AN ADAPTIVE DATA PLACEMENT SCHEME FOR PARALLEL DATABASE COMPUTER SYSTEMS

Kien A. Hua* and Chiang Lee

IBM Mid-Hudson Laboratories
Kingston, NY 12401

ABSTRACT

The capacity and performance of database management system (DBMS) using a conventional (von Newmann-type) computer are limited by the total I/O channel bandwidth, the aggregate processing power, and the amount of main memory. With the advent of micro-processor, memory, and communication technology, it is econominally feasible to develop a parallel database computer system. The parallel processing techniques are employed to utilize the available resources in a coordinated fashion to solve the DBMS capacity and performance problems. Relations in such an environment are declustered into fragments and spreaded across computers. To achieve the optimal performance in data processing, it is essential for each computer to have a perfectly balanced load (i.e., identical amount of data). However, fragment sizes may vary due to insertions to and deletions from a relation. To retain good performance, the system needs to periodically rebalance data loads among the computers.

In this paper, we present an adaptive data placement scheme which balances computer work loads during query processing. The entire scheme is built on top of the popular grid file structure (but not limited to grid file). The adaptivity of the scheme and its relevant features are discussed. The cost of load rebalancing is estimated. The result shows that under our assumptions, it is always beneficial to perform load rebalancing before performing a join on skewed data.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

1. INTRODUCTION

In a database processing environment, the fact that disk I/O is the main bottleneck has been a consensus according to researches in the past. As the speed of microprocessors improves rapidly with the development of RISC technology, the problem becomes even more serious. It is more than likely that in the foreseeable future, this situation will not change. This problem is usually addressed today by data declustering in which each relation is partitioned into fragments and spread equally among all disk drives in the system. For instance, a hashed strategy is employed in the Teradata DBC/1012 [Ter88]. A randomizing function is applied to the key attribute of each tuple to select a storage unit. A similar approach is used by the Grace database machine [Kit84]. This storage structure is very efficient for relational operations such as JOIN. However, it supports some of the other operations, such as Range-SELECT, very poorly. Gamma [Dew86] offers more flexibility by allowing a relation to be partitioned in any of three ways: round-robin distribution, hashed or range partitioned. As implied by its name, round-robin distributes the tuples among all storage units in a round-robin fashion. The hashed strategy is similar to that used in the Teradata DBC/1012. In the range partitioned strategy, each computing unit is assigned a range of key values in such a way that the partition of the relation based on the key attribute will result in a balanced data load at each computing unit.

The data placement strategies discussed so far are built around the ideas of hashing and sorting of relations so that relational operations applied on the partitioning attribute can be performed very efficiently. These storage structures, however, would

* Author's current address: University of Central Florida, Department of Computer Science, Orlando, Florida 32816-0362.

not be able to support effectively queries that involve non-partitioning attributes. For instance, for the following database:

SPJ (supplier_number, part_number, project_number, quantity)

S (supplier_number, supplier_name, S_city, status)

P (part_number, part_name, color, weight, P_city)

J (project_number, project_name, J_city)

Let's consider the following queries:

Q1: Find names and cities of those suppliers who supply parts with quantity $\geq$ 50 in any project.

Q2: Find names, colors and weights of those parts that are supplied by a supplier located in New York City.

To perform Q1, two relations, ***S*** and ***SPJ*** need to be joined over supplier_number. To perform Q2, ***S*** and ***P*** need to be joined with ***SPJ*** over attributes supplier_number and part_number. If the relations ***SPJ*** and ***S*** were partitioned into data fragments by hashing on the common attribute supplier_number, then Q1 will perform very efficiently. On the other hand, since part_number is not the partitioning attribute of ***SPJ***, performing Q2 will require a re-hash on part_number and data redistribution, assuming hash-join is used. This process is typically very expensive. Similarly, there are queries that need to join ***P*** and ***SPJ***, to join ***J*** and ***SPJ***, and to join ***S***, ***P***, ***J*** and ***SPJ***, etc. Therefore, not only the partitioning attribute supplier_number, but also the other attributes part_number, and project_number of relation ***SPJ*** are used frequently in the join queries. Another disadvantage is that retrieving tuples using a non-clustering index may involve excessive disk I/O's. In the worst case, the retrieval operation can result in k pages read, where k is the number of qualified tuples. These drawbacks of the horizontal partitioning schemes have placed a limit on the performance improvement of existing parallel database computers.

In this paper, we will present a data placement scheme in which a relation is declustered into data fragments and distributed across multiple processing nodes using multiple attributes of the relation. The technique used to manage the partitioning information is based on the grid file structure originally proposed in [Nie84]. This file structure has the following key advantages over conventional file structures:

1. Since the data partitioning is based on multiple attributes, relational operations that involve any of the partitioning attributes can perform very efficiently.
2. Since the tuples are clustered on multiple indices, the number of I/O's is small even for secondary key accesses.

We extended this file concept to control data placement in a multiprocessor environment.

Since effective data placement is an important lever for load balancing, it is normally the determining factor for the performance of a multiprocessor system. In our design, an initial distribution algorithm is used to distribute the partitioned data fragments to the storage units. The primary objective of this distribution algorithm is to provide a balanced data load for each computing unit in the system. As time goes on, the initially balanced load may become disrupted due to data insertions and deletions. We provide a data redistribution algorithm for the data reorganization in order to correct this data skew problem. Data reorganization is typically very expensive. Due to its high cost, it may be even more desirable to operate the system in the data skew condition than to perform the data rebalancing. In Bubba [Cop88], the system estimates the data reorganization cost and decides whether to tolerate the skewed data or to perform the data redistribution. In this paper, we will present a data reorganization technique that minimizes the redistribution cost so that the system can rebalance its data load more frequently to avoid serious data skew. In addition, a split algorithm was designed, and can be used should the data reorganization algorithm fail to rebalance the data load due to an extremely nonuniform distribution of data in the data space. A merge algorithm is also provided to combine "adjacent" data fragments whose sizes drop below a certain threshold due to data deletions. In the following sections, we will discuss the algorithms in details.

This paper is organized as follows. The design environment and assumptions are described in Section 2. The load balancing scheme, including the initial distribution algorithm and the data reorganization algorithm, is introduced in Section 3. The split and merge algorithms that make our load balancing more adaptive to a changing environment are presented in Section 4. In Section 5, we report on an estimation of savings on redistribution of data for load balancing. Finally, in Section 6, we summarize this research and briefly discuss some future work.

2. ENVIRONMENT AND ASSUMPTIONS

There are basically three different architectures for multiprocessor database machines: Shared Everything (SE), Shared Disk (SD), and Shared Nothing (SN) [Bhi88]. In SE architecture, all disks and memory modules are shared by processors. Data are equally accessible from all processors. In SD architecture, each processor can directly access any disk, but each processor has its own private memory. In SN architecture, each processor has its own private memory and dedicated disk devices. There has been debate about which architecture is superior to the others, although people generally agree that SN architecture is more scalable to achieve high system throughput. For ease of illustration, in this paper we assume that SN architecture is used as the system architecture. However, one should note that the load balancing scheme and its adaptivity are not limited to the type of architecture being used.

We will call each processor with its associated private memory and disk system a processing node (PN). A pool of these PNs are interconnected through an interconnection network. A relation is partitioned based on the grid file concept and distributed into the PNs. As proposed in [Nie84], a grid file mainly contains two parts: a number of linear scales and a multi-dimensional directory. In this paper, this file structure is used to hold the partitioning information for a SN multiprocessor system. The linear scales are used to specify the key ranges in each of the dimensions (keys). The directory contains a number of cells that are formed by partitioning the data space based on the multi-dimensional key ranges. Each cell thus represents a cluster of tuples based on the multi-key data partitioning strategy. Our management scheme is different from the original grid file [Nie84] in that the cell entries contain information about a data fragment (e.g., a file) and its host PN instead of the address of a disk allocate unit (e.g., a page). Each cell in our directory therefore has two entries:

Cell Size:	A number that indicates the number of tuples being assigned to the cell as a result of the data partitioning.
ID:	The Identification of a PN which currently has the tuples being assigned to the cell. We will use PN_i to denote the PN whose ID is i.

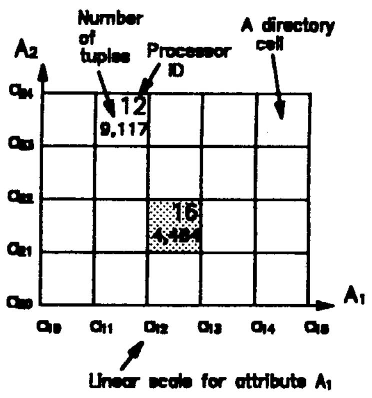

Figure 1. A 2-key partitioning directory

For instance, a two-key (i.e., only two of the attributes are critical to the application) partitioning directory is given in Figure 1. In this example, the directory indicates that there are 4,464 tuples that satisfies the partitioning predicate:

$$a_{12} < A_1 \leq a_{13} \text{ and } a_{21} < A_2 \leq a_{22}$$

and these tuples are currently residing in PN_{16}. For convenience, we will use a large (small) cell to mean a directory cell with a large (small) number of tuples assigned to it, and a large (small) PN to signify a PN that has many (few) tuples assigned to it. Note that the linear scales are not necessarily divided into equal intervals.

In this paper, we focus on the data partitioning problem in a multiprocessor system. Once a relation has been declustered into partitions, how each partition is structured locally in a storage system of a PN is beyond the scope of this paper and will not be discussed. In general, any uniprocessor file structure (e.g., conventional indices, grid files) can be used in conjunction with the proposed partitioning scheme to implement a complete file system for a multiprocessor database computer.

In the subsequent sections, we will discuss the management of the partitioning directory and its use for load balancing in a multiprocessor system. For conciseness in presentation, we assume that each tuple of a relation is equally likely to be accessed by a transaction or a query. However, one should note that the presented algorithms can easily be modified for the case of access skew, a situation in which some of the tuples are being accessed more frequently than other tuples.

3. THE LOAD BALANCING SCHEME

These notations will be used in the following sections:

np:	Number of processing nodes in the system.
d:	Number of dimensions in the directory.
$\|NP_i\|$:	Number of tuples that have been allocated to PN_i.
I_i:	Number of intervals in the ith dimension.
$\|C_{i_1,\ldots,i_d}\|$:	Number of tuples in cell $C_{i_1,\ldots,i_d}$, where $1 \le i_j \le I_j$.
$\|C'_{i,j}\|$:	Number of tuples in the jth largest cell in PN_i.

3.1 The Initial Distribution Algorithm

A simple initial distribution scheme is to divide the linear scales into evenly spaced intervals, and to assign each cell to a distinct PN. This simple scheme has a serious problem. If the data are not uniformly distributed in the data space as in the case shown in Figure 2, some PNs will have more data than the other PNs (i.e., data skew). One solution may be to adjust the partition intervals on the linear scales so that the cell sizes are about the same. The complexity of this approach, however, explodes exponentially as the number of dimensions and the number of intervals increase.

Alternatively, one can partition the relation into a much higher number of cells. These cells then can be assigned to the PNs so as to balance the data load for each PN. A greedy algorithm to achieve this goal is given in the box titled "Initial Distribution Algorithm".

Conceptually, The allocation algorithm is executed by first sorting in descending order the cell sizes in the incomplete directory (i.e., the ID entries are yet missing). The cells are then assigned to the PNs in the sorted order. The assignment is done by allocating the currently largest cell in the sorted list to the currently smallest PN. The cell is then removed from the list. This process is repeated until the sorted list becomes empty.

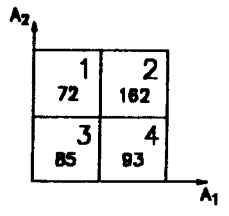

Figure 2. A partitioning example using a simple algorithm

Initial Distribution Algorithm

```
1. For each dimension, partition the linear scale into np equally spaced intervals;
2. Compute $||C_{i_1,\ldots,i_d}||$ for $1 \le i_j \le I_j$ and $1 \le j \le d$;
3. Sort the list of $||C_{i_1,\ldots,i_d}||$, where $1 \le i_j \le I_j$ and $1 \le j \le d$, into descending order. Let LC be the sorted list.
4. Let $||NP_i|| = 0, 1 \le i \le np$;
5. Repeat_Until LC is empty
   a. Find a $PN_i$ where $||NP_i|| = \min_{1 \le j \le np} (||NP_j||)$;
      /* If there are more than one PNs that satisfy this condition, select one arbitrarily. */
   b. Assign the first (i.e., largest) cell, $C_{i_1,\ldots,i_d}$ from $LC$, to the $PN_i$ ;
   c. $LC = LC \setminus \{C_{i_1,\ldots,i_d}\}$;   /* Remove $C_{i_1,\ldots,i_d}$ from $LC$ */
   d. $||NP_i|| = ||NP_i|| + ||C_{i_1,\ldots,i_d}||$ ;
End_Repeat;
```

An application of this algorithm is illustrated in Figure 3. In this example, we again assume that the dimension of the directory is two for clarity. Step 1 of the algorithm divides the directory into 16 cells. There are four PNs in this example. Note that the directories shown in Figure 2 and Figure 3 are assumed to be for the same relation (i.e., same data distribution in the data space). After sorting the cell sizes into descending order as stated in steps 2 and 3 in the algorithm, the process of assigning the cells to PNs as described in step 4 of the algorithm is illus-

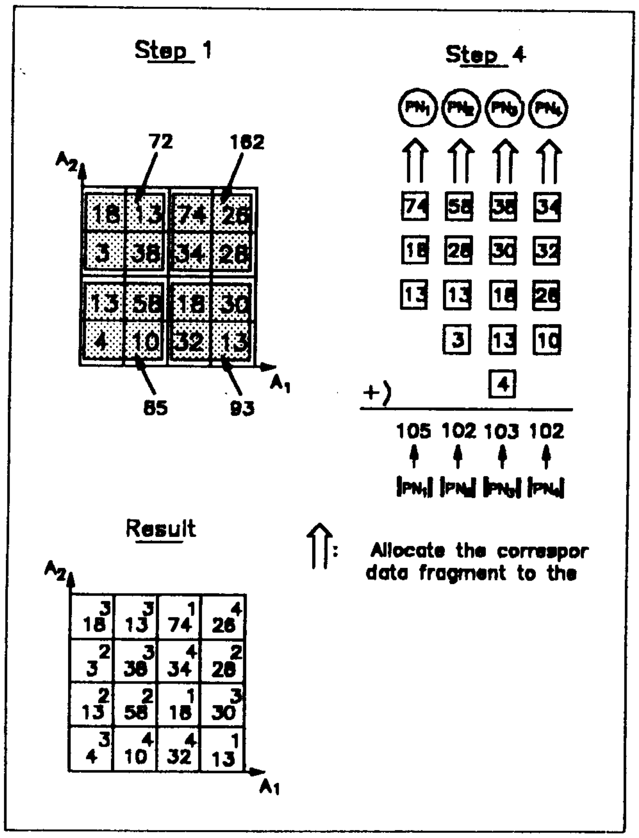

Figure 3. An example of the initial distribution algorithm

trated in Figure 3. The largest cell (i.e., cell size = 74) is first assigned to PN_1; the second, third, and fourth largest cell are then assigned to PN_2, PN_3 and PN_4 respectively. In this example, PN_4 now becomes the smallest PN, and therefore the next (i.e., fifth) largest cell (i.e., cell size = 32) is allocated to it. This process is continued until the smallest cell (i.e., cell size = 3) is assigned to PN_2.

We see that the proposed algorithm balances the data load very well for this particular example. In general, we expect this algorithm to perform very well when the number of cells in the directory is relatively larger than the number of PNs. In Section 4, we will present adaptive algorithms (i.e., Split/Merge algorithms) that decide how many intervals each dimension should have. These algorithms will also remove the restriction that the linear scales are partitioned into evenly spaced intervals. Permitting different interval sizes on linear scales allows us to handle the non-uniformity in the distribution of data in the data space.

3.2 The Data Rebalancing Algorithm

As insertions and deletions occur to the local data in each PN, the size of local files may change so that gradually, a tortured distribution of file sizes will appear. This causes unbalanced disk load (i.e., data skew) to the PNs and degrades overall system performance. In this situation, a redistribution of data is necessary to resume good system performance. However, the cost of rebalancing may even be higher than the cost of processing data in a skewed circumstance. For example in Bubba's design, a simple data redistribution (was called reorganization in the original paper) algorithm was proposed. In their algorithm, the cost of redistribution is estimated before data in PNs are redistributed. If it is higher than the cost of processing data under skewed situation, the redistribution will not be performed. In other words, the system will tolerate a possibly high cost on processing skewed data simply because redistribution of the entire relation would cost higher. In their design, however, the redistribution process did not take advantage of the already balanced part of the data within PNs but reshuffle data fragments entirely. In the proposed algorithm, we minimize the cost of redistribution so that the system can rebalance its PN load before data are seriously skewed. The algorithm is given in the box titled "Data Redistribution Algorithm".

The main objective of this algorithm is to rebalance the PN loads at as little cost as possible. Basically, the algorithm first sorts the cell sizes in each PN into sorted lists. The second step is an iteration process. In each iteration, the largest PN (i.e., has the most data) is determined, and its size (i.e., the count of retained data records) is used as the basis for the other PNs to add some more of their own cells to the respective retaining lists in order to balance their loads with the largest PN. The priority for adding cells to a retaining list is to select the larger cells first. This iteration process continues until some PN runs out of cells. The remaining cells are then merged into a single sorted list and they are allocated to the PNs using the initial distribution algorithm. We see that this algorithm tries to leave as many larger cells at their current host PN as possible. Only those smaller cells are moved between PNs. The attempt to retain the larger cells to their host PNs provides us a low cost data rebalancing algorithm. This issue will be treated in more detail in Section 5.

```
Data Rebalancing Algorithm
Phase I:
  1. Each PN sorts their cell sizes into descending
     order. Let $L_i$ be the sorted list generated by
     PN$_i$: $L_i = \{\|C'_{i,1}\|, ..., \|C'_{i,k}\|\}$.
  2. $\forall i$, PN$_i$ sends $L_i$ to a designated coordinator.
Phase II (performed by the coordinator):
  1. Let $L'_i = \phi$ and sum$_i = 0$ for $1 \le i \le np$
  2. Repeat_Until $\exists i, 1 \le i \le np, \ni L_i = \phi$
       Find j such that sum$_j = \max_{1 \le i \le np}$ (sum$_i$)
       /* If there are more then one j that
       satisfy this condition, select one arbi-
       trarily. */
       For $i = 1$ to $j - 1$ and $j + 1$ to $np$ do
         Repeat_Until (sum$_i \ge$ sum$_j$) or
         ($\exists i, 1 \le i \le np, \ni L_i = \phi$)
           sum$_i$ = sum$_i$ + $\|C'_{i,1}\|$;
           $L'_i = L'_i \cup \{C'_{i,1}\}$;
           $L_i = L_i \setminus \{C'_{i,1}\}$; /* Remove $C'_{i,1}$
           from $L_i$ */
           For each $\|C'_{i,k}\| \in L_i$, rename it to
           $\|C'_{i,(k-1)}\|$;
         End_Repeat;
       End_For;
     End_Repeat;
Phase III (performed by the coordinator except
Step 3):
  1. Merge $L_i$, $1 \le i \le np$ into a single sorted list
     $LC$;
  2. $\|NP_i\|$ = sum$_i$ for $1 \le i \le np$;
  3. Apply step 5 of the initial distribution algo-
     rithm to $LC$;
```

We give an example to illustrate how the algorithm works. Shown in Figure 4 is still a two-dimensional directory with 16 cells. Let us assume that it represents the same file as in Figure 3 after some number of insertions and deletions to that file. For example, the upper-left cell originally contained 18 records. It now contains 7 records. This figure shows a data skew example in which $\|NP_2\| \ll \|NP_3\|$ (81 tuples v.s. 134 tuples respectively).

In Figure 5, we show the rebalancing process and the final result:

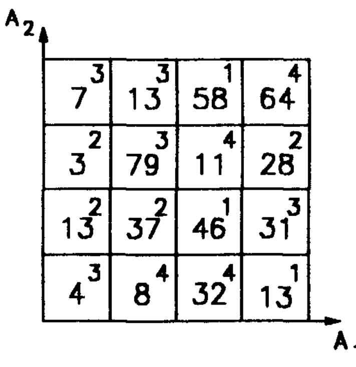

Figure 4. A data skew example

Phase I: Each PN_i sorts its cell sizes to generate the sorted lists: $L_1 = \{58, 46, 13\}$ $L_2 = \{37, 28, 13, 3\}$ $L_3 = \{79, 31, 13, 7, 4\}$ $L_4 = \{64, 32, 11, 8\}$. Each PN_i then sends its list L_i to a designated coordinator, say PN_1.

Phase II:

Iteration 1: After deciding to keep the largest cell (i.e., 79) in its host PN (i.e., PN_3), other PNs add their larger cells to their retaining list in such a way as to balance with the number of tuples in PN_3.

Iteration 2: After Iteration 1, PN_1 becomes the largest PN (it has 58 + 46 = 104 tuples). Other PNs then continue to add to their retaining lists the remaining larger cells with the attempt to balance their sizes with the current maximum (i.e., T_1 = 104). At the end of this iteration, Phase II is terminated because PN_2 runs out of cells to continue this phase.

Phase III: We apply the initial distribution algorithm to the leftover cells.

Result:

- PN_1 has 3 cells and 112 tuples.
- PN_2 has 7 cells and 114 tuples.
- PN_3 has 2 cells and 110 tuples.
- PN_4 has 4 cells and 111 tuples.

Note that the communication required during the entire redistribution process is only the transfer of sorted lists of numbers (i.e., cell sizes, not real tuples) to the designated coordinator and of some smaller data buckets (containing real tuples) among

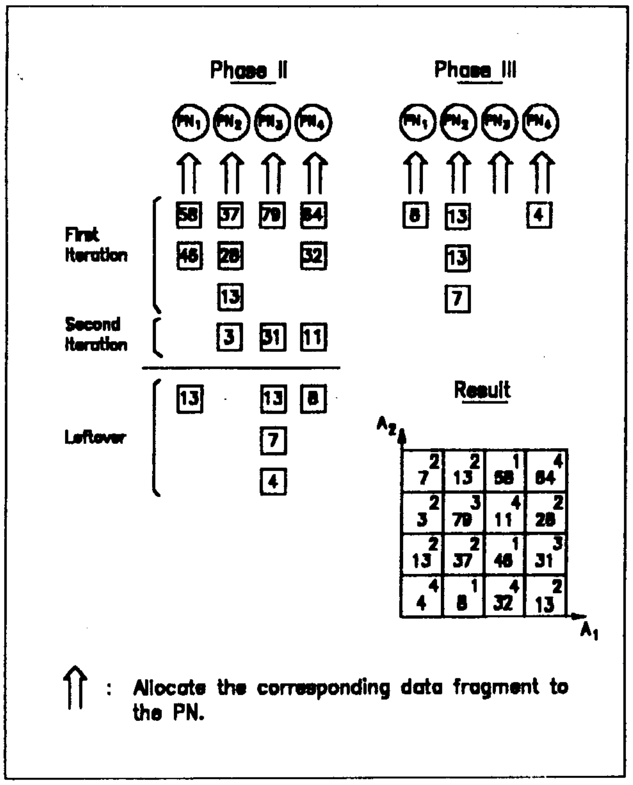

Figure 5. A data rebalancing example

PNs. As shown in Figure 5, the smaller data buckets transferred include two 13-tuple buckets transferred from PN_1 and PN_3 to PN_2, one 8-tuple bucket from PN_4 to PN_1, one 7-tuple bucket from PN_3 to PN_2, and one 4-tuple bucket from PN_3 to PN_4. Larger data buckets are retained to their host PNs.

Also note that there are a few adjustments that can be made to further reduce the redistribution cost. For instance, the coordinator for computing the redistribution process can be assigned to a PN which has the most number of cells (not neccessarily the largest PN). In this strategy, since we avoid transfering the largest sorted list (of cell size), a small performance improvement is achieved.

4. ENVIRONMENT ADAPTATION ALGORITHMS

Data distribution changes in real-world applications. There is a possible situation, although it does not occur frequently, that the data rebalancing algorithm could fail to restore the system to a load-balanced state. This would be the case when we encounter extremely nonuniform data distribution. For instance, if the largest cell as shown in Figure 4 was 279 instead of 79, it would be imposible to rebalance the load by simply transfering data fragments among PNs, since 279 is much larger than any other cells in the directory. In this situation, a split(s) on the largest cell(s) is necessary before the data rebalancing algorithm can produce a satisfactory environment for parallel processing. Unlike the original grid file [Nie84] in which a split is triggered by the overflow of a data bucket, in our environment a split is primarily a preprocessing procedure that complements the data rebalancing algorithm in the case of extremely nonuniform data distribution.

As the data distribution changes, an earlier dense area in the data space may become sparse later. For efficiency, the intervals (i.e., cross sections) split when an area is dense need to be merged when it becomes sparse. A merge algorithm is designed accordingly. Interestingly, the split and the merge algorithms can accomadate the grid partition to the environment. In the next two sections, we will present the algorithms. The following additional notations will be used in the presentation:

RIF_i :	The Relative Importance Factor of the attribute corresponding to the dimension i. For instance, RIF_i could be determined based on the frequency of reference. $\sum_{1 \le i \le d} RIF_i = 1$.
$\|C_{max}\|$:	The size of the largest cell C_{max} in the directory.
$R_{m,n}$:	The set of cells that lie in the hyper-rectangle that contains a set of cells located in the nth interval of the mth dimension. $R_{m,n} = \{C_{i1,...,id} \mid i_m = n \text{ and } 1 \le i_k \le I_k \text{ for } k \ne m\}$.
$\|R_{m,n}\|$:	The total number of tuples in the hyper-rectangle, $R_{m,n}$.
$P_{i1,...,id}$:	The identification of the PN currently assigned the cell $C_{i1,...,id}$. Note that $1 \le P_{i1,...,id} \le np$.

Also, the following binary expression will be used in the algorithms:

$$\text{Equal}\ (P_{i1,...,id}, P_{j1,...,jd}) = \begin{cases} 0, & \text{if } P_{i1,...,id} = P_{j1,...,jd} \\ 1, & \text{otherwise} \end{cases}$$

4.1 The Split Algorithm

A split strategy for grid partitioning was discussed in [Nie84] which favors some presumably more important attribute(s) by splitting the corresponding dimension(s) more often than others. Our

Split Algorithm

```
1. Find $i$ such that
   $\frac{RIF_i}{I_i} = \max_{1 \le j \le d}\left(\frac{RIF_j}{I_j}\right), \quad 1 \le i \le d$;
2. If $i$ is unique
       then select the ith dimension as the split dimension
       else the split dimension is selected as the one with highest RIF among the dimensions found in Step 1;
3. Find the largest cell $C_{max}$ in the directory;
4. In the split dimension, determine the split interval as the one in which $C_{max}$ is located;
5. Within the split interval, choose the split point so as to split $C_{max}$ into two cells of equal size;
6. In the directory, split at the split point all the cells that lie in the split interval.
```

split policy is a little more sophisticated. The proposed algorithm takes into consideration the current number of intervals (i.e., I_i) on each dimension in addition to the relative importance factors (i.e., *RIFs*) of the attributes. The split dimension i is determined as:

$$\frac{RIF_i}{I_i} = \max_{1 \le j \le d}\left(\frac{RIF_j}{I_j}\right), \quad 1 \le i \le d$$

This expression states that the split dimension should be chosen as the one that has relatively the least number of intervals for its relative importance. The complete algorithm is given in the box titled "Split Algorithm".

The computation of the split algorithm can be performed by a single PN (i.e., the coordinator). The split point information can then be broadcast to all the PNs which in turn can update their local directories (if the directory is replicated on all PNs), and split the data fragments accordingly. Alternatively, the algorithm can be executed on all PNs to avoid the communication overhead. This approach, however, ties up the PNs, impeding other useful work.

In practice, the split process should be needed rarely. The majority of nonuniform data distribution cases would involve many cells and the data placement strategy as described in Section 3 should be adequate for most situations.

4.2 The Merge Algorithm

Before we present the algorithm, the merge criteria need to be clarified. First, in order to find out which intervals on what dimension should be merged, we define

$$\frac{RIF_1}{I_1} = \frac{RIF_2}{I_2} = \cdots = \frac{RIF_d}{I_d} \tag{1}$$

This expression states that the number of intervals on a dimension should be proportional to the RIF of the corresponding attribute. We define that:

$$\|\bar{C}\| = \frac{\|R\|}{\prod_{1 \le i \le d} I_i} \tag{2}$$

where $\|\bar{C}\|$ is the ideal number of records each cell should have. For instance, $\|\bar{C}\|$ can be selected as:

$$\|\bar{C}\| = \frac{\text{disk I/O unit}}{\text{tuple size}}$$

for efficient I/O performance. Expression (2) specifies that we would like the size of each cell to be roughly $\|\bar{C}\|$. Solving expressions (1) and (2), we obtain that:

$$I_i = RIF_i \cdot \left(\frac{\|R\|}{\|\bar{C}\| \cdot \prod_{1 \le j \le d} RIF_j} \right)^{\frac{1}{d}} \tag{3}$$

This is the number of intervals we would like to have for dimension i given the relation size $\|R\|$ and the RIFs of the partition attributes.

In order to determine if two adjacent intervals need to be merged, we need to develop a threshold for the capacity of each interval in the dimension:

$$\|R_{i,n}\| \le \theta_i = \eta \times \frac{\|R\|}{I_i} \tag{4}$$

The term $\|R\|/I_i$ represents an average number of records that locate in each interval of the ith dimension. η is a coefficient. In a growing database, it is unwarranted to merge adjacent intervals as soon as possible. η should be chosen so as to allow a temporarily shrinking interval to grow back to a satisfactory size. In the related paper [Nie84], the respective η was selected to be 70%. This same number would be appropriate for our purpose. By substituting (3) into (4), we obtain:

Merge Algorithm

```
For $m = 1$ to $d$ Do
    Repeat_until $\{R_{m,n} \mid \|R_{m,n}\| + \|R_{m,n+1}\| < \theta_m\} = \phi$
        For $n = 1$ to $(I_m - 1)$ Do
            $C_n = \sum_{i_1=1}^{I_1} \cdots \sum_{i_{m-1}=1}^{I_{m-1}} \sum_{i_{m+1}=1}^{I_{m+1}} \cdots \sum_{i_d=1}^{I_d} (\, \mathrm{Equal}(P_{i_1,\ldots,i_{m-1},n,i_{m+1},\ldots,i_d}, P_{i_1,\ldots,i_{m-1},n+1,i_{m+1},\ldots,i_d})$
                $\times \min(\|C_{i_1,\ldots,i_{m-1},n,i_{m+1},\ldots,i_d}\|, \|C_{i_1,\ldots,i_{m-1},n+1,i_{m+1},\ldots,i_d}\|)\,);$
        Find min such that $C_{\min} = \min_{1 \le n \le I_m - 1} (C_n)$;
        For $n \ne \min$ and $1 \le n \le d$ and $1 \le i_n \le I_n$ Do
            If $\|C_{i_1,\ldots,i_{m-1},\min,i_{m+1},\ldots,i_d}\| \le \|C_{i_1,\ldots,i_{m-1},\min+1,i_{m+1},\ldots,i_d}\|$
                then Assign cell $C_{i_1,\ldots,i_{m-1},\min,i_{m+1},\ldots,i_d}$ to $P_{i_1,\ldots,i_{m-1},\min+1,i_{m+1},\ldots,i_d}$
                else Assign cell $C_{i_1,\ldots,i_{m-1},\min+1,i_{m+1},\ldots,i_d}$ to $P_{i_1,\ldots,i_{m-1},\min,i_{m+1},\ldots,i_d}$ ;
        For $n \ne \min$ and $1 \le n \le d$ and $1 \le i_n \le I_n$ Do
            $\|C_{i_1,\ldots,i_{m-1},\min,i_{m+1},\ldots,i_d}\| = \|C_{i_1,\ldots,i_{m-1},\min,i_{m+1},\ldots,i_d}\| + \|C_{i_1,\ldots,i_{m-1},\min+1,i_{m+1},\ldots,i_d}\|$;
        For $k = \min + 1$ to $(I_m - 1)$ Do Rename $R_{m,k}$ to $R_{m,k-1}$;
        $I_m = I_m - 1$;
    End_Repeat;
End_For;
```

$$\theta_i = \eta \cdot \frac{\|R\|^{\left(1 - \frac{1}{d}\right)} \cdot \left(\|\bar{C}\| \cdot \prod_{1 \le j \le d} RIF_j \right)^{\frac{1}{d}}}{RIF_i} \qquad (5)$$

In the merge algorithm, we merge two adjacent cross sections $R_{i,n}$ and $R_{i,n+1}$ if the sum of their number of tuples is less then θ_i (i.e., $\|R_{i,n}\| + \|R_{i,n+1}\| < \theta_i$).

The main idea of the algorithm is to find a pair of adjacent intervals that satisfy the merge condition, and whose cost to merge is the least among all merge candidates. In each iteration, a pair of such intervals is found and merged. The cost of merging two intervals is determined by the amount of data to be transferred. Therefore, for a pair of adjacent cells, if their data are currently located in the same PN, the cost will be minimum (i.e., 0). The iteration process proceeds until none of the adjacent intervals satisfy the merge condition. In the merge algorithm, although a cell may get reassigned several times to different PNs during the computation, the reassignments merely represent the states of the computation. The actual data transfer needs to be performed only once, after the merge algorithm has completed and, preferably, a data rebalancing process has been done. In other words, by comparing the images of the directory before and after the application of the merge and data rebalancing algorithms, one can determine the final destination of the data fragments. In this fashion, the partition can be refined with minimal data transfer cost.

5. AN ESTIMATION OF SAVINGS ON LOAD BALANCE

In this section, we estimate the savings of using our algorithm to rebalance the skewed data in PNs before a join operation. The estimate is obtained by calculating the load rebalance cost and subtracting it from the difference of costs of performing join on skewed data and performing join on non-skewed (uniformly distributed) data. We compute the savings for a wide range of skew degrees, and the results are compared against two simple algorithms. It is interesting that the cost of our proposed load rebalancing algorithm is so low that it can be considered as a pre-processing phase in performing a join operation, if data skew has appeared. In the following, we first present an evaluation model. Algorithms and derivation of cost functions are illustrated next. Finally, the results of comparisons and explanations are described.

5.1 Evaluation Model

In the evaluation, we compute the cost of joining two relations, R_1 and R_2. For simplicity, we assume both of the relations are partitioned on a two-dimensional 16*16 grid. Also, for ease of illustration and simplicity in computation, we assume the cells in column i are assigned to PN_i. Join is performed on the attribute on the horizontal axis. The partition intervals in this dimension for both R_1 and R_2 are exactly the same. According to this assumption, there will be no need to exchange tuples between PNs during the execution of join. Each PN will only process its private data. Note that use of any other way to assign cells to PNs could not only complicate the estimation task but also cause extra communication cost for exchanging data. This simply increases the join cost which in turn enlarges the savings of rebalancing the load before performing join operations. Therefore, these assumptions do not prejudice our case.

In the join operation, we further assume that the partitions of R_1 have data skew, but not those of R_2. Tuples of R_2 are uniformly distributed among the cells. Distribution of R_1 tuples is a step function, in which the first column (i.e., data stored in PN_1) of the grid has data skew with a degree of $\sigma \times 100\%$, where σ ($0 \leq \sigma \leq 1$) is called the degree of skew. In other words, the first column of the cells contains $\sigma \times 100\%$ more tuples than each of the other columns has. Tuples within each column are still uniformly distributed.

A set of parameters is designed for cost evaluation. The parameters are similar to those used in [Lak88]

Parameters

- **Workload Parameters**

$\|R\|$:	The relation size in tuples of both relation R_1 and R_2.
r:	The size of a tuple in bytes.
$\|C1_s\|$:	The size of each cell in tuples of the skewed (first) column in the directory of R_1.
$\|C1_u\|$:	The size of each cell in tuples of the unskewed columns in the directory of R_1.
σ:	The degree of data skew, which is defined as $\sigma = \dfrac{\|C1_s\| - \|C1_u\|}{\|C1_s\|}$

- **System Parameters**

μ:	The CPU processing rate in MIPS.
ω_{io}:	The I/O bandwidth between processor and secondary storage.
ω_{comm}:	The communication channel bandwidth between PNs.
I_{CPU}:	The CPU pathlength for processing a tuple in any step of query processing.

- **Measurement Parameters**

T_{io}:	The time cost in seconds for disk accesses.
T_{comm}:	The time cost in seconds for transferring data between PNs.
T_{CPU}:	The CPU cost in seconds for processing tuples.
$C(LC)$:	The total cost of the LC algorithm (given later).
$C(RR)$:	The total cost of the RR algorithm (given later).
$C(RH)$:	The total cost of the RH algorithm (given later).
$C_{skew}(JN)$:	The total cost of join execution on skewed data.
$C_{unif}(JN)$:	The total cost of join execution on uniformly distributed data.

The relation size is assumed to be 1 million tuples for each relation. The size of each tuple is 200 bytes. CPU MIPS of each PN is 20. The bandwidth of disk I/O is 5 MBytes/sec and that of each communication channel among PNs is 10 MBytes/sec. The CPU pathlength for processing a tuple in any step costs 500 instructions.

Finally, note that in the cost computation, the split and merge costs will not be considered in any algorithms because they are not expected to be performed frequently in a real-world application.

5.2 Derivation of Cost Functions

In this subsection, we derive cost functions of several data rebalancing algorithms. Since the cost of our proposed algorithm has been demonstrated to be low, it is abbreviated as LC (Low Cost) algorithm hereafter for the sake of convenience. We also give two other algorithms, range readjust (RR) and rehash (RH) algorithms, and derive their cost functions. The join costs with and without data skew are also computed. These costs are then used to compute the savings on performing rebalancing on skewed data.

(1) Cost of LC Algorithm.

Recall the LC algorithm presented in Section 3.2 that at the end of phase II, each PN will retain as much private data as possible and the leftover data will be distributed evenly to all the PNs. Under the skew assumption in Section 5.1, all Cl_u cells are retained in their current host PNs. Let x_1 be the number of Cl_r cells remaining unassigned at the end of phase II of the algoirthm. We then have

$$x_1 = \left\lfloor \frac{16 \times \|Cl_u\|}{\|Cl_s\|} \right\rfloor = \lfloor 16 \times (1 - \sigma) \rfloor$$

During phase III, these cells are assigned to the appropriate PNs according to step 5 of the initial distribution algorithm. Let x_2 be the number of Cl_r cells that are kept in PN_1 by phase III. We then have

$$x_2 = \left\lceil \frac{16 - x_1}{16} \right\rceil = \left\lceil 1 - \frac{\lfloor 16 \times (1 - \sigma) \rfloor}{16} \right\rceil$$

Assuming the cost of computing the LC algorithm is regligeable comparing to the disk I/O's and data transfer cost, we can derive the rebalance cost as

$$C(LC) = \max\left(\frac{(16 - (x_1 + x_2)) \times \|Cl_s\| \times r}{\omega_{io}}, \frac{(16 - (x_1 + x_2)) \times \|Cl_s\| \times r}{\omega_{comm}} \right)$$

The first term is the cost to load the tuples to be transferred, and the second term is the communication cost for transferring those tuples. Since these two operations can be performed in parallel, the maximum of them is counted. Note that the cost to store tuples back to disks in the receiving PNs is not in the expression because it is identical to the cost of loading them in the sending PN, and these two actions are performed in parallel.

(2) Cost of RR algorithm.

The range readjust algorithm is less intelligent than the LC rebalancing algorithm. In this algorithm, the rebalance is accomplished by shifting the boundaries of the ranges so that data located in each partition interval are of equal size. Since we have assumed in this particular example that data of each column in the directory are mapped to one PN, PN load will be balanced when the data sizes of the directory columns are equal. This algorithm

Range Readjust Algorithm

1. Each PN builds in parallel a local tuple distribution table which describes the number of tuples located within each small predetermined range;
2. Each PN sends its tuple distribution table to a designated coordinator;
3. After receiving the local tables from all PNs, the coordinator finds the new ranges of tuples that each PN should contain;
4. These new ranges are broadcast to all PNs;
5. Using the newly defined range information, each PN checks in parallel if any tuples need to be transferred to other PNs. If so, transfer them to appropriate PNs.

achieves that by surveying the current, unbalanced tuple distribution in each PN and readjusting the range of each PN properly. Tuples not belonging to the newly defined range need to be transferred to the proper destinating PNs. They determine the communication cost required in this RR algorithm. Note that the main difference between this algorithm and the LC algorithm is that repartitioning of tuples based on new ranges is required. This is in general an expensive step. For ease of understanding, a more detailed English-like description of the algorithm is provided in the box titled "Range Readjust Algorithm".

There are two main tasks in the algorithm. The first task is to load tuples from disk to build a distribution table in each PN. The second task is to load the tuples that need to be transferred and send these tuples to the proper PNs. We have

$$C(RR) = \frac{16 \times \|Cl_s\| \times r}{\omega_{io}} + \max\left(\frac{16 \times (\|Cl_s\| - \|\bar{C}\|) \times r}{\omega_{io}}, \frac{16 \times (\|Cl_s\| - \|\bar{C}\|) \times r}{\omega_{comm}} \right)$$

The first term represents the cost for loading tuples from disk to build the distribution table; the second term is the cost of loading the affected tuples and sending them to the proper PNs. The CPU time is ignored in this expression. The communication time for transferring the distribution table and broadcasting the new range information is also assumed to be negligible. The derivation of the expression is based on the fact that C(RR) is dictated by the bottleneck PN_1, which has skewed data.

(3) Cost of RH algorithm.

This algorithm solves the data skew problem by rehashing tuples in each PN using a new hash function. More specifically, when a rebalance of load needs to be performed, each PN will load its data and hash them into buckets using a pre-determined hash function. The number of buckets is identical to the number of PNs in the system. We assume that the hashing is perfect so that bucket sizes are equal within each PN. After the hashing stage, buckets are transferred to their corresponding PNs and the load of each PN is in perfect balance. Since hashing techniques are often seen in the literature, we presume that readers can figure out the necessary processes without a detailed algorithm being provided.

The cost of this algorithm is derived as follows.

$$C(RH) = \max(T_{io}, T_{CPU}, T_{comm})$$

$$T_{io} = \frac{16 \times \|C1_s\| \times r}{\omega_{io}}$$

$$T_{CPU} = \frac{16 \times \|C1_s\| \times I_{CPU}}{\mu}$$

$$T_{comm} = \frac{15}{16} \times \frac{\|R\| \times r}{\omega_{comm}}$$

Similar to the previous algorithms, these equations are obtained by assuming all PNs perform hash in parallel. Thus, the PN that has data skew (PN_1) causes the bottleneck and dominates the cost of the algorithm. In the equations, T_{io} is the time cost to load tuples from disk for hashing. We have assumed that hashed data are stored in a set of communication blocks, each corresponding to a PN. During the hash process, once a communication block is full, it is immediately transferred to the corresponding PN, not written back to the disk. T_{CPU} is the cost of building hash table in memory. T_{comm} is the cost to transfer buckets. Since we have assumed that this hash is a perfect randomization process, every tuple will have 1/16 possibility to be kept in the original PN; or, 15/16 possibility to be hashed to those buckets that will be transferred to other PNs. In total, therefore, there are $(15/16) \times \|R\|$ tuples transferred from their original PN to other PNs.

(4) Cost of Join under Data Skew.

Since a general hash algorithm is faster than the other (e.g., sort-merge join and nested-block join) algorithms for join operation under most conditions [Dew84], we will simply use hash-join as the algorithm in our performance study. Note that use of any slower algorithm will enlarge the savings, which is in favor of us as will be seen later, for doing load rebalancing.

As in other papers [Dew86], we separate a join operation into two phases: a hash phase and a probe phase. In the hash phase, tuples of one relation are hashed and a hash table is built. In the probe phase, tuples of another relation are used to probe the hash table. Satisfied tuples will be selected as result. These two phases are performed sequentially. All PNs, however, perform each of these two phases in parallel. The bottleneck PN is still the one having data skew. Since hash-join algorithms are well-known, a detailed description is omitted here. The join cost is computed below.

$$C_{skew}(JN) = T_{hash} + T_{probe}$$

$$T_{hash} = \max(T_{hash\_io}, T_{hash\_cpu})$$

$$T_{hash\_io} = 2 \times \left\{ \frac{16 \times \|C1_s\| \times r}{\omega_{io}} + \frac{16 \times \|C2_u\| \times r}{\omega_{io}} \right\}$$

$$T_{hash\_cpu} = \frac{16 \times \|C1_s\| \times I_{CPU} + 16 \times \|C2_u\| \times I_{CPU}}{\mu}$$

$$T_{probe} = \max(T_{probe\_io}, T_{probe\_cpu})$$

$$T_{probe\_io} = \frac{16 \times \|C1_s\| \times r}{\omega_{io}} + \frac{16 \times \|C2_u\| \times r}{\omega_{io}}$$

$$T_{probe\_cpu} = \frac{16 \times \|C1_s\| \times I_{CPU} + 16 \times \|C2_u\| \times I_{CPU}}{\mu}$$

(5) Cost of Join Without Data Skew.

In this case, since the size of each cell in the directory is equal to $\|C2_u\|$ for both relations, the cost of join can be obtained by replacing $\|C1_s\|$ of $C_{skew}(JN)$ with $\|C2_u\|$. The result is as follows.

$$C_{unif}(JN) = T_{hash} + T_{probe}$$

where "unif" stands for uniform distribution.

$$T_{hash} = \max(T_{hash\_io}, T_{hash\_cpu})$$

$$T_{hash\_io} = 2 \times \left\{ \frac{16 \times \|C2_u\| \times r}{\omega_{io}} + \frac{16 \times \|C2_u\| \times r}{\omega_{io}} \right\}$$

$$T_{hash\_cpu} = \frac{16 \times \|C2_u\| \times I_{CPU} + 16 \times \|C2_u\| \times I_{CPU}}{\mu}$$

$$T_{probe} = \max(T_{probe\_io}, T_{probe\_cpu})$$

$$T_{probe\_io} = \frac{16 \times \|C2_u\| \times r}{\omega_{io}} + \frac{16 \times \|C2_u\| \times r}{\omega_{io}}$$

$$T_{probe\_cpu} = \frac{16 \times \|C2_u\| \times I_{CPU} + 16 \times \|C2_u\| \times I_{CPU}}{\mu}$$

5.3 Comparisons And Explanation

Figure 6 shows the cost (in seconds) of performing various rebalancing algorithms versus the degree of data skew σ. The result shows that the LC algorithm is consistently the one of least cost among various algorithms. The RH curve shows a constant cost within a wide range of σ, because T_{comm} term dominates $C(RH)$ for medium and small σ's, and T_{comm} does not vary with σ. Only when the degree of skew is extremely large will the T_{io} be performed mostly in one PN so as to dominate $C(RH)$. The RR algorithm performs better than RH when σ is not too high. This is because both the T_{io} and T_{comm} in $C(RR)$ are low for small σ. When σ is high, the cost for building the distribution table in the data skew PN swamps the algorithm's performance.

Another curve shown in Figure 6 is the effect of skew, which is defined as $C_{skew}(JN) - C_{unif}(JN)$. This is the extra cost in the execution of join under data skew. Savings can be accordingly defined as follows:

$$savings = (C_{skew}(JN) - C_{unif}(JN)) - C(rebalance)$$

where $C(rebalance)$ is the cost of rebalance. There will be positive savings if a curve of rebalancing cost is under the curve of skew effect. As shown in Figure 6, the proposed LC algorithm will always provide some savings for a join operation. RR cannot provide any savings until σ is larger than 0.55. Similarly, RH can provide savings only if σ is greater than 0.76. Note that because LC algorithm consistently provides savings, it can be used as a preprocessing step before a join on skewed data.

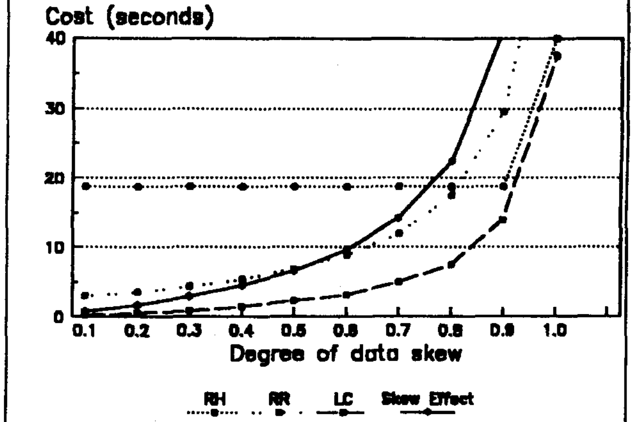

Figure 6. Cost comparison of various rebalancing strategies and skew effect on JOIN

6. CONCLUSION AND EXTENSION OF THIS WORK

In this paper, we present a data-to-processor mapping and an adaptive load-balancing scheme within a multi-processor environment. Details of the algorithms are presented. Our scheme adopts the well-known grid file concept. However, we have extended the concept to the allocation of data on processors, in addition to the originally proposed idea for data partitioning. We also propose an adaptive threshold and a way to measure the proper number of intervals in each grid dimension, under a changing environment, for grid split and merge processes. More importantly, application of our scheme is not limited to the original grid file only. We choose the traditional grid file simply for the ease of presenting our idea. Any similar grid file structures, such as [Kit89, Kri88] can be used to implement the proposed scheme.

We have also evaluated the load-rebalance costs of the proposed scheme and two other less intelligent schemes. The result shows that the proposed scheme can always provide some savings for join under any degree of data skew. This suggests that the algorithm can be used as a pre-processing phase in performing a join operation. For the other schemes, rebalancing is worth being performed only under medium or high degree of skew.

Currently, we are working on some extensions of this research. First, a more complex evaluation model for performance comparison needs to be designed to provide more accurate estimation of the savings of the rebalancing process. Secondly, we have assumed in this research that for each tuple, the probability of being accessed by queries/transactions is identical. This probability can be obtained by collecting statistical information from query/transaction processing. However, it may be more practical to collect statistics for each block of data (or a cell of tuples in our case) rather than for each tuple. An open issue is whether to dynamically balance PN load during run time by giving the access probability for each cell and the cell sizes. The issue can be further generalized to consider the multiple join case, in which rebalancing the load for each join operation may not give a better performance than only rebalancing the load for some joins (a typical local optimum vs. global optimum issue). This is due to the fact that the accumulated overhead for rebalancing processes can be high. Also, the problem can be more complex if the number of PNs available for each join is different.

REFERENCES

[Ben79] Bentley, J. L.,"Multidimensional Binary Search Trees in Database Applications", IEEE Transactions on Software Engineering, Vol. SE-5, No. 4, July 1979, pp.333-340.

[Bhi88] Bhide, A.,"An Analysis of Three Transaction Processing Architectures," Proceedings of the VLDB Conference, Los Angeles, Calif., 1988, pp.339-350.

[Cop88] Copeland, G., Alexander, W., Boughter E., and Keller, T.,"Data Placement in Bubba", Proceedings of the ACM SIGMOD Conference, Chicago, Illinois, June 1988, pp.99-108.

[Dew84] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D.,"Implementation Techniques for Main Memory Database Systems," Proceedings of ACM SIGMOD Conference, 1984, pp.1-8.

[Dew86] DeWitt, D. J. et al,"GAMMA: A Performance Dataflow Database Machine," Proceedings of the International Conference on VLDB, August 1986, pp.228-237.

[Kit84] Kitsuregawa, M., Tanaka, H., and Moto-oka, T.,"Architecture and performance of Relational Algebra Machine GRACE," Proceedings of the International Conference on Parallel Processing, Chicago, Illinois, August, 1984.

[Kit89] Kitsuregawa, M., Harada, L., and Takagi, M.,"Join Strategies on KD-Tree Indexed Relations", Proceedings of the International Conference on Data Engineering, Los Angeles, Calif., Feb. 1989, pp.85-93.

[Kri88] Kriegel, H. P. and Seeger, B.,"PLOP-Hashing: A Grid File without Directory", Proceedings of the International Conference on Data Engineering, 1988, pp.369-376.

[Lak88] Lakshmi, M. S. and Yu, P. S.,"Effect of Skew on Join Performance in Parallel Architectures," Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, Austin, Texas, 1988.

[Nie84] Nievergelt, J., Hinterberger, H., and Sevcik, K. C.,"The Grid File: An Adaptable Symmetric Multikey File Structure", ACM Transactions on Database Systems, Vol. 9, No. 1, March 1984, pp.38-71.

[Ozk88] Oskarahan, E. A. and Bozsahin, C. H.,"Join Strategies Using Data Space Partitioning", New Generation Computing, Vol. 6, No. 1, 1988, pp.19-39.

[Soc79] Sockut, G. H. and Goldberg, R. P.,"Database Reorganization - Principles and Practices," ACM Computing Surveys, Vol. 11, No. 4, Dec. 1979.

[Su83] Su, S. Y. W.,"A Microcomputer Network System for Distributed Relational Databases: Design, Implementation, and Analysis," Journal of Telecommunication Networks, Vol. 2, No. 3, 1983, pp.307-334.

[Ter88] Teradata Corporation,"DBC/1012 Data Base Computer Concepts and Facilities," Teradata Document C02-0001-05, Los Angeles, Calif., 1988.