

Improving the R*-tree with Outlier Handling Techniques *

Tian Xia
tianxia@ccs.neu.edu

Donghui Zhang
donghui@ccs.neu.edu

College of Computer and Information Science
Northeastern University
360 Huntington Avenue, Boston, MA 02115

ABSTRACT

The R*-tree, as a state-of-the-art spatial index, has already found its way into commercial systems like Oracle. In this paper, we aim at improving query performance of the R*-tree. We focus on five widely used spatial queries: range query, aggregation query, nearest neighbor query, skyline query, and join query. The idea is to store *outlier* objects in internal tree nodes. The new structure is named the R^0 -tree. Here an outlier is an object which is located far from other objects or has large extent (we consider both point objects and objects with extent). If such objects are stored at higher levels of the tree, the lower-level nodes have smaller minimum bounding rectangles and thus the index performs better. To support the dynamic nature of the index, several structural and algorithmic changes are needed. The paper discusses these changes. In particular, we show how to identify and handle the outlier objects during page overflow/underflow, using gain/loss metrics. Extensive experiments reveal that the R^0 -tree significantly outperforms the R*-tree for all the five queries.

Categories and Subject Descriptors

H.2.8 [Database Management]: Database Applications—*Spatial database and GIS*

General Terms

Algorithms, Design, Performance

Keywords

R*-tree, outlier

1. INTRODUCTION

Spatial databases have many applications like mapping, urban planning, transportation planning, resource manage-

*Partially supported by NSF CAREER Award IIS-0347600.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GIS'05, November 4, 2005, Bremen, Germany.

Copyright 2005 ACM 1-59593-146-5/05/0011 ...\$5.00.

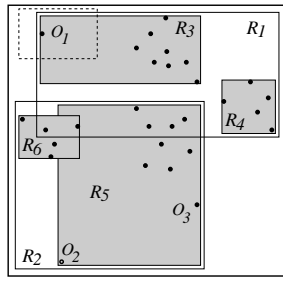
ment, geo-marketing, archeology and environmental modelling [6]. Some major database vendors have provided extensible spatial database support, e.g. Oracle Spatial Cartridge [14] and IBM Informix Spatial DataBlade [11]. Given a large collection of spatial data, it is crucial to index them so as to support efficient query processing. One of the most successful spatial index structures is the R*-tree [3] (a variation of the R-tree [8]), which has already been implemented in commercial database management systems (DBMS) like Oracle [1].

The R*-tree is a balanced disk-based tree structure, which recursively clusters spatial objects (and the clustered disk pages) based on the proximity of their spatial locations. Every index entry is associated with a minimum bounding rectangle (MBR) of all objects in the sub-tree. In order for the R*-tree to achieve good query performance, it is desirable to have small MBRs. If every MBR stored in the tree is nearly as large as the whole space, the index is useless as any query needs to examine roughly the whole tree. The most important improvement of the R*-tree over the original R-tree is that it utilizes forced reinsertion. That is, if a disk page (node) overflows, some entries are removed from the page and reinserted into the index.

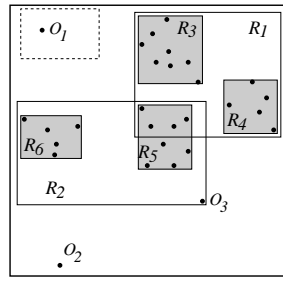
Arge *et al.* [2] presented the Priority R-tree, which, unlike other variations, can guarantee optimal worst-case range query performance. However, to guarantee so, all data have to be known in advance, i.e. the index is static. We focus on dynamic variations of the R*-tree. Our previous work [20] introduced several optimizations to the R*-tree, e.g. we introduced a better way of picking objects to reinsert. Three metrics were defined: the *quality* of an MBR, the *gain* of removing some objects from an MBR, and the *loss* of adding some objects to an MBR. The optimizations (reviewed in Section 2) rendered an 20% improvement on range query performance.

This paper is a novel extension to [20]. We propose a new index structure named R^0 -tree, which handles *outlier* objects gracefully. Here, an outlier object is located far from other objects, or has large spatial extent. The R^0 -tree not only incorporates all improvements proposed in [20], but also has some important structural and algorithmic changes to the R*-tree, by storing the outlier objects at higher index levels.

Furthermore, while [20] only addressed the range query, in this paper we discuss five widely used spatial queries, i.e. the range query (find objects in or intersecting with the query region), the aggregation query (find some aggregate value of objects in a query region), the nearest neighbor query (find the object whose distance to a given location is the closest),



(a) The page layouts of a R^* -tree



(b) The page layouts of a R^0 -tree.

Figure 1: Motivation example: by storing outlier objects at higher levels, the R^0 -tree has smaller MBRs and better query performance.

the skyline query (find every object which is not dominated by any other object, i.e. does not have a smaller X and a smaller Y than the other object), and the join query (find all pairs of objects, one from dataset A and the other from dataset B, such that they intersect).

Our motivation to the R^0 -tree is that if an object O is far from all the rest objects, in the R^* -tree it is inevitable that the leaf page that contains O will have a large MBR, and consequently all ancestor nodes of the leaf page also have large MBRs. The outlier identification is integrated throughout the construction/update of the R^0 -tree, e.g. in the reinsertion process, in the overflow/underflow handling, and etc. An example is shown in Figure 1. In the R^* -tree, O_1 is an outlier object. O_1 is contained in a leaf node R_3 , whose parent node is R_1 . Both nodes have large MBRs. To perform a range query of finding objects in the dashed rectangle, three tree nodes (the root, R_1 and R_3) need to be accessed. In contrast, the R^0 -tree, as shown in Figure 1(b), maintains object O_1 in the root node. The benefit is that tree nodes R_3 and R_1 have smaller MBRs, and the above range query only need to access one node (the root). Similarly, by maintaining O_2 and O_3 at higher levels, the leaf node R_5 has a much smaller MBR. We point out that unlike O_1 and O_2 , object O_3 is NOT stored in the root node. Instead, it is stored in node R_2 as a *local* outlier. Note that although in the example we considered point objects, our idea applies to objects with extent as well. Our outlier identification algorithm handles point objects and objects with extent in a uniform way.

In order to implement the idea of storing outlier objects at higher levels of the tree, some interesting issues need to be studied: How to dynamically handle these outlier objects during updates – in particular, how to handle page overflow/underflow? How are the query algorithms affected? In order for commercial systems to incorporate our new idea, two important questions arise: How much performance improvement can we get? How much time we need to construct/update the R^0 -tree compared to the R^* -tree? This paper addresses the above questions.

The key contributions of this paper are summarized below.

1. We propose the R^0 -tree which implements the idea of storing outlier objects in index nodes to improve query performance. Changes are needed for the insertion/deletion algorithms. We describe methods to handle page overflow and underflow, which dynamically adjust the tree such that outlier objects can be

promoted/demoted systematically while the tree still maintains a guaranteed minimum fan-out. Once the structure is built, we argue that insignificant changes are needed for the query algorithms.

2. We report extensive experimental comparison on real datasets between the R^* -tree and the R^0 -tree for five widely used spatial queries, for update cost and index sizes. The R^0 -tree is unanimously superior. In particular, for the range query, the R^0 -tree is more efficient than our previous result [20], which is more efficient than the R^* -tree.

The rest of this paper is organized as follows. Section 2 shows the related work. Section 3 presents the structure of the R^0 -tree, and the details of R^0 -tree constructions. Section 4 presents modified algorithms for the five important spatial queries. The performance results appear in Section 5. Finally, Section 6 concludes this paper.

2. RELATED WORK

Over the past twenty-five years or so, there has been extensive work on indexing spatial data. Quite a few index structures have been proposed, as surveyed by [7, 19, 4]. There is no consensus on the *best* spatial index structure. However, the R-tree and its variants [8, 3] are widely implemented and have found their ways into commercial systems [13, 1].

In this section, we first give a review of our previous work [20] as the background of this paper. Then we compare our work with the promotional-based index proposed by Kanth, *et al.* [12].

2.1 Review of the *Quality/Gain/Loss* Metrics

In [20], we mathematically defined the *quality* of an MBR, the *gain* of removing some objects from an MBR, and the *loss* of adding some objects to an MBR. The work was motivated by a piece of common knowledge and an observation about the R^* -tree.

It is well known ([3]) that to improve range query performance, we should make each MBR: (a) have a small area; and (b) have a square-like shape. The reason is that an MBR, with a large area and/or a long-and-thin shape, is likely to intersect with other MBRs and the query regions. The R^* -tree aims at improving on this, by introducing forced reinsertion. If a node overflows, a fixed percentage (e.g. 30%) of objects (denoted as p objects) are removed from

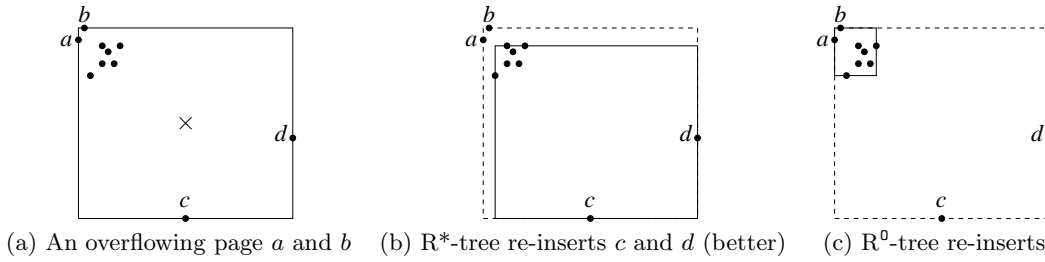


Figure 2: To re-insert outlier objects instead of the objects that are the farthest to the center of the original MBR, may result in a smaller MBR.

the overflowing page (and reinserted into the index). The R*-tree picks the objects whose distances to the center of the MBR are the largest. We observe that the action may not achieve the two goal (making a MBR have a small area and a square-like shape). In Figure 2(a), if two objects are to be removed, the R*-tree will remove a and b , resulting a slightly smaller MBR. Based on our metrics, we should remove the objects with maximum *gain*. In this example, objects c and d (Figure 2(c)).

Definition 1. Given a rectangle r with width w and height h , the **quality** of the rectangle is defined as

$$Q(r) = \frac{1}{w * h} * \left(\frac{\min\{w, h\}}{\max\{w, h\}} \right)^\alpha \quad (1)$$

where $\alpha \in [0, 1]$ is a constant.

Definition 2. If a rectangle r_1 is shrunk to r_2 (r_1 spatially contains r_2), the **gain** is defined as

$$G(r_1, r_2) = 1 - \frac{Q(r_1)}{Q(r_2)} \quad (2)$$

The **loss** of expanding a rectangle from r_2 to r_1 is the same as the *gain* of shrinking a rectangle from r_1 to r_2 . Also, all the three metrics can be defined for entry sets instead of rectangles, by considering the MBRs of the sets. For instance, the *quality* of a set of objects is the quality of their MBR.

Based on the metrics, we proposed several improvements on the R*-tree. First, the forced reinsertion should remove a set of entries which will bring the maximum gain. These entries are defined as the *p-boundary* below. Second, while the R*-tree always remove a fixed percentage of entries, we do better. If by removing fewer than p (say 5) entries we already can achieve a parameter β (e.g. 98%) of the gain of removing the *p-boundary*, why should we perform the costly reinsertion operation on all p entries? We define the *minP-boundary* as the set of (fewer than p) entries that should be removed. Third, to accommodate a new object, the subtree whose expansion has the least loss (versus the least area increase) is chosen. And finally, when splitting a node, to determine which group an entry belongs to, we pick the group which has minimum loss to incorporate the entry.

Definition 3. Given a set S of spatial objects and a number p , the **p-boundary** of S is a subset P consisting of p objects such that the gain of removing these objects, $G(S, S - P)$, is maximal among all cases of removing p objects.

Definition 4. Given a set S of spatial objects, a constant $\beta \in [0, 1]$ and a number p , the **minP-boundary** of S is a subset T consisting of the smallest number of objects such that $G(S, S - T) \geq \beta \cdot G(S, S - P)$, where P is the *p-boundary* of S .

Several algorithms were proposed in [20] to find the *p-boundary* (minP-boundary). An exhaustive algorithm finds the exact set of the *p-boundary* (minP-boundary) by checking all combinations of shrinking the borders of an MBR. However, its time complexity is too high. Another more efficient greedy algorithm finds an approximate set of the *p-boundary* (minP-boundary). The basic idea of the greedy algorithm is to always choose the border which, if we remove all objects on it, will lead to the largest average gain per removed object. Experiments showed that the greedy algorithm achieves comparable gain as the exhaustive algorithm with much less running time.

2.2 Comparing with Promotional-based Index

A work related to ours is the *promotion-based index* proposed by Kanth, *et al.* [12]. They can store objects with large extent at higher levels of the tree in the following way. To insert a new object A into a node N , if N contains a promoted object B and A spatially contains B , A is stored in N . Otherwise, A is inserted into some sub-tree using the same strategy of the R*-tree. If a page overflows, they first try to demote one promoted object, then try to promote one object to parent, and last try to split the node as in the R*-tree. Take promotion as an example: there are two kinds of criteria to promote an object to higher levels. (1) Nesting-based promotion: if an object contains at least a threshold number (e.g. six) of other objects in the node, the object can be promoted. (2) Extent-based promotion: if removing an object shrinks the area of its enclosed MBR for a threshold fraction (e.g. 20%), it can be promoted. The demotion criteria is the reverse of the promotion criteria.

While the promotion-based index only focused on the range query, this paper addresses five queries. Even for the range query itself, our R⁰-tree is better in various aspects. In the following discussions, we compare our R⁰-tree with their structure from the theoretical viewpoint.

First, our structure works well for both point objects and objects with extent, while the promotion-based index works well for a special kind of objects with extent, where objects enclose each other. In their experiments, they showed good performance on a set of data which has 36 levels of nesting (object O_1 is enclosed in O_2 , which is enclosed in O_3, \dots , which is enclosed in O_{36}). Most real datasets are not like that. If their method is applied to point data, the perfor-

mance improvement is not as significant (they reported 5% improvement on range queries, while we reached 48%). Our index has better insertion cost as well, since an outlier object can stop at an index level. But their method, if applied to point data, will insert every object to some leaf node, since no containment relationship exist among point objects.

Second, once a leaf node overflows, the promotion-based index only promotes one object, if exists. The R^0 -tree will ‘promote’ multiple objects based on re-insertion. Having the ability to promote multiple objects is better, since it increases the chance to solve the overflow problem via promotion (versus splitting, the last resort). Imagine the case when there exist at least two objects on each boundary of the MBR. Promoting a single object does not reduce the MBR at all. In this case the promotion-based index will determine that promotion is not possible. But what if promoting two objects will significantly reduce the MBR? A side note is that, even though the original R^* -tree does forced re-insertion as well, our R^0 -tree is again better, not only because we can choose a better set of objects to be re-inserted (with maximum gain). The original R^* -tree always chooses a fixed percentage of objects to re-insert. Suppose it chooses to re-insert 20 objects. What if after the removal of 5 objects, to remove more objects hardly result in any gain (the MBR does not shrink or only shrink a little bit)? The R^0 -tree is more versatile in that it can choose to remove only 5 objects in this case. It is better because to avoid re-inserting too many objects saves on the insertion cost.

Third, to our best understanding, the promotion-based index does not guarantee a minimum fan-out. If an index node overflows, before splitting the node, it will try to demote or promote an object stored in the node. It is possible that no promotion or demotion could be applied. For instance, if removing any single object does not change the area of the MBR, promotion is not possible. In this case, the only choice is to split the node. Notice that when overflows, the node may have only a few index entries plus many promoted objects. The promotion-based index does not deal with such nodes specially. Thus in the resulted two nodes, the number of index entries may be smaller than any pre-defined threshold. A possibility is that the split of an index node results in a node with no index entries, but only promoted objects. Thus, the tree may not be balanced. In our experiments, we compare our R^0 -tree only with the original R^* -tree, not with the promotion-based index.

3. THE R^0 -tree

3.1 The Structure

Similar to the R^* -tree, the R^0 -tree is a height-balanced, disk-based, and dynamically updateable tree structure. The leaf nodes only contain objects and are all located at the same level. Index nodes contain index entries and are located at higher levels. At the highest level, there is a single node: the root node, which contains at least two children unless it is a leaf node. A difference is that an index node in an R^0 -tree may also contain some objects.

Assume the page capacity of an index page is M . That is, the number of index entries plus the number of outlier objects should not exceed M . An index node (except the root) must have at least m index entries to ensure the fan-out of the tree. Typically in the R^* -tree, m is half of the node capacity M . However, in the R^0 -tree, an index node

may split while containing some outlier objects. In order for the split nodes to satisfy the minimum fan-out, m should be smaller (but not too much smaller) than $M/2$, e.g. we choose $m = 0.4M$.

To allocate space between the index entry part and the object part in an index node, a naive way is to pre-determine a cut-off point, e.g. $M - 2m$ entries are reserved to store outlier objects. This static allocation, although simple, is not space efficient. A split may occur at a node which is not full. A better way is to dynamically allocate the space to both parts. In this approach, there is no fixed cut-off point, and as long as there is space in the node, either an index entry or an outlier object can be put into the node.

3.2 Identifying Outliers in a Page

To identify outliers in the dataset and store them at higher levels, we do not perform additional data scans or clustering algorithms on the whole dataset for update efficiency. Instead, our outlier identification process works seamlessly with the insertion and deletion process of the R^0 -tree, which will be presented in Section 3.3 and 3.4. In this section, we address the issue of how to identifying outliers in a page.

In the R^* -tree, when a page first overflows, a fixed number of objects in the page are removed and reinserted again. We utilize the reinsertion process in the R^0 -tree as an opportunity to identify outliers in a page. However, as we stated in Section 2.1, the original R^* -tree reinsertion algorithm may not identify real outliers. Therefore, in the R^0 -tree implementation, we use an improved version of reinsertion algorithm which finds the minP-boundary (which was reviewed in Section 2.1). In addition, if the gain of removing the minP-boundary is too small (say, below a threshold of 0.1), no object is identified as an outlier. In the following sections, the outlier identification algorithm is specifically referred to the greedy algorithm to identify the minP-boundary [20].

3.3 Insertion and Overflow Treatment

To insert a new object O into the sub-tree rooted by node N , we check the index entries stored in N . If O is not contained in the MBR of any index entry stored in N , we will store it as an outlier in N . Otherwise, there exists an index entry E where $E.mbr$ contains $O.mbr$. We recursively insert O into the sub-tree rooted by $page(E)$. If there exist multiple index entries whose mbr contains the object, the tie is resolved by choosing the one whose mbr has a smaller area.

When a page N overflows, it temporarily contains $M+1$ entries. To handle the overflow, we first try to remove some outlier entries from the page and re-insert the objects back into the index. There are three differences from the R^* -tree’s re-insertion. First, the set of outliers to be removed is computed using the outlier identification algorithm. Equipped with the gain/loss metrics, our algorithm tends to result in an MBR with higher gain. Second, while the R^* -tree always chooses a fixed percentage of entries (say p entries) to re-insert, our outlier identification algorithm is more versatile. For instance, if removing all the p entries does not bring any gain or only brings in a very small gain, the outlier identification algorithm reports “no outlier found” and the R^0 -tree does not re-insert any object. Also, if by removing less than p objects, we already can get a significant gain, the R^0 -tree will remove less than p objects. This is good because removing fewer objects means better insertion cost.

Third, the outlier entries may contain both index entries and outlier objects.

If re-insertion is not possible, we can either split the overflowing page N , or demote some outlier object to lower levels (if N is an index page). Our intuition is: the larger $num_of_children(N)$ is, the more likely we should split N , and vice versa. For example, if all the $M+1$ entries are index entries, the only choice is to split N . To the other extreme, if N has less than $2m$ index entries, split is not a choice, otherwise one of the result page will violate the minimum fan-out requirement. Our preliminary experimental results show that a good breaking point is $m + M/2$. That is, if $num_of_children(N) < m + M/2$, push down an outlier, otherwise split. To push down an outlier object, we choose the sub-tree which has minimum loss to accommodate the new object.

Here an important implementation issue arises. A naïve implementation is to compare every outlier object against every child entry to find the pair with the minimum loss. However, this is expensive in terms of CPU cost. A better way is to keep the outlier objects in increasing order of optimal loss. Here the *optimal loss* associated with an outlier object is the minimum loss if we push down the object into some child node. Notice that right before an object was first registered as an outlier into an index page, we had already compared it against all child entries (so as to determine that the object was not contained in the MBR of any child entry). Thus at that time, we could compute the optimal loss of the object. Once we maintain the set of outlier objects in increasing order of optimal loss, the order does not change unless there is some change in the set of child entries (addition, deletion, or MBR change). At that time, we could choose to re-order the outlier objects.

If we choose to split an index node N , we first divide the set of index entries into two groups as in the R*-tree, then assign each outlier object into one of the groups. In both cases, once the seed of each group is chosen, a component task is to assign an entry into a group. We choose the one which has minimum loss to accommodate the new entry.

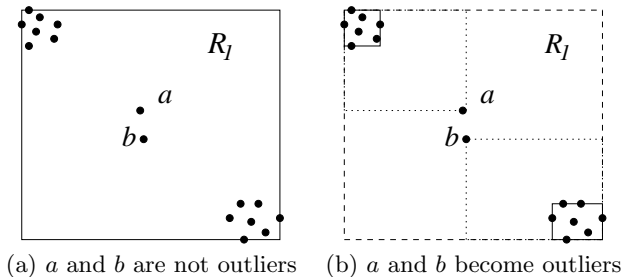


Figure 3: An example which illustrates the usage of outlier identification after a page splits.

If the overflowing page is a leaf page, we need to split the page as described above. That is, each object is assigned to the group with minimum loss. Afterwards, for each group we have a choice to identify some outlier objects to be stored in the parent node. Consider the example in Figure 3, objects a and b are not considered to be outliers in Figure 3(a), while they become outliers after the splitting of R_1 in Figure 3(b). If they are stored in the parent page, the resulted two leaf pages have smaller MBRs.

However, there is a tradeoff here. If we choose to identify outlier objects after a leaf split, we will have to choose a smaller number as the minimum required number of objects in a leaf page. For instance, without the outlier identification step, theoretically we can require each leaf page to be at least half full. But with outlier identification, this requirement has to be relaxed. In the R⁰-tree, the minimum number of objects in a leaf page is chosen to be 40% of the page capacity, which allows the identification of limited number of outliers.

3.4 Deletion and Underflow Treatment

In the R⁰-tree, an object to be deleted may exist in either an index page or a leaf page. If it is in an index page, we remove it and adjust the MBR of the page and its ancestors, if needed. No underflow treatment is necessary (an index node underflows if it has fewer than m index entries).

Underflow may be triggered by the deletion of an object from a leaf node. The algorithm to handle the underflow is shown in Figure 4. It is adapted from the R*-tree's underflow treatment algorithm, where the main differences lie in steps 3 and 4.

Algorithm *UnderflowTreatment*

Input: Underflowed leaf node L

1. $N = L; S = \emptyset;$
 2. **if** N is root, re-insert all entries in S into the tree and **return**. Notice that if an entry is an index entry, it should be inserted into a proper level;
 3. **if** N is a leaf node, and there exists an outlier object O in N 's parent node which could be pushed down into the entry pointed to N ,
 - (a) Insert O into N and adjust its MBR accordingly;
 - (b) **return**;
 - end if**
 4. **if** N is an index node,
 - (a) **while** $num_of_children(N) < m$ and N contains some outlier object,
 - i. Find an outlier O in N which, if pushed down to a lower level, introduces the least loss;
 - ii. Insert O to the next lower level;
 - iii. Propagate and adjust the subtree rooted by N properly;
 - end while**
 - end if**
 5. **if** N has less than m entries,
 - (a) Remove the index entry E_N in the parent node P ;
 - (b) Add N 's entries to S ;
 - end if**
 6. $N = P$
 7. Repeat step 2.
-

Figure 4: Algorithm *UnderflowTreatment*

In step 3, when a leaf node underflows, instead of re-inserting objects from them immediately, we first try to drag down an outlier object from its parent node. If this is successful, the node does not underflow anymore. Otherwise, the leaf node is removed and the objects from it are re-inserted. This underflow may propagate to the parent node, which is an index node.

Step 4 of the algorithm handles index page underflow. We argue that to re-insert all entries from the underflowing index page can be very expensive, since the page may at the same time be pretty full (containing many outlier objects). To avoid excessive re-insertion, we push down the outlier objects to the next lower level before we re-insert all its entries. In fact, chances are that after we push down some outlier objects, some child page may split and the underflowed page does not underflow any more.

4. QUERYING THE R^0 -tree

In order for commercial systems to adapt their implemented R^* -tree to our R^0 -tree, the system designers want to know that the new index can support all existing queries with equal or better query performance, and the adaption work is not too difficult. In this section, we pick five widely-used spatial queries which are supported by the original R^* -tree, and show how we can adapt from the original algorithms to the R^0 -tree query algorithms without complex transformations. Other spatial query algorithms (e.g. closest pair query [6]) can also be easily adapted in the similar ways. Later in Section 5 we show the experimental results of the five representative queries.

4.1 The Range Query & Aggregation Query

In general, the range query on the R^0 -tree is performed in a top-down recursive manner as the R^* -tree. The difference is that we have to deal with data objects in the index nodes of the R^0 . We descend from the root, and at each node, the outliers are also compared with the query range. In addition, one optimization issue is, if there are many outlier objects, we may pre-sort them according to the X-axis and/or Y-axis, so that at query time, not all outlier objects stored in an index node need to be examined.

To answer the aggregation query, we modify the R^0 -tree a little by storing some aggregation information in each index node. Similar to the aR-tree [15], along with each index entry, we store the aggregate value of all objects in the sub-tree. Then the aggregate query is performed similar to the range query, by aggregating the values of objects found on-the-fly. If we see that the MBR of an index entry is contained by the query range, the value stored at the entry contributes to the query result and there is no need to further browse the sub-tree.

In both the range query and the aggregation query, the R^0 -tree tends to have better performance, since the MBRs in the tree tend to be smaller.

4.2 The Nearest Neighbor Query

There are many research work on the Nearest Neighbor (NN) query. Two representatives are [17] and [9]. In the original version, the entry pointing to the root node is first pushed into a priority queue, ordered by the *MINDIST* (or minimum distance) to the query location. At each step, an entry is popped from the queue and all its child entries are inserted into the queue. The first object popped from the

queue is the nearest neighbor. The adaption is only that, when processing an index node, a child entry can be either an index entry or an object.

The R^0 -tree tends to have better query performance for the following reasons. First, since objects are stored at higher levels the tree, the search may stop prematurely. For instance, if the query result is an outlier object stored in the root node, the search immediately stops. Second, seeing objects early brings the opportunity to prune the priority queue early. That is, the distance from the query point to the object acts as an upper bound to the distance from the query point to the nearest neighbor. And any index entry in the queue, whose *MINDIST* is no smaller than the upper bound, can be pruned. Third, in general, if the query point is inside the MBR of an index entry E , examining $page(E)$ is inevitable. This is because the *MINDIST* is zero. Now that in the R^0 -tree, the MBRs tend to be smaller, the same query point tends to be inside fewer number of MBRs.

4.3 The Skyline Query

Given a set of point objects, the skyline query returns all objects that are not *dominated* by any other objects. An object p_1 *dominates* another object p_2 , if p_1 is at least as good as p_2 in all dimensions and better in at least one dimension. Here, better means smaller. A classic example of the skyline query is to find those hotels that are cheap and near to the beach (Figure 5). Recently an efficient solution to the skyline query is the *branch-and-bound skyline (BBS)* algorithm, proposed by Papadias, *et al.* [16]. The BBS algorithm works as follows. If p_1 dominates p_2 , it must be true that the L_1 distance from p_1 to *origin* is smaller than the L_1 distance from p_2 to *origin*. So the algorithm starts with locating the object p_1 where the L_1 distance to *origin* is the smallest, which must be in the skyline. This can be achieved by using the best-first nearest neighbor search. The difference is that the index/leaf entries in the priority queue are sorted by the minimum L_1 distance to *origin*. Any MBR whose lower-left corner is dominated by p_1 can be pruned. The algorithm continues with the unpruned space, and stops until the priority is empty. Several variations of skyline queries were also discussed in that paper (e.g. ranked skyline query, constrained skyline query and etc), which can be solved by extending the BBS algorithm. In our paper, we focus on the conventional skyline query.

The BBS algorithm could be directly applied on the R^0 -tree. The performance can be improved for the following reasons. In the R^* -tree, the BBS algorithm begins to prune index entries only after it finds in the leaf node the first object of the skyline. However, in the R^0 -tree, some objects are stored at higher levels. It is possible that pruning begins without even accessing the leaf nodes, if objects in the

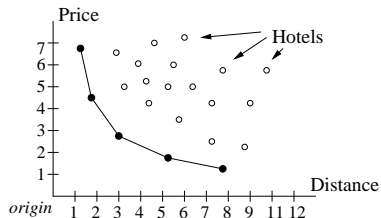


Figure 5: Example of the skyline query.

skyline are found in the index levels. Also, since we define the outlier (point) objects to be located far away from other objects, it is likely that some outlier objects are also in the skyline.

4.4 The Join Query

Both the depth-first R-tree join algorithms [5] and the breadth-first R-tree join algorithm [10] can be extended to the case of using the R^0 -trees. The difference is that in the joining stack we need to maintain pairs of (object O , index entry E) besides pairs of pure index entries. When such a pair is popped from the stack, we should compare O with the child entries in $page(E)$. Since there may have many pairs in the stack corresponding to the same index entry, the global optimization proposed in the breath-first R-tree join [10] could be applied in our R^0 -tree, to avoid multiple loading of the same page.

Again, the R^0 -tree tends to have better query performance since a smaller MBR tends to intersect with less MBRs in the other tree.

5. PERFORMANCE RESULTS

5.1 Experimental Setup

We compare the spatial query performances of the R^0 -tree and the R^* -tree¹. Especially, on the range query, we also compare the improved R^* -tree [20] with them. Both the R^0 -tree and the R^* -trees are implemented in Java. Our experiments are performed using some real datasets, which are acquired from the R-tree-Portal [18]. All our data structures and algorithms are performed on a Dell PC with a 2.66-GHz Pentium 4 processor, running WinXP Professional.

We use various real cartographic datasets in our experiments, which are listed as follows.

1. **NE data:** Containing 123,593 postal addresses (points), which represent three metropolitan areas (New York, Philadelphia and Boston).
2. **US data:** Containing 81,043 railroads (line segments) in the United States. These line segments have small MBRs.
3. **CAMix data:** Containing 62,556 locations and 7,697 railroads (poly-lines) in California. The poly-lines have relatively large MBRs.
4. **CARr data and CARd data:** CARr data contains 7,697 railroads (poly-lines) in California and CARd contains 21,831 roads (line segments) in California.

The first three datasets (NE, US and CAMix data) represent three different kinds of spatial data, i.e. point objects, objects with extents and their mixture. They are used in the range query, the aggregation query, the nearest neighbor query and the skyline query. The last two datasets (CARr and CARd data) are used in spatial join query. As we pointed out in the related work, the promotion-based index is not a balanced tree as shown in our preliminary experiments, we decide not to compare with it.

¹Since we focus on dynamic variations of the R^* -tree (e.g. trees are built by dynamic insertions), we do not compare with some static variations (e.g. the Priority R-tree) that need to know the data in advance.

In our experiments, unless otherwise stated, we choose the page size to be 1KB, which is most appropriate to the size of the data we use. Consequently, each tree node has capacity of 50 entries. The fan-outs of the R^0 -tree and the R^* -tree are both 40% of the node capacity. In our overflow treatment of the R^0 -tree, the breaking point to split an index node is that 90% of the node capacity are index entries. For each index, we use an LRU buffer with capacity of 128 disk pages. In the range query, the aggregation query, the nearest neighbor query and the skyline query, we assume that initially only the root node is in the buffer. In implementing the improved R^* -tree with gain/loss metrics, we set several parameters as follows, which are explained in [20]. In computing the quality, we use $\alpha = 0.5$. We use the greedy algorithm to identify minP-boundary in the reinsertion process, where $\beta = 0.9$. We set p (# objects to be reinserted) to be 30% of the node capacity. Our R^0 -tree incorporates modifications of the improved R^* -tree. Finally, to avoid the randomness, for all queries we report the average results of 100 runs.

This section is organized as follows. Section 5.2 compares the performance of tree construction. Section 5.3 presents the comparisons of the range query. In Section 5.4, we compare the performance of the aggregation query, specially the SUM query. Then in Section 5.5, we compare the performance of the nearest neighbor query. Finally, Section 5.7 presents the comparisons of the spatial join.

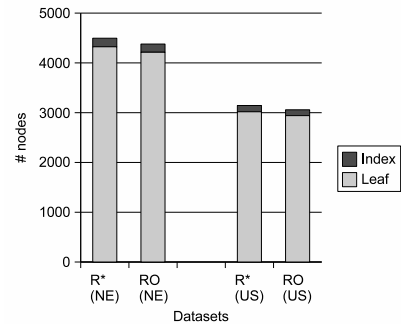


Figure 6: Index size comparison.

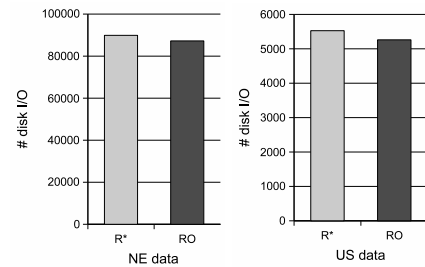


Figure 7: Disk I/O comparison during tree construction.

5.2 Tree Construction

To have an idea of how outlier objects stored in the index levels will affect the trees, we compare the index sizes and the construction cost of the R^0 -tree and the R^* -tree on the NE data and the US data. Figure 6 shows the comparison of

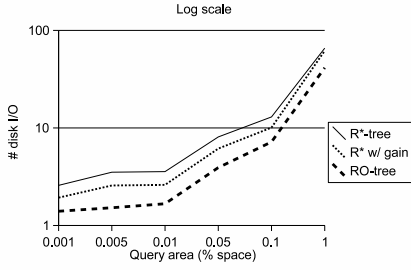


Figure 8: Range query on the NE data by varying the query range (page size = 1K).

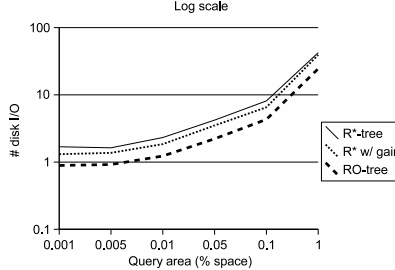


Figure 9: Range query on the US data by varying the query range (page size = 1K).

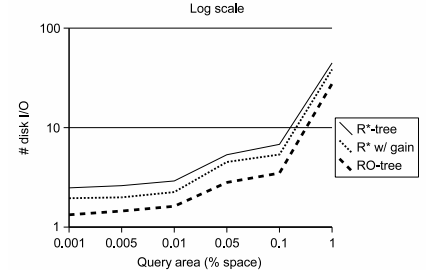


Figure 10: Range query on the CAMix data by varying the query range (page size = 1K).

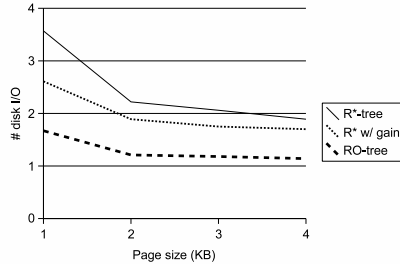


Figure 11: Range query on the NE data by varying the page size (query size = 0.01%).

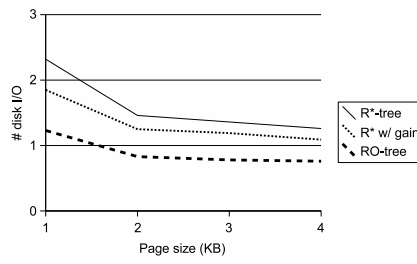


Figure 12: Range query on the US data by varying the page size (query size = 0.01%).

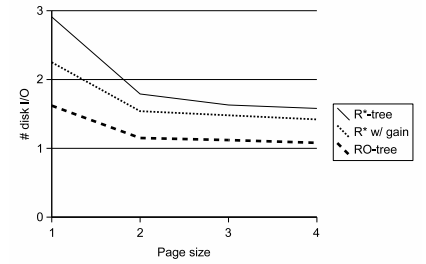


Figure 13: Range query on the CAMix data by varying the page size (query size = 0.01%).

the number of nodes. Since some objects are stored higher in the index levels, the number of leaf nodes and index nodes of the R^0 -tree is less than that of the R^* -tree. Since the disk I/O time dominates the total CPU time, we compare the construction disk I/Os in Figure 7. Notice that since the US data is stored in a special order such that objects close to each other are stored together, loading the US data costs much less I/Os than loading the NE data. From the figure, we conclude that the construction of the R^0 -tree is comparable with that of the R^* -tree. And with less number of re-insertions, the R^0 -tree is slightly better than the R^* -tree.

5.3 The Range Query

In this section, we compare the range query (or the intersection query) performances of the R^* -tree and the improved R^* -tree with gain/loss metrics (denoted as R^* w/ gain) and the R^0 -tree. In this set of experiments, we use the NE data, the US data and the CAMix data. We perform randomly generated range queries by (1) varying the query size from 0.001% to 1% of the whole space, while the page size was 1KB; and (2) varying the page size from 1KB to 4KB, while the query size was 0.01% of the whole space. Again, for each size, we report the average time for 100 random queries. Figure 8 and 11 show the query results on the NE data, Figure 9 and 12 show the results on the US data, and Figure 10 and 13 show the results on the CAMix data.

In all datasets, the R^0 -tree has the most efficient performance among the three structures, up to 48% improvement. With the increase of the query size, more nodes need to be visited regardless of its size and shape, therefore the query

improvement decreases. In Figure 11, 12 and 13, # disk I/O drops because the heights of all trees decrease by one. Our experiments show that when the page size increases, there are fewer index nodes. The total number of objects stored in the index nodes decreases, and the effect of higher level objects reduces. Therefore, the improvement decreases with the increase of the page size. However, the R^0 -tree always outperforms the R^* -tree. It is worth mentioning that the performances of the R^0 -tree on the NE data and the CAMix data are a little better than that on the US data. It is because the NE data consists of three clusters (the metropolitan areas) with many noises, and the CAMix data contains objects with large extents, while US data does not have as many outliers.

5.4 The Aggregation Query

In this section, we compare the aggregation (SUM) query performance of the R^0 -tree and the R^* -tree on the NE data, the US data and the CAMix data. Again, the query ranges are from 0.001% to 1% of the whole space. Figures 14, 15 and 16 show the results on the NE data, the US data and the CAMix data, respectively. We augment all the trees with a *count* in each index entry, so that if an index entry is fully contained in the query, the whole subtree pointed by it does not need to be visited [15].

The R^0 -tree shows up to 49% improvement of the aggregation performance over the R^* -tree. As the outliers are stored at higher levels and the MBRs become smaller, it is likely that a whole subtree will be pruned. Thus, the aggregation query shows more improvement than the range query. Especially, when the query range is large, unlike the range

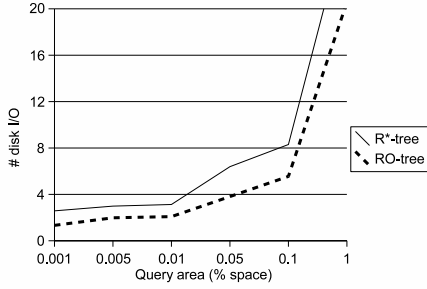


Figure 14: Aggregation query on the NE data.

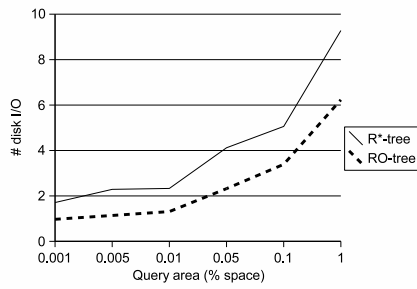


Figure 15: Aggregation query on the US data.

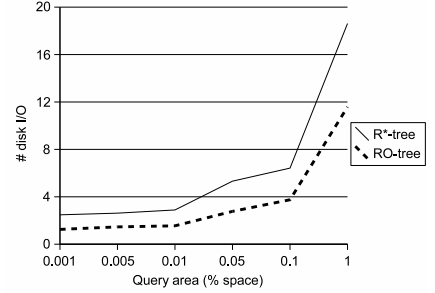


Figure 16: Aggregation query on the CAMix data.

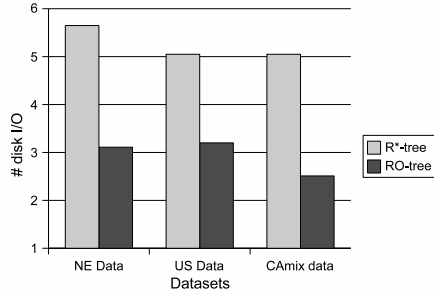


Figure 17: Nearest Neighbor Query.

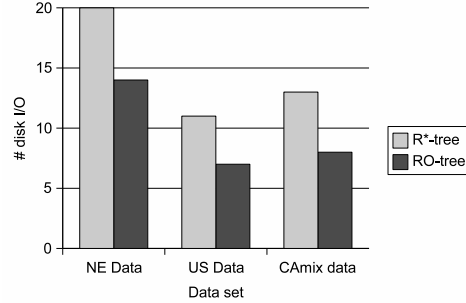


Figure 18: Skyline Query.

query, it is more likely that the query will stop at the index level, and the aggregation query performance shows more improvement than the range query performance.

5.5 The Nearest Neighbor Query

We perform the nearest neighbor query on the NE data, US data and the CAMix data. We pick the query points randomly in the space. Figure 17 shows the results of the query performances.

In all datasets, the R⁰-tree has better performance, up to 51% improvement. This is a larger improvement than other queries. It is expected. The reason why the R⁰-tree is much better in this case was discussed at the end of Section 4.2. In short, a query may stop prematurely if its nearest neighbor is met in an index page (i.e. it is stored as an outlier object), it is possible that pruning is performance earlier, and the query point may be contained in the MBRs of fewer number of tree nodes.

5.6 The Skyline Query

In this section, we compare the skyline query performance of the R⁰-tree and the R*-tree. We implement the BBS algorithm [16] for both tree structures, and the skyline query asks for all objects in the skyline in the NE, US and CAMix data. Our experimental results show that NE data has 10 objects in the final skyline, US data has 14 objects in the skyline, and the CAMix data has 39 objects in the skyline. Figure 18 shows the number of disk I/Os for querying each dataset.

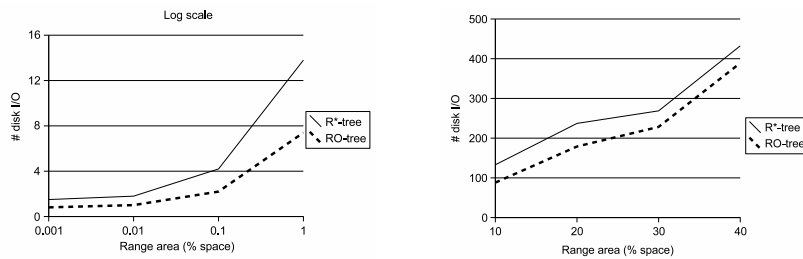
For the skyline query, the R⁰-tree also outperforms the R*-tree, up to 40% better. As we discussed in Section 4.3,

it is likely that some outlier objects are in the final skyline, and the pruning mechanism is likely to start before reaching the leaf nodes.

5.7 The Join Query

In this section, we compare the spatial join performance of the R⁰-tree and the R*-tree. The spatial join is performed between the CArr data and the CArD data. We aim at finding all pairs of objects whose MBRs intersect with each other. Furthermore, since in practice we seldom query the whole space, in our implementation the join operation is associated with a query range, and only objects intersecting the query range are considered in the join. This could be done by performing two range queries, one on each dataset, and then perform a join query on the range query result. However, we implement a better approach, which is a synchronous algorithm integrating the range query algorithm with the join query algorithm. To get an idea why this is better, we point out that an MBR in one dataset, even though intersecting with the query region, may not need to be examined at all by the synchronous approach, if it does not intersect with any MBR from the other dataset. The join range is randomly generated with sizes varying from 0.001% to 40% of the whole space. Because the number of the disk accesses increases dramatically with the increase of the range, we divide the results of the spatial join performance into two parts, which is shown in Figure 19.

With the help of outlier objects stored higher in the index levels, we can avoid many unnecessary joins. If a MBR is not within the join range, it is not examined at all. From the results, we observe the improvement of the spatial join



(a) Join range between 0.001% and 1% (b) Join range between 10% and 40%

Figure 19: Spatial Join between Carr and Card data.

performance up to 46%. Similar to the range query performance, with the increase of the query size, the improvement decreases. Nevertheless, the R⁰-tree still exhibits better performance than the R*-tree as always.

6. CONCLUSIONS

This paper explored the idea of identifying and storing outlier objects at higher levels of the spatial tree index. We proposed the R⁰-tree, which results from applying the above idea to the R*-tree. In order to implement the idea, the R⁰-tree's structure, new update algorithms and query algorithms were presented. Here we considered five most widely used queries. The performance results showed that, compared with the R*-tree, our R⁰-tree has comparable construction cost and index size, but much better query performance. For the range query, aggregation query and join query, the R⁰-tree is about 48% better. For the skyline query, the R⁰-tree is up to 40% better. For the nearest neighbor query, the R⁰-tree is 51% better. Furthermore, the algorithmic changes from the R*-tree is not difficult to apply. Based on our findings, we claim that among all variations of the R-tree, the R⁰-tree is the best up to now. In the future, if one decides to choose a variation of the R-tree to implement, either in commercial systems or for research, the R⁰-tree is strongly recommended.

7. REFERENCES

- [1] N. An, K. Kanth, and S. Ravada. Improving Performance with Bulk-Inserts in Oracle R-Trees. In *VLDB*, 2003.
- [2] L. Arge, M. Berg, H. J. Haverkort, and K. Yi. The Priority R-tree: A Practically Efficient and Worst-case Optimal R-tree. In *ACM SIGMOD*, pages 347–358, 2004.
- [3] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *ACM SIGMOD*, pages 322–331, 1990.
- [4] C. Böhm, S. Berchtold, and D. A. Keim. Searching in High-dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, 33(3):322–373, 2001.
- [5] T. Brinkhoff, H. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins using R-trees. In *ACM SIGMOD*, pages 237–246, 1993.
- [6] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest Pair Queries in Spatial Databases. In *ACM SIGMOD*, pages 189–200, 2000.
- [7] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
- [8] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *ACM SIGMOD*, pages 47–57, 1984.
- [9] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318, 1999.
- [10] Y. Huang, N. Jing, and E. Rundensteiner. Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations. In *VLDB*, pages 396–405, 1997.
- [11] IBM. IBM Informix Spatial DataBlade Module. <http://www-3.ibm.com/software/data/informix/pubs/specsheets/SWSEC27152000D.pdf>.
- [12] K. V. R. Kanth, A. El Abbadi, D. Agrawal, and A. K. Singh. Indexing Non-Uniform Spatial Data. In *Proc. of Int. Database Engineering & Applications Symposium (IDEAS)*, 1997.
- [13] K. V. R. Kanth, S. Ravada, and D. Abugov. Quadtree and R-tree Indexes in Oracle Spatial: A Comparison using GIS Data. In *ACM SIGMOD*, pages 546–557, 2002.
- [14] Oracle. Oracle8 Spatial Cartridge. <http://technet.oracle.com/prod-ucts/oracle8/info/sdods/xsdo7ds.htm>.
- [15] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP Operations in Spatial Data Warehouses. In *Proc. of Symposium on Spatial and Temporal Databases (SSTD)*, pages 443–459, 2001.
- [16] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *ACM SIGMOD*, pages 467–478, 2003.
- [17] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *ACM SIGMOD*, pages 71–79, 1995.
- [18] Y. Theodoridis. The R-tree-portal. <http://www.rtreeportal.org>, 2003.
- [19] J. S. Vitter. External Memory Algorithms and Data Structures. *ACM Computing Surveys*, 33(2):209–271, 2001.
- [20] D. Zhang and T. Xia. A Novel Improvement to the R*-tree Spatial Index using Gain/Loss Metrics. In *ACM Int. Symposium on Advances in Geographic Information Systems (GIS)*, pages 204–213, 2004.