PLOP-Hashing: A Grid File without Directory

Hans-Peter Kriegel Bernhard Seeger

Praktische Informatik,University of Bremen,D-2800 Bremen,West Germany

Abstract

In this paper we consider the case of nonuniform weakly correlated or independent multidimensional record distributions. After demonstrating the advantages of multidimensional hashing schemes without directory, we suggest piecewise linear expansions in order to distribute the load more evenly over the pages of the file. The resulting piecewise linear order preserving hashing scheme (PLOP-Hashing) is then compared to the 2-level grid file which turned out to be the most popular scheme in practical applications.

1.Introduction

Let us consider a file of d-attribute composite keys $K = (K_1, .., K_d)$, $d > 1$. Our goal is to maintain a file efficiently supporting the following operations:

1. insertion and deletion of records
2. exact match query
3. partial range and partial match query

Obviously, there is a large variety of schemes, organizing files under these requirements. There are tree-based methods, like the K-D-B-tree [Rob 81] or multidimensional B-trees ([SO 82], [Kri 84]), multidimensional dynamic hashing (MDH) schemes with directory, like the grid file, and MDH without directory, like the quantile hashing [KS 87a]. Furthermore, in the last years hash trees were proposed ([Ouk 85],[WK 85],[Oto 86],[Fre 87]) which are MDH schemes where the directory is organized by a tree structure, such as a K-D-B-tree. Unfortunately, there is no scheme with a good overall performance. If the records are strongly correlated, hash trees will outperform all other schemes. Nevertheless, the grid file and the quantile hashing are more efficient than hash trees in case of independent or weakly correlated records.

In this paper our goal is to propose a new MDH scheme without directory, which is more efficient than the grid file and quantile hashing. We do not expect best performance for strongly correlated records. The basic motivation to propose a scheme without directory is derived from the following disadvantages of the grid file:

1. the directory size is $O(n^{1+(d-1)/d*b})$ [Reg 85] even for uniform record distribution, where b is the capacity of a bucket and n is the number of records in the file. For nonuniform distributions, the directory can grow exponentially.
2. the expansion of the directory costs $O(N^{1-1/d})$ disk accesses, where N is the number of directory elements. Unfortunately, the expansion must be carried out in one step.
3. To answer an exact match query, two disk accesses are guaranteed, but two disk accesses are also needed.

This paper is organized as follows. In section 2 we give a brief introduction to MDH schemes without directory. In the third section we present our new scheme, in particular, we introduce the address function of our scheme and the organization of the binary trees. A description of the algorithms for expansion and contraction of the file is given in section four. In the fifth section, we describe the retrieval algorithms. In the sixth section, we report on the performance of an implementation of our scheme in comparison to the 2-level grid file. Section 7 concludes the paper.

2.Multidimensional dynamic hashing without directory

MDH schemes without a directory are based on (one-) linear hashing [Lit 80]. Using a hashing function H, we compute the address of a short chain of buckets, where the number of chains depends on the number of records stored in the file. Giving up the directory we have to allow overflow records, i.e. records which cannot be placed in the first bucket of the corresponding chain, called primary bucket. The overflow records are stored in a so-called secondary bucket which is chained with the primary bucket. The primary bucket resides in the primary file, the secondary bucket in the secondary file. This very simple type of treating overflow records is called bucket chaining, one chain of buckets is called a page. Obviously, overflow records can be handled more efficiently with other strategies, such as recursive hashing [RS 84] and overflow handling in the primary file [Lar 85].

The expansion and contraction of the file is triggered by a rule, called control function. For instance, if the storage utilization exceeds a fixed threshold the file is expanded by one page. A global pointer p points to the page which will be expanded next. The records in page p are divided into two groups of about equal size. One group remains in the old page p, the other group is allocated in the new page. Assuming an initial file of one page, the expansion of the file is linear, if

1. After doubling the file size, the file consists of 2^L pages, $L > 0$, addressed by 0,1,..,2^L-1. Then a sequence $\{p_j\}_{j=0}^{2^L-1}$, $0 \leq p_j < 2^L, p_i \neq p_j$ for $i \neq j$, is determined in which the pages will be expanded. The page which will be split first is p_0.

CH2550-2/88/0000/0369$01.00 © 1988 IEEE

2. If page p_j, $0 \leq j \leq 2^L - 2$, is expanded, the page which will be expanded next is page p_{j+1}

A variable L denotes the level of the file, i.e. it indicates how often the file size has doubled. In case of linear hashing [Lit 80] the sequence $p_j = j$, $0 \leq j < 2^L$ is chosen in 1.

The basic paradigm of hashing schemes is that best retrieval performance is achieved, when the records are distributed as uniformly as possible over all the pages of the file. Assuming uniform record distribution, a linear expansion of the file creates a uniform load. However, nonuniformly distributed records will affect retrieval performance in case of linear expansion. Last year quantile hashing was proposed. Although it uses linear expansions, quantile hashing avoids this performance penalty for nonuniform independent record distributions. The basic concept of quantile hashing is the estimation of the optimal grid position of the data space using stochastic approximation methods. Nevertheless, there are some drawbacks:

1. the estimation is influenced by the sequence, in which the records are inserted. In particular a sorted sequence implies a slow adaptation to the optimal grid.
2. a slow adaptation leads to a large reorganization overhead

In this paper, we suggest to expand the file in a piecewise linear fashion. The pages are arranged in k groups, $k < m$, where k varies in the number of records stored in the file. The group of pages which is presently expanded linearly is referred to by a pointer gp. In one step, one of the pages of group gp is split. After completely doubling the number of pages in the selected group, pointer gp will refer to a new group of pages which will be expanded next in a linear fashion. For good performance the group with the highest load factor is selected. We would like to emphasize that this is the first MDH scheme without directory with a free choice of the group of pages to be expanded next.

3.PLOP-Hashing and its address function

Two basically different order preserving address functions have been suggested for MDH schemes without directory. One of them is the interpolation function [Bur 83] which generates a one-dimensional key from a d-attribute composite key using z-ordering [OM 84] and then computes the address using a one dimensional order preserving hashing function. The other address function is the one used in MOLHPE [KS 86] and quantile-hashing [KS 87] which was originally suggested to compute directory addresses in multidimensional extendible hashing [Oto 84]. In our approach we will use this address function to compute page addresses.

As already indicated, the data space is partitioned by an orthogonal grid, see figure 1. The partitioning points of the grid on each axis are defined by d binary trees, $d \geq 1$, comparable to the scales of the grid file. Each inner node of such

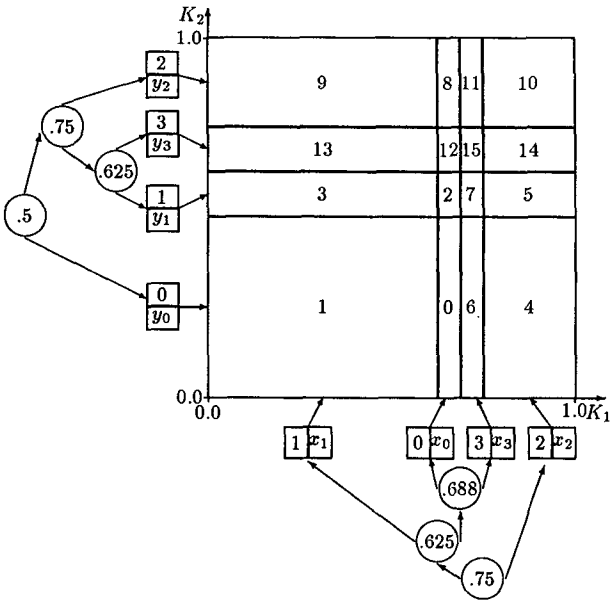

Figure 1: Partition of the data space $[0,1)^2$ generated by PLOP-Hashing

a binary tree stores a partitioning point representing a (d-1)-dimensional hyperplane that cuts the space into two rectangular shaped regions. Each leaf is associated with a d-dimensional slice S(i,j) of the data space which is bounded by two neighboring partitioning hyperplanes, $0 \leq i < m_j, 1 \leq j \leq d$, where m_j is the number of slices in the j^{th} axis. Such a slice S(i, j) is addressed by the index i stored in the corresponding leaf, $0 \leq i < m_j$, $1 \leq j \leq d$.

The whole data space is the union of d-dimensional rectangles which are not cut by any partitioning hyperplane and are therefore called cells. All the d-dimensional points lying in one cell are stored in one page. The address of that page is computed using the index i_j of all slices $S(i_j, j)$, whose intersection results in the corresponding cell, $0 \leq i_j < m_j, 1 \leq j \leq d$. Additionally for an index i, each leaf contains the number x_i^j of points which are in the slice S(i, j), $0 \leq i < m_j, 1 \leq j \leq d$. This information is used to expand the file in a piecewise linear fashion. As already mentioned, the desired page address is computed by $G(i_1, .., i_d)$, where G is the address function and i_j is the index of the j^{th} axis. In order to give an intuitive understanding of the function G, we depict the addresses of figure 1 produced by the function G dependent on the index array I=(i_1, i_2) in figure 2. Originally, the file consisted of 4 pages with addresses 0,1,2,3. One doubling of the file corresponds to adding the pages with addresses 4,5,6,7. These new addresses are in lexicographical ordering with respect to (i_1, i_2). One further doubling of the file creates new pages with addresses 8,..,15. Again these new addresses are in lexicographical ordering, but now with respect to (i_2, i_1).

We are now ready to present the address function G. Ob-

i_2 \ i_1	0	1	2	3
3	12	13	14	15
2	8	9	10	11
1	2	3	5	7
0	0	1	4	6

Figure 2: Addresses of a file with 16 pages depending on the index (i_1, i_2)

viously, G will depend on the level L of the file which indicates how often the file size has doubled. The expansion axis s which is the axis where the next expansion is carried out is advanced in a cyclic order, i.e. s = L MOD d + 1. A cyclic order of the expansion axes is not necessary, but it simplifies the explanation of the address function. For example, in figure 1 the level is L=4 and s=1. Each axis has its own level L_j, $1 \leq j \leq d$, which is given by

$$L_j = \begin{cases} L\,DIV\,d + 1 & \text{if } j \in \{1,..,s-1\} \\ L\,DIV\,d & \text{if } j \in \{s,..,d\} \end{cases}$$

The function G is given by

$$G(i_1,..,i_d) = \begin{cases} 0 & \text{if } \max\{i_1,..,i_d\} = 0 \\ i_z * \prod_{j \in M} J_j + \sum_{j \in M} c_j * i_j & \text{otherwise} \end{cases}$$

where z,M,t,J_i,c_i are given as follows:

$z = \max\{\, j \in \{1,..,d\} \mid \lfloor \log_2 i_j \rfloor = \max_{1 \leq k \leq d} \lfloor \log_2 i_k \rfloor \}$

$M = \{1,..,d\} \setminus \{z\}$

$t = \lfloor \log_2 i_z \rfloor$

$$J_i = \begin{cases} 2^{t+1} & \text{if } i < z \\ 2^t & \text{otherwise} \end{cases}, i \in M$$

$$c_i = \prod_{r=i+1, r \neq z}^{d} J_r \quad , i \in M$$

The most important property of the address function G is that it allows to cut a slice into two. This is exactly what the grid file does when expanding the directory. Selecting a slice and cutting it into two corresponds to a piecewise linear expansion. In a more general setting the address function allows distributing the objects of c slices over c+1 slices and thus supports partial expansions as introduced for one-dimensional linear hashing [Lar 80] and for multidimensional hashing in [KS 86]. For the sake of simplicity, we will not treat partial expansions in this paper.

Another important aspect of the address function is that an assumption about the data space is not necessary. Moreover with some little modifications, PLOP-Hashing allows a dynamic varying domain for each axis and thus needs no information on an upper bound of the domain. Contrary to PLOP-Hashing the grid file has to know the limits of the data space

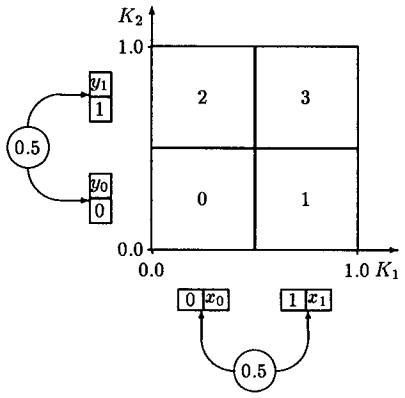

Figure 3: Initial situation of the file: L=2, s=1

and, furthermore these limits cannot be changed during the whole life cycle of the file. In particular for the organization of spatial objects, dynamic domains can improve performance essentially, as demonstrated in [SK 87].

As mentioned before, the address function G was originally proposed to compute the addresses of the directory for multidimensional extendible hashing [Oto 84]. In case of the grid file [NHS 84] the problem how to implement the directory should is left open. The directory which is a d-dimensional dynamic array is in most implementations organized by simple lists. Thus an expansion or contraction of the directory is very time consuming. Using the address function G combined with the binary trees as proposed in this section is an efficient possibility for an implementation of the directory of the grid file. However, as we will see, using the address function G without a directory as in PLOP-Hashing is even more efficient.

4.Expansion and contraction of the file

The dynamic behavior of our scheme is best explained by tracing an example. Let us start from a file on level L = 2 and s = 1 storing 2-dimensional records. Since L = 2, the file consists of 4 pages. The corresponding situation is depicted in figure 3. As mentioned before, the control function can be tuned to the particular application. For instance, we can control the storage utilization as proposed for linear hashing [Lit 80], or we can control the length of the chains similar to the grid file. In our example we have chosen the following control function:

(Ex α) Determine the slice S(gp,s) of the expansion axis s containing the maximum number of records. If the load factor in this slice is more than α an expansion of the file is required.

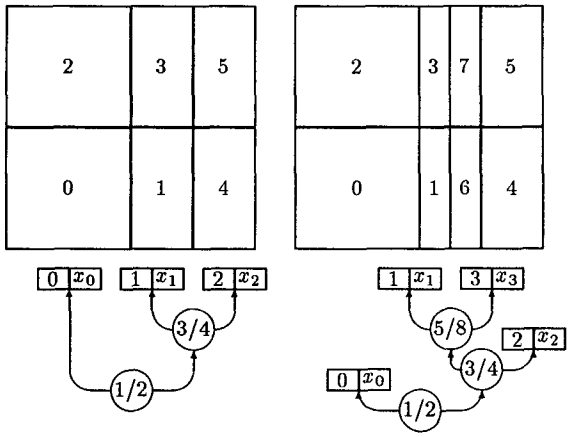

Figure 4: Expansion of the file

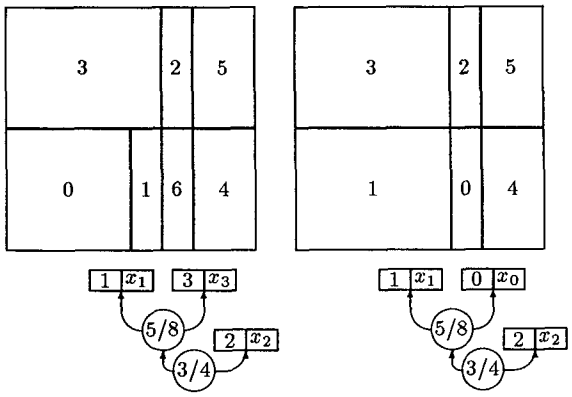

Figure 5: Contraction of the file

In the following we have chosen $\alpha = 100\%$. Assuming $x_1 > x_0$ and $x_1 > 2b$, i.e the load factor is more than 100% , then the slice S(1,1) will be expanded by another slice. Thus we will insert a new partitioning point into the binary tree for the x-axis. For simplicity sake, we will choose the middle of the interval to be split as the partitioning point. This can be improved by computing an estimate for the 50%-quantile (median) of the split interval. Contrary to the grid file, the expansion of one slice is not carried out in one macro step, but step by step. This is only possible since our scheme uses no directory and allows overflow records. Thus our scheme is more dynamic than the traditional grid file which may require $O(N^{1-1/d})$ page accesses for one insertion, where $N > O(n)$ is the number of elements in the grid directory. Since our scheme expands one slice page by page in a linear fashion, the insertion time is bounded by the time it takes to expand the file by one page. The right side of figure 4 shows the situation after completing the expansion.

After further insertions, the control function (Ex 100%) may again require expansion of the file. Until the number of slices in the x-axis has doubled, the x-axis remains expansion axis. Now, let $x_1 = \max\{x_0, x_1, x_2\} > 2b$. Then the slice with the index $i_1 = 1$ corresponding to the interval [0.5,0.75) is selected for expansion. Again this slice is expanded step by step, resulting in the situation depicted in figure 4. Since now the number of slices in the x-axis has doubled, the next expansion will be carried out in the direction of the y-axis. In addition to expansion of the file we have to consider contraction and reorganization, i.e. adaptation of the file to the present distribution. For this purpose, we choose the following control function for contraction of the file:

(Co β) Within the merging axis, determine the pair of neighboring slices with the minimum number of objects. If the load factor of both slices is below β a contraction of the file, i.e merge of these slices to one slice is required.

In the remainder, the value of β is 45%. Most likely, the merging axis is identical to the expansion axis. However, there is one exception after just doubling the file size, where the expansion axis is updated and the merging axis refers to the preceding expansion axis. In our example on the right side of figure 4 the expansion axis is the y-axis and the merging axis is the x-axis. Let us now assume that merging is required by the control function. Since we want to guarantee a dynamic insert and delete behavior and we want to keep the set of page addresses compact, merging of the two slices proceeds as follows. First, the binary tree is reorganized, i.e the two leaves with index 1 and 0 are merged, the corresponding partitioning point 0.5 is deleted. Then the first merge step is carried out, where pages 2 and 3 in figure 4 are merged and the primary bucket of page 7 - the page with the maximum address - is copied onto the primary bucket of page 3, see the left side of figure 5. Thus we can guarantee that at most three pages are involved in a merge step. One further merge step finishes the contraction, see right side of figure 5. For the operations splitting a slice and merging two slices there is one basic rule to be observed: it is not allowed to perform two operations at the same time. The first operation has to be completed before the second operation may start.

5.Queries

The algorithms for queries are similar to those given for MOLHPE [Kri 86] due to using the same address function G.

Let us consider first an exact match query. Given a key $K = (K_1, .., K_d)$, an index i_j for each axis j, $1 \leq j \leq d$, is obtained by searching the corresponding binary tree for the

given component key K_j. The only complication arises from the fact whether or not the desired page is in the slice which is presently expanded or in one of the two, respectively three, slices which are presently involved in a merge. Because the case of contracting the file is very technical, we will only consider the case of expanding the file. For this purpose we introduce an expansion index Ex = $(ex_1, .., ex_s, .., ex_d)$ which indicates the address G(Ex) of the page to be expanded next. By lexicographically comparing the two index arrays I = $(i_1, .., i_d)$ and Ex we can determine whether the page is in the expansion axis and furthermore whether the page has already been expanded or not. With this knowledge the desired address can be computed.

Example 5.1:
Let us consider a file on level L = 2, where d = 2, s = 1 and Ex = (1,1). The corresponding situation is depicted in figure 6.

1. Let us assume we perform an exact match query for key K = (0.2,0.7). By searching the binary tree we determine the index array I = (0,1). Since $ex_1 \neq i_1$, the page containing K is not in the expansion slice and thus its address is G(0,1) = 2.
2. Let us assume we perform an exact match query for key K' = (0.7,0.2). The corresponding index array is I' = (1,0). Since $ex_1 = i'_1$, the page containing K' is in the expansion slice. Since $I' = (1,0) <^L (1,1) = Ex$ (where $<^L$ denotes the lexicographical ordering), the page containing K' has already been expanded. Therefore we have to recompute the index i_1. In our example the new value is $i_1 = 3$ and the corresponding address is G(3,0) = 6.

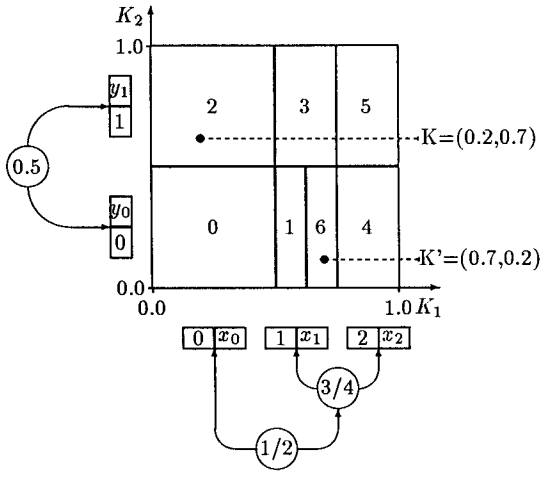

Figure 6: Example for address computation

The computation of the address of a neighbor page of a given page is the basic operation in a range query where we have to access to all pages which intersect the search region of the specified rectangular range. Let us assume we are given the index I of a page and we want to determine the address of its right neighbor page with respect to the j^{th} axis, $1 \leq j \leq d$. All we have to do is to determine the right neighbor index r_j of the given index i_j in the binary tree of the j^{th} axis. In order to support this operation efficiently, the leaves of the binary trees are linked in both directions using pointers. Given I and the right neighbor index r_j, the address of the right neighbor page is determined by computing

$$G(i_1, .., i_{j-1}, r_j, i_{j+1}, .., i_d)$$

Again complications can arise, if a split or merge process is executed.

6.Performance of PLOP-Hashing

In order to demonstrate the performance of our scheme, we will compare it in a theoretical as well as in a practical setting with the grid file [NHS 84]. In particular we will demonstrate in this section that in case of nonuniform correlated records, PLOP-Hashing outperforms the grid file. First we will give a brief introduction to the grid file.

The grid file is a multidimensional dynamic hashing scheme with directory based on extendible hashing [FNPS 79]. Originally the directory was organized by a d-dimensional dynamic array, where each array component corresponds to a cell of the grid partition. The partitioning points are stored in main-memory-resident scales, one for each axis of the data space. The grid file avoids overflow records by splitting the corresponding block. Therefore the grid file realizes the two-disk-access principle for an exact match query: using the scales a record is converted into a d-dimensional index. The index provides direct access to the correct component of the grid directory on disk, yielding the address of the data bucket, which contains the record. Because the directory size is $O(n^{1+(d-1)/(d*b)})$ [Reg 85] for uniformly distributed records, and can be exponential for some nonuniform record distributions, a 2-level organization was suggested to avoid this performance penalty [Hin 85]. The first level of the directory consists of the root directory, which is organized like the grid directory. The components of the root directory refer to subdirectories, also organized as grid directories, where a subdirectory is exactly in one bucket. The components of the subdirectories point to the data buckets. If we keep the root directory in main memory, the two-disk-access principle for an exact match query is also realized for the 2-level grid file. Nevertheless, the asymptotic behavior of the 2-level version is the same as for the 1-level version.

6.1 Worst case performance of PLOP-Hashing

Let us now consider keys K=$(K_1, .., K_d)$ with $K_i = K_j$, $1 \leq i, j \leq d$, where the key components are uniformly distributed.

	grid file	PLOP-Hashing
directory size	$O(n^d)$	-
exact match query	2	$O(n^{1-1/d})$
insertion	$O(n^{d-1})$	$O(n^{1-1/d})$
size of the resident scales	O(n)	$O(n^{1/d})$

Table 1: Worst case performance of PLOP-Hashing

Obviously, this distribution does not appear in practice, but it will give us the worst case behavior of PLOP-Hashing which is reported in table 1. Comparing both schemes for this record distribution the grid file seems to be impractical. Nevertheless the performance of PLOP-Hashing is also degenerated. However, we would like to emphasize once more that the above distribution forces PLOP-Hashing for the given control functions (Ex α) and (Co β), $\beta < \alpha/2$, to its worst case behavior. Even for this distribution, the grid file is still far away from its worst case performance. In the worst case the grid directory can grow exponentially.

Now we want to explain the data presented in table 1. Using the control function (Ex α) and (Co β), $\beta < \alpha/2$, guarantees a linear growth of the data pages in the number of records. Since we partition the data space symmetrically, the number of slices is $O(n^{1/d})$ and thus the binary trees need $O(n^{1/d})$ space. Eventually PLOP-Hashing reaches the state that in the expansion axis no slice can be split and no two neighboring slices can be merged. Thus the number of records in one slice and obviously in one page is $O(n^{1-1/d})$. Insertion of records can trigger a split of a slice and thus the worst case for an insertion is also $O(n^{1-1/d})$.

Let us now consider the grid file and particularly a quadratic region of a non-empty data bucket. An insertion of an partitioning point in each axis, which halves the region, produces 2^d subregions, where only two subregions contain records. The grid file supports a symmetric partitioning of the data space. Then every region of a data space must be as quadratic as possible. Thus to yield three non-empty buckets from our original bucket, we must introduce d+1 new partitioning points. Thus the number of partitioning points is O(n) and the directory growth is $O(n^d)$. Introducing a new partitioning point costs $O(N^{1-1/d})$ disk accesses, where N is the number of directory elements. Thus an insertion can cost $O(n^{d-1})$.

6.2 Experimental comparison of the grid file and PLOP-Hashing

To compare the average case performance of the grid file and PLOP-Hashing, we used implementations of these in Modula-2 on an Olivetti M24 PC under MSDOS. We are thankful to Dr. Klaus Hinrichs, who has implemented the 2-level grid file [Hin 85] at the ETH Zuerich and who has made this implementation available to us.

To demonstrate the performance of the schemes, we have generated 4 files (F1)-(F4), each of which consisted of 30,000 records. In all our experiments we consider only the case of the growing file, where no deletions occur. As a performance parameter we have measured the number of disk accesses per operation. When building up the file from empty, the following values were measured:

1. the average number of disk accesses for an insertion
2. the average storage utilization
3. the average number of disk accesses for a successful and unsuccessful exact match search
4. the worst case number of disk accesses for an exact match search
5. the worst case number of disk accesses for an insertion during the last 2,000 insertions
6. the number of directory blocks

For the sake of clarity, we will give an exact description of our experiments. In the following we consider records whose keys are in the unit cube $[0,1)^d$ and follow a uniform distribution, a Gaussian distribution N(m,v) with mean value m and variance v, or a geometric distribution Geo(z) with parameter z, $0 \leq z \leq 1$. Each key component K can be represented as a bitstring $(b_1, b_2, ..)$, where $K = \sum_{j \geq 1} b_j * 2^{-j}$. A key component K follows a geometric distribution Geo(z) with parameter z, if for any j, the bit satisfies $Pb(b_j = 0) = z$, where Pb(X) denotes the probability that the event X is true. Let us mention that we have chosen a fixed bucket size of 512 bytes. Because the performance of both schemes depends on the distribution of the records and on the capacity b of a bucket, we generated the following files:

(F1) b=10, d=2, both key components follow a uniform distribution
(F2) b=10, d=2, both key components follow a N(0.5,0.1)-distribution
(F3) b=10, d=2, both key components follow a Geo(0.3)-distribution
(F4) b=31, d=2, both key components follow a N(0.5,0.1)-distribution

For each of the files (F1) - (F4) we have generated the following five query files for comparing the 2-level grid file and PLOP-Hashing:

(RQ1) 20 quadratic range queries with volume 0.25
(RQ2) 20 quadratic range queries with volume 0.1
(RQ3) 20 quadratic range queries with volume 0.01
(PMQ1) 20 partial match queries where the first axis is unspecified
(PMQ2) 20 partial match queries where the second axis is unspecified

Here the volume of range query is the volume of the specified range divided by the volume of the key space. For these queries we have chosen the average number of disk accesses per query, where the average is computed over 20 queries. In the appendix, we report on the performance of both schemes in ten tables for the files (F1) - (F4) and for the queries (RQ1)-(RQ3),(PMQ1),(PMQ2). In the tables we use the abbreviation

avg for average, wc for worst case, 2LGF for the 2-level grid file and PLOP for PLOP-Hashing. We will give a short summary of the results.

Non-uniform distributions in combination with small bucket capacities (long records) typically occur in CAD-applications. This aspect is reflected in most of our experiments. Let us first consider the case of building up the file from empty. Concerning insertions, the average number of disk accesses in PLOP-Hashing is roughly 2 less than in the grid file, the worst case is up to 35% better than the grid file. Considering average search performance, PLOP-Hashing is between 54% and 66% of the grid file. Although the worst case can be 3 for PLOP-Hashing, it is usually 2 for large bucket capacities. Storage utilization of PLOP-Hashing is clearly above the grid file which directly influences partial match and range queries. In all experiments PLOP-Hashing is superior to the grid file. Let us additionally mention that the size of the binary trees does not exceed 4K bytes in all our experiments.

Now let us consider the case of a file completely built up where we only asked partial match and range queries. In all experiments PLOP-Hashing is superior to the grid file. For range queries PLOP-Hashing is around 10% better, for partial match queries the superiority is up to 50%. The large differences occur for partial match queries. This is due to the fact that the storage utilization of PLOP-Hashing is practically independent of the density of population, whereas the storage utilization of the grid file may be very low for densely populated areas.

7.Conclusion

The contribution of this paper can be summarized as follows.

1. A new multidimensional hashing scheme without directory, called PLOP-Hashing, is proposed. The principle of this scheme is that the file will be not expanded linearly, but in a piecewise linear fashion. By performing expansions in densely populated areas, PLOP-Hashing adapts to nonuniform distributions. In contrast to quantile hashing, the performance of PLOP-Hashing is independent of the sequence in which the records are inserted.
2. PLOP-Hashing is the winner in our comparison to the 2-level grid file. In particular average retrieval cost for complex queries, such as partial match queries, is considerably lower for PLOP-Hashing.

Future work considering PLOP-Hashing should deal with control functions and overflow strategies more efficient than those suggested in this paper. Additionally the problem should be considered how records following a nonuniform correlated distribution can be efficiently organized. As shown in [SK 87], variants of PLOP-Hashing are very suitable for the organization of spatial data. It is exactly in this area of spatial data where a lot of research remains to be done in the future.

References

[Bur 83] Burkhard, W.A.: Interpolation-based index maintenance', BIT 23, 274-294, 1983

[FNPS 79] Fagin, R., Nievergelt, J., Pippenger, N., Strong, H.R.: 'Extendible Hashing - a fast access method for dynamic files', ACM TODS 4,3,315-344, 1979

[Hin 85] Hinrichs, K.:'Implementation of the Grid File: Design Concepts and Experience', BIT 25, 569 - 592, 1985

[Kri 84] Kriegel, H.P.:'Performance comparison of index structures for multi-key retrieval', Proc. 1984 ACM/SIGMOD, 186-196

[KS 86] Kriegel, H.P., Seeger, B.:'Multidimensional order preserving linear hashing with partial expansions', Proc. Int. Conf. on Database Theory, 1986, 203-220, Lecture Notes in Computer Science 243

[KS 87] Kriegel, H.P., Seeger, B.:'Multidimensional dynamic quantile hashing is very efficient for nonuniform record distributions', Proc. Int. Conf. on Data Engineering, 10-17, 1987, an extended version will appear in Information Science

[Lar 80] Larson, P.Å.: 'Linear hashing with partial expansions', Proc. 6^{th} Int. Conf. on VLDB, 224-232, 1980

[Lar 85] Larson, P.Å.: 'Linear hashing with overflow handling by linear probing', ACM TODS,10,1,75 - 89, 1985

[Lit 80] Litwin, W.: ' Linear hashing: a new tool for file and table addressing', Proc. 6^{th} Int. Conf. on VLDB, 212-223, 1980

[NHS 84] Nievergelt, J. , Hinterberger, H., Sevcik, K.C.:'The grid file: an adaptable, symmetric multikey file structure', ACM TODS, 9, 1, 38-71, 1984

[NH 85] Nievergelt, J., Hinrichs, K.:'Storage and access structures for geometric data bases', Proc. Int. Conf. on Foundations of Data Organization, 335-345, 1985

[OM 84] Orenstein, J.A., Merrett, T.H.:'A class of data structures for associative searching', Proc 3^{rd} ACM SIGACT/SIGMOD Symp. on PODS, 1984

[Oto 84] Otoo, E.J.:'A mapping function for the directory of a multidimensional extendible hashing', Proc. 10^{th} Int. Conf. on VLDB, 491-506, 1984

[Oto 85] Otoo,E.J.:'A multidimensional digital hashing scheme for files with composite keys', Proc. ACM SIGMOD Int. Conf on Management of Data, 214-229, 1985

[Oto 86] Otoo, E.,J.:'Balanced multidimensional extendible hash tree', Proc 5^{th} ACM SIGACT/SIGMOD Symp. on PODS, 1986

[Ouk 85] Ouksel, M.:'The interpolation based grid file', Proc 4^{th} ACM SIGACT/SIGMOD Symp. on PODS, 1985

[Reg 85] Regnier, M.: 'Analysis of grid file algorithms', BIT 25, 335-357, 1985

[Rob 81] Robinson, J.T.:'The K-D-B-tree: a search structure for large multidimensional dynamic indexes', Proc. ACM SIGMOD Int. Conf on Management of Data, 10-18, 1981

[RS 84] Romamohanarao, W., Sacks-Davis, R.:'Recursive Linear Hashing', ACM TODS,9,3,369-391, 1984

[SO 82] Scheuermann, P., Ouksel, M.:'Multidimensional B-trees for associative searching in database systems', Information Systems Vol.7, No.2, 123-137, 1982

[SK 87] Seeger, B., Kriegel,H.P.:'Spatial access methods based on dynamic hashing', submitted for publication

[WK 85] Whang, K.-Y., Krishnamurthy, R.:'Multilevel grid files', draft report, IBM Research Lab., Yorktown Heights, 1985

Appendix

	2LGF	PLOP
avg insertion	4.55	2.68
wc insertion	12	8
avg successful search	2.0	1.04
avg unsuccessful search	2.0	1.21
wc search	2	3
avg storage utilization	60.8	64.8
directory blocks	104	0

Table 1: Building up (F1)

	2LGF	PLOP
avg insertion	4.51	2.46
wc insertion	12	9
avg successful search	2.0	1.06
avg unsuccessful search	2.0	1.27
wc search	2	3
avg storage utilization	60.4	62.3
directory blocks	101	0

Table 2: Building up (F2)

	2LGF	PLOP
avg insertion	4.85	3.46
wc insertion	16	10
avg successful search	2.0	1.07
avg unsuccessful search	2.0	1.32
wc search	2	3
avg storage utilization	52.5	59.5
directory blocks	151	0

Table 3: Building up (F3)

	2LGF	PLOP
avg insertion	4.10	2.45
wc insertion	12	7
avg successful search	2.0	1.04
avg unsuccessful search	2.0	1.27
wc search	2	2
avg storage utilization	59.2	63.2
directory blocks	34	0

Table 5: Building up (F4)

	RQ1	RQ2	RQ3	PMQ1	PMQ2
2-level grid file	1422.8	585.7	73.6	85.3	96.9
PLOP-Hashing	1264.7	522.9	63.5	71.5	71.6

Table 5: Queries to (F1)

	RQ1	RQ2	RQ3	PMQ1	PMQ2
2-level grid file	3646.3	964.1	71.5	99.2	111.9
PLOP-Hashing	3493.3	919.9	67.2	72.3	79.4

Table 6: Queries to (F2)

	RQ1	RQ2	RQ3	PMQ1	PMQ2
2-level grid file	1174.4	404.3	58.7	84.0	91.9
PLOP-Hashing	984.9	337.6	44.7	74.9	81.1

Table 7: Queries to (F3)

	RQ1	RQ2	RQ3	PMQ1	PMQ2
2-level grid file	1198.8	326.4	28.4	66.7	63.7
PLOP-Hashing	1122.3	309.1	24.9	35.1	50.7

Table 8: Queries to (F4)