A Mechanism To Detect Changing Access Patterns and Automatically Migrate Distributed R-tree Indexed Multidimensional Data *

Scott T. Leutenegger Rostislav M. Sheykhet Mario A. López
Mathematics and Computer Science Department
University of Denver, Denver, CO 80208-0189
{leut,rsheykhe,mlopez}@cs.du.edu

ABSTRACT

We present an algorithm to migrate multidimensional data in a non-replicated distributed environment. Our proposed algorithm is intented to improve query performance for mobile objects. Our algorithm automatically detects access pattern changes and migrates portions of the data from current sites of residence to sites recently accessing the data most frequently, thus reducing remote communication costs. We assume the data is indexed by an R-tree multidimensional index and that a global R-tree is used to locate and retrieve portions of the data set. We present a distributed R-tree structure and experimentally explore a specific access detection and migration mechanism. Our experimental results show that when compared to the no-migration case, and depending on network speed, our migration scheme may result in a query time reduction of a few percent to an order of magnitude.

1. INTRODUCTION

In this paper we present a new algorithm to automatically migrate multidimensional data among sites in a distributed system assuming static non-replicated data. Our target application is the following. Assume the data items are moving objects such as airplanes, cell phones, soldiers, or mobile computing devices. A collection of base stations are responsible for the objects within their territory. As objects move from one geographic territory to another, the data representing the physical objects needs to move from one base station to another. This prevents delays resulting from communication overhead that would result if the locations accessing the data, such as an airport control center, are distant from the data being accessed.

A particularly compelling case is that of mobile computing devices connected via an ad hoc network. Base stations querying distant objects will result in expensive multi-hop queries. In a non-mobile setting an attractive solution is to use replication or limited replication to reduce costs. Unfortunately, mobile computing devices are constrained by battery costs, hence updating of replicas, such as the locations of the mobile objects being indexed, would incur a significant power cost. Thus, localization of data towards the query center is needed to reduce multi-hop queries, and replication should be avoided to reduce power costs associated with consistency protocols.

Indexing of mobile objects is a complex problem that requires solutions of three fundamental problems: 1) when and how to migrate the objects among responsible base stations; 2) how to update locations and query the system efficiently; and 3) how to handle insertions and deletions. In this paper we address only the first problem.

Our work is differentiated from past distributed DBMS work by three key ideas. First, distributed multidimensional indexing support is not provided in any commercial systems. The academic literature is sparse with most work focusing on parallel indexing using gridfiles or R-trees [2, 8, 9] where the motivation is to use parallelism to speedup queries rather than manage data stored across many sites in a distributed database. The second difference is motivated by a key observation which has not been explored often in past research: data access patterns and their consequent workloads in a database are often not static. For example, a soldier may query objects to the north, but then based on some action to the south begin querying objects to the south. Third, the data being indexed will be moving objects. In this preliminary work we do not directly consider this issue, but have begun to address the issue in other work [11].

Past methods to minimize remote access cost for distributed data include replication in conjunction with off-line optimization based on static access patterns automatic migration of site location for single dimensional data, and a combination of migration and replication for single dimensional data. We reject static reallocation based schemes since they require administrative action to initiate reallocation, and hence are not nimble enough to respond to quickly changing access patterns. Thus, we consider how best to automatically detect access pattern changes and automatically respond by migrating data to the site most frequently ac-

*This work has been partially supported by the National Science Foundation under grants IRI-9610240 and ACI-9733658.

cessing it.

We are aware of only three works that address the automatic migration of distributed data. In [10] the authors propose an economic model that would drive the automatic migration of portions of data. The paper's focus was on an overall distributed DBMS design and devoted little attention to the migration issue. A more relevant paper is that of Brunstrom et. al. [1]. The authors assume the data is partitionable into sets of tuples that are frequently accessed together and devise mechanisms to migrate these partitionable sets to site accessing them most frequently. The algorithms presented in [1] assume that each partitionable set must track access patterns from all sites. This would not be practical if the granularity of the partitioning size is small or the number of sites is large. Our algorithms address this issue. In [12, 13] an algorithm is developed to tune the number of replicas and adaptively migrate replicas to the sites that access them most frequently. Thier work provides a more flexible paradigm, allowing both migration and dynamic limited replication, but was not designed to accommodate two important needs of our target applications. First, the algorithm of Wolfson et al does not address how to index and migrate *multidimensional* data. Second, the mechanism proposed in [13] assumes that a set of tests (expansion, contraction, and switch tests) are run periodically for all objects. If the number of objects is very large, as in our target applications, the costs of the adaptive algorithm would be too high to allow rapid response to changing access patterns. Furthermore, the power costs of the tests would be prohibitive.

Although these three works are similar in spirit, they do not sufficiently address our environment. A fundamental difference is that we assume queries from sites are range queries in a multi-dimensional space. Since data retrieved is likely close in multidimensional space we exploit this fact and organize the data across sites and on disk using a multidimensional index. Sequential R-trees [3] are a common multi-dimensional indexing technique that can be used to exploit spatial location for good clustering. We assume the reader is familiar with R-trees.

In this paper we propose and implement a distributed R-tree structure and devise mechanisms to move leaf level nodes from one site to another based on access patterns. Our structure is similar to the Master-Rtree [5], proposed for parallel databases, but we make the structure more amenable to the distributed environment. We assume the global R-tree structure is created for static data sets using STR packing [6]. Furthermore, we assume the R-tree index is used for global and local primary file organization, i.e. actual data is stored in the leaf level nodes of the index. Thus, when we migrate leaf level nodes we migrate the actual data.

2. DISTRIBUTED R-TREE ARCHITECTURE AND MIGRATION POLICIES

In this section we present our distributed R-tree structure, explain our migration mechanism, and discuss possible eviction policies.

2.1 Distributed R-tree Structure

Our distributed R-tree structure is a small modification of the "Master R-tree" architecture proposed in [5]. The Master R-tree architecture was proposed to speedup up R-tree queries via parallelism. The Master R-tree assumes that leaf level nodes are partitioned across several sites and that all non leaf level nodes are in a tree at one site, the master site. The level above the leaf which contains pointers to the leaf nodes thus has pointers that includes a site id as well as a page id. Declustering of leaf level nodes to support parallelism was studied in [4, 9]. In our work declustering is dynamic and driven by access patterns.

We discard the notion of a single copy of the master tree used in [5]. Instead, we replicate the non-leaf level portion of the tree across all sites. Assuming a branching factor of 100 or more, this amounts to no more than 1% memory overhead. The replicas of the non-leaf portion of the tree must be updated when nodes are migrated. This is done via broadcasts as explained below.

As an illustrative example assume a 2 site tree with 6 pages where pages 1,2, and 6 are located on site 1 and pages 3,4, and 5 are located on site 2. The non leaf level trees at both sites are replicated and identical. If site 1 submits a query that retrieves data only in pages 2 and 6 the query is completely local, thus reducing communication overheads. Our migration algorithms explained below move leaf level nodes to the site most frequently accessing them, thus increasing the probability of a query being satisfied locally.

2.2 Leaf Node Migration Policy

A migration may require that nodes first be evicted from the migrations target site. The eviction policy is described in the following section. Our policy is completely distributed versus centralized. This reduces overhead, allows for more responsive re-allocations, and increases availability.

Each leaf level R-tree node is augmented to contain an ownerID and access counter. The access counter is an indication of access frequency from one site relative to other sites. The ownerID contains the siteID of the site that has recently accessed the node most frequently. The current owner and the current location can be different as explained below. We refer to the access counter of node A as counterA and the ownerID of node A as ownerA. Two global parameters used are minThreshold and maxCounter. Parameter minThreshold is the minimum value the access counter must achieve before a node is migrated. It is used to prevent hysteresis. Parameter maxCounter is the maximum value the access counter can achieve, it is used as a crude form of aging. Even with this crude method we have achieved excellent results. Future work will investigate the benefit of using exponential aging.

When site S retrieves D data items from leaf level node A one of two actions are taken:

(1): If ownerA = S, counterA is assigned $min[maxCounter, (counterA{+}D)]$. If before this change counterA < minThreshold and after this change counterA > minThreshold, AND locationA $\neq$ S, a migration of node A from locationA to site S is performed and the value of locationA is changed to S.

Time	Site Accessing	# items retrieved	counterA	ownerA	locationA
0			0	2	2
1	1	70	70	1	2
2	1	100	170	1	2
3	2	20	150	1	2
4	1	60	10	1	1
5	1	10	20	1	1
6	2	50	30	2	1

Table 1: Migration Example

(2): If ownerA $\neq$ S, counterA is assigned $abs(counterA - D)$. Furthermore, if before this change to counterA it was the case that counterA $<$ D, then ownerA is set to S.

Hence, the ownerID contains the id of the site that has recently accessed the node most often. If many sites are accessing a node frequently it is unlikely that a migration would be beneficial since the node would likely just be migrated again shortly in the future to a different site. Instead, to prevent hysteresis, we ensure that a site accesses minThreshold data items more from a node than the other sites combined.

Consider the following concrete example. Assume node A is initially located on site 2, counterA is set to 0, ownerA is set to 2, and locationA is set to 2. Assume minThreshold is set to 200. Further assume time progresses from 0 (initial time) to time 6 corresponding to the rows in Table 1. Row 1 of Table 1 is indicates that at time step 1 site 1 retrieved 70 data items from node A, hence ownerA is changed from 2 to 1 and counterA is changed to 70. Although ownerA is changed a migration does not occur since counterA is below minThreshold. Note, a migration occurs after time step 4 when node A is migrated from site 2 to site 1.

When a migration occurs, the non-leaf level portions of the global tree need to be updated. Let *ptr* be the pointer in the parent of the migrated leaf level node that points to the migrated node. All sites are notified via a broadcast to change siteID of *ptr* in their copy of the non-leaf level tree to the new location.

Our migration threshold (minThreshold) is based on the number of actual data items returned since the communication bandwidth consumed in the absence of migration is proportional to this number. A simpler policy would be to just count the number of times a site accesses a node regardless of how many data items are retrieved from that node. Conversely, if data items are of different sizes the migration threshold could be based on the total number of bytes retrieved rather than the number of data items. We intend to investigate these variant policies later.

One final issue is what to do when a query arrives to a site that is in the process of migrating the needed node. In a read only environment, simply supply the query with the data from the local copy but do not update any counters. This is what we have done in our experiments. If the data is not static, then insertion and update algorithms that are cognizant of moving nodes must be constructed.

2.3 Eviction Policies

In our system we assume each site has a fixed storage capacity. Let L be the number of leaf level nodes, S be the number of sites, and F, $F \geq 1.0$, be the allowed imbalance factor. Then, the maximum number of nodes that can reside at a given site is $\frac{L \times F}{S}$. If one site accessed all the data most frequently all data would migrate to that site if some upper limit was not placed on the number of resident leaf level nodes. Given there is a maximum number of nodes, when data is migrated to a site some nodes may need to be evicted from the site before the site can accept the new nodes.

We assume an LRU based eviction policy. Keeping track of an LRU stack for all disk resident data, especially if the data is in the terabytes, may not be practical. Instead we assume each site maintains an LRU stack for the R most recently accessed pages. When a page not in the LRU stack is retrieved, the least recently accessed page in the stack is dropped from the stack. When a site needs to accept a migration and is filled to capacity a victim is removed from the site. The victim is chosen at random from leaf level nodes not in the LRU stack. Basically, the LRU stack is only used to identify the hot set at the site to prevent eviction of hot pages. In our studies we assume an LRU stack of $max(0.01 * L, 10)$, which for the experiments reported in this paper is 10.

Once a victim is chosen a new home must be found for the node. To find a home for nodeV (the victim), possibilities include:

1. Chose at random any site with sufficient space.
2. Move to the least loaded site.
3. If the ownerV $\neq$ locationV, migrate to owner V
4. If ownerV $=$ locationV, migrate to site accessing V second most frequently. This would require adding a second access counter and a secondary owner field to each node.

In options 3 and 4, the site the victim is migrated to may also be full causing more evictions. Given that the access counter of ownerV or the secondary owner of node V is not above minThreshold we question the efficacy of these algorithms. Option 2 requires keeping track of the least loaded site. Although this might improve results some relative to option 1, in our studies we assume option 1. In the future we intend to investigate the relative benefits from the different options.

3. METHODOLOGY

In this first step work, we consider the case that the location of the indexed objects is not changed, but the query patterns of the sites does change. In the future we intend to also investigate the impact of dynamically changing object locations as well as query patterns.

In all experiments we compare the measured response time of our migration algorithm to that of a static partitioning algorithm where the initial declustering is random. No attempt was made at comparing with a "good" static declustering since a premise of the work is that access patterns change frequently and hence any static declustering would soon deteriorate.

All results were obtained by measuring wall clock time for Q queries and taking the average. Given the length of time to run the experiments most results reported are for 1000 - 3200 queries. The specific number of queries for each experiment are reported in Section 4.

All experiments were run on a cluster of 333 Mhz Pentium II PCs with 128MB of memory running RedHat Linux 5.1, kernel version 2.0.35. The data sets used were small enough to fit in memory, thus observed experimental difference are primarily due to differences in communication cost. All experiments were run on a dedicated cluster of machines with no concurrently running jobs to interfere. Furthermore, the machines are connected on an isolated 100Mbit Ethernet preventing network interference.

We considered running experiments over the Internet, but variances in communication times make results irreproducible and hard to interpret. Not having dedicated long haul communication lines, we decided to simulate the effect by running all experiments on a 100Mbit Ethernet and having communications calls invoke a *msleep* Unix call proportional to the amount of data sent divided by the simulated network speed. Thus, sending 4096 bytes on a 100 Kbit per second network would incur a sleep of 0.32768 seconds. This methodology allows us to vary network speeds as we chose. We consider network speeds from 10Kbits/second to 10Mbits/second.

4. RESULTS

In this section we present experiment results that provide proof of the utility of our proposed migration scheme. In all experiments the data set used contains 100,000 two dimensional uniformly distributed points within the unit square. We assume 167 data points fit per R-tree node resulting in 599 leaf level nodes. All experiments assume our LRU eviction scheme and that the maximum capacity of a site is $\frac{599 \times 1.1}{number\ of\ sites}$ Furthermore, in all experiments we assume the global R-tree is first built using the STR packing algorithm [6] and the initial location of leaf level nodes is determined by random declustering across all sites.

Queries for all experiments are synthetically generated range queries within the unit square. Each range query is a square with side lengths of 0.1, thus retrieving 1% of the data on average. A client at site S chooses the center of each query from a binormal distribution with mean (C_x, C_y) and a default standard deviation value of 0.02. By binormal we mean each of the x and y-coordinates of the query center point are each chosen from an independent normal distribution with mean C_x *or* C_y and standard deviation STD. Assume a sample query center point is x,y. Then, the query bounding box is [(x-0.05,y-0.05),(x+0.05,y+0.05)]. If a portion of the query box falls outside the unit square we assume the query box is truncated to be contained within the unit square.

In most of our experiments we assume network speeds of 100Kbits/second or 1Mbit/second. The benefit of our migration scheme is magnified for slower networks. Thus, if slower wireless networks or the Internet are to be used then our results provide a lower bound on the expected performance improvement from using migration.

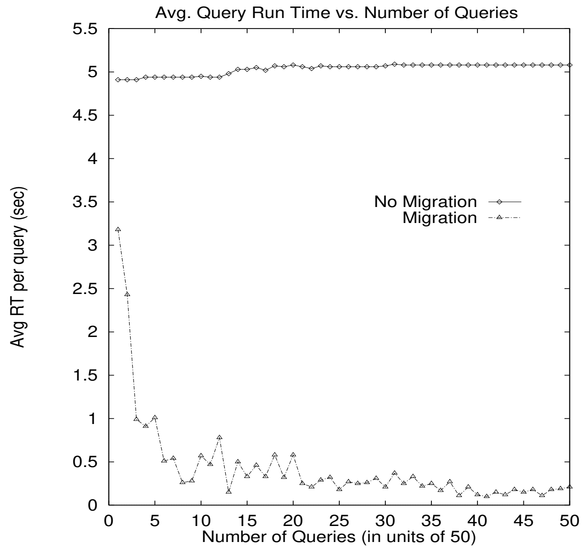

Figure 1: Two Sites, Fixed Query Location

In all experiments minThreshold was set to 334 (200% of the maximum node occupancy), and maxCounter was set to 668 (400% of the maximum node occupancy).

4.1 Two Sites, Fixed Query Location

This first experiment does not involve changing access patterns but demonstrates that our migration policy enables the tree to overcome the initial random declustering and migrate nodes to the sites accessing them most frequently. We assume two sites with each site having a client running range queries. The mean query center points are (0.1,0.1) and (0.9,0.9) for sites 1 and 2 respectively. Network speed is fixed at 100Kbits/second. We ran 2500 queries per site assuming the query center points do not move. The numbers presented are averaged over both sites since each site is originating queries.

In Figure 1 we plot the average query response time for the first 50 queries, the next 50 queries, and so on, up to the last 50 queries. The lines are "bumpy" since when averaged over only 50 queries unlucky query regions or lucky query regions can have a significant impact on the average response time.

The top line assumes no migration and the bottom assumes migration. In the no migration case about half of the requested data must be fetched from the remote site. For the first 50 queries the ratio of non-migration response time over migration response time is 1.51. The improvement continues over subsequent groups of 50 queries. When averaged over all 2500 queries the non-migration policy requires over an order of magnitude more time (11.65) than the migration policy.

4.2 Two Sites, Fixed Query Location, Effect of Network Speed

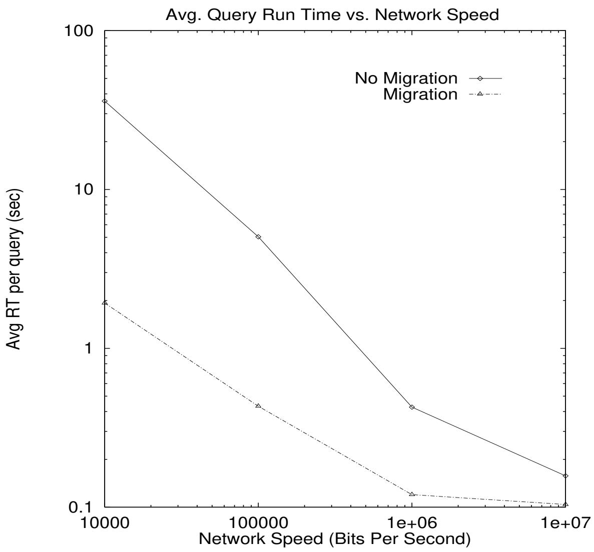

Figure 2: Two Sites, Fixed Query Location, Vary Network Speed

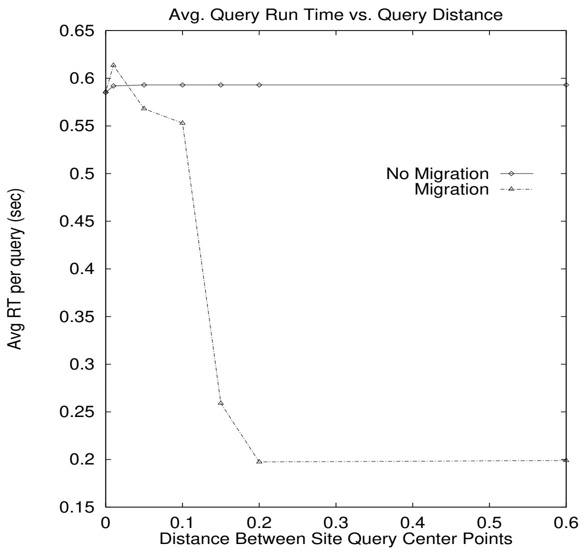

Figure 3: Two Sites, Impact of Query Disjointness

In this experiment we consider the effect of network speed. We ran 1000 queries at each client and report the average response time per query averaged over both sites. The experiment is the same as above except we plot the average over all queries and vary network speed.

In Figure 2 plot average query response time versus network speed. Note both axes are logarithmic. For slower network speeds the improvements obtained from migration are quite dramatic. The ratio of non-migration over migration response times are (18.65,11.65,3.55,1.51) for network speeds up (10K,100K,1M,10M) respectively. Even at a network speed of 10Mbits/second improvements are obtained.

4.3 Two Sites, Disjointness of Query Locations

In this experiment we consider how disjoint queries from different sites must be in order for our migration scheme to work. We vary the distance between the mean query center points for two sites. Specifically, we start with query center points of (0.2,0.5) and (0.8,0.5) for site 1 and site 2 respectively. We then reduce the distance between the two from 0.6 to 0 by moving both sites center points in towards (0.5,0.5). In Figure 3 we plot the mean response time versus the distance between mean query center points. We assume a 1Mbit network speed and STD of 0.02. The benefit of using migration is magnified for slower network speeds.

When the distance is greater than 0.15 there are few queries from the two sites that intersect and hence migration significantly reduces response time relative to no migration. As the distance shrinks less than 0.15 the queries from the two sites begin to intersect. When the distance is 0 all queries issued from both sites are competing for the same pages and hence our minThreshold parameter prevents any migrations. The experimental results show that when queries issued from the two sites are not disjoint there is no benefit from migration, but due to the minThreshold parameter the algorithm is robust enough to prevent spurious migrations and resultant performance degradation. Note, the differences in response time in the 0 - 0.05 distance range are insignificant since they are within the bounds of experimental error.

4.4 Two Sites, Moving Query Locations.

In this experiment we consider the impact of changing the access patterns. We assume client 1 at site 1 starts with (0.1,0.1) as the mean query center point and client 2 at site 2 starts with (0.9,0.9) as the mean query center point. After 10 queries, the mean center point of each client moves. We assume mean query center points move distance δ clockwise along a square path delimited by corners (0.1,0.1) and (0.9,0.9). We consider δ values of 0.01, 0.02, 0.05, and 0.1. For δ equal to 0.01 we run 3,200 queries. Thus, each client makes one complete circuit around the path. For the other values we reduce the number of queries so the each experiment results in one circuit around the box. Thus, for $\delta =$ (0.01, 0.02, 0.05, 0.1) we run (3200, 1600, 640, 320) queries respectively.

In Figure 4 we plot the response times versus δ for 100K, 1M, and 10M network speeds. Note the y-axis is logarithmic. We include results for 100K, 1M, and 10Mbit networks. Increasing the speed at which query moves, i.e. increasing δ, does degrade the performance of the migration policies some. At $\delta = 0.1$ queries are moving very fast yet the migration policy still provides an improvement over no migration. Specifically, at $\delta = 0.1$ the ratio of no-migration response time over migration response times are (1.21,1.27,1.04) for network speeds of (100K,1M,10M) respectively. For a slower moving query window, $\delta = 0.01$, the the ratio of no-migration response time over migration response times are (1.49,1.60,1.30) for network speeds of (100K,1M,10M) respectively. Thus, as long as some small number of queries are repeated in the same area before jumping to a new area (10 in our experiments), the migration mechanism is able to handle fast moving access patterns. If we change the number of repeated queries to a larger number than 10 much more dramatic improvements can be obtained.

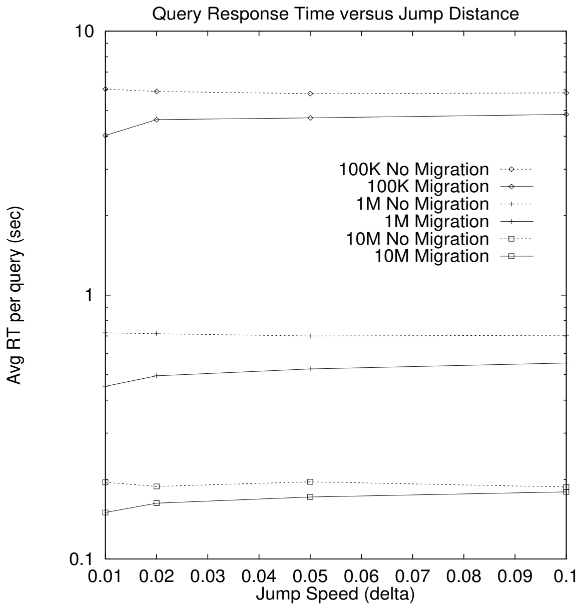

Figure 4: Two Sites, Moving Query Locations

4.5 Other Results

Due to space limitations we were not able to include other experimental results. In [7] we present additional results. We highlight the conlusions here and refer the reader to the technical report. First, if query locality is decreased by increasing the query center point STD, both policies are negatively affected although the migration policy is better able to handle the decrease in locality. Second, we ran moving query experiments for a system with 4 sites and obtained similar conclusions as the two site experiment.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a new algroithm which detects changes in access patterns and migrates leaf level R-tree nodes (data) to sites accessing them most frequently, thus offering the potential to reduce remote accesss and hence query time. We have demonstrated up to an order magnitude reduction in query response time. In all experiments considered, the migration algorithm always performed as well or better than the no-migration algorithm. We have shown that the amount of performance improvement is determined by three factors:

1. Network Speed. For slow long haul or ad-hoc networks running at 10 - 100Kbits per second up to an order of magnitude improvement in performance is possible. For fast wired networks, the penalty for a remote access is reduced and hence the benefit of using migration is reduced. For 1-10Mbit networks we still obtain experimental improvements of a few percent to 355%.
2. Locality of reference. We have considered two ways to decrease reference locality. First, we varied the standard deviation of query center point locations. Second, we varied the speed at which the mean query center points moved around a square path within the unit square. In both cases our migration algorithms experienced little performance degradation as locality was reduced.
3. Disjointness of site accesses. When the areas different sites access are disjoint the migration mechanism provides considerable improvement. When there is a strong co-location of queries no benefits are obtained from our migration algorithm, but no penalty is paid either. Thus, our migration algorithm reduces query response time when site accesses are disjoint.

Due to lengthy run times of our experiments we could not explore all issues to our satisfaction. We have provided proof of concept, yet more work needs to be done in tuning and exploring the overall applicability of our migration scheme. The first issue to be addressed is how to handle updates of the object locations. Second, the tradeoff in update cost versus query inaccuracy of using slightly out of date data needs addresses. Third, we intend to more directly explore the impact assuming multi-hop ad hoc networks. Finally, if the coordinates of the data being indexed are changed frequently, then restructuring of the logical R-tree itself may be needed.

6. REFERENCES

[1] A. Brunstrom, S.T. Leutenegger, and R. Simha. Experimental evaluation of dynamic data allocation strategies in a distributed database with changing workloads. In *Proc. of 4th International Conf. on Information and Knowledge Management (CKIM)*, 1995.

[2] S. Ghandeharizadeh and D.J. DeWitt. A multiuser performance analysis of alternative declustering strategies. In *Proc. of the 7th International Conference on Data Engineering (ICDE 90)*, pages 466–475, 1990.

[3] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, pages 47–57, Boston, Mass, June 1984.

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

[5] N. Koudas, C. Faloutsos, and I. Kamel. Declustering spatial databases on a multi-computer architecture. In *Proc. of Extended Database Technologies (EDBT)*, 1996.

[6] S.T. Leutenegger, M.A. Lopez, and J.M. Edgington. Str: A simple and efficient algorithm for r-tree packing. In *Proc. of the 14th International Conference on Data Engineering (ICDE 97)*, pages 497–506, 1997.

[7] S.T. Leutenegger, R. Sheykhet, and M.A. Lopez. An automatic r-tree based migration scheme for efficient relocation of geographically distribute multi-dimensional data sets. Technical Report COMP.00.02, University of Denver, 2000.

[8] J. Li, J. Srivastava, and D. Rotem. Algorithms for loading parallel grid files. In *Proc. ACM Sigmod*, pages 347–356, 1993.

[9] B. Schnitzer and S.T. Leutenegger. Master-client r-trees: A new parallel r-tree architecture. In *Proc. of 1999 Conference of Scientific and Statistical Database Systems*, 1999.

[10] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, and C. Staelin. An economic paradigm for query processing and data migration in mariposa. In *Proc. 3rd Int. Symp. on Parallel and Distributed Info. Systems*, pages 58–67, sep 1994.

[11] S. Šaltenis, C. Jensen, S. Leutenegger, and M. Lopez. Indexing the positions of continuously moving points. In *Proc. ACM SIGMOD*, May 2000.

[12] O. Wolfson and S. Jajodia. Distributed algorithms for dynamic replication of data. In *Proc. of the Symposium on Principles of Database Systems (PODS 92)*, 1992.

[13] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.