Towards Self-Tuning Data Placement in Parallel Database Systems

Mong Li Lee[1]* Masaru Kitsuregawa[2] Beng Chin Ooi[1]†

Kian-Lee Tan[1]† Anirban Mondal[1]

[1] Department of Computer Science
National University of Singapore, SINGAPORE
{leeml,ooibc,tankl,anirbanm}@comp.nus.edu.sg

[2] Institute of Industrial Science
University of Tokyo, JAPAN
kitsure@tkl.iis.u-tokyo.ac.jp

ABSTRACT

Parallel database systems are increasingly being deployed to support the performance demands of end-users. While declustering data across multiple nodes facilitates parallelism, existing data placement may no longer be optimal due to skewed workloads and changing access patterns. To prevent performance degradation, the placement of data must be reorganized, and this must be done on-line to minimize disruption to the system.

In this paper, we consider a dynamic self-tuning approach to reorganization in a shared nothing system. We introduce a new index-based method that faciliates fast and efficient migration of data. Our solution incorporates a globally height-balanced structure and load tracking at different levels of granularity. We conducted an extensive performance study, and implemented the methods on the Fujitsu AP3000 machine. Both the simulation and empirical results demonstrate that our proposed method is indeed scalable and effective in correcting any deterioration in system throughput.

1. INTRODUCTION

Given the explosive growth of data on the Internet and prevalence of web-based applications such as e-commerce, the issues of managing huge volumes of data and providing fast and timely answers to queries have assumed paramount importance. The WWW is a dynamic environment where the number of users grows rapidly and changing access patterns may exhibit high skew. Web-sites of stock trading database and many other data-intensive applications which are inherently dynamic in nature have unpredictable workload patterns: they may see heavy access to some particular blocks of data just yesterday, but has low access frequency today.

Shared-nothing parallel computing infrastructure such as clusters of WWW servers and network of workstations [2, 11, 15] has become increasingly widespread because they are built from high performance, low-cost commodity hardware. The availability and scalability of these systems are important for entrepreneurs in the e-commerce business as it implies that they can build their system gradually depending upon the workload. Such a computing cluster comprises a number of processing elements (PEs), each of which has its own memory and disk. Data is typically partitioned across all the PEs to exploit the I/O bandwidth of the PEs. To further facilitate efficient query and update evaluation, data at each PE is indexed.

While the initial data placement can be effective for static databases, changes in access patterns can cause the performance of the system to degrade rapidly as some PEs become bottlenecks. For a system to be responsive to swings in query patterns and to better utilize all resources, dynamic re-balancing of the workload among the PEs is necessary. This can be achieved by tuning the data placement in the various PEs.

Reorganization of data has been extensively studied in centralized database systems by both researchers and major DBMS providers. [18] investigated how to compact a primary B^+-tree which had become sparse and and [19] described a method for defering secondary index updates. [8] examined the problems of changing from one access method to another, such as from B^+-tree to linear hashing and [9] studied how indexed sequential files can be compacted and clustering organizations described by hypergraphs can be created. [14] detailed methods to increase concurrency during the restoration of clustering indexes in IBM's DB2. [7] described restartable algorithms for online construction of an index.

In parallel database systems, new challenges in online reoganization arise as data and indexes are partitioned across multiple disks, and load imbalance occurs when access patterns change. [12, 13] presented various file striping heuristics for data allocation, data redistribution and load balancing in a shared memory multiprocessing environment. [16] showed

*Contact author is currently a visiting faculty at the University of Wisconsin-Madison.

†The work of these two authors were resulted from the exchange programme funded by the Japan Society for the Promotion of Science (JSPS).

SIGMOD '2000 Dallas, Texas USA

how records can be distributed into variable sized fragments and migrated when load imbalance occurs in a shared nothing system. These works involve a significant amount of sophisticated bookkeeping.

Indexes such as B-trees in a multiprocessor environment is typically replicated in all the PEs to provide fast response to queries. However, this has a very serious drawback during reorganization and database updates. Any changes and underflow/overflow in a B-tree has to be propagated to all copies of the B-tree. [4, 3, 5, 6] proposed various techniques aimed at reducing the cost of maintaining replicated search structures and increasing concurrency.

[1] studied two methods to maintain indexes during on-line reorganization: OAT (One-At-a-Time page movement) moves one data page at a time from the source PE to the destination PE and modifies the indexes for records in that page, while BULK (bulk page movement) copies all the data to be moved at the destination PE and then modifies the indexes at the source and destination PEs. In both cases, the conventional B^+-tree insertion algorithm is used to insert the keys into the index in the destination PE. Similarly, the conventional B^+-tree deletion algorithm is used to delete the keys of the migrated data from the index in the source PE.

It is important to maintain a good balance between overheads incurred by data migrations such as communication, concurrency, index maintenance, and improvements in system throughput as a result of the migrations. This calls for efficient index updates and incremental data migration as overheads and heavy data movement may have an adverse effect on system throughput and cause the destination PE to become the next bottleneck.

In this paper, we consider a unique self-tuning approach to data reorganization in parallel database systems. Our work differs from previous research in that we use a two-tier index structure to facilitate fast and efficient data access and migration in a cluster. The novelty of this strategy is four-fold.

1. The amount of data to migrate is obtained from branches of the index at the source PE. This allows the entirety of the branches to be pruned easily without excessive overhead. The granularity of migrated data can be dynamically fine-tuned by using branches at different levels of the index.

2. The migrated data is bulkloaded into a separate rooted tree at the destination PE instead of inserting the migrated data one record at a time. This not only speeds up the insertion time, it also allows the rooted tree to be easily "attached" to the index at the destination PE. Data availability is also maximized.

3. An immediate cost reduction occurs even though the fast detachment and re-attachment of branches only applies to the primary index, and conventional B^+-tree insertions and deletions has to be used for the secondary indexes. This is because index modification is a major overhead in data migration, especially when we have multiple indexes on a relation.

4. Further reduction in migration overheads is achieved when the index structure at all the PEs is of the same height. We introduce an adaptive B^+-tree called the *aB^+-tree* which maintains the global height-balanced property of indexes in all the PEs by allowing some indexes to grow "fatter" than normal, while others are kept "lean". Algorithms to search and update the aB^+-tree are also given.

We perform an extensive performance study on the proposed strategy. Our result shows that the proposed strategy is scalable and effective in correcting any workload skews. We also confirm the results by implementing the techniques on the Fujitsu AP3000 machine.

The remainder of this paper is organized as follows. In Section 2, we present our dynamic self-tuning data placement strategy. Section 3 gives the design of the aB^+-tree. Section 4 presents the experimental study and implementation and reports our findings. Finally, we conclude in Section 5.

2. DATA PLACEMENT AND MIGRATION

Data is initially range partitioned across all the PEs. Range partitioning is superior to round-robin and hashing as it can support range queries efficiently in addition to exact match queries. Unfortunately, it can lead to *data skew* where certain values for an attribute occurs more frequently than other values. This causes PEs dealing with large partitions of data to become performance bottlenecks. Similarly, while a shared-nothing system is scalable, *load skew* or load imbalance can occur when access patterns change leading to queries or updates on certain values for an attribute to occur more frequently. This causes PEs with frequently queried or updated data to become "hot" spots. Data skew or load skew can result in disk or processor becoming completely utilized for a small number of PEs, while the disks or processors of the other PEs are only lightly utilized. While data and load skews are inevitable, reducing these skews can increase throughput and reduce response time.

In this section, we present an efficient mechanism for on-line reorganization in a shared nothing context where data is indexed. We employ a two-tier index structure as the basic indexing mechanism. The first tier directs the search to the PE where the data is stored. Since the data is range partitioned, this layer is essentially a partitioning vector with $n-1$ values and n "pointers" for a system of n PEs. We can expect this layer to take up not more than a few pages (even for a system of 1000 PEs), and hence can be easily cached in main memory for fast access. Furthermore, this layer is replicated across all PEs to ensure that there is no central PE through which retrievals and updates requests must pass. Otherwise, the PE containing the layer will easily become a bottleneck. Node replication reduces contention but requires a coherence protocol to maintain consistency. However, the maintenance of copies of the layer will hardly be required, if at all, since the layer is often read, but rarely updated. The second tier is a collection of B^+-trees, one at each PE. Each B^+-tree independently indexes the data at its PE. Thus, though the B^+-tree is a balanced structure, the two-tier structure need not be height balanced (since the number of records at the PEs can be different).

2.1 Illustrating Examples

We shall first illustrate with examples the proposed strategy. As data is range partitioned, we can only move data from one PE to its neighbouring PEs. Here, a neighbouring PE refers to the PE with the immediate preceding or succeeding range. So, all but two PEs will have two neighbours; the PE with the starting value, and the PE with the ending value will have only one neighbour. Given this non-overlapping data partitioning and hence non-overlapping indexes, our approach to on-line data reorganization is to migrate the data indexed by a branch of the B$^+$-tree in an overloaded PE and insert it in the destination PE index by bulkloading [10]. Note that the detachment of a branch from the B$^+$-tree in the source PE requires one pointer update. After bulkloading the migrated data into a B$^+$ subtree, the attachment of the subtree to the B$^+$-tree at the destination PE also requires only one pointer update.

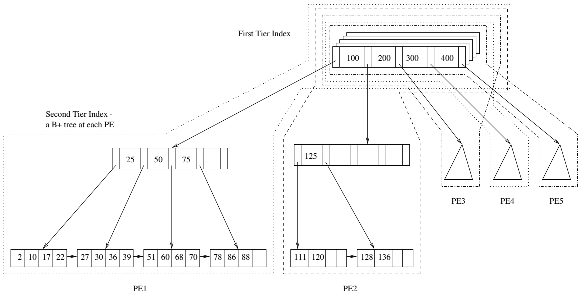


Figure 1: A sample 'global' index structure for illustration.

Consider the example 'global' index structure shown in Figure 1. Suppose the key attribute value ranges from 1 to 500 and we have 5 PEs. Moreover, assume that the records are initially range partitioned on the key attribute across the 5 PEs such that PE i is allocated the range $[(i{-}1)*100{+}1, i*100]$. From the figure, we note that there is an obvious data skew in PE 1 while PE 2 is relatively sparsely populated. The data skew in PE 1 also increases the chances of more queries and updates being directed to PE 1 since more records reside there. In order to resolve the data skew, we dynamically move one or more branches from the B$^+$-tree which has more records (the B$^+$-tree in PE 1 in our example) to the neighbouring B$^+$-tree (the B$^+$-tree in PE 2) which has relatively fewer records. Figure 2 shows the index structure after removing the data skew in PE 1. The range of the B$^+$-tree in PE 1 is now narrower with the last key entry *75* replacing the first key entry *100* in the first tier index. The latter now becomes the first key entry in the B$^+$-tree in PE 2.

Next, suppose there are 10000 searches to be performed. Ideally, if there is no data skew or any data skew has been dealt with, then an average of 2000 searches would be directed to each PE. However, we may have an exceptionally large number of searches for records whose keys fall in a certain range, say 0-75. This query skew may cause PE 1 to receive say 3000 queries, which is 50% more than the average load. The B$^+$-tree in PE 1 will be accessed 50% more than the rest of the B$^+$-trees in the index structure and can easily become a bottleneck if the inter-arrival time of the queries is less than the time needed to process a query[1]. In order to resolve load skew, we again dynamically move one or more branches from the B$^+$-tree which is more heavily accessed (the B$^+$-tree in PE 1 in our example) to the B$^+$-tree in the neighbouring PE (the B$^+$-tree in PE 2) which has relatively fewer queries (Figure 3). The ranges in the two B$^+$-trees are again adjusted accordingly. We observe that the B$^+$-tree in PE 1 is now "slimmer" while the B$^+$-tree in PE 2 is now "fatter". Note that what we want to achieve is to be able to "take away" one branch from the B$^+$-tree in PE 1, and "pluck it" into the B$^+$-tree in PE 2 without much complexity. In addition, there is minimal disruption as the B$^+$-trees in PE 1 and PE 2 continue to process queries during the migration period.

[1]Note that when the inter-arrival time of the queries are far apart, it may be argued that there is no need to "balance" the load, since the response time would be the same anyway. However, this is not true when the system serves multiple users and multiple applications. If a single application accesses some PE more often than others, then this may lead to a load imbalance in the overall system.

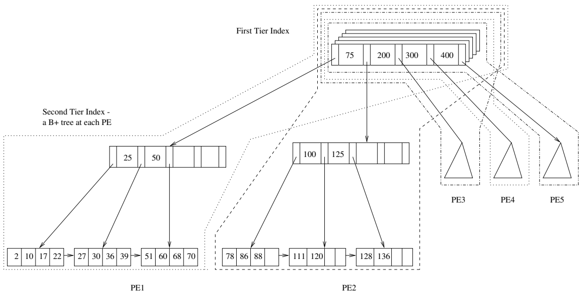


Figure 2: Resultant index structure after migration of data from Node 1 to Node 2 to remove data skew in Node 1.

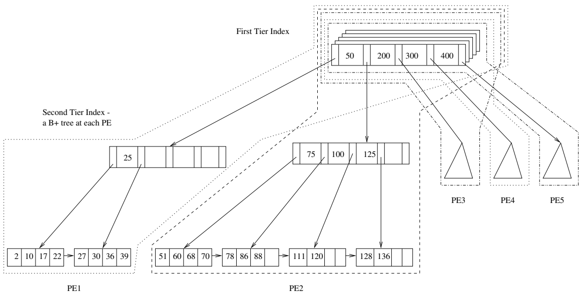


Figure 3: Resultant index structure after migration of data from Node 1 to Node 2 to remove load skew in Node 1.

The migration of branches in the B$^+$-trees also requires that the index entries in the first tier node copies to be updated. While the tier 1 entries at the source and destination PEs are updated in the process of the migration, the other copies at other PEs are updated in a lazy manner by piggy-backing update messages onto messages used for other purposes (such as during migration). The 2-tier index structure remains usable even when some copies of tier 1 have not been updated. Using our example, suppose PE 4 receives a request to retrieve the record with key value 60 after the rightmost branch in PE 1's B$^+$-tree has been moved to the B$^+$-tree in PE 2. Suppose the tier 1 copies at PEs 1 and 2 are updated while that for PEs 3, 4 and 5 have not been updated. Therefore, the search will be directed to PE 1. But, at PE 1, the system will automatically re-direct the search to continue in the B$^+$-tree in its right neighbour (PE 2) since its tier 1 entries indicates that the record can be found in PE 2.

2.2 Tuning Strategies

Several issues need to be addressed when supporting data migration in response to changing access patterns in shared nothing systems. Among these are the initiation of reorganization when load imbalance occurs, determination of the amount of data to be migrated from the overloaded PE, and efficient integration of the migrated data in the destination PE. We present our solutions to these issues here.

1. **Initiation of Data Migration:**

   We can choose to initiate data migration when the load, or response time, or the number of jobs in the queue of a PE exceed a certain threshold. In a centralized approach, a control PE periodically polls every PE for their workload statistics. The control PE will then determine if there is any imbalance in the load or response time among the PEs and trigger the migration of data from the "hot" PE to its neighbouring PEs. This approach has better control when multiple nodes are overloaded. In this case, the most overloaded node is picked for data migration first. Only upon its completion then will the next overloaded node be considered (if it is still overloaded then). A more scalable approach, however, is to use distributed data balancing where a PE determines that it is overloaded and checks its left and right neighbours' loads. For simplicity, we have adopted a centralized approach for the initiation of data migration.

2. **Determination of Amount of Data to Migrate:**

   When reorganization is initiated, it is necessary to determine the amount of data to migrate from the overloaded node. One key factor we have in mind is that of efficiency: the amount of the data should be determined quickly, and should facilitate efficient update to the index structure. Thus, we propose that the amount of data be obtained from subtrees in the index structure. This is efficient as removing the subtrees to be migrated requires only a simple pointer update.

   Our approach is a top-down adaptive strategy: at the root, we determine the number of subtrees to be migrated; if a certain subtree's accesses are too large, we can move down to the next level, and repeat the process there. In order to achieve this, we need to maintain statistics on the access pattern. In this paper, we employ a straightforward and practical way to keep only the number of accesses to each PE. Given this minimal information, we adopt the assumption that the accesses are evenly distributed across all subtrees of the root node at the PE. This assumption is recursively applied to the subtrees at each node, i.e., at any node, all accesses are assumed to be directed evenly at its subtrees. We note that to maintain the property of B^+-tree that each node be at 50% utilized, if the amount of data obtained leads to a node falling below the utilization, then the entirety of the node will be transmitted.

   However, keeping minimal information may not be sufficient for workloads that are skewed towards some subtrees. This may call for detailed statistics to be maintained on the accesses for every level of the B^+-tree (or even nodes or individual records). One can then obtain a fairly exact amount to migrate, but the overhead of maintaining the statistics and updating them can be very costly.

3. **Integration of Migrated Data:**

   Without loss of generality, assume that we are moving data from PE p to PE q. Let us refer to the B^+-trees at the two PEs as pB^+-tree and qB^+-tree. Traditionally, the migrated data are inserted one at a time into qB^+-tree. This can be inefficient especially if the number of records moved is large. To speed up the process, we exploit the concept of bulkloading. We bulkload the migrated data into a newly created B^+-tree at q. We shall refer to this as the newB^+-tree. The idea is to try to build newB^+-tree such that its height is the same as that at certain level of qB^+-tree. The newB^+-tree can then be easily integrated into the qB^+-tree since the range of key values in the newB^+-tree is always smaller (or larger) than the range of key values in qB^+-tree. During this migration period, the pB^+-tree remains usable as the newB^+-tree is being built in PE q. Like the detachment process, the attachment of the newB^+-tree to the qB^+-tree is essentially a pointer update.

   To realize the proposed mechanism, we need to determine a suitable height for newB^+-tree. We present our approach here. Let the height of a branch in pB^+-tree that has been picked for migration be pH, and the height of qB^+-tree be qH. There are two cases to consider:

   - pH $\leq$ qH. In this case, for the migrated branch of pB^+-tree, the corresponding newB^+-tree will be constructed to be of the same height.
   - pH $>$ qH. For a B^+-tree of order d (and maximum of $2d$ entries), the minimum and maximum number of records to construct a tree of height qH are $2d^{qH-1}$ and $(2d)^{qH}$ respectively. Let the number of records moved to q be N. We adopt the following heuristics: we will construct k branches of height qH with minimum number of records ($k \geq 1$), and the remaining records are evenly allocated to these k branches, i.e., each of the k branches have such number of records as given by the expression

     $$2d^{qH-1} + \frac{N - k \cdot 2d^{qH-1}}{k}$$

Figures 4 and 5 give the algorithms for a branch migration between the source and destination PEs initiated by load imbalance. For simplicity, we only show the case when pH = qH and when the migration involves one branch of the tree. In Figure 4, the algorithm determines the source and destination PEs for the migration (if the load exceeds a certain threshold at the source PE, say 10-20% above the average load of the PEs in the system). The required data (and keys) are then extracted (using routine *extract_keys*), and transmitted to the destination PE (using routine *transmit*). The data and the corresponding branch can then be pruned (using routine *delete_branch*). In Figure 5, the migrated data is first bulkloaded to a tree of the appropriate height (routine *bulkload*). The tree can then be integrated into the existing index structure, and the corresponding separators updated accordingly.

Before we leave this section, we note that migration can wrap around the PEs by allowing the first PE to contain two ranges. Suppose we have 5 PEs with the following key ranges: PE 1 is assigned 1-20, PE 2 21-40, PE 3 41-60, PE 4 61-80 and PE 5 81-100. If both PEs 4 and 5 are both overloaded, then we have the flexibility to migrate data with keys say, ranging from 91 to 100 to PE 1. In this case, PE 1 will have two key ranges, 91-100 and 1-20. At the same time, we can also achieve a smoother load distribution among the PEs by cascading the migration from the most heavily loaded node to the least loaded node which can be several nodes away (*Ripple* migration strategy). For example, PE 4 transfers a branch to PE 3, which in turn transfers a branch to PE 2, which in turn transfers a branch to PE 1. In this way, we can also get a better spread of the load across the PEs. Note that we can schedule the migrations to minimize network congestion.

```
Algorithm remove_branch()
/* Find the PE with the heaviest load */
PE: an array that records load and index information in each PE;
source = 0;
/* Determine the source PE with heaviest load */
for (i=1; i < NUM_PE; i++)
    if (PE[i].Load > PE[source].Load)
      source = i;
if (PE[source].Load > THRESHOLD) {
/* Determine the destination PE */
  if (source == NUM_PE - 1) destination = source -1;
  else if (destination == 0) source = 1;
        else if (PE[source+1].Load > PE[source-1].Load)
               destination = source - 1;
               else destination = source + 1;

  if (destination > source) {
  /* Extract all the keys indexed by the rightmost pointer P_m in the */
  /* B+-tree of the overloaded PE (source) and transmit them to the PE */
  /* on the right (destination). The branch pointed to by P_m is deleted. */
    Keys = extract_keys (PE[source].Root→ P_m);
    transmit (destination, add_branch(Keys, 1));
    delete_branch (PE[source].Root→ P_m);
  }
  else {
  /* Similarly, extract the keys indexed by the leftmost pointer P_0 and */
  /* transmit them to the PE on the left. The index entries are shifted */
  /* one place to the left after deleting the branch pointed to by P_0. */
    Keys = extract_keys (PE[source].Root→ P_0);
    transmit (destination, add_branch(Keys, 0));
    delete_branch (PE[source].Root→ P_0);
    for (i=0; i≤m; i++) {
       PE[source].Root→ P_i = PE[source].Root→ P_{i+1};
       PE[source].Root→ K_i = PE[source].Root→ K_{i+1};
    }
    PE[source].Root→ P_{m-1} = PE[source].Root→ P_m
  }
}
end
```

Figure 4: Algorithm for initiating data migration by detaching a branch from the B$^+$-tree in the source PE

3. AB+-TREE: THE ADAPTIVE B+-TREE

From the last section, we observe that when the heights of the B$^+$-trees at the source and destination PEs are the same, migrating a branch from the source PE to the destination PE is an easy task: the migrated branch is reconstructed to be of the same height, and attached to the destination PE. Furthermore, there is no need to maintain any additional statistics. The adaptive B$^+$-tree (aB$^+$-tree) index structure is another two-tier index structure designed to take advantage of this. The structure has two nice properties. First, it is globally height balanced without requiring the PEs to contain approximately the same number of records. Second, the tree is able to exploit the bulkloading mechanism discussed in the previous section without the cost of maintaining additional statistics.

```
Algorithm add_branch (Keys, Right)
/* Given a set of keys, construct a B+-tree by bulkloading and attach it */
/* to the B+-tree in the destination PE. This requires finding the */
/* separator to be inserted in the root node of the destination index. */
P_new = bulk_load (Keys);
Let n be the number of index entries in the root node of the
B+-tree atdestination;
if (Right) {
  PE[destination].Root→ P_{n+1} = PE[destination].Root→ P_n;
  for (i=n-1; i≥0; i- -) {
      PE[destination].Root→ P_{i+1} = PE[destination].Root→ P_i;
      PE[destination].Root→ K_{i+1} = PE[destination].Root→ K_i;
  }
  PE[destination].Root→ P_0 = P_new;
  PE[destination].Root→ K_0 = find_separator();
}
else {
    PE[destination].Root→ K_n = find_separator();
    PE[destination].Root→ P_{n+1} = P_new;
}
end
```

Figure 5: Algorithm for attaching a branch to the B$^+$-tree in the destination PE

The first tier of the aB$^+$-tree is the same as that of the basic 2-tier structure discussed in the previous section. In the second tier, each PE has a variation of B$^+$-tree that indexes the data at the PE. In our variant B$^+$-tree, the root node can be a "fat" node, i.e., for a B$^+$-tree of order d (and maximum of $2d$ entries), the root node can contain more than $2d$ entries[2]. Furthermore, all the B$^+$-trees across all PEs are of the same height. To ensure this, the height is essentially determined by the PE with the fewest number of records. For PEs with more records, the root of the B$^+$-trees may therefore contain more than $2d$ entries in order to keep the height the same across all PEs.

3.1 Insert Algorithm

Inserting a new record into the aB$^+$-tree involves determining the PE to store the record (by searching the first tier), and inserting the record into the corresponding B$^+$-tree. The algorithm is similar to that of B$^+$-tree conventional insertion algorithm except that we have to determine when the tree grows (since all the B$^+$-trees in the PE must be of the same height). To preserve height balancing, all the B$^+$-trees in the system will grow together. This is done using

[2]Another way of looking at this is that each PE contains multiple B$^+$-trees of the same height.

the following mechanism. When a PE's B$^+$-tree root node (recall that root node is a fat node) is full, it will check to see whether all the B$^+$-trees root nodes at other PE contain more than $2d$ entries. Note that this can be achieved by maintaining statistics at each PE, rather than communicating with every PE during runtime. If some PEs' B$^+$-tree root node contains fewer than $2d$ entries, it means that aB$^+$-tree is not ready to grow, and an additional page is assigned as part of the fat node. On the other hand, when all the PEs' root nodes contain more than $2d$ entries, each of them will be split and a new root node will be allocated. The height of each tree will increase by one at the same time. Note that the first tier is unaffected by the growth.

We observe that it is possible that some PEs contain a lot more records than others. Thus, the roots of the B$^+$-trees in these PEs are "fat" and contain many pages while others may contain only one page. This is not a critical issue for two reasons. First, such extreme case is not expected to be common in practice. Second, since the fat root node is the root of the B$^+$-tree, it is not expected to be very large, i.e., the fat root node can be kept memory resident.

3.2 Search Algorithm

Given an exact match query at any PE, the first tier of the aB$^+$-tree is accessed to determine which PE, say p_i, to go to next. The query is passed to the p_i and the B$^+$-tree there is traversed to retrieve the required record. Many such queries can be processed by the processors concurrently as different B$^+$-trees are traversed. Figure 6 gives the details of the search algorithm for exact match queries.

For a range query issued at a PE, we can determine the set of PEs whose dataset satisfy the query. This can be done easily by examining the first tier of the aB$^+$-tree. The query can then be channelled to all the candidate PEs to return the portion of data that is stored there. Figure 7 shows how our range search is conducted.

```
Algorithm search(K)
K : search key value;
/* Given a search key value, search the first tier of the aB+-tree */
/* to find the PE that contains the record. */
/* Search will be subsequently continued in that node. */
i = get_PE (K);
if i < 0 then abort
/* search_tree is a conventional B+-tree search routine */
transmit (i, search_tree(K));
/* Receive the result from PE i */
receive (i, Record);
return Record;
end
```

Figure 6: Algorithm for exact match query

3.3 Deletion Algorithm

Deleting a record involves searching for the record (using the search algorithm), and then deleting the record as in the traditional B$^+$-tree. However, it is possible that the deleted record causes an underflow that results in the B$^+$-tree shrinking in height. We address this problem as follows. We will first try to initiate data migration in its neighbouring PE to "donate" some branches to it. This minimizes the need to "shrink" the trees. In the event that this is not possible (because the neighbours will underflow and shrink too if data is taken from them), then we will proceed with a global shrinking process in order to maintain global height-balance. In other words, when a tree shrink, all trees will also shrink. As a result of the shrinking, some B$^+$-trees will become fat. The algorithm is essentially similar to the traditional B$^+$-tree deletion, where entries from all nodes are concatenated, together with the separators at the parent node being pulled down.

```
Algorithm range_search(K1, K2)
K1, K2: the range values of the range query;
Result : list of records returned to the calling routine;
/* Find all the PE that may contain records falling in the range [K1, K2] . */
Result = ∅;
for (i=0; i < NUM_PE; i++)
    if (range of data at PE[i] intersects range [K1, K2]) then
    /* Btree_range_search is a conventional B+-tree rangesearch routine */
        transmit(i, Btree_range_search(K1, K2);
        receive (i, List);
        Result = Result ∪ List;
return Result;
end
```

Figure 7: Algorithm for range query

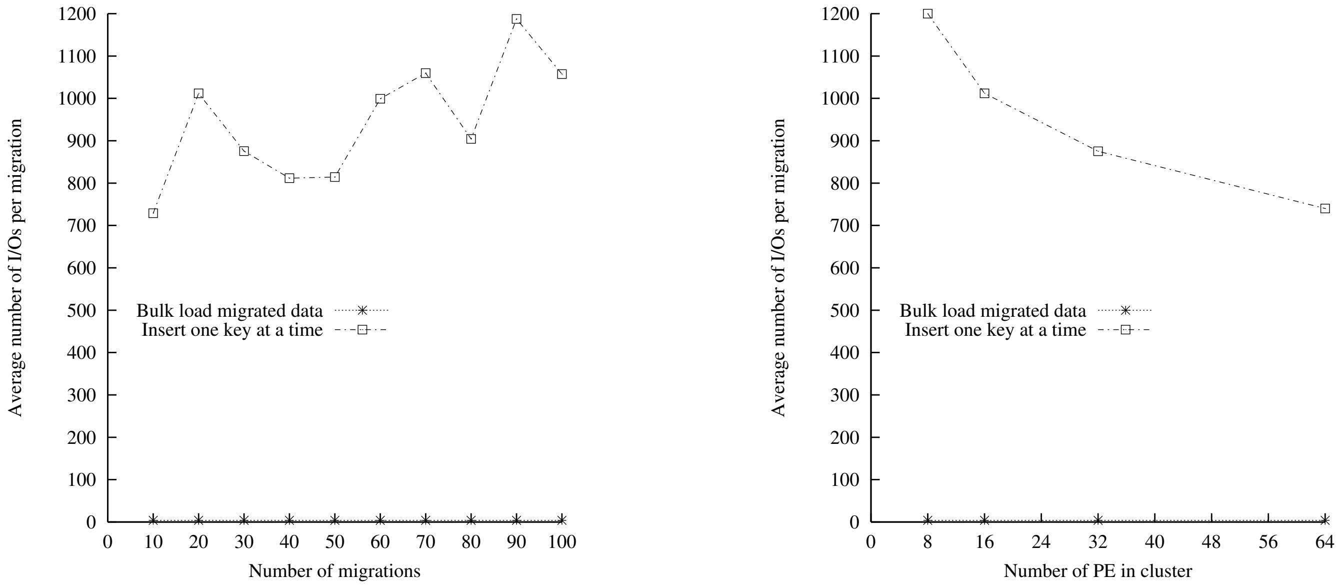
4. PERFORMANCE STUDIES

In this section, we describe our experiments to study the performance of our self-tuning data placement strategy using the proposed aB$^+$-tree. Our performance evaluation consists of both simulation (with an actual implementation of aB+-tree) and implementation on the Fujitsu AP3000 machine. The simulation study allows us to perform sensitvity analysis which we are unable to do on the Fujitsu AP3000 machine (because of the limited number of processors and disk space allocated for our experiments).

For the simulation study, the metrics used are the impact on the response time of the transactions and the load directed to the PEs. We examine the costs of our reorganization strategy in terms of the number of pages accessed. We note that the number of messages generated to update copies of the first tier index will definitely be fewer than existing replicated index structures. As such, we do not study this metric in our experiments. We use a shared nothing parallel database architecture where each PE consists of a processor with its own disk(s) and memory. The PEs in the system communicate with each other by exchanging messages across the interconnection network, set at 100 Mbit per second. Table 1 summarizes the parameters and their values used in our experiments.

The simulation experiments comprise two phases:

1. Phase 1.
   We first create an initial aB$^+$-tree with the tuple key values generated using a uniform random distribution. The B$^+$-trees in the second tier are distributed to the PEs. Then we generate 10000 queries using a zipf distribution which concentrates the queries in a narrow key range. Therefore, we have about 40% of the queries directed to a "hot" PE. This load skew will initiate the migration of branches in the "hot" PE to its neighbouring PEs. Creating an actual aB$^+$-tree given



(a) A 16-PE cluster

(b) Effect of varying the number of PEs.

Figure 8: Cost of migration

Parameter	Default	Variations
System Parameters		
index node size	4K	
number of PEs in the cluster	16	8, 32, 64
network bandwidth (in Mbyte/s)	200	
Database Parameters		
number of records (in million)	1	0.5, 2.5, 5
size of key	4 bytes	
time to read or write a page	15 ms	
interarrival time is exponential with mean $1/\lambda$	10	5, 15, 20 25, 30, 40
Query Parameters		
number of queries	10000	
distribution of queries using zipf distribution, zipf factor	0.1	

Table 1: Parameters and their values.

the number of PEs and relation size allows us to know the the actual number of keys migrated and their key range values when a branch is detached from the B$^+$-tree in the source PE and attached to the B$^+$-tree in the destination PE. This information is captured at each migration and used in the second phase.

2. Phase 2.
   Here, we use the simulation package, CSIM [17], which easily allows us to measure the response time of the queries and the number of queries waiting in the queue. We model each of the PEs as a resource and the queries as entities. We use the same 10000 queries generated using the zipf distribution. The migration of a branch in a "hot" PE to its neighbouring PE is simulated by adjusting the range of key values indexed by the B$^+$-trees in the source and destination PEs. This is possible with the trace obtained in the first phase of our study.

4.1 Cost of Migration

In this section, we first evaluate the cost of migration using our reorganization technique. As reference, we compare our technique with the traditional technique of inserting the keys of the migrated data one at a time. Here, we consider the number of page accesses when data is migrated. This metric tracks the number of index pages accessed when the B$^+$-trees in the source and destination PEs have to be modified due to data migration. The results are shown in Figure 8. The average number of IOs per migration for the "Insert one key at time" approach fluctuates with the amount of data indexed by the branch to be migrated. It is clear that the traditional method of deleting the keys of the migrated data from the source PE index and inserting these keys to the destination PE index is very expensive because each key requires us to start from the root and go down to the appropriate leaf page. For this experiment, we did not use any buffer replacement strategy because we want to study the effect of limited buffers and to get the true costs of these techniques. We expect the costs of the two methods to be comparable if sufficient buffers are available because the index nodes are likely to stay in the buffer pool between successive insertions and deletions. In contrast, the number of page accesses required for the proposed method is low and relatively constant even when buffering is minimal. Only the root nodes of the indexes in the source and destination PEs are accessed to update the pointers when the data indexed by a branch of the B$^+$-tree is migrated.

4.2 Impact of Migration on Maximum Load

In this set of experiments, we study the effect of data migration on the maximum load among the PEs. This metric tracks the maximum number of queries directed to a PE. Using this metric, we identify the PEs which are potential bottlenecks and correct the problem by initiating data migration to spread the load of the "hot" PE to its neighbouring PEs. This set of experiments is conducted in the first phase using the actual aB$^+$-tree constructed. No data migration occurs if the loads of all the PEs are within 15% of the average load[3]. Otherwise, data migration is initiated and a branch at the root level of the overloaded PE's B$^+$-tree is transferred to its neighbouring PE. Note that the load threshold can be adjusted depending on how close we want the loads of the PEs to be near the ideal.

We observe that our adaptive approach requires us to migrate subtrees from multiple levels. A simple strategy would be to migrate a predetermined number of subtrees from a fixed level only (static approach). To assess the benefits of the adaptive approach, we compare it with the static strategy under two different granularities: *static-coarse* where only branches at the root level can be migrated, and *static-fine* where branches at one level below the root of the B$^+$-trees can be migrated. For this experiment, we wanted the B$^+$-trees in the PEs to have at least three levels of index nodes. Therefore, we used a page size of 1024 bytes and 2 million records to build the initial B$^+$-tree and distributed the second tier B$^+$-trees to 8 PEs. Figure 9 shows the result of this experiment. All the three strategies show performance gain as data is moved away from the "hot" PE. The performance gain is more gradual for static-fine compared to static-coarse because the amount of data migrated is limited to those indexed by branches of the B$^+$-tree. The static-fine strategy is important when workloads are extremely skewed towards a particular subtree. If this subtree is to be migrated entirely to the next processor, it will cause an undesirable sharp increase in the destination processor's workload, causing it to be the next bottleneck. In this situation, it will be better to migrate subtrees at the next lower level. Figure 9 shows that our adaptive approach gives superior performance as it is able to determine the right amount of data to migrate.

Since the adaptive approach performs best, we shall not discuss the static approach further in subsequent experiments. In the next experiment, we study the maximum load and load variation among 16 PEs after 10000 queries. The result is shown in Figure 10. PE 16 is overwhelmingly overloaded while PEs 1-15 are lightly loaded. Migrating data indexed by branches at the root level of the B$^+$-trees is able to reduce the maximum load in the "hot" PE by 40%.

Finally, we investigate scalability and sensitivity of data migration on maximum load when we vary the number of PEs in the system and the size of the dataset. We observe that the maximum load drops when we increase the number of PEs in the system. This is expected because the dataset is now distributed over more PEs which in turns distribute the load. In addition, the set of queries used is generated using the zipf distribution over 16 buckets to create load skew in our default system of 16 PEs. When we use a highly skewed set of queries generated using the zipf distribution over 64 buckets, there is hardly any reduction in the maximum load. Instead, the bulk of the load is still directed to the "hot" PE in the system which is gradually corrected by data migration. Figure 11 shows the results of these experiments.

Figure 12 shows that maximum load in the system of 16 PEs when we vary the size of the dataset. We observe that the

[3]Average load is the total number of queries divided by the number of PEs in the system

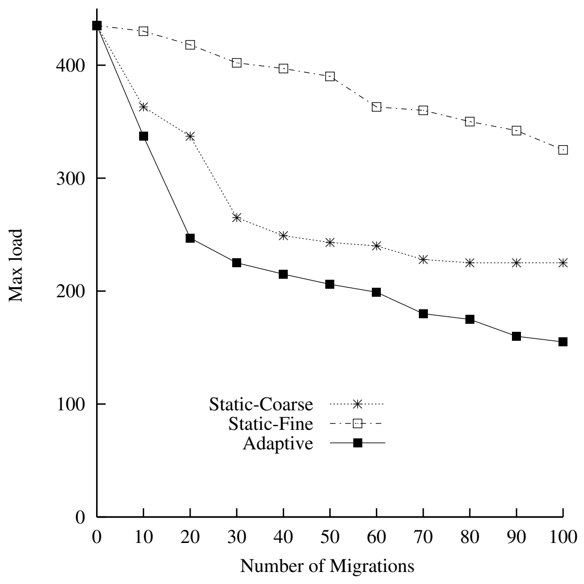


Figure 9: Comparison of maximum load when granularity of migrated data vary

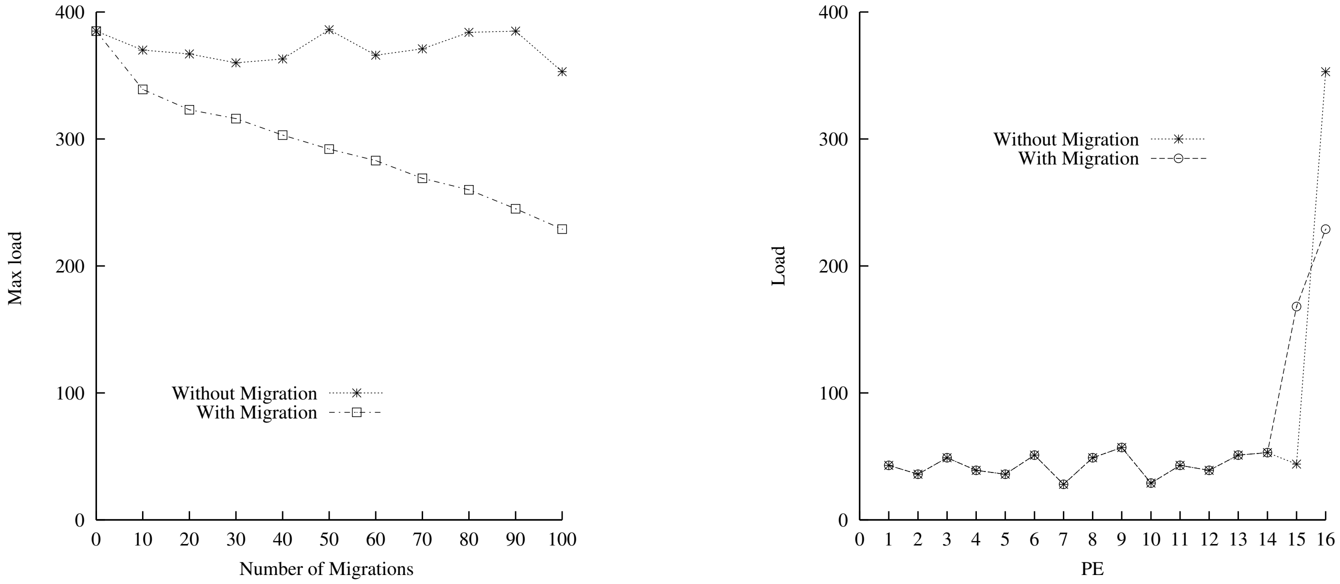
maximum load does not change much as the zipf distribution dictates the proportion of queries being directed to each PE. In all cases, we see that the maximum load has been reduced by 50% after migration of data from the overloaded PE.

4.3 Impact of Migration on Response Time

In this set of experiments, we examine the effect of data migration on response time of a query. No data migration occurs if the job queues of all the PEs has less than 5 queries waiting to be processed. Otherwise, data migration is initiated by picking the PE with the most number of queries waiting in the queue as the source PE. Figure 13 shows the average response time of a PE and the response time of a query for the most heavily loaded PE in a system of 16 PEs. Our results affirm the effectiveness of distributing some index branches (and hence data pages) in the overloaded PE to their neighbouring PE to reduce the average response time, thus increasing the throughput of the system. In fact, the response time of a query in the "hot" PE differs greatly from the average response time of 30 ms [4] in the lightly loaded PE. Given the extreme skews in the queries, the "hot" PE received a disproportionate number of queries. This extreme response time variation is narrowed with data migration.

Experiments to study the scalability and sensitivity of data migration on the average response time of a query includes varying the mean interarrival time of the queries, the number of PEs in the system and the size of the tuples in the dataset. The results are shown in Figures 14 and 15. We note that the average response time increases exponentially when the mean interarrival time is less than 15 ms. This also occurs when the number of PEs in the system is less than 32. The average response time of the system remains quite the same at about 480 ms when the size of the dataset is less

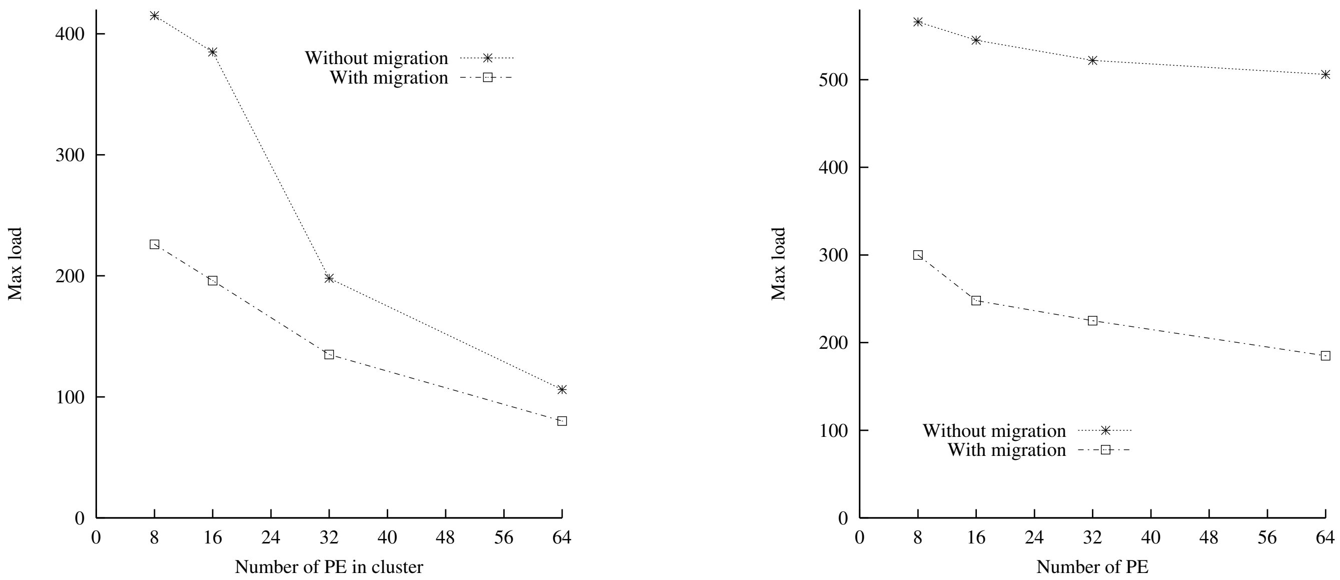
[4]With 1 million records distributed over 16 PEs, the average height of the B$^+$-trees in the PEs is 1. Hence, an average of 2 page accesses is needed to retrieve a required tuple.



(a) Maximum load in a system of 16 PEs

(b) Load variation in the PEs

Figure 10: Effect of migration on maximum load



(a) Query set generated using zipf distribution over 16 buckets

(b) Query set generated using zipf distribution over 64 buckets

Figure 11: Comparison of maximum load when number of PEs vary

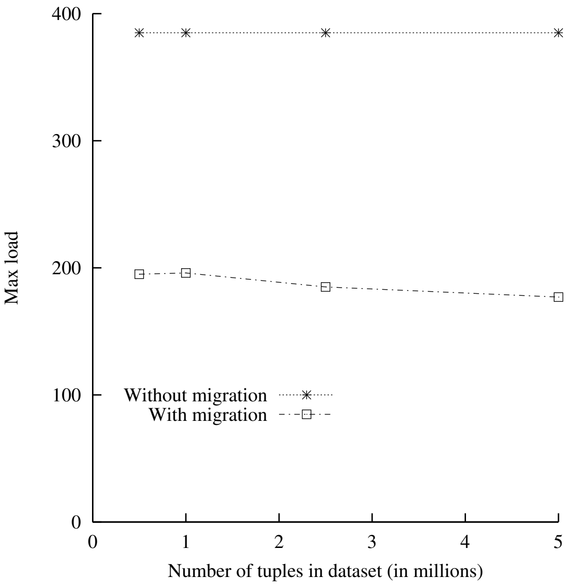


Figure 12: Comparison of maximum load when size of dataset vary

than 2.5 million tuples. There is a sharp increase when the dataset is 5 million due to the increase in the height of the B^+ trees in the PEs. In all these cases, data migration is able to improve the average response time by at least 60%. We see that our data migration strategy is able to correct performance degradation of the system effectively.

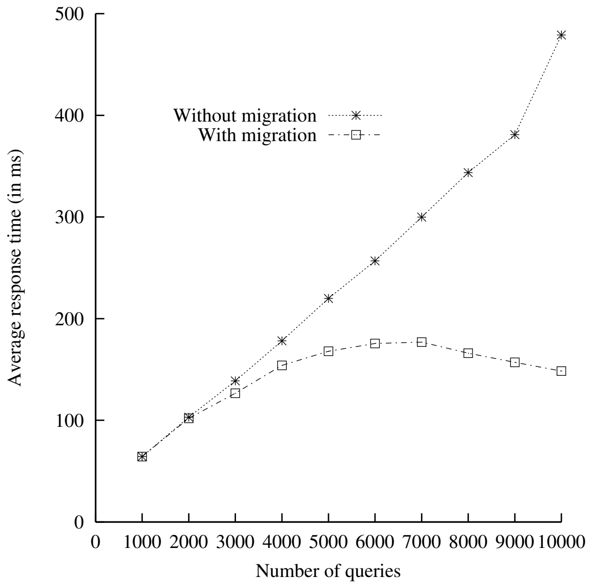
4.4 Empirical Results on Fujitsu AP3000

As mentioned, we also implemented our reorganization techniques on the Fujitsu AP3000 machine. The Fujitsu AP3000 machine is a massively parallel processor system based on 32 Sun UltraSparc workstations connected by Fujitsu's proprietary high speed switch (200 Mbyte/s), the APnet. Given the high bandwidth of the network, it is hardly a bottleneck during reorganization. We investigate how our techniques perform in a real multi-user environment with competing processes.

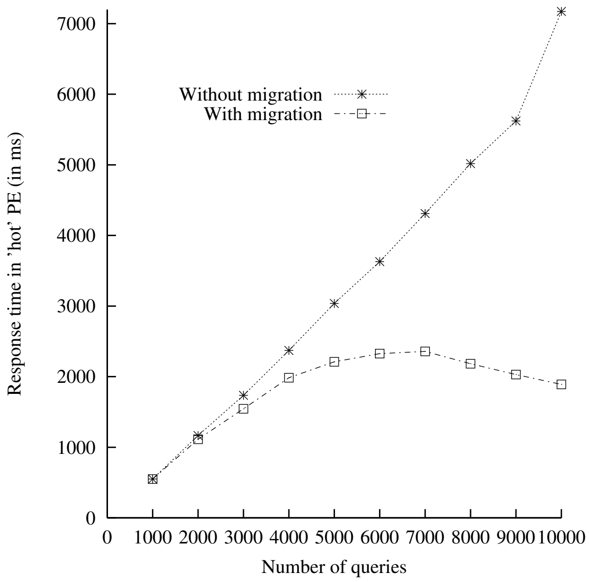
We run experiments on AP3000 to study the impact of migration on the response time and load directed to the processors. Figure 16 shows the response time in the overloaded processor in a 16 node cluster and the average response time when the number of processors in the cluster varies. Although we could only use up to 16 processors, the empirical results obtained confirm the results from the simulation experiments. In general, while the experimental curves are roughly the same, the actual response time obtained on AP3000 is higher than the simulation results due to competing processes in a multi-user environment.

5. CONCLUSION

To the best of our knowledge, this is the first paper to propose an index-based tuning technique which enables fast determination of the amount of data to be migrated from an overloaded PE and efficient bulkloading of the migrated data in the destination PE. The granularity of the data to be migrated can be suitably varied by using branches at different levels of the index. This adaptive and incremental tuning



(a) Average response time for a system of 16 PEs



(b) Response time in "hot" PE

Figure 13: Effect of migration on response time

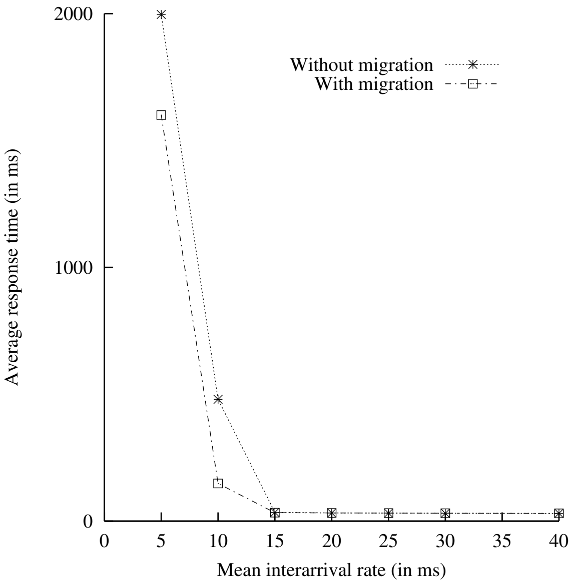


Figure 14: Comparison of response time when the mean interarrival rate vary

strategy allows the system to respond sensitively to access changes, minimizing heavy data movement and costly index updates.

We have designed a two-tier index structure to facilitate data access and data migration in clusters. The first tier is a partitioning vector to direct the search in the PE where the data is stored while the second tier is a collection of non-overlapping B^+-trees, one at each PE. The first layer is replicated in all the PEs to ensure that there is no central PE through which retrievals and updates must pass. This design also eases maintenance as updates to copies of this layer during migration is done lazily by piggybacking them on messages used for other purposes. We see further reduction in data migration overheads when the two tier index structure is globally height-balanced as in our adaptive B^+-tree (aB^+-tree).

We have demonstrated how a seemingly simple strategy can be yet scalable and effective in correcting any degradation in system performance when access patterns changes dynamically. We are currently extending this research to distributed spatial indexes.

REFERENCES

[1] K.J. Achyutuni, E. Omiecinski, and S.B. Navathe. Two techniques for on-line index modification in shared nothing parallel databases. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1996.

[2] T.E. Anderson, D.E. Culler, and D.A. Paterson. A case for now (network of workstations). *IEEE Micro*, 15(1):54–64, 1995.

[3] T. Johnson and P. Krishna. Lazy updates for distributed search structure. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 337–346, 1993.

[4] P. Krishna and T. Johnson. Implementing distributed search structures. Technical report available at *cis.ulf.edu:cis/tech-reports/tr92/tr92-032.ps.Z*, 1992.

[5] B. Kroll and P. Widmayer. Distributing a search tree among a growing number of processors. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 265–276, 1994.

[6] D. Lomet. Replicated indexes for distributed data. In *Proc. of Conference on Parallel and Distributed Information Systems*, pages 108–119, 1996.

[7] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. *SIGMOD Record*, 21(2):361–370, 1992.

[8] E. Omiecinski. Concurrent storage structure conversion: From b^+-tree to linear hash file. In *Proc. of 4th International Conference on Data Engineering*, pages 589–596, 1988.

[9] E. Omiecinski, L. Lee, and P. Scheuermann. Concurrent file reorganization for record clustering: A performance study. In *Proc. of 8th International Conference on Data Engineering*, pages 265–272, 1992.

[10] R. Ramakrishnan. Database management systems. McGraw-Hill, 1997.

[11] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the power of parallelism in a pile of pcs. In *Proc. of IEEE Aerospace*, 1997.

[12] P. Scheuermann, G. Weikum, and P. Zabback. Disk cooling in parallel disk systems. *IEEE Bulletin of the Technical Committee on Data Engineering*, 17(3):29–40, 1994.

[13] P. Scheuermann, G. Weikum, and P. Zabback. Data partitioning and load balancing in parallel disk systems. *VLDB Journal*, 7(1), 1998.

[14] G.H. Sockut and B.R. Iyer. A survey of online reorganization in ibm products and research. *IEEE Bulletin of the Technical Committee on Data Engineering*, 19(2):4–11, 1996.

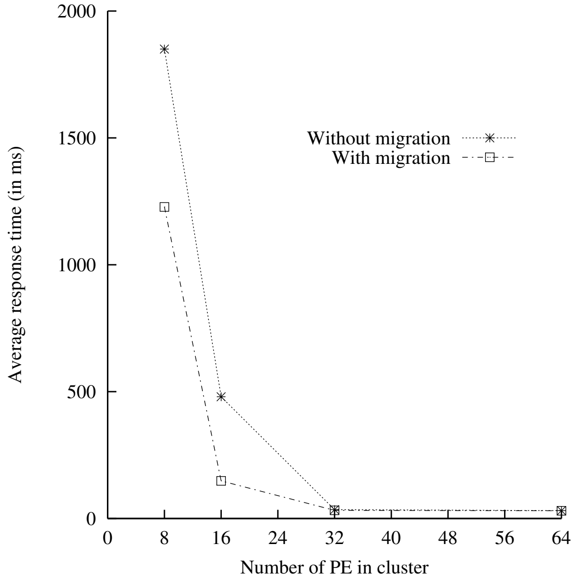
[15] T. Tamura, M. Oguchi, and M. Kitsuregawa. Parallel database processing on a 100 node pc cluster: Cases for decision support query processing and data mining. In *Proc. of SC97: High Performance Networking and Computing*, 1997.

[16] R. Vingralek, Y. Breitbart, and G. Weikum. Snowball: Scalable storage on networks of workstations. *Distributed and Parallel Databases*, 6(2), 1998.

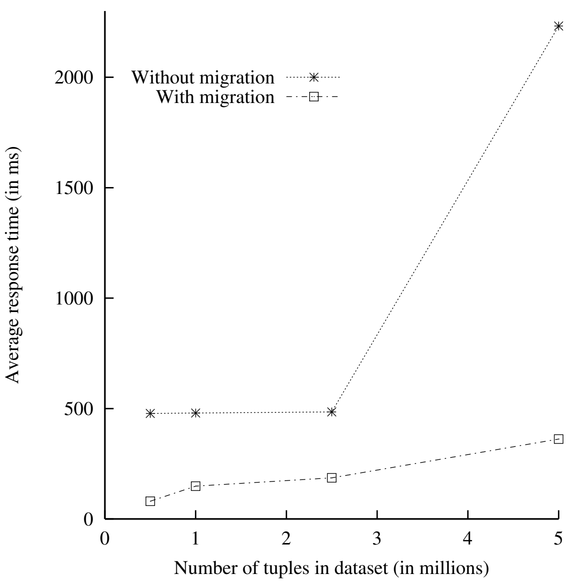
[17] K. Watkins. Discrete event simulation in c. McGraw-Hill, 1993.

[18] C. Zou and B. Salzberg. On-line reorganization of sparsely-populated b+ trees. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 115–124, 1996.

[19] C. Zou and B. Salzberg. Safely and efficiently updating references during on-line reorganization. In *Proc. VLDB*, 1998.

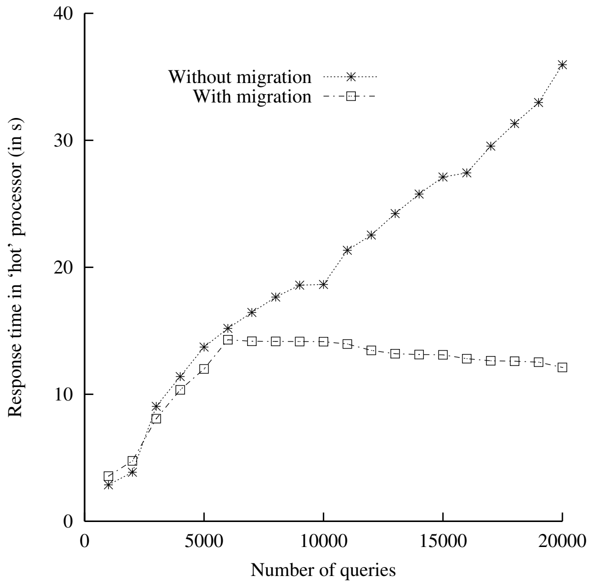


(a) Vary number of PEs in system with 1 million tuples

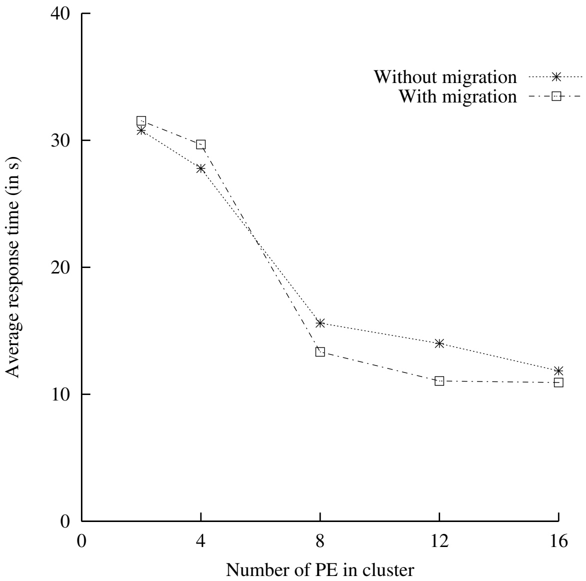


(b) Vary size of dataset in a system of 16 PEs

Figure 15: Comparison of response time



(a) Response time in "hot" PE (16 node cluster)



(b) Average response time when number of PE in cluster vary

Figure 16: Experiments on response time in AP3000