High Concurrent R-tree Operations when Tracking Continuous Movement in Main Memory

Cezar Chitac[1], Robertas Kerpys[1], Raluca Marcuta[1], Simonas Šaltenis[2]

Department of Computer Science, Aalborg University
Denmark
[1]{cchita09, rkerpy10, rmarcu09}@student.aau.dk
[2]simas@cs.aau.dk

Abstract

There is a growing need to accurately track moving objects in a given area. In order to support efficient queries on positions of tracked objects, a customized spatial data structure is used. The application domain requires efficient updates to be performed in order to maintain the reported position of objects as fresh as possible. With the evolution of main memory databases this goal becomes easier to achieve. However, in order to take full advantage of highly parallel modern CPUs, one must take into consideration the issue of concurrency in such a system. Being a fact that locking and, or latching greatly affect the behavior of the system with regards to concurrency between operations, the goal of the present paper is to introduce a concurrency algorithm that avoids locking and latching as much as possible. Based on this, two new approaches are presented that ensure high concurrency when structural modifications due to underfull and overfull nodes are taken into account.

1 Introduction

Many applications and users today need to track objects subject to continuous movement. In order to achieve this, objects need to send information about their whereabouts on a regular basis. As more objects are tracked, the system managing the update processes faces important concurrency issues, since two or more objects may want to access the same resource given by where their information is stored. The direct logical solution is to use locking and, or latching in order to ensure that updates maintain the consistency of the structure that stores the positions of the tracked objects [8, 9]. However, such an approach is naive at best, as in many real life scenarios the system must face a high rate of updates. Even in the case where locking is not applied on the whole structure, a high update rate introduces serious delays that translate in loss of accuracy [1]. In addition, from a hardware point of view ensuring concurrency is vital. Current trends show that the speed of single-core CPUs is only marginally improving, while the focus is concentrated on adding more cores. This leads to multiple threads that can execute at the same time if concurrency is considered, increasing parallelism and by extension the speed of the overall system.

For example let us consider the classic R-tree structure [2], that this paper uses as a starting point. Locking and latching would be performed either on the entire tree or on subtrees. While the first case allows updates to perform one by one, thus representing a serious bottleneck performance wise, the second case allows to some degree two or more updates to run at the same time. However this works only for updates that do not perform on the same subtree structure. To complicate matters even more, the system must be able to answer queries as well. Conflicts that may arise from queries and updates accessing the same information, are generally solved in the same manner, further decreasing the performance in real time.

It is in this context that we introduce a concurrency algorithm that aims at minimizing locking and latching as much as possible, since completely avoiding them is impossible. Two approaches are proposed in order to deal with the problems that underfull and overfull nodes introduce, without affecting performance. Both approaches are theoretically described in detail, starting from the common semantics and ranging to in depth pseudo-code explanations of each operation. In order to increase concurrency we formulate a series of assumptions drawn from the application domain. We consider a minimum time between updates, as well as a maximum distance an object can move between updates. This leads to the update process being performed efficiently, in the sense that an object that is not moving will not update as long as a chosen distance is covered by its movement trajectory. The

first approach is based on an enhanced version of the R-tree structure and improves the way splits and merges are treated in a concurrent environment. A thorough analysis sustains the correctness of the approach, but in the same time highlights the increased algorithmic complexity that results from the low degree of locking/latching. A totally new view is offered in the second approach, distinguishing it, to the best of our knowledge, from all other related work. Elaborate heuristics are introduced to decrease complexity, by eliminating splits and merges, while preserving the main focus of this research, that of concurrency.

The remaining part of the paper is structured as follows. Section 2 introduces the settings that the paper addresses and presents the semantics of the target domain. Next an overview of the related work is given in Section 3. Section 4 presents in detail the index structure used. Concurrency of updates and queries is discussed in depth in Section 5. Sections 6 and 7 contain the two proposed approaches. Finally we conclude and consider future work in Section 8.

2 Preliminaries

We consider a set of moving objects in a two-dimensional space. The rough goal is to make knowledge about their positions accessible to different users and keep the information up-to-date. The objects are uniquely identified through a field called *oid* and their positions are represented in the form of a pair of coordinates *(x, y)*. The information gathered from the moving objects is organized in an index structure, which is build in order to facilitate concurrent read queries and updates of objects' positions.

One of the assumptions is that the objects send updates of their positions to a central server, on a regular basis. The maximum speed in the system is known, v_{max}, and an update timestamp, t_u, signifying the last update time for an object is retained in the index structure. The frequency of the updates differs from one object to another and is dependent on the object's speed and a threshold, named δ. A shared-prediction update policy is used, that guarantees that at any time the reported object's position is not further away from the current real position of the object by more than the threshold δ. In other words, whenever an object moves away from the last reported position by δ space units, it needs to send information about its current position to the server.

As it results from the argument above, we can state that the actual position of an object at the time of a query is in an area given by a circle of maximum radius δ. A range query returns all objects which are, at a chosen time, in the range area given by the two opposite corner points of a rectangle, lower left and upper right: (x_{low}, y_{low}) and (x_{high}, y_{high}).

In order to ensure that no objects are overlooked because their last reported positions are not in the range but the real ones are, the query range is enlarged by δ in all directions. The query rectangle is approximately given by the corner points: $(x_{low} - \delta, y_{low} - \delta)$ and $(x_{high} + \delta, y_{high} + \delta)$. This way, the query range encloses objects that *may be* in the original range. The following assumption ensures that an enlargement by δ is sufficient:

Query execution time is shorter than the shortest interval between two consecutive updates of an object ($t_e - t_s < \delta/v_{max}$).

Updates of positions are allowed to take place during a processing of a query which starts at t_s and ends at t_e. A query is based on the updates as of time t_s plus some of the fresh updates after t_s that are encountered by the query during the tree traversal. The retrieved positions are then processed in order to construct an unified view of the system's state at a time of reference. We chose the time of reference to be the moment the query starts, t_s. The goal is to return all objects that *may have been* in the original query range at time t_s.

A query encounters an object during its tree traversal at a time $t \geq t_s$, and is able to construct based on the last reported position of the object, the possible area where the object may have been at t_s. The intention is to roll back or forward time to t_s, depending on whether the last reported position is as of a time following or preceding t_s.

This is done using the information about the maximum speed of an object (which is the maximum speed of the system) and the time of the last update, t_u. The possible area is a circle of radius $min(v_{max}|t_s - t_u|, \delta)$, centered in the last reported position seen by the query. In order to follow the *may have been* desired semantics, only objects for which the intersection between their possible position area and the query's initial range without enlargement is not null are retrieved by the query. The semantics can be easily changed to *must have been* by requiring the possible area to be totally included into the original query range.

Two cases are distinguished depending on whether the query reads a position which was updated before a query started or after. In Figure 1 the object's last update happens before the query starts and reads the object. The last reported position seen at time t is not in the original range but the object moved since its last update and at time t_s it is covered by the range. The range enlargement allows us to return and consider this object for the *may have been* set. Since we are interested in the position at time t_s, the possible area where the object might have been is constructed, as explained above and it is marked by the gray circle in the figure. From the not null intersection between the circle and the initial range, we conclude that the object belongs to the resulting set.

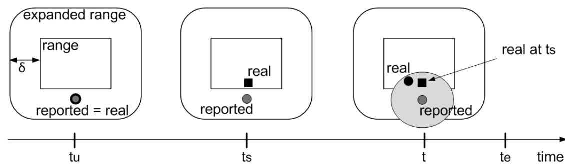


Figure 1: *The positions of an object relative to the query range during the running time of the query. Result is constructed by rolling time forward from last position report (t_u) to t_s*

The case in which the update of an object happens after the query starts and it is seen by the query during its traversal, is illustrated in Figure 2. The object was in the range at time t_s but moved and reported its new position which is not covered any more by the range. The new position is enclosed in the enlargement since the object could not have moved more than δ. As in the first case, the possible area where the object might have been at t_s is constructed. The not null intersection with the unexpanded range enables us to conclude that the object *may have been* in the range at t_s.

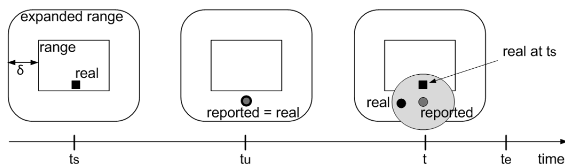


Figure 2: *The positions of an object relative to the query range during the running time of the query. Result is constructed by rolling time back from last position report (t_u) to t_s*

3 Related Work

A significant amount of research on concurrency in R-trees accumulated during the past two decades. In this section we review the most relevant work to which our proposed methods relate.

Ng and Kameda [8] propose three traditional locking algorithms on the R-tree index structure. The simplest algorithm they consider is locking the entire tree which during updates reduces the concurrency because exclusive lock is acquired on the root of the tree. The second algorithm Ng and Kameda introduce locks the whole tree only during splits or merges, thus allowing to execute multiple insert and delete operations as long as they do not rise any underfulls or overfulls of the node. The third algorithm uses lock-coupling protocol which locks individual nodes instead of locking the entire tree. The lock is acquired on a node by a search or an update operation and can only be released when a lock on a child node is granted.

One of the first works which tries to reduce locking is the paper of Kornacker and Banks [4]. Their approach uses right-link pointers which connect sibling nodes in order to compensate for structure modifications. A R-link tree allows multiple update operations to execute concurrently and unfinished splits are caught by following the right-link pointers. This proposed solution locks only one node at a time in the case of search operations. However, this method employs lock-coupling when splits are performed.

A paper by Kornacker et al. [5] improves the R-link tree approach by minimizing the required additional information stored in the node and in each entry. They present concurrency algorithms for a generalized tree structure (GiST), which employs right-link pointers as the R-link tree does. The algorithms can be applied on particular tree structures as well.

Rastogi et al. [9] presents a different approach on concurrent R-trees by employing logical and physical versioning. When update transactions update a data item, a new version of that item is created. Thus multiple versions of data items are retained in logical versioning and part of the tree structure is copied and items are updated on that structure in physical versioning. By using the aforementioned methods, updates can traverse the tree with no latching and searchers can perform lookups also without obtaining latches. Since the method implies copying parts of the tree structure involved in the modifications, these parts are not minimal and cause significant overhead of the algorithms. In contrast, our first method only copies the necessary parts for modification i.e. a node.

Song et al. [10] summarizes research on concurrency in R-trees and presents yet another approach on how to boost performance in the multidimensional index structures. Authors propose the partial lock coupling technique, which employs lock coupling during updates of MBRs (minimum bounding rectangles) only when the MBR of a node is shrunk. Their algorithm tries to reduce locking during splits and holds exclusive locks only during the physical node split time interval.

All the previously mentioned methods try to reduce locking as much as possible, because such a technique represents a bottleneck for concurrency in R-trees. Eliminating locking all together is not a realistic goal but it is possible to further minimize it by making use of semantics of the application domain. We investigate two algorithms which employ minimal locking and take advantage of assumptions related to a system of moving objects. Partial locking is common for both proposed approaches in the case of shrinking a node's MBR, as done in [10]. The first algorithm permits execution of search and update operations concurrently and without holding

additional locks during tree traversal. However, in the case of split or merge of a node, latches are necessary to ensure consistency of the data. Based on the outcome of the first algorithm, another R-tree structure variant is proposed, which does not employ the regular approaches of splits or merges. The second method enhances concurrency further more, by eliminating the need of latching. Both our algorithms are build upon the u-R-tree index structure which was proposed by Šidlauskas et al. [11]. The u-R-tree is a modification of the R-tree which employs bottom-up update strategy proposed by Lee et al. [7].

4 Index Structure

4.1 R-tree

During the past two decades the R-tree [2] index structure was a focus of research in spatial databases. The R-tree index structure is a height-balanced data partitioning tree similar to a B-tree. It has two type of nodes: internal and leaf nodes. Internal nodes consist in the collection of entries. Each entry stores the pointer to the child node and the minimum bounding rectangle that spatially bounds the child's entries, called *MBR*, see Figure 3. Moving objects are stored by *oid* and their coordinates, thus the index key is formed using the coordinates. Every node has in addition metadata (number of entries, leaf flag). The R-tree node structure is illustrated in Figure 3 (ignore the gray elements for now).

Searching in the R-tree starts at the root level and performs in a top-down manner. Due to overlapping *MBRs*, several paths are possible. Update operations on the regular R-tree are separate delete and insert operations. During the delete operation, the search function is issued to locate an object. The object is deleted and its corresponding *MBR* in the ancestor node is adjusted to ensure its minimality. Then, changes are propagated up the tree if necessary. If the node becomes underfull an expensive reinsertion has to be performed. The insert operation searches for the most suitable subtree where a new object should be inserted. When a leaf node is found, the new object is inserted in the node. The changes involving the enlargement of the *MBR* are propagated as well. A node can become overfull, in which case the node is split in two new nodes.

4.2 Update Efficient R-tree

Since the ordinary R-tree index structure is not suitable for update intensive systems, our study uses the update efficient R-tree (u-R-tree) [11]. Additional modifications are made to the u-R-tree in order to facilitate concurrency between updates and updates on one hand, and on the other hand, between updates and queries. As mentioned in Subsection 4.1 the R-tree index is formed of spatial information i.e. coordinates of the objects. In order to get information of the object, the top-down tree traversal, starting from the root level, is performed. Since all the information is stored at the leaf level of the tree, this kind of traversal is not efficient for update operations. To increase update performance, the bottom-up approach of the R-tree employs a secondary index on *oid*, see Table 1. It stores a pointer to the node where object identified by *oid*, P_{new} reside and an offset idx_{new}. Using this secondary index, an object on the leaf level is found without an expensive top-down search.

This paper proposes extending the secondary index table by adding two additional columns for pointer to the old node, P_{old}, and offset in the old node, idx_{old}, of the object in the tree structure. The underlined fields in the secondary index table reflects our changes. P_{new} is a pointer to the node where most recent version of the object resides, while P_{old} is a pointer to the node where slightly outdated version of the object is. Offsets idx_{new} and idx_{old} determine at which place in the node objects reside. The newly introduced P_{old} pointer and offset idx_{old} in the secondary index structure is due to duplicate instances of the object in the tree. As it is presented in Subsection 6.3, these duplicates are due to the versioning of an object.

oid	P_{new}	idx_{new}	P_{old}	idx_{old}

Table 1: *Secondary index*

The regular u-R-tree structure differs from the R-tree structure by the elements depicted in gray color in Figure 3. The underlined element, update stamp (t_u), is added on top of the u-R-tree structure, that indicates when the item was modified. The backward pointer (*parent ptr*) together with *parent idx* allows us to access the parent of the corresponding entry when the *MBR* changes need to be propagated. A copy of the node's *MBR* is added, which lets us know if the *MBR* was invalidated without accessing the parent node.
The update tuple for the u-R-tree consists in the object id and new coordinates for that object (oid, x_{new}, y_{new}).

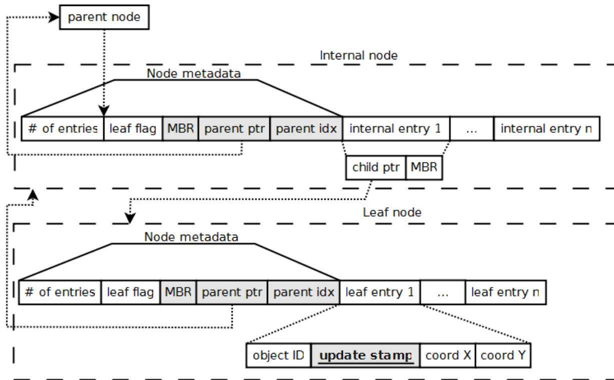


Figure 3: *Structure of a u-R-tree node*

5 Update and Query Algorithms

In the following we consider the interleaving between queries and update operation without the complications resulting from dealing with underfull and overfull nodes. These structural modifications will be discussed later on, as they make the object of the two approaches.

Common for both approaches is the distinction between two types of updates: local and non-local, and the way concurrent queries interpret them. Local updates only change value information regarding the position of the object, without affecting the tree structure itself. A non-local update on the other hand performs changes to the index structure also, due to moving the object from one node to another node. The decision to move an object to another node in the tree is based on the desire to minimize the penalty of *MBR* enlargement in order to decrease overlapping between rectangles.

Local updates do not cause queries concurrent with them to miss objects, a query can either see the consistent old position or the consistent new position of an object, depending on when the update occurs. This implies that the local update is performed atomically either by latching the corresponding entry for a small period of time or as a hardware-facilitated atomic operation.

The problem is more complex in the case of non-local updates. The main issue consists on how this move from one node to the other is performed. Information regarding an object must exist in the structure at all times, so the insertion of the new position has to be performed before the deletion of the old one. At the same time, we have to make sure that the object is not missed by a query that already visited the node where the new position is inserted. This is done at the cost of sometimes retrieving the old position.

A non-local update proceeds first by inserting the new position of the object that updated in a selected node. Then its former position, from the old node, is marked to indicate that the object has performed a non-local update. This version becomes logically deleted and it must be dealt with after ensuring that no query needs this information. Returning to the previously mentioned assumption, at most one update can occur between the start (t_s) and end time (t_e) of a query. Therefore these logically deleted positions can be dealt with at the beginning of the next update of each object. The process that handles this is called garbage collection, and its purpose is to make available the space in memory that is no longer needed. Once an object's position is garbage collected, the position becomes physically deleted, meaning that the information contained is no longer available.

Figure 4 presents the two types of updates, following the movement of object p_1 in its bounding rectangle R_1 for the local case, and the relocation to another bounding rectangle, R_2, for the non-local update. In this latter case, the reader can observe that in the interval after the first non-local update of p_1 and its second update, the positional information of the object exists in both rectangles.

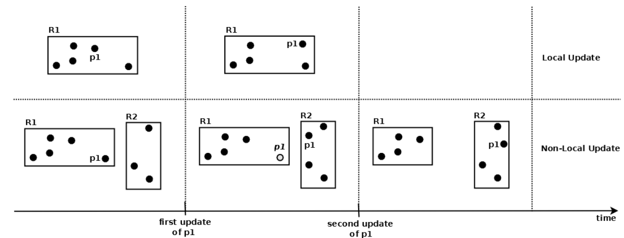


Figure 4: *Local and non-local updates of* p_1

In the following we show that it is safe to delete an old position on the next update and that a query cannot miss an object due to its movement to another node. A query scans the leaf node entries in the index from left to right. Let us consider the worse case, in which the new position is inserted into a node to the left, already scanned by the query. That implies that the non-local update happens during the running interval of the query, see Figure 5. A query will retrieve the old position. As stated before, query running time is shorter than the interval between two consecutive updates of an object. This assumption ensures that the old position cannot be physically removed before the query ends, since another update cannot come before t_e. If the insertion takes place into a node to the right not yet scanned by the query, then the freshest position is retrieved.

If a non-local update happens before t_s, then it is possible that another update of the same object comes before

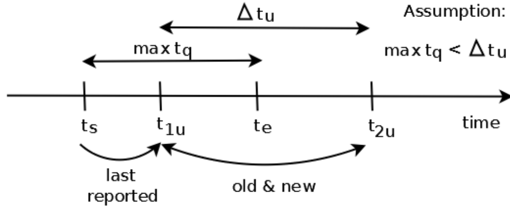


Figure 5: Available positions of an object that performs a non-local update during a query

t_e and physically removes the marked old position. In this case the new position is available for reading during the entire running time of the query, as it can be seen in Figure 6.

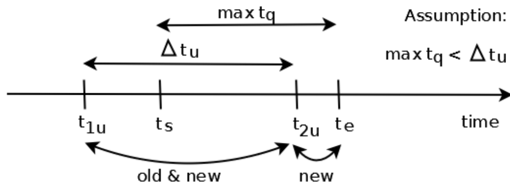


Figure 6: Available positions of an object that performs a non-local update before a query starts

A more accurate assumption takes into consideration the update execution time, in order to show the impossibility of the following case. A query starts just before a non-local update finishes its execution, leaving two copies of the object in the system. The query passes the node where the new version is inserted before the update finishes. If the update execution time lasts long enough, another update for the same object deletes the logically marked old version. To ensure the impossibility for the query to miss both versions of an object, an extension is introduced to the previous assumption:

Query execution time plus update execution time is shorter than the interval between two consecutive updates of an object.

Since an object may exist twice in the structure, for a short interval of time between two updates, the algorithms must ensure that the freshest position of the object is considered, if it is possible. A query can identify if the position it reads is new or old based on the time of update (t_u). An object has a negative t_u if its position is old, and positive otherwise. An update sets t_u, and whenever it is non-local it generates two instances of the same object, but with $t_u s$ having different signs.

At this point the distinction between the two types of updates, local or non-local, and the mechanism of ensuring that queries do not miss an object because of its migration to another node are established. The next subsection describes the common steps that are followed during an update of on object.

5.1 Overview of Update Process

When an object sends information on its new position, a bottom-up update process starts, presented in Figure 7. A look-up is performed in the hash-table using the oid of the object, which returns the two pointers P_{new} and P_{old}. If the object exists in two leaf nodes of the index structure, $P_{old} \neq Nil$, it means that the last update of the object was of type non-local and that the old location should be physically deleted. At this point, changes may have to be performed to the MBR of the node, which may propagate upwards through the parent pointers. Structural modifications need to take place in case the node becomes underfull. In the first approach, the *MergeNode* function is called on the leaf-node in order to merge it with a chosen sibling, process explained in more detail in subsection 6.5. The second approach deals with an underfull node differently, by avoiding movement of the objects in the time between their updates. Forced non-local updates are used to repopulate the node or to eliminate it, as it is presented in subsection 7.4.

The next step is to identify the type of update, local or non-local, using heuristics. In order to process the update, three parameters are passed to the update functions: the object identifier, oid, and the new position coordinates, x_{new} and y_{new}. A local update in the first approach simply modifies the position of the object and the *MBRs* if necessary. This is done also in the second approach, except in the case of leaf overfull with simulation of split in process. Then the update must undergo special treatment, because the node is evacuating objects in order to repair its state. This is done by treating the local update as a non-local one, placing the object in another node, previously created when the evacuating process was initiated. The object's position is updated in the new node and logically deleted in the old one. The *MBR* changes are performed and propagated for the new location.

The non-local update process is more complex, because it has to ensure that no readers miss an object due to its change of location in the index structure. This is accomplished by keeping the old position in the structure, along with the new one until a new update of the object. However, concurrency control between reads and updates when no overfull nodes are encountered, does not suffice. A non-local update may cause a node split, which implies the need to manage concurrent updates and reads that work on the node in question. This is particular to each approach and will be presented in the dedicated following sections.

A candidate node in which to insert the new position

is found through a top-down traversal of the index, using different heuristics corresponding to each approach. If the insertion causes the split of the node, then the function *SplitNode* deals with this case for approach 1, by creating a separate working copy for a part of the index structure, details explained in subsection 6.4. The second approach simulates the split by starting an evacuation process for the node. The "split" process proceeds by causing other objects to be placed in the new node on their upcoming updates, until the node reaches a non-overfull state, see subsection 7.3.

If the candidate node has enough space, then the update can proceed with the insertion of a new entry, corresponding to the new position, x_{new} and y_{new}. Pointers in the hash-table, $MBRs$ and the update timestamps need to be actualized when the update is finalized in order to mark the old location of the object logically deleted and make the new one visible to other operations. This step is also included in the split phases of each approach.

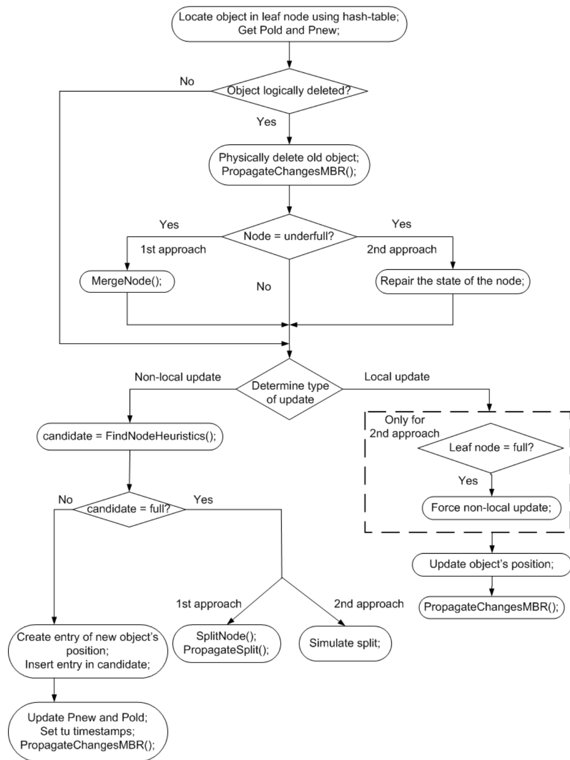


Figure 7: *Overview of the update process*

5.2 Queries

The search algorithm, proposed by Guttman [2], was modified in order to ensure that no objects get overlooked by a reader, due to the structural modifications that may take place in the interval a query scans the index tree. The search remains a top-down traversal, starting from the root of the tree and visiting the nodes for which the *MBR* overlaps with the query's range rectangle. Algorithm 1 presents the main steps that can be distinguished during the traversal. As a parameter set it accepts a node where to start the search, N, and an expanded by δ query range, Rec.

When at leaf level, the algorithm scans the entries of a node one by one, line 6 to 14. As argued in Section 2, a filtering of the results has to be done because the enlarged query range may return additional objects that could not have been in the query range at the time t_s. Results are filtered by constructing a circle of radius $min(v_{max}|t_s - t_u|, \delta)$, which is centered at the last reported position of the object. This possible area were the object may have been at t_s is intersected with the query's unexpanded range and the object is considered only if the intersection is not null, line 8 and 9.

A non-local update, leaves the tree structure with two variants of the object: the new version and the slightly outdated version. So, in order to distinguish between these two, a check for logical deletion is performed. An object is logically deleted when its $t_u < 0$ and it is retrieved by the query only if its update stamp is greater than the query start time, $|t_u| > t_s$. This means that the object suffered an update during the query running interval and it is preferable for the query to consider this slightly outdated version, and not risk missing the object all together by searching for the newer version.

Algorithm 1: *Search(node:N, rectangle:Rec)*

1 **if** N *is not a leaf* **then** `// internal node`
2   **foreach** *entry:e in the internal node* **do**
3   **if** *e.MBR* $\cap$ $Rec \neq \emptyset$ **then**
4   invoke $Search$ on the child node pointed by *e.child ptr*;
5 **else** `// leaf node`
6   **foreach** *entry:e in the leaf node* **do**
7   **if** *e.MBR* $\cap$ $Rec \neq \emptyset$ **then**
8   construct a circle of radius $min(v_{max}|t_s - e.t_u|, \delta)$, centered at $(e.x,\ e.y)$;
9   **if** *circle* $\cap$ $(Rec - \delta) \neq \emptyset$ **then**
10   **if** $e.t_u < 0$ **then**
11   **if** $|e.t_u| > query.t_s$ **then**
12   consider the e;
13   **else**
14   consider the e;
15   consider the freshest entries in the result set if they are duplicated;

5.3 MBR Updates

A very frequent operation that is important, in the sense that it can become a bottleneck as we get to higher levels in the tree, is that of *MBR* update. In order to not affect performance drastically the update can be done either atomically or by minimal latching for the time of the update. While the first option requires that the *MBR* is no bigger than one word, the second one gives more freedom to the data types but introduces a small delay. An informed decision can be taken by experiment results, so that concurrency is least affected.

A more complex issue is that of propagating the change of an *MBR* up the tree. Eliminating latches and locking totally can lead to lack of integrity of the structure, as shown in [10]. The authors present an example to show how concurrent shrinking and enlargement of the *MBRs* of two children can translate in loss of the integrity of the parent *MBR*. For this reason, we chose to employ the partial lock coupling method, proposed in [10]. The method consists in acquiring shared-latches in case of *MBR* enlargements and using lock coupling in case of shrinking. This means that the exclusive-lock on the child is not released until an exclusive-lock on the parent is obtained. In case a node is full an exclusive-latch is used instead of the shared-latch in order to preserve consistency and not allow a split to occur.

Thus, the update of the *MBR* itself is done atomically or by minimal latching while the propagation is dealt with by using partial lock coupling. The locking and latching is minimal and cannot be further minimized, without losing consistency, as argued above. Concurrency is therefore minimally affected, only to the extent needed to ensure integrity of data.

Next Sections 6 and 7 focus on maintaining consistency when queries and updates are concurrent with node splits and merges, following two distinct approaches.

6 First Approach: Split-Supporting Index

In the following section we present our first algorithm for high-concurrent R-tree operations. In comparison with the R-link [4] tree our approach differs in various ways. The biggest difference is the locking mechanism. The R-link tree allows concurrent search and update operations as well as the presented method but the locking involved in these operations is different. While R-link tree employs shared locks on nodes when a search operation is executed, our approach executes a search operation without locking or latching. This can be done because we make use of the moving objects domain knowledge. Since R-link tree uses lock-coupling during insert operations and applies exclusive locks on a parent and a child, search operations cannot acquire shared locks on these nodes. In our case during update operations we do not acquire any locks or latches as long as the nodes are not split. Furthermore the exclusive latch on a node, in our aproach, is between updates only, because search operations do not acquire a shared latch.

6.1 Main Idea

Structural inconsistencies can result from a node being overfull or underfull. The first case can appear when a non-local update tries to insert an object in a node that is full. In this case the algorithm performs a node split, creating on the side the minimal substructure that reflects the structural modifications caused by the insertion. Upon completion, the substructure is integrated in the structure using atomic operations [3].

The second case can appear once an object is physically deleted and the number of remaining objects in the node is less than the selected minimum number of objects in a node. The algorithm performs in this case a merge, copying entries to one of the neighbouring nodes.

Updates can perform concurrently as long as two distinct non-local updates do not try to split the same node, in which case, one update is blocked and retried later. The latching is minimal and only used in node splitting and merging.

The u-R-tree leaf level node structure is augmented with an additional element - a latch which is used for concurrency control. Modifications can be seen in Figure 8.

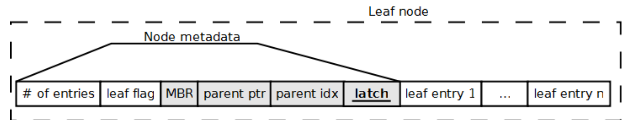


Figure 8: *Structure modifications of a u-R-tree node*

6.2 Local Update

A local update only requires to overwrite the old position of the object, given by x and y, with the newly acquired coordinates and then propagate the changes to *MBR*s upwards. The position of the object in the tree is not modified, therefore local updates can run concurrently without causing any problems. Local updates differ from non-local ones in the sense that they involve just altering information related to the object itself, while non-local updates alter the way objects are stored as well.

6.3 Non-local Update

When an object updates and its new position would require the minimum bounding rectangle (*MBR*) to increase by more than ϵ, the update is considered to be a non-local update, as done in [6, 7]. The algorithm then performs

a top-down traversal in a breadth-first-search approach to find the best *MBR* where to insert the object, line 1 in Algorithm 2. This is done by reading nodes starting with the root and selecting on each level the *MBR* that would require the least enlargement in order to bound the object.

When the object with minimal penalty of the *MBR* is found, a check is performed to see if the node is involved in a split, line 3. If this is the case, then the update is put in a priority queue and retried later. Algorithm 2 inserts the object if the node is not full. After inserting, it logically deletes the old version of the object. This is done by setting the time of update t_u of the old position to the time the update started, with negative sign and setting P_{old} to point to the old node and P_{new} to point to the node where the object was inserted and recomputing object's offset values idx_{new} and idx_{old} respectively, lines 8 to 10. A query cannot miss an object since at any moment of time at least one position of the object is available. It will see either both copies of the object or at least one. The logical deletion of an object means that the object is marked for garbage collection. If the node where to insert is full, then the *NodeSplit* function is called (see Subsection 6.4).

Algorithm 2: *NonLocalUpdate(entry:e_{new})*

```
1  find leaf node, LN, where to insert a new entry with
   object's updated position;
2  obj := object with id e_new.oid;
3  if LN.latch = ON then
4  |   enqueue update in Q_priority ;
5  |   return;
6  if LN ≠ full then
7  |   add e_new in node LN;
8  |   set old-entry.t_u to −currentTime, using obj.P_new
   |   and obj.idx_new;
9  |   set obj.P_old in hash-table to obj.P_new and offsets
   |   respectively;
10 |   set obj.P_new to point to LN and obj.idx_new to
   |   location of e_new in LN;
11 |   propagateChangesMBR();
12 |   return;
13 if LN == full then
14 |   NodeSplit(LN, Nil, Nil, e_new);
```

6.4 Node Split

6.4.1 Overview

A node becomes full when it stores the maximum number of objects that it can store. When an update wants to insert a new object in a full node, the *NodeSplit* function is called. The function's purpose is to produce two nodes, each with at least the minimum number of objects, and considering the R-tree heuristics for minimizing the resulting rectangles' areas. Figure 9 illustrates the split of rectangle N_1 determined by the non-local update of point p_1.

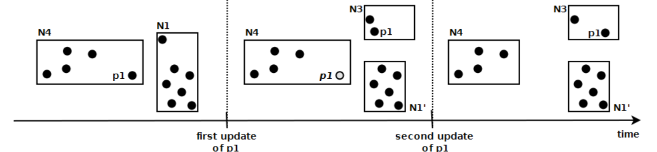


Figure 9: *Split of R_2 due to the non-local update of p_1*

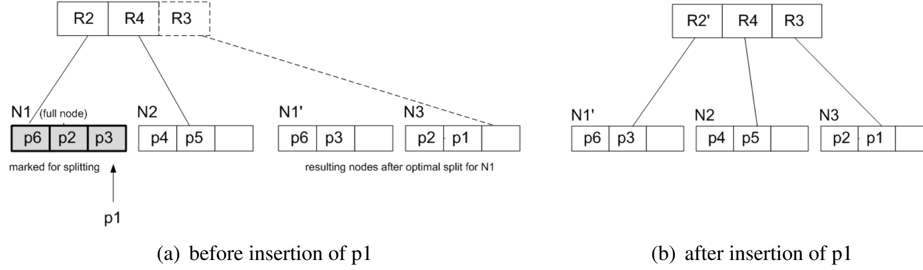
Structural modifications of the index in aforementioned case are reflected in Figure 10. First of all the algorithm ensures that no other update of type non-local can come for objects which are involved in the split by latching the node (node is greyed in Figure 10 (a)). When this preliminary step is finished the two new nodes ($N1'$ and $N3$) are constructed on the side of the index. Objects from node being split are copied and distributed between the two new nodes and object that caused the split (p_1) is inserted into *N3*. When nodes *N1'* and *N3* are ready they are introduced into main index in two atomic operations. This is done by first inserting node containing the object that caused the split, *N3*. Then a swap is performed to change the pointer from *R1* that was full, to point to *N1'* as seen in Figure 10 (b).

A more detailed explanatory walk-through of a node split is given bellow, taking into consideration the cases that can appear and the way the algorithm handles them.

6.4.2 Split

Function *NodeSplit* takes as parameters the node being split, two nodes that were constructed in previous iteration (*Nil* if *NodeSplit* is called for the first time), and the object that caused the split. First of all Algorithm 3 blocks other updates of objects from the node being split in order to avoid conflicts with other updates of objects from the node in question. This is done by setting the bit appointed for latching for all of the leaf level nodes involved in the split. This is reflected in pseudo-code in lines 2 and 5. If latches on the leaf level nodes could not be acquired the update is enqueued in $Q_{priority}$ and retried later.

After this preliminary step is finished function *NodeSplit* creates two substructures by dividing the full node in two nodes of minimal area, line 10. The two substructures are created separately. Sub-trees from the splitting node are copied into the two new nodes, line 11. If *NodeSplit* function is called not for the first time the two nodes from the previous iteration are inserted into the newly created substructures, lines 13 to 15.



(a) before insertion of p1

(b) after insertion of p1

Figure 10: *Split of the node N1*

When the new substructures are completed the algorithm introduces them in the tree structure by performing two atomic operations. This is done by first inserting the group containing the object that caused the split, line 29. Then a swap is performed to change the pointer from the MBR that was full to point to the second group of objects, line 30. But before this step could be completed Algorithm 3 checks in the hash table if any of the objects involved in the split are logically deleted, in order to deal with intermediate positions. An object is logically deleted if its old pointer is not *Nil* (P_{old}≠*Nil*), line 19. Two cases are distinguished here: an object that is logically deleted can have its old position in the node (lines 24 to 25), or its new position in the node (lines 21 to 22). A node split creates a new position for each object involved, therefore in the case of objects that are logically deleted the algorithm stores the pointer to the position involved in the split in an auxiliary variable. The purpose of this, is to be able to perform the physical deletion for these intermediate positions, once the node split is over, since at that point these objects have three copies and the auxiliary information is redundant. Physical deletion for the objects that were not logically deleted before the split is not performed right away, but on the next update of each object, thus the algorithm marks these objects as logically deleted, line 26. This is safe to do due to the assumption that query execution time plus update execution time is shorter than the shortest interval between two consecutive updates of an object. Performing physical deletion right away can cause an object miss for a concurrent query, since it acts similar to a new non-local update of the object. The old node dies slowly with the physical deletion of each entry and it deletes itself with the last entry. Note that the objects involved in the split are available to new local updates as soon as the node split creates their new positions in the tree structures and modifies their new pointers and offset values. Physical deletion is performed for each of these objects at the beginning of their next update.

A query is able to see at least one copy of each object. This is because the algorithm inserts the new substructures using atomic operations, starting with the one containing the object that caused the split, in order to make available the freshest position of that object to queries. A query has a top-down breadth-first-search, and satisfies the assumption that it finishes between two consecutive updates of the same object. Therefore it will see the objects involved in the split either in their old position, if it reaches the node while the *NodeSplit* is running, or the new position, in case it reaches the node after the node split finished.

Local updates are allowed to happen during node splits. An object involved in a split is locally updated once in the full node, when the update comes, and one more time in the node resulting from the split. This is done by the object's own update that sees that the object is involved in a split, by reading the node latch. The old version of the object, the one before the split, is updated right away. After, the update waits until P_{new} is updated to P'_{new}, and then performs the local update, for the second time, on the new version of the object, thus maintaining informational integrity.

6.4.3 Propagating Split

When performing the two atomic operations to include the newly created substructure one particular case scenario distinguishes itself. The algorithm first tries to insert in the parent node the group of objects containing the object that caused the split. The situation in which the parent node is also full can appear, line 16. In this case there is no space for new item in the parent node. The algorithm will perform recursively, line 17. First it will block in the same manner all objects involved in the new node split by calling *TraverseAndSet* function which recursively traverses an internal node and sets latches to all leaf level children. The substructure grows on the side, and will be included in the tree structure by two atomic operation performed at the highest level to where the propagating split takes it. The way the algorithm handles a node split is similar to creating a copy of the substructure involved and performing the update operations on this substructure [1, 9]. However the present algorithm creates a step by step minimal substructure, at each step blocking only the minimal number of updates that would create

Algorithm 3: *NodeSplit(node:N, node:L, node:R, entry:e_{new})*

```
1  if N == leaf node then
2      if TestAndSet(N) == false then
3          enqueue update into Q_priority;
4  else
5      TraverseAndSet(N,list,latched);
6      if latched == false then
7          RemoveLatches(list);
8          enqueue update into Q_priority;
9          return;
10 create two new nodes: N_L and N_R;
11 copy entries from N to N_L and N_R without L and R ;
12 add entry e_new to N_R;
13 if L ≠ Nil AND R ≠ Nil then
14     install entry for L into N_L;
15     install entry for R into N_R;
16 if N.parent == full then
17     NodeSplit(N.parent, N_L, N_R, e_new);
18 else
19     foreach entry: e in leaves of subtrees rooted in
       N_R and N_L where P_old ≠ Nil do
20         if e.t_u > 0 then
21             garbage collect entry in e.P_new at
               e.idx_new;
22             e.P_new:=e.P'_new; e.idx_new:=e.idx'_new ;
23         else
24             garbage collect entry in e.P_old at
               e.idx_old;
25             e.P_old := e.P'_new; e.idx_old := e.idx'_new ;
26     mark each leaf level entry in N_L and N_R as
       logically deleted;
27     compute MBRs of N_L and N_R;
28     update t_u for both versions of the object;
29     construct and atomically add entry to N.parent to
       point to N_R;
30     replace pointer to N in N.parent with pointer to
       N_L;
31     propagateChangesMBR();
```

conflicts.

Algorithm 4: *TraverseAndSet(node:N, list:lst, bool:latched)*

```
1  if N == leaf node then
2      latchSet := TestAndSet(N);
3      latched := latched AND latchSet;
4      if latchSet then lst.add(N);
5      return;
6  foreach entry: e in node N do
7      TraverseAndSet(N, lst, latched);
8  return;
```

6.5 Merge

As argued before, in the regular R-tree, update operations are not efficient due to the reinsertion of deleted objects. The same logic applies for the merge of underfull nodes. To avoid the case in which the entries of an underfull node are simply deleted and reinserted one by one in the index structure, Algorithm 5 follows another approach. The underfull node is merged with a chosen sibling node that is found using heuristics.

Two cases are distinguished in the merge procedure: *(i)* the sibling node has free positions for all the entries of the underfull node, *(ii)* the sibling node would become overfull with the insertion of entries from the underfull node. The first case is reflected by lines 4 to 22 of Algorithm 5. First of all, latches on the underfull and the sibling nodes are acquired. Next, a new node is constructed on the side of the index structure. The entries of the two nodes are copied into the new node. Intermediate values resulting from objects which are already logically deleted are handled in the same way as in the split Algorithm 3. Lines 18 to 19 scan all entries of the newly constructed node and updates pointers P_{new} and P_{old} together with corresponding offsets in the hash-table, in order to reflect the new locations of the objects. When the new node is ready to be introduced in the index structure, the connections between the parent of the underfull node and the new node are created, by atomic swap operations of child and parent pointers, line 20. After the merging is performed, the number of entries of the parent decreases by one and the node can become underfull. In this case, the process propagates upwards by calling procedure *Merge* on the parent.

In the second case *(ii)*, a procedure similar to *NodeSplit*, described in Section 6.4, is called. The main difference between the regular split and the *MergeSplit* procedure is that, instead of taking one entry as argument, all entries from the underfull node are considered, in order to perform a redistribution of entries between the resulting

two nodes.

Algorithm 5: *Merge(node:N)*

1 S := *FindSiblingtoMerge(N)*;
2 let *M* = maximum number of entries in a node;
3 **if** *N.# of entries + S.# of entries* $<$ *M* **then**
4   **if** *N is a leaf* **then**
5   **if** *TestAndSet(N)* $==$ *false* ***OR*** *TestAndSet(S)* $==$ *false* **then**
6   enqueue merge into $Q_{priority}$;
7   **else**
8   *TraverseAndSet(N,list,latched)*;
9   *TraverseAndSet(S,list,latched)*;
10   **if** *latched* $==$ *false* **then**
11   *RemoveLatches(list)*;
12   enqueue merge into $Q_{priority}$;
13   **return;**
14   construct new node on the side, N_{new};
15   copy entries of N into N_{new};
16   copy entries of S into N_{new};
17   logically delete items in *N* and *S*;
18   **foreach** *object in each entry of* N_{new} **do**
19   update pointers and offsets in the hash-table for each leaf node;
20   atomically swap pointers from *N.parent* to *S* with *N.parent* to N_{new};
21   **if** *parent(*N_{new}*) = underfull* **then**
22   *Merge(parent(*N_{new}*))*;
23 **else**
24   call *MergeSplit(S, N.entries)*;

7 Second Approach: Split-Free Index

The development of the second approach is driven by the goal of further enhancing concurrency in the system. The splits and merges that previously needed latching of nodes are now reduced to non-local updates. This implies only the use of short term latches for implementing atomic operations and the partial locking needed in MBR updates. The advantages brought by this approach are the decrease in latching and the algorithmic simplicity. These are achieved at the cost of needing to adjust heuristic parameters and no longer ensuring a minimal number of entries in a node.

7.1 Motivation

The main disadvantages of existing R-tree structure variants and also of our first considered approach, are the high complexity and performance decrease determined by node splits and merges. The complexity of the first approach derives from the so called *artificial updates*, which happen when an object is moved to another node of the index structure even though it did not report a new position. During the process of splits of merges, the objects artificially update. Additional space and concurrency control are needed when having to store three versions of the same object: a logically deleted one, a new one and a copied one involved in a split or merge. Also, the garbage collection of old entries is complicated, since the cause of the logically deletion can be twofold: an artificial update or a non-local update.

In an attempt to avoid these drawbacks we propose a second approach, where the splits and merges are emulated by *forced non-local updates* of objects. Forced updates differ from the artificial ones in the sense that they correspond to the time when the object locally updates and do not imply moving an object in the interval between its updates. Queries are able to deal with the movement of objects in the index structure due to concurrent non-local updates, as it has been described in the previous Section 5. In the next subsections we discuss how the new approach works and the new index structure that it requires.

7.2 Main Idea

The second approach simulates node splits and deals with underfull nodes by determining objects to perform non-local updates to another node. The index structure rearranges gradually without the necessity of copying subparts of the tree or locking.The purpose of rearranging the objects between the nodes is to maintain the number of entries in a node in an optimal interval. The state of a node which corresponds to this latter case, will be from now on referred to as the normal state.

Four heuristic parameters are used to decide in which state a node finds itself. By modifying these parameters it is possible to control the dynamics of the structure. As is can be observed in Figure 11, a node has four states, depending on its number of entries in relation to the four parameters, plus the normal state. A node is considered physically underfull if its number of entries is under a limit called *PU*. The state logically underfull corresponds to the number of entries being in the interval *(PU, LU]*. If the number of entries is greater than the limit *LO*, the node is considered logically overfull and if it is equal to parameter *PO*, the node is physically overfull. The gray area in the figured node denotes the preferred interval for the number of entries, for which the node is in the normal state.

The distinction between logical and physical states of underfull or overfull is required in order to maintain a node in the normal state for as long as possible. The goal is to recognize an impending overfull or underfull node early enough and take action. In addition these four

parameters ensure that the distribution of objects across the tree structure is balanced. Basically, once a node loses many objects, due to non-local updates, and LU is reached, the node will have a greater probability to attract objects, in order to regain its normal state. If the process continues despite the effort of attracting objects, the node will reach the PU limit, moment from which the node cannot be saved anymore. In this latter case all updates of objects from such a node are directly considered non-local. This way the time that a node spends with a very small number of objects is minimal. Similar LO is used to indicate that a node is about to reach its limit, moment when a new node is created in order to compensate splits, and some objects of the old node are moved to the newly created one, by performing non-local updates on their next update.

Thus, LU and LO are used to signal that a node is heading towards states that are not desired, and for some time actions can be performed to prevent this from happening.

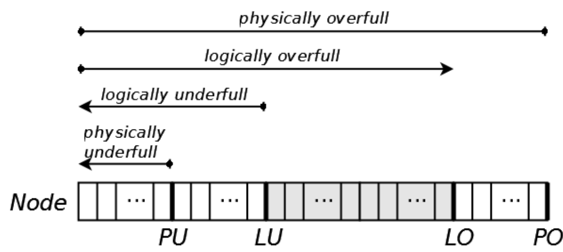


Figure 11: *Node fill factor in relation to four parameters*

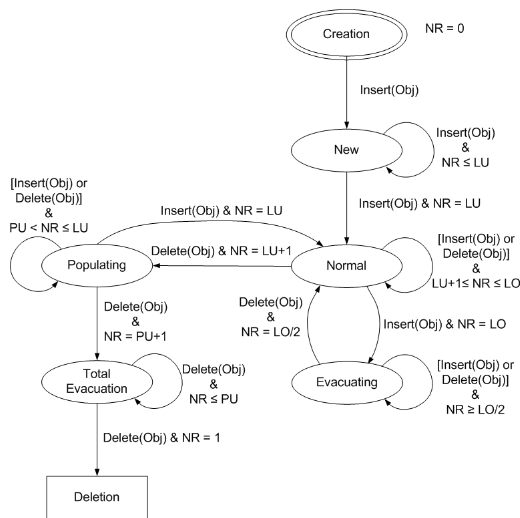


Figure 12: *Diagram of node states*

In order to better illustrate a node's states with regard to the actions that can be performed, from now on we refer to the naming convention presented in Figure 12. The state diagram illustrates all the states of a node during its lifetime and the conditions that determine the transition from one state to another. All nodes start with a state called *Creation* that coincides with the physical creation of the node. This is a short-lived state that ends as soon as the object that caused the creation of the node is inserted. Once this happens, the *New* state is reached. The node will stay in this state, until enough objects are inserted in the node. The idea is to bring a newly created node to the *Normal* state. When the node holds LU objects and an insert operation comes, the number of objects will be LU+1, causing the node to reach its *Normal* state. From this moment on this state becomes an intermediate position between the *Populating* and the *Evacuating* states.

The *Normal* state symbolizes the interval in which we want the number of objects to evolve. When LO, the higher limit of the interval is reached, the state becomes *Evacuating*. In order to return to the normal state the object that causes the number to reach LO+1 will create a new node and symbolize this node for evacuation. A part of the objects will be forced through non-local updates to the new node, thus decreasing the number of objects and returning the node in the normal state.

Similar when LU is reached, the node's state becomes *Populating*. The node needs objects to be inserted in order to return to the *Normal* state, therefore the node signals this using a value called *iNeed*. The higher this value is, the more objects a node needs. In the case the number of objects keeps decreasing while in *Populating*, the node can reach the *Total Evacuation* state. In this situation the only next possible state is *Deletion*. The reason for this is based on the fact that as long as the node has lost objects, even if it was flagged to attract, PU indicates the limit from which we consider that a node cannot be saved anymore. In this state, insertion is not permitted and all objects from the node that update, are forced into non-local updates. The node loses all objects step by step and is deleted after it loses the last object.

7.3 Dealing with Node Splits

Several actions have to be performed in order to ensure that the mechanism of virtual split emulates a real structural modification split:

- an overfull node needs to be flagged for evacuation;
- a new node has to be created and flagged for population;
- a part of the objects of the overfull node need to perform forced non-local updates;
- forced non-local updates are directed to the new node;
- a part of the insertions in the flagged for evacuation

node need to be redirected to other nodes;

- the old node keeps a pointer to the new one for fast access;
- the old node has to be unflagged when a fixed number of entries have been evacuated;

The creation of a new node is performed by the first object that wants to insert in a node that has reached LO number of objects. At the same time the node that is logically overfull begins to be evacuated, in the sense that a number of objects will move through non-local updates to the new node.

One design decision refers to the *MBR* of the newly created node. Considering that the objects that are forced to move in the new node affect its *MBR*, it suffices to choose as the initial bounding rectangle, the area covering the first object in the node. This way, the bounding rectangle will be enlarged step by step. Choosing a fixed part of the old node's *MBR* or the *MBR* entirely as the *MBR* of the new node, would cause an increase in the overlapping between rectangles and affect the query performance. Having a prefixed new area is unnecessary because the forced updates redirected to the new node are directly inserted without checking the bounding rectangle penalty.

The next question that needs to be answered is: how many and which objects from the old node should be forced to move into the new one? To minimize the overlap area, the objects located in the neighborhood of the one that caused the "split" are the ones redirected to the new node. The original rectangle is virtually split in two parts: the persistent part and the evacuating part. Local and non-local updates of objects from the persistent part require no special treatment. In contrast, on each update (local or non-local) of an object from the evacuating part, the link to the new node is used to locate the destination leaf where to insert. The neighbor-area that needs evacuating is determined by an algorithm, *PickCut*, which considers the distribution of the objects in the bounding rectangle and computes the axis (X or Y) and an exact coordinate value of the split. The number of objects from the evacuating area has to be at least *LU + 1*, in order to ensure that the new node does not remain logically underfull. From this and the fact that is not preferable for the old node to remain logically underfull, it results that $LO > 2LU$.

The information about which part of the rectangle is the evacuating one has to be stored in order for the objects in the node to determine to which half they belong. The information has to be non-relative to the rectangle area because the *MBR* suffers modifications due to objects leaving and entering the node. Three fields are needed: one bit to store the axis choice (X or Y), a variable for the value of the coordinate, and one bit to store which part of the bounding rectangle is the evacuating one. These are stored directly in the node for fast access.

The redirection from the old node to the new node must stop before the old node reaches the limit of logically underfull, *nr_of_entries* = *LU* and the evacuation flag removed. The new node attracts objects not only from the old node but also other non-local updates. While a newly created node is populating, the node is not considered logically underfull. The process also runs within a limit, and it stops when the new node reaches a normal state, when its number of entries is between *LU* and *LO*. Then the new node has to be unflagged and treated as a regular node. Non-local updates are attracted to a flagged new node in the sense that the new node has priority over other equally suited nodes, (same penalty of area enlargement), when searching for an optimal leaf where to insert.

The case of non-local updates that find as the optimal leaf a node that is logically overfull and evacuating has to be considered. The insertion process checks the split information and finds out in which half the object belongs to. If the persistent part is the appropriate one, the insert is permitted. Otherwise, the insertion is redirected, using the sibling link, to the new node that needs populating.

Before a new node is created, the parent is checked to see if it is not full. In case the parent is full also the insertion propagates up the tree structure and will insert in the lowest level where it finds space. In this case the object is still inserted on a leaf level, the difference being that one or more internal nodes have to be created as well, depending on what level the insert found free space to insert. After insertion the old leaf node, that was logically full, will hold a pointer to the newly created leaf node, but the parameter that expresses the desire of a node to attract or repel objects, has to be modified accordingly. Since the insertion propagated to a higher level in the structure and more nodes have been created, more objects will have to be inserted in the newly created substructure in order to ensure the population of both the newly created leaf node and all its ancestors up to the level where the insertion took place. The above mentioned parameter will decrease over time as the substructure gets populated with objects as a result of non-local updates or forced non-local updates.

7.4 Dealing with Underfull Nodes

A logically underfull node (*nr_of_entries* $\leq$ *LU*) needs either to be repopulated or eliminated all together. If its number of entries reaches the limit *PU*, then the node starts evacuating and no insertions are permitted into it. The evacuation process is similar to the node split one in the sense that objects are forced out of the node by forcing their local updates to be non-local. The node is emptied one object at a time. The node will finally be removed when it has no entries and the space reused for introduc-

ing new nodes.

Otherwise, if $PU < nr_of_entries \leq LU$, the node can still recover from the underfull state if non-local updates choose it as the optimal leaf where to insert. This is encouraged by signaling the node with a flag and making it a priority over other candidates with a close area enlargement insert penalty.

7.5 No-Split Index Structure

In order to facilitate the changes for concurrency and be able to have an index structure without splits, we introduce additional variables in the node. Figure 13 represents the structure for internal and leaf level nodes, needed by the algorithm with no splits.

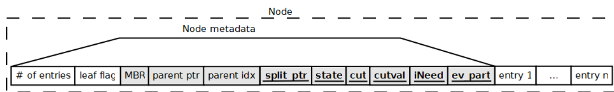


Figure 13: *Modified structure of the u-R-tree node*

The space required for each additional flag in the node is provided in Table 2. As we can see from Table 2, flag *state* can hold four states thus it represents four states node can be in: *Normal*, *Evacuating*, *Populating* and *New* state. *Normal* state denotes that node has the number of elements between *LU* and *LO*. The *Evacuating* state denotes that the node has reached the *LO* or *PU* limit and needs to push away some or all of its entries. The *Populating* state denotes that the node has reached *LU* limit and needs to get some elements. The *New* state characterizes a newly created node which tends to achieve the *Normal* state. The node's state diagram can be seen in Figure 12.

The variable *cut* is used for storing the value according to which axis the node was divided, it stores 1 for y and 0 for x axis. In *cutval* is stored the actual value of the divided coordinate axis. In order to know which part of the node is evacuating we have one bit flag, called *ev_part*. The *iNeed* flag is used for determining if the subtree, rooted in the node, is in need for the additional entries. It can contain negative and positive values signifying that the subtree wants to push away or attract some entries respectively. If *iNeed* is 0 then the subtree is balanced. An evacuating node has also a pointer, *split_ptr*, to another node where updates are redirected.

name	state	cut	cutval	iNeed	split_ptr	ev_part
required space	2 bits	1 bit	8 bits	8 bits	1 word	1 bit

Table 2: *Space required for each field*

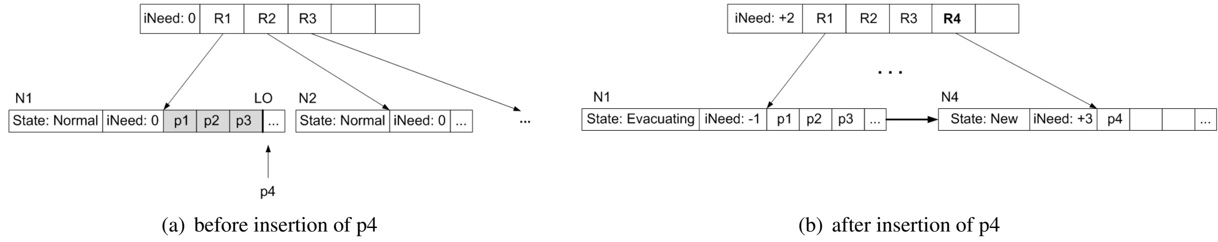
7.6 Basic Example

An abstract view of the main idea of this approach is presented in Figure 14. Let us consider we have a leaf node, *N1*, that contains three elements *p1, p2* and *p3*. In addition the *LO* limit is three. In this moment the node in question is still in *Normal* state. One level above the tree we have the *MBR* of this leaf node, that is *R1*. For reasons of simplicity in understanding, the children of *R2* and *R3* are not shown in the figure. Also the higher levels of the tree structure do not appear. Each node has an *iNeed* value, that shows a node's desire to attract or repel objects. For leaf nodes this value is established based on the state the node is in, while for internal nodes the value is computed based on the values of the children. Since the node is in *Normal* state, the *iNeed* value associated is zero, 14(a).

When a non-local update will try to insert *p4* in this leaf node, it will see that the node has reached the *LO limit*. Therefore, instead of inserting, a new node is created and *p4* inserted as the first element, 14(b). The state of the first leaf node becomes *Evacuating*, symbolizing that the node must get rid of some elements, as it is approaching the *PO* limit. In case of the newly created node, its state will be set to *New*, state in which it will remain until it will reach *LU+1* objects for the first time. A temporary pointer is created from the *Evacuating* node to the *New* node. In the end the new *iNeed* values are computed. The node that is evacuating objects has *iNeed* = -1, while the new node has *iNeed* = +3. The positive or negative amount is given by how much the node must attract or repel objects. A node that was just created needs many objects and as a consequence it will have a high positive value. A node that just entered *Evacuating* state must repel objects but to a different extent. In case more objects are inserted while in *Evacuating*, the *iNeed* value will change accordingly to symbolize an increased desire to repel objects. The internal node's *iNeed* value is given by the values of its children, symbolizing whether objects are needed or not at that level in the tree.

7.7 Algorithm

The update process depicted by Algorithm 6 receives a parameter of type entry, composed of the *oid* of the object that needs to update and the new position coordinates. Two main checks have to be performed to see: (i) if a logically marked old entry of the object exists in the structure and (ii) the type of update. In case (i) happens, the old entry, found by following the P_{old} pointer to the corresponding node, is physically removed from the node. At this moment the *iNeed* value needs to be updated, since the node might have changed its state. The second decision, (ii), takes into consideration the state of the node and *MBR* enlargement penalty. The function *Determine-*



(a) before insertion of p4

(b) after insertion of p4

Figure 14: *Split emulation of node N1*

TypeUpdate(LN, e_{new}) returns the choice *local* for a small penalty of inserting e_{new} in node *LN*. The same is returned if the node's state is *New*, since the node needs to be populated and a non-local update would decrease even more the number of objects in the node. The *MBRs* of new nodes are permitted to unconditionally expand, a limitation is considered only when deciding which updates are suitable for redirecting to the new one.

Algorithm 6: *Update(entry: e_{new})*

1 get P_{new} and P_{old} from hash-table using $e_{new}.oid$;
2 **if** $P_{old} \neq Nil$ **then**
3   N_{old} := leaf node pointed by P_{old};
4   *PhysicalDelete(N_{old}, e_{new})*;
5   *ModifyiNeed(N_{old})*;
6 LN := leaf node pointed by P_{new};
7 type := *DetermineTypeUpdate(LN, e_{new})*;
8 **if** *type == local* **then**
9   *LocalUpdate(LN, e_{new})*;
10 **else**
11   N := *FindNodeHeuristics(e_{new})*;
12   *NonLocalUpdate(N, e_{new})*;
13 **return;**

When an update is cataloged as *non-local*, the function *FindNodeHeuristics(e_{new})* is called. The function's purpose is to find and return the best node where the entry e_{new} is to be inserted. This is done by first checking the siblings to see if any is a possible candidate. Candidates are prioritized by *MBR* enlargement and need of objects, given by *iNeed*. If two or more nodes are suitable for insert, meaning that the *MBR* enlargement is approximately the same, the *iNeed* values are compared. In this case the node with the highest *iNeed* value would be the best candidate, since a high value translates into the node having less than LU objects. The algorithm therefore performs a check one level up the tree and if no candidate is found performs a top-down tree traversal. On each level of the tree, the above steps are repeated ensuring that the object will be inserted in a node that requires a small or no *MBR* enlargement and in addition needs objects more than the other candidate nodes.

A node's need for objects reflects up the tree since a node's *iNeed* value takes into account the *iNeed* values of all the children of that node. However, the function's return node cannot be a physical overfull or underfull node. Once the node is found, it is returned as input for the function *NonLocalUpdate*.

Algorithm 7 sketches the situations that may appear after a logically deleted entry is removed. The node containing the old entry is passed as a parameter to the function *PhysicalDelete* along with necessary information in order to locate the entry inside the node. Removing an object from a node can change the state of the node, in the way presented earlier in the state diagram in Figure 12, when Delete(Obj) transitions occur at the moments when the number of entries in the node represents a state-border.

Algorithm 7: *PhysicalDelete(node:N_{old}, entry:e_{new})*

1 *obj* := object with id $e_{new}.oid$;
2 garbage collect old entry in N_{old};
3 *propagateChangesMBR()*;
4 *obj*.P_{old} := Nil;
5 **if** N_{old}*.state == "populating" **AND***
6   N_{old}*.# of entries == PU* **then**
7   N_{old}*.state := "evacuating"*;
8 **else if** N_{old}*.state == "normal" **AND***
9   N_{old}*.# of entries == LU* **then**
10   N_{old}*.state := "populating"*;
11 **else if** N_{old}*.state == "evacuating" **AND***
12   N_{old}*.# of entries == LO/2-1* **then**
13   N_{old}*.state := "normal"*;
14   *split_ptr* := Nil;
15 **else if** N_{old}*.state == "evacuating" **AND***
16   N_{old}*.# of entries == 0* **then**
17   garbage collect node N_{old};
18   *propagateChangesMBR()*;
19 **return;**

Algorithm 8 deals with the regular local updates that

just update an object's position in a node (lines 16 to 19), but also with the situations in which the local update must be forced to perform non-locally (lines 2 to 14). The first case in which this happens is when the node's state is *Evacuating* and the number of entries is below the *PU* limit. This means that the node is in the process of being eliminated. A suitable node is found by the *FindNodeHeuristics* function and passed as a parameter to the *NonLocalUpdate* function, which deals with inserting the entry with the updated object position in the node.

If the node's state is *Evacuating* because it was overfull, only some of the local updates need to be forced out. As discussed before in Section 7.2, the objects belonging to the "evacuating" area of the rectangle have their updates redirected to the newly created node, located with the help of the *split_ptr* pointer of the current node. If the object is located in the other part of the rectangle, its update is treated as a regular local one.

Algorithm 8: *LocalUpdate(node: LN, entry: e_{new})*

```
1  old-entry := entry in LN corresponding to e_new.oid;
2  if LN.state == "evacuating" then
3      if LN.# of entries ≤ PU then
4          N := FindNodeHeuristics(e_new);
5          NonLocalUpdate(N, e_new);
6      else
7          if InEvacArea(LN, e_new) == true then
8              N := node pointed by LN.split_ptr;
9              NonLocalUpdate(N, e_new);
10         else
11             old-entry.x := e_new.x;
12             old-entry.y := e_new.y;
13             compute LN.MBR;
14             propagateChangesMBR();
15 else
16     old-entry.x := e_new.x;
17     old-entry.y := e_new.y;
18     compute LN.MBR;
19     propagateChangesMBR();
20 return;
```

The function *InEvacArea*, illustrated in Algorithm 9, computes the position of the object relative to the cut of the rectangle that was performed when the "split" process began. The cut information is kept in the node fields: *cut, ev_part* and *cutval*. Depending on which axis the cut was executed, the x or y coordinate of the new position is compared with the value separating the two parts, *cutval*. The comparison also must take into account which part of the rectangle is the evacuating one, the one with values of the *cut* coordinate higher or lower than the *cutval*, information kept in the *ev_part* field. The function returns true if the object is in the evacuating part and false otherwise.

Algorithm 9: *InEvacArea(node: LN, entry: e_{new})*

```
1  if LN.cut == "X" then
2      if LN.ev_part == "higher" AND
3         e_new.x > LN.cutval then
4          return true;
5      else if LN.ev_part == "lower" AND
6             e_new.x < LN.cutval then
7          return true;
8  else if LN.cut == "Y" then
9      if LN.ev_part == "higher" AND
10        e_new.y > LN.cutval then
11         return true;
12     else if LN.ev_part == "lower" AND
13            e_new.y < LN.cutval then
14         return true;
15 return false;
```

Algorithm 10 performs the non-local update, which receives as an input the node in which the new entry must be integrated. This node could have been previously found using the *FindNodeHeuristics* function, or in the case of forced local updates, the node is a newly created one which needs populating. First of all, a check to see if the node is in the *Evacuating* state is performed. At this point in the algorithm, the node can be in this state because it was overfull. The situation when it was evacuating because of the physical underfull condition is treated inside the *FindNodeHeuristics* and *LocalUpdate* functions, before calling the function *NonLocalUpdate*. Therefore, an evacuating node must redirect updates of the objects in the evacuating part to the newly created node (lines 3 to 8).

A non-local update implies an insertion into the selected node. As it was depicted by the node states diagram in Figure 12, an Insert(Obj) transition can modify the state of the node when the number of entries in the node represents a state-border. Therefore, if the node is in *Normal* state but its number of entries has reached the limit *LO*, the node changes its state to *Evacuating*. The update process proceeds with the creation and integration of a new node, having the state *New*, and with the insertion of the new entry in this node (lines 9 to 22).

If the insertion of the new entry is permitted in the node received as a parameter by the *NonLocalUpdate* function, then the process must check if the node changes its state from *Populating* or *New* to *Normal*, due to the increasing of its number of entries (lines 29 to 32).

After an object was inserted or deleted from a node

Algorithm 10: *NonLocalUpdate(node:LN, entry:e_{new})*

```
1  obj := object with id e_new.oid;
2  old-entry := entry in old node pointed by obj.P_old;
3  if LN.state == "evacuating" AND
4    InEvacArea(LN, e_new) == true then
5      N := node pointed by LN.split_ptr;
6      NonLocalUpdate(N, e_new);
7      ModifyiNeed(N);
8      return;
9  if LN.state == "normal" AND
10   LN.# of entries == LO then
11     LN.state := "evacuating";
12     (LN.cut, LN.cutval) := PickCut(LN);
13     create and add new node, N_split;
14     N_split.state := "new";
15     ModifyiNeed(N_split);
16     LN.split_ptr := pointer to N_split;
17     insert e_new into N_split;
18     old-entry.t_u := −currentTime using obj.P_new;
19     obj.P_old := obj.P_new;
20     obj.P_new := location of N_split;
21     propagateChangesMBR();
22     return;
23 insert e_new into LN;
24 ModifyiNeed(LN);
25 old-entry.t_u := −currentTime using obj.P_new;
26 obj.P_old := obj.P_new;
27 obj.P_new := location of LN;
28 propagateChangesMBR();
29 if (LN.state == "populating"
30   OR LN.state == "new")
31   AND LN.# of entries == LU+1 then
32     LN.state := "normal";
33 return;
```

the *iNeed* values of the node and its parents may have to be updated. In order to do this the *ModifyiNeed* and *ModifyiNeedInternal* functions are used, as described in Algorithms 11 and 12. The *iNeed* can take four values to indicate whether the node needs objects or evacuates, and one additional value to suggests the lack of relevance towards a new insert or delete. A leaf node needs objects while its state is populating. The median of this interval is computed and compared to the number of objects in the node. Depending on the outcome the value of *tempiNeed* can be two or one, indicating the node needs many objects or not that many objects. Similar if the node is in the evacuating state same principle applies and values are attributed with negative sign. If the node is not in one of the above states, the value zero is assigned to *tempiNeed*. Once the variable *tempiNeed* is computed, it is compared with the old *iNeed* value. If the values differ, *iNeed* takes the new value and the process continues on the parent level. Otherwise no change is needed.

On internal node levels, the *iNeed* value is computed using Algorithm 12. The value is based on the *iNeed* values of all the children nodes of the node in question. The values are added in order to give a general idea whether objects are needed or being evacuated in the substructure for which that node is root. The process for internal nodes is recursive and changes to *iNeed* values propagate up to root level of the entire tree structure. If a node's computed value of *iNeed* is the same with its old value, the process stops immediately as no more changes have to be made.

Algorithm 11: *ModifyiNeed(node: N)*

```
1  if N.state == "populating" then
2      lim := (LU + PU)/2;
3      if N.# of entries > lim then
4          tempiNeed := 1;
5      else
6          tempiNeed := 2;
7  if N.state == "evacuating" AND
8    N.# of entries > PU then
9      lim := (LO/2 + PO)/2;
10     if N.# of entries > lim then
11         tempiNeed := -2;
12     else
13         tempiNeed := -1;
14 else
15     tempiNeed := 0;
16 if N.iNeed ≠ tempiNeed then
17     N.iNeed := tempiNeed;
18     ModifyiNeedInternal(parent(N));
19 return;
```

Algorithm 12: *ModifyiNeedInternal(node: N)*

```
1 foreach child of N: child do
2     tempiNeed += child.iNeed;
3 if N.iNeed ≠ tempiNeed then
4     N.iNeed := tempiNeed;
5     ModifyiNeedInternal(parent(N));
6 return;
```

8 Conclusions and Future Work

The paper proposes two new approaches that make use of the semantics of the application domain in order to enhance concurrency while minimizing locking and latching. The novelty of the second approach is reflected in the lack of splits and merges while maintaining the integrity of the tree structure. Both methods are presented in detail and pseudo-code algorithms are offered for each operation. We conclude that the main focus, that of concurrency performance improvement, is achieved by both approaches with the advantage of algorithmic simplicity for the later one. Concurrency issues are considered for updates, queries and the interleaving of the two. The methods base their correctness on the assumption presented and discussed in detail in Sections 2 and 5.

The next goal on the agenda is implementing the second approach in order to make a comparative study with other approaches (e.g. CGiST proposed in [5]). Another goal consists in implementing the first approach as well, in order to conduct experiments with regards to the performance of both methods proposed in this paper.

References

[1] Jens Dittrich, Lukas Blunschi, and Marcos Antonio Vaz Salles. Indexing moving objects using short-lived throwaway indexes. In *Proceedings of the 11th International Symposium on Advances in Spatial and Temporal Databases*, SSTD '09, pages 189–207, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[3] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 265–279, London, UK, 2002. Springer-Verlag.

[4] Marcel Kornacker and Douglas Banks. High-concurrency locking in r-trees. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 134–145, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[5] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein. Concurrency and recovery in generalized search trees. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, SIGMOD '97, pages 62–72, New York, NY, USA, 1997. ACM.

[6] Dongseop Kwon, Sangjun Lee, and Sukho Lee. Indexing the current positions of moving objects using the lazy update r-tree. In *MDM '02: Proceedings of the Third International Conference on Mobile Data Management*, pages 113–120, Washington, DC, USA, 2002. IEEE Computer Society.

[7] Mong Li Lee, Wynne Hsu, Christian S. Jensen, Bin Cui, and Keng Lik Teo. Supporting frequent updates in r-trees: a bottom-up approach. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '2003, pages 608–619. VLDB Endowment, 2003.

[8] Vincent Ng and Tiko Kameda. Concurrent access to r-trees. In *Proceedings of the Third International Symposium on Advances in Spatial Databases*, SSD '93, pages 142–161, London, UK, 1993. Springer-Verlag.

[9] Rajeev Rastogi, S. Seshadri, Philip Bohannon, Dennis W. Leinbaugh, Abraham Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databases. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 86–95, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[10] Seok Il Song, Young Ho Kim, and Jae Soo Yoo. An enhanced concurrency control scheme for multidimensional index structures. *IEEE Trans. on Knowl. and Data Eng.*, 16:97–111, January 2004.

[11] Darius Šidlauskas, Simonas Šaltenis, Christian W. Christiansen, Jan M. Johansen, and Donatas Šaulys. Trees or grids?: indexing moving objects in main memory. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 236–245, New York, NY, USA, 2009. ACM.