# Concurrency Control in Distributed MRA Index Structures

Neha Singh* S. Sudarshan

Indian Institute of Technology Bombay
Mumbai, India
s.neha@bcg.com, sudarsha@cse.iitb.ac.in

## Abstract

Answering aggregate queries like *sum*, *count*, *min*, *max* over regions containing moving objects is often needed for virtual world applications, real-time monitoring systems, etc. Since the data set is usually very large and some queries require significant processing resources, quite often such data is stored in a distributed system wherein each system handles a partition of the whole space and manages all objects in that partition. Objects keep switching from one system to another as they change their location. Currently there are no known efficient techniques for getting aggregates over moving objects while ensuring that their position updates remains atomic to the read.

We introduce an efficient technique for finding aggregates over mobile objects with data stored in a distributed system by extending the multi-resolution aggregate trees to work in a distributed system and over mobile objects. We propose a multi-phase update algorithm that is highly concurrent with respect to read queries and at the same time makes modifications to the aggregate tree atomic with respect to read queries. The analysis and experimental results prove the efficacy of this update algorithm as compared to the naive update technique.

## 1 Introduction

We address the problem of answering aggregate queries over a region in a multi-dimensional space containing mobile point data, when the data is stored in a distributed system. Such aggregate queries occur commonly in spatial applications like Networked Virtual Environment(NVE), real-time traffic monitoring, etc. In NVEs, the game characters, either the human character of the computer generated characters can be treated as point data. Aggregation is an important operation required in NVEs for either getting summary data or for getting information about the surrounding game state to make decision about the next step. In many cases, state information needed in NVEs is in the aggregate form rather than raw data. For instance scripts like, "run in fear if the number of enemies exceed the number of friends around" or "find the minimum health player", etc., require aggregates like count and min respectively on a region of the space. In many real-time applications, such as traffic monitoring, it is quite often required to get aggregates over a region, for example the count of the number of cars in a particular region.

In a centralized system, we can address this issue by building a dynamic index for each type of aggregate as done in [8]. In a distributed or peer-to-peer computing system however, the data now is split across various systems. We may need to read from multiple systems to answer a query; the data that they store is subject to concurrent updates as data points change their location. This brings us to the question of how the data points on a given space are mapped onto the end systems or peers.

We observe that most such spatial applications require locality of data to be preserved and hence split the data by partitioning the space and mapping the partitions onto the peers. This partitioning is not static and changes as the hot spots change, by splitting of nodes. Every peer has a local state and they communicate asynchronously with each other by passing messages over the communication channel. As entities move from one node's region to another, they are transferred using messages. Thus the problem of getting the aggregate over a sub space reduces to getting partial aggregates from multiple peers and combining them, taking concurrent updates into account, to get a consistent result.

In a distributed computing system absence of a global common memory or clock makes it difficult to record the global state of the system efficiently. Distributed snapshots can be obtained by recording the

* Current affiliation: Boston Consulting Group

*International Conference on Management of Data*
*COMAD 2008, Mumbai, India, December 17–19, 2008*

local state of the different relevant peers. However, even if peers read the time from a single common clock (maintained at one process), various indeterminate transmission delays during the read operation will cause the peers to identify various physical instants as the same time. Thus, inevitably the collection of local state observations will be made at different times, with the possibility of intervening updates. Thus recording a consistent global state of a distributed system is not a trivial task.

In this paper, we describe an efficient method to get spatial aggregates over mobile data points. We extend the multi-resolution tree, wherein aggregates are stored with decreasing resolutions as the depth of the partitioning tree increases, to support a dynamic object set. Our read query and aggregate tree update protocols ensure that updates are atomic to the read. We first present the naive update protocol and then a highly concurrent multi-phase update protocol. We then present the analysis and experimental results to substantiate our claims.

The rest of the paper is organized as follow. In Section 2, we present some related work and in Section 3, the system model. In Section 4, we describe the reader's protocol. In Section 5, we talk about the update protocol required to maintain the distributed aggregate tree and the data points move. Here, we first describe the naive update protocol and then the highly concurrent multi-phase update protocol. In Section 6, we present a simplified update protocol for a very common special case of updates. Then we show the analytical results comparing the concurrency of described update protocols. In Section 7, we present our experimental results, and we conclude in Section 8.

## 2 Related Work

Several tree structures containing aggregate information at internal nodes have been proposed for efficiently answering aggregate queries on spatial regions, with relatively static data; see e.g. Tao and Papadias [7]. The Multi-Resolution Aggregate (MRA) tree [4] is a recent example in this class of structures, which addresses the issue of generating progressively more accurate approximations to the aggregate result. However [7], [4], as well as other earlier structures, consider a centralized architecture, and assume a relatively low update rate.

Tanin et al. [5] propose a distributed quad-tree structure for a peer-to-peer environment. We build upon their approach for constructing a distributed quad-tree, and add aggregate information. However, unlike our work, they address neither aggregate queries, nor concurrency control issues. Gao et al. [3] address several different kinds of aggregation combined with selection, using distributed quadtrees. The problems addressed are motivated by sensor networks. However, they do not address concurreny control issues.

Many problems of atomicity and consistency arise when we assume a model with dynamic data wherein the data points are exchanged among the different systems using message communication.

Much work on tracking and getting summary data of continuously moving objects, for example in the case of vehicular traffic movement, etc., has been done in the context of spatio-temporal databases; in this body of work, objects move along predictable paths, for example as line segments at a constant speed for a period of time, and this fact is exploited to minimize index updates. Work in this area includes [2, 6, 1], all of which consider centralized settings. We consider a case where movement is not continuous and predictable, in a distributed environment. Thus the solutions proposed above do not match our model. Further, none of this work considers the issue of concurrency control.

To the best of our knowledge, there has been no work on concurrency control in index structures designed for aggregate queries over moving objects in a distributed or peer-to-peer systems.

## 3 System Model

The system consists of a distributed set of peers. These peers may be end-systems in a distributed system or peer-to-peer servers. The global space ($R_{space}$) is divided using quad-tree based spatial partitions. Quad-tree based partitioning, like most other tree-based partitionings, preserves locality of data. The advantage of quad-tree based partitioning over other candidate tree-based partitioning data structures like R-tree, KD-tree, etc., is that the partitions are independent of the order in which the data points are inserted, the decomposition implicitly known to all peers, and dynamic load balancing is relatively less expensive.

The index structure stores the point data representing the objects in the form of ⟨point_loc, state⟩. These points objects are stored in the leaf nodes of the quad-tree. Each node stores the aggregate of the region indexed by it. The leaf nodes store aggregates as $\langle sum, count, min, max\rangle$ whereas the intermediate nodes store the aggregate in the form $\langle sum, count, min_{array}, max_{array}\rangle$, where $min_{array}$ and $max_{array}$ is the list of $min$ and $max$ values for each of the child nodes. We will see later that storing this list of values rather than a single overall $min$ and $max$ helps reduce the cost of index maintenance. The number of children $n_c$ (fanout) is constant in the case of a quadtree ($n_c = 4$) but may be variable for other tree structures.

To map the nodes onto the peers, we use a hashing function. We note that each quad-tree can be uniquely identified by its centroid as described in [5]. Hence we use this as the key for hashing. Each peer stores the data points assigned to it, and optionally provides processing resources for queries.

Since the data points are mobile, in case the location of a data point moves out of a peer's region, it passes the object's data to the relevant peer using message communication. We assume these peers to be processes connected by a bidirectional channel. Message send and receive is asynchronous. They are delivered reliably with finite but arbitrary time delay. We assume FIFO ordering for the communication channel. The read and the update protocols too use message communication for getting the required aggregate data and for updating the aggregate tree respectively. The updates are processed at each node in the same order in which they are received.

## 4 Reader's Protocol

Consider an aggregate read query to get the aggregates set $\langle sum, count, min, max \rangle$ over the query region $Q \subseteq R_{space}$. It can be initiated from any peer by sending message *getAggregate* to the root node of the distributed MRA tree. The query traverses the index structure top-down using message passing and selectively exploring the nodes. The read protocol uses locks for concurrency control.

A naive way to ensure concurrency control is to acquire locks on the nodes while traversing down the tree and release the locks only after the read is complete. However keeping the nodes locked, especially the root node for the entire duration of the read, will reduce the concurrency of the index structure for concurrent updates. Thus we need to release locks early and at the same time ensure that the updates are atomic for the read. To ensure both the above conditions, we use the well known crabbing protocol. In the crabbing protocol, the root is first locked in shared mode. After acquiring locks on all required children in shared mode, the lock on the parent node is released. This prevents an update coming down from the root from overtaking a read.

Algorithm 1 describes the function invoked at each node on receiving the *getAggregate* message. Given a query region $Q$ and a node $N$, there can be the following possible relations between $Q$ and $N$ - *contained, partially overlapping, enclosing* and *disjoint* - as described in [4]. The node's aggregate is not relevant for the query in case it is disjoint with $Q$ (lines 2-3) while the aggregate over all data points of $N$ is needed in case $Q$ encloses it. In both these case further traversal is not needed. Further traversal is needed only if $Q$ is partially overlaps or is contained in $N$ (lines 8-21). The crabbing protocol is used for acquiring locks on the children nodes and releasing it (lines 19 and 21).

**Algorithm 1** Reader Protocol

```
   deliverGetAggregate(Q: QueryRegion , Qnode: QueryingNodeID)
   Let N be the current node
 1: if (N is the root node) Get S-lock
 2: if (Q ∩ R^N = Φ) then
 3:    Ignore {disjoint}
 4: else if (N is a leaf node) then
 5:    Send Agg for region Q ∩ R^N to Qnode
 6: else if (Q ∩ R^N = R^N) then
       {All data points of N contained in the query region}
 7:    Send Agg of N to Qnode
 8: else
       {Enclosing or partially overlapping. Further traversal necessary}
 9:    Let L ← List of nodes
10:    for each child N_i of N
11:       if (Q ∩ R^{N_i} = Φ) then
12:          Ignore {disjoint}
13:       else
14:          Insert N_i into L
15:    for each node N_i in L
16:       if ( (Q ∩ R^N ⊇ R^{N_i}) ∧ (Query just for min/max))
             {Use cached data of child's min/max}
17:          Return Agg of R^{N_i} to Qnode
18:       else
19:          Get S-lock on N_i
20:          Send getAggregate(Q, Qnode) to N_i
21:    Release S-lock on N
```

## 5 Maintenance of Index Structure

The points being indexed by the distributed MRA tree are mobile and hence their location may change. Also new data points may be added or deleted. All these actions require the distributed MRA tree to be updated, and the aggregates stored in the internal nodes of the MRA tree must be correspondingly updated. The update to the data point is first received by the leaf node responsible for that point. So the updates percolate from the leaf nodes to the higher levels of the index structure hierarchy. Our aim is to update the distributed MRA index such that these modifications are atomic with respect to the aggregate read queries. To understand the update protocols, we first present the following definitions. We regard each update, whether it is an insert, delete, move, as a transaction. We do not consider transactions involving multiple updates.

**Definition 1 (Update Tree):** The set of all the nodes ($U_T$) of the distributed MRA tree whose stored aggregate values can be affected by an update transaction $T$.

Note that the exact set of nodes of the distributed MRA tree whose aggregate needs to be modified may depend on the particular update transaction, and on the current aggregate values of the nodes. However the set $U_T$ contains all the nodes whose aggregate can be affected by the update transaction, whatever be the current value of aggregate in the nodes and the value propagated up by the update.

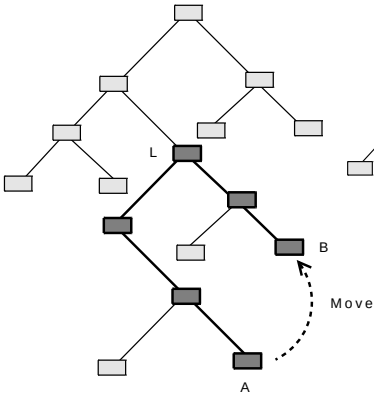


Figure 1: Update tree of a *move* operation.

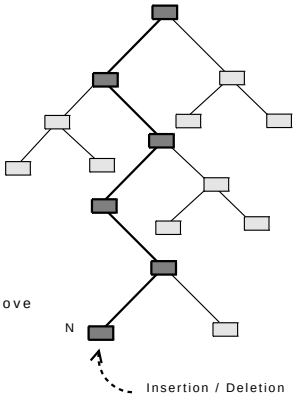


Figure 2: Update tree of an *insert* or *delete* operation.

We mainly consider two broad classes of updates. The first class consists of *insert* or *delete* operations when the data point is either added or deleted from $R_{space}$. The second class consists of the *move* operation in which the location of an object changes. The former case is simpler, and we discuss the latter case in more detail. The *move* operation is also the more common update in our target applications. There are two cases for the *move* operation. The first case is where the new location of the object is in the same leaf partition. This is simpler, and more frequent. In this case there will be no change in aggregate for any node and hence there is no need to update the MRA tree. The second case, which is the less frequent but more interesting, is when the object moves out of a nodes indexed space of one leaf and needs to be transferred to another leaf node. The nodes may be on different peers of the distributed system. In the following sections, whenever we refer to the *move* operation, we refer to the second case.

Let $N_T$ be the set of the leaf nodes affected by an update transaction. The cardinality of this set is 1 in the case of *insert* or *delete* and 2 in case of *move* operation. In both these cases, when the aggregate of the leaf node changes, the aggregate of their ancestors may need to be updated. We discuss the nature of the update tree for these two classes of updates:

- *Insert/Delete Operation:* Consider a point inserted or deleted at the node $N_{leaf}$. All ancestor nodes $N$ of $N_{leaf}$ are affected by this operation. Hence the update tree consists of a single path from the root to $N_{leaf}$ as shown in Figure 2.

- *Move Operation:* Consider a point moving from a location $L_A$ in a region belonging to leaf node $A_{leaf}$ to another location $L_B$ in a region belonging to leaf node $B_{leaf}$. Let $N_{AB}$ be the lowest common ancestor of nodes $A_{leaf}$ and $B_{leaf}$. All ancestors of $A_{leaf}$ and $B_{leaf}$ upto node $N_{AB}$ are affected by this move transaction. For all ancestors of $N_{AB}$, the move operation is just a transfer of point within the same region and hence their aggregate is not affected. This is also true for $N_{AB}$, but since it caches the *min* and *max* values of its children which can change in this operation, this node is also part of the update tree. Thus the update tree consists of two legs, one connecting $N_{AB}$ to $A_{leaf}$ and the other connecting $N_{AB}$ to $B_{leaf}$ as shown in Figure 1.

**Definition 2 (Conflicting Updates):** Two updates $U_1$ and $U_2$ are said to be conflicting updates if $U_{1T} \bigcap U_{2T} \neq \phi$, where $U_{1T}$ and $U_{2T}$ are the corresponding update trees.

The above definition basically says that two updates are conflicting if their corresponding update trees have any common nodes. For any two conflicting updates $U_1$ and $U_2$, let $P$ be the subtree formed by joining the common nodes in their update trees. We call $P$ as the *conflicting upadate's common twig pattern*. Here we discuss the nature of this pattern $P$. The nature of the graph $P$ is important we use its properties in deciding the serial order of execution of two conflicting updates. Let $N_1$ and $N_2$ be the root nodes of $U_1$ and $U_2$. Without loss of generality, let us assume that the height of $N_1$ is less than or equal to that of $N_2$. We make the following observations about the graph $P$.

- $P$ is a connected graph
- $P$ always contains of $N_1$
- $P$ is either a single path or consists of two paths rooted at $N_1$
- $N_1$ is the unique highest node of $P$

This is because the update tree is formed by connecting a leaf node to all its ancestors (in the case of addition and deletion of data points) or two leaf nodes to its ancestors until the two paths meet at a common node which is their lowest common ancestor. Given that updates $U_1$ and $U_2$ are conflicting, they have at least one common node say $N$. Now due to the nature of the update tree, all ancestors of $N$ will also be in both the updates until we reach the node in $N_1$. Ancestor nodes of $N_1$ may be in the update tree of $U_2$, but are not in the update tree of $U_1$. Thus $P$ contains a path rooted at $N_1$. Similarly if the updates $U_1$ and $U_2$ both have two legs and have a common node in the other leg also, all its ancestor nodes up to $N_1$ will also be in $P$. Thus $P$ can have a single path or two paths rooted at $N_1$ and so $P$ is a connected graph. Also P cannot have any node higher than $N_1$ since $U_1$ does not have any node higher than that in its update tree.

Thus $N_1$ is the unique highest node of $P$. We use the order of lock point at this unique highest node of $P$ – $N_1$ – for deciding the serial order between two conflicting updates.

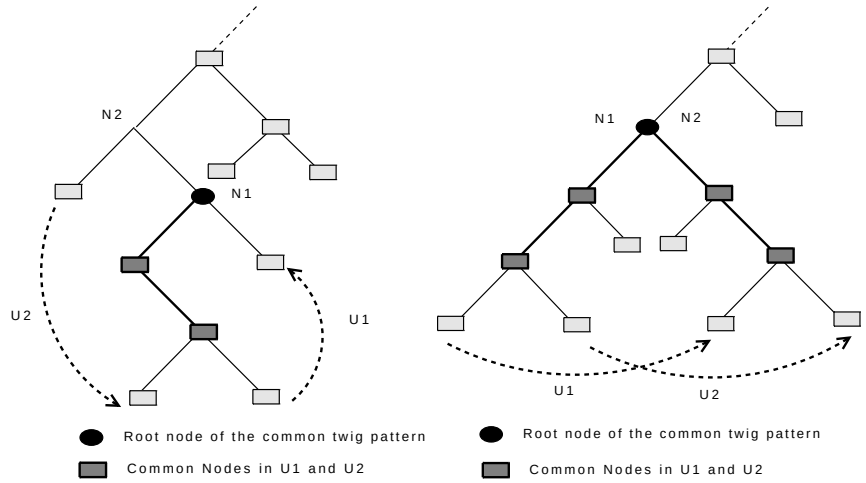


Figure 3: Common twig pattern of two conflicting updates (a) having one leg (b)having two legs

In the following sections, we present two algorithms for maintaining the distributed aggregate index structure such that the updates appear atomic to concurrent read queries.

### 5.1 Naive Update Protocol

In both the update protocols that are described in this section, we use locking for concurrency control. For acquiring and releasing locks, we use the concept of the update tree. Update tree represents all the nodes that can be affected by that transaction. Thus the naive way is to get an X-lock on all the nodes of this tree and then update them. Now there are two ways to acquire the locks – either top-down or bottom-up the update tree. Acquiring the lock bottom-up can lead to a deadlock with the read query coming from top-down. Hence we acquire the locks top-down. The naive update protocol consists of two main phases called the *acquire lock phase* and the *update phase*. As the name suggests, locks are acquired on the nodes top-down iin the update tree in the *acquire locks phase*. In the *update phase*, the aggregate tree is updated bottom-up and locks released. Bottom-up propagation of updates is essential for aggregates like *min* and *max*.

Consider a point that moves from a location $L_A$ in a region belonging to leaf node $A_{leaf}$ to another location $L_B$ in a region belonging to leaf node $B_{leaf}$. Let $N_{AB}$ be the lowest common ancestor of nodes $A_{leaf}$ and $B_{leaf}$. All ancestors of $A_{leaf}$ and $B_{leaf}$ up to node $N_{AB}$ are affected by this move transaction as explained in the previous section. We now describe the naive protocol to update the aggregate tree for the *move* operation. *Insert* and *delete* are simpler cases and discussed later. The steps are as follows. $A_{leaf}$ sends the transfer message with object data to $B_{leaf}$. $B_{leaf}$ sends an *updateInit* message to the root node of the corresponding update tree (node $N_{AB}$). After getting this message, the root node starts the *acquire lock phase*. In this phase, X-locks are acquired on all nodes of the update tree top-down starting with the node $N_{AB}$ using messages. The root node first acquires an X-lock on itself and then sends the message getNaiveXLock(*updateId*, $A_{leaf}$) and getNaiveXLock(updateId, $B_{leaf}$) to its children in the path to leaf nodes $A_{leaf}$ and $B_{leaf}$ respectively. Sending the ids of the leaf nodes is important as the intermediate nodes of the update tree use this information to direct their messages along the path to that leaf node. In our case, the node's id consists of its coordinates and hence it is easy to find the child containing that leaf node.

On receiving the *getNaiveLock* message, each node invokes the *deliverGetNaiveXLock* function as described in Algorithm 2.

When this message reaches the leaf nodes, they start the *update phase*. In this phase, the update to the aggregate tree propagates bottom-up. The nodes update their values and release locks. The root node however releases the lock only after the update phase is over in all the legs of the update tree. It uses the map *deliverCount* to keep track of the number of update messages received with a given *updateId*. Note that the update phase may start at different points in time in the different legs of the update tree depending on their length.

For *insert* and *delete* operations, the root node of the update tree, which is also the root node of the aggregate tree, sends *getNaiveXLock* to its child along the relevant leaf node. The rest of the steps are same as in the *move* case.

Our locking algorithm uses messages to acquire and release locks. Each message invokes the corresponding *deliver* function at the nodes. Usually locks are held by processes, but in our case, there is no one process which is alive for the entire duration of the update at each node. Hence we hold locks for a given *updateId*. Each update has a unique *updateId* which can be generated by appending a monotonically increasing inte-

**Algorithm 2** Acquire Lock Phase

**deliverGetNaiveXLock**(*uId*:UpdateID, *N*:NodeID)
1: Get X-lock
2: **if** (this node itself is *N*)
// Start the Update Phase
3: Apply the update to aggregate
4: Let *newAgg* ← Change in the aggregate value
5: Send naiveUpdate(*newAgg*) to parent
6: Release X-lock
7: **else**
8: Let N' ← Child node along the path to N
9: Send getNaiveXLock(uId, N ) to N'

**Algorithm 3** Update Phase

Require: *deliverCount*: UpdateID → Integer
**deliverNaiveUpdate**(*uId*:UpdateID , *delAgg*: $\delta$Agg)
1: Apply *delAgg* to the aggregate
2: Let *newAgg* ← Change in the aggregate value
3: **if** (this node is the root node for the update tree)
4: deliverCount[*uId*] ++
5: **if** (updateOperation[*uId*] == MOVE)
6: **if** ( deliverCount[*uId*] < 2) return
7: Release X-lock
8: **else**
9: Send naiveUpdate(*newAgg*) to parent
10: Release X-lock

ger to the *id* of the initiating leaf node. Each message of that update carries this unique *updateId*.

The serialization order of this update protocol for a read and an update transaction is the order of S-lock by the read query and the X-lock by the update query. For concurrent intersection update queries, the serial order is same as the lock points at the highest (unique) node of the common twig pattern.

### 5.2 Multi-Phase Update Protocol

In the naive update protocol, the root node is locked for the entire duration of the update. Now the root node is the first node to be read by any conflicting read. So, it being locked implies that any conflicting read query cannot read any of the common nodes for the entire duration of the update. This results in low concurrency and higher read time. In order to increase concurrency we need to allow concurrent read on common nodes while it is not being updated i.e., which locks are acquired and other nodes are being updated. However this seems to be rather difficult if the updates are to be atomic for the read queries. Read queries come from top-down and updates are propagated bottom-up. So the root node is the last node to be updated and the first node to be read. In the naive case, we prevented the read query from reading any values of the update tree nodes until the update has been reflected on all the relevant nodes. We now present the multi-phase update protocol which satisfy the atomicity constraints, and yet is highly concurrent.

| | S | U | X |
|---|---|---|---|
| S | True | True | False |
| U | | False | False |
| X | | | False |

Table 1: Compatibility Matrix

The key modifications introduced in this protocol are as follows. First, the nodes are updated top-down rather than bottom up as in the naive case. This is done by splitting the update in three phases namely *acquire lock phase*, *propagate phase* and *refresh phase*. Second, we hold the X-lock on the nodes for a very small duration during the update phase. Concurrent read queries (using S-locks) are allowed while acquiring locks and updating other nodes. This is done by introducing a new locking mode that we call U-lock, which is compatible with the S-lock. Third, to prevent read query from overtaking top-down update and reading inconsistent values, we use crabbing protocol for acquiring X-locks to update nodes.

The new lock mode that has been introduced, U-lock, is basically to lock nodes for possible future modifications. Its compatibility matrix is shown in Table 5.2. Before updating the nodes, they are locked in U-lock mode. U-lock can be upgraded to X-lock when the node's data is to be modified. The compatibility matrix shows that U and S mode are compatible, which signifies that the read query can proceed while the update is modifying other nodes of the update tree. However U and U modes are incompatible, which means that conflicting updates need to wait for each other.

**Algorithm 4** Acquire Lock Phase

**deliverGetULock**(*uId*:UpdateID, *N*:NodeID)
1: Get U-lock
2: **if** (this node itself is the leaf node *N*)
// Start the propagate phase
3: Let *updatedAgg* ← Change in aggregate value
4: Store as pendingUpdate
5: Send mpUpdate(*uId*, *updatedAgg*) to parent
6: **else**
7: Let N' ← Child node along the path to N
8: Send getULock(*uId*, *N*) to N'

The first phase of the multi-phase update protocol is the *acquire lock phase*. As shown in Algorithm 4, locks are acquired top-down starting from the root node, similar to the naive case. However here the nodes get locked in U-lock mode rather than in X-lock mode. Once the leaf nodes get locked, they start the propagate phase (Algorithm 5). In this phase, updates get propagated from leaf nodes upwards. However the

**Algorithm 5** Propagate Phase

```
    Class Update
      uID: update ID;
      δAgg: δSUM: Real, δCOUNT: Real, newMAX:
      Real, newMIN: Real
      isResetMin: Boolean;
      isResetMax: Boolean;
      LCAid: nodeID //nodeID of the lowest common
      ancestor for this update
      updateOperation: Integer;

Require: updateCount: updateID → Integer
    deliverUpdate(U: Update)
 1: if (Node is the root node of this update tree)
 2:   updateCount[U.getID()] ++;
 3:   if (U.UpdateOperation == MOVE)
 4:     if (updateCount[U.getID()] < 2)
          // propagate phase ends in one leg
 5:       Store U as pendingUpdate
 6:       Return
      // else propagate phase ends on both legs
      //start the refresh phase
 7:   Upgrade U-lock to X-lock
 8:   Apply U and pendingUpdate
 9:   Let L ← List of child nodes from which update
      messages with this updateId were received
10:   for all nodes N_i of L
11:     Get X-lock on N_i
12:     Send refresh(updateId) to N_i
13:   Release X-lock
14: else
15:   Let newAgg ← Change in aggregate
16:   Add newAgg to pendingUpdate
17:   Send update(newAgg) to parent
```

**Algorithm 6** Refresh Phase

```
    deliverRefresh(uId: updateID)
1: Let δAgg ← pendingUpdate with key uId;
2: Apply δAgg to the aggregate
3: if ( Node is a leaf node) Release X-lock
4: else
5:   Let N ← Child node from which this update
     message with this updateID was received
6:   Get X-lock on N;
7:   Send refresh(uId) to N;
8:   Release X-lock
```

updates are not reflected in the nodes as yet. They are just stored as *pendingUpdates* (lines 15-16). Class *Update* shows how the pendingUpdates are stored at each node. When this *update* message reaches the root node of the update tree, it can conclude that the propagate phase has ended in that leg of the tree.

After the propagate phase ends in both the legs of the update tree, the root node starts the refresh phase (lines 7-13). In this phase, the U-locks of the nodes get upgraded to X-locks and the stored *pendingUpdates* are reflected top-down. The crabbing protocol is used for acquiring and releasing the X-locks on the nodes (lines 10-13). When a refresh message reaches a node, it invokes the *deliverRefresh* function (Algorithm 6). Similar to the root node case, the crabbing protocol is used for acquiring and releasing X-locks (lines 6-8). When the *refresh* message reaches the leaf node (line 3), it implies that the update has ended in that leg of the update tree.

The serialization order of this update protocol for read and update transactions is the order of acquire of S-lock by the read query and the X-lock by the update query. For concurrent intersecting update queries, the serial order is the same as the lock points at the unique highest node of the common twig pattern.

### 5.2.1 Correctness and Efficiency

We now discuss the key features of the above protocol and show how each of these steps is important. We also present scenarios that are essential to understanding the correctness of the multi-phase update protocol. The first key point is that in the acquire lock phase, we acquire U-locks from top-down. For the naive case, we had argued that acquiring the X-locks bottom-up can lead to deadlock as the read queries come from top-down. However in this case, the U-locks are compatible with the S-locks. This might make us curious whether the U-locks can be acquired bottom-up thus merging the acquire lock and the propagate phases. However following example shows that this can lead to a deadlock between concurrent conflicting updates.

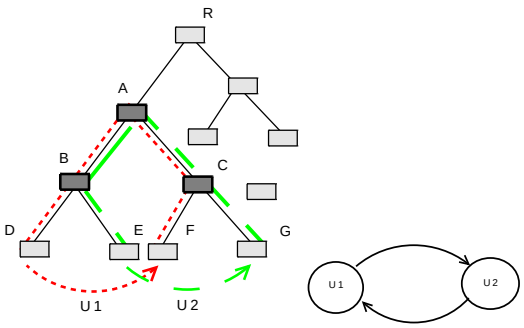


Figure 4: (a) Conflicting Updates (b) Wait-for Graph

Let $U_1$ and $U_2$ be update trees of the two updates (Figure 4). Consider the following order of acquiring locks. Let $U_1$ acquire U-lock on D, B and then A. Then $U_2$ acquires U-lock on G, C and then E. Now $U_2$ waits for U-lock on B and $U_1$ waits for U-lock on C resulting in a deadlock.

The other feature is the use of crabbing protocol. For updating the values of the nodes, the U-locks are upgraded to X-locks using crabbing protocol. This is to ensure that once the updates start getting reflected in all the nodes top-down, no read query overtakes that update. Each read query sees either the state

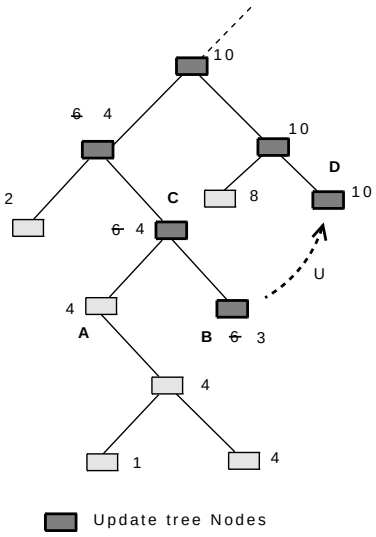


Figure 5: Scenario II

before the update or after the update on all common nodes. Thus the update to the distributed MRA tree is atomic with respect to read queries. We now present the three scenarios that are essential in understanding the correctness of the protocol.

**Scenario I** Consider the case wherein the max of a leaf node $N$ changes to say $m$ such that it needs to be propagated all the way up to the root node of the corresponding update tree. This value $m$ is propagated up during the propagate phase. Now what if the maximum value of that leaf node $N$ gets changed during the time between it being propagated up the tree where the updates are stored as *pendingUpdates* and the *pendingUpdates* being executed at the nodes. However this cannot happen as we acquire the U-locks before propagating any values and keep this lock for the entire duration between propagate and refresh phase. This ensures that no other update can change the node's value being propagated up the tree.

**Scenario II** Consider Figure 5 which shows the max aggregate values at the different nodes in the aggregate tree. Let the initial max values of nodes be as shown. Consider an update $U$ which transfers the *max* value object from node B to node D thus reducing the *max* at node B from 6 to 3. So in the propagate phase, the max value of A gets propagated up the node C. Now we may be curious that since there is no U-lock on A, what if its *max* value which has been propagated up the tree, changes between the propagate and the refresh phases. However we can argue that this cannot happen. This is because the *max* value of A is also cached at node C. For every update transaction, our definition of update tree ensures that if a node is part of the update, then all nodes

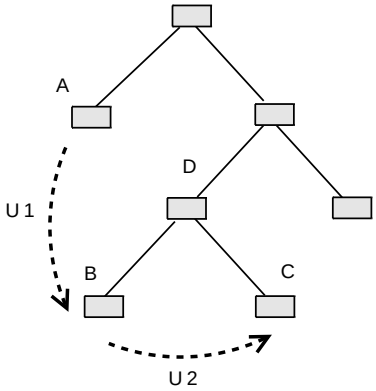


Figure 6: Multiple Updates to an object

containing the cached aggregate values are also part of the update tree. Hence any transaction attempting to modify the *max* value of A would intersect with $U$ on at least C. Thus they would need to be executed serially.

At this point we may also note that caching the min and max values on parent node in the MRA tree helps reduce the update latency by greatly reducing the number of nodes required to be locked. If we had rather assumed the model in which these values are not cached at the parent node, then for updating the min or max at any intermediate node of the tree we would have to lock not only the child nodes whose values we propagate up but also the other child nodes whose value we read. This would imply that we have to acquire lock on all the child nodes of the intermediate update tree nodes, even if they are not themselves part of the update tree. This would hugely increase the number of nodes required to be locked for any updates to the distributed MRA tree causing much increase in the latency since the different nodes may be located at different peers.

**Scenario III** We now consider the case of multiple updates to an entity. With reference to the Figure 6 consider an entity to be transferred by an update $U_1$ from $A$ to $B$ and then by $U_2$ from $B$ to $C$. Logically, $U_1$ should get reflected on the common nodes $B$ and $D$ before $U_2$. In our model, we had assumed that the communication channel is FIFO, updates are executed at the nodes in the order they are received. Thus serial order of execution of updates at node $B$ makes sure that $U_1$ completes before $U_2$ begins.

## 6 Divisible Aggregates

The updates we had considered till now modify any or all of the aggregates from the set $\langle$*sum, count, min, max*$\rangle$. However, while most update transactions will

modify aggregates like sum and count, very few of them will result in change in *min* and *max* aggregates. We now consider the special case of the update transactions in which the changes are only to the divisible aggregates like sum and count. As defined in [8], an aggregate *agg* is *divisible* if there exists a function $f$ such that

$$agg(A\ B) = f(agg(A), agg(B))$$

whenever B $\subseteq$ A. The aggregate *sum* is an example of this aggregate since $sum(A\ B) = sum(A) - sum(B)$, whenever B$\subset$ of A. However, *min* and *max* are not divisible aggregates.

For the cases wherein only the divisible aggregates like *sum* and *count* are modified, we observe that the change to be made to all nodes of the update is known and does not depend on the current aggregate values at the nodes as in the case of non-divisible aggregates like *min* and *max*. Thus there is no need to propagate these changes bottom-up, as now the nodes can be updated top-down. Hence the propagate phase is not needed. This also eliminates a separate phase for acquiring lock top-down. Thus such updates can have only one phase called *update phase*. In this phase, X-locks are acquired top-down using crabbing protocol so that no read query overtakes the update, the nodes are updated, and the locks released (Algorithm 7). Thus the updates are still atomic with respect to the read queries.

**Algorithm 7** Update Phase

**deliverMPUpdate**($U$: Update)
1: Apply $U$ to the aggregate
2: **if** ( Node is a leaf node) Release X-lock
3: **else**
4: Let N $\leftarrow$ Child node along this update tree
6: Get X-lock on N;
7: Send *update*($U$) to N;
8: Release X-lock

### 6.1 Comparative Analysis of Update Methods

To get an estimate of the difference in concurrency provided by the naive and the multi-phase update protocol, we show the relative time for which the root node of the update tree remains locked in each protocol. This is an important indicator of their relative concurrency because the read query accesses the aggregate tree top down. Thus the read query can read any node of the update tree only after it is able to read its root node.

Consider U to be the update tree and $U_R$ to be its root node. Note that the time taken for each update phase depends on the length of the longest leg of the update tree. Let the number of nodes in the longer leg of the update tree be $m$+1 nodes and $m$ be the number of edges. Consider that the processing delay at each node on the 0 and the transmission delay between all nodes to be a constant time $d$. Figure 7 shows the time for which the node $U_R$ is locked for read queries. For the naive algorithm, the root node acquires and X-lock and retains it for the entire duration of the update. For each phase, the message has to travel the tree leg once and hence each phases takes duration $dm$. For the multi-phase protocol, the node $U_R$ is locked in X-lock mode only for the first $2d$ duration of the refresh phase ($d$ for sending the *getXlock* messages to the child nodes and another $d$ for getting the *ack*). The third bar in the figure refers to the special case when the updates is only to the divisible aggregates.

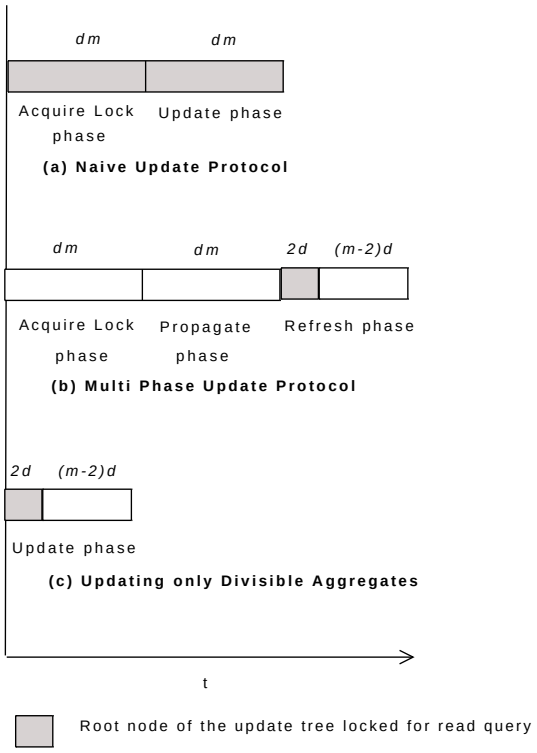


Figure 7: Timeline of the state of update tree's root node under the update protocols

To get an idea of the amount of time the MRA tree index structure is locked for the read operations, recall that among the updates arising from the *move* operation, we only considered the one in which the new location of the object is in a different leaf node (partition). We ignored the more frequent case where the new location of the object is in the same leaf, since the aggregate index structure need not be updated in this case. Also among the former category of updates, most updates will modify only the divisible aggregates. For the remaining few updates that change the min and the max aggregates, we have a choice between the naive and multi-phase update protocols.

## 7 Experimental Evaluation

We implemented the system in Java. The peer-to-peer overlay setup uses the Pastry DHT for hashing. The quadtree node to peer mapping is determined by hashing the unique centroid of each node onto the peer overlay. The Pastry implementation used is the freely-available version called FreePastry provided by Rice University.

The DHT is run on all the participating peer nodes and our application runs on top of it. We ran the simulation on 25 peers by running 5 instances of the application and the underlying DHT on each peer.

Testing of the protocols was done using synthetic data. We generated a dataset with various number of objects as follows. The total number of data points varied from 100 to 10000. These were organized into 10-100 clusters of points depending on the total number of points needed. The number of data points in each cluster is taken from a normal distribution whose mean and variance varied depending on the total number of data points required for that experiment.

Each cluster spans 10% of the $[0,1]^2$ space on each dimension. Its centroid is uniformly distributed around the space. Each object's data value is randomly taken from distribution ($\mu = 100$, $\sigma = 50$) with care to avoid negative values.

Each peer specifies its threshold which defines the number of point objects it can support. In case the number of objects exceed the threshold of a system, the quad-tree node having the maximum entities is chosen and split into into four sub-regions. To avoid splitting of regions to very small sub-regions, we fix the maximum and minimum resolution of the quadtree by using bounds on tree depth of $f_{max} = 14$ and $f_{min} = 1$ respectively.

For the experiments in the following sections, we need to get the partitioning quad-tree of varying depth so that we can get update trees of various depths. In order to vary the tree depth, we vary the threshold of the peers; all peers have the same threshold in our experiments. To understand this, assume that there are $N$ data points and $P$ is the number of peers. Thus the average number of data points supported by each peer is $\mu = N/P$. Assuming all peers have the same threshold, say $T$, the minimum value of $T$ required for running the application is $\mu$. If $T$ is very large, say $T = N$, or equivalently $T/\mu = P$, then all the entities can be supported on one single peer and it then becomes a centralized system, with no partition and hence the depth of the partitioning tree is 0.

As we decrease this value of $T$ and make it closer to $\mu$, the minimum and the maximum depth of the tree keeps steadily increasing. When $T$ becomes very close to $\mu$, the tree needs to split such that the threshold is satisfied for all the peers. Thus to get update tree with greater number of nodes, we move this value of threshold closer to $\mu$. The graph in Figure 8 shows the minimum and the maximum depth of the quad-tree at different values of the threshold.

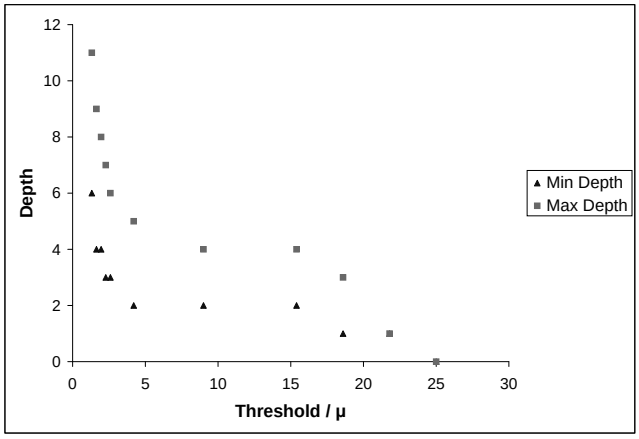


Figure 8: Variation of the minimum and maximum depth of the partitioning quad-tree with threshold

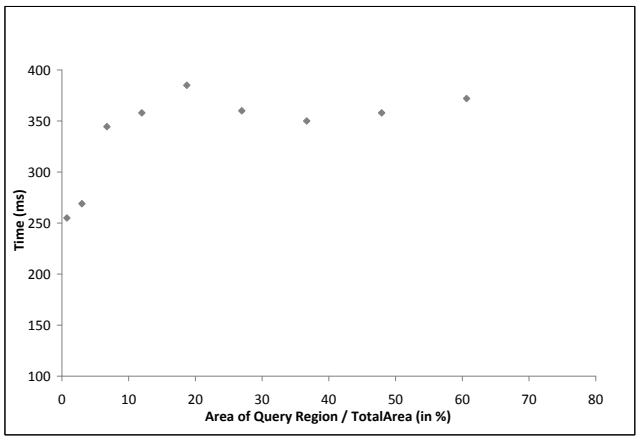


Figure 9: Variation of the read query duration as the query region increases, with no updates

### 7.1 Read Query with no Updates

Figure 9 shows the variation of the read query time as the area of the query region increases, with no simultaneous updates. The read protocol is the same for both the naive and the multi-phase update protocol, and in the absence of updates the read performance is identical (this was also experimentally verified).

We can see that as the area of the query region is increased, there is no corresponding increase in the time take for the read query. This is because the increase in area of the query does not relate to any corresponding increase in the number of nodes explored in the aggregate tree. If a node is completely contained within the query region, then the query is not propagated to the subtree.

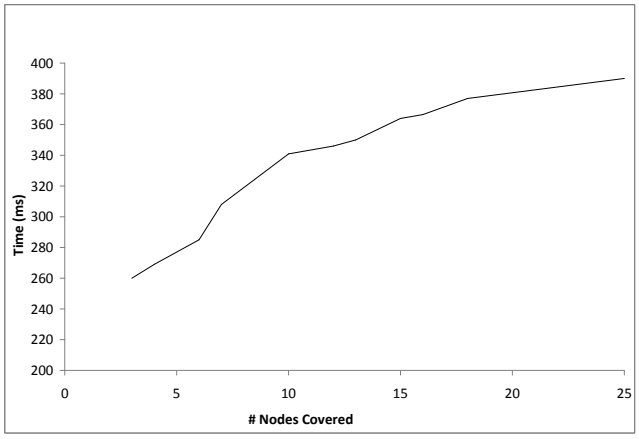


Figure 10: Variation of read query duration with the number of nodes read, with no updates

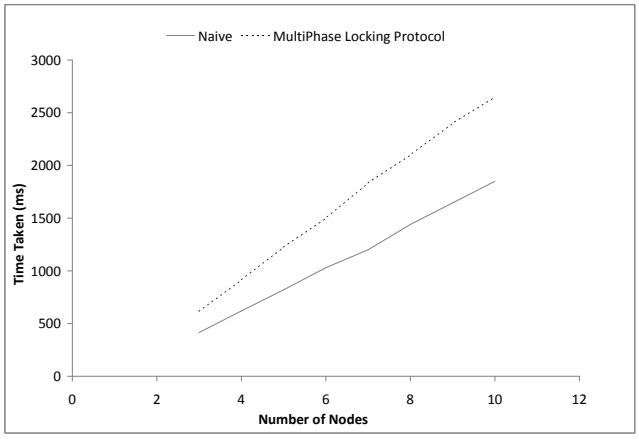


Figure 11: Variation of average time taken for an update with the number of nodes in the update tree

Figure 10 shows the variation of the read query duration with the increase in the number of nodes in the read query tree, again with no updates. As can be seen from the graph, the duration of the read query directly depends on the number of nodes explored by the query. One point to note is that the increase in the read query duration as the number of nodes increase does not happen in steps of the communication delay between the nodes, rather it increases by a lesser amount. This is because the tree nodes at the same depth are traversed essentially in parallel. However the increasing trend in the graph is due to the queue at the querying node and the processing time to recompute the new aggregate after receiving the read reply, which amounts to the increase in the read query duration as the number of nodes read increases.

### 7.2 Comparison of the Update Protocols

To evaluate the relative duration of the update protocols, we studied the time taken for the update as the number of nodes in the update tree increases. As expected, Figure 11 shows that the time taken for update using both the protocols is directly proportional to the number of nodes in the update tree. This is due to the proportional increase in the communication delay. Another observation is that the update duration is greater in the multi-phase update protocol than then naive one. This is because the number of messages exchanged in the multi-phase update protocol is greater than in the naive case on account of more number of phases.

### 7.3 Read Query with Simultaneous Updates

We evaluated the read query duration under different workloads, under the naive update protocol and the multi-phase update protocol. The results are shown in Figure 12 (a) and (b). We show the read time under three different workloads – no update, and with $F_1$ and $F_2$ frequencies of update, where $F_1$ and $F_2$ correspond to one update in every 3 seconds and 1.5 seconds respectively. As expected the time taken by the read query increases as the frequency of updates increase. This is because the read query has to wait for a longer time at the nodes for getting the read lock. The time taken by a read query also increases with the number of nodes read increases; this increase is much larger in the case of the naive update protocol, as compared to the multi-phase update protocol. This is because the nodes remain exclusive locked for a much longer time in the naive protocol, as compared to the multi-phase update protocol.

## 8 Conclusion

In this paper we have addressed the problem of finding aggregate over mobile point data. We extend the multi-resolution aggregate index to support mobile data and also run on a distributed system. The protocols proposed here can be used for supporting an aggregate index over mobile point data in a centralized system as well. The key function is to update the aggregate index when the location of the data objects change, while ensuring that the updates are atomic with respect to the read queries. We proposed a multi-phase update protocol which shows high concurrency for the read queries and compared it with the naive update protocol. We showed how the read query and multiple simultaneous updates to the index structure are serializable with respect to each other. We then established through analysis and detailed experimentation that the multi phase update protocol enables the aggregate index structure to remain locked for read queries for a fraction of the time as compared to in the naive case.

As part of future work, we plan to extend our ap-

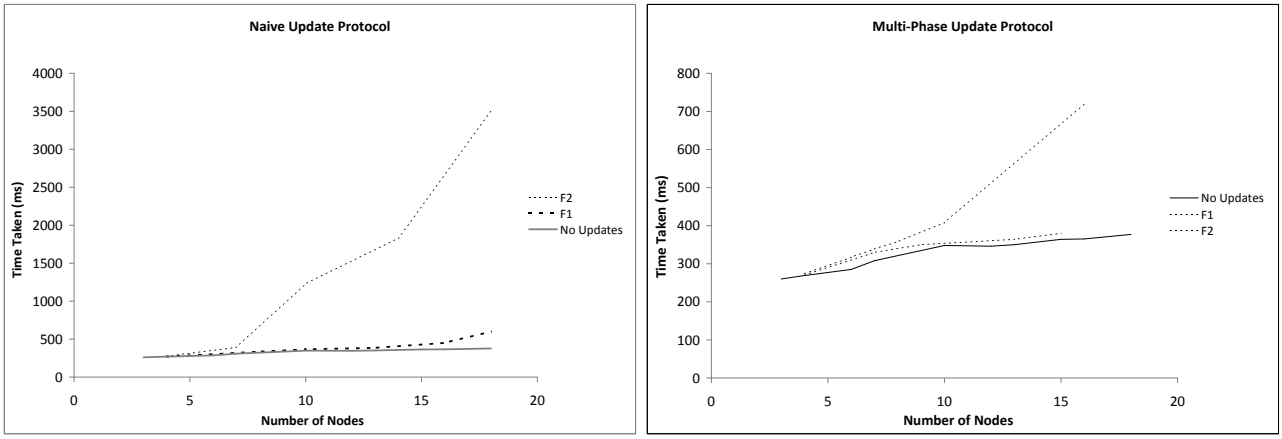


Figure 12: Time taken by the read queries with concurrent updates for different workloads with the (a) Naive Update Protocol (b) Multi-phase Update Protocol. F stands for the frequency of updates. Here F2 > F1

proach to handle more complex update transactions, involving multiple updates affecting more than two index tree leaf nodes.

## References


[1] M. Erwig, R. H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *Geoinformatica*, 3(3):269–296, 1999.

[2] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. A data model and data structures for moving objects databases. In *SIGMOD*, pages 319–330, 2000.

[3] J. Gao, L. J. Guibas, J. Hershberger, and L. Zhang. Fractionally cascaded information in a sensor network. In *Symp. on Information Processing in Sensor Networks (IPSN)*, 2004.

[4] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. *SIGMOD*, pages 401–412, 2001.

[5] E. Tanin, A. Harwood, and H. Samet. Using a distributed quadtree index in peer-to-peer networks. *The VLDB Journal*, 16(2):165–178, 2007.

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

[7] Y. Tao and D. Papadias. Range aggregate processing in spatial databases. *IEEE Transactions on Knowledge and Data Engineering*, 16(12):1555–1570, 2004.

[8] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling games to epic proportions. In *SIGMOD*, pages 31–42, New York, NY, USA, 2007. ACM.