so the number of records in the file is kept approximately constant),
- the *shrinking file* (repeated deletions).

We have tested the behavior of the grid file with two simulation programs. One for the three-dimensional case of a growing file, the other for the two-dimensional case (for ease of displaying results graphically) under all three types of loads mentioned above. The justification for restricting our experiments to two and three dimensions is that the bucket occupancy (the primary objective of our simulation) appears to be largely independent of the dimensionality of the record space. For a growing file this is plausible on a priori grounds: buckets are split when they are full, regardless of the nature of their contents and independent of different splitting policies. In fact, the average bucket occupancy for $k = 2$ and $k = 3$ turn out to be the same. With respect to merging, one can readily see that a bucket has more buddies to merge, the more dimensions there are; thus bucket occupancy will not be worse in higher dimensional grid files.

The sample spaces used in the experiments are as follows: attribute values of each record are chosen independently of each other from uniform and piecewise uniform one-dimensional distributions to obtain uniform and nonuniform data distributions over the record space. Two standard, integer-valued random number generators from a program library were used.

5.2 The Growing File

Average bucket occupancy. We observed the average bucket occupancy while inserting 10,000 records from a two-dimensional uniform distribution. Figure 11 shows two typical curves depicting the average bucket occupancy over time, one for bucket capacity $c = 50$, the other for $c = 100$. As soon as the number n of inserted records reaches a small multiple of the bucket capacity c, average bucket occupancy shows a steady state behavior with small fluctuations of around 70 percent. It is tempting to conjecture that it approaches asymptotically the magical value $\ln 2 = 0.6931\ldots$, which often shows up in theoretical analyses of processes that repeatedly split a set into two equiprobable parts (see also [6]).

In Section 4 we mentioned splitting policies that do not necessarily refine a

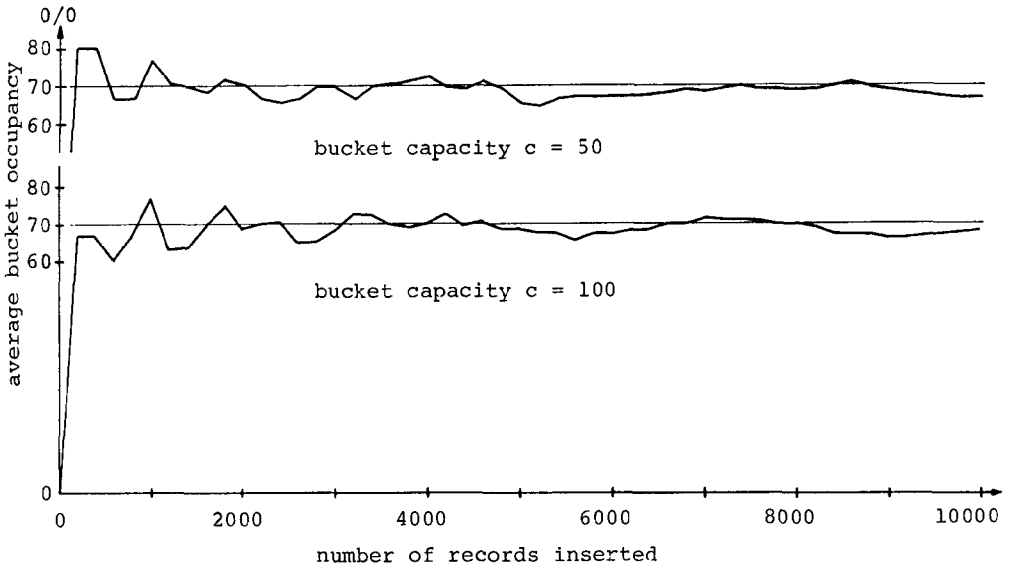


Fig. 11. Average bucket occupancy of a continuously growing grid file.

partition at interval midpoints; for example, a ternary system might always split an interval into three thirds. If such a policy also splits an overflowing bucket into three, then average bucket occupancy drops to 39 percent. Thus the advice: it is possible to use different splitting policies at a moderate increase in the size of the directory, but it is impractical to depart from the rule of splitting one bucket into two.

Growth of the directory. The constant average bucket utilization observed above implies a linear growth of the number of buckets with the amount of data stored. Since a bucket may be shared by many grid blocks, each of which requires its own entry in the grid directory, the question remains open as to how fast the directory grows with the amount of data stored. The number of directory entries per bucket is a good measure of the efficiency of the grid directory.

The assumption of independent attributes is crucial for the size of the directory. Correlated attributes, for example $y = a*x$, are unlikely to significantly affect average bucket occupancy, but they are very likely to increase directory size substantially. Even in the case of independent attributes, the asymptotic growth rate of directory size as a function of the number of records is unknown to us. As an example of the problem, consider random shots into the unit square as illustrated in Figure 12. To model the case of bucket capacity $c = 1$, we divide any grid block that gets hit twice into two halves, alternating directions repeatedly if necessary, until every grid block contains at most one point. The grid file is obviously not immune to the worst-case catastrophe that may strike all address computation techniques, namely that all points come to lie within a tiny area. Conventional practice in hashing ignores this worst case, as it is very unlikely. Another well-known probabilistic effect, however, the birthday paradox, is likely to happen: even if the number of records (people) is much smaller than the

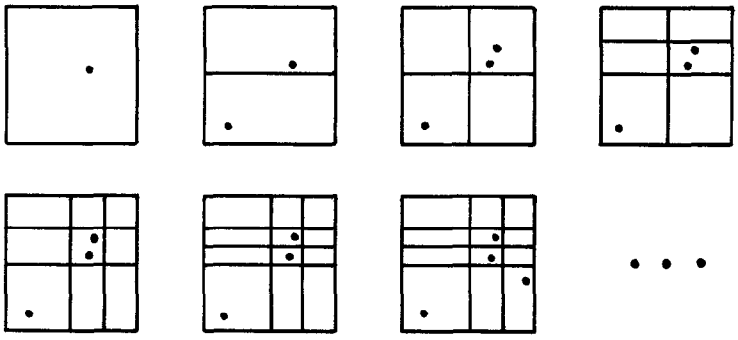


Fig. 12. Random shots into unit square trigger repeated halvings.

number of grid blocks (days in a year), the probability of two or more points colliding in some grid block (having a common birthday) is high. We conjecture that, in this model, the expected number of grid blocks grows faster than linearly in the number of points shot into the unit square. The point at which a superlinear growth rate begins to be noticeable, however, depends strongly on bucket capacity c, and turns out to be sufficiently large so that grid directory size is modest for practical values of n and c.

Figure 13 shows the number of buckets and the number of grid blocks during insertion of 10,000 records from a two-dimensional uniform distribution into buckets of capacity $c = 100$. The "straight line" depicting the number of buckets has a slope of 70 percent, as expected. The number of grid blocks also appears to grow linearly, but the fluctuations from a straight line (oscillating between one and two directory entries per bucket) have a larger period and amplitude.

This "staircase phenomenon" also occurs in extendible hashing; intuitively, it can be explained as follows. When records are inserted from a space with uniform distribution there are moments when practically all grid blocks have equal size, and almost every grid block has its own bucket. Under the assumption of uniformity (which is essential to this argument!), within a short time span a few buckets whose regions are randomly chosen from the entire record space will overflow; the resulting partition refinements affect all parts of the space, leading to a rapid increase in the number of grid blocks. At this moment the directory has a lot of spare capacity to accomodate further insertions, buckets get split without triggering a partition refinement, until we are back to a "one-grid-block-per-bucket" state, but with a directory that has doubled in size.

Figure 14 shows an experiment to determine the influence of bucket capacity on directory growth by plotting the number of grid blocks per bucket as a function of the normalized number n/c of records. The dashed line connects points where the directory has grown to 40,000 entries. 200,000 records packed into buckets of capacity 20 require a directory with only 2 entries per bucket. Given the small size of a directory entry (small compared to a bucket), we consider an average of 10 directory entries per bucket to be a modest investment. With $c = 1$, the birthday paradox causes this value to be reached with about 100 records. Already, with $c = 2$, a grid directory of 40,000 entries accomodates 9000 records in 6400

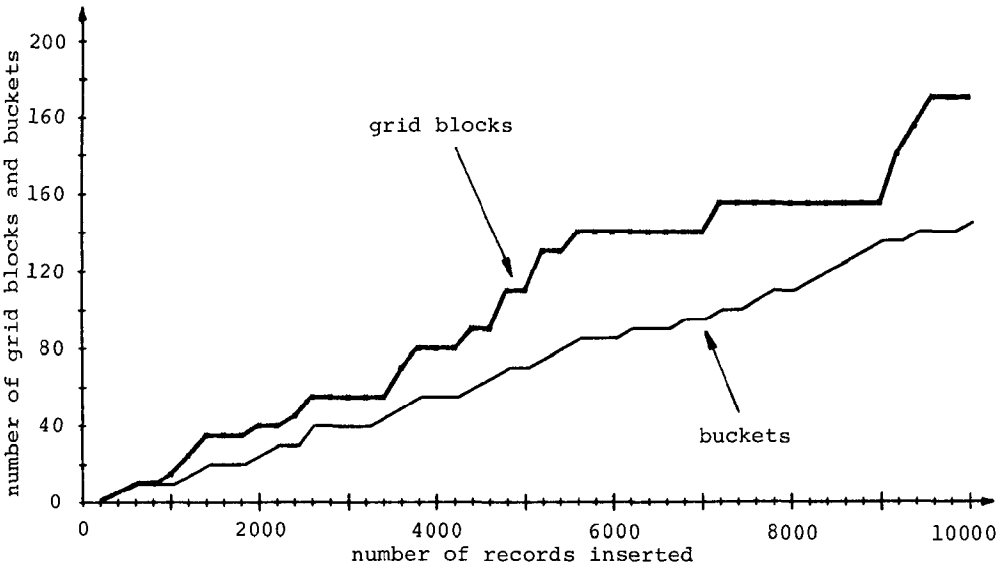


Fig. 13. Number of grid blocks and number of buckets as a function of the stored data.

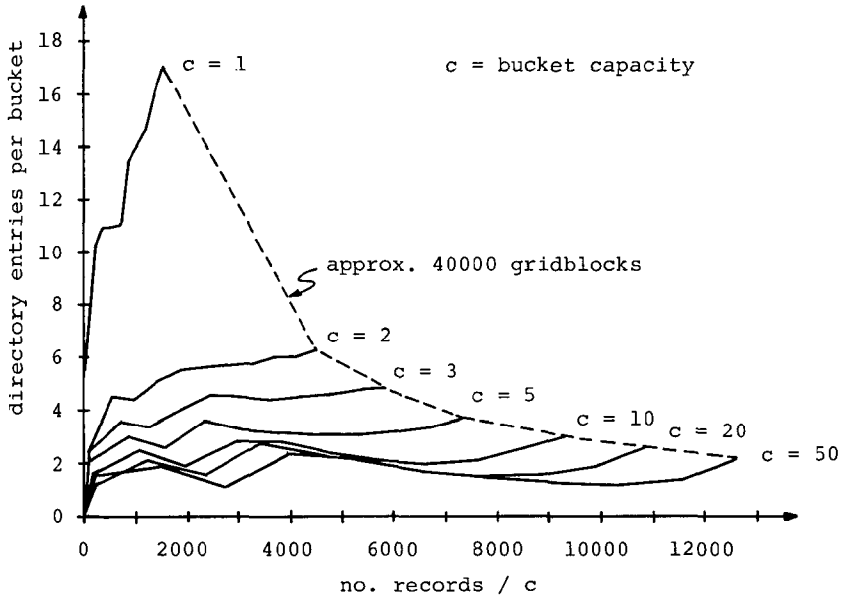


Fig. 14. Grid blocks per bucket as a function of the normalized number n/c of records.

buckets. Such small bucket sizes are only used to demonstrate the effect: we consider 10 or more records per bucket realistic, and for such bucket capacities directory size is no problem.

Visualization of the geometry of bucket regions. Finally, we show how the grid file adapts its shape to the data it must store. Figure 15 shows the bucket regions

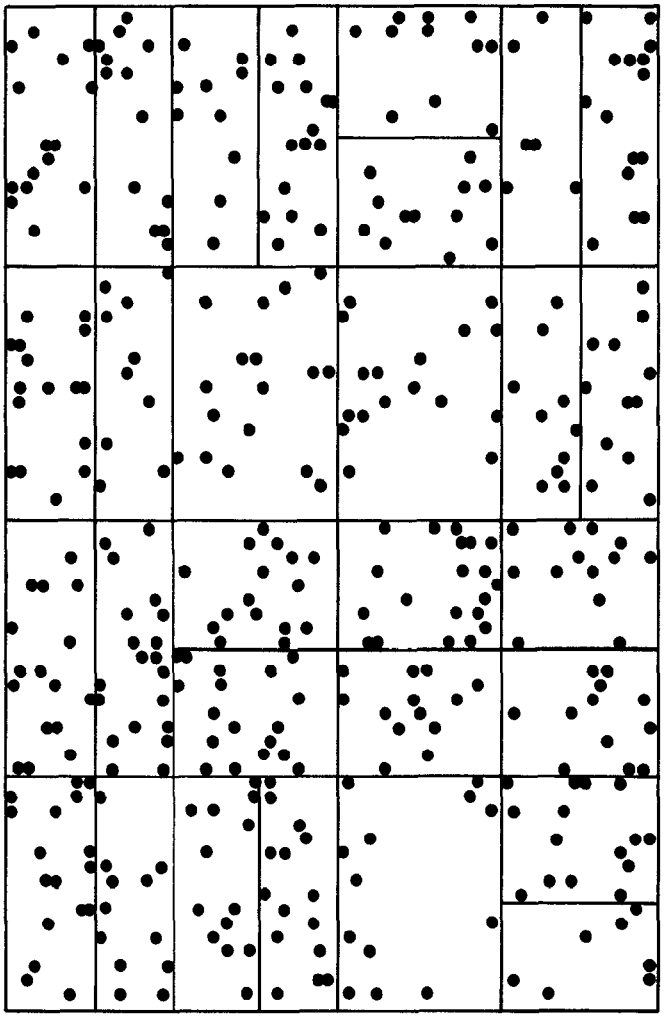


Fig. 15. Bucket regions with uniform record distribution (400 insertions, bucket capacity $c = 20$).

obtained after inserting 400 records from a uniform record distribution and from splitting done at interval midpoints. Figure 16 shows how the grid file "absorbs" a nonuniformity: These bucket regions are obtained from a nonuniform distribution in which the probability is five times greater that a record is drawn from the upper-left quadrant of the space than from the rest.

5.3 Steady-State File

A dynamic file is in a *steady state* if the number of records remains approximately constant, because, in the long run, there are as many insertions as deletions. Whereas a *growing* file is a test for the splitting policy of a file system, a steady-state file tests the interaction between the splitting and merging policies.

In order to determine whether an average bucket occupancy of around 70

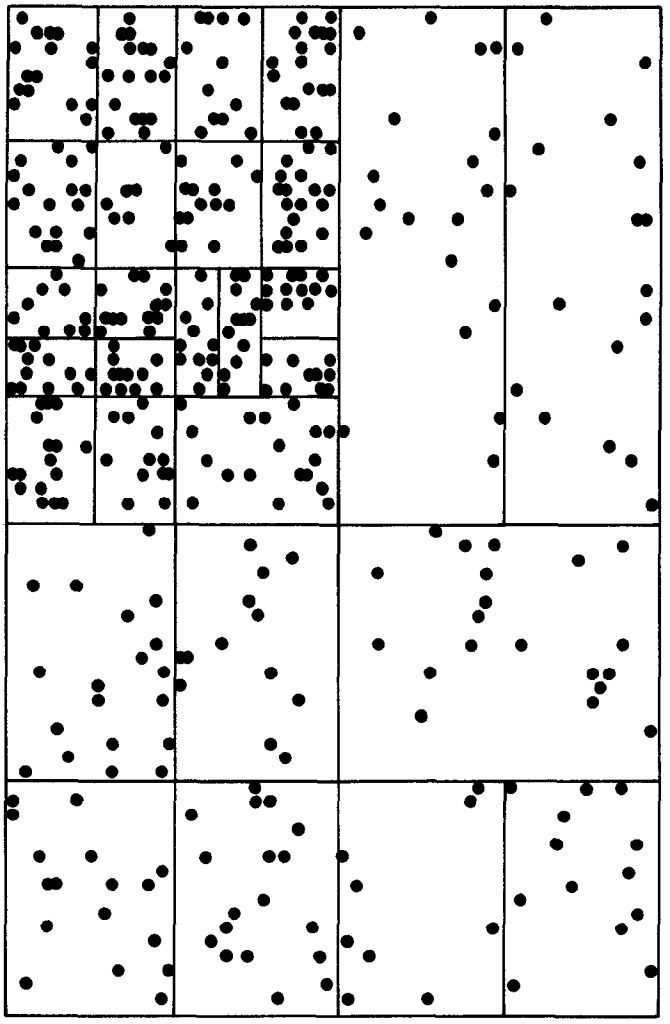


Fig. 16. Bucket regions with nonuniform distribution (400 insertions, bucket capacity $c = 20$).

percent can be maintained in the presence of deletions, we ran the following simulation. The file is initialized by inserting 5000 records into an empty file with a bucket capacity of 16; then, 5000 accesses are generated from a uniform distribution, about half of them insertions and half of them deletions. Different values of the *merging-threshold* (the percent-occupancy which the resulting bucket should not exceed when two buckets are merged) are tested. The bucket capacity of 16 was chosen just large enough so that the merging-threshold can be varied in small steps. The buddy system is used as a merging policy. Figure 17 shows that the average bucket occupancy is rather insensitive to the value of the merging threshold. Even a merging threshold of 100 percent achieves only an average bucket occupancy of around 70 percent, a threshold of 50 percent suffices to reach about a 60 percent average occupancy. Figure 18 shows the time

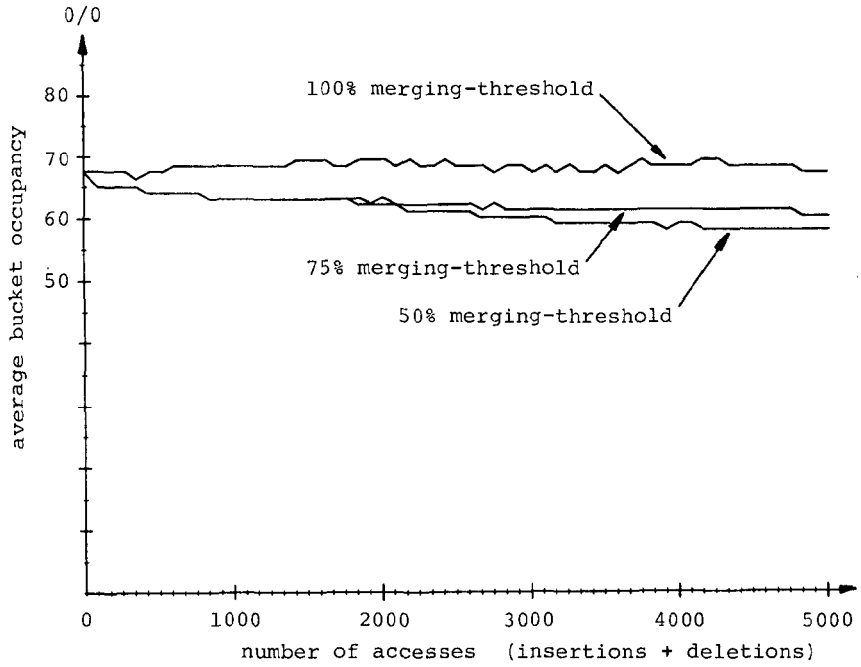


Fig. 17. Average bucket occupancy of grid file in steady-state, buddy system.

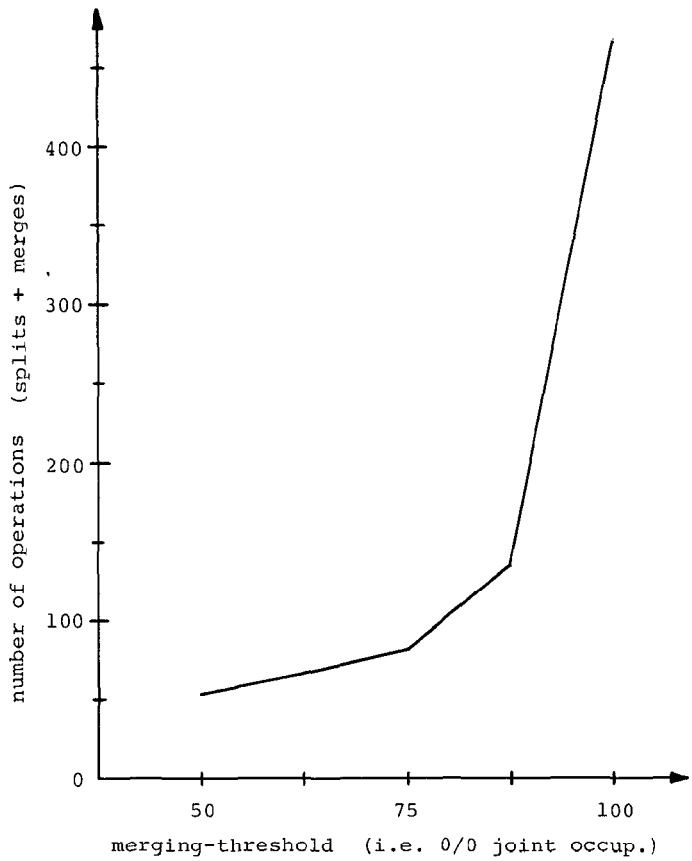


Fig. 18. Number of SPLIT and MERGE operations plotted against merging-threshold with 5000 accesses in steady state.

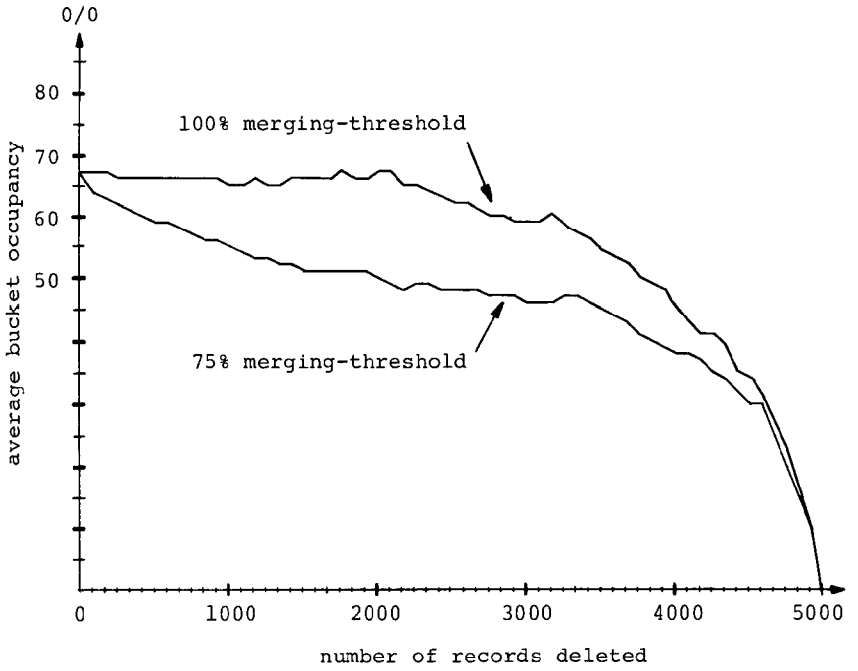


Fig. 19. Average bucket occupancy of a shrinking file, operated under the buddy system with different merging thresholds.

penalty that a high merging threshold entails: if the bucket resulting from a merge is too full, it will soon split again. We recommend a merging threshold of around 70 percent, that is, the average occupancy observed in the growing file.

5.4 The Shrinking File

In order to compare the effectiveness of the merging policies based on the buddy system and the neighbor system, we ran a simulation of a file that shrinks from an initial content of 5000 records down to empty. Figure 19 shows that the buddy system does not guarantee a high average bucket occupancy over a long stretch of deletions. Setting the merging threshold to 100 percent, which may be reasonable if we know the file is in a shrinking phase, helps considerably in the early part. Notice that the merging threshold can easily be adjusted dynamically. In contrast to the buddy system, the neighbor system suffers no degradation in average bucket occupancy, as Figure 20 shows.

In conclusion, we believe that the experiments reported above show that the space utilization of the grid file is good. This is true for a file filling up in its early stages, as well as for a file operating in a steady state or going through brief shrinking phases.

6. REVIEW OF PRIOR MULTIKEY ACCESS TECHNIQUES

In recent years, the increasing usage of databases and integrated information systems has encouraged the development of file structures specifically suited to

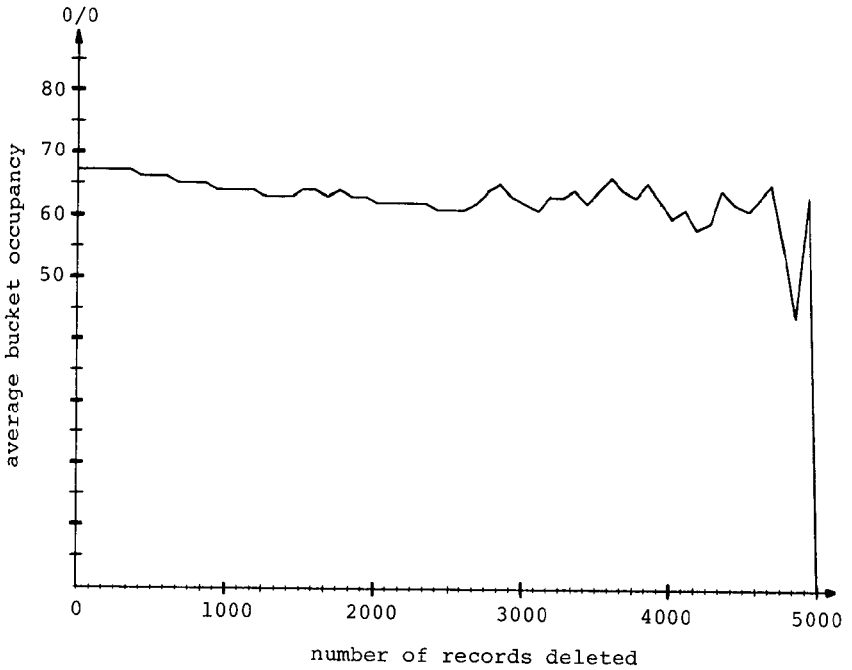


Fig. 20. Average bucket occupancy of a shrinking file, operated under the neighbor system with a merging threshold of 75 percent.

access by combinations of attribute values. Inverted files were among the earliest such file structures. They have been used pervasively in most applications that require multikey access, and thus have been accepted as a standard against which to evaluate alternative approaches.

Several criteria are of importance in assessing multikey file structures. These include operation speed, space utilization, and adaptability under file growth, among others. The specific context in which the file structure is to be used determines the relative importance of various criteria.

The retrieval time in which to obtain all records that satisfy constraints on the values of a combination of attributes depends on several factors. In an inverted file, for example, the appropriate inverted lists must be accessed and processed in order to locate all relevant records, then the records themselves must be retrieved. In most large information systems, the time to move blocks of data from and to secondary storage (typically disks) dominates the processing time in main memory. Hence the number of required block transfers from secondary storage is frequently used as the measure of efficiency in both retrieval and update operations. For this reason, it is important that the information required to perform any operation be as localized as possible within blocks on secondary storage.

A second performance criterion is the space requirement; it must be discussed separately for data storage and access mechanisms. Some file organizations avoid filling each block of storage in order to permit graceful file growth; the size of

such "holes" affects space requirements. Access mechanisms may require significant amounts of storage on disk as well as in main memory. In an inverted file, for example, inverted lists are often so large that they must be stored on disk, but indices to locate the inverted list for each attribute/value pair are retained in main memory.

Inverted files are well suited for accessing records on the basis of Boolean conditions on attributes, but they exhibit drawbacks which have motivated the development of alternate structures. First, retrieval of the inverted lists may require an excessive number of disk accesses. Second, the overhead required for insertions and deletions can become prohibitive in terms of space and time. Finally, in environments where several attributes are equally significant, a file structure that treats all significant attributes symmetrically is appealing.

In the remainder of this section we briefly describe a variety of multikey file structures, each designed to perform better than an inverted file (or other alternatives) in at least some circumstances. Many of the approaches are generalizations of well-known single key file structures. For example, Rothnie and Lozano [29] describe a generalization of hashing in which a bucket address for a record is formed by concatenating the results of hash functions, each of which is applied to the value of one attribute. A critical design decision in setting up such a *multikey hash file* structure is the determination of the number of bits to be allocated to represent the hashed value of each attribute. The more attribute values specified, the smaller the number of buckets that need to be accessed in order to obtain the required records. Because it is difficult to specify a combination of hash functions that lead to a uniform occupancy of buckets, it is necessary to tolerate either a low average bucket occupancy, or a high likelihood that buckets will overflow (more than one storage block is needed to hold the records corresponding to a single bucket). Also, like most hashing schemes, multikey hashing is inappropriate when the selection condition involves ranges of values rather than specific values.

Several generalizations of inverted files have been proposed. Lum describes *combined indices*, in which several attributes are concatenated in various orders and then treated as a single, aggregate key [18]. If more than three attributes are combined, both the storage space and update time become excessive. By combining them in groups of three, however, the number of disk accesses to retrieve inverted lists can be reduced substantially, at the cost of some increased complexity [22]. Bit-encoded inverted lists form the basis of *compressed bitmaps*, described by Vallarino [32]. The bit-encoded inverted lists form a large sparse bit array, which is then represented in highly compressed form and used to locate records specified by a selection condition. Another organization that exploits compression in providing multikey access is the *transposed file* organization, used in ROBOT (Retrieval Organization Based On Transposition) [1, 19]. In this organization, vectors consisting of the values of a particular attribute for all records are stored in a highly compressed form. Thus, retrievals and updates that refer to only a few attributes do not involve memory transfers of irrelevant attributes. This approach is most effective when the majority of operations deal with a significant portion of the records (i.e., one to three percent) and selection conditions involve only a few attributes.

Various generalizations of tree structured indices permit multikey access to files. *Quad trees* [7] are a two-attribute generalization of binary search trees. The straightforward generalization to k dimensions is impractical because the tree nodes become large and contain many nil pointers. These problems are avoided in *k-d trees* [2, 3], which can be thought of as an efficient implementation of the k-dimensional generalization of quad trees. *k-d* trees share many properties with binary search trees.

Similarly, *binary TRIEs* can be generalized to support multikey access [25, 31]. This is achieved by representing each attribute value as a bit string and interleaving these strings. The result is then used as the key in a standard binary TRIE. This organization is particularly effective for handling nearest-neighbor searches [31].

The multiple-attribute tree database organization orders the records lexicographically on the key fields, with the more significant attributes placed toward the higher end of the sorting field [13]. Then the key fields are separated from the records and organized into a doubly-chained tree. The tree can then be used to locate all relevant records for a given query. If both the number of records and the number of attributes are large, several disk accesses may be required to locate records satisfying specified constraints on key values.

Casey describes a complex tree-based multikey access structure in which records are grouped according to the frequency with which they are retrieved together [5]. *Superimposed coding* is used in each node to characterize the records below the node in the tree. Probably because of its complexity, this organization has not been widely used in practice. The importance of this structure is due to the fact that, more than with any other multikey file structure, the selection conditions used in accessing the file influence its organization. A similar, but more practical, approach is suggested by Pfaltz, Berman, and Caglet [26].

Several generalizations of B-trees which would allow multikey access have been proposed recently. For example, Robinson [28] describes *k-d-B-trees*. The leaf nodes of the tree are *pointer pages* that contain pointers to those records which correspond to a "region" (or hyper-rectangle) in k-dimensional space. The internal nodes are *region pages* that reflect the partitioning of a region into nonoverlapping, jointly exhaustive subregions. The root of the tree represents the initial partitioning of the entire k-dimensional space. Efficient utilization of I/O channels is obtained by requiring pointer and region pages to be approximately the size of blocks of secondary storage. Related approaches are taken in [9] and [30].

Quintary trees are a file structure intended to provide faster access than other tree-based multikey file structures, at the cost of requiring more space [15]. Quintary trees consist of k levels, corresponding to the k attributes in decreasing order of importance. Each level resembles a binary tree branching on the values of the corresponding attribute.

Along with *k-d-B*-trees, other multikey file organizations have been proposed recently that are also based on the idea of partitioning k-dimensional space and then storing the records corresponding to each cell of the partition in a single block of secondary storage. One such organization is the *multidimensional directory* suggested by Liou and Yao [17]. Attributes are ordered by priority, and

numbers $d_1, d_2, \ldots, d_k$ are chosen such that $B = d_1 \times d_2 \times \cdots \times d_k$ equals approximately the number of blocks of secondary storage required to hold the blocks of the file. The larger d values are associated with the attributes that appear more frequently in selection conditions. Then, each attribute is used in turn to divide each region at one level into d subregions of approximately equal record population. This results in B regions, each containing approximately one block's worth of records. A multidimensional directory, which contains one entry per secondary storage block, is used to locate those blocks that may contain the records that are relevant to a given selection condition.

Multipaging [20] is another organization that uses splitting factors d. In contrast to [17], all attributes are treated alike. The range of values of attribute i is partitioned into d intervals such that approximately the same number of records have values of attribute i in each interval. These partitions impose a grid of hyper-rectangles in k-dimensional space. Unfortunately, correlations among the attributes and statistical variations cause the occupancies of the hyper-rectangles to be quite uneven. When each grid partition corresponds to a single block on secondary storage, either average occupancy in each block is very low or many blocks overflow. Given a k-tuple of attribute values, the corresponding interval in each of the k dimensions can be determined, and a block address for the record can be calculated without using an index.

Dynamic multipaging [21] is an extension designed to overcome the difficulty of handling insertions and deletions in the original multipaging method. Whenever block overflows cause the average number of block accesses per query to exceed some threshold, the partition on one of the attributes is refined by splitting one of the intervals. If attribute i is split, then the fraction $1/d_i$ of the blocks of the file are split, thus increasing the number of blocks by the factor $(d_i + 1)/d_i$. Such reorganizations require substantial effort. Recently, Burkhard presented a multikey access scheme called *interpolation-based index maintenance* [4] which uses a grid partition of the search space, at intervals determined by a radix, similarly to the grid file. This is a multidimensional generalization of Litwin's linear hashing [16], and relates to the grid file as linear hashing relates to extendible hashing [6]—the correspondence between regions (grid blocks) in space and data buckets is given by formulas ("interpolations"), rather than through a directory. The trade-offs involved in the decision of using a directory, as in the grid file, or avoiding it, as in interpolation-based index maintenance, are an interesting topic for research.

7. CONCLUSIONS

Each of the multikey file structures in use today has its strengths and its weaknesses, and also environments for which it is well suited. Nonetheless, for a significant class of environments, there is a need for a file structure that provides a different balance among the performance criteria. The grid file is designed to handle efficiently a collection of records with a modest number (say $k < 10$) of search attributes whose domains are large and linearly ordered. Within this usage environment, it combines several of the better properties of the file structures reviewed above: A high data storage utilization of 70 percent, combined with

insensitivity to record clusters; smooth adaptation to the contents to be stored, in particular to file growth; a directory which, up to large but realistic file sizes, is compact by the standards of multikey files; fast access to individual records (two disk accesses); and efficient processing of range queries.

We have presented in detail the reasoning that led to the design of the grid file. In summary:

- range queries demand *grid partitions of the search space*,
- efficient update after modification of a grid partition demands *convex assignments of grid blocks to data buckets*,
- the *two-disk-access principle* demands representation of an assignment by means of the *grid directory*.

We have fixed those decisions that appear to us to be essential, and left others open in order to give the implementor freedom to adapt the file system to his environment. In particular, we have treated the following three aspects of the grid file as parameters to be specified by the implementor:

- splitting policy,
- merging policy,
- implementation of the grid directory.

Simulation results show that the grid file uses space economically over a wide range of operating conditions. Although dynamic space partitioning periodically leads to a rapid increase in the number of grid blocks, the allocation of buckets to grid blocks absorbs these bursts: The number of buckets grows in proportion to the number of records. For independent attributes, the number of directory entries per bucket also appears to grow linearly up to large practical file sizes, although asymptotically it may grow faster. Attribute correlations affect the size of the directory, but do not significantly affect the average bucket occupancy.

ACKNOWLEDGMENT

We are grateful to W. Willinger for writing an early version of the simulation program, and to the following people for communicating to us their experiences about ongoing implementations of the grid file: K. Hinrichs of ETH Zurich (grid file for storing geometric objects); S. Banerjee, S. M. Joshi, S. Sanyal, and S. Srikumar of the Tata Institute for Fundamental Research in Bombay (relational database systems based on the grid file); H. Hickhoff and H. P. Kriegel of the University of Dortmund (performance comparison of grid file and various types of multidimensional trees). We also thank A. B. Cremers, A. Frank, Th. Haerder, J. O. Jesperson, O. V. Johansen, J. Koch, M. Mall, H. Samet, J. W. Schmidt, M. Tamminen, and the referees for helpful comments that have improved this paper. This paper supersedes [24], which describes some early results.

REFERENCES

1. BATORY, D.S. On searching transposed files. *ACM Trans. Database Syst. 4*, 4 (Dec. 1979), 531–544.

2. BENTLEY, J.L. Multidimensional search trees used for associative searching. *Commun. ACM 18*, 9 (Sept. 1975), 509–517.
3. BENTLEY, J.L. Multidimensional binary search trees in database applications. *IEEE Trans. Softw. Eng. SE-5*, 4 (July 1979), 333–340.
4. BURKHARD, W.A. Interpolation-based index maintenance. In *Proc. ACM Symp. Principles of Database Systems* (1983), 76–89.
5. CASEY, R.G. Design of tree structures for efficient querying. *Commun. ACM 16*, 9 (Sept. 1973), 549–556.
6. FAGIN, R., NIEVERGELT, J., PIPPENGER, N., AND STRONG, H.R. Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst. 4*, 3 (Sept. 1979), 315–344.
7. FINKEL, R.A., AND BENTLEY, J.L. Quad trees—a data structure for retrieval on composite keys. *Acta Inf. 4* (1974), 1–9.
8. GONNET, G.H., AND LARSON, P.A. External hashing with limited internal storage. In *Proc. ACM Symp. Principles of Database Systems* (1982), 256–261.
9. GUETING, H., AND KRIEGEL, H.P. Multidimensional B-tree: an efficient dynamic file structure for exact match queries. Forschungsbericht Nr. 105, Informatik, Univ. Dortmund, Dortmund, West Germany, 1980.
10. HINRICHS, K., AND NIEVERGELT, J. The grid file: a data structure designed to support proximity queries on spatial objects. In *Proc. Workshop on Graph Theoretic Concepts in Computer Science* (Osnabruck, 1983).
11. HINTERBERGER, H., AND NIEVERGELT, J. Concurrency control in two-level file structures. Working paper, Informatik, ETH Zurich, 1983.
12. JOSHI, S.M., SANYAL, S., BANERJEE, S., AND SRIKUMAR, S. Using grid files for a relational database management system. Speech and Digital Systems Group, Tata Institute of Fundamental Research, Homi Bhabha Road, Bombay 400 005, India.
13. KASHYAP, R.L., SUBAS, S.K.C., AND YAO, S.B. Analysis of the multiattribute tree database organization. *IEEE Trans. Softw. Eng. 2*, 6 (Nov. 1977).
14. KNUTH, D.E. *The Art of Computer Programming. Vol. 3, Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
15. LEE, D.T., AND WONG, C.K. Quintary trees: a file structure for multidimensional database systems. ACM *Trans. Database Syst. 5*, 3 (Sept. 1980), 339–353.
16. LITWIN, W. Linear hashing: a new tool for file and table addressing. In *Proc. 6th International Conference on Very Large Data Bases*, 1980, pp. 212–223.
17. LIOU, J.H., AND YAO, S.B. Multidimensional clustering for database organizations. *Inf. Syst. 2* (1977), 187–198.
18. LUM, V.Y. Multiattribute retrieval with combined indices. *Commun. ACM 13*, 11 (Nov. 1970), 660–665.
19. BARNES, M.C., COLLENS, D.S. Storing hierarchic database structures in transposed form. Datafair, 1973.
20. MERRETT, T.H. Multidimensional paging for efficient database querying. In *Proc. ICMOD* (Milano, Italy, June 1978), pp. 277–289.
21. MERRETT, T.H., AND OTOO, E.J. Dynamic multipaging: a storage structure for large shared data banks. Rep. SOCS-81-26, McGill Univ., 1981.
22. MULLIN, J.K. Retrieval-update speed trade-offs using combined indices. *Commun ACM 14*, 12 (Dec. 1971), 775–778.
23. NIEVERGELT, J. Trees as data and file structures. In *CAAP '81, Proc. 6th Colloquium on Trees in Algebra and Programming*, E. Astesiano and C. Bohm, Eds., Lecture Notes in Computer Science 112, Springer Verlag, 1981, pp. 35–45.
24. NIEVERGELT, J., HINTERBERGER, H., AND SEVCIK, K.C. The grid file: an adaptable, symmetric multikey file structure. In *Trends in Information Processing Systems, Proc. 3rd ECI Conference*, A. Duijvestijn and P. Lockemann, Eds., Lecture Notes in Computer Science 123, Springer Verlag, 1981, pp. 236–251.
25. ORENSTEIN J.A. Multidimensional TRIEs used for associative searching. *Inf. Process. Lett. 14*, 4 (June 1982), 150–157.
26. PFALTZ, J.L., BERMAN, W.J., AND CAGLEY, E.M. Partial-match retrieval using indexed descriptor files. *Commun. ACM 23*, 9 (Sept. 1980), 522–528.
27. RIVEST, R.L. Partial-match retrieval algorithms. *SIAM J. Comput. 5*, 1 (1976), 19–50.

28. ROBINSON, J.T. The k-d-B-tree: a search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD Conference 1981*, ACM, New York, pp. 10–18.
29. ROTHNIE, J.B., AND LOZANO, T. Attribute-based file organisation in a paged environment. *Commun. ACM 17*, 2 (Feb. 1974), 63–69.
30. SCHEUERMANN, P., AND OUKSEL, M. Multidimensional B-trees for associative searching in database systems, *Inf. Syst. 7*, 2 (1982), 123–137.
31. TAMMINEN, M. The extendible cell method for closest point problems. *BIT 22* (1982), 27–41.
32. VALLARINO, O. On the use of bit maps for multiple key retrieval. *ACM SIGPLAN Notices 11*, (Mar. 1976), 108–114.

Received 1982; revised July 1982; accepted February 1983