The Multilevel Grid File - A Dynamic Hierarchical Multidimensional File Structure

Kyu-Young Whang
Center for Artificial Intelligence Research and Computer Science Department
Korea Advanced Institute of Science and Technology
P.O. Box 150 Cheong-Ryang Ni, Seoul, Korea

Ravi Krishnamurthy
Hewlett-Packard Laboratories
1501 Page Mill Road, Bldg. 3U
Palo Alto, CA 94304-1181

ABSTRACT : We present a new dynamic hashed file organization that solves most problems associated with the directory of the grid file proposed by Nievergelt et al. Our scheme is a multilevel extension of the grid file that supports multiattribute accesses to the file for exact-match, partial-match, and range queries as well as graceful expansion and shrinkage of the file in a dynamic environment. This new file organization offers a number of advantages over the grid file such as compact representation of the directory, linear growth of the directory in the number of records, easy splitting and merging of the directory, efficient processing of partial-match queries, and finer granularity of the directory. Moreover, it provides a new concept, called *abstract databases*, that can be employed in practical database applications.

1. Introduction

We present a new dynamic hashed file organization that allows accesses to the file using multiple attributes. We call it the *multilevel grid file (MLGF)*. The MLGF is an extension of the grid file proposed by Nievergelt et al. [Nie84]. This extension solves many drawbacks of the grid file caused by its multidimensional array directory.

Before proceeding, we define some terminology. A *file* is a collection of *records*, where a record consists of a list of *attributes*. A subset of the attributes, called *organizing attributes*, participates in organizing the file. For simplicity, we assume that the set of organizing attributes forms a key that uniquely determines a record. Let us note that we need this assumption only to follow a historical convention in defining terminology. Historically, in the literature, it has been implicitly assumed that an exact-match query (which we shall define later) retrieves a unique record by matching values for all organizing attributes. This assumption is not at all necessary otherwise. In general, the set of organizing attributes can be either a subset or a superset of a key. In fact, it may even be an arbitrary set of attributes. We define a file to have a *multidimensional organization* if there is more than one organizing attribute.

A *domain* of an attribute is a set of values from which a value for the attribute can be drawn. We define the *domain space* to be cartesian product of the domains of all organizing attributes. (The domain space has been called by various names in the literature: region [Rob81], record space, embedded space [Nie84], attribute space [Oto85a], base space [Fus85], to name a few.) We call any subset of the domain space a *region*.

A query is a predicate that must be satisfied by the records retrieved from the file. We define an *exact-match query* as a conjunct of equality predicates (such as Attribute A = 'a') for every organizing attribute. A *partial-match query* is similar to an exact-match query except that only a proper subset of organizing attributes is included in the predicate. A *range query* is a conjunct of equality predicates and at least one range predicate (such as Attribute A > 'a'). Finally, we define a *multiattribute access* or *retrieval* as an access to the file attempting to retrieve records matching values for more than one attribute.

The multiattribute retrieval problem has been studied extensively in the literature under the topic of *partial-match retrieval problems* [Riv76] [Bur76a] [Bur76b] [Aho79] [Bol79]. Many of these approaches employ hashing techniques for organizing the files. Although conventional hashing used as a file organization technique has the advantages of being simple and fast, many of these organizations do not provide the capability of dynamically adapting to the environments in which data volume grows or shrinks by large factors.

Recently, extensive progress has been made on the development of file organizations based on hashing that can dynamically vary the size of the file. Dynamic hashing by Larson [Lar78], virtual hashing by Litwin [Lit78], linear (virtual) hashing by Litwin [Lit80] [Lit 79], and extendible hashing by Fagin et al. [Fag79] are pioneering research contributions in this direction. Many variations of these schemes are also available in [Lit81], [Lar80], [Lar82], [Sch81], and [Mul81]. A hybrid approach between the conventional B-tree [Bay72] and the extendible hashing method can be found in [Lom81].

All these techniques, however, emphasize only single-attribute cases. These techniques must be extended for the design of multidimensional file organizations. Extensions of linear hashing have been reported in [Bur83], [Ouk83], and [Oto85a]. Similarly, the ideas behind the extendible hashing method have greatly affected the design of the grid file [Nie84]. Dynamic multipaging [Mer82] based on extendible arrays has been the major motivation for the multidimensional extendible hashing directory [Oto84] as well as the multidimensional linear hashing scheme [Oto85a]. Just for comparison purposes, we note that multidimensional extensions of the conventional B-tree have been reported in [Rob81] and [Sch82]. A static multidimensional directory that induces a domain partitioning similar to that of [Rob81] has also been reported in [Lio77].

The grid file is an extension of dynamic hashed organizations incorporating multiattribute accesses using a multidimensional file organization. The grid file consist of two components: the directory and the main data in the file. The directory partitions the domain space into grid-like regions and associates for each region a pointer to the physical block that contains the records that belong to the region. A one-dimensional array

DATABASE SYSTEMS FOR ADVANCED APPLICATIONS '91
Ed. A. Makinouchi

for each attribute, called the linear scale, is also associated with the directory. These linear scales define the intervals of the partitions of individual domains.

The idea behind the design of the grid file is conceptually very simple; but, the implementation may pose many difficulties because of the envisionment of the directory as a large multidimensional array. Although, the designers of the grid file left the representation of the directory open, their two-disk-access guarantee for exact-match queries requires an array-like structure. The purpose of this paper is to propose a novel mechanism that solves these problems. We first identify potential drawbacks of the grid file and then present our scheme that overcomes these drawbacks. Further, we discuss an additional advantage that our scheme provides, when applied to database environments, but that has not been considered seriously in conventional database systems.

In the following we describe the overall organization of this paper. In Section 2 we discuss the properties of the grid file and present motivations of our work. In Sections 3 and 4 we describe the architecture of the MLGF: we first introduce the single-attribute case in Section 3, and subsequently, extend it to the multidimensional case in Section 4. We subsequently present algorithms for querying and manipulating the MLGF in Section 5. In Section 6 we discuss the advantages of our scheme as well as enhancements achieved compared with the grid file. We reinforce the result of this discussion by a quantitive analysis in Section 7. Finally, Section 8 summarizes the contributions of this paper and proposes future research.

2. Motivations

In this section we discuss drawbacks of the grid file and present motivations for this paper.

2.1. Compactness of directory

A major problem in the grid file is the size of the directory. Simulation results presented in [Nie84] indicate that the size of the directory grows approximately linearly in the size of the file. We note, however, that the coefficient of the linear function grows as the data distribution deviates from a uniform one. Furthermore, If a strong correlation (some value relationship between x and y values) among attributes exists, the growth of the directory is of a much higher order - a high polynomial order in the size of the file; and an exponential order in the number of organizing attributes (i.e., the *dimensionality* of the directory). Incidentally, the authors of the grid file addressed both attribute correlation and nonuniformity in the record distribution. We believe, however, the attribute correlation has much more serious effect on the asymptotic growth of the directory.

In Sections 3 and 4 we shall present a solution to this problem. Here, we make an assumption that will be used throughout the paper. Since we are dealing with problems associated with large directories, we assume that the directory does not fit in main memory and must reside in the secondary storage.

2.2. Splitting of directory

Another difficulty associated with the grid file is the one encountered when the directory splits. Because the directory is a multidimensional array, splitting it necessarily accompanies copying the entire directory. Therefore, a directory split would cause a large amount of I/O accesses. Here, we conclude that we need an organization for the directory that gracefully adapts to dynamic changes. Our scheme presented in Sections 3 and 4 also solves this problem.

2.3. Merging of directory and data blocks

In the grid file merging occurs in two stages: merging of data blocks and merging of cross sections (*hyperplanes*) of the directory. In this subsection, we discuss performance issues associated with these two stages of merging. For merging data blocks, two algorithms–namely, the buddy system and neighbor system–were proposed in [Nie84]. Since the grid directory has a plain array structure, however, it does not provide a convenient data structure to recognize a buddy or a neighbor that satisfies the convexity property. The *convexity property* requires that a region be a k-dimensional "rectangle". Hence, algorithms for detecting a buddy or a mergeable neighbor must be executed each time a block is merged. Frequent execution of these algorithms will be very costly, especially when the directory is stored in the disk.

The next stage, directory merging, was hardly investigated in [Nie84] under the assumption that, in practical environments, the merged hyperplane will split again sooner or later, rendering the merger unnecessary. We believe that this assumption is incorrect because, as the distribution of records changes, the hyperplane to be split will not necessarily be the same as the one just merged. If the hyperplanes to be merged were left unmerged, we would end up with a large number of redundant directory entries that would contribute to creating an oversized directory.

There are two problems associated with merging hyperplanes. First, as in splitting, merging the directory necessitates copying the entire directory. Second, to merge two hyperplanes, we must first determine whether they are "mergeable". To be mergeable, each directory entry in one hyperplane must point to the same block as the corresponding entry in the other hyperplane. Detecting this condition is costly because a large number of directory entries have to be searched.

The problems discussed above have a serious effect on performance when the directory is stored in secondary storage. Because the directory is a multidimensional array, (assuming the dimensionality is greater than one) logically contiguous directory elements are not necessarily stored in contiguous physical blocks–let alone in the same physical block. As a result, searching the directory is very likely to incur one block access per one directory entry. Therefore, we expect that the operations associated with merging the directory as well as merging the data blocks will cause a large amount of I/O accesses. Our scheme presented in Section 3 and 4 solves these problems by keeping a convenient representation of buddies and merging the directory only "locally", while maintaining a conceptual grid-like partitioning of the domain space.

2.4. Partial-match queries

The grid file guarantees a bound of two disk accesses for exact-match queries. Nevertheless, performance may degrade severely for partial-match queries for the following reasons. First, to process a partial-match query, an entire hyperplane of the directory orthogonal to the domains of the attributes specified in the query has to be examined. As discussed in Section 2.3, searching the directory of a grid file is likely to incur one block access per one directory entry. Consequently, a large number of I/O accesses are expected in processing such a query. To illustrate, consider a three-dimensional grid directory consisting of 1000 elements having equal sized partitions in all axes; then, since there must be ten hyperplanes in an axis, a query matching one attribute will cause 1000/10 = 100 I/O accesses. Furthermore, the situation gets worse if the dimensionality gets larger. For example, for a six-dimensional directory consisting of the same number of elements, the same query will cost $1000/1000^{1/6} = 315$ I/O accesses.

Although the above example describes a somewhat pessimistic case, we expect the effect of excessive directory accesses could be significant even in practical situations. In our scheme described in Sections 3 and 4, we alleviate these problems by keeping the directory compact (i.e., by not storing redundant directory entries), by keeping the directory entries that are close together in the domain space in the same physical block as much as possible.

3. Single-Attribute Case

In this section we use Extendible Hashing [Fag79] as the single attribute analogy of the grid file and present our extension to the multilevel counterpart of extendible hashing. In this way, the motivation, concept, and its ramifications are easier to understand. In the next section we shall repeat this process in the context of the grid file. Here we assume that the organizing attribute consists of exactly one attribute. Even though extendible hashing can allow a composite of attributes, we omit this possibility for the sake of simplicity, and without loss of generality.

3.1. Extendible Hashing

Extendible Hashing (EH) is an access technique that can be used to find all the records that have a particular value for the organizing attribute. This value is termed as *key-value*. The reader should not confuse this with the concept of keys in the traditional framework of functional dependencies. EH consists of two levels: the directory (i.e., the root) and the leaves. The leaves contain the records. Typically, each leaf can be viewed as a block in the secondary storage.

The directory is used to locate the block (i.e., leaf) containing the records that have a particular key-value. Succinctly, the directory is an array of pointers to leaf blocks. There are 2^d entries in the directory, where d is called the *depth* of the directory–a parameter associated with the directory. Also, associated with a directory is a hashing function H that maps the values from their original (possibly clustered) domain to an integer in $\{0,1,2,.....,2^e\}$, where e is an integer at least as large as d. The hashing function H partitions any set of records into 2^d partitions, based on the first d bits of the hashed value. Therefore, the i^{th} directory entry consists of a pointer to a block, p, that contains records associated with all the key-values that hash into the i^{th} partition.

The (array-like) indexing capability into the directory is an important property that is used to guarantee that any search requires no more than two accesses. But guaranteeing this capability requires that the updates to the directory be performed without destroying this property. In EH the updates that require extending the directory are done by doubling the entries. Thus the number of entries in a directory is always some power of 2.

In summary, EH provides an efficient access technique, in which the user is guaranteed no more than two block accesses to locate the data. However, there are some problems that may seriously degrade the performance of this method: they are the problems we discussed in Section 2 that are applied to the single-attribute case. Although these problems are less serious in this case, we address them here to motivate a solution that we intend to generalize in Section 4.

1. There are 2^d entries in a directory; i.e., the size of the directory is determined by the parameter d alone. Ideally, one would prefer that the number of entries, 2^d, is no more than the number of leaf blocks. In EH, however, the size of the directory can be disproportionately large when compared to the number of leaf blocks.
2. Extending the directory is an expensive operation especially when the directory is large. Directory extension involves doubling the number of directory entries, including those entries that did not overflow. As the block that overflows is the only block that is split, it would be better if we alter only the corresponding entry in the directory.

3.2. Multilevel EH method

In this subsection we propose a multilevel generalization of the EH method, *multilevel extendible hashing(MLEH)*, that solves the above two problems. Even though the following solution is quite similar to the dynamic hashing method presented by Larson [Lar78], we approach our solution as a generalization of the EH method, because we intend to repeat this process of generalization, in the next section, for the multiattribute case.

Let us consider an example of EH organization given in Figure 1. The content of the directory can be summarized by the following set of records (i.e., tuples):

D_1 = { (00, Block A), (010, Block B), (011, Block C), (1, Block D) }

Abstractly, D_1 consists of a set of records (x,y) such that x represents a *region* in the domain space and y is a pointer to the block where the records in that region are stored. Intuitively, if x = 010, then it represents a region all of whose hash values start with 010.

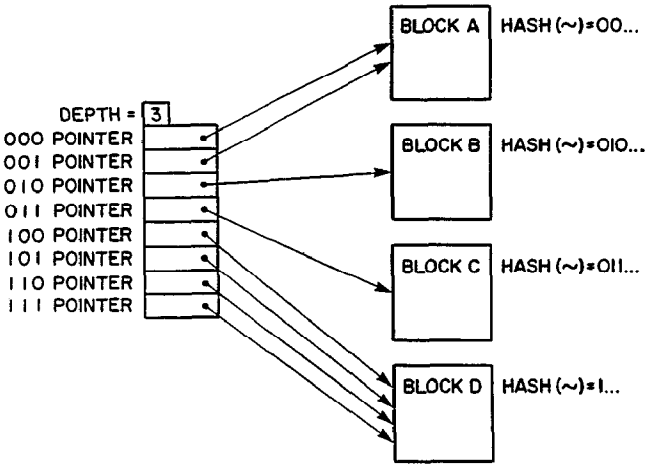

Figure 1. An EH example.

The main difference between this set representation and that of the EH method is that, in EH, the address of the i^{th} entry (i.e., the entry corresponding to the i^{th} partition) is computed, whereas the i^{th} entry must be searched in the set representation. This searching involves two problems:

1. Given an entry number, say i, how to match (i.e., identify) a record for that entry.
2. How to find that matched record efficiently.

The match for the i^{th} entry can be defined as the record (x,y) such that x is a prefix of the binary representation of y. The process of finding the match is termed *prefix matching*.

In order to address the second problem, we observe that D_1 is itself a "file". Therefore, it needs to be organized in some fashion. Thus, the problem of finding a record in D_1 is quite similar to that of looking for a record in the original file (say D_0). Based on this observation, we propose to organize the set D_1 using the extendible hashing method. Here, the first attribute is used as the organizing attribute and the identity function is used as the hashing function. Such an organization is shown in Figure 2, in which the directory for D_1 is called D_2. Finally, the directory for D_1 (i.e., D_2) can also be organized using the multilevel EH method, creating D_3 as its directory.

In short, D_3 is the directory for the file D_2; D_2 is the directory for the file D_1; and D_1 is the directory for the original file D_0. This hierarchy of abstractions provides a multilevel approach to organizing the directory. Note that, in each directory D_i, the regions represented by the set of x-values are mutually exclusive.

This proposed modification solves both problems associated with the EH method. First, the number of entries in the directory is usually much less than in the EH method. This is evident from the following observations. Any block in the original file (i.e., D_0), is pointed to by

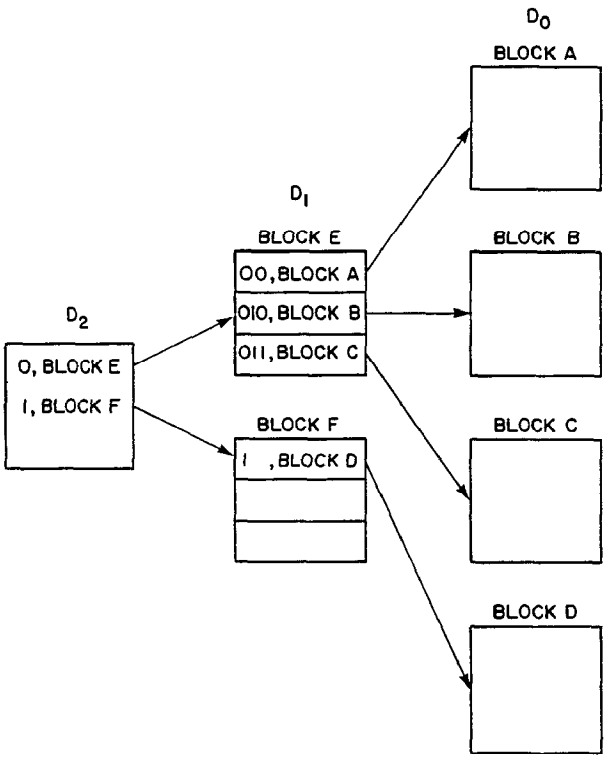

Figure 2. An MLEH example.

exactly one entry in the directory and an empty region is not represented in the directory. Thus, the number of entries in each directory D_i is no more than the number of blocks in the associated file D_{i-1}.

Second, in the event a block overflows, the process of updating the directory is quite simple. It consists of

1. modifying the old record corresponding to the overflowed block,
2. adding a new record to represent the newly created block due to the split operation, and
3. propagating this effect up the hierarchy, if necessary.

As compared to the overhead incurred due to the doubling of the directory in EH, the above process is very efficient.

4. Multiattribute Case

In this section we describe the grid file technique as a k-dimensional extension of EH and then generalize the technique to the *multilevel grid file (MLGF).*

4.1. Grid File

In EH, the domain space can be represented as values in a single dimension. Consequently, each region in this domain space is also one dimensional. In contrast, the grid file views the domain space as a k-dimensional space when there are k organizing attributes in the record. Thus, a region in the grid file is also a k-dimensional space.

As in EH, the grid file also consists of two levels: the directory (i.e., the root) and the leaves (i.e., data blocks). Typically, data blocks are stored in the secondary storage and contain the records in the file. Each data block represents a region in the domain space and has all the records that lie within the region.

The directory is viewed as a k-dimensional array. Each element in this array represents a region in the domain space, and has a pointer to the corresponding data block. As in EH, more than one entry in the directory may point to the same data block. The size of the directory is determined by a linear scale for each dimension. A linear scale $LS_j[0..m_j]$ is a one-dimensional array of values for the j^{th} attribute such that $LS_j[i] < LS_j[i+1]$ for $0 \le i \le m_j - 1$ and $1 \le j \le k$. Thus, a linear scale partitions the corresponding domain, and a value v in the domain is said to occur in the i^{th} partition, if $LS_j[i] \le v < LS_j[i+1]$. Intuitively, the LS_j can be viewed as an order-preserving hashing function that maps the values in the domain for attribute j to an integer from 0 to $m_j - 1$. If the number of partitions in each dimension are the same (i.e. $m_j = m_{j+1}$ for all j), the size of the directory is m^k.

An exact-match query search algorithm consists of the following. First, compute the partition for the value for each dimension. Then, look up the directory for the pointer to the data block, which may require one access. The second access is required for accessing the data block.

4.2. multilevel Grid File

As we did in the generalization of EH, we propose to create a hierarchy of abstractions to represent the directory. The content of the directory of the grid file can also be summarized by the following set notation for the directory.

$D_1 = \{(x\ y) \mid x$ represents a region, and y is a pointer to the data block $\}$.

The region is a k-dimensional space that can be represented by a k-tuple where the i^{th} element of the k-tuple is the partition number on the i^{th} dimension. y is a pointer to the data block as before.

In a way similar to the technique used for MLEH, we can organize D_1 by creating a directory D_2 for the file D_1. We call this organization the *multilevel grid file (MLGF)*. For example, a three-level MLGF can be described as follows: D_3 is the directory for the file D_2; D_2 is the directory for the file D_1; D_1 is the directory for the original file D_0. Once again, this hierarchy of abstractions provides a multilevel approach to organizing the directory.

Besides the multilevel property, there is one other major deviation from the grid file approach. The partition number is computed using a hashing function (possibly order-preserving) rather than a linear scale. This enables the MLGF method to represent a partition using the prefix of the hash value in the same way it is done in EH and MLEH. The advantages of this approach are evident in the splitting and merging operations. This will be elaborated in Section 6. Before we discuss the pros and cons of this technique, let us describe an example.

Example 4.1: Figure 3 depicts an MLGF representation of a two-attribute file and its two-dimensional (logical) grid, where the physical blocks are shown by dotted inner rectangles. In this example there are twelve blocks. Accordingly, there are twelve entries in the directory D_1, as shown in Figure 3. We have assumed, in this example, a blocking factor of (maximum of) five entries per block. Even though there are only twelve entries in D_1 there are four physical blocks associated with it. The directory of D_2 has four entries, all contained in one physical block. An entry (10,0) in D_2 represents that all records whose hash values of the first attribute start with 10, and whose hash values of the second attribute start with 0. Note that the block in D_1, which the entry (10,0) in D_2 points to, has the region (i.e. (10,0)) subdivided into the following regions: (100,00), (100,01), and (101,0).

From this example, let us note the following points:

1. The empty blocks (e.g. (00,1)) are not represented anywhere in this structure.
2. Each physical block can represent a region of varying size; nevertheless, only one entry in the directory corresponds to that block.
3. The fact that the domain of a particular dimension is completely

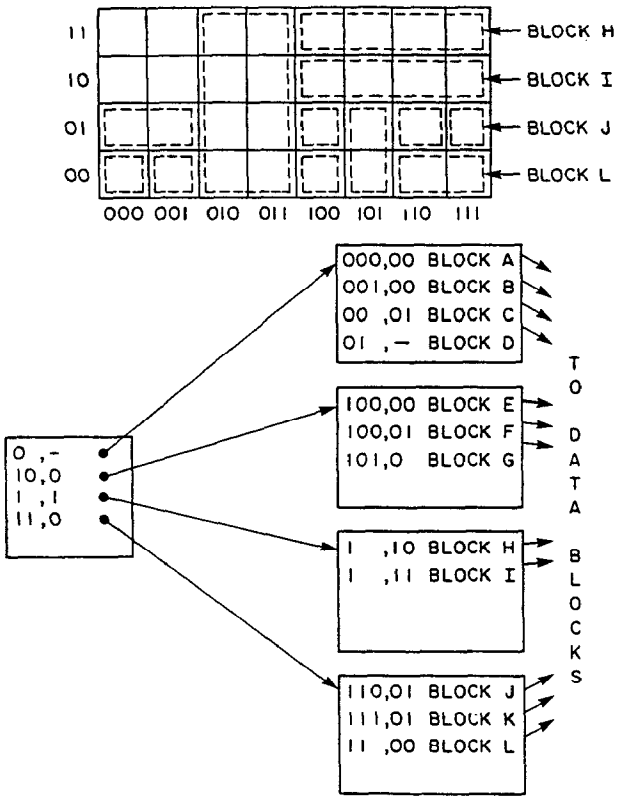

Figure 3. An MLGF and its space partition.

represented is denoted by the symbol "-" in the figure. That is, all hash values starting with both 1 and 0 are represented by this symbol.

5. Algorithms

In this section we describe the three operations–query, insertion, and deletion–for manipulating a multilevel grid file. We define some parameters that are used as program variables in describing the algorithms: 1) *Root* specifies the pointer to the root level of the MLGF directory, 2) *Key Record* is the record of key values to be inserted or deleted, and 3) *Hash Record* is the corresponding record of hashed values. Further, the directory is viewed as a set of (*Region Vector, Ptr*) pairs, where *Region Vector* is the record of hash values representing the region, and *Ptr* points to the physical block corresponding to the region.

5.1. Query

Following the approach taken by Robinson [Rob81], we define a query region as follows. "A query can be expressed by specifying a region, the *query region*. It is convenient to think of a region as a cross product of intervals."[Rob81]. In this representation, an exact-match query is specified by the intervals representing single points; a partial-match query is represented by specifying full domains for some of the intervals; and finally, a range query can be specified by giving intervals that are not full domains.

The algorithm to output all the records satisfying a query is as follows:

1. Terminate if *Root* is NULL; otherwise *PagePtr* := *Root*.
2. Read the block pointed by *PagePtr*.
3. IF the block read is a leaf block
   a. THEN for each record in the block that satisfies the query, (i.e., the record is in the query region), output the record.
   b. ELSE for each element (*Region Vector, Ptr*) in the block do the following: if the intersection of the region specified by the *Region Vector* and the query region is nonempty, then recurse from Step 2 using *Ptr* as *PagePtr*.

Note that, in our scheme, the problem of checking intersection in Step 3b simply reduces to prefix matching, and is quite efficient.

5.2. Deletion

In this subsection we limit our discussion to the deletion of a single record. Generalizing this algorithm for the cases of more than one record and of query-dependent deletion can be done in an obvious manner.

We define additional terminology. A *parent directory entry* of a block is the directory entry whose *Ptr* points to the block. The block containing the parent directory entry is the *parent block*. In general, a block is associated with a region vector representing the region of the block. The region vector of a specific block is called the *original region vector*, and accordingly, the region vector of its buddy the *buddy region vector*. Lastly, the parameter *Current Block* stands for the block that is considered at a specific moment.

Deleting a record consists of searching for the record to be deleted and deleting it subsequently from the appropriate block. The algorithm is as follows.

1. Terminate if *Root* = Null;
2. Do an exact-match query on *Key Record* to be deleted, thereby finding the block from which *Key Record* is to be deleted. Set this block to be *Current Block*. If *Key Record* does not exist, then signal an error and terminate.
3. Delete *Key Record* from *Current Block*; If the block becomes empty, free *Current Block* and iterate Step 3 by setting *Key Record* to be the parent directory entry and *Current Block* to be the parent block.
4. Invoke the function MERGE-DOMAIN for *Current Block*. This function finds a domain on which to merge and a mergeable buddy in the same parent block where its parent directory entry is, if they exist.
5. IF a mergeable buddy was found THEN do
   a. If the buddy region is nonempty, do the following: merge *Current Block* and the buddy block to form a merged block; free the buddy block and delete the parent directory entry of the buddy block; update the *Ptr* in the parent directory entry of *Current Block* to point to the merged block.
   b. Replace the original region vector in the parent directory entry by the merged region vector.
   c. Set *Current Block* to be the merged block.
   d. Iterate Steps 4 and 5 for (new) *Current Block*.
6. ELSE
   a. IF *Current Block* is the result of a merge operation THEN Recurse from Step 4 with the parent block as *Current Block*.
   b. ELSE terminate.

Here, the function MERGEABLE is left unstated. Any solution must do the following: find another directory entry in the same parent block whose components of the region vector for all but one domain are identical to those of the original region vector. For the domain on which the components differ, the difference is limited to the last bit. There may be many choices; here we assume any one is chosen based on some criteria (e.g., the total number of tuples in the blocks to be merged is less than a certain threshold). Also, note that an empty buddy region is mergeable. An empty buddy region must be properly contained in the region of the parent block without a corresponding directory entry in the parent block.

5.3. *Insertion*

In this subsection we limit our discussion to insertion of a single record. Generalizing this algorithm for the cases of more than one record and of query-dependent insertion can be done in an obvious manner. Insertion of a record consists of searching for the region that the record belongs to and adding the record in the appropriate block. The algorithm is as follows.

1. IF *Root* = NULL THEN
   Create a root block with a directory entry (*Region Vector*, *Ptr*), where each component of *Region Vector* is "-", and *Ptr* points to a data block (also created) containing *Key Record* to be inserted.
2. ELSE
   Do an exact-match query on *Key Record* to be inserted, thereby finding the region where *Key Record* is to be added. This region may or may not correspond to a data block because empty regions are not represented in the MLGF.
3. Adding *Key Record*:
   a. IF the data block exists THEN add *Key Record* to that block. If *Key Record* already exists, signal an error and terminate.
   b. ELSE do
      1) Add a directory entry that points to a newly created block at the next level (say, *B*) and whose region vector is *Hash Record*.
      2) From the next level add a directory block for each nonleaf level of the MLGF containing one directory entry that points to the newly created block in the next level and whose region vector is identical to that of its parent directory entry.
      3) At the leaf level create a data block containing one record (i.e., *Key Record*).
      4) Apply merging algorithm as given in the previous section for this newly created block *B* until no more merging occurs. This step finds a region containing the largest empty region that can be obtained.
4. SPLITTING: IF SPLIT_HEURISTIC is true THEN do
   a. Choose a domain to be split.
   b. Construct two new region vectors by appending one bit (0 and 1 respectively) to the component hash value (for the chosen domain) of *Region Vector* representing the splitting block.
   c. Partition the splitting block based on these two new region vectors; thus creating two new blocks and free the old block.
   d. Replace the directory entry for the splitting block by the two new directory entries for the constructed region vectors with corresponding pointers to the appropriate (newly created) blocks. Repeat 4 for the parent block to accomodate overflow.

Like in the deletion case, the function SPLIT-HEURISTICS is left unstated. It should be used to decide when and which domain to split based on some criteria. Nevertheless, it should be noted that there is an important aspect in choosing a splitting domain. When a directory block splits, a domain cannot be chosen if the set of directory entries in the splitting block does not partition in the domain into two disjoint nonempty sets. In particular, if a splitting block contains a directory entry having a component hash value for a domain that is equal to the common prefix of the block (i.e., the component hash value for the domain of the parent directory entry), then a partition cannot be obtained for that domain. This happens because the region represented by this directory entry intersects both subregions divided, and thus, the directory entry must appear in both subsets, rendering them nondisjoint.

6. Discussions

In this section we discuss advantages and disadvantages of the MLGF as compared with the grid file and other conventional file organizations. In Sections 6.1, 6.2, 6.3, and 6.4, we begin by discussing enhancements of the MLGF over the grid file on the issues proposed in Section 2. We then present, in Section 6.5, additional advantages of the MLGF that have not been addressed in the design of the grid file. In Section 6.6 we compare features of the MLGF with those of the K-D-B-tree. Finally, in Section 6.7, we discuss other miscellaneous issues.

6.1. *Compactness of directory*

In the MLGF we keep only those directory entries that represent nonempty regions. As a result, the number of directory entries is bounded by (i.e., always less than or equal to) twice the number of records regardless of the record distribution and correlations among attributes. (An exceptional case is when there are chains consisting of directory blocks at successive levels that contain only one directory entry. We believe this occurs very rarely, since such a chain is created only when a new record is inserted into an empty region existing at a nonleaf level.) Therefore, the asymptotic growth of the directory must be *linearly* dependent on the growth of data.

6.2. *Splitting of directory*

Splitting the directory is simple in the MLGF. When a block overflows, only the directory entry pointing to this block is affected (and, recursively, so is the directory entry in the next higher level, if the directory block overflows). A directory entry is split into two, simply by adding one bit (0 or 1, respectively) at the end of the component hash value of the region vector for the splitting domain. Therefore, splitting the directory is a local operation affecting only one or a few directory entries. This aspect contrasts with splitting the directory of a grid file, when an entire hyperplane is created, and the entire directory copied. Incidentally, directory split occurs in the grid file much less frequently than in the MLGF because a split of a hyperplane creates a number of directory entries that can absorb many of future splits of data blocks. Nevertheless, the cost of one split is exorbitant due to the cost of copying the entire directory. Clearly, the MLGF is not only more efficient in cost but also distributes the cost more evenly over all splits of data blocks.

6.3. *Merging of directory and data blocks*

In the MLGF merging the data blocks is made easy by maintaining a convenient representation of buddies in the directory. For example, the buddy of a directory entry represented by 01001 for the merging domain must be the one represented by 01000 for the same domain with identical representations for the other domains. The existence of this directory entry can be easily determined by accessing the directory hierarchy; in case it exists, the data block to be merged is found by following the pointer. The technique just described contrasts with that of the grid file, in which an algorithm for finding buddy from a flat array-structured directory has to be run every time a block is merged, thereby causing a large amount of I/O accesses.

Merging the directory entries is easier in the MLGF than in the grid file because it occurs only "locally" as in the case of splitting. With the MLGF, merging is carried out by simply taking one bit off the original bit strings representing the directory entries and creating a new entry pointing to the newly merged block. As in the case of splitting, the local merging operation may be recursively propagated to the next higher level, if the occupancy of the directory block falls below a certain threshold. Our scheme contrasts with that of the grid file, in which an entire hyperplane is merged, and further, an algorithm for detecting an "mergeable" hyperplane has to be run dynamically every time a block is merged, thereby causing a large number of I/O accesses. Note that, unlike in the case of splitting, the possibility of merging a hyperplane must be checked for every block merger in the grid file. This aspect renders merging in the grid file even costlier.

6.4. *Partial-match queries*

We observed in Section 2.4 that partial-match queries are potentially very costly due to searching a large portion of the directory, i.e., a hyperplane. This problem is greatly alleviated in our scheme due to compactness of the directory: since the number of directory entries is less than that of the grid file, searching a hyperplane must be less costly. Yet

another important enhancement comes from the multilevel architecture of the directory. Since a directory entry representing a region points to a physical block containing lower level directory entries subdividing the region, those lower level directory entries, which are close to one another in the domain space, tend to be in the same physical block. Therefore, searching a hyperplane must be less costly because more than one directory entry can be searched in one block access. This contrasts with the case of the grid file, in which one block access is needed per one directory entry. The following example should clarify this point.

Example 6.1: Consider a two-level, two-dimensional directory stored in physical blocks with a capacity of four (i.e., the directory blocking factor = 4). Suppose the top-level directory consists of four entries. Assuming a uniform distribution of records, we shall have the directory in Figure 4. Here, dotted inner squares represent physical blocks.

From the figure we note that, even though a hyperplane contains four directory entries, searching the hyperplane will cause only two block accesses. In general, under the assumption of uniform distribution of records and cyclic splitting of individual domains, the number of block accesses for searching a hyperplane would be $D^{1-1/d}/b^{1/d}$ (derived in section 7), where D is the number of entries in the directory, d the dimensionality, and b the directory blocking factor.

Let us note that the problems associated with partial-match queries addressed here are *inherent* in any multidimensional organizations. These problems can be better visualized by considering a case in which the file is accessed through a key attribute that uniformly determines a record; even in this case, if there is more than one organizing attribute, an entire hyperplane of the directory, and accordingly, its corresponding data blocks, have to be accessed.

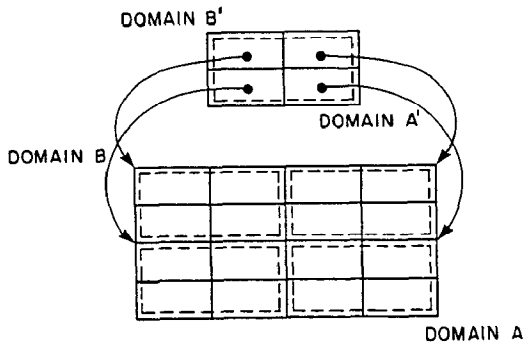

Figure 4. A two-level, two-dimensional directory with the directory blocking factor of four.

6.5. Abstract Database

An important advantage of the MLGF comes from the availability of abstract databases, against which a preevaluation of a query can be performed. We define an *abstract database*, DB-A, of a database, DB-B, as a database derived from DB-B in such a way that the result of a query obtained from DB-A is a superset of the result obtained from DB-B.

Each level of the directory in our scheme can be considered a k+1-attribute file: each entry in a directory consists of values for k attributes of the file and a pointer to a block of the next lower level directory. Thus, queries can be processed against the directory only, provided that the attribute values specified in the queries are transformed into proper representations (by a hash transformation, in this case). The result of processing a query against the directory must be a superset of the result that would be obtained from the data file, provided that the result is transformed into proper representations (through a pointer, in this case). Therefore, we conclude that the lowest level of the MLGF directory forms an abstract database of the data file. In the same manner, each level of the directory is an abstract database of the next lower level.

The concept of abstract databases–first introduced in [Kri84] as bucket databases–has a profound effect on performance, especially when only a limited amount of main memory is available. Example 6.2 illustrates this point.

Example 6.2: Suppose that the size of the database considered is 10 Mbytes and the size of an abstract database is 100 Kbytes. We assume further that 500 Kbytes of main memory is available for processing queries. Since sufficient main memory is available for the abstract database, it can be kept in main memory. We process a query in two steps. First, we *roughly* process (i.e., preprocess) the query against the memory-resident abstract database to obtain the superset of the result. During this process, relevant records from the original database are brought in main memory. The set of these records is called a *reduced database* [Ber81] for this query. Let us assume that the size of the reduced database is 10 Kbytes. We subsequently process the query once again against the reduced database, which we expect to fit in main memory. In either step, since all the data reside in main memory, the query can be evaluated with little I/O's. In fact, I/O accesses are needed only to bring the reduced database in main memory. Thus, the more reduction we achieve, the better the performance should be. The reductive power of preprocessing a query using an abstract database has been investigated in detail in [Kri84]. Let us note that, in the MLGF, we can use any level of the directory as the abstract database.

6.7. Related research

Perhaps, the closest to the MLGF is the K-D-B-tree developed by Robinson [Rob81]. The K-D-B-tree comprises two types of blocks: region blocks and point blocks. Region blocks contain a collection of *(region, blockId)* pairs, where *blockId* is a pointer to the block in the next lower level of the tree. Point blocks contain a collection of *(point, location)* pairs, where *location* gives the location of a data record. Nevertheless, there are a number of differences between the two schemes.

1. The K-D-B-tree splits a region according to the even-record-distribution strategy. Thus, when a block splits, half the records are moved to one block, while the other half to another. Accordingly, the region is split based on the boundary value of these two halves. On the other hand, in the MLGF, regions always split based on a predetermined grid-like boundary.
2. The even-record-distribution splitting strategy does not induce grid-like partitions of the domain space. Instead, it produces an irregular partitioning. Thus, merging is not easy because a region almost always has to merge with the one it originally split from. In contrast, in the MLGF, a region has multiple alternatives when merging–one for each axis in the domain space. Since these alternatives provide additional freedom in merging, we also expect a better storage utilization in the MLGF.
3. The K-D-B-tree does not address the problem of representing multidimensional regions inside a physical block. It is clear, however, that original attribute values must be stored as discriminator values. It should be pointed out that finding an efficient representation of a multidimensional region using variable discriminator values may not be trivial. (We believe the representation must be a tree-like structure.) In comparison, the MLGF directory stores in a block a bit encoding of the trie representing further partitioning of the region that the block belongs to.
4. Since the K-D-B-tree stores original attributes values, the leaf level of the tree is almost equivalent in size to the data file, in case all the attributes are considered in the K-D-B-tree. In contrast, by storing hashed bit strings, the MLGF directory provides a compact representation. At the same time, the number of levels in the MLGF directory must be minimal because of compact representation of index entries and a large index blocking factor.

5. The K-D-B-tree distinguishes the leaf level blocks (point blocks) from the other blocks (region blocks). In contrast, the MLGF employs a consistent representation in all the directory levels. This strategy helps keep the file querying manipulation algorithms simple and clean.

The prefix B-tree [Bay77] bears some resemblance in concept to the one-dimensional version of the MLGF, i.e., MLEH. It differs from the MLEH in two important ways:

1. It partitions the domain space along arbitrary boundaries.
2. It represents a region implicitly by using a set of discriminator values. In contrast, the MLEH represents a region explicitly by using a unique bit string.

Let us note that these differences would cause a multidimensional extension of the prefix B-tree to encounter with problems similar to those of the K-D-B-tree.

The hierarchical multidimensional extendible hashing [Oto85b] also shares a common concept with the MLGF and has a linear growth of the directory, but it uses an array for the directory structure inside a physical block as extendible hashing does for the entire directory. In comparison, the MLGF uses a simple list of directory tuples for efficient storage and easy manipulation.

6.8. *Other issues*

Range queries can be handled efficiently in the MLGF, provided the hashing function is order preserving. As indicated in [Fag79], order-preserving hashing functions have rarely been used in practice because they do not provide sufficiently uniform distribution over the address space. In the MLGF, however, skewing of the hashing function is not much of a concern because we store only those directory entries representing a nonempty region. After all, there should be no serious waste in time or storage cost due to a nonuniform distribution.

In [Nie84], it has been shown that the buddy system, when used for merging, gives a lower utilization of main storage space than the neighbour system. This happens mainly because an equal-sized buddy is frequently unavailable for merging, while a mergeable neighbour is. By definition, a buddy is a neighbour, but not *vice versa*. On the contrary, the buddy system proves to be a more systematic and flexible in merging the directory. In the neighbour system, the directory can hardly be shrunk because of irregular partition of the domain space. In conclusion, we believe that the buddy system is better than the neighbour system in a highly dynamic environment. In the MLGF the buddy system is inherently imbedded in the design.

7. Performance Analysis

In this section we present a simple performance analysis for the MLGF and the grid file. The analysis is intended to quantify our informal discussion on the properties of the two file organizations, but is not intended to provide accurate estimation of the costs involved in such organizations. Primarily, we present a worst case analysis; later, we try to relax the worst case condition to investigate its effects. Throughout this section, we assume an equal number of partitioning of the domain space in all domains. Before proceeding, we define some notation:

- n: number of records in a file
- d: number of organizing attributes (dimensionality)
- b: number of directory entires in a physical block (directory blocking factor)
- D_G: size of the directory in a grid file
- D_M: size of the lowest level of the directory in an MLGF

7.1. *Directory size*

Grid File: In the worst case, the asymptotic growth of the directory is $O(n^d)$.

MLGF: The number of directory entries in the lowest level is bounded by the number of blocks in the data file, and consequently, by the number of records in the file. Thus, the asymptotic growth of the directory will be $O(n)$.

7.2 . *Hyperplane search*

Grid File: We note that the number of hyperplanes in a domain is $D_G^{1/d}$. Hence, the number of directory entries contained in a hyperplane is $D_G^{1-1/d}$. Since the directory is organized as a multidimensional array, in all domains except one, a hyperplane search is likely to incur one block access per one directory entry. Thus, the total number of block accesses for searching a hyperplane will be $D_G^{1-1/d}$.

MLGF: The MLGF directory is organized in such a way that the directory elements representing regions that are close to one another in the domain space tend to be in the same physical block. Specifically, an entry in an upper level (H) of the directory points to a physical block containing b entries of the next lower level (L) of the directory. Hence, any hyperplane of level L that contains a directory entry in this physical block will find an average of $b^{1/d}$ directory entries that belong to the hyperplane and that reside in the same block. As a result, the number of block accesses for searching a hyperplane of the lowest level of the directory will be $D_M^{1-1/d}/b^{1/d}$. Since the cost of searching the upper levels of the directory is dominated by that of searching the lowest level, we ignore its effect.

7.3. *Partial-match queries*

Grid File: Combining the effects of directory size and hyperplane search, we obtain the total number of directory block accesses for processing a partial-match query that specifies one attribute to be $O(n^{d(1-1/d)})$.

MLGF: Similarly, we estimate the number of directory block accesses as $O(n^{1-1/d}/b^{1/d})$.

Here, for simplicity, we did not consider accesses to data blocks. It should be mentioned, however, that the MLGF has important advantages in accessing data blocks: In the MLGF there is no redundant access to the same data block, because a data block is pointed by one and only one directory entry. In contrast, in the grid file, a number of directory entries can point to the same data block.

7.4. *Exact-match queries*

Grid File: The number of block accesses for processing an exact-match query is at most two.

MLGF: We estimate the number of levels in the directory as $\lceil \log_b n \rceil$. Including one access to the main data block, the number of block accesses for an exact-match query will be $\lceil \log_b n \rceil + 1$.

7.5. *Splitting of directory*

To investigate the cost of splitting the directory, we calculate the accumulated number of block accesses caused by splitting the directory starting from an empty file to a file of size D_G or D_M. We consider accumulated cost because the pattern of splitting is quite different in two file organizations. As discussed in Section 6.2, the cost of splitting the directory in the grid file is very high, but a split of the directory absorbes many future splits of data blocks. The measure based on accumulated cost would average out differences in splitting patterns.

Grid File: Given a directory size, D_G, a growing file must have experienced directory splits as many times as the total number of hyperplanes. Since the number of hyperplanes in one domain is $D_G^{1/d}$, the total number of hyperplanes is $d \times D_G^{1/d}$. For each split, the entire directory is copied: each block in the directory is read in and written out to a new block. Thus, two block accesses are needed for each directory block. Since

the average size of the directory over the period of the file growth is $D_G/2$, the accumulated number of block accesses for splitting the directory is $2 \times (D_G/2b) \times d \times D_G^{1/d} = d/b \times D_G^{1+1/d}$.

MLGF: The accumulated number of splitting operations on the directory is bounded by the number of entries in the lowest level, D_M, assuming that the propagation of splits to upper levels of the directory is negligible. Associated with each splitting operation, there are two block accesses: one for reading and one for writing a directory block. Thus, the accumulated number of block accesses for splitting the directory is $2 \times D_M$.

7.6. *Merging of directory*

The cost of merging the directory can be obtained in a way similar to the one used for splitting. However, for merging, we have additional costs to consider: 1) the cost of finding a buddy or a neighbour to be merged; 2) (in the case of the grid file) the cost of checking the feasibility of merging a hyperplane every time a data block is merged. Since these costs vary widely according to specific algorithms used, the details will not be further discussed. Let us note, however, that these additional costs will be much higher in the case of the grid file, as discussed in Sections 2.3 and 6.3.

7.7. *Summary*

In Table 1, we summarize the results of performance analysis.

	Grid File	MLGF
directory size	$O(n^d)$	$O(n)$
hyperplane search	$D_G^{1-1/d}$	$D_M^{1-1/d}/b^{1/d}$
partial-match query	$O(n^{d(1-t/d)})$	$O(n^{1-t/d}/b^{t/d})$
exact-match query	2	$\lceil \log_b n \rceil + 1$
directory split	$d/b \times D_G^{1+1/d}$	$2 \times D_M$

Table 1: Performance of the MLGF and the grid file.

Example 7.1: Consider a file having the following characteristics: n = 10,000, d = 3, b = 250, number of attributes = 10, block size = 4K bytes, size of a pointer = 4 bytes, size of a hashed key = 4 bytes, size of a directory entry = 16 bytes. Then, the estimated performance can be summarized as in Table 2.

In Example 7.1 it is shown that the MLGF organization provides drastic enhancement in performance when compared with the grid file organization. A major cause of the difference in performance is the size of the directory. Specifically, the size of the grid file directory grows exponentially, while the size of the MLGF directory grows linearly in the size of the file.

	Grid File	MLGF
directory size	$\simeq 10^{12}$	$\simeq 10^4$
partial-match query	$\simeq 10^8$	$\simeq 74$
exact-match query	2	3
directory split	$\simeq 1.2 \times 10^{14}$	$\simeq 2 \times 10^4$

Table 2: Example performance figures.

Let us note that the figure in Table 2 reflect only the worst case performance. Let us now relax the worst case condition and examine the effect of the relaxation on the performance. In general, the size of the grid file directory is $u \times n^v$, where $1 \leq v \leq d$, and u is a constant. The value of v depends on correlation among attributes. We assume that the value of v is 1.1, which is slightly higher than in the case without any correlation (v = 1). We assume that the value of u is 1.0 to make it compatible with that for the MLGF. Then, using the same characteristics of the file as in Example 7.1, we obtain the following performance estimation for the grid file: directory size $\simeq 2.5 \times 10^4$, cost of processing a partial-match query $\simeq$ 858, cost of processing an exact-match query = 2, and accumulated cost of splitting the directory $\simeq 2.2 \times 10^6$. Comparing these figures with those of the MLGF in Table 2, we conclude that, even with a small amount of correlation, performance of the grid file can be notably worse than that of the MLGF.

8. Conclusions and Further Study

We have presented an extension of the grid file that provides a multilevel access mechanism while maintaining a grid-like partitioning of the domain space. The extension offers a number of advantages such as compact representation of the directory, easy splitting and merging of the directory, efficient processing of partial-match queries, and the concept of abstract databases. We believe that the contribution of this paper is a major step towards the design of multidimensional file organizations implementable in practical systems.

Through the work described in this paper, the authors have found many common features and ideas shared by various seemingly different file organizations: binary search trees [Knu73], K-D trees [Ben75], K-D tries [Ore82], B-trees [Bay72], K-D-B-trees [Rob81], digital B-trees [Lom81], tries [Fre60], quad trees [Fin71], linear hashing [Lit80], virtual hashing [Lit78], dynamic hashing [Lar78], extendible hashing [Fag79], multidimensional linear hashing [Oto85a], [Ouk83], [Bur83], dynamic multipaging [Mer82], multidimensional directory [Lio77], and grid files [Nie84], etc. We are currently working on the generalization of the ideas behind all these file organizations by characterizing the way they partition the domain space and the way they represent the partitions. A nice formalism on this issue is the purpose of our current study.

From the discussions in Sections 2.4 and 6.4, we have learned that the performance degrades for partial-match queries as we add more dimensionality. Intuitively, this happens because the size of the hyperplane to be searched gets larger as the dimensionality gets larger. The implication is that, given a query distribution, there must be an optimal set of organizing attributes that gives the best performance. This optimal set may not necessarily be the set of all attributes because of the above mentioned property. We define the problem of finding an optimal set of organizing attributes the *physical database design* problem [Ham76] [Wha83] [Wha84] [Wha85a] [Wha85b]. A systematic approach to the physical database design for the multidimensional file organizations is left as an interesting topic for further study.

Finally, a systematic and comprehensive simulation research on performance and storage utilization of the MLGF organization is an important topic of our immediate future work.

Acknowledgements

The authors are grateful for invaluable feedback they received from many people on an earlier version of this paper. Steve Morgan, Kyoji Kawagoe, Ekow Otoo, Per-Ake Larson, and David Lomet carefully read the paper and contributed very thoughtful comments. The authors wish to thank Witold Litwin and Tim Merret for providing a forum of lively discussion and helpful feedback. Sang-Wook Kim suggested some improvement in the insertion algorithm.

References

[Aho79] Aho, A.V. and Ullman, J.D., "Optimal Partial-Match Retrieval When Fields are Independently Specified," *ACM Trans. Database Systems*, Vol. 4, No. 2, pp. 168-179, June 1979.

[Bay72] Bayer, R. and McCreight, E., "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica*, Vol. 1, No. 3, pp. 173-189, Sept. 1972.

[Bay77] Bayer, R. and Unterauer, K., "Prefix B-Trees," *ACM Trans. Database Systems*, Vol. 2, No. 1, pp. 11-26, Mar. 1977.

[Ber81] Berstein, P.A. and Chiu, D.W., "Using Semijoins to Solve Relational Queries," *JACM*, Vol. 28, No. 1, pp. 25-40, 1981.

[Ben75] Bentley, J.L., "Multidimensional Binary Search Trees Used for Associative Searching," *Commun. ACM*, Vol. 18, No. 9, pp. 509-517, Sept. 1975.

[Bol79] Bolour, A., "Optimality Properties of Multiple-Key Hashing Functions," *J. ACM*, pp. 196-210, April 1979.

[Bur76a] Burkhard, W.A., "Partial Match Retrieval," *BIT*, Vol. 16, pp. 13-31, 1976.

[Bur76b] Burkhard, W.A., "Hashing and Trie Algorithms for Partial Match Retrieval," *ACM Trans. Database Systems*, Vol. 1, No. 2, pp. 175-187, June 1976.

[Bur83] Burkhard, W.A., "Interpolation-Based Index Maintenance," In *Proc. ACM Symp. Principles of Database Systems*, pp. 76-89, 1983.

[Fag79] Fagin R. et al., "Extendible Hashing-A Fast Access Method for Dynamic Files," *ACM Trans. Database Systems*, Vol. 4, No. 3, pp. 315-344, Sept. 1979.

[Fin74] Finkel, R.A. and Bentley, J.L., "Quad Trees: A Data Structure for Retrieval on Composite Keys," *Acta Informatica*, Vol. 4, pp. 1-9, 1974.

[Fre60] Fredkin, E., "Trie Memory," *Commun. ACM*, Vol. 3, No. 9, pp. 490-499, Sept. 1960.

[Fus85] Fushimi, S. et al., "Algorithm and Performance Evaluation of Adaptive Multidimensional Clustering Technique," In *Proc. Intl. Conf. on Management of Data*, ACM SIGMOD, Austin, Texas, pp. 308-318, May 1985.

[Ham76] Hammer, M. and Chan, A., "Index Selection in a Self-Adaptive Database Management System," In *SIGMOD* ACM SIGMOD, Washington, D.C., pp. 1-8, June 1976.

[Hin85] Hinrichs, K.H., "The Grid File System: Implementation and Case Studies of Applications," Ph. D Dissertation, Swiss Federal Institute of Technology, Zurich, 1985.

[Knu73] Knuth, D.E., *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

[Kri84] Krishnamurthy, R. and Morgan, S., "Query Processing on Personal Computers: A Pragmatic Approach," In *Proc. 10th Intl. Conf. Very Large Data Bases*, Singapore,1984.

[Lar78] Larson, P., "Dynamic Hashing," *BIT*, Vol. 18, pp. 184-201, 1978.

[Lar80] Larson, P., "Linear Hashing with Partial Expansions," In *Proc. 6th Intl. Conf. Very Large Data Bases*, IEEE, Montreal, Canada, pp. 224-232, Oct. 1980.

[Lar82] Larson, P., "Performance Analysis of Linear Hashing with Partial Expansions," *ACM Trans. Database Systems*, Vol. 7, No. 4, pp. 566-587, Dec. 1982.

[Lio77] Liou, J.H. and Yao, S.B., "Multi-Dimensional Clustering for Data Base Organizations," *Information Systems*, Vol. 2, pp. 187-198, 1977.

[Lit78] Litwin, W., "Virtual Hashing: A Dynamically Changing Hashing," In *Proc. 4th Intl. Conf. Very Large Data Bases*, IEEE, Berlin, pp. 517-523, 1978.

[Lit79] Litwin, W., "Linear Virtual Hasing: A New Tool for Files and Tables Implementation," Res. Rep. MAP-1-021, I.R.I.A., Jan. 1979.

[Lit80] Litwin, W., "Linear Hashing: A New Tool for File and Table Addressing," *In Proc. 6th Intl. Conf. Very Large Data Bases*, IEEE, Montreal, Canada, pp. 212-223, 1980.

[Lit81] Litwin, W., "Trie Hashing," In *Proc. Intl. Conf. on Management of Data*, ACM SIGMOD, New York, pp. 19-29, 1981.

[Lom81] Lomet, D.G., "Digital B-trees," In *Proc. 7th Intl. Conf. Very Large Data Bases*, IEEE, Cannes, France, pp. 333-344, Sept. 1981.

[Mer82] Merret, T.H. and Otoo, E.J., "Dynamic Multipaging: a Storage Structure for Large Shared Data Banks," *Proc. Intl. Conf. on Database, Improving Usability and Responsiveness*, P. Scheuermann ed., Academic Press, pp. 237-256, 1982.

[Mul81] Mullin, J.K., "Tightly Controlled Linear Hashing without Seperate Overflow Storage," *BIT*, Vol. 21, pp. 390-400, 1981.

[Nie84] Nievergelt, J. et al., "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Trans. Database Systems*, Vol. 9, No. 1, pp. 38-71, Mar. 1984.

[Ore82] Orestein, J.A., "Multidimensional Tries used for Associated Searching," *Information Processing Letters*, Vol. 14, No. 4, pp. 150-157, June 1982.

[Oto84] Otoo, E.J., "A Mapping Function for the Directory of a Multidimensional Extendible Hashing," In *Proc. 10th Intl. Conf. Very Large Data Bases*, Singapore, pp. 493-506, Aug. 1984.

[Oto85a] Otoo, E.J., "A Multidimensional Digital Hashing Scheme for Files with Composite Keys," In *Proc. Intl. Conf. on Management of Data*, ACM SIGMOD, Austin, Texas,pp. 214-231, 1985.

[Oto85b] Otoo, E.J., "Balanced Multidimensional Extendible Hash Tree," draft paper, Carleton University, Ottawa, Canada, 1985.

[Ouk83] Ouksel, M. and Scheuermann P., "Storage Mapping for Multidimensional Linear Hashing, " In *Proc. ACM Symp. Principles of Database Systems*, Atlanta, Georgia, pp. 90-105, 1983.

[Riv76] Rivest, R.L., "Partial-Match Retrieval Algorithms," *SIAM J. Computing*, Vol. 5, No. 1, pp. 19-50, Mar. 1976.

[Rob81] Robinson, J.T., "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," In *Proc. Intl. Conf. on Management of Data*, ACM SIGMOD, New York, pp. 10-18, 1981.

[Sch81] Scholl, M., "New File Organization Based on Dynamic Hashing," *ACM Trans. Database Systems*, Vol. 6, No. 1, pp. 194-211, Mar. 1981.

[Sch82] Scheuermann, P. and Ouksel, M., "Multidimensional B-Trees for Associative Searching in Database Systems," *Information Systems*, Vol. 7, No. 2, pp. 123-137, 1982.

[Wha83] Whang, K.-Y., Wiederhold, G. and Sagalowicz, D., "Estimating Block Accesses in Database Organizations - A Closed Noniterative Formula," *Commun. ACM*, Vol. 26, No. 11, pp. 940-944, Nov. 1983.

[Wha84] Whang, K.-Y., Wiederhold, G. and Sagalowicz, D., "Seperability - An Approach to Physical Database Design, " *IEEE Trans. on Computers*, Vol. C-33, No. 3, pp. 209-222, Mar. 1984.

[Wha85a] Whang, K.-Y., "Property of Separability in Physical Design of Network Model Databases," *Information Systems*, Vol. 10, No. 1, pp. 57-63, 1985.

[Wha85b] Whang, K.-Y., "Index Selection in Relational Databases," In *Proc. Intl. Conf. on Foundations of Data Organization*, Kyoto, Japan, pp. 369-378, May 1985 (invited paper).

[Wie83] Wiederhold, G., *Database Design*, McGraw-Hill Book Company, New York, 1983, Second Edition.