# Spatio-Temporal Data Services in a Shared-Nothing Environment

Marios Hadjieleftheriou
University of California, Riverside
marioh@cs.ucr.edu

Vassil Kriakov
Polytechnic University
vassil@milos.poly.edu

Yangui Tao
Boston University
ygtao@cs.bu.edu

George Kollios
Boston University
gkollios@cs.bu.edu

Alex Delis
The University of Athens
ad@di.uoa.gr

Vassilis J. Tsotras
University of California, Riverside
tsotras@cs.ucr.edu

## Abstract

*Recently, there has been a proliferation of applications that produce spatio-temporal data that has to be processed, stored and queried efficiently. These applications necessitate the execution of millions of updates in order 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* present *as well as* past *(historical) data versions in an on-line fashion. We propose a system that exploits the inherent parallelism of a shared-nothing computing environment for storing and indexing the spatio-temporal data. We describe our proposed system architecture, data organization, and outline techniques for ensuring robustness and scalability under excessive query loads and high update rates.*

## 1 Introduction

Managing update intensive spatio-temporal data is becoming a daunting task. With recent advances in data collection applications, spatio-temporal data is produced in a massive scale, having to be processed, stored and queried efficiently. Advances in sensor network and wireless technologies have contributed substantially to the creation of datasets that have to be frequently updated. Similarly, with scientific applications producing Terabytes of raw data per day, the efficient execution of millions of update operations is essential for maintaining spatio-temporal datasets up-to-date.

The common denominator of all aforementioned applications is that their underlying access methods should be able to support a large number of update operations per second for keeping the data up-to-date. We argue that it is unrealistic, in such environments, to deploy any known disk-based spatio-temporal access methods as the latter would pose a bottleneck for update-intensive operations.

In addition, a core requirement of a wide range of spatio-temporal applications is to store historical information. Users should be given the ability to query any (past or current) versions of the data efficiently. Due to the large aggregate size of historical information, past data versions should reside on secondary storage and should be indexed using an efficient access method. Storing the data on a single workstation, given large update rates, is impractical. Alternatively, it is more natural to distribute them among multiple, cooperating sites.

We propose an infrastructure that utilizes the inherent parallelism of a shared-nothing architecture 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 the applications to archive past versions of the spatio-temporal data using a very cost-effective approach. Finally, large volumes of query requests about current or past versions of data can be processed efficiently in a decentralized collaborative fashion. A salient feature of the proposed 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 sites guarantees robust handling of diverse spatio-temporal query types, yielding the best possible performance by taking advantage of multiple processing units.

The main contributions of the paper are: i) a spatio-temporal data management system that can support very high update rates to the most recent version of the data; ii) a system that stores all past versions of the data and efficiently supports concurrent queries with varying access patterns to any ver-

This work was partially supported by NSF grants IIS-9907477, EIA-9983445, IIS-0220148, IIS-9733642, Career Award 0133825, and US Dept. of Education P200A300-100.

sion of the data; iii) dynamic load-balancing techniques for distributing data efficiently across the cluster in order to minimize query response times; and finally, iv) an experimental evaluation of the proposed architecture using a system prototype.

## 2 System Architecture

In this section we discuss in detail the proposed spatio-temporal data management system and outline the infrastructure components. The requirements from the system are: i) the ability to query both present and past versions of the stored data in an on-line fashion; ii) sustaining very high update rates with minimal performance and responsiveness degradation; iii) capability to dynamically adapt to voluminous queries that might create excessive traffic and congest only parts of the system.

Our framework utilizes a *Master* site whose key objective is to index the recent history (including the current versions) of the spatio-temporal data. We attain this goal by using a *Multi-Version R-Tree (MVR-Tree)* [6, 14].

The multi-version approach [1] is a typical method employed for storing multiple versions of spatio-temporal data. Consider an ephemeral R-Tree [2] indexing the current version of a spatio-temporal dataset. As updates occur, the R-Tree evolves by applying the corresponding data 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. Data records in the leaf nodes of an MVR-Tree maintain the temporal evolution of the ephemeral R-Tree data objects. Each record is thus extended to include the lifetime of the object. The tree clusters 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. This is achieved by using various policies to determine whether a specific node should artificially expire, by copying only the live entries to newly created nodes. (A detailed analysis of the structure can be found in [6].)

The MVR-Tree enables us to keep a small fragment of the spatio-temporal data at the *Master* (the most recent versions) and distribute older versions to other sites. It achieves that goal by clustering data that are adjacent in space and time in a small number of tree nodes. Tree nodes that contain only expired data versions are migrated over the network to other sites. Furthermore, we store the MVR-Tree in main memory, guaranteeing that very high update rates can be sustained by avoiding disk accesses on the *Master*.

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 make the majority of cluster workstations *Storage* sites, that are responsible for archiving older data versions in secondary storage and handling queries about data under their jurisdiction. In our environment we want to support multiple concurrent users issuing possibly overlapping queries, hence, we choose to deploy R-Trees instead of MVR-Trees on the *Storage* sites for the following reasons: i) 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; ii) R-Trees have a smaller directory structure (with larger fanout) that is query and update efficient and introduces smaller space overhead; and iii) R-Trees are more robust than MVR-Trees for long period queries.

We also employ a *Query Server* that is responsible for distributing user queries to the appropriate sites and for helping compile the results that are finally furnished to the users.

A workstation can be selected to play any or all of the above roles. Depending on system load and the state of the network reassigning roles aids in avoiding/overcoming bottlenecks.

### 2.1 Migrating Data to Storage Sites

Our networked approach is based on migrating all tree nodes that contain expired versions of the data to *Storage* sites. Essentially, we use the MVR-Tree policies to determine when a node should migrate. 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 migration policies: i) round-robin; ii) least-loaded *Storage* site for update efficiency; iii) proximity-clustering where data is uniformly spread out across the cluster, and iv) 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 this architecture, ideally we would like to place the data in such a way 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. 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, in the long run, the nodes will be uniformly distributed. 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. The Least-loaded *Storage* site policy tries to commit the mi-

grated data as soon as possible in order to relieve the *Master* from the responsibility of that data. Proximity-clustering was proposed in [4]. This algorithm distributes leaf nodes intelligently based on proximity measurements between nodes. 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 proximity heuristics.

### 2.2 Sustaining High Update Rates

The major characteristic of update intensive operations is that disk I/O dominates the update cost. State of the art networked systems that use disk-based access structures are bounded in terms of both the number and the volume of updates that they can sustain [8]. Furthermore, 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 use much slower random accesses for loading disk pages in main memory. Indicatively, random access of a single page requires 5msec, sequential access of multiple pages is 12 times faster and one memory access needs 50nsec, an 100,000 increase in speed. 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. We choose to keep the MVR-Tree in the main memory of the *Master* site to take advantage of all previous arguments. In addition, since we place older data in *Storage* sites, we anticipate the average cost of updates to decrease substantially, since multiple CPUs amortize the cost, and, at the same time, we guarantee that a large percentage of the most recent version of the MVR-Tree can fit in the main memory of the *Master*.

### 2.3 Dynamic Load-Balancing

A distributed architecture by itself may not be sufficient in providing excellent performance for queries involving past versions of the data. In order to improve the performance gain due to concurrent processing it is necessary to guarantee that the largest possible number of *Storage* sites participate in answering a query. Thus, it is desirable that the load distribution among sites is uniform. Since the *Coordinator* monitors the load of all *Storage* sites and performs dynamic resource allocation among nodes, it can also undertake the function of load sharing. To achieve the desired load adjustments, the *Storage* sites perform additional data migrations. This is feasible since the leaf nodes or whole subtrees of an R-Tree can easily be removed and inserted in another tree. It is the responsibility of the *Storage* sites to determine if they are overloaded and to issue migration requests when the load surpasses a predefined threshold for a specified period of time.

When a *Coordinator* receives a *Migration Request*, it initiates the load-balancing algorithm which finds the minimum and maximum loaded *Storage* sites, and compares their load spread to a system-wide threshold. 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 will receive portion of the data indexed by the requesting *Storage* site. More details about the load-balancing procedures appear in the full version of the paper.

## 3 Experimental Evaluation

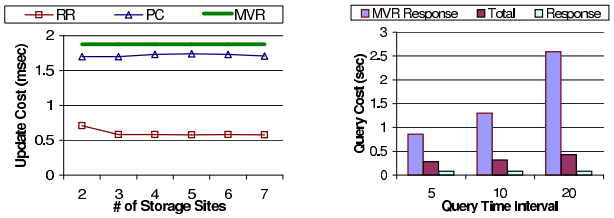
In this section we present a preliminary experimental evaluation of our prototype system and algorithms. We implemented both a main memory MVR-Tree and an external memory R-Tree. We setup a cluster of 8 workstations connected via a high-speed switch and designated one machine as the *Master* and *Query* server, and the rest as *Storage* sites. We generated a synthetic workload of vehicles moving on the freeway system of Illinois. The simulations 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 version of the dataset only. We also generated synthetic range query workloads with various characteristics, which were injected in the system during the simulation.

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 buffered a large number of operations in main memory and let the trees consume them sequentially with the fastest possible rate. Then, we measured the average processing time per operation (we kept the systems idle every time the buffer had to be re-populated).

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 1(a). 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. We used the round-robin policy for the rest of our experiments.

To test the querying efficiency of the proposed system we measured the *average response time* (Response) and the *average total time* (Total) (the time required to fetch all data sat-

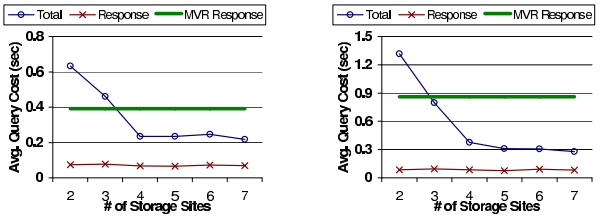
isfying a query request). Figure 1(b) shows query response times for a COW with 7 *Storage* sites and increasing query time intervals. We observe that the average response time per query is very small (4 to 32 times smaller in comparison to the single-CPU MVR-Tree), and remains practically unaffected by the length of the query time interval. The same does not hold for the single-CPU MVR-Tree, where the average response time degrades substantially.




(a) Scalability experiment. (b) Query responsiveness.

**Figure 1.**

Figures 2(a) and 2(b) show querying efficiency for varying numbers of *Storage* sites and time interval queries that cover 0.25% and 1% of the data universe, respectively. In the figures, we also plot the average response time for the single CPU MVR-Tree (MVR Response).




(a) 0.25% interval queries. (b) 1% interval queries.

**Figure 2.**

Intuitively, for larger query sizes the total query cost becomes higher. Nevertheless, it drops substantially when the number of *Storage* sites increases.

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.

## 4 Related Work

Previous research has addressed various aspects of the distributed indexing problem. The adaptive parallel B+-Tree (AB+-Tree) is proposed in [10]. In a follow-up of the AB+-Tree idea, an R-Tree version is presented in [11]. In [3] a combination of indexing and hashing is applied for distributed indexing of one-dimensional data. In [9] 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 [12] in the context of parallel disk systems. A "semi-distributed" version of R-Trees is described in [7]. A single-processor parallel-disk version of the R-Tree is proposed in [5]. An improvement on this work is presented in [13]. 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 [8].

## 5 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.

## References


[1] J. Driscoll, N. Sarnak, D. Sleator, and R.E. Tarjan. Making data structures persistent. In *STOC*, 1986.

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

[3] T. Honishi, T. Satoh, and U. Inoue. An index structure for parallel database processing. In *RIDE*, pages 224–225, 1992.

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

[5] I. Kamel and C. Faloutsos. On packing r-trees. In *CIKM*, pages 490–499, 1993.

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

[7] N. Koudas, C. Faloutsos, and I. Kamel. Declustering spatial databases on a multi-computer architecture. In *EDBT*, 1996.

[8] V. Kriakov, A. Delis, and G. Kollios. Management of Highly Dynamic Multidimensional Data in a Cluster of Workstations. In *EDBT*, Heraklion, Greece, March 2004.

[9] B. Kroll and P. Widmayer. Distributing a search structure among a growing number of processors. In *SIGMOD*, pages 265–276, 1994.

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

[11] 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 *ACM GIS*, pages 28–33, 2001.

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

[13] B. Schnitzer and S. Leutenegger. Master-client r-trees: A new parallel r-tree architecture. In *SSDBM*, pages 68–77, 1999.

[14] Y. Tao and D. Papadias. MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB*, pages 431–440, Roma, Italy, September 2001. Morgan Kaufmann.