The Grid file

an adaptable, symmetric multi-key file structure

ETH

Eidgenössische Technische Hochschule
Zürich

Institut für Informatik

J. Nievergelt, H. Hinterberger, K.C. Sevcik

THE GRID FILE:
an adaptable, symmetric multi-key file structure

December 1981 46

THE GRID FILE: an adaptable, symmetric multi-key file structure

J. Nievergelt, H. Hinterberger, K.C. Sevcik

Abstract

Traditional file structures that provide multi-key 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 multi-key access to highly dynamic files. We study the dynamic aspects of file structures that treat all keys symmetrically, that is, avoid the distinction between primary key 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 on the average for single record retrieval. We discuss in detail the design decisions that led to the grid file, present simulation results of its behavior, and compare it to several other multi-key access file structures.

Key words and phrases:

File structures, data base, dynamic storage allocation, multi-key searching, multidimensional data.

Address of authors:

J. Nievergelt & H. Hinterberger:
Institut für Informatik
ETH
CH-8092 Zürich
Switzerland

K.C. Sevcik:
Computer Systems Research Group
University of Toronto
Toronto, Ont. M5S 1A4
Canada

Table of contents

Index of technical terms and notation

(with page numbers where the term is explained)

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, hash files. They allow execution of common file operations, such as FIND, INSERT, 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 leads to a 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 multi-key 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 multi-dimensional 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 multi-dimensional data do not exist.

In view of the diversity of file structures for single-key access, one might expect an even greater variety for multi-key access. In addition to the traditional inverted file, many other schemes have been proposed. [Ben 75], [Ben 79], [Cas 73], [GuKr 80], [Lum 70], [McBC 73], [Riv 76], [RoLo 74], [SchO 80], [Val 76], present 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 multi-key file structures.

Consider a file F as a collection of records $R = [a_1, a_2, .., a_k]$, where the a_i 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, such as [Doe, John, -, 1951, 123456789]. Multi-key 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)

Multi-key access problems come in two kinds. In information and document retrieval an object (say a book) is characterized by many attributes (e.g. alchemy, biology, chemistry,..), but the domain of each attribute is small (it contains perhaps only the two values "relevant" and "irrelevant"). We do not consider this case. We only discuss the other typical case of multi-key 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: $l_i <= a_i <= u_i$, where l_i and u_i denote lower and upper bounds on attribute value a_i 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 $(r_1, r_2, .., r_k)$, where r_i is a value-range of the i-th 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, [Tam 81] applies the related technique of "extendible cells" to closest point problems, and [JSBS 81] study the problem of executing relational data base queries on the grid file.

We approach the problem of designing a practical multi-key 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. Fig. 1.1 shows a 3-dimensional bitmap.

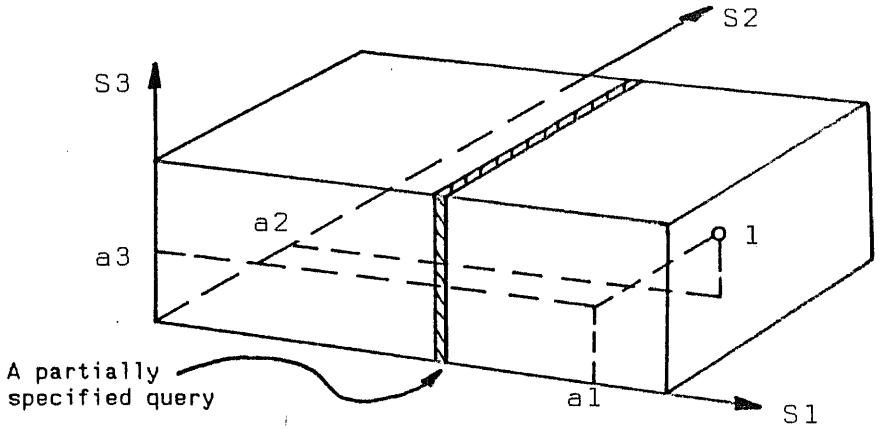


Fig. 1.1 The 3-dimensional bitmap. A "1" indicates the presence of a record with attribute values determined by its position in the map, a "0" indicates absence.

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 zeroes), 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 non-uniform way over the data space.

Efficiency of a file processing system is measured mainly by response times to multi-key access requests. The major component

of this response time is the time spent in accessing peripheral storage media. In today's systems these 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 disk access principle":

> A fully specified query must retrieve a single record in at most two disk accesses: the first one to the correct portion of the directory, the second to the correct data bucket.

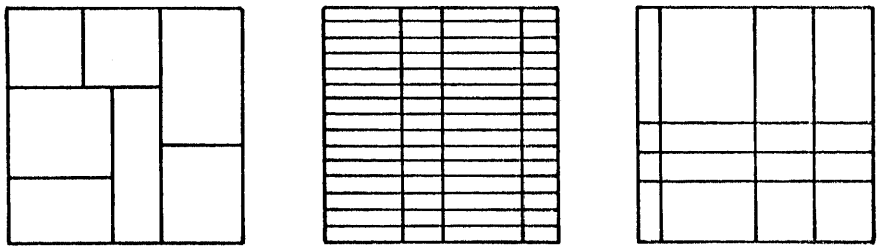
This idea is as old as disk storage devices: the hope to realize it led to the traditional index sequential access methods introduced in the late fifties. It turned out, however, that index sequential access techniques cause more than two disk accesses on average. This is true for traditional techniques 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 [FNPS 79]. It was never applied to dynamic multi-key files, where each secondary key directory of an inverted file typically introduces an additional disk access.

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: 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 [Nie 81] 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 this partition is implemented. Let us consider three examples.

Multi-dimensional trees of various kinds (e. g. [Ben 75]) are an example of multi-key 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 2.1a, 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 2.1b shows the resulting partition. The grid file presented in this paper is based on grid partitions of the search space illustrated in figure 2.1c. 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.



a) b) c)

Fig. 2.1 Space partitions created by different search techniques.

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 kinds of partitions shown in figure 2.1, such as:

- systematic region boundaries;
- economy of representation: one boundary line of figure 2.1c does the work of many in figure 2.1a.

We introduce the following terminology and notation for the 3-dimensional case; generalization to k dimensions is obvious. On the record space $S = X \times Y \times Z$ we obtain a grid partition

P = Q x R x S by imposing intervals on each axis and dividing the record space into blocks which we call grid blocks, as shown in figure 2.2.

Record space S = X x Y x Z

Grid partition P = Q x R x S

Intervals of the partition $Q = (q_1, q_2, \ldots, q_k)$
$R = (r_1, r_2, \ldots, r_k)$
$S = (s_1, s_2, \ldots, s_k)$

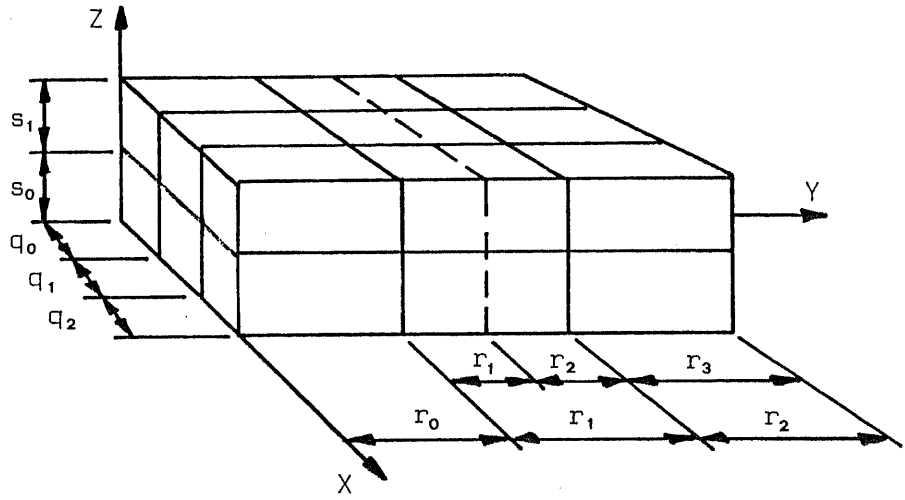


Fig. 2.2 A 3-dimensional record space X x Y x Z, with a grid partition P = Q x R x S. The picture shows the effect of refining P by splitting interval r_1.

During 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 = Q x R x S is modified by altering only one of its components at a time. A 1-dimensional partition is modified either by splitting one of its intervals in two, or by

merging two adjacent intervals into one. Fig. 2.2 shows this for the partition r_1. Notice that the intervals "below" the one being split or the two being merged retain their index (r_0 in Fig. 2.1), while the indices of the intervals "above" the point of splitting or merging are shifted by +1 or -1, respectively ($r_2 \leftrightarrow r_3$ in Fig. 2.1).

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 1-dimensional partitions. Thus the 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. Any conceivable difference in access time to different buckets is ignored, and the time required for a file operation is 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 between grid blocks and buckets: this is accomplished by the grid directory, to be described in section 3.1. In order to obtain an efficient file structure, constraints on access time, on update time, and on memory utilization must be met. In particular, we aim at:

- the two-disk-access principle
- splitting and merging of grid blocks involve only two buckets
- maintain a reasonable lower bound on average bucket occupancy.

The first two points about processing efficiency are discussed in sections 3.2 and 3.3, respectively. The third point about 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 the 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 define and maintain the dynamic correspondence between grid blocks in the record space and data buckets. Hence we must define the class of possible correspondences before we can design a data structure. For reasons of efficiency, only a small subset of all possible assignments of grid blocks to buckets will be allowed.

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
- 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 must be k-dimensional rectangles. We call such an assignment of grid blocks to buckets a box-shaped assignment. Fig. 3.1 shows a typical box-shaped 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.

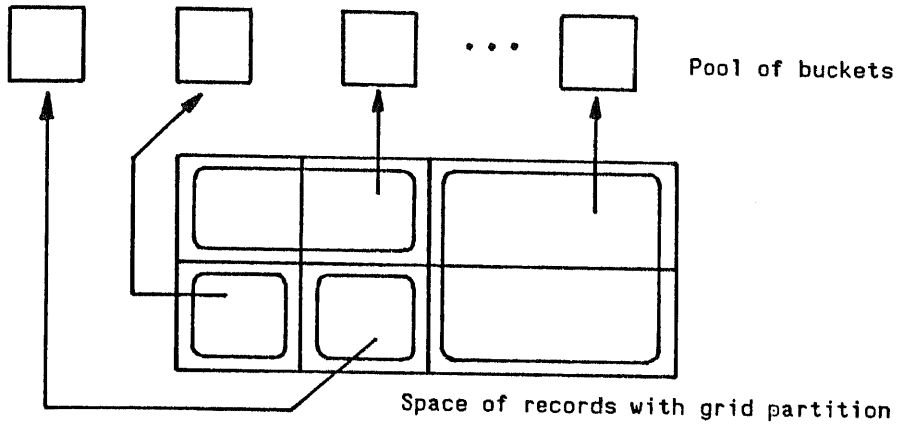


Fig. 3.1 A box-shaped assignment of grid blocks to buckets.

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 on which precisely those operations are defined that arise when the box-shaped assignments defined above must be updated because of bucket overflow or underflow. The difficult choice of just how much to specify about this data structure and how much to leave open as a later implementation decision is discussed in section 3.4. Here 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 1:1 correspondence with the grid blocks of the partition;

second, k 1-dimensional arrays called linear scales; each scale defines a partition of a domain S .

For the sake of notational simplicity we present the case $k = 2$, from which the general case $k > 2$ is easily inferred.

A grid directory G for a 2-dimensional space is characterized by:

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

It consists of:

- a 2-dimensional array G(0..nx-1, 0..ny-1) ("grid array")
- 1-dimensional arrays X(0..nx), Y(0..ny) ("linear scales")

1) Operations defined on the grid directory:

- Direct access: G(cx,cy)

- Next in each direction

  nextxabove: cx ← (cx + 1) mod nx
  nextxbelow: cx ← (cx - 1) mod nx
  nextyabove: cy ← (cy + 1) mod ny
  nextybelow: cy ← (cy - 1) mod ny

- Merge

  mergex: given px, 1 <= px < nx, merge px with nextxbelow;
          rename all elements above px; adjust X-scale.
  mergey: similar to mergex for any py, 1 <= py < ny.

- Split

  splitx: given px, 0 <= px <= nx, create new element px + 1 and
          rename all cells above px; adjust X-scale.
  splity: similar to splitx for any py, 0 <= py <= ny.

2) Constraints on the values

The restriction to box-shaped assignments of grid blocks to buckets leads to constraints on the values of grid directory elements. We formulate these recursively in order to mirror the splitting history that occurs in dynamic files as discussed in section 3.3. A legal grid directory must be reduced and meet the value constraints, according to the following definition:

- a k-dimensional 1x1x .. x1 grid directory is reduced;
- a k-dimensional grid directory meets the value constraints if all its elements have the same value in the domain of bucket addresses;
- a k-dimensional grid directory larger than 1x1 .. x1 is reduced and meets the value constraints if and only if there is a (k-1)-dimensional hyperplane that partitions the grid directory into two non-empty parts such that:
  the sets of values of each part are disjoint,
  both parts meet the value constraints,
  at least one part is reduced.

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 .. 2000, and attribute "initial" with domain a .. 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); Y = (a, f, k, r, z).

A FIND for a fully specified query $(r_1, r_2, \ldots,)$, such as FIND [1980, w], is executed as shown in Fig. 3.2. 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

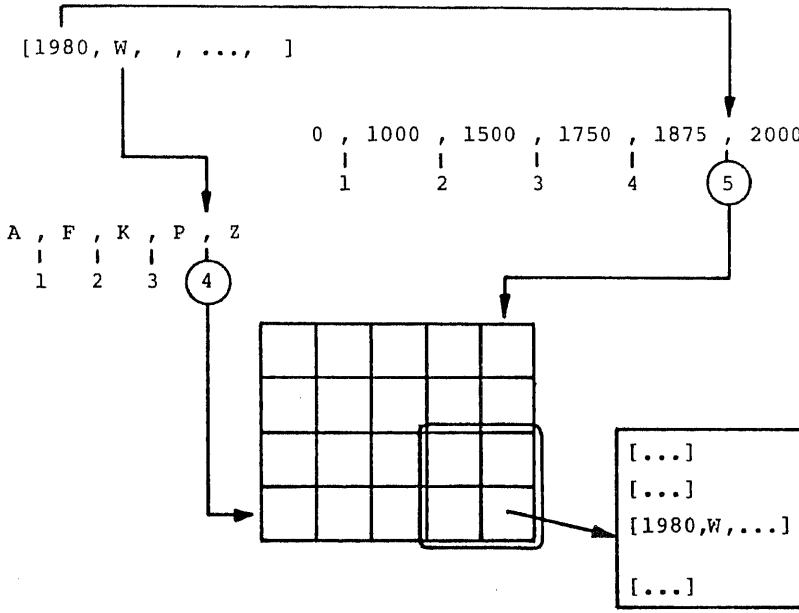


Fig. 3.2 Retrieval of a single record requires two disk accesses.

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

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

Fig. 3.3 illustrates the fact that the precision of most range queries is high. In particular, precision approaches 1 for queries that retrieve many records (compared to bucket capacity).

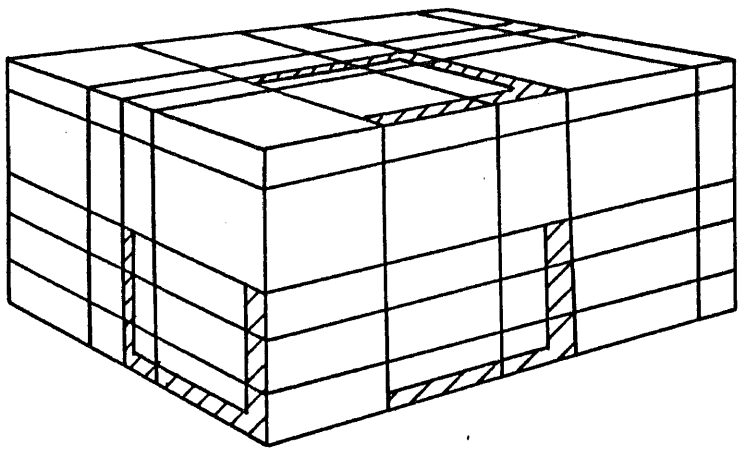
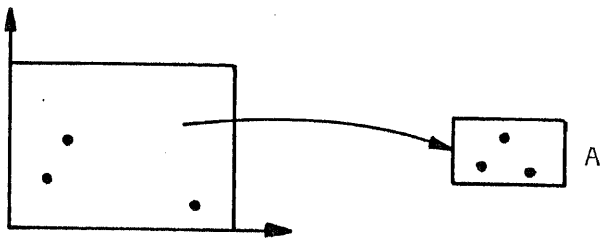


Fig. 3.3 A range query causes irrelevant records to be retrieved only at the fringes of the answer.

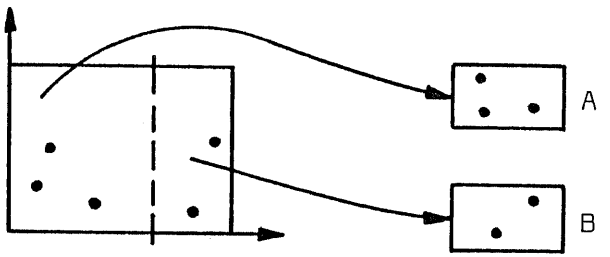
3.3 Dynamics of the grid file

The dynamic behavior of the grid file is best explained by tracing an example: building up a file under repeated insertions. When deletions occur, the inverse operations from those described below 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 1:1 correspondence with the grid blocks, we draw the bucket pointers as originating directly from the grid blocks.

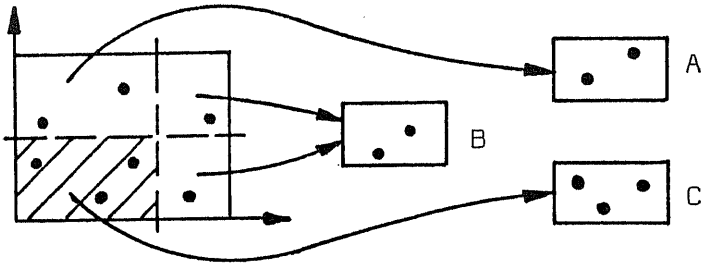
Initially a single bucket A, of capacity c = 3 in our example, is assigned to the entire record space.



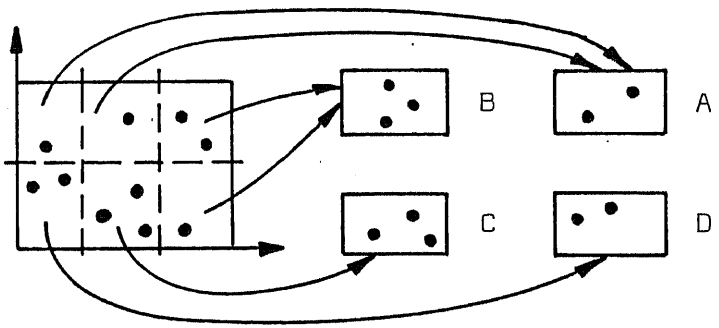
When bucket A overflows, the record space is split, a new bucket B is made available, and those records that lie in one half of the space are moved from the old bucket to the new one.



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.



Whenever there is more than one bucket, the set of buckets currently in use (and hence the set of regions of these buckets) is organized in the form of a (multi-dimensional) "twin system" (or buddy system): each bucket and each region has a unique potential twin with which it can merge again. A pair of twins is created when a bucket and its region are split, and disappears when the two twins merge. In the picture above, C and D are twins, the pair (C,D) is a potential twin of A, and the pair (A, (C, D)) is a potential twin of B. Thus deletions in a cluster of high density where the regions were split to a fine granularity (such as the regions C and D above) may trigger merging: buckets C and D can be merged whenever their joint occupancy permits.

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 data base, 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 multi-key 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 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
- 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 whose region consists of a single grid block. Its occurrance is relatively rare: the majority of all overflows involve buckets whose region consists of 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 policy chooses the dimension according to a fixed schedule, perhaps cyclically. A splitting policy may favor some attribute(s) 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. A favored attribute plays a role similar to the primary key in an inverted file.

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 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 Fig. 4.1). What data structure should be chosen to implement a grid directory?

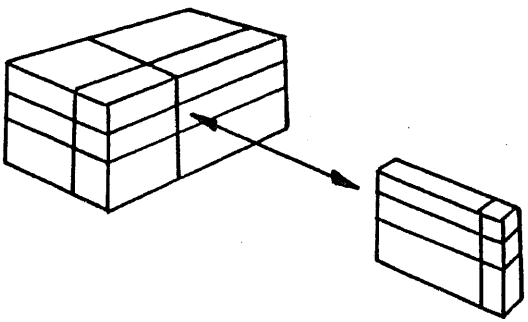


Fig. 4.1 Insertion and removal of an arbitrary (k-1)-dimensional cross section.

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 4.2. This technique is the multi-dimensional counterpart of the directory used in extendible hashing [FNPS 79]. 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 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 non-uniformity in the data leads to an oversized directory, and this approach cannot be recommended in general.

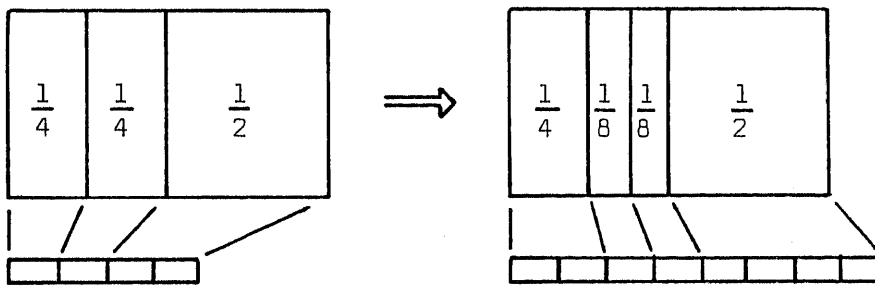


Fig. 4.2 Representation of the grid directory by an array that occasionally doubles in size.

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 at the end of the

k-dimensional old array is described in [Josh 81]. A time stamp 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 view of the fact that in most applications split and merge occur rarely as compared to "direct access" and "next", the conventional allocation of a multidimensional array appears to be best. Split and merge operations cause a shift of a fraction of all grid cells, and hence take longer than they would in a linked list. On the other hand, direct access is faster, and memory utilization is optimal, thus a larger fraction of the directory may be kept in central memory.

4.4 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-structured data 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 B-tree), other processes may be affected (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 [Nie 81]) has the property that access paths to separate buckets are completely disjoint, thus allowing simpler concurrency control protocols.

5. Simulation experiments for determining space utilization

The performance of a file system is determined by two criteria: processing time and memory utilization. We designed the grid file so as to minimize the number of disk accesses, and we now must show that this overriding concern for speed is compatible with a reasonable utilization of available space. This is achieved by means of simulation experiments.

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.

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%, but it must be prevented from becoming arbitrarily small under any circumstances. Point 2) is less important, because 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), and thus the grid directory requires only a fraction of the space required by data storage. 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.

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

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

We have tested the behavior of a simulated, continuously growing grid file with two Pascal programs. One for the two-dimensional case (for ease of displaying results graphically) and one for the three-dimensional case. The justification for restricting our experiments to 2 and 3 dimensions is that the bucket occupancy (the primary objective of our simulation) appears to be totally independent of the dimensionality of the record space. 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.

The attribute values of each record are chosen independently of each other from uniform and piecewise uniform 1-dimensional distributions to obtain uniform and non-uniform data distributions over the record space. Two standard integer-valued random number generators were used to generate these attribute values. The grid directory is implemented as a k-dimensional array. A binary tree that records the splitting history is used to represent the linear scales of the attribute domains. Records are stored sequentially in buckets in the order of arrival.

5.2 Results

1) Average bucket occupancy

We observed the average bucket occupancy while inserting 10'000 records from a two-dimensional uniform distribution. Fig 5.1 shows two typical curves depicting the average bucket occupancy over time, one for bucket capacity $c = 50$, the other for $c = 100$.

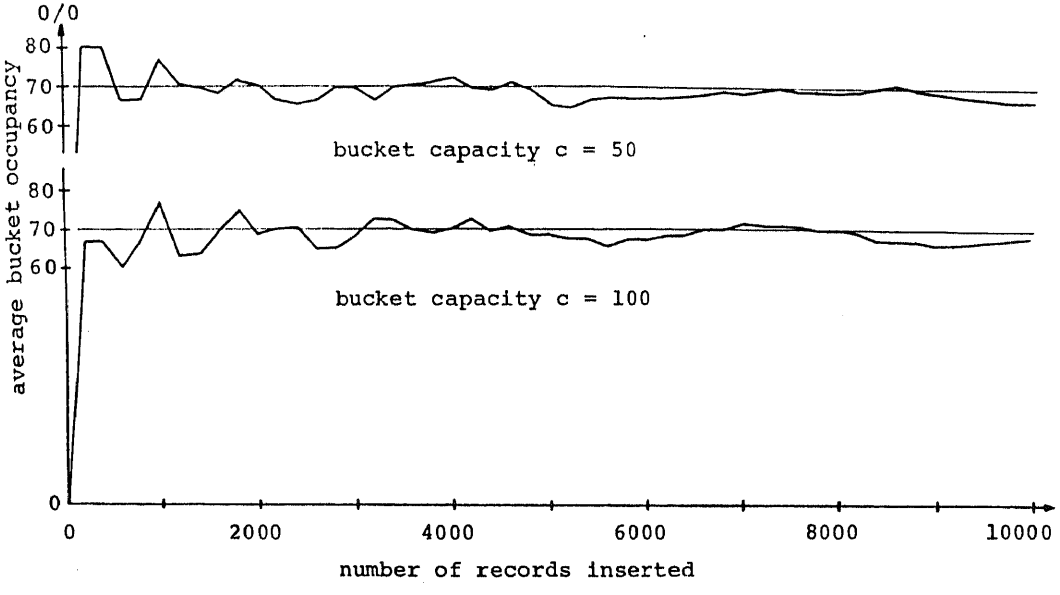


Fig. 5.1 Average bucket occupancy of continuously growing grid file.

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 around 70%. It is tempting to conjecture that it approaches asymptotically the magical value ln 2 = 0.6931.., which often shows up in theoretical analyses of processes that repeatedly split a set into two equiprobable parts (see also [FNPS 79]).

In section 4 we mentioned splitting policies that do not necessarily refine a 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%. 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 a bucket into two.

2) 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 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.

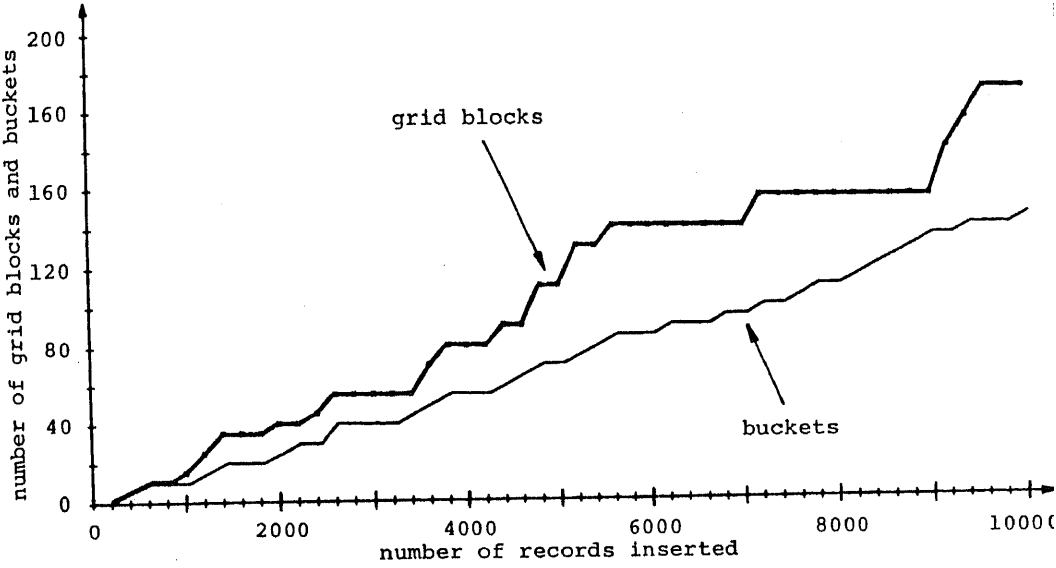


Fig. 5.2 Number of grid blocks and number of buckets as a function of the stored data.

Fig. 5.2 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% as expected. It is remarkable that the curve representing the number of grid blocks also appears to grow linearly, but the fluctuations from a straight line have a larger period and amplitude. The average number of directory entries per bucket oscillates between one and two.

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 will overflow whose regions are randomly chosen from the entire record space; 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 refinements, until we are back to a "one-grid-block-per-bucket" state but with a directory that has doubled in size.

3) Visualization of the geometry of bucket regions

Finally we show how the grid file adapts its shape to the data it must store. Fig. 5.3 shows the bucket regions obtained after inserting 400 records from a uniform record distribution and splitting done at interval midpoints. Fig. 5.4 shows how the grid file "absorbs" a non-uniformity: these bucket regions are obtained from a non-uniform distribution in which the probability is 5 times greater that a record is drawn from the upper left quadrant of the space than from the rest.

In conclusion, we believe that the experiments reported above show conclusively that the space utilization of the grid file is good in the frequently occuring situation of a growing file.

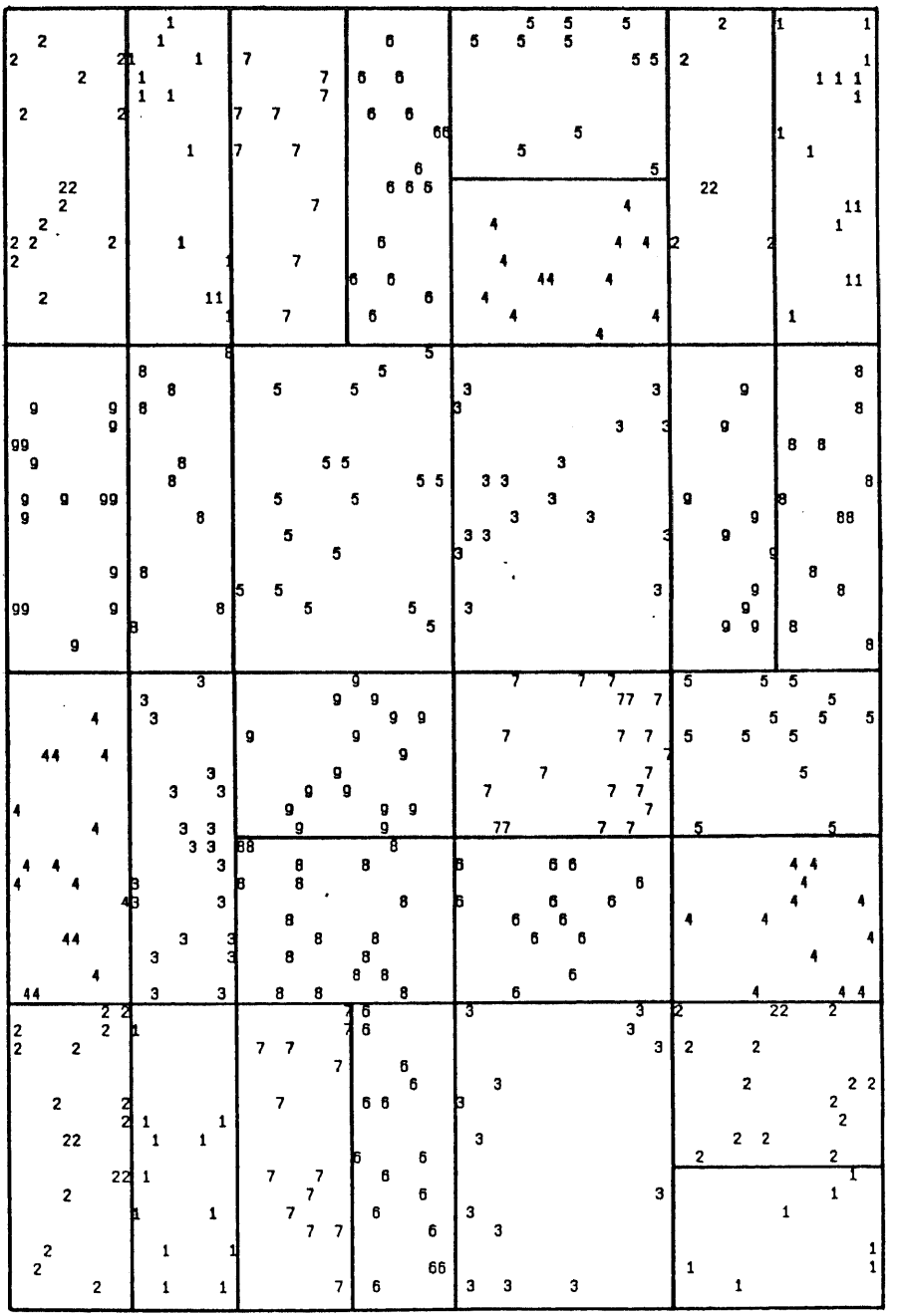


Fig. 5.3 Bucket assignments with uniform record distribution
(400 insertions, bucket capacity c = 20)

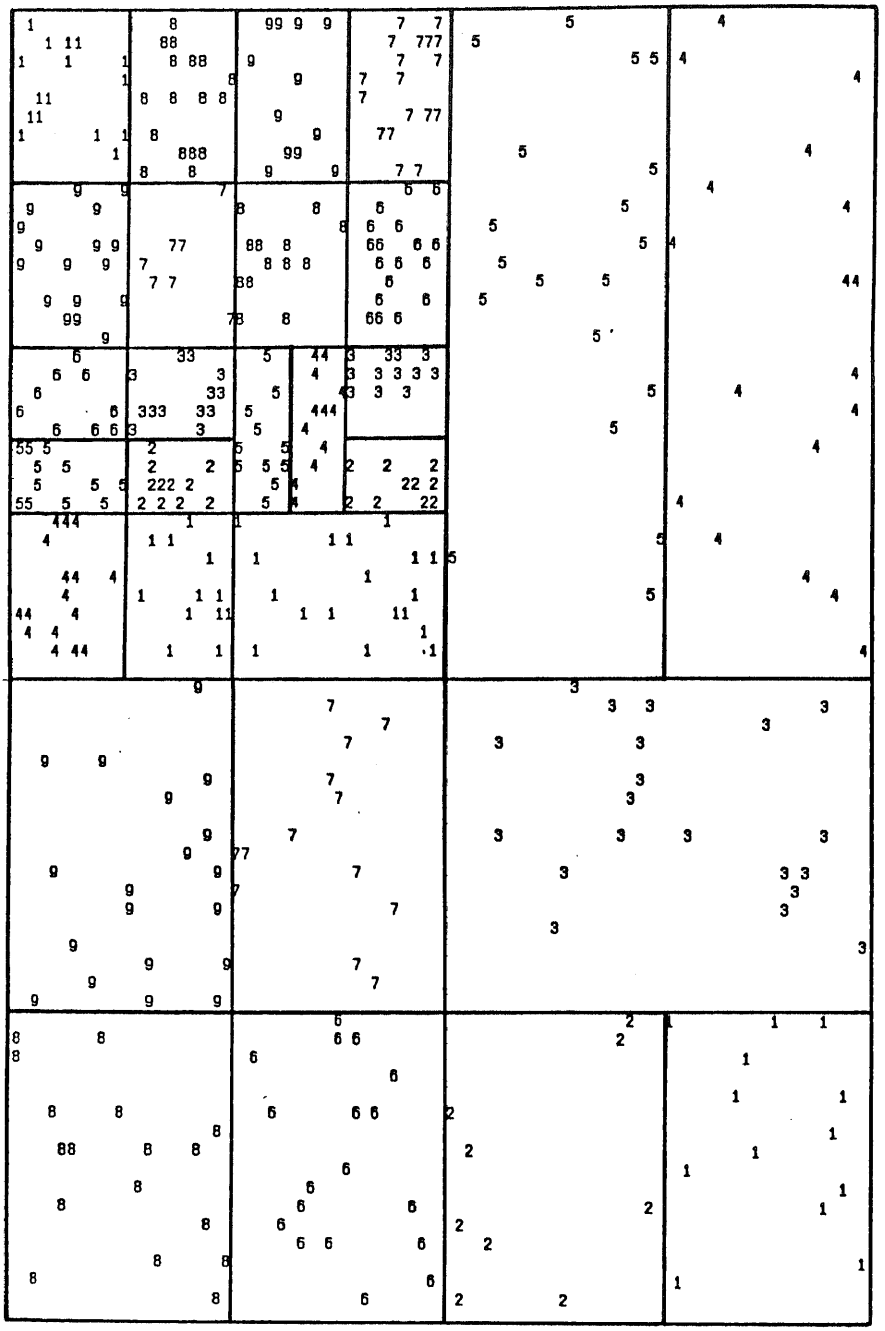


Fig. 5.4 Bucket assignments with non-uniform record distribution (400 insertions, bucket capacity c = 20)

6. Review of prior multi-key access techniques

In recent years, the increasing usage of data bases and integrated information systems has encouraged the development of file structures specifically suited to 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 multi-key access, and thus have been accepted as a standard against which to evaluate alternative approaches.

Several criteria are of importance in assessing multi-key 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 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 that 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 multi-key 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 [RoLo 74] 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 a multi-key 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, multi-key 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 [Lum 70]. 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 [Mull 71]. Bit-encoded inverted lists form the basis of compressed bitmaps, described by Vallarino [Val 76]. 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 multi-key access is the transposed file organization used in ROBOT (Retrieval Organization Based On Transposition) [McCo 72], [Bato 80]. 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. 1%-3%) and selection conditions involve only a few attributes.

Various generalizations of tree structured indices permit multi-key access to files. Quad trees [FiBe 74] 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 [Ben 75], [Ben 79] 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 multi-key access [Oren 81], [Tamm 81]. 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 [Tamm 81].

The multiple-attribute tree data base organization orders the records lexicographically on the key fields with the more significant attributes placed toward the higher end of the sorting field [KaSy 77]. 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 multi-key access structure in which records are grouped according to the frequency with which they are retrieved together [Case 73]. 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 multi-key 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 [PfBC 80].

Several generalizations of B-trees to allow multi-key access have been proposed recently. For example, Robinson [Rob 81] 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 non-overlapping jointly exhaustive sub-regions. 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 [GuKr 80] and [SchOu 80].

Quintary trees are a file structure intended to provide faster access than other tree-based multi-key file structures at the cost of requiring more space [LeVo 80]. 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 multi-key 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 [LiYa 77]. Attributes are ordered by priority, and numbers $d_1, d_2, \ldots, d_k$ are chosen such that $B = d_1 \times d_2 \times \ldots \times d_k$ equals approximately the number of blocks of secondary storage required to hold the blocks of the file. The larger d_i 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_i sub-regions 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 [Merr 78] is another organization that uses splitting factors d_i. In contrast to [LiYa 77], all attributes are treated alike. The range of values of attribute i is partitioned into d_i intervals such that approximately the same number of records have values of attribute i in each interval. These partitions impose a grid of B hyper-rectangles in k-dimensional space. Unfortunately, correlations among the attributes and statistical variations cause the occupancies of the B 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 [MeOt 81] 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.

Each of the multi-key file structures reviewed above has its strengths and its weaknesses, and 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 combines several of the better properties of the file structures

we have discussed: A directory which is compact by the standards of multi-key files (less than two entries per data bucket); a high data storage utilization of 70%, combined with insensitivity to attribute correlations and record clusters; smooth adaptation to file growth.

7. Conclusions

A review of the literature on file systems for multi-key access to records indicates that most of the work done so far has solved only part of the overall problem. Whereas every system surveyed provides certain advantages, they all are resistant to adaptation in a dynamic environment. The grid file proposed in this paper adapts its shape continuously to the contents to be stored, and thus achieves:

- fast access to individual records (two disk accesses)
- efficient processing of range queries
- high memory utilization (70%)
- simple concurrent access due to absence of pointer chains.

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 box-shaped 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 two aspects of the grid file as parameters to be specified by the implementor:

- splitting policy
- implementation of the grid directory.

Simulation results show that the grid file uses space economically. 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, with a high memory utilization of 70%, regardless of data distribution; the number of directory entries per bucket oscillates between one and two for uniform data distributions.

References

[Bato 80]
D. S. Batory: On searching transposed files,
Proc. ACM SIGMOD Conf., 1981.

[Ben 75]
J.L. Bentley: Multi-dimensional Search Trees used for Associative Searching, CACM 18, 9, 1975, 509-17.

[Ben 79]
J.L. Bentley: Multidimensional Binary Search Trees in Database-Applications, IEEE Trans. Softw. Eng., Vol. SE-5, No.4, July 1979, 333-40.

[Cas 73]
R.G. Casey: Design of Tree Structures for Efficient Querying, CACM 16, 9, 1973, 549-56.

[FNPS 79]
R. Fagin, J. Nievergelt, N. Pippenger, H.R. Strong: Extendible Hashing - a Fast Access Method for Dynamic Files, ACM Trans. Database Systems, Vol. 4, No. 3, 1979, 315-44.

[FiBe 74]
R. A. Finkel, J. L. Bentley: Quad trees - a data structure for retrieval on composite keys, Acta Informatica, Vol. 4, 1974, 1-9.

[GuKr 80]
H. Gueting, H.P. Kriegel: Multidimensional B-tree: An efficient dynamic file structure for exact match queries, Forschungsbericht Nr. 105, Informatik, Univ. Dortmund, W.Germany, 1980.

[Josh 81]
S. M. Joshi: A memory allocation scheme for the grid directory, personal communication.

[JSBS 81]
S. M. Joshi, S. Sanyal, S. Banerjee, S. Srikumar: Evaluation of Queries in a Grid File Environment. Speech and Digital Systems Group, Tata Institute of Fundamental Research, Homi Bhabha Road, Bombay 400 005, India, personal communication.

[KaSY 77]
R. L. Kashyap, S. K. C. Subas, S. B. Yao: Analysis of the multi-attribute tree database organization, IEEE Trans. Software Engr., Vol. 2, No. 6, Nov 1977,.

[Knu 73]
D.E. Knuth: The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley Publ. Co., 1973.

[LeWo 80]
D. T. Lee, C. K. Wong: Quintary trees: A file structure for multidimensional databasee systems, ACM Tods Vol. 5, No. 3, Sep 1980, 339-353.

[LiYa 77]
J. H. Liou and S. B. Yao: Multi-dimensional clustering for data base organizations, Information Systems Vol. 2, 1977, 187-198.

[Lum 70]
V.Y. Lum: Multi-Attribute Retrieval with Combined Indices, CACM 13, 11, 1970, 660-65.

[McBC 73]
McBarnes, D.S. Collens: Storing Hierarchic Database Structures in Transposed Form, Datafair 1973.

[Merr 78]
T. H. Merrett: Multidimensional paging for efficient data base querying, Proc. ICMOD, Milano, Jun 1978, 277-289.

[MeOt 81]
T. H. Merrett, E. J. Otoo: Dynamic multipaging: A storage structure for large shared data banks, Report SOCS-81-26, McGill University, 1981.

[Mull 71]
J. K. Mullin: Retrieval-update speed tradeoffs using combined indices, Comm. ACM, Vol. 14, No. 12, Dec 1971, 775-778.

[Nie 81]
J. Nievergelt: Trees as Data and File Structures,
in CAAP '81, Proc. 6-th Coll. on Trees in Algebra and Programming, E. Astesiano and C. Bohm (eds.), Lecture Notes in Computer Science 112, 35-45, Springer Verlag 1981.

[NHS 81]
J. Nievergelt, H. Hinterberger, K. C. Sevcik:
The Grid File: an adaptable, symmetric multikey file structure,
in Trends in Information Processing Systems, Proc. 3rd ECI Conf., A. Duijvestijn and P. Lockemann (eds.), Lecture Notes in Computer Science 123, 236-251, Springer Verlag 1981.

[Oren 81]
J. A. Orenstein: Multidimensional TRIEs used for associative searching, Report SOCS-81-23, McGill Univ., Aug 1981.

[PfBC 80]
J. L. Pfaltz, W. J. Berman, E. M. Cagley: Partial-match retrieval using indexed descriptor files, Comm. ACM, Vol. 23, No. 9, Sep 1980, 522-528.

[Riv 76]
R.L. Rivest: Partial-Match Retrieval Algorithms, SIAM, J. Comp., Vol. 5, No. 1, 1976, 19-50.

[Rob 81]
J. T. Robinson: The k-d-B-tree: A search structure for large multi-dimensonal dynamic indexes, Proc. SIGMOD Conf. 1981, 10-18.

[RoLo 74]
J.B. Rothnie, T. Lozano: Attribute-Based File Organisation in a Paged Environment, CACM 17, 2, 1974, 63-69.

[Sch0 80]
P. Scheuermann, M. Ouksel: Multidimensional B-Trees for Associative Searching in Database Systems, Report No. 80-12-DBM-05, Dept. of El. Eng. and Comp. Sci., Northwestern University, Evanston, Ill. 60201.

[Schn 77]
B. Schneiderman: Reduced combined indices for efficient multiple attribute retrieval, Information Systems, Vol.2, 1977, 149-154.

[Tam 81]
M. Tamminen: The extendible cell method for closest point problems, Report Helsinki University of Technology, Lab. of Information Processing, 1981.

[Val 76]
O. Vallarino: On the use of bit maps for multiple key retrieval, ACM SIGPLAN Notices, Vol. 11, Mar 1976, 108-114.

Acknowledgement

We are grateful to W. Willinger for writing an early version of the simulation program, to M. Tamminen for his discussions of the extendible cell method, and to S. Banerjee, S. M. Joshi, S. Sanyal, S. Srikumar of the Tata Institute for Fundamental Research in Bombay for information about their design of a relational data base system based on the grid file. This paper supersedes [NHS 81], which describes some early results.

Berichte des Instituts für Informatik

*Nr. 1	N. Wirth:	The Programming Language PASCAL
*Nr. 2	N. Wirth:	Program development by step-wise refinement
Nr. 3	P. Läuchli:	Reduktion elektrischer Netzwerke und Gauss' sche Elimination
Nr. 4	W.Gander, A. Mazzario:	Numerische Prozeduren I
*Nr. 5	N. Wirth:	The Programming Language PASCAL (Revised Report)
*Nr. 6	C.A.R. Hoare, N.Wirth:	An Axiomatic Definition of the Language PASCAL
*Nr. 7	W. Gander, A. Mazzario:	Numerische Prozeduren II
Nr. 8	E. Engeler, E. Wiedmer E.Zachos:	Ein Einblick in die Theorie der Berechnungen
*Nr. 9	H.P. Frei:	Computer Aided Instruction: The Author Language and the System THALES
*Nr. 10	K.V.Nori, U.Ammann, K.Jensen, H.H.Nägeli, Ch.Jacobi:	The PASCAL 'P' Compiler: Implementation Notes (Revised Edition)
Nr. 11	G.I. Ugron, F.R.Lüthi:	Das Informations-System ELSBETH
*Nr. 12	N. Wirth:	PASCAL-S: A subset and its Implementation
*Nr. 13	U. Ammann:	Code Generation in a PASCAL Compiler
Nr. 14	K.Lieberherr:	Toward Feasible Solutions of NP-Complete Problems
*Nr. 15	E. Engeler:	Structural Relations between Programs and Problems
Nr. 16	W. Bucher:	A contribution to solving large linear problems

*Nr. 17	N. Wirth:	Programming languages: what to demand and how to access them and Professor Cleverbyte's visit to heaven
*Nr. 18	N. Wirth:	MODULA: A language for modular multiprogramming
*Nr. 19	N. Wirth:	The use of MODULA and Design and Implementation of MODULA
Nr. 20	E. Wiedmer:	Exaktes Rechnen mit reellen Zahlen
*Nr. 21	J.Nievergelt, H.P.Frei, et al.:	XS-0, a Self-explanatory School Computer
Nr. 22	P. Läuchli:	Ein Problem der ganzzahligen Approximation
Nr. 23	K. Bucher:	Automatisches Zeichnen von Diagrammen
Nr. 24	E. Engeler:	Generalized Galois Theory and its Application to Complexity
Nr. 25	U. Ammann:	Error Recovery in Recursive Descent Parsers and Run-time Storage Organization
Nr. 26	E. Zachos:	Kombinatorische Logik und S-Terme
*Nr. 27	N. Wirth:	MODULA-2
*Nr. 28	J.Nievergelt, J. Weydert:	Sites, Modes and Trails: Telling the User of an Interactive System where he is, what he can do, and how to get to places
Nr. 29	A.C. Shaw:	On the Specification of Graphic Command Languages and their Processors
*Nr. 30	B.Thurnherr, C.A. Zehnder:	Global Data Base Aspects, Consequences for the Relational Model and a Conceptual Schema Language
*Nr. 31	A.C. Shaw:	Software Specification Languages based on regular Expressions
Nr. 32	Engeler:	Algebras and Combinators

*Nr. 33 N. Virth: A Collection of PASCAL Programs

*Nr. 34 R.Marti, J. Rebsamen, B.Thurnherr: Meta Data Base Design - Consistent Description of a Data Base Management System

*Nr. 35 H.H.Nägeli, R.Schoenberger: Preventing Storage Overflows in High-level Languages

J. Hoppe: A Simple Nucleus written in Modula-2

Nr. 36 N. Virth: MODULA-2 (second edition)

Nr. 37 Hp. Bürkler, C.A.Zehnder: EDV-Projektentwicklung - Ein Arbeitsheft für Informatik-Studenten

*Nr. 38 H.Burkhart, J.Nievergelt: Structure-oriented editors

*Nr. 39 A.Meier, C.A.Zehnder: Flächenmodell-Register: Die Strukturen wichtiger geographischer Datensammlungen der Schweiz

Nr. 40 N. Virth: The Personal Computer Lilith

Nr. 41 T.M.Fehlmann: Theorie und Anwendung des Graphmodells der Kombinatorischen Logik

Nr. 42 E. Graf: Probabilistische Algorithmen und Computer-unterstützte Untersuchungen von probabilistischen Primalitätstests

Nr. 43 H. Burkhart: Konzepte zur Systematisierung der Benutzerschnittstelle in interaktiven Systemen und ihre Anwendung auf den Entwurf von Editoren

Nr. 44 J.Nievergelt, F.P.Preparata: Plane-sweep Algorithms for Intersecting Geometric Figures

Nr. 45 M. Reimer, J.W. Schmidt: Transaction Procedures with Relational Parameters

Nr. 46 J. Nievergelt, H. Hinterberger, K.C. Sevcik: THE GRID FILE: an adaptable, symmetric multi-key file structure

* out of stock