Spatio-Temporal Data Services in a Shared-Nothing Environment

Marios Hadjieleftheriou*, Vassil Kriakov‡, Yangui Tao†, George Kollios†, Alex Delis§, Vassilis J. Tsotras*
* Computer Science Department
University of California, Riverside
Email: marioh, tsotras@cs.ucr.edu
† Computer Science Department
Boston University
Email: gkollios, ygtao@cs.bu.edu
‡ Computer Science Department
Polytechnic University
Email: vassil@milos.poly.edu
§ Computer Science Department
The University of Athens
Email: ad@di.uoa.gr

Abstract— **Recently, there has been a proliferation of applications creating spatio-temporal data that has to be processed, stored and queried efficiently. Existing applications may produce Terabytes of raw data per day that routinely necessitate the execution of millions of update operations, so as to keep the underlying database up-to-date. Consequently, there is a need for spatio-temporal data management systems that are able to support such update intensive operations. Moreover, these systems should offer users the capability to examine *past and present* versions of the data in an on-line fashion. In this respect, we propose a system that exploits the inherent parallelism of a shared-nothing computing environment for storing and indexing the spatio-temporal data. Our infrastructure consists of a cluster of workstations (COW) connected via a Gigabit/sec network, with servers whose operation is based on a distributed multi-version indexing scheme. We describe our proposed system architecture, data organization, as well as pertinent algorithms. We discuss various optimizations whose objective is to ensure robustness and scalability under highly dynamic situations manifested by excessive query loads and high update rates. Finally, initial experimental results using a system prototype and simulated environments support the effectiveness of our approach.**

I. Introduction

Managing update intensive spatio-temporal data is becoming a daunting task. With recent advances in data collecting disciplines (satellites, sensor networks, etc.), spatio-temporal data is produced in a massive scale and have to be processed, stored and, most importantly, queried efficiently. With scientific applications producing Terabytes of raw data on a daily basis, the efficient execution of millions of update operations is essential for maintaining datasets up-to-date. Example of such environments, already in use, include the Terra spacecraft [6] which produces around 200 GB/day and the Landsat 7 which generates 150 GB/day of geophysical data [26]. Collaborative projects among hundreds of researchers from numerous countries necessitate reliable data accessibility with consistent response times to queries. The Human Genome Project [2] is an example of such a collaborative effort that combines talent from 18 countries. Similarly, advances in sensor network and wireless technologies have contributed substantially to the generation of vast datasets updated thousands of times per second in applications such as traffic analysis and cellular phone network management (for example a traffic control system that tracks the positions of all vehicles coming in and out of Manhattan). Finally, recognizing the importance of data visualization, numerous other scientific applications necessitate efficient management of spatio-temporal imaging datasets that can reach the scale of Petabytes in sizes; examples include the SkyServer [1] for astronomical research and MRI scans for medical research [4]. There is no sign that this trend will change in the near future, making the design of robust spatio-temporal data systems for update intensive applications both a top priority and a timely research objective.

The common denominator of all aforementioned applications is that their underlying access methods should be able to support significant update operations in a way that maintains data up-to-date. At the same time, access structures should provide guarantees about the response time of user requests. It is unrealistic to deploy any known disk-based spatio-temporal access method [25], [19], [8], [20], [24] in the hope that it could support data update rates of thousands of operations per minute. In addition, a core requirement of a wide range of spatio-temporal applications is to be able to store historical information, hence, giving the ability to users to browse past versions of data for analysis purposes (e.g., comparison, assessment, pattern discovery, evolution trends, etc.). This makes the "update-intensive" requirement even stronger as inactive/older data cannot be purged but must be archived

This work was partially supported by NSF grants IIS-9907477, EIA-9983445, IIS-0220148 and Career Award 0133825.

in secondary storage. With the previously described update rates and data volumes it is prohibitive to store such historical information on a single workstation. However, it is natural to share this information on-line among multiple cooperating sites providing for efficient querying.

In the above context, we propose an efficient data architecture that utilizes the inherent parallelism of a shared-nothing infrastructure for storing and indexing spatio-temporal data. By taking advantage of existing clusters of workstations (COW) – run-of-the-mill computers connected via high speed LANs – multiple processing units with independent secondary storage devices can share the load of update intensive tasks. The abundance of aggregated disk space in such clusters enables applications to store the update history of the spatio-temporal data using a very cost-effective approach. Finally, large volumes of query requests about current or past version of the data can be processed efficiently in a decentralized collaborative fashion. The individual processing units can be any of today's low-cost workstations that feature abundant primary and secondary space and ample processing power. A salient feature of our adopted infrastructure is its adaptability to application needs accomplished by on-demand up/downsizing through simple addition or removal of workstations.

Moreover, the efficient allocation of data among multiple COW sites guarantees that the system can be optimized for handling efficiently multiple types of spatio-temporal queries. Nearest neighbor queries and range queries with small or large time intervals are directed to as many sites as possible concurrently, yielding the best possible performance by taking advantage of multiple processing units.

The main contributions of the paper are:

- A spatio-temporal data management system that can support very high update rates to the most recent version of the data.
- A system that stores the complete update history of the data and efficiently supports concurrent queries with varying access patterns to any version of the data.
- Dynamic load-balancing algorithms for distributing data efficiently across the cluster in order to minimize query response times.
- To investigate and assess advantages and trade-offs of the proposed architecture, through experimental evaluation using a prototype.

The following section describes the system architecture with emphasis on data migration, update rates and load balancing. Section 3 presents the experimental evaluation while section 4 discusses related work. Finally, the section 5 gives directions for future research and concludes the paper.

II. System Architecture

In this section we discuss in detail the proposed spatio-temporal data management system and outline its infrastructure components as well as its underpinning algorithms. The desiderata of such a system are:

1) Ability to query both present and past versions of the stored data in an on-line fashion.
2) Sustaining very high data update rates with minimal performance and responsiveness degradation, if any.
3) Capability to dynamically adapt to voluminous and/or frequent queries that might create excessive traffic and congest specific data regions whereas other remain lightly used. This adaptation should deliver predictable response times to user requests.

In order to satisfy all requirements we opt for a shared-nothing architecture that utilizes a cluster of workstations to distribute the data among multiple sites (Figure 1).

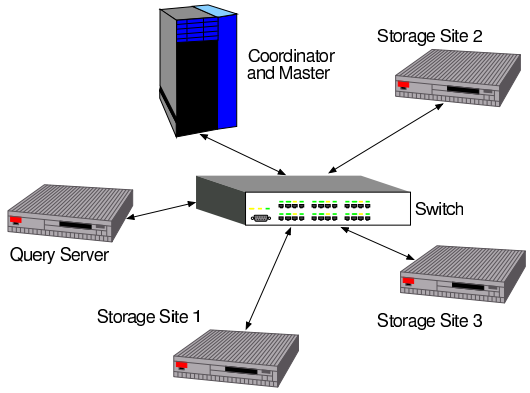


Fig. 1. A cluster of workstations connected via a high-speed switch.

Our framework utilizes a *Master* site whose key objective is to index the recent history (including the current version) of the spatio-temporal data. We attain this goal by using a *Multi-Version R-Tree (MVR-Tree)* [12], [24]. The MVR-Tree enables us to keep a small fragment of the spatiotemporal data at the Master (the most recent versions) and distribute older versions to other sites. It achieves that by clustering data that are adjacent in space and time in a small number of tree nodes, separately from the current/active data versions. Tree nodes that contain only expired data versions can be easily migrated over the network to other sites. We also keep the MVR-Tree in main memory to guarantee that very high update rates can be sustained, by avoiding slower secondary storage devices. We do not use an R-Tree on the Master simply because the MVR-Tree is better at indexing data with long time intervals, that are updated infrequently. Such data create long rectangles on the R-Tree index nodes [8] compromising the structure's performance. Instead, the MVR-Tree guarantees efficient clustering by creating artificial data copies, splitting long rectangles into multiple smaller ones.

We appoint a *Coordinator* whose responsibility is to make load-sharing decisions so that data is distributed intelligently across the COW, assisting in increasing concurrent query processing rates.

We appoint most of the sites of the cluster as *Storage Sites*, whose responsibility is to utilize their plentiful resources to archive the old data versions in secondary storage and for handling queries about data resident in their jurisdiction. We have various choices for indexing the past versions at the Storage sites; for example, we could use MVR-Trees (as in the Mater) or plain R-Trees [3], [7]. It should be noted that while previous work on managing spatiotemporal data has suggested

various advantages in using MVR-trees [12], [24], such works have not considered query parallelism. In our environment, we want to support many concurrent users issuing possibly overlapping queries. Under this consideration, we choose to deploy R-Trees [3], [7] instead of MVR-Trees on the Storage Sites. In particular this choice is supported by the following reasons:

1) First, it would be difficult to apply any dynamic load-sharing policies that need to move data between Storage Sites if MVR-Trees were used, since they are append only structures; this operation is much simpler and more efficient for R-Trees.
2) In addition, R-Trees have a smaller directory structure (with larger fanout) that is query and update efficient and introduces smaller space overhead. (A similar approach has been used in the MV3R-Tree [24], where R-Trees are used to index the leaf level of a multi-version structure.)
3) Finally, the R-Tree is more robust than the MVR-Tree for long period queries [12].

We also employ a *Query Server* that undertakes the responsibility of distributing user queries to the appropriate Storage Sites and compiling the results that are finally furnished to users.

Any workstation can be selected to play one or all of the above roles. Depending on system load and the state of the network, redistribution of the roles aids in avoiding and overcoming bottlenecks.

A. Migrating Data to Storage Sites

The multi-version approach [5], [21] is a typical method employed for storing the update history of temporal data. A widely used multi-version structure for multi-dimensional spatio-temporal data is the MVR-Tree [12], [24]. In order to illustrate how the MVR-Tree can be used to help distribute data among COW sites, a better understanding of the features of the structure is required. Consider a spatio-temporal dataset and assume that an R-Tree indexes the objects' shapes and locations. As updates occur, the R-Tree evolves by applying the corresponding object insertions and deletions — the old versions of the data are lost forever. On the other hand, storing all versions of the data and the evolution history of the R-Tree nodes corresponds to making the tree *multi-version*.

A multi-version structure is a directed acyclic graph of nodes. Moreover, it has multiple root nodes each of which is responsible for recording a consecutive part of the ephemeral R-Tree evolution. The root nodes can be accessed through a linear array called the $root^*$. Each entry in the $root^*$ contains a time interval and a pointer to the tree that is responsible for that interval. Data records in the leaf nodes of an MVR-Tree maintain the temporal evolution of the ephemeral R-Tree data objects. Each data record is thus extended to include the lifetime of the object: *insertion-time* and *deletion-time*. Similarly, index records in the directory nodes of an MVR-Tree maintain the evolution of the corresponding index records of the ephemeral R-Tree and are also augmented with *insertion-time* and *deletion-time* fields.

The MVR-Tree is created incrementally following the update sequence of the dataset objects. Consider an update at time t. The MVR-Tree is searched to locate the target leaf node where the update must be applied. This step is carried out by taking into account the lifetime intervals of the index and the leaf records visited while traversing the tree. After locating the target leaf node, an insertion adds a new data record with lifetime $[t, \infty)$. A deletion at time t' updates the deletion-time of the corresponding data record from ∞ to t'. All nodes may contain both live and expired versions of data. A node that contains only expired data is called "dead." The MVR-Tree uses various policies to determine whether a specific node should artificially expire, by copying all live entries to other nodes and leaving the dead entries behind. This process is necessary for clustering all live entries together in a small number of nodes to guarantee efficient access to the current state of the dataset. It also de-clusters the evolution history into disjoint time intervals, indexed by separate (but overlapping) trees.

A sample MVR-Tree appears in Figure 2 (for simplicity the figure shows a conceptual view of the structure, not an accurate representation). The $root^*$ contains three trees with jurisdiction intervals consecutive in time. The trees index overlapping data spaces at different times. The first two trees point to entries that are already dead. The third tree points to the most recent state of the data, but also contains some expired versions. All tree nodes contain consecutive versions of the same data (e.g., entry B in trees I and II). Also, some long lived entries have artificially expired (e.g., entry A) in order to decrease their extent on the time dimension.

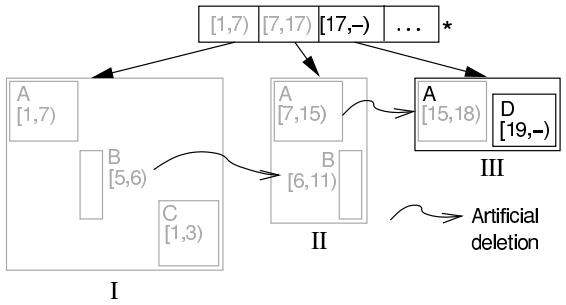


Fig. 2. A simple MVR-Tree.

Our networked approach is based on migrating all dead-leaf nodes to Storage Sites. Essentially, we use the MVR-Tree policies to determine when a node should become dead and at that time move it to an appropriate site. To better facilitate concurrent processing, all Storage Sites maintain a queue of pending requests. The Master initiates a node migration request and transmits it to the indicated Storage Site, which might have to append it in a queue until enough resources are available to process it. Meanwhile, the Master may elect to place more requests on other sites. A number of heuristics can be applied to determine the receiving Storage Site of a dead tree node yielding various feasible migration policies:

1) Round-robin.
2) Least-loaded Storage Site for update efficiency.
3) Proximity-clustering where data is uniformly spread out

across the cluster.

4) Temporal-clustering where every Storage Site holds a small time interval of the evolution history.

In order to take full advantage of the parallel processing potential of our architecture, ideally we would like to place the data such that the number of workstations that are required to answer a random query is maximized. Intuitively, this means that each workstation should index as larger part of the universe as possible. This maximization policy essentially requires assigning nodes to workstations that contain other nodes that are as far away as possible in the data-space. Thus, for a given query all workstations will probably contain data that may qualify as answers.

The round-robin policy distributes the dead leaf nodes across the workstations in sequence. The idea behind this algorithm is that the nodes will be uniformly distributed in the long run. If the leaf nodes of the Master MVR-Tree display uniform distribution, then the round-robin policy offers results close to the optimal policy described above. Otherwise, the distribution is random.

In order to keep the system operational while data migration is taking place, the node that is being moved to a new site must also remain on the Master MVR-Tree until the process is complete and all data has been committed. Any user query that refers to a time instant covered by the Master's jurisdiction interval has to be handled by the Master itself, hence consuming resources, locking part of the tree, and compromising update performance. The Least-loaded Storage Site policy tries to commit the migrated data as soon as possible in order to relieve the Master from the responsibility of that data. If the system is heavily loaded certain workstations may have too many pending requests (user queries, load-balancing operations, etc.), thus, the Master should choose the one with the shortest queue as the recipient.

Proximity-clustering was proposed in [10]. This algorithm distributes leaf nodes intelligently based on proximity measurements between nodes. The original algorithm was used for distributing the nodes of a single structure to multiple disks, in order to maximize the number of disks that need to be accessed for a single query, thus parallelizing the query execution process [10]. Here, we modify the algorithm in order to locate the workstation that contains the structure with the leaf level which best accommodates the migrated node based on the following Proximity heuristic. The proximity between two nodes R and S is defined as:

$$P(R,S) = \frac{\#\ of\ queries\ retrieving\ both\ R\ and\ S}{total\ \#\ of\ queries\ answered}$$

The measure of proximity can be computed by computing the integral of the number of queries that retrieve both R and S, for all query sizes. Another similar but simpler heuristic is to assign each leaf node to the R-Tree with the root node that needs to be enlarged the most in order to accommodate it. A drawback of this simple heuristic is that once all trees have been expanded to cover the data space, all new leaf nodes will be already contained in the trees, thus limiting the effectiveness of the heuristic.

A different set of heuristics attempts to cluster historical data according to time. If each Storage Site holds only a specific fragment of the evolution history, multiple concurrent queries that refer to disjoint time intervals can be processed all at once by redirecting them to the appropriate sites. A drawback of this heuristic is that as time always progresses, the jurisdiction time intervals of each Storage Site cannot be rigid. Special migration policies should be in effect in order to expand the jurisdiction intervals of some Storage Sites in order to make more room on others.

B. Sustaining High Update Rates

Since many spatio-temporal applications generate enormous amounts of updates per time instant, it is essential for a spatio-temporal management system to support high update rates. Usually, disk I/O dominates the update cost. State of the art systems utilize disk based index structures that are bounded in the number of updates that they can sustain by the limited disk access rates [14]. The maintenance cost of arranging the pages of a dynamic index structure so that it can support sequential I/O is prohibitive. Thus, most practical multi dimensional indices are bound to use much slower random accesses for loading disk pages in main memory. Indicatively, the access cost of a single page in cutting-edge hard drives is in the order of 5msecs while the access time to data stored in main memory is in the order of 50nsec, or a factor of 100 thousand times increase in speed. Even sequential I/Os that are a factor of 12 times faster than random I/Os are much slower in comparison.

Furthermore, main memory is becoming less expensive and most of today's high-end workstations are equipped with a few gigabytes of RAM, which is enough for storing millions of spatio-temporal objects. For example, assuming that the representation of a vehicle consumes 64 bytes (a 2-dimensional point and its velocity, along with some extra information), a dataset of 1 million vehicles consumes approximately 60 Megabytes of space.

The use of virtual memory to seamlessly increase the space available by volatile memory in modern operating systems presents two distinct advantages: even if a dataset far exceeds the capacity of available main memory, provisions to store it on disk are already taken by the OS. Second, if the dataset needs to be persisted on secondary storage, it can be done so efficiently, with minimum changes in design.

We choose to keep the MVR-Tree in main memory on the Master site to take advantage of all previous arguments. In addition, since we place older data to Storage Sites, we anticipate the average cost of updates to decrease substantially, since multiple CPUs amortize the costs involved in the updates. The more workstations that are attached to the cluster, the cheaper the overall cost of each update becomes, since the trees resident at Storage Sites become shorter, smaller and, thus, more robust. Also, by migrating older data to Storage Sites, we guarantee that the most recent version of the MVR-Tree can fit in the main memory of the Master.

C. Dynamic Load-Balancing

A distributed architecture by itself may not be sufficient in providing optimal performance for queries involving expired versions of the data. In order to optimize the performance gain due to concurrent processing it is necessary to guarantee

that the largest possible number of Storage Sites participate in answering any given query. Thus, it is desirable that the load distribution among sites is uniform. This is the job of the Coordinator, which monitors the load of each Storage Site and dynamically (i.e. while the system is online) schedules resource allocations among the sites. To achieve the desired load adjustments, the Storage Sites perform data migration since it is anticipated that the load follows the data. This data migration is feasible since leaf nodes in the R-Trees of Storage Sites do not follow any ordering and can easily be removed and inserted in the tree of another site. This process might reduce the quality of R-Trees (causing increased dead space and/or overlap at lower utilization), but it is our intention that a single Storage Site does not hold data which is spatially clustered.

To reduce the processing cost due to the Coordinator and to improve the scalability of the system, the load-balancing decision making consumes as few resources as possible. It is the responsibility of the Storage Sites to determine if they are overloaded and to request migration from the Coordinator. When a site's load surpasses a predefined threshold for a specified period of time, the site issues a *Migration Request* message to the Coordinator. This global load threshold can be adjusted dynamically and directly affects the number of control messages exchanged between Storage Sites and the Coordinator. If a site is denied the request, it increments this threshold, so as to increase the interval when the next request would be sent (if it continues to be overloaded).

When a Coordinator receives a *Migration Request* message, it initiates the load-balancing algorithm. In keeping with our goal of minimal processing at the Coordinator, this algorithm is designed to make decisions fast. It finds the minimum and maximum loaded Storage Sites, and compares their load spread to a system-wide threshold. This threshold controls the imbalance level that the system is willing to tolerate before any migration actions are taken. If the load spread is below the threshold, the site which sent the request is informed that no migration will occur. Otherwise, the Coordinator selects a destination client, which is to receive portion of the data indexed by the requesting Storage Site. The client with minimal load is selected to receive data from the requesting Storage Site, unless it is already in the process of receiving data from another site.

If the Master's migration policy is based on temporal clustering the load-balancing algorithm must also consider the jurisdiction intervals assigned to Storage Sites. In that case, the destination client selected has a minimal load and a jurisdiction interval that is as close as possible to the requesting Storage Site's jurisdiction. Thus, the requestor will relinquish portion of its jurisdiction, along with the data in that jurisdiction, to the destination client.

When the Coordinator has granted a migration authorization, the overloaded Storage Site must carefully select data for migration. To accomplish this data selection efficiently, Storage Sites store statistics on the load distribution of the nodes within their R-Trees. These statistics consist of the read/write frequency of each node and reflect the current query workload. Thus, if queries are skewed, Storage Sites will be able to pinpoint the "hot" data, and will select portion of that data for migration. The amount of data to migrate is related to the workload skew and is proportional to the desired load reduction relative to the current load. The desired load reduction is half of the difference between the overloaded Storage Site's load and the destination client's load. The data selection process consists of traversing the tree along the "hottest" path, until a subtree is reached which contains enough data for migration to achieve the desired load reduction. This entire subtree is transferred to the destination client and is deleted from the overloaded Storage Site.

For skewed workloads where most queries target a small subset of data, it is beneficial to have this data distributed across all participating Storage Sites. Furthermore, since the Storage Sites index historical data, it is likely that interval or timestamp queries will exhibit a noticeable spatial and/or temporal skew. This gives the Coordinator a significant benefit in distributing the set of currently queried data over the sites. Our data selection technique correctly identifies the hot data, and migrates portions of it to other Storage Sites. As workload patterns change, the system dynamically adapts to optimize its performance by increasing the level of parallelism. Even though this process involves a certain overhead, in an imbalanced system the expected performance increase can easily outweigh the overhead. Thus, in addition to supporting high update rates, our system provides improved performance for concurrent queries on historical data.

An important consideration in correctly accomplishing our load balancing scheme is the definition of *load*. Even though it is common to consider disk I/O rates, CPU load, process queue length, and similar performance information as load criteria, we resort to a more readily available measurement: the number of elements retrieved per unit of time. For this information to be meaningful across all Storage Sites, it is necessary to consider it relatively to the maximum capacity of each Storage Site. The maximum capacity is the maximum number of elements that a Storage Site can retrieve per unit of time. This normalization guarantees that the load values are relevant even in a system composed of heterogenous Storage Sites. Given that a Storage Site contains sufficient data, the maximum capacity can be measured by requesting all data indexed by this site. This procedure can be performed prior to the system's "go-live," and does not need to be repeated. Thus, we define *load* as the number of elements retrieved per unit of time as a percentage of the maximum number of elements a Storage Site can retrieve per unit of time. Defining load in such a way indirectly captures a workstation's intrinsic hardware performance characteristics.

Thus, Storage Sites monitor their own load and when it overcomes a preset limit they issue a *Load Update* message to the Coordinator. The Coordinator keeps a table of each Storage Site's load and timestamp of when the information was last updated (see Figure 3). Thus, the Coordinator can explicitly request a load update if it considers any of the data in the load table to be stale. Updating the load in such a way (as opposed to periodic updates) reduces the number of messages transmitted between the Coordinator and Storage Sites; *Load Update* messages are only exchanged when necessary. In

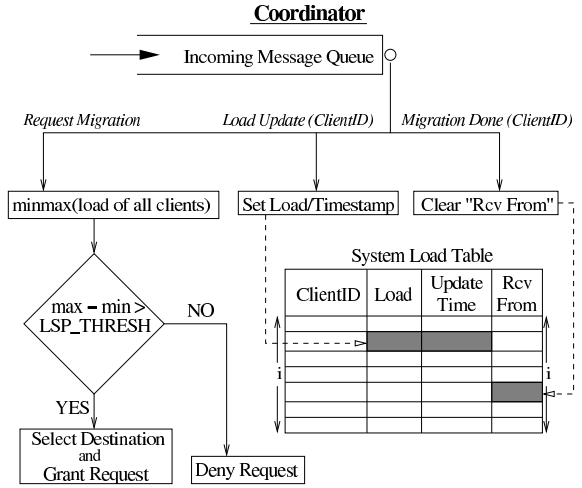


Fig. 3. The Coordinator maintains a global view of the system's load distribution. LSP_THRESH is the load spread threshold which controls the level of load imbalance the system can tolerate.

addition, the load table contains information on the current migration status of Storage Sites. During the load-balancing decision making this information is used to remove Storage Sites currently involved in data migration from the set of potential data recipients, so as to avoid selecting them as destination clients when migration is requested.

With the Coordinator in place, the Master can migrate expired data according to its preferred policy, while the Coordinator maintains the data distribution optimal for the current workload. While the Master provides optimal indexing for the current version of the data, the constant interaction between the Coordinator and Storage Sites results in superior performance for queries involving past versions of the data. An additional benefit of using the Coordinator is that it enables a zero-administration up-scaling scheme of the system. When more resources are needed, the system's capacity can be increased simply by attaching additional Storage Sites. The Coordinator will detect the imbalance since the new sites' load will be 0% and will issue data migrations to these sites allowing them to participate in the maintenance of data, effectively reducing the overall system load.

III. Experimental Evaluation

In this section we present a preliminary experimental evaluation of our prototype system and algorithms.

A. Setup

We implemented both a main memory MVR-Tree and an external memory R*-Tree as described in [24] and [3] respectively, using the GNU C++ compiler. We used TCP sockets for all necessary communication between cluster workstations, and POSIX threads for concurrency. We setup a cluster of 8 workstations connected via a high-speed switch and designated one machine as the Master and Query server, and the rest as Storage Sites. All machines had Pentium 4 1.4GHz CPUs. Storage Sites had 512 Megabytes and the Master 1 Gigabyte of main memory. All workstations were running Linux.

We generated a synthetic workload of vehicles moving on the freeway system of Illinois. The simulation lasted 100 time instants (every time instant corresponds to 1 minute) and involved 500,000 vehicles that issued updates every few time instants. A total of almost 10 million updates were generated throughout the experiment, which is equivalent to an average of 100,000 location updates per time instant. Archiving the history of all updates corresponds to a 100-fold increase in space requirements, when compared with keeping the most current state of the dataset only. We also generated synthetic range query workloads with various characteristics, which were injected in the system during the simulation. The workloads contained 10 queries per time instant, while the area covered by the query was 0.25% and 1% of the total universe space, and the time intervals spanned from 5 to 20 time instants.

We compared the system against the straightforward approach, namely, a single-CPU running a main memory MVR-Tree with historical data persisted to secondary storage. We let the two systems yield the maximum processing rate of incoming updates and query requests — that is, we buffered a large number of operations in main memory and let the trees consume them sequentially with the fastest possible rate. In the end, we measured the average processing time per operation (we kept the systems idle every time the buffer had to be re-populated).

B. Scalability

We run a scalability experiment that measures the average cost per update operation with increasing number of Storage Sites. We implemented the round-robin (RR) and the proximity-clustering (PC) migration policies. The results are shown in Figure 4. The main memory parallel MVR-Tree can sustain up to 2000 updates per second (i.e., 0.5msecs per update) with the round robin policy. In comparison, the traditional single-CPU MVR-Tree can support only one quarter of that. The proximity-clustering policy performs poorly for any number of Storage Sites, canceling out the gains offered by the parallel architecture, due to excessive per site computations and increased message exchange which contributes to heavier network traffic. For our prototype system, the cost per update operation does not improve when increasing the number of Storage Sites beyond 3. One reason is that the main memory MVR-Tree on the Master site can handle most of the updates very fast, with the given dataset. We are planning to test the system using heavier data workloads.

C. Query Efficiency

To test the querying efficiency of the proposed system we measured the *average response time* (Response), i.e., the elapsed time between the submission of a query request and when the first query results returned to the user, and the *average total time* (Total), which is the time required to fetch all data satisfying a query request. The results are shown in Figures 5 and 6 for all query workloads. We used the round-robin policy for these experiments. In the figure, we also plot the average response time for the single CPU MVR-Tree

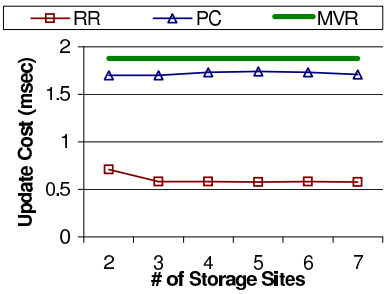
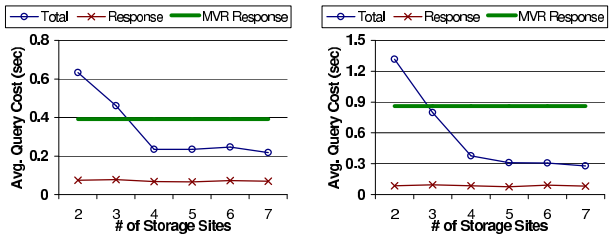


Fig. 4. Scalability experiment.

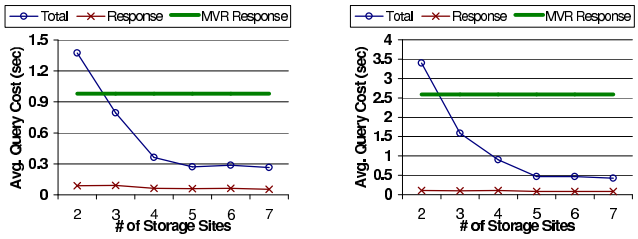
(MVR Response); for clarity, we omit the average total time since it was very close to its response time (the main reason being the absence of network message exchange overhead).



(a) 0.25% extent. (b) 1% extent.

Fig. 5. Small period queries.

Intuitively, for larger query sizes the total query cost becomes higher. Nevertheless, it drops substantially when the number of Storage Clients increases, which makes the system very scalable even for large result sizes that need to fetch many data. In comparison, the single CPU MVR-Tree offers very large response times, even larger than the total query time of the parallel architecture, for more than 3 sites.



(a) 0.25% extent. (b) 1% extent.

Fig. 6. Large period queries.

D. Query Responsiveness

Figure 7 shows query response times for a COW with 7 Storage Sites and increasing query time intervals. A significant observation is that the average response time per query is very small (4 to 32 times smaller in comparison to the single CPU MVR-Tree), and remains practically unaffected by the query size. This suggests that the average time that a user has to wait before receiving the first query results is a system specific constant which is independent of the query characteristics. The same does not hold for the single CPU MVR-Tree, where the average response time degrades fast with increasing query time interval sizes. Another observation is that the response time does not improve with increasing number of Storage Sites. Response time is affected mainly by the height of the R-Trees. If the number of sites becomes large enough such that, after re-distributing the data, the height of the structures becomes smaller, we expect the query response time to improve.

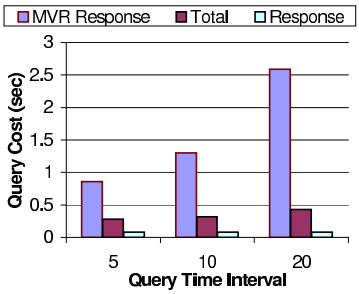


Fig. 7. Query responsiveness.

E. Closing Remarks

It is evident that the shared-nothing MVR-Tree approach offers a lot of potential for designing a system that can sustain very high update rates, while, at the same time, ensures query responsiveness. Spatio-temporal management systems that can keep up with current application needs have not been developed or designed yet. A scalable system like the one proposed in this paper is a promising approach that can guarantee successful integration of spatio-temporal access methods in highly dynamic environments. The proposed architecture constitutes indeed a highly efficient spatio-temporal framework that encompasses both historical and on-line query capabilities.

IV. Related Work

Previous research has addressed various aspects of the distributed indexing problem. The adaptive parallel B+-Tree (AB+-Tree) is proposed in [16] as a multi-tier architecture which preserves the global height of the tree. The global-height balance is required due to the sequential ordering of the leaves which imposes a logical ordering of the distributed sites as well. This restricts migration of data to occur only between sites which are logically adjacent. In a follow-up of the AB+-Tree idea, an R-Tree version is presented in [18] where a simulation study is used to analyze the effectiveness of the proposed mechanism. This mechanism involves a two-tier scheme composed of a Master site which holds information about the roots of client's R-Trees and their loads and makes periodic evaluations of the balance of load in the system. If the Master deems the system to be imbalanced, it composes lists of destination and source clients, the amount of data to be migrated, and broadcasts these data to all clients.

A shared-memory environment is discussed in [17]. In [9] a combination of indexing and hashing is applied for distributed

indexing of one-dimensional data. In [15] a distributed search tree is introduced as an improvement to distributed linear hashing methods. A striping and load-balancing mechanism adaptable to changing access patterns is proposed in [22] in the context of parallel disk systems. A "semi-distributed" version of R-Trees is described in [13] and optimal data sizes and response times are analytically and experimentally derived. A single-processor parallel-disk version of the R-Tree is proposed in [11]. An improvement on this work is presented in [23] where a master-client R-Tree indexes data in a shared-nothing environment. Finally, a distributed R*-Tree based mechanism for indexing multidimensional data in a cluster of workstations is introduced in [14] where load balancing is achieved through data migration and algorithms based on access statistics. The main difference with this work is that we index both past and present versions of the spatio-temporal data and, in addition, we use an MVR-Tree on the Master in order to cluster data more efficiently in both space and time.

V. Conclusions

In order to satisfy our requirements, we opted for a shared-nothing infrastructure using a cluster of workstations. We introduced a robust spatio-temporal management system based on an MVR-Tree server and multiple R-Tree storage clients. Our proposed system is tailored for highly dynamic environments that demonstrate varying query access patterns and high update rates. The system is able to sustain predictable response times to user requests under heavy loads. Its main feature is the ability to store the complete update history of the spatio-temporal data giving the ability to users to pose queries about both past and present states of the data. We showed experimentally that the proposed system can live up to our expectations, sustaining thousands of updates per second without substantial performance degradation. We plan to extend the system to support predictive queries and investigate the implications of various sophisticated load-sharing approaches and their effect on user queries.

References

[1] SkyServer. http://skyserver.sdss.org/dr1/en/.

[2] The Human Genome Project. http://www.ornl.gov/sci/ techresources/Human_Genome/home.shtml.

[3] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of ACM Management of Data (SIGMOD)*, pages 220–231, 1990.

[4] V. Calhoun, T. Adali, and G. Pearlson. (Non)stationarity of Temporal Dynamics in fMRI. In *Annual Conference of Engineering in Medicine and Biology*, volume 2, October 1999.

[5] J. Driscoll, N. Sarnak, D. Sleator, and R.E. Tarjan. Making data structures persistent. In *Proc. of the ACM Symposium on Theory of Computing*, 1986.

[6] The earth observing system data and information system. http://spsosun.gsfc.nasa.gov/ eosinfo/EOSDIS_Site/index.html.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM Management of Data (SIGMOD)*, pages 47–57, 1984.

[8] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *Proc. of Extending Database Technology (EDBT)*, pages 251–268, 2002.

[9] T. Honishi, T. Satoh, and U. Inoue. An index structure for parallel database processing. In *IEEE International Workshop on Research Issues on Data Engineering*, pages 224–225, 1992.

[10] I. Kamel and C. Faloutsos. Parallel r-trees. In *Proc. of ACM Management of Data (SIGMOD)*, pages 195–204, 1992.

[11] I. Kamel and C. Faloutsos. On packing r-trees. In *Proc. of Conference on Information and Knowledge Management (CIKM)*, pages 490–499, 1993.

[12] G. Kollios, V. J. Tsotras, D. Gunopulos, A. Delis, and M. Hadjieleftheriou. Indexing animated objects using spatiotemporal access methods. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 13(5):758–777, 2001.

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

[14] V. Kriakov, A. Delis, and G. Kollios. Management of Highly Dynamic Multidimensional Data in a Cluster of Workstations. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT)*, Heraklion, Greece, March 2004.

[15] B. Kroll and P. Widmayer. Distributing a search structure among a growing number of processors. In *Proc. of ACM Management of Data (SIGMOD)*, pages 265–276, 1994.

[16] M. Lee, M. Kitsuregawa, B. Ooi, K. Tan, and A. Mondal. Towards self-tuning data placement in parallel database systems. In *Proc. of ACM Management of Data (SIGMOD)*, pages 225–236, 2000.

[17] G. Matsliach and O. Shmueli. An efficient method for distributing search structures. In *International Conference on Parallel and Distributed Information Systems*, pages 159–166, 1991.

[18] A. Mondal, M. Kitsuregawa, B. C. Ooi, and K. L. Tan. R-tree-based data migration and self-tuning strategies in shared-nothing spatial databases. In *Proc. of ACM Symposium on Advances in Geographic Information Systems (GIS)*, pages 28–33, 2001.

[19] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *The VLDB Journal*, pages 395–406, 2000.

[20] K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying mobile objects in spatio-temporal databases. In *Proc. of Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 59–78, 2001.

[21] B. Salzberg and V.J. Tsotras. A comparison of access methods for temporal data. *ACM Computing Surveys*, 31(2), 1999.

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

[23] B. Schnitzer and S. Leutenegger. Master-client r-trees: A new parallel r-tree architecture. In *Proc. of Scientific and Statistical Database Management (SSDBM)*, pages 68–77, 1999.

[24] Y. Tao and D. Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *The VLDB Journal*, pages 431–440, 2001.

[25] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Multiversion linear quadtree for spatio-temporal data. *Proc. of ADBIS*, pages 279–292, 2000.

[26] T.L. Zeiler. LANDSAT Program Report 2002. Technical report, U.S. Geological Survey - U.S. Department of Interior, Sioux Falls, SD, 2002.