A New Algorithm for Processing Joins Using the Multilevel Grid File

Sang-Wook Kim* Wan-Sup Cho** Min-Jae Lee** Kyu-Young Whang**

*Information and Electronics Research Center
Korea Advanced Institute of Science and Technology

**Department of Computer Science
Korea Advanced Institute of Science and Technology
and
Center for Artificial Intelligence Research

{wook,wscho,mjlee,kywhang}@mozart.kaist.ac.kr

Abstract

Join is an operation that is frequently used and the most expensive in processing database queries. In this paper we propose a new efficient join algorithm(called the *MLGF-Join*) for relations indexed by the *multilevel grid file*(MLGF), a multidimensional dynamic hashed file organization. The MLGF-Join uses the domain space partition maintained in the directory of the MLGF. The MLGF-Join can process a join with one scan of the relations to be joined, assuming a main memory buffer is available that is sufficiently large for a range-oriented subjoin. Besides, the MLGF-Join does not require the costly preprocessing such as sorting in the sort-merge join algorithm and domain partitioning in the hash join algorithm. We also discuss the advantages of the MLGF-Join over others based on multidimensional dynamic file organizations such as the grid file, K-D tree, and multikey hashing.

1 Introduction

Join is an expensive operation in processing queries in database systems. Effective support of join makes query processing efficient, improving overall performance of database management systems. In this paper we propose a new efficient algorithm for processing join operations.

Join algorithms have been studied extensively in the literature. The nested-loop join algorithm and the sort-merge join algorithm[3] are traditional ones. Recently, the hash join algorithm[10] and many variations have also been proposed[5][16].

Hash join algorithms are classified into three categories: the simple hash join algorithm[5], the Grace hash join algorithm[10], and the hybrid hash join algorithm[16]. Although they differ in detailed procedures, they employ a common basic strategy. First, the partitioning phase begins by partitioning two relations R and S into n disjoint hash buckets using a common hash function. A record belongs to a hash bucket according to its hash value of the join attribute. Since the same hash function is applied to both R and S, the records in the i-th hash bucket of R may join with only those in the i-th hash bucket of S. Next, the join phase performs n subjoins separately, each of which is done on the i-th hash buckets of R and S. This strategy saves the cost by avoiding unnecessary attempt for joining records in the buckets that have different hash values.

A multidimensional dynamic file organization is a file structure for efficient processing of queries involving more than one attribute[1]. The multidimensional dynamic file organization adapts to dynamic situations where record insertions and deletions occur by splitting and merging domain space. The *domain space* is the Cartesian product of domains of all the attributes. The status of domain space partition is reflected in the directory of the file. Just as in hash join algorithms, we can use this domain space partition for join to avoid unnecessary attempt for joining the records in R and S that are in hash buckets that cannot be joined. Furthermore, we can obviate the need for the costly partitioning phase of the hash join algorithm by maintaining the status of domain space partition dynamically in the directory. The partitioning phase of the hash join algorithm incurs a significant overhead since it must be done each time a join is processed.

Recently, many multidimensional dynamic file organizations have been proposed in the literature. Typical examples are the K-D tree[2], K-D-B tree[15], multidimensional extendible hashing[13], multidimensional linear hashing[14], interpolation-based indexes[4], grid file[12], BANG file[7], and multilevel grid file[18][19].

Proceedings of the Fourth International Conference on Database Systems for Advanced Applications (DASFAA'95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995

[1] A set of attributes that participates in organizing a file is called organizing attributes[18]. Since all the attributes mentioned in the following sections are organizing attributes, we simply call them *attributes*.

Join algorithms based on them have also been suggested. They are the join algorithms using the K-D tree[11][8] and the grid file[8][1], and the super join algorithm[17] using multikey hashing.

The K-D tree[2] and the modified grid file proposed by Becker et al.[1] employ unbalanced tree structures for their directories. In database environments, a directory often does not fit in main memory due to the size and thus must reside in disk. If a tree is extremely unbalanced because of data skew, there is a big difference in performance between the best and worst cases. In the worst case, a large number of nodes in the tree should be traversed for querying a record. This increases the number of disk accesses and prolongs the response time. Therefore, the multidimensional dynamic file organization having an unbalanced tree as its directory structure is inadequate for database environments.

In the grid file[12] and multikey hashing[17], the directory is organized as a simple array structure. Thus, the size of a directory may become abnormally larger than is necessary since the growth of the directory is highly affected by data distributions, data skew, or correlation among different attributes(henceforth, we call *data characteristics)*[9]. As a result, the join algorithms using these structures[17][8] suffer from the overhead of processing large directories.

This paper proposes an efficient algorithm for joining relations indexed by the *multilevel grid file(MLGF)*[18][19]. We call the new algorithm the *MLGF-Join*. The MLGF is a multidimensional dynamic hashed file organization employing a balanced tree structure for its directory. It has been shown in [9] that the size of the MLGF directory is linearly dependent on the size of the data file regardless of data characteristics. Thus, the MLGF-Join solves many problems of earlier ones in the literature. Assuming we use an LRU buffer replacement algorithm and a sufficient main memory buffer is available, the MLGF-Join performs a join with one scan of each relation, minimizing the number of disk accesses for join processing.

The organization of the paper is as follows. Section 2 reviews the characteristics of the MLGF. Section 3 explains the basic strategy of the MLGF-Join, and Section 4 presents the detailed algorithm. After discussing the advantages of the MLGF-Join in Section 5, we conclude the paper in Section 6.

2 Multilevel Grid File

This section briefly describes the dynamic and structural characteristics of the MLGF.

2.1 Dynamic characteristics of MLGF

The MLGF is a multidimensional dynamic hashed file organization that consists of two components: a directory and data pages. The directory is a hierarchical index structure for the records stored in data pages. The directory reflects the status of the domain space partition with the directory entries being in one-to-one correspondence with the regions in the domain space. The regions represented by the lowest level of the directory are in turn in one-to-one correspondence with data pages allocated for themselves.

The MLGF adapts to dynamic situations where record insertions and deletions occur by splitting and merging data pages. When a new record is inserted into the MLGF, the region that the record belongs to is found by searching the directory, and the record is inserted into the data page allocated for that region. If the data page overflows, the region is split into two equal-sized subregions, and the records in the overflowed data page are distributed into the two new data pages according to the subregions that they belong to.

When records are deleted repeatedly, the MLGF shrinks. When a record is deleted from the MLGF, the region that the record belongs to is found by searching the directory, and the record is deleted from the data page allocated for that region. If the number of records in the data page falls below a certain threshold (i.e., the data page underflows), the region of the data page is considered for merging with one of its buddies. A *buddy* of region *A* is defined to be an adjacent, equal-sized region that forms a rectangle when merged with region *A*. When merging actually occurs, all the records in the two data pages are consolidated into one, and the other data page is deallocated.

The MLGF employs the *local splitting strategy* that splits a region locally when the corresponding data page overflows[20][9]. The local splitting strategy prevents the MLGF from creating unnecessary directory entries. Figure 1 compares the splitting strategy of the MLGF with those of other file organizations: the grid file[12] and multikey hashing[17].

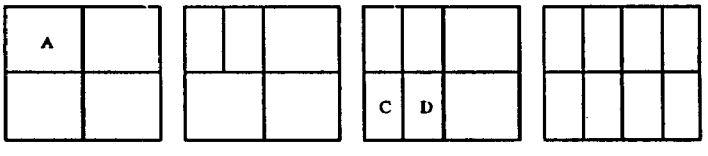


Figure 1. Region splitting strategies.

Figure 1(a) represents the status of domain space partition at the time region *A* is to split. Figure 1(b) shows the partition that the MLGF generates after splitting region *A* locally. Figure 1(c) shows the partition that the grid file generates, in which the entire hyperplane containing region *A* is split. Note that, in this partition, both regions *C* and *D* point to the same data page. Figure 1(d) represents the partition that multikey hashing generates. It causes even more region splits to satisfy the equi-depth requirement of the directory for array-index computation of the extendible hashing[6].

The local splitting strategy makes the directory size of the MLGF increase linearly in the number of records regardless of data characteristics[9], thereby minimizing the storage overhead for the directory. This strategy also enhances the performance of join since the size of the directory to be accessed is much smaller than in other algorithms[17][8].

2.2 Structural characteristics of MLGF

A directory entry consists of a *region vector* and a *pointer(PTR)* to a data page or a lower-level directory page. A region vector in an n-dimensional MLGF consists of n hash values that uniquely identify the region. A region vector indicates the position, shape, and size of a region. The i-th hash value of a region vector is the common prefix of the hash values for the i-th attribute of all the records that belong to the region. Thus, a directory entry in an n-dimensional MLGF is represented as a record with $(n+1)$ attributes, and a directory is represented as a relation having such records. Hence, the MLGF can handle data records and directory entries uniformly.

Figure 2 shows the status of a two-dimensional domain space partition and the directory entries corresponding to the regions in the domain space. Directory entry *d1*, <00, 0, PTR1>, represents the region for all the possible records whose hash values of the first attribute begin with '00' and those of the second attribute begin with '0'. Directory entry *d3* <1, –, PTR3> represents the region for all the possible records whose hash values of the first attribute begin with '1' regardless of those of the second attribute; i.e., the symbol '–' in directory entry *d3* represents the entire domain of the second attribute.

In order to determine whether a region is included in another region, the MLGF uses *prefix-matching*. Prefix-matching checks whether a hash value is a prefix of another hash value. For example, the hash value '101' prefix-matches with the hash value '10100', but the hash value '101' does not prefix-match with the hash value '10000'. For the two hash values a and a' of a common attribute A, if a is a prefix of a', the region represented by a includes the region represented by a'.

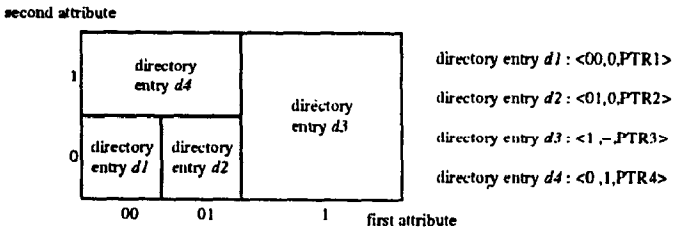
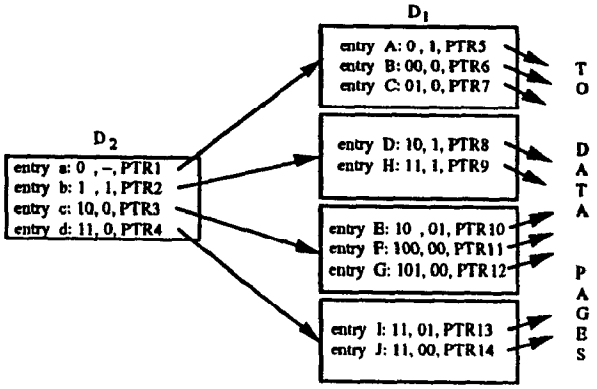


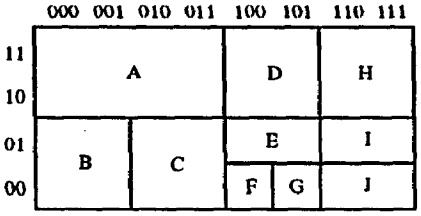
Figure 2. Directory entries and the corresponding regions.

The MLGF employs a multilevel balanced tree structure for its directory. Just as the lowest-level directory is built on top of the data file, the i-th level directory D_i is built on top of the next-lower-level directory D_{i-1} treating it as a data file. This process is repeated until the highest-level directory (the *root-level*) can fit in a disk page. These structural characteristics are maintained by the insertion and deletion algorithms presented in [19].

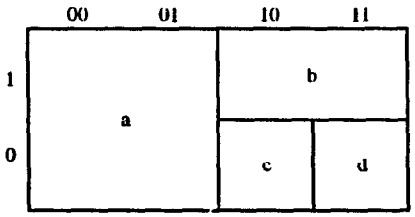
To illustrate the multilevel nature of the MLGF directory, consider a two-dimensional MLGF with a two-level directory consisting of D_1 and D_2 in Figure 3(a). Figure 3(b) and Figure 3(c) show the status of the domain space partition induced by D_1 and D_2, respectively. The rectangles in Figure 3(b) are the regions that the directory entries in D_1 represent, with the symbols in rectangles representing the corresponding directory entries. Thus, the directory D_1 has ten directory entries. The directory D_2 has four directory entries and is the root for the MLGF. The directory entry in D_2 with the region vector <10,0> represents region c in Figure 3(c) pointing to the directory page in D_1 that contains three entries, *E*, *F*, and *G* in Figure 3(b), which form a finer partition of region c. Note that the first(second) element of the region vector <10,0> is the common prefix of the first(second) elements of the region vectors of the directory entries, *E*, *F*, and *G*.



(a) The structure of a two-level MLGF directory.



(b) Regions represented by directory entries in D_1.



(c) Regions represented by directory entries in D_2.

Figure 3. A two-level MLGF directory and its domain space partition.

3 Join Strategy Using MLGF

This section presents the basic strategy of the MLGF-Join. We first discuss how the domain space partition information maintained in the directory of the MLGF can be utilized in

join processing. Then, we describe a join strategy.

Before proceeding, we define some terminology used in the following sections. A *predicate* means a condition in a query and consists of simple predicates that are connected by logical operators such as 'AND', 'OR', or 'NOT'. Using conditional operators such as '<', '>', '=', '!=', '<=', or '>=', a *simple predicate* defines a condition 1) between an attribute of a relation and a constant, or 2) between an attribute of a relation and an attribute of another relation. The latter is called a *join predicate*. The attributes in a join predicate are called *join attributes*[2]. The domain of a join attribute is called a *join domain*.

The basic strategy of the MLGF-Join for joining two relations R and S is to minimize the number of disk accesses. As in the hash join algorithm, unnecessary disk accesses can be avoided by isolating a set of records of S that can possibly be joined with a given set of records of R. This can be done by subdividing the join domain of each relation into a set of *subjoin ranges* based on the domain space partition information in the MLGF directory. The sets of subjoin ranges of two relations may not be identical. Relations R and S are subsequently joined by performing subjoins for each subjoin range. We call a subjoin for each subjoin range a *range-oriented subjoin*.

To explain the basic join strategy, let us use an example database. Figure 4 shows the domain space partition induced by the MLGF directories for R and S. Relations R and S consist of four and six data pages, respectively. For simplicity, we assume each directory has only one level (the root level). Relation R has attributes A and B, and relation S attributes B and C. The join attribute is B.

Before performing range-oriented subjoins, we first subdivide the join domains for R and S into sets of subjoin ranges represented as hash values. We use two conditions for determining a subjoin range. We explain why we use these conditions after describing the detailed join strategy.

- Condition 1: A subjoin range should overlap with as small a number of regions as possible.
- Condition 2: A subjoin range satisfying Condition 1 should be as large as possible.

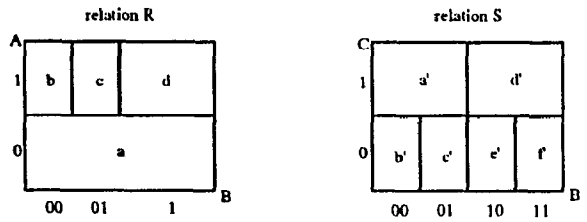


Figure 4. Domain space partitions represented by the MLGF directories for relations R and S.

In Figure 4, the first subjoin range of R satisfying the two conditions has the hash value '00' and overlaps two regions a and b. If the subjoin range became larger, it would overlap another region c violating Condition 1. On the other hand, if the subjoin range became smaller, it would violate Condition 2. Similarly, we can determine the two sets of subjoin ranges: {00, 01, 1} for R and {00, 01, 10, 11} for S.

Using the sets of subjoin ranges of R and S thus obtained, we then readjust the set of subjoin ranges. When a subjoin range I in a set is divided into finer subjoin ranges $I_1, I_2, ..., I_j$ in the other set, the subjoin range I is replaced by the subjoin ranges $I_1, I_2, ..., I_j$. In the example of Figure 4, the set of subjoin ranges is {00, 01, 10, 11} since the subjoin range '1' in the first set is subdivided into two subjoin ranges '10' and '11' in the second set.

Finally, for each subjoin range I_i, we perform a range-oriented subjoin. We use the *prefix-matching* operation mentioned in Section 2 for examining whether a directory entry should participate in a range-oriented subjoin for a subjoin range. A directory entry participates in a range-oriented subjoin whenever its hash value of the join attribute prefix-matches with the hash value representing the subjoin range.

The range-oriented subjoins are executed in the lexicographical order of their hash values. Since multiple regions of a relation may participate in a range-oriented subjoin, each range-oriented subjoin is partitioned into a set of region-oriented subjoins. The *region-oriented subjoin* is a join between the sets of records in regions of R and S overlapping a subjoin range. When all the region-oriented subjoins for a subjoin range I_i are completed, so is the range-oriented subjoin. Likewise, when range-oriented subjoins for all the subjoin ranges are completed, the entire join is finished.

Let us examine how the join is processed using the set of subjoin ranges {00, 01, 10, 11} in Figure 4. We first find the directory entries participating in the range-oriented subjoin for the first subjoin range '00'. In relation R, the hash values of the join attribute of the directory entries a ('–') and b ('00') prefix-match with the hash value '00'. Similarly, in relation S, the hash values of the join attribute of the directory entries a' ('0') and b' ('00') also prefix-match with the hash value '00'. Therefore, the range-oriented subjoin for the subjoin range '00' consists of four region-oriented subjoins (a, a'), (a, b'), (b, a'), and (b, b'). Similarly, for the subjoin ranges '01', '10', and '11', we have the sets of region-oriented subjoins $\{(a, a'), (a, c'), (c, a'), (c, c')\}$, $\{(a, d'), (a, e'), (d, d'), (d, e')\}$, and $\{(a, d'), (a, f'), (d, d'), (d, f')\}$ in a nested-loop fashion. Processing region-oriented subjoins for a range-oriented subjoin resembles the nested-loop join algorithm. Thus, we call the relation in the outer loop, such as R, the *outer relation*, and the relation in the inner loop, such as S, the *inner relation*.

We now discuss the number of disk accesses incurred in processing joins based on our strategy. First, we consider the number of disk accesses for a data page corresponding to a region that overlaps only one subjoin range. Though a data page of the outer relation is accessed only once, a data page of the inner relation can be accessed more than once since a range-oriented subjoin is performed using a set of

[2] If a conditional operator in a join predicate is '=', the join is called an *equijoin*. This paper deals with equijoins since the join algorithms based on hashing are applicable only to equijoins.

region-oriented subjoins in the nested-loop fashion. Nevertheless, the number of data pages corresponding to the regions that overlap a subjoin range is much smaller than the number of all the data pages in a relation. Thus, if we assume that the main memory buffer is large enough to contain all the data pages corresponding to the regions that overlap a subjoin range and that the LRU algorithm is employed for buffer replacement, a data page in the inner relation also needs only one disk access.

Second, we discuss the number of disk accesses for a data page corresponding to a region that overlaps more than one subjoin range. Since this data page cannot be processed completely in a range-oriented subjoin, it should be processed again in the next range-oriented subjoin. However, this data page can be accessed from the main memory buffer without additional disk accesses by an LRU buffer replacement algorithm since it has already been brought in during the previous range-oriented subjoin. In summary, the join using our strategy is performed through one scan of all the data pages of each relation assuming that the main memory buffer is large enough to contain the data pages processed in each range-oriented subjoin.

Last, we explain the reason why we use the two conditions in determining a subjoin range. Condition 1 restricts the number of data pages(a page per region) to be processed in a range-oriented subjoin, keeping the size of the main memory buffer small. Once Condition 1 is satisfied, Condition 2 minimizes the number of range-oriented subjoins.

4 MLGF-Join

This section describes the MLGF-Join implementing the basic strategy described in Section 3. Section 4.1 presents the algorithm in detail. Section 4.2 shows an example explaining various execution steps of the algorithm.

4.1 Join algorithm

We first define the notation to be used in describing the MLGF-Join. *R* and *S* are relations participating in the join: *R* is the outer relation and *S* the inner relation. *Page_R* and *Page_S* are input parameters of the recursive procedure *Recursive_Join* that processes a range-oriented subjoin. The size of *Page_R* and *Page_S* is one disk page. *Join_Range* is another input parameter of *Recursive_Join*. It has a hash value of the subjoin range to be processed in *Recursive_Join*. *PTR* is a component of the directory entry of the MLGF pointing to a data page or a lower-level directory page.

The MLGF-Join consists of two parts: *Main_Program* and *Recursive_Join*. *Main_Program* in Figure 5 reads the root directory pages of the MLGF's for *R* and *S* into *Page_R* and *Page_S*, respectively, and call *Recursive_Join*. The third input parameter '–' in *Recursive_Join* means that the subjoin range to be processed is the entire join domain. The purpose of *Main_Program* is just to make the top-level recursive call, and actual join processing is done in *Recursive_Join*.

```
Main_Program
  read the root directory pages of R and S into Page_R and Page_S,
  respectively
  call Recursive_Join (Page_R, Page_S, ' - ')
end_Main_Program
```

Figure 5. Main_Program.

Recursive_Join in Figure 6 has three input parameters: *Page_R*, *Page_S*, and *Join_Range*. *Page_R* and *Page_S* have data records or directory entries to be processed. *Join_Range* has the hash value representing the subjoin range to be handled. *Recursive_Join* deals with four different cases depending on the contents of *Page_R* and *Page_S*.

```
Recursive_Join(Page_R, Page_S, Join_Range)
  Case 1:(each of Page_R and Page_S has a page of the lowest-level
         directory) OR (each of Page_R and Page_S has a page of a
         higher-than-the-lowest-level directory)
         Join_Range is partitioned into a set of Subjoin_Ranges based on
         the directory entries in Page_R and Page_S
         for each Subjoin_Range in the lexicographical order
           for each directory entry r in Page_R whose hash value of the
               join attribute prefix-matches with Subjoin_Range
             for each directory entry s in Page_S whose hash value of
                 the join attribute prefix-matches with Subjoin_Range
                 read the next-lower-level pages pointed by PTR's of
                 directory entries r and s into Page_R' and Page_S',
                 respectively
                 call Recursive_Join(Page_R', Page_S', Subjoin_Range)
             end_for
           end_for
         end_for
  end_Case 1
  Case 2: (Page_R has a page of a higher-than-the-lowest-level
         directory) AND (Page_S has a page of the lowest-level directory)
         Join_Range is partitioned into a set of Subjoin_Ranges based on
         the directory entries in Page_R and Page_S
         for each Subjoin_Range in the lexicographical order
           for each directory entry r in Page_R whose hash value of the
               join attribute prefix-matches with Subjoin_Range
               read the next-lower-level page pointed by PTR of directory
               entry r into Page_R'
               call Recursive_Join(Page_R', Page_S, Subjoin_Range)
           end_for
         end_for
  end_Case 2
  Case 3: (Page_R has a page of the lowest-level directory) AND (Page_S
         has a page of a higher-than-the-lowest-level directory)
         Join_Range is partitioned into a set of Subjoin_Ranges based on
         the directory entries in Page_R and Page_S
         for each Subjoin_Range in the lexicographical order
           for each directory entry s in Page_S whose hash value of the
               join attribute prefix-matches with Subjoin_Range
               read the next-lower-level page pointed by PTR of directory
               entry s into Page_S'
               call Recursive_Join(Page_R, Page_S', Subjoin_Range)
           end_for
         end_for
  end_Case 3
  Case 4: (each of Page_R and Page_S has a data page)
           for all the records in Page_R and Page_S whose hash values of
               the join attributes prefix-match with Join_Range
               perform join
           end_for
  end_Case 4
end_Recursive_Join
```

Figure 6. Procedure Recursive_Join.

Case 1: Both *Page_R* and *Page_S* have pages of the lowest-level directory or those of a higher-than-the-lowest-level directory. We obtain a set of *Subjoin_Range*'s satisfying the two conditions described in Section 3. For each *Subjoin_Range*, we find pairs of directory entries *(r, s)*, where *r* is in *Page_R*, and *s* is in *Page_S*, whose hash values of the join attributes prefix-match with *Subjoin_Range*. For each such pair *(r, s)*, we read the pages pointed by *PTR's* of *r* and *s* into *Page_R'* and *Page_S'*, respectively. These two pages are the subjects of the region-oriented subjoin to be processed by the next-level recursive call. Finally, we call *Recursive_Join* recursively using *Page_R'*, *Page_S'*, and *Subjoin_Range* as input parameters.

Case 2: *Page_R* has a page of a higher-than-the-lowest-level directory, and *Page_S* has a page of the lowest-level directory. This happens when the height of the directory of *S* is smaller than that of *R*. As in Case 1, we obtain a set of *Subjoin_Range*'s satisfying the two conditions. For each *Subjoin_Range*, we find pairs of directory entries *(r, s)*, where *r* is in *Page_R*, and *s* is in *Page_S*, whose hash values of the join attributes prefix-match with *Subjoin_Range*. For *r*, as in Case 1, we read the next-lower-level directory page pointed by *PTR* of *r* into *Page_R'*. For *s*, however, there is no more next-level directory page to be read into *Page_S'*. Thus, we pass *Page_S* itself as an input parameter of the next-level recursive call instead of *Page_S'*. Case 2 solves the problem of the difference between the heights of the directories of *R* and *S*.

Case 3: *Page_R* has a page of the lowest-level directory, and *Page_S* has a page of a higher-than-the-lowest-level directory. This case is identical to Case 2 except that *R* and *S* are exchanged and can be processed similarly.

Case 4: Each of *Page_R* and *Page_S* has a data page. We perform a region-oriented subjoin on two sets of records in *Page_R* and *Page_S*. We restrict the subjects of the region-oriented subjoin to the records whose hash values of the join attributes prefix-match with *Join_Range*. This restriction prevents duplicate processing of the same records when the regions of the data pages span more than one subjoin range.

In summary, *Recursive_Join* is performed by calling itself recursively along the path from the root directory page to the data page. *Recursive_Join* partitions a join into a set of range-oriented subjoins, and then into a set of region-oriented subjoins. These range-oriented subjoins are done in the lexicographical order of the hash values representing the subjoin ranges.

4.2 An example execution of the MLGF-Join

This subsection explains how the MLGF-Join is executed using an example. Figure 7 shows domain space partitions induced by each level of the MLGF directories for *R* and *S*. The symbols in the rectangles represent directory entries corresponding to the regions in the domain space. Relation *R* has attributes *X* and *Y*, and relation *S* attributes *Y* and *Z*. The join attribute is *Y*.

Figure 8 shows execution steps for joining *R* and *S* in the order that *Recursive_Join* is called. Each round enclosure corresponds to a recursive call and the number at the upper left corner represents the order in which the recursive call is made. Two small rectangles in a round enclosure are *Page_R* and *Page_S*, the input parameters of the Recursive Call. A symbol above a rectangle is the directory entry. The directory entry is the subject of the region-oriented subjoin to be processed in the recursive call. Each entry in a rectangle represents a directory entry and its hash value of the join attribute. Hash values below two rectangles correspond to subjoin ranges to be processed in the next-level recursive call. An arrow starting from the subjoin range indicates that a recursive call is made for processing one of the region-oriented subjoins that belong to the subjoin range. A rectangle in the lowest-level recursive call represents a data page pointed by *PTR* of the directory entry shown above the rectangle.

Recursive Call 1 is made by *Main_Program* for processing the root-level directory entries in *Page_R* and *Page_S*. Since each of *Page_R* and *Page_S* has a page of a higher-than-the-lowest-level directory, this call belongs to Case 1 in Figure 6. *Join_Range* '–' is subdivided into a set of *Subjoin_Ranges* {0, 10, 11}. For the first *Subjoin_Range* '0', a range-oriented subjoin is performed through a set of region-oriented subjoins. We select directory entries *a* and *a'* as subjects of the first region-oriented subjoin since their hash values of the join attribute prefix-match with '0'. Then, the pages pointed by *PTR's* of *a* and *a'* are read into *Page_R'* and *Page_S'* and passed as input parameters to Recursive Call 2. *Join_Range* to be processed in Recursive Call 2 is *Subjoin_Range* '0' in Recursive Call 1. Since each of *Page_R* and *Page_S* in Recursive Call 2 has a page of the lowest-level directory, this call also belongs to Case 1 in Figure 6. *Join_Range* '0' is subdivided into a set of *Subjoin_Ranges* {00,01}. For the first *Subjoin_Range* '00', a range-oriented subjoin is performed through a set of region-oriented subjoins. The directory entries *A* and *A'* are selected as subjects of the first region-oriented subjoin since their hash values of the join attribute prefix-match with '00'. Then, the pages pointed by *PTR's* of *A* and *A'* are read into *Page_R'* and *Page_S'* and passed as input parameters to Recursive Call 3. *Join_Range* to be processed in Recursive Call 3 is *Subjoin_Range* '00' in Recursive Call 2.

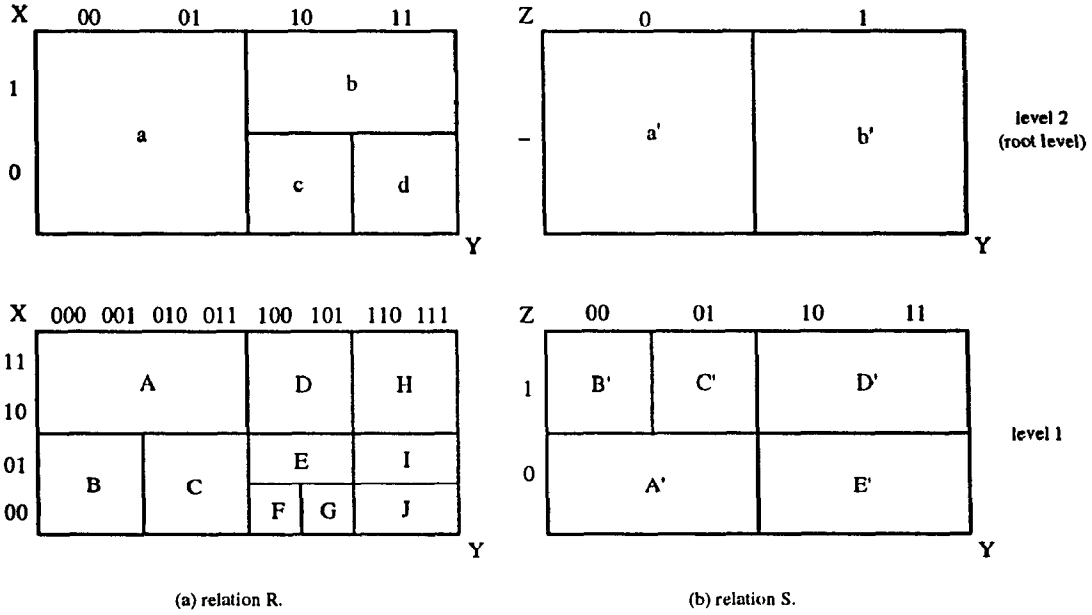
Since each of *Page_R* and *Page_S* in Recursive Call 3 has a data page, this call belongs to Case 4 in Figure 6. Thus, join is performed for the sets of records in *Page_R* and *Page_S* whose hash values of the join attribute prefix-match with *Join_Range* '00'.

When Recursive Call 3 returns to Recursive Call 2, we select *A* and *B'*, whose hash values of the join attribute prefix-match with '00', as subjects of the next region-oriented subjoin for *Subjoin_Range* '00'. Then, we read the pages pointed by *PTR's* of *A* and *B'* into *Page_R'* and *Page_S'* and pass them as input parameters to Recursive Call 4. Since each of *Page_R* and *Page_S* in Recursive Call 4 has data pages, we join the sets of records in *Page_R* and *Page_S*, whose hash values of the join attribute prefix-match with *Join_Range* '00'.

When all four region-oriented subjoins for

Subjoin_Range '00' in Recursive Call 2 are completed, a new range-oriented subjoin for *Subjoin_Range* '01' is started. When all the region-oriented subjoins from Recursive Call 7 to Recursive Call 10 are completed, the range-oriented subjoin for *Subjoin_Range* '01' is finished, and Recursive Call 2 returns to Recursive Call 1.

Likewise, when the range-oriented subjoin for *Subjoin_Range* '0' in Recursive Call 1 is completed, the range-oriented subjoins for *Subjoin_Ranges* '10' and '11' are started in turn. Finally, when Recursive Call 30–the last region-oriented subjoin–finishes, the entire join of *R* and *S* is completed.



(a) relation R. (b) relation S.

Figure 7. Domain space partitions represented by each level of the MLGF directories for R and S.

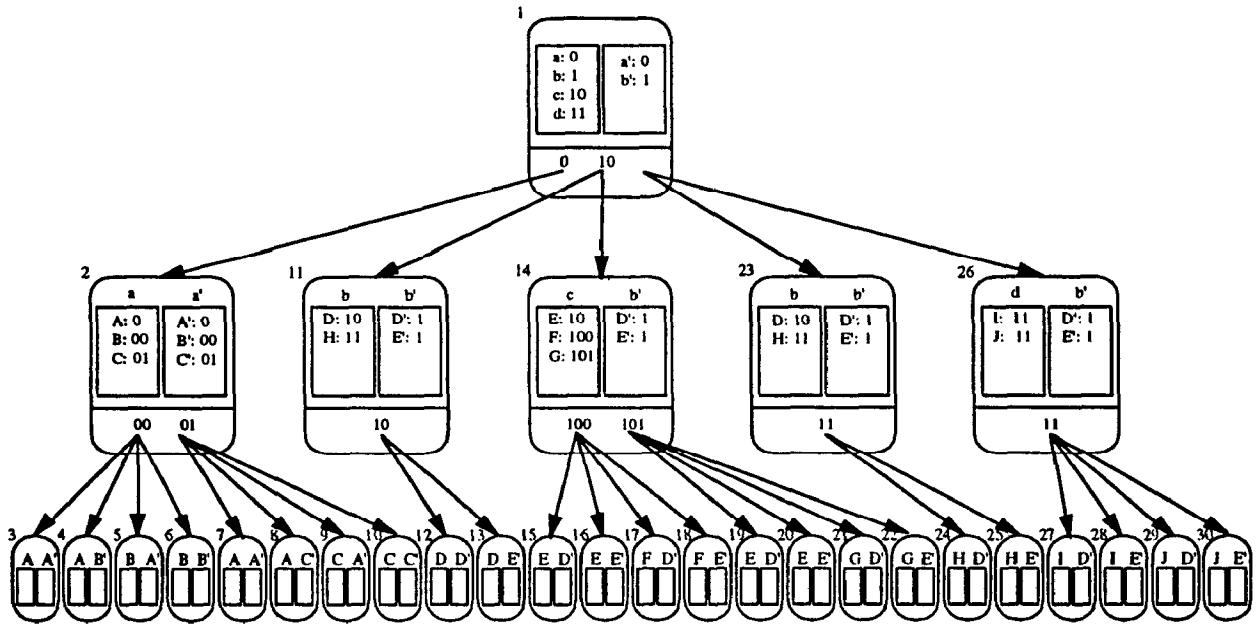


Figure 8. Execution steps for join.

5 Advantages of the MLGF-Join

This section discusses the advantages of the MLGF-Join over the hash join algorithm, the sort-merge join algorithm, and other join algorithms based on the grid file, K-D tree, and multikey hashing.

5.1 Preprocessing overhead

The MLGF-Join does not require the preprocessing phase needed in the sort-merge join algorithm and the hash join algorithm. Instead, the MLGF dynamically maintains the status of domain space partition in its directory. Thus, there is no need for preprocessing each time join is performed.

5.2 Directory handling

As explained in Section 2.2, each level of the directory of an n-dimensional MLGF is regarded as an $(n+1)$ attribute relation. This representation of the directory enables data pages and directory pages to be handled in the same fashion, making the MLGF-Join simple and clean. In contrast, the join algorithms in [17][11][1][8] do not mention any mechanisms for handling the directory possibly because of their complicated directory structures.

5.3 Overhead for accessing directories

The overhead for accessing directories has never been considered in [17][11][1][8]. However, it could be significant since the directory itself resides in disk in database environments.

The MLGF directory increases linearly in the number of data records independent of data characteristics such as distributions, skew, or correlation among different attributes[9]. Therefore, the size of the directory to be accessed in processing a join is also linearly dependent on the size of the number of data records. Thus, in the MLGF-Join, there is no performance degradation due to different data characteristics.

Directory sizes of the grid file[12] and multikey hashing[17] are highly affected by data characteristics because of their directory structures. In some cases, the sizes of their directories can be much larger than those of the data files. This degrades the join performance seriously due to the overhead of accessing the directories.

5.4 Identification of subjects of a range-oriented subjoin

The subjects of a range-oriented subjoin are easily identified in the MLGF-Join. As described in Section 4, the directory entries that are the subjects of a region-oriented subjoin can be found easily by prefix-matching with a subjoin range. This process is efficient due to the simplicity of the prefix-matching operation.

The algorithms in [11][8] use the concept called a *wave*–a set of data pages–as a subject for a subjoin. While the algorithm using the grid file[8] finds the wave easily using the *linear scale*[12], the one using the K-D tree should traverse the tree-structured directory repeatedly to find a wave. This technique could cause significant overhead when the directory resides in disk.

5.5 The number of disk accesses

As mentioned in Section 3, range-oriented subjoins are performed in the lexicographical order of the hash values representing subjoin ranges. Assuming that the main memory buffer is large enough to keep all the data pages belonging to a range-oriented subjoin, the LRU algorithm would cause only one disk access per page for the join. Similarly, since the MLGF-Join handles directory pages just like data pages, a directory page would also need one disk access. Therefore, a join can be completed with only one scan of all the pages in two relations.

6 Conclusions

We have proposed a new efficient join algorithm that we call the *MLGF-Join*. The MLGF-Join uses the MLGF, a multidimensional dynamic hashed file organization, as the main data structure. It processes a join by partitioning it into a set of range-oriented subjoins, and subsequently into a set of region-oriented subjoins, which are processed independently of one another.

The main advantages of the MLGF-Join are as follows:

- With a buffer sufficiently large for a range-oriented subjoin, a join requires only one access for a directory or data page.
- The MLGF-Join avoids costly preprocessing for each join by maintaining the MLGF dynamically.
- The MLGF-Join is simple and clean because it handles data pages and directory pages uniformly.
- The number of directory pages to be accessed for processing a join is linearly dependent on the size of the data file regardless of the characteristics of the data.
- Subjects of a range-oriented subjoin can be identified easily using the efficient prefix-matching operation.

As a further study, we are considering a systematic and comprehensive simulation research on the performance of the MLGF-Join. Currently, the MLGF-Join handles only spatial point data in an n-dimensional domain space. We also plan to extend the MLGF-Join for handling spatial non-point data.

Acknowledgments

This research was partially supported by KT (Korea Telecom) Grant GI22210 and by KOSEF (Korea Science and Engineering Foundation) Grant GD00468 through CAIR (Center for Artificial Intelligence Research). Ju-Won Song carefully read the preliminary version of this paper and contributed helpful comments.

References

[1] Becker, L., Hinrichs, K., and Finke, U., "A New Join Algorithm for Computing Joins with Grid Files," In *Proc. Intl. Conf. on Data Engineering*, IEEE, pp. 189-196, 1993.

[2] Bentley, J. L., "Multidimensional Binary Search Trees in Database Applications," *IEEE Trans. Software Engineering*, Vol. SE-5, No. 4, pp. 333-340, July 1979.

[3] Blasgen, M. W. and Eswaran, K. P., "Storage and Access in Relational Databases," *IBM Systems Journal*, Vol. 4, No. 4, pp. 363-377, 1977.

[4] Burkhard, W. A., "Interpolation-Based Index Maintenance," In *Proc. 2nd ACM Symp. Principles of Database Systems*, ACM SIGMOD, pp. 79-89, 1983.

[5] DeWitt, D. et al., "Implementation Techniques for Main Memory Database Systems," In *Proc. Intl. Conf. on Management of Data*, ACM SIGMOD, pp. 1-8, 1984.

[6] 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.

[7] Freeston, M., "The BANG File: A New Kind of Grid File," In *Proc. Intl. Conf. on Management of Data*, ACM SIGMOD, pp. 260-269, 1987.

[8] Harada, L. et al., "Query Processing Method for Multi-Attribute Clustered Relations," In *Proc. 16th Intl. Conf. on Very Large Data Bases*, pp. 59-70, 1990.

[9] Kim, S. W. and Whang, K. Y., "Asymptotic Directory Growth of the Multilevel Grid File," In *Proc. Intl. Symposium on Next Generation Database Systems and Their Applications*, pp. 257-264, Fukuoka, Japan, Sept. 1993.

[10] Kitsuregawa, M., Tanaka, H., and Moto-Oka, T., "Application of Hash to Database Machine and Its Architecture," *New Generation Computing*, Vol. 1, No. 1, pp. 63-74, 1983.

[11] Kitsuregawa, M., Harada, L., and Takagi, M., "Join Strategies on K-D-Tree Indexed Relations," In *Proc. Intl. Conf. on Data Engineering*, IEEE, pp. 85-93, 1989.

[12] 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.

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

[14] Ouksel, M. and Scheuermann, P., "Storage Mapping for Multidimensional Linear Hashing," In *Proc. 3rd ACM Symp. Principles of Database Systems*, ACM SIGMOD, pp. 90-105, 1983.

[15] 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, pp. 10-18, 1981.

[16] Shapiro, L. D., "Join Processing in Database Systems with Large Main Memories," *ACM Trans. on Database Systems*, Vol. 11, No. 3, pp. 239-264, Sept. 1986.

[17] Thome, J. A., Ramamohanarao, K., and Naish, L., "A Superjoin Algorithm for Deductive Databases," In *Proc. 12th Intl. Conf. on Very Large Data Bases*, pp. 189-196, Aug. 1986.

[18] Whang, K. Y. and Krishnamurthy, R., Multilevel Grid Files, IBM Research Report RC11516, 1985.

[19] Whang, K. Y. and Krishnamurthy, R., "The Multilevel Grid File–A Dynamic Hierarchical Multidimensional File Structure," In *Proc. 2nd Intl. Conf. on Database Systems for Advanced Applications*, pp. 449-459, Apr. 1991.

[20] Whang, K. Y., Kim, S. W., and Wiederhold, G., "Dynamic Maintenance of Data Distribution for Selectivity Estimation," *The VLDB Journal*, Vol. 3, No. 1, pp. 29-51, Jan. 1994.