The Grid File: An Adaptable, Symmetric Multikey File Structure

Presentation: Saskia Nieckau

Moderation: Hedi Buchner

The Grid File: An Adaptable, Symmetric Multikey File Structure

1 Multikey Structures

A wide selection of file structures is available for managing a collection of records identified by a single key. Because of the increasing usage of databases and integrated information systems, we now need file structures, that allow efficient access to records by combinations of attribute values. This is, what is called multikey access. Balanced file structures for multidimensional data, i.e. that each record is identified by several attributes, are needed. These file structures have to be efficient, also in a highly dynamic environment, i.e. when there is a high rate of insertions and deletions. The adaptability in a dynamic environment, space utilization, and operation speed are important criteria in assessing a multikey file structure. There are also other performance criteria, which have to be discussed. The retrieval time, in which all requested records are obtained, depends on several factors. In general, the time to move blocks of data from and to peripheral storage (typically disks) dominates the processing time in main memory. Therefore, the number of required transfers from peripheral storage, that is the number of disk accesses, is used as the measure of efficiency.

2 The Grid File

Each of the multikey file structures in use today is well suited for certain environments. The grid file is an adaptable, symmetric, multikey file structure. Adaptable means, that it adapts to its contents under insertions and deletions. It is a highly dynamic file. The access time is uniform over the entire file, and a single record is retrieved in at most two disk accesses (two-disk-access principle). The grid file is symmetric, because it treats all keys symmetrically. That means, it is a file structure, which avoids distinction between primary and secondary keys. Every key-field is treated as the primary key.

2.1 Bitmap Representation

The grid file is designed to handle efficiently a collection of records with a modest number (less than 10) of search attributes, whose domains are large and linearly ordered. A record is characterized by a number of attributes. Thus, we have a linearly ordered attribute space, and a k-dimensional key to retrieve a record is represented by a point in this attribute space. The attribute space can be represented by a bitmap. 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. Now, we have one bit for each possible record in the space. The bit is set to 1, if the record is present in the file, and to 0, if it is not. But the size of this bitmap is impossibly large. Since this bitmap contains a lot of zeros, it can be compressed. Here we need a compression scheme, that is compatible with the operations executed on a file. FIND, INSERT and DELETE must be executed efficiently.

2.2 Grid Partition of the Search Space

2.2.1 Grid Blocks

The partitions are obtained by dividing the domain of each attribute into intervals.
Example for the two-dimensional case (generalization to k dimensions is obvious):

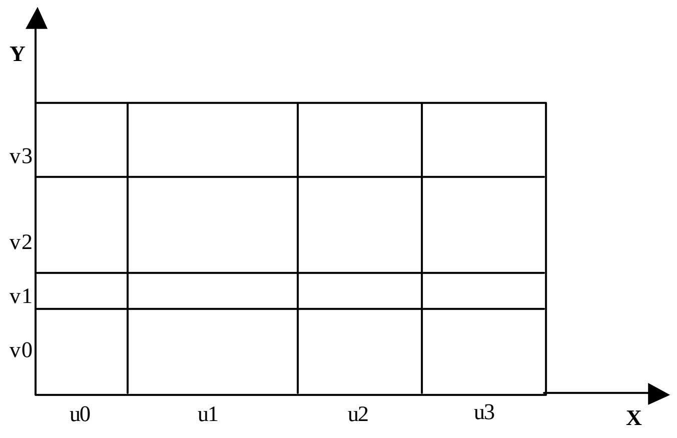

Figure 1: Grid Partition of the Search Space

As seen in Figure 1, on the record space S = X x Y we obtain a grid partition P = U x W by imposing intervals U = (u0, u1, u2, u3), V = (v0, v1, v2, v3) on each axis and dividing the record space into blocks, which we call grid blocks. With grid partitions each boundary cuts the entire search space into two. All dimensions are treated symmetrically. A file structure allocates storage in units of fixed size, called disk blocks, pages or buckets, depending on the level of description. A storage unit, that contains records, is called bucket. A bucket has a capacity c, which is the number of records it can contain.

2.2.2 Partition Modification

The grid partition is dynamic and can be modified. The grid partition P = U x V is modified by altering only one of its components at a time. A one-dimensional partition is modified by splitting one of its intervals, or by merging two adjacent intervals into one.

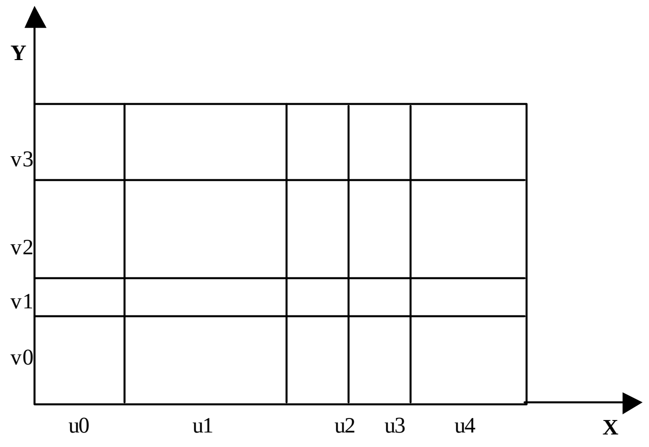

Figure 2: The interval u2 was split.

In Figure 2 the interval u2 was split. Notice, that the intervals “below” the one being split or the two being merged retain their index, while the indices of the intervals “above” the point of splitting or merging are shifted by +1 or –1.
This grid partition over the search space is an abstract space and ordered. Therefore, you can retrieve records in it. We need other operations, that relate the grid blocks and the records to each other. Finally, we can find a record by given attributes in the grid partition, and then actually retrieve it in physical storage.
The assignment of grid blocks to buckets is the task of the grid directory. Consequently, a grid file structure consists of data buckets for storing records and of a grid directory for the correspondence between grid blocks and buckets. The data structure, used to organize the set of buckets, is characteristic for the file system. The structure used to organize records within a bucket is of minor importance for the file system as a whole, because it does not influence the access time.

2.3 The Grid Directory

The grid directory has the function of a bucket management system, which is superposed onto the grid partitions. Designing this bucket management system, three aspects have to be considered:

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

2.3.1 Assignments

Because we want to retrieve a record in at most two disk accesses (the first access to the correct part of the directory, and the second one to the correct data bucket), it can easily be seen, that all the records in one grid block must be stored in the same bucket. But, if each grid block had its own bucket, the bucket occupancy would be illogically low. Therefore, it must be possible for several grid blocks to share one bucket. Thus, we have a many-to-one correspondence between grid blocks and buckets. We call the set of all grid blocks assigned to the same bucket A the region of A. A bucket region has to form a rectangular box in the space of records. The regions of buckets are pair wise disjoint. Together they span the space of records.

The grid directory represents and maintains the dynamic correspondence between grid blocks in the record space and data buckets. Thus, we need a data structure and operations, that update the assignment, if bucket overflow and underflow makes it necessary.
The grid directory consists of two parts:

- A dynamic k-dimensional array called grid array. Its elements are pointers to the data buckets and are in one-to-one correspondence with the grid blocks of the record space.
- K one-dimensional arrays called linear scales, each scale defines a partition of a domain.

Lets assume two dimensions and consequently a record space S = X x Y.
A grid directory G for a two-dimensional space is characterized by

- integers nx > 0, ny > 0 (“extent” of the directory), and
- integers 0 <= cx < nx, 0 <= cy < ny (current element of the directory, which is equal to the current grid block).

Thus, the directory consists of

- a two-dimensional array G(0 …, nx-1, 0 …, ny-1), the grid array, and
- two one-dimensional arrays X(0 …, nx), Y(0 …, ny), the linear scales.

2.3.2 Operations on the Grid Directory

The following operations are defined on the grid directory.

Direct access: G(cx, cy)

In this case the element, which we search for, is set as the current element.

Example: We have a record space with the attribute "year" and the domain "0 … 2000", and the attribute "initial" with the domain "a … z". In Figure 3 we have following grid partition: X = (0, 1000, 1500, 1750, 1875, 2000) and Y = (a, f, k, p, z).

FIND [1980, w] is executed like this:

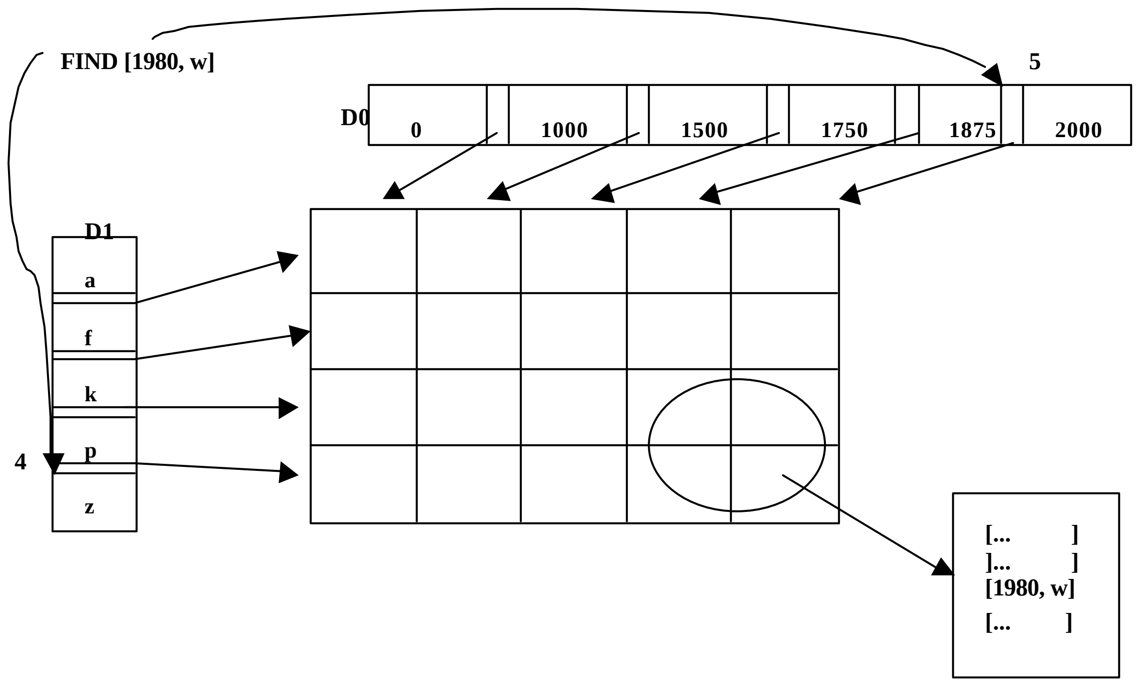

Figure 3: Direct Access on the Grid Directory

The attribute value 1980 is converted into interval index 5 through a search in scale X, and the attribute value w is converted into interval index 4 through a search in scale Y. The interval indices 5 and 4 provide direct access on the correct element of the grid directory, where the bucket address is located.

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

The interval, which should be merged, is given by the element of the grid directory. Lets call it px or py. Now this interval is merged with the one below. Consequently, all elements of the grid directory, which lie above px or py, have to be renamed, i.e. their indices are shifted by –1. Finally, the linear scales of the dimension, in which the two intervals have been merged, have to be adapted.

We can only split or merge in one dimension at a time. Here we assumed two dimensions. Hence there are algorithms for merging the dimension X and for merging the dimension Y.

- 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

Again the interval, which should be split, is given by px or py. A new element, i.e. a new interval, is created. Now all the indices of the intervals above px or py are shifted by +1. Finally, the linear scales are adapted again.

- 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

Example: Building up a file under repeated insertions

Since the elements of the grid directory are in one-to-one correspondence with the grid blocks, instead of showing the grid directory, the buckets pointers are drawn as originating directly from the grid blocks. The bucket capacity is 3.

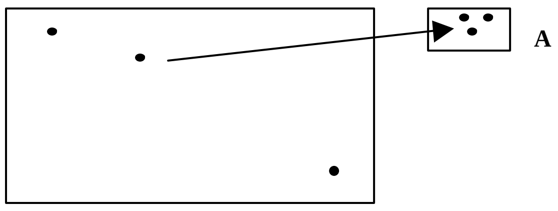

Figure 4: File with only three records.

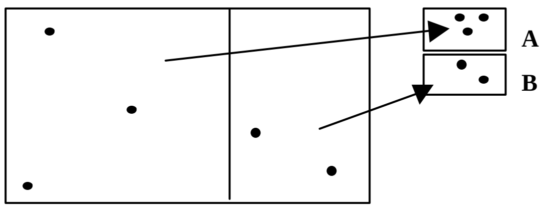

Figure 5: Insertion of two records

In Figure 4 a single bucket A is assigned to the entire record space. In Figure 5 bucket A overflows, the record space is split and a new bucket B is made available. Those records, that lie in one half of the space (= in one grid block), are moved from the old bucket to the new one.

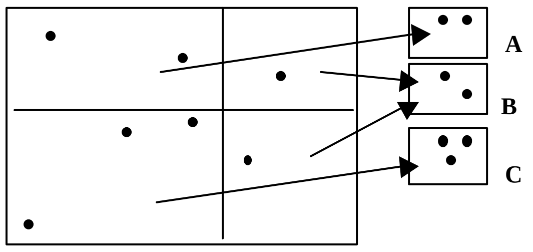

Figure 6: Insertion of two more records.

In Figure 6 bucket A overflows again. Its grid block is split according to some splitting policy. Those records of A, that lie in the lower-left are moved to a new bucket C. Since bucket B did not overflow, it is left alone. Therefore, its region now consists of two grid blocks.

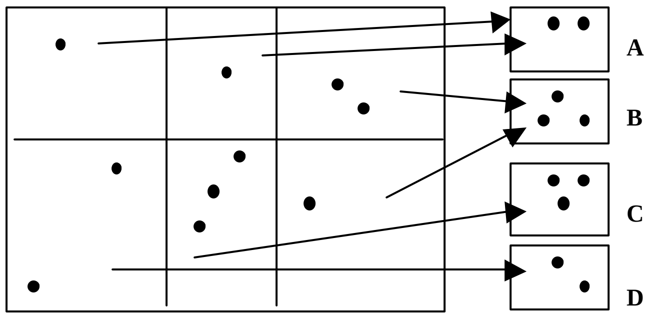

Figure 7: Insertion of three more records

In Figure 7 bucket C overflows, the shown refinement of grid partition takes place and bucket C is split into buckets C and D.

2.4. Environment-Dependant Aspects

The grid file is a general-purpose data structure, a general design is desirable. Following decisions have to be made in order to optimise performance in a specific environment:

- Choice of splitting policy
- Choice of merging policy
- Implementation of the grid directory

2.4.1 Splitting Policy

The different splitting policies result in different refinements of the grid partition.
As seen in the example above, if a partition refinement occurs, the dimension (the axis, along which the grid block should be split) and the location (the point at which the linear scale is partitioned) have to be chosen. The simplest splitting policies choose the dimension according to a fixed schedule. Other splitting policies may favour some attributes by splitting the corresponding dimensions more often than others.
The location of a split on a linear scale need not necessarily be chosen at the midpoint of the interval. Little is changed, if the split point is chosen from a set of values, that are convenient for a given application; for example, months or weeks on a time axis.

2.4.2 Merging Policy

There are two kinds of merging: bucket merging and merging of cross sections in the grid directory. In most applications, it is needless to reduce the directory size as soon as possible, because it will soon grow back to its earlier size. For bucket merging there are different policies.
A merging policy is controlled by following decisions:

1. Which pairs of neighbouring buckets are candidates for merging their contents into a single bucket?
   There are two different systems for determining pairs. Lets first look again at the example above. It can be seen, that the history of repeated splitting can be represented in the form of a binary tree, which imposes a twin system on the set of buckets (and hence on the set of regions): each bucket and its region have a unique twin, from which it is split off. In this example, C and D are twins, the pair (C, D) is A's twin, and the pair (A, (C, D)) is B's twin. Merging can proceeded from the leaves up in the twin system tree. The twin system is also called the buddy system. A bucket can merge with exactly one adjacent twin (or buddy) in each of the k dimensions.

The second system is the neighbour system. Here, a bucket can merge with either of its two adjacent neighbours in each of the k dimensions, providing that the resulting bucket region is convex.

If according to the chosen merging policy there are more than one bucket, which are able to merge with a particular bucket, it has to be specified, which one has priority.

2. At what bucket occupancy merging is actually triggered?

   To determine the bucket occupancy at which merging is triggered, a merging threshold of p percent is set. The contents of two buckets are actually merged, if the occupancy of the resulting bucket is at most p percent. Merging thresholds around 70 percent are reasonable, those above 80 percent lead to poor performance.

2.4.3 Implementation of the Grid Directory

The data structure to implement a grid directory has to be chosen, for example linked lists or k-dimensional arrays.

Linked lists are suitable for data structures, where there are lots of insertions and deletions. But since with linked lists you need to "follow" pointers to find an element, fast access is not guaranteed if these pointers lie on different pages. This is very likely to happen because a grid directory of several dimensions extends over several pages. Therefore, it might be better to represent the grid directory by a k-dimensional array.

2.5 Simulation experiments

To analyse the behaviour and performance of the grid file, some simulations have been done. The performance of a file is determined by two criteria: processing time and memory utilization. In view of the fact, that the grid file holds the two-disk-access principle, we only have to find out, how the available space is used. In the simulation runs, we will look closer at following objectives:

- Estimation of average bucket occupancy

  The average bucket occupancy does not need to be close to 100 percent, but it must be prevented from becoming arbitrarily small.
- Estimation of directory size

  The asymptotic growth rate of the directory size is not known, but for realistic file sizes the grid directory only needs a small part of the space, which is needed for data storage.

- Evaluation of splitting and merging policies

The following types of files are tested and their performances are compared:

- The growing file; it results from repeated insertions.
- The steady-state file; here in a long run, the number of insertions is equal to the number of deletions.
- The shrinking file; it results from repeated deletions.

2.5.1 The growing file

Analysing the behaviour of a growing file is a test for the splitting policy of a file system. First we look at the average bucket occupancy.

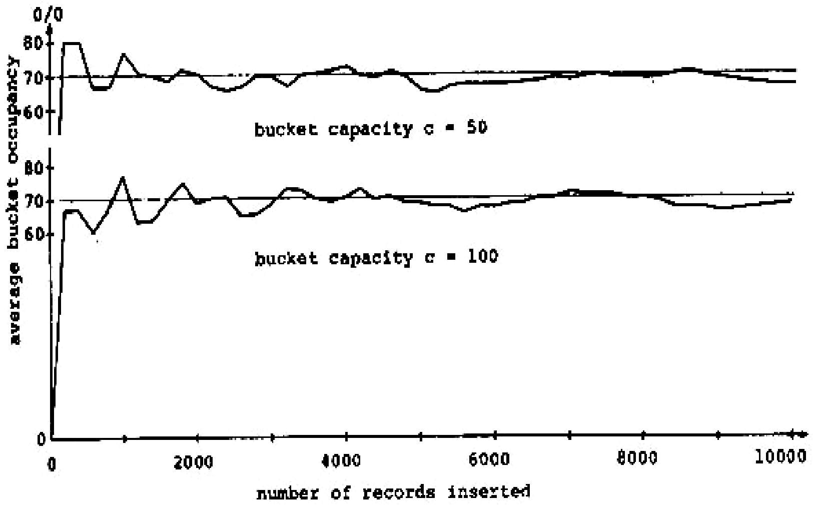

Figure 8: Average Bucket Occupancy of a Growing File

Under an insertion of records the bucket occupancy always develops as seen in Figure 8. As soon as the number of inserted records reaches a small multiple of the bucket capacity c, the average bucket occupancy shows a steady-state behaviour with small fluctuations around 70 percent. Because of the fact, that a bucket may be shared by many grid blocks, nothing can be said about the directory growth in comparison to the amount of data stored. For instance, correlated attributes, i.e. attributes that depend on each other such as age and date of birth, are unlikely to significantly affect the average bucket occupancy, but they are very likely to increase directory size significantly. Another aspect to consider is, that the grid file is not immune to the worst-case situation. This is, if all records come to lie within a tiny area. In that case the number of grid blocks grows faster than the number of inserted records. The point, at which a linear growth rate begins, depends strongly on bucket capacity c and turns out to be sufficiently large.

2.5.2 The steady-state file

The number of records remains approximately constant, because in a long run there are as many insertions as deletions. A steady-state file tests the interaction between the splitting and the merging policies. In Figure 9 different values of the merging threshold (the percent occupancy, which the resulting bucket should not exceed, when two buckets are merged) are tested.

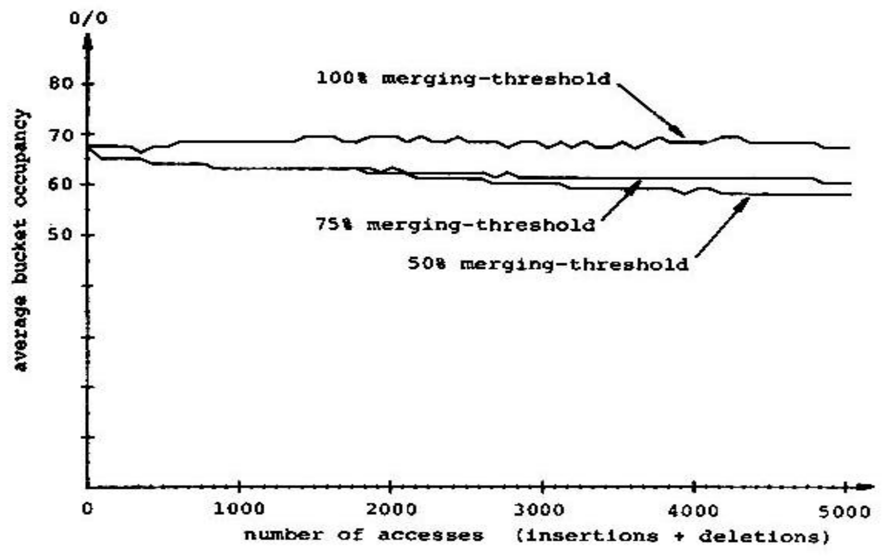

Figure 9: Average Bucket Occupancy with Different Merging Thresholds

2.5.3 The shrinking file

With testing a shrinking file, the effectiveness of the two merging policies, which are based on the buddy and the neighbour system, are compared. Looking at the average bucket occupancy of a shrinking file operated under the buddy system with different merging thresholds, the following can be seen in Figure 10:

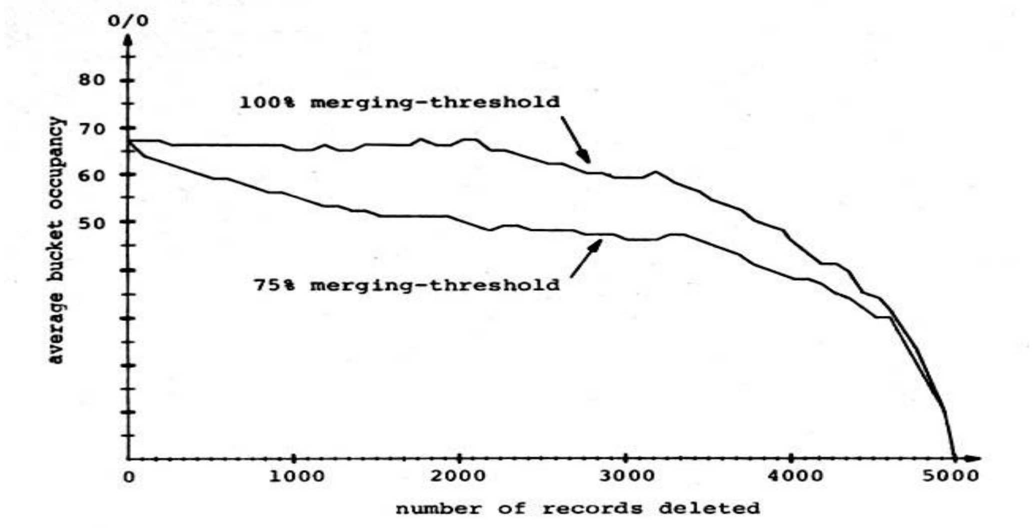

Figure 10: File Operated under the Buddy System

The buddy system does not guarantee a high average bucket occupancy over a long period of deletions. Since we might know, that the file is shrinking, we could set the merging threshold to 100 percent. But this only helps in the early part.

In comparison, as seen in Figure 11, the neighbour system does not go through any degradation in average bucket occupancy.

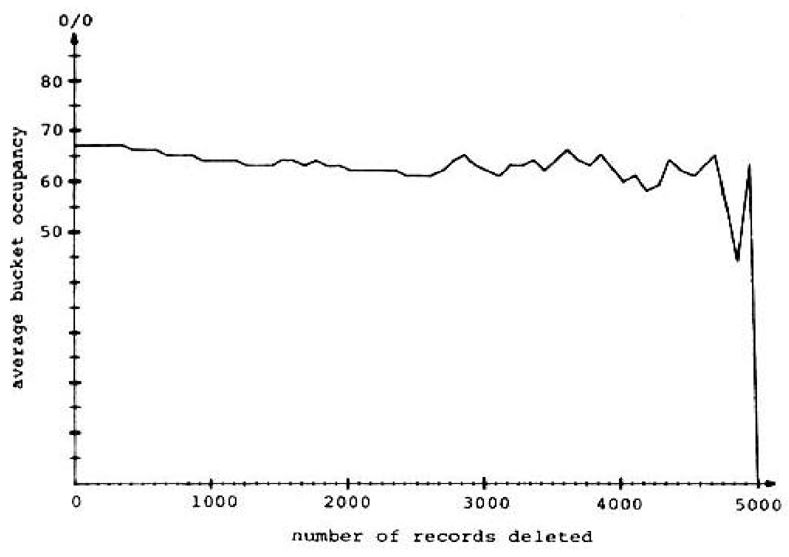

Figure 11: File Operated under the Neighbour System

3 Extension

Simulation results show, that the grid file uses space economically over a wide range of operation conditions. Taking everything into account, it can be seen, that 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. But attribute correlations affect the size of the directory. In the grid file, the growth of the directory is highly affected by data distribution and correlation among different attributes.

3.1 Non-local Splitting Strategy

The grid file applies the non-local splitting strategy; i.e. it splits the entire cross section (hyperplane), that includes the region corresponding to the overflowed data bucket. As you can see in Figures 12 and 13. By doing this unnecessary directory entries may be created.

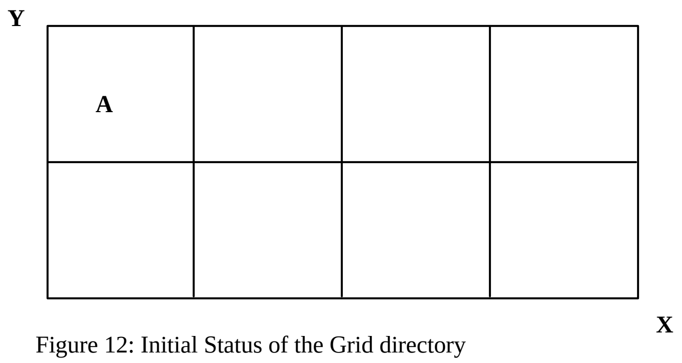

Figure 12: Initial Status of the Grid directory

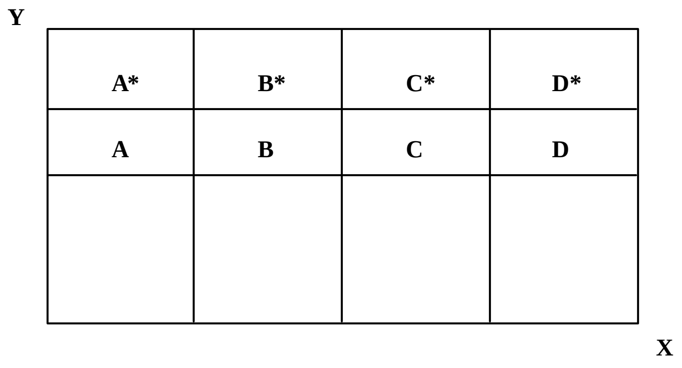

Figure 13: Non-local Splitting Strategy of the Grid File

The problem occurs, when data is distributed nonuniformly, because the number of directory entries no longer grows in a linear fashion.

3.2 Local Splitting Strategy

To solve this problem of non-linear growth of the directory of the grid file, a local splitting strategy can be applied, as it is done in the multilevel grid file. This can be seen in Figure 14:

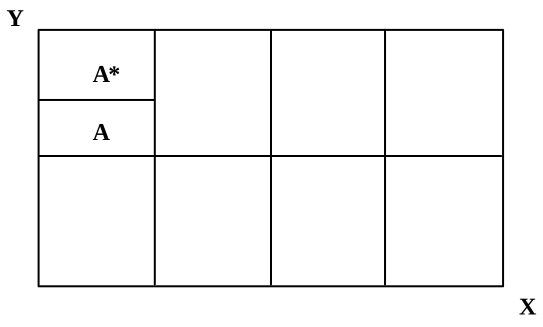

Figure 14: Local Splitting Strategy of the Multilevel Grid File

3.3 The Multilevel Grid File

In the multilevel grid file the directory entries are in one-to-one correspondence not only with the regions on the attribute space, but also with the data buckets. Therefore, each data bucket contains only those records, that belong to the region represented by one directory entry. A directory entry consists of a region vector and a pointer to a data page or a lower-level directory page. For a two-level directory, for example, the level directory D2 is built on top of the level directory D1. A region represented by a directory entry is subdivided into disjoint regions, which are represented by the next lower-level directory. The number of directory entries in the lowest level directory is identical to the number of data buckets. With the local splitting strategy and by keeping only those directory entries, that represent non-empty regions, the multilevel grid file adapts to nonuniform and correlated data distribution. Extensive experiments with various data distributions have shown that the size of the multilevel grid directory increases linearly in the number of records.

4 Conclusion

For a file filling up in its early stage, and for a file in steady-state or going through brief shrinking phases, the space utilization of a grid file is good. The grid file is designed to handle efficiently a collection of records with a modest number of search attributes. Within this usage environment, it combines quite a few of the better properties of multikey file structures:

- A high data storage utilization of 70 percent.
- Smooth adaptation to the stored contents, particular under file growth.
- A directory, which is quite compact.
- The two-disk-access principle and hence fast access to individual records.
- Efficient processing of range queries.

5 References

- Jürg Niervergelt, Hans Hinterberger, Kenneth C. Sevcik: The Grid File: An Adaptable, Symmetric Multikey File Structure. ACM Transactions on Database Systems (TODS), Volume 9 (1): 38-71 (1984)
- Sang-Wook Kim, Kyu-Young Whang, Jin-Ho Kim: Linearity in directory growth of the multilevel grid file. Information and Software Technology 39: 897-908 (1997)