R-tree-based Data Migration and Self-Tuning Strategies in Shared-Nothing Spatial Databases

Anirban Mondal[1] *Masaru Kitsuregawa*[2] *Beng Chin Ooi*[1] *Kian Lee Tan*[1]

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

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

ABSTRACT

In order to provide fast and timely answers to queries in the context of spatial databases and GIS, we present our solution for effective data migration and tuning strategies in shared-nothing parallel spatial databases. Our purpose is to improve the performance of the indexes. Our approach has the following features. First, our scheme is self-tuning, dynamic as well as query-centric and it can adapt to dynamically changing user access patterns. Second, a global distributed R-tree-based indexing method is employed to facilitate effective data migration. Third, unlike traditional partitioning strategies where each processing element (PE) contains data from a single region of space, we allow each PE to store data from multiple and disjoint regions. This minimizes overlap in regions as well as coverage.

We implemented the proposed scheme and conducted an extensive performance study on Fujitsu's AP3000 machine with 32 workstations using real datasets. Our experimental results show that our load-balancing strategy can distribute the load effectively across the PEs in the system, thereby reducing response times of incoming queries.

1. INTRODUCTION

Nowadays, GIS is being deployed in several diverse applications such as cultural resource management, preservation planning, historic roads, maps and documentation of resources. The effectiveness of GIS in integrating geo-referenced data, analyzing and retrieving the answers to complex spatial queries, and integrating spatial features with attribute data has resulted in tremendous popularity of GIS.

In order to provide fast and timely answers to user queries, the spatial data needs to be indexed efficiently. The typically large sizes of spatial data and the CPU-intensive nature of spatial operations pose significant challenges to the efficient indexing of spatial data. An extremely popular and widely used spatial indexing structure is the R-tree [2]. A substantial amount of research work [3] has been done to exploit parallelism to provide better performance in the context of R-trees. Our work also exploits parallelism in case of R-trees to provide faster response times.

The deployment of shared-nothing parallel systems [1] is an attractive option for handling spatial data because of their cost-effectiveness (such systems are built from high performance and low-cost commodity hardware) and scalability. Such systems comprise of processing elements (PEs), each of which has its own memory and disk. Data are typically declustered across the PEs and indexed to facilitate speedy retrieval of data.

However, a well-known problem with such shared-nothing parallel systems is the workload skews among the various PEs caused by dynamically changing user access patterns that render the initial data placement ineffective. Hence the data needs to be reorganized [4, 6] online for load-balancing purposes. The problem becomes more challenging because spatial objects may overlap one another in space, thereby making it more difficult to determine a good placement scheme.

In this paper, we present our solution for effective data migration and tuning strategies in shared-nothing parallel spatial databases for purposes of improving the performance of the indexes. Our approach has the following features:

1. Our scheme is self-tuning, dynamic as well as query-centric. Hence, it can adapt to dynamically changing user access patterns.

2. We adopt a global distributed R-tree-based indexing scheme. The index has two tiers. In the first tier, information on the regions allocated to each PE is maintained. In the second tier, each PE indexes the data assigned to it using an R-tree. Such a structure turns out to facilitate migration easily: sub-branches of the R-tree at the overloaded PE can be "pruned" and migrated to a lightly loaded PE to be integrated to the R-tree there.

3. Unlike traditional partitioning strategies where each PE contains data from a single region of space, our approach allows each PE to contain data from multiple regions, that may be far apart in space. This minimizes overlap in regions as well as coverage, thereby leading to reduced response times.

We implemented the proposed scheme and conducted an extensive performance study on Fujitsu's AP3000 machine with 32 Sparc Workstations using real datasets. Our performance results show that our proposed approach is effective in distributing the load across the PEs in the system, thereby reducing the response times of the incoming queries.

2. DISTRIBUTED R-TREE-BASED INDEXING SCHEME

Our proposed system architecture is a shared-nothing system comprising of a set of PEs, each of which has its own memory and disk. The PEs are connnected in a Local Area Network (LAN) and they are centrally controlled by a Master PE, to which all the PEs periodically send messages concerning their load status. Whenever the Master PE detects any imbalance in the system, it initiates migration of data from the overloaded PEs to the lightly loaded PEs.

We propose a two tier distributed indexing scheme (shown in Figure 1) in which the first tier maintains information on the regions indexed by each PE. This is simply a set of Minimum Bounding Rectangles (MBRs) of the regions allocated to each PE. Since each PE has the flexibility of indexing data from multiple disjoint regions in space, it is possible for a PE to have more than one MBR in the first tier. The first tier resides at the Master PE.

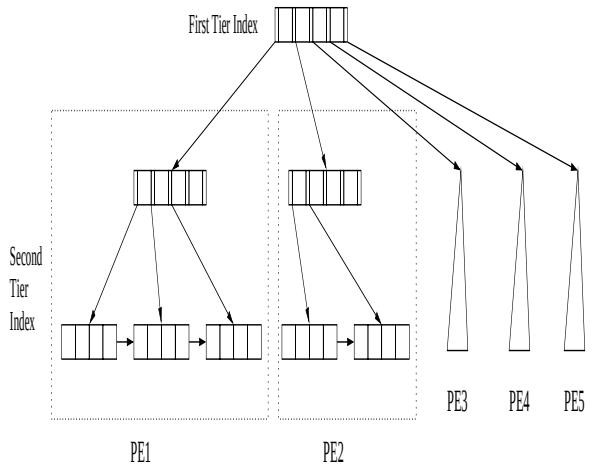


Figure 1: Our proposed two-tier distributed indexing scheme for multidimensional data

At the second tier, each PE indexes the data allocated to it using an R-tree [2]. An R-tree is a height-balanced structure for indexing spatial data.

We relax the fill factor of our R-tree to the point where each node may contain as few as one element. This is done so as to minimize the coverage required to enclose non-overlapping regions.

Consider the example shown in Figure 2. In Figure 2(a), we have four regions R1, R2, R3 and R4 such that R1 and R2 overlap, and R3 and R4 overlap, but there is no overlap between the (R1,R2) pair and the (R3,R4) pair. Traditionally, if all the MBRs of the four regions fit into a single node, then the parent node will have an MBR that bounds the four regions, and a pointer pointing to it (see Figure 2(b)). In our R-tree version, we store the regions R1 and R2 in one node, and R3 and R4 in another. Thus, there will be two MBRs at the parent node instead — MBR X bounding the regions R1 and R2, and MBR Y bounding the regions R3 and R4 (see Figure 2(c)). From Figure 2, it is clear that under the traditional R-tree structure, the MBR at the parent node would have been MBR Z, which has a much larger false enclosing space than that of our R-tree version. Minimizing coverage allows us to prune away some searches quickly, thereby facilitating the efficiency and effectiveness of our data migration strategies.

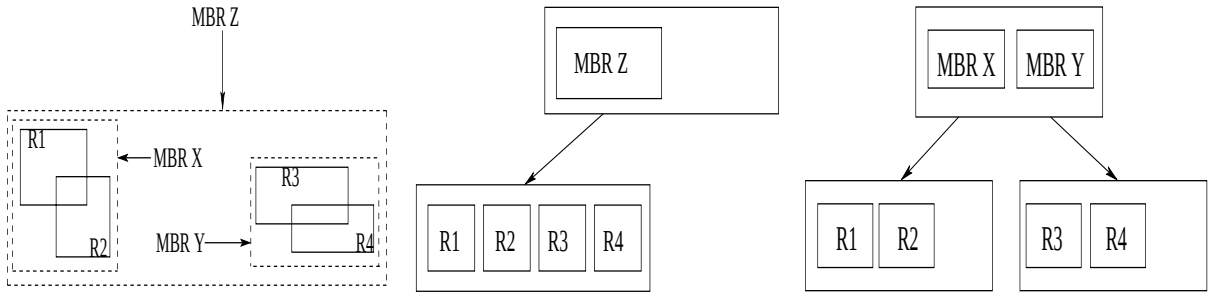
While the search and delete operations follow standard R-tree algorithms, we distinguish between two types of insertions: *single record* insertions and *sub-tree* insertions. The former deals with newly added data that are inserted one at a time, and a standard insertion algorithm can be employed. The latter handles the special case when a large number of objects are to be inserted as a result of data migration during reorganization. In our work, the set of objects to be inserted are bulkloaded into an R-tree structure first. We shall refer to this R-tree as the *R-subtree*. The R-subtree is then integrated into the existing R-tree at the PE. (An alternative approach is to maintain multiple R-trees at a PE.) Let the heights of the R-subtree and the R-tree be h_1 and h_2 respectively. There are essentially two cases to consider: (The case for $h_1 > h_2$ can be dealt with by exchanging the role of the R-subtree and the R-tree.)

1. $h_1 = h_2$**:** In this case, if a single node can contain all the entries of the two root nodes, we can integrate the two root nodes into one. Otherwise, we create a new root node that has the two trees as sub-trees.

2. $h_1 < h_2$**:** In this case, we 'extend' the height of R-subtree (by creating dummy parent nodes) until it is equal to h_2 and adopt the solution above. For typical database sizes, the height of an R-tree is not very large, so we do not expect the difference in height between R-subtree and R-tree to be significant enough to degrade performance.

Note that the R-subtree is not combined with other nodes because we expect the migrated data to be clustered together and such a combination is likely to lead to large and ineffective coverage.

The initial placement of the data across the PEs is done as follows. *One* single R-tree is created at one PE to index all the existing data. Then data is extracted from the branches of this R-tree and sent over



(a) Four regions in space (b) Traditional Approach (c) Proposed Approach

Figure 2: Minimizing coverage in the R-tree

to the PEs in the system. For example, assume that the number of PEs in the system is 3 and the number of *root-level branches* of the initial single R-tree is 9. In this scenario, the data from branches 1, 2 and 3 will be sent to PE_1; the data from branches 4, 5 and 6 will be sent to PE_2 and so on. Note that all the PEs will not receive the same amount of data, in case the number of root-level branches of the initial single R-tree is *not* exactly divisible by the number of PEs in the system.

3. SELF-TUNING STRATEGY

In this section, we address several issues that arise when supporting migration of multidimensional data. In our case, we have defined *Load* of a PE as the number of disk accesses at that particular PE.

When the system load is imbalanced (i.e., some PEs are heavily loaded while others are lightly loaded), data are migrated from the heavily loaded PEs to the lightly loaded PEs. We adopt a centralized decision-making approach in which each PE periodically sends its workload statistics to a designated Master PE and the Master PE initiates migrations when necessary. Our approach is simple and enables the Master PE to make better load-balancing decisions since it has a global view of the workload statistics. Morever, we allow multiple source PEs to migrate data to multiple destination PEs concurrently. Also, there is no restriction on the destination PEs, thus allowing us the flexibility to migrate data to *any* PE that is lightly loaded.

Once a source PE and a destination PE have been selected, the data to be migrated must be determined efficiently. The migrated data are obtained from subtrees in the index structure. This facilitates easy pruning of the migrated data from the tree (simply remove the branch), and easy integration of the data into the tree at the destination PE.

The proposed scheme also calls for some statistics to be maintained. Specifically, for each R-tree at a PE, for each entry in the root node, we keep track of the number of accesses on that sub-tree. While we can maintain more detailed information, we feel that the increased overhead of maintaining such detailed statistics would make the approach less attractive.

The amount of data to migrate depends on the degree of load imbalance. We have maintained the number of accesses for each sub-tree at the root node of the PE. We can thus determine the number of sub-trees that should be migrated, starting from the most heavily loaded subtree.

4. ALGORITHMS

In this section, we give a brief overview of our algorithms for balancing the loads of the PEs and the migration of spatial data. Periodically, each PE sends its workload statistics to a particular PE, which is designated as the Master PE. Our load-balancing algorithm executes in two phases and requires the use of 2 parameters $Threshold_{min}$ and $Threshold_{max}$. $Threshold_{min}$ is essentially application-dependent and also depends on the desired degree of load-balancing. The essence of this parameter is that when the difference in load between two PEs exceeds this parameter, it implies that the load imbalance is heavy enough to necessitate migrations. The significance of $Threshold_{max}$ is that when the difference in load between two PEs exceeds this parameter, it implies that *more than one* destination PE should be selected for the given source PE for migration purposes because the source PE is severely overloaded. $Threshold_{max}$ is given by the following:

$$Threshold_{max} = (TotalLoadofthesystem)/(NumberofPEs)$$

- Phase 1: First, the Master PE sorts the PEs in descending order according to their load values. Then it finds out the difference in load values between the first PE in the sorted list and the last PE in the sorted list. If the difference exceeds $Threshold_{min}$, the first PE in the sorted list is added to the *src_list*, which is a list of all source PEs and the last PE in the list is added to the *dest_list*, which is a list of all destination PEs. The amount of data to migrate is determined by the load imbalance between the two PEs under consideration and it is stored in *num_list*. If the difference in load values of the two PEs under consideration does not exceed $Threshold_{min}$, then both the PEs are added to *Extra_DestList*, provided their respective loads are less than $Threshold_{max}$. If

the difference in load values of the two PEs under consideration exceeds $Threshold_{max}$, the more heavily loaded PE is added to *Extra_SrcList*. For finding the *number of destination PEs* (n_{mig}) in this case, we use the following formula:

$$n_{mig} = (Load\ Difference)/(Threshold_{max})$$

Note that each PE in *Extra_SrcList* may have different values of n_{mig}.

Similarly, the Master PE determines the difference in load values between the second PE in the list and the second-to-last PE in the list and so on. Each time, the Master PE notes down the necessary migration information, in case any migration is required.

- Phase 2: The PEs in *Extra_SrcList* are sorted in descending order according to their loads. Also, we sort the PEs in *Extra_DestList* in ascending order of their loads. We designate the current number of existing elements in the *Extra_DestList* as n_{dest}.

  We take the most heavily loaded PE from *Extra_SrcList* and see the number, n_{dest}, of destination PEs available in *Extra_DestList*. If *Extra_DestList* is empty, the algorithm stops its execution, otherwise the heavily loaded PE under consideration is added to *src_list*. If n_{dest} is less than the value of n_{mig} for this PE, we extract all the items from *Extra_DestList* and add these items to *dest_list* with an indication that these items correspond to the source PE currently under consideration. If n_{dest} is greater than n_{mig} for this PE, we extract the first n_{mig} items from *Extra_DestList* and add them to *dest_list*. Items that have been extracted from *Extra_DestList* are deleted and the value of n_{dest} is updated accordingly.

  This algorithm is repeated with the other PEs in *Extra_SrcList* and it terminates when *Extra_DestList* becomes empty. In this manner, the Master PE uses *Extra_SrcList* and *Extra_DestList* to involve more PEs in the migration procedure so that the load may get more evenly distributed. At the end of Phase 2, the Master PE encodes these lists into a message and broadcasts this message to all the PEs in the system.

After receiving the encoded message from the Master PE and decoding this message, each PE has a copy of the migration information. (Note that our algorithm precludes the possibility of a particular PE being *both* a destination PE and a source PE at the same time.) If a particular PE finds its own rank number in the *src_list*, it identifies its corresponding destination PE from the *dest_list* and finds out how much data it should migrate to its corresponding destination PE from the corresponding entry in the *num_list*. Then it extracts the required amount of data from the branches or subbranches of its own R-tree and transmits the data to its corresponding destination PE.

If a PE finds its own rank number in the *dest_list*, it finds out its corresponding source PE from the *src_list* and the amount of data that it should receive from the *num_list*. Once the migrated data has arrived, it performs the bulkloading of this migrated data in order to incorporate the migrated data smoothly and efficiently into its own index structure.

5. PERFORMANCE STUDIES

Our performance evaluation consists of experiments on our implementation of the distributed spatial indexes on the Fujitsu AP3000 machine.

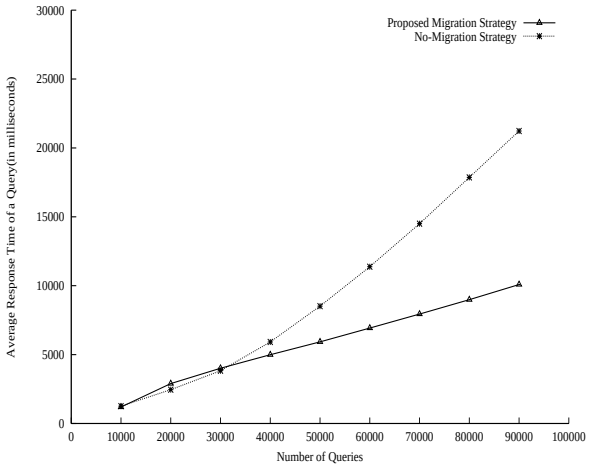
The Fujitsu AP3000 machine that we used is a massively parallel processor system based on 32 Sun UltraSparc Workstations connected by Fujitsu's proprietary high speed switch(200 Mbyte/s), the APnet. Fujitsu's Parallel Server AP3000 series is a distributed memory parallel server based on powerful 64-bit UltraSPARC technology. The AP3000 series is scalable and it can be expanded from the entry-level four-node configuration up to a 1,024-node supercomputing configuration. Given the high bandwidth of the network, it is hardly a bottleneck during reorganization. Our experiments enable us to investigate how our techniques perform in a real multi-user environment with competing processes.

In our study, we focus on window queries only. To model skewed workload, queries are directed to regions following a Zipf distribution. In practice, more queries are processed by each PE due to overlapping regions of different PEs. A low *zipf factor* (such as 0.1) implies a very highly skewed query distribution, in which most of the queries will be directed to only a few PE in the system, and we designate such PEs as the 'hot' PEs.

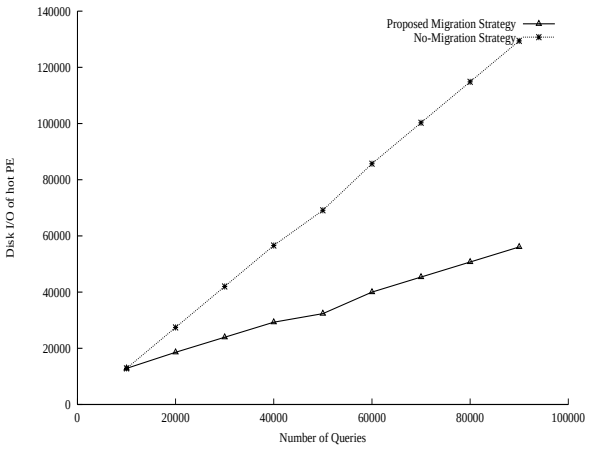
Our default experimental setup involves 32 PEs in the system, 30 PEs for indexing the data and 2 PEs for acting as coordinators. Initially, the size of the dataset in each of the 30 PEs is kept above 200000 rectangles. Hence, for a system of 30 PEs, the dataset comprises of 6 million rectangles. Even though we have performed experiments on other datasets, owing to space constraint, we only present the results of a representative set of experiments that we have performed on a real-life dataset (Greece-Roads) which we had enlarged for the purpose of our experiments. For more details about the results of our performance studies, please see [5]. As default, we set the *zipf factor* at 0.1 and the interarrival rate of the incoming queries at 10 microseconds. The system checks the load situation periodically after every 10000 queries.

- **Performance of our proposed migration strategy:** In this experiment, we analyze the performance of our algorithms. Figure 3a shows the average response time of the system over time after certain number of queries have been processed, while Figure 3b shows the disk I/Os incurred at the 'hot' PE in the same experiment. Figure 4 shows the effectiveness of our load-balancing strategy in distributing the load across different PEs in the system.

  From Figure 3a, we see that when the number of queries is small (< 30000), the average response



(a) Average response time of the system



(b) Disk I/O of 'hot' PE

Figure 3: Performance of our proposed migration strategy

time for our proposed scheme is slightly higher due to migration-related *disturbances* to the PEs and overhead costs of migration. Overhead costs of migration refers to data extraction cost in case of source PEs and bulkloading cost for destination PEs. As the number of queries increases, we observe that our proposed scheme outperforms the no-migration scheme by a significant margin. This is because of the reduction in disk I/O activity of the 'hot' PE as shown in Figure 3b.

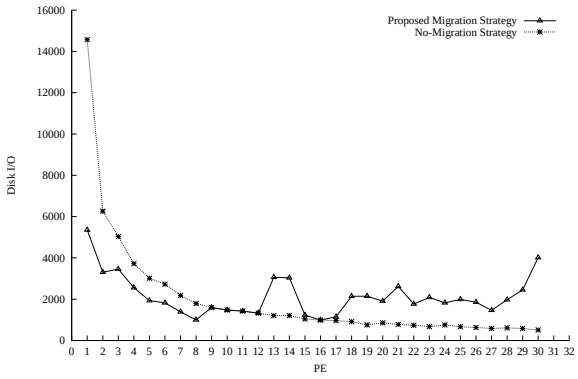


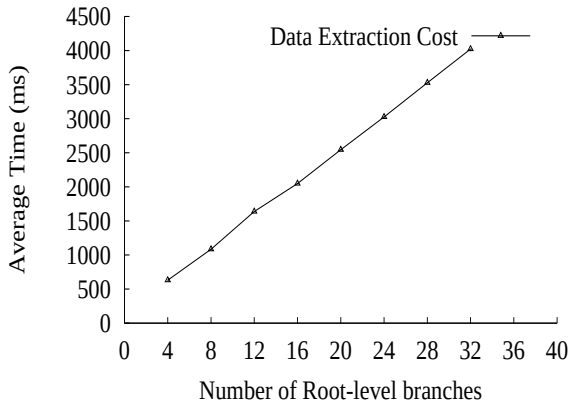
Figure 4: Load Distribution at interval of the 50000^{th} to the 60000^{th} query

From Figure 4, we observe that our proposed scheme can distribute the load more evenly than the no-migration scheme (especially, reducing the loads of the 'hot' PEs, namely, PE_1, PE_2 and PE_3). Hence, from Figure 3 and Figure 4, we conclude that our approach is effective in correcting the degradation in system performance owing to the overloading of certain PEs by a skewed query distribution.

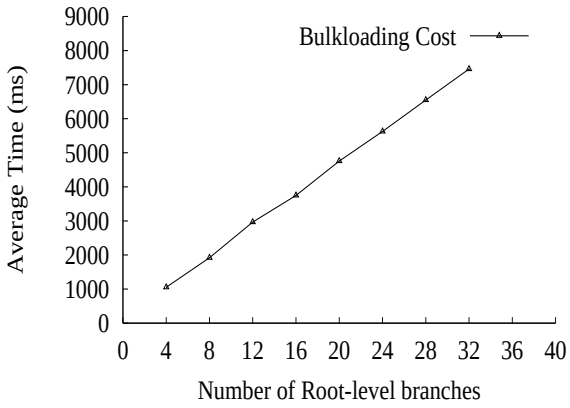
- **Cost of Migration**: The migration process has its own inherent costs and causes *disturbances* to the normal query processing work of the PEs in the system. The data extraction step causes *disturbances* only to the source PE, while the bulkloading step causes *disturbances* only to the destination PE. Figure 5a shows the cost of data extraction in terms of the time taken for extracting a specified number of *root-level branches* of the R-tree at a particular source PE, while Figure 5b shows the cost of bulkloading in terms of the time taken for bulkloading a specified number of *root-level branches* of the R-tree at a destination PE. By *root-level branches*, we mean the branches emanating from the root node of the R-tree at a particular PE. We did this experiment on the R-tree of a particular PE, where the number of *root-level branches* was 64 and each *root-level branch* was indexing around 4096 rectangles. As shown in Figure 5, both the cost of data extraction and the cost of bulkloading increase linearly with the increase in the number of *root-level branches* of the R-tree under consideration. Although this implies that cost of migration is high, our experiments illustrate that the benefit of our proposed approach far outweighs the overhead costs of migration. Hence, migration is still an attractive option for correcting query skews across the PEs.

6. CONCLUSION

This paper proposes an index-based tuning technique for efficient indexing of multidimensional data. The technique is based on migrating data from heavily loaded PEs to lightly loaded PEs such that the waiting times of queries at the heavily loaded PEs are reduced to a



(a) Cost of Data Extraction from a source PE



(b) Cost of Bulkloading at a Destination PE

Figure 5: Experimental Results for Cost of Migration

considerable extent. Our approach is dynamic and any imbalance in the system is detected automatically and migrations are also initiated automatically without any intervention on the part of the user. Our experimental results show that our proposed load-balancing strategy can distribute the load effectively across the PEs in the system, thereby reducing response times of incoming queries.

7. REFERENCES

[1] T. E. Anderson, D. E. Culler, and D. A. Paterson. A case for now (network of workstations.). *IEEE Micro*, 15(1):pages 54–64, 1994.

[2] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proc. ACM SIGMOD*, pages 47–57, 1984.

[3] I. Kamel and C. Faloutsos. Parallel r-trees. *Proc. ACM SIGMOD*, pages 195–204, 1992.

[4] M.L. Lee, M. Kitsuregawa, B.C. Ooi, K.L. Tan, and A. Mondal. Towards self-tuning data placement in parallel database systems. *Proc. ACM SIGMOD*, pages 225–236, 2000.

[5] A. Mondal, B.C. Ooi, K.L. Tan, M. Kitsuregawa, and M.L. Lee. Query-centric load-balancing using r-trees in shared-nothing spatial databases. *Unpublished manuscript, Available from the authors*, 2001.

[6] C. Zou and B. Salzberg. On-line reorganization of sparsely-populated b+ trees. *Proc. ACM SIGMOD*, pages 115–124, 1996.