Quadtree and R-tree Indexes in Oracle Spatial: A Comparison using GIS Data

Ravi Kanth V Kothuri
Spatial Technologies
Oracle Corporation
NEDC, Nashua NH 03062.
Ravi.Kothuri@oracle.com

Siva Ravada
Spatial Technologies
Oracle Corporation
NEDC, Nashua NH 03062.
Siva.Ravada@oracle.com

Daniel Abugov
Spatial Technologies
Oracle Corporation
NEDC, Nashua NH 03062.
Daniel.Abugov@oracle.com

ABSTRACT

Spatial indexing has been one of the active focus areas in recent database research. Several variants of Quadtree and R-tree indexes have been proposed in database literature. In this paper, we first describe briefly our implementation of Quadtree and R-tree index structures and related optimizations in Oracle Spatial. We then examine the relative merits of the two structures as implemented in Oracle Spatial and compare their performance for different types of queries and other operations. Finally, we summarize our experiences with these different structures in indexing large GIS datasets in Oracle Spatial.

1. INTRODUCTION

Spatial searching is a fundamental primitive in non-traditional databases such as GIS, CAD/CAM and multi-media applications. With the rapid proliferation of these databases in the past decade, extensive research has been conducted on the design of efficient data structures to enable fast spatial searching. Several data structures have been developed in this context. These include Quadtrees [26, 27, 31], R-trees [13, 28], hB-trees [19], TV-trees [18], SS-trees [33], and SR-trees [15]. Subsequent research has improved these basic structures further by proposing new techniques for query processing [4, 5, 9, 14, 16, 20, 21, 22, 30], faster and better index creation [12, 17, 29, 32], and better split-strategies in dynamic updates [2, 3]. These techniques are especially effective for low-dimensional spatial data such as those in GIS and CAD/CAM applications. Commercial database vendors like IBM, and Oracle have also started implementing these indexing techniques to cater to the large and diverse GIS and CAD/CAM application markets. In this paper we examine different approaches supported by Oracle for indexing such low-dimensional data and present our experiences with their relative performance on large GIS datasets.

For indexing low-dimensional spatial data, Oracle Spatial allows users to choose between one of two spatial indexes: a (Linear) Quadtree [27] or an R-tree [13, 28, 2, 3, 17, 11]. These indexes are implemented using the extensible indexing framework of Oracle [24] and incorporate and enhance some of the best proposals from existing spatial indexing research. The Linear Quadtree (or Quadtree for short) computes tile approximations for geometries and uses existing B-tree indexes for performing spatial search and other DML operations. This approach results in simpler index creation, faster updates and inheriting of built-in B-tree concurrency control protocols. The R-tree is implemented logically as a tree and physically using tables inside the database and the search involves recursive SQL for traversing the tree from root to relevant leaves. This approach may be more efficient for queries due to better preservation of spatial proximity but may be slow in updates or index creation and implements its own concurrency protocols on top of Oracle's table-level concurrency mechanisms. In the following sections, we describe these implementations in more detail and compare their relative behavior primarily with respect to query performance (since updates are not high in most GIS applications). Such a comparison of two popular indexing strategies, *inside a commercial database*, provides useful insight into the following aspects of indexing: (1) the strengths and weaknesses of the Quadtree and the R-tree implementations for indexing spatial data, and (2) whether a

ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA

"true" tree-based structure in a database could be as efficient as structures implemented using built-in indexes like B-trees or hash tables.

The rest of the paper is organized as follows: Section 2 describes how Oracle Spatial supports spatial data and presents the model for query processing using either of its two indexes, Quadtrees and R-trees. Sections 3 and 4 present implementation details of Quadtrees and R-trees in Oracle Spatial. Section 5 describes the testbed and compares the two index structures. The final section summarizes the results.

2. ORACLE SPATIAL

Oracle Spatial models 2-4 dimensional spatial data using an *sdo_geometry* data type. For the 2-dimensional case, this data type models all the spatial data types defined by the Open GIS Consortium (OGC) and caters to most data occurring in GIS, CAD/CAM applications. Supported spatial data includes simple primitive elements such as points, lines, curves, polygons (with and without holes), and complex elements that are made up of a combination of primitive elements. The sdo_geometry data type is implemented as an Oracle object datatype. This approach extends all the benefits of Oracle's object-relational database technology including replication to spatial data.

2.1 Indexing Framework

Quadtree and R-tree indexes on spatial data are implemented using the extensible indexing framework of Oracle [6, 24]. This framework allows for the creation of new domain-specific indexes and associated query operators and provides for the integration of user-specified query, update and index creation routines inside Oracle server. Oracle Spatial supports a "spatial_index" indextype for indexing spatial data. Quadtree and R-tree indexes are supported as part of this "spatial_index" indextype. Since these indexes are implemented as part of the extensible indexing framework, spatial indexes can be easily created on "sdo_geometry" columns of database tables using an extended SQL syntax. As part of such index creation, the corresponding spatial index creation routines are executed and the constructed spatial index is stored in the database as a "spatial_index" table. The index table stores index information such as R-tree nodes in the case of R-trees and Quadtree tiles in the case of Quadtrees. The metadata for the entire index is stored as a row in a separate metadata table. This metadata includes the name of the index table storing the index, dimensionality, root pointer fanout parameters for an R-tree and the tiling level parameter for a Quadtree index.

In addition to SQL-level index creation, inserts and updates to database tables that have a spatial index also automatically trigger an update of the corresponding spatial indexes. In addition to these advantages, extensible indexing also ensures statement or session-level concurrency and table-level recovery.

To query the constructed spatial indexes, new predicates, referred to as *operators*, are defined. These operators can be included in the "where" clause of a SQL statement to select data that satisfy a specified query criterion with respect to a specified query window. Such operators are executed using index-associated procedures for query processing and allow for incremental processing of queries (see [24, 6] for more details). In what follows, we describe the query operators supported by Oracle Spatial. These operators have identical semantics irrespective of whether they are executed using underlying Quadtree or R-tree indexes.

2.2 Queries on Spatial Indexes

Oracle Spatial provides an operator called *sdo_relate* that identifies data geometries that interact with the query geometry for a specified criterion, in Oracle Spatial terms, a *mask*. These masks or criteria are based on the 9-intersection model of [7, 8] and the *sdo_relate* operator is equivalent to the *ST_relate* operator of SQL/MM.[1]

- anyinteract: identify data geometries that intersect the specified query geometry
- inside: identify data geometries that are "completely inside" the query geometry
- coveredby: identify data geometries that "touch" on at least one boundary and are inside the query geometry otherwise
- contains: reverse of inside
- covers: reverse of coveredby
- touch: identify geometries whose boundary "touches" the boundary of the query geometry but disjoint otherwise
- equal: identify geometries that are exactly the same as the query geometry
- overlapbdyintersect: identify geometries that overlap each other and boundaries intersect.
- overlapbdydisjoint: identify geometries that overlap with the query geometry but the boundaries are disjoint.

[1]Most SQL/MM suggested methods are available in one form or the other in Oracle Spatial.

Since indexes for geometries work with approximations, Oracle Spatial provides the *sdo_filter* operator which returns results directly from the index without performing geometry-level comparisons. This operator gives a fast but approximate answer for geometries intersecting a query window. In addition to these operators, Oracle spatial also provides the following set of metric (or distance) operators:

- sdo_within_distance (or within-distance) queries: identify geometries that are within a specified distance from the query geometry
- sdo_nn (nearest-neighbor) queries: identify the k nearest neighbors [14, 25] for a specified query geometry.

In this paper, we focus mostly on *sdo_relate* and *sdo_within_distance* operators since they are the most frequently used operators in GIS applications. In what follows, we describe the general methodology for processing such queries in Oracle Spatial. The techniques described are applied for both types of indexes, Quadtrees or R-trees.

2.3 Query Processing in Oracle Spatial

To efficiently process the above queries on complex spatial data, Oracle Spatial uses a multi-stage query model as shown in Figure 1(a). In the first stage, referred to as the *primary filter*, the spatial index is used for query filtering. Candidate geometries that may satisfy a given query criterion are first identified in this stage with the help of the *exterior* approximations in the spatial index. In the case of a Quadtree, Quadtree tiles [27] are used as the exterior approximations and in the case of an R-tree, minimum bounding rectangles (MBRs) are used. In the intermediate stage, candidate geometries are compared with the query using the *interior* approximations of both query and candidate geometries and some candidate geometries are either accepted or eliminated based on the query criterion. The rest of the geometries whose interaction is not determined in the intermediate filter are then passed through to the final stage, referred to as the *secondary* filter, and the exact result set is determined and returned to the user. Whereas the secondary filter uses computational geometry algorithms to determine interaction between query and candidate geometries, the primary and intermediate filters use exterior and interior approximations of query and data geometries (from the index). In what follows, we illustrate the processing of these two filters with a simple example. More detailed explanation in the context of specific indexes is discussed in the subsequent sections.

Next, we illustrate the processing of primary and intermediate filters for an R-tree index. Similar processing is performed for Quadtrees. Consider the query geometry Q and data geometry G of Figure 1(b). We assume the query is finding all geometries that intersect the query geometry (window query with anyinteract (intersection)-type of interaction). The exterior approximations, which are MBRs for *Oracle Spatial* R-trees [24], are shown as dashed boxes and the interior rectangles are shown as solid boxes for the geometries. The primary filter determines that query Q may interact with geometry G using their exterior approximations, the MBRs, which happen to intersect each other. Next, the intermediate filter is used which determines that the exterior of the geometry G is *inside* the interior of the query Q. It then determines that the geometry satisfies the *anyinteract* criterion of the query and includes the geometry directly in the result set thus bypassing the secondary filter. Since comparison of two geometries (query geometry and a candidate data geometry) in the secondary filter is expensive due to both loading of geometry data and the expensive computational geometry required to determine the relationship between the two geometries, elimination/acceptance in the intermediate filter reduces query processing time considerably. Experimental results validating this theory are presented in [23] for the case of R-trees. Similar results are also reported for Quadtrees in [1].

3. INDEX-BASED FILTERING IN ORACLE SPATIAL

In this section, we first describe the implementation of primary and intermediate filters using Quadtree and R-tree indexes. Then, we describe the implementation of intermediate filter for each of these indexes.

3.1 Quadtree as the Primary Filter

Oracle Spatial uses the following strategy in Quadtree indexing: a user-specified tiling level is chosen and each data geometry is approximated by a minimal set of covering tiles. Each tile is associated with a tile-code obtained by using z-ordering of all tiles at the specified level. During tessellation, the tiles for the geometry are divided into *interior* and *boundary* based on whether or not they are completely interior to the geometry. All the covering tiles for a geometry are then inserted into the spatial_index table along with the rowid of the geometry. As a result of this approach, a geometry may have multiple rows in the spatial_index table where each row stores a (different) tile-code, interior/boundary status, and the rowid of the geometry. A B-tree index on the tile_code, rowid, status is then constructed for speeding up queries.

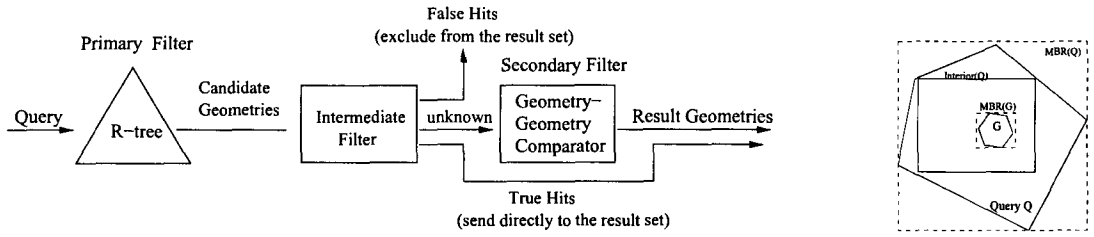

Figure 1: (a) Multi-stage Query Processing in Oracle Spatial. (b) Interior approximations for geometries.

At query time, the query geometry is likewise tessellated into a set of tiles. Using each tile of the query geometry, the rowids of all geometries whose tiles match the query tiles are identified by a tile-code join using the B-tree index on the tile-codes.

The above approach also makes updates much easier to handle. When a new geometry is inserted into the dataset, it is tessellated into the user-specified-level tiles and the tiles are inserted into the spatial_index table. The B-tree index is automatically updated to include the tiles for the new geometry. Likewise, when a geometry is deleted the corresponding rows are deleted from the spatial_index table and the B-tree index is once again internally updated by the server. In spite of these advantages, Quadtree has one drawback: the need to choose an appropriate tiling level for tesselating the data and query geometries. Much experimentation with different levels is needed in order to optimize performance for a specific dataset.

3.2 R-tree as the Primary Filter

In Oracle Spatial, R-trees maintain their logical tree structure and are implemented as a table where each node of the R-tree corresponds to a row in the table and a child pointer in the R-tree corresponds to the rowid of child row in the same table. The root rowid (pointer) of the R-tree is stored in the metadata for the index and allows navigation from the root of the R-tree to the leaf nodes. Leaf nodes of the R-tree store one MBR for each data geometry along with the geometry rowid. Queries and updates obtain the root of the R-tree and navigate down the tree to the leaves. More details on this implementation can be obtained from [24]. Note that each time an R-tree node is visited, the corresponding row from the spatial_index is selected using a SQL statement internally. This means query and update processing in R-trees involves processing of more recursive SQL statements than in the case of Quadtrees. As a result, some DML operations especially updates are likely to be more costly.

The algorithms used for R-trees in Oracle Spatial involve scalable adaptations of existing research solutions and a set of in-house optimizations and enhancements. The details are not in the scope of this paper. Major optimizations such as the interior approximations for intermediate filter are described in the next subsection.

3.3 Intermediate Filter in Oracle Spatial

In the primary filter, the spatial index uses the exterior approximations of query and candidate data geometries to determine whether or not they interact. If they do interact, then the query and the geometry along with any interior approximations are passed on to the intermediate filter. The intermediate filter takes the query q, and a candidate geometry g and returns *true* if the geometry is to be included in the result set, *false* if not, and *unknown* if it cannot determine the relationship, in which case the query-candidate pair is passed to the secondary-filter to determine the exact relationship. These paths from the intermediate filter are shown with *unknown, true, false* labels in Figure 1(a). To determine these relationships, the intermediate filter uses the interiors of the query and the candidate geometries (passed on from the primary filter). In the case of quadtrees, the interior and the boundary tiles of the candidate geometries are compared with the interior and the boundary tiles of query geometry. In the case of R-trees, only the interior of the query geometry is used and the interiors of the candidate data geometries are not computed. The operational behavior of the intermediate filter for different queries for both Quadtree and R-tree indexes are described in [23].

3.4 Interior Approximations in Quadtrees

In the case of Quadtrees, interior tiles for data geometries are identified at index creation time. The tiles for each geometry are labeled with a "status" indicating whether they are interior tiles or boundary tiles. If any part of the boundary of a geometry touches a tile, the status is set to "boundary". Otherwise, if the tile is completely inside the geometry, the status is set to "interior". These status labels along with the tile-

codes are compared with those of the query tiles at query time to implement the intermediate filtering as described above. Note that such labeling of the tiles that cover query and data geometries imposes very little overhead (one byte per row) on storage space, and does not effect query or update performance.

3.5 Interior Approximations in R-trees

In the case of R-trees, we decided to compute interior approximations for query geometries only and use them in the intermediate filter. This approach alleviates the need for changing existing indexes (only the query behavior will change and not the physical R-tree storage), and avoids any ill-effects of decreasing the fanout for a fixed block-size. In addition, since data geometries are generally much smaller than query geometries, the loss of any improvements from using the data interiors will be relatively small compared to the gains of using the query interior.

R-trees take a two-pronged approach to finding interior approximations. For convex geometries, they divide the geometry into 4 pieces using the x- or the y-extent (whichever is larger) and compute the largest inscribed rectangle for each of these pieces using the technique of [10]. Several interesting properties from [23] show that the interior MBRs could be used as one single contiguous geometry.

The above approach for finding interiors of convex polygons cannot be effectively extended to concave geometries as described in [23]. For concave geometries, an alternate technique for finding the interior approximations for concave query windows is used. The idea is to first recursively divide the MBR of the concave geometry into quadrants 4 times i.e., perform a level-4 tiling using the MBR as the tiling domain. Then among the tiles that cover the geometry tiles that are interior to the geometry are identified. Candidate MBRs are also tiled and the tiles for the candidate MBR are searched among the interior tiles for the concave query window. Recent results [23] show that this two-pronged approach for implementing the intermediate filter in R-trees achieves substantial improvements in query performance. As will be seen in the experiments, the interior area captured using this approach could be larger than that for some quadtree indexes constructed using a tiling level of 14 since the tiling domain in that case is the entire data space whereas the tiling domain for the R-tree is just the MBR of the concave geometry.

3.6 Interior Approximations Summary

Table 1 summarizes the use of interior approximations for Quadtrees and R-trees. As described, Quadtrees use interior tiles for both query and data geometries. The tiles for each data geometry in the spatial_index table are labeled *interior* or *boundary* based on whether or not they are interior to the data geometry. Likewise, the interior tiles to a query are also identified. Together, the interiors of the query and candidate geometries are used in the intermediate filter to bypass the secondary filter whenever possible (as described in Section 3.1). In contrast, R-trees only use interiors of the queries. As a consequence, bypassing the secondary filter is not possible in all cases but will still be effective in most cases (as described in Section 3.5).

	Quadtree	R-tree
Data	Interior tiles	None
Convex Query	Interior tiles	Interior MBRs
Concave Query	Interior tiles	Interior tiles

Table 1: Summary of the use of interior approximations for Quadtree and R-trees

With all these optimizations incorporated into the indexes, we compare the performance of both Quadtrees and R-trees for different queries.

4. EXPERIMENTS

In this section, we describe experimental results that compare the performance of Quadtrees and R-trees. We used two datasets: the US Blockgroup (USBG) dataset consisting of about 230K arbitrarily-shaped polygon geometries, and the US Business Area (ABI) dataset consisting of over 10M data points.

For both datasets, Quadtree and R-tree indexes are created. In order to optimize performance, Quadtrees need to be fine-tuned by choosing an appropriate tiling level. Unlike Quadtrees, no specific tuning is required for R-trees. In the next subsection, we demonstrate the dependence of Quadtree performance on the tiling level using a subset of USBG data.

4.1 Impact of Tiling level on the Performance of Quadtrees

To determine the impact of tiling level on the performance of Quadtrees, we choose a subset of USBG dataset. This subset consists of all geometries in 100-mile radius from the center of Manhattan, NY and contains 23982 geometries (i.e., 10% of the original size of the USBG dataset). Table 2 shows the index creation performance as the tiling level increases. For small tiling levels of 2 to 8, each geometry fits completely inside a single tile and the index creation times

are small. As tiling level increases from 12 to 14 and then from 14 to 16, the number of tiles per geometry increase by a factor of 3 and 7. Although this means the storage and the index creation costs increase as the tiling level increases (as shown in Table 2), the benefit is that *finer* and *better* tile approximations are obtained for geometries. As a result, fewer and fewer (tile approximations of) data geometries will interact with (the tile approximations of) a query geometry improving query performance of the quadtree index. However, increasing the tiling level beyond a certain level makes the tile approximations too fine and too many: potentially affecting the query times adversely. Finding the right tiling level is critical to the performance of quadtree. This phenomenon is illustrated in Figure 2 using a 5-mile and a 10-mile radius window around the Manhattan center. Figure 2 (a) shows the results for a 5-mile-radius query window. As the tiling level increases, the primary filter starts returning fewer and fewer results due to better tile-approximations of the data/query geometries: at level 2, the tiles are too coarse and hence all 23982 geometries are returned whereas at level 14, only 2216 geometries are returned. Due to the decrease in the number of returned geoemtries the query times decrease substantially. However, as the tiling level increases to 16 and then to 18, the number of tile approximations increase substantially (as shown in Table 2) and the associated searching costs on the large number of tiles outweigh any further gains from better tile approximations (and reduced data-query interactions). The same phenomenon also occurs for the secondary filter query wherein the resulting candidates from the index (primary filter) are checked with the query geometry for intersection. Similar results are also obtained for a 10-mile-radius query window. In all these results, a tiling level of 14 obtains best query performance for the subset of USBG data. This experiment illustrates that picking the right tiling level is critical to the performance of a quadtree.

In the next subsection, we compare the performance of Quadtree and R-tree indexes on the USBG and the ABI datasets.

4.2 Quadtree and R-tree Comparison

For the ABI dataset, the Quadtree is tessellated at level 18 and for the USBG dataset it is tessellated at level 14. These tiling levels for the Quadtree are chosen after much experimentation to optimize the overall query performance. In contrast, R-tree requires no such fine-tuning. We conducted the experiments using Oracle Release 9.2.0 running on a Sun 400MHz machine with 1GB memory (note: the tests do not need so much memory) First, we present the times for index creation and index updates for both indexes and then describe the results for queries.

Tiling Level	Avg. # of tiles per geometry	Indexing time(s)
2	1	21
4	1	21
6	1.01	21
8	1.03	21
10	1.16	24
12	1.59	26
14	4.35	46
16	29.81	259
18	359.01	3031

Table 2: As tiling level increases, number of tiles per geometry increase as well as the indexing time.

4.2.1 Creation and Update times

In Table 3, we describe the time for creation of these indexes. The table indicates that the creation time for the Quadtree index on the ABI dataset is one-fourth of that for the R-tree. This is because the time for clustering in the R-tree for a large dataset of 10M is relatively expensive. The multiple SQL statements for selecting subsets of the dataset and writing R-tree nodes one by one takes a substantial portion of this creation time. In the case of the Quadtree, the cost is considerably less since tessellation of each data geometry into a single tile (ABI data is point data), insertion into the spatial_index table, and creation of the B-tree index on the spatial_index table take much less time. The next row in table 1 indicates that the Quadtree creation time for USBG dataset is several times higher than the R-tree creation time. This is because tessellation of each block group geometry into level-14 tiles is quite expensive and dominates the overall creation time.

The table also indicates the time for inserting rows into the Blockgroup indexes. We insert three sets of geometries: The first set consists of 10000 simple point data obtained from the ABI dataset. The second set consists of 20000 simple polygon geometries obtained from the USBG dataset (New York region) and the third set consists of 10000 complex polygon geometries also obtained from the USBG data (Alaska region). This is to examine the variation of the quadtree times with the complexity of the geometries. For insertion of point data, the Quadtree is nearly twice as fast as the R-tree. Since tessellation of points is very fast, all the Quadtree needs to do is insert the tile into the spatial_index table and have the associated B-tree updated. For these inserts, the quadtree inherits the relative fastness of B-tree updates and gains from being implemented on top of built-in indexes. Similar

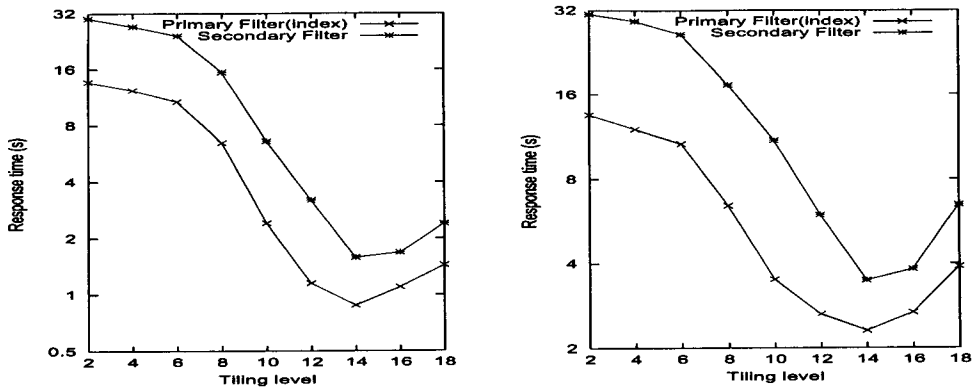

Figure 2: Anyinteract Query Performance for different tiling levels for (a) 5-mile radius, and (b) 10-mile radius windows: Both primary and secondary filter performance decreases as tiling level increases from 2 to 14 and starts to increase again beyond tiling level 14. The initial decrease is due to better approximations in the index and fewer resulting candidates from index. The subsequent increase is due to increase in the number of tiles per geometry and associated searching costs.

results are also reported for 20000 *small*-polygon inserts. However, when inserting 10000 *large* and *complex* polygons from the USBG data, the tessellation time dominates for the Quadtree and it is several times slower than the R-trees. Also note that the insertion times for R-tree increase almost linearly as the dataset is increased from 10000 to 20000 and is independent of the complexity and sizes of the polygons.

Performance Criterion	Quadtree	R-tree
ABI Index Creation	4203s	20406s
USBG Index Creation	8330s	454s
10K-point inserts (ABI)	67s	131s
20K-small-polygon inserts (USBG)	183s	330s
10K-large-polygon inserts (USBG)	3600s	166s
Storage(ABI)	909MB	824MB
Storage(USBG)	728MB	24MB

Table 3: Comparison of Quadtree and R-tree create and update times and storage characteristics: Quad-tree is faster for ABI (point) data in index creation/update. R-tree is faster for large-polygon USBG data. R-tree needs less space in both cases.

The last two rows of Table 3 present the storage space consumed by the Quad-tree and R-tree indexes. For the ABI (point) data, both indexes require almost the same amount of memory whereas for the USBG (polygon) data, R-tree storage consumption is $1/30^{th}$ of that for the Quadtree. This is because the Quadtree is tiled at level 14 to optimize query performance and this generates many tiles for each data geometry consuming a lot of storage space.

4.2.2 Queries

Next, we examine the performance of queries on the two indexes for the two datasets. We consider two types of queries: window queries and within_distance queries (R-trees are much faster than "Linear" Quadtrees for nearest-neighbor queries and are not presented here). For the window queries, we first use the 3230 counties in the United States as a set of concave queries. Then, we generate convex windows using some of the most-densely populated areas and randomly generated query center points within the areas. For instance, for the ABI dataset we used the New York Manhattan center as one query center. Using such query centers, we generated a convex query window of different radii from 0.25 miles width to 100 miles width. For within-distance queries, we use similar queries by specifying the center and a distance of 0.25 to 100 mile radius. All queries are of the form:

```
select geometry from geom_tab
where sdo_<operator>(geometry,
<query_geom>, ...)='TRUE'
```

Note that the queries retrieve geometries as opposed to doing a "select count(*)". We have chosen to use these queries since this is how the GIS and CAD/CAM applications use Oracle Spatial in a realistic scenario. As a consequence, with the increase in the number

of result geometries, the query time grows and the geometry-retrieval time may dominate the actual index response time. We will come across this phenomena for large queries that retrieve a lot of geometries.

Table 4 presents the average query response times for both Quadtree and R-tree indexes on the USBG dataset using the 3230 county polygons as query windows. We observe that R-tree performs better than the Quadtree for all masks. For anyinteract, R-tree is faster by 35%; for inside by 65%. This is because the interior-tile optimization performed in R-trees for the (concave) county queries is more effective than that for the Quadtree at level-14 for USBG. Specifically, the interior area captured by the R-tree is more than that in the Quadtree. By increasing the tiling level to more than 14 for the Quadtree, this interior area may improve but that will result in larger storage requirement for the Quadtree index and adversely affects the query performance. R-tree is faster by 90% for most of the other masks like contains, equal, and covers. This is because for these masks, R-tree first compares the "single" MBRs of query and data geometries and eliminates most secondary filters. For example, for a contains mask, the query MBR is checked for containment in the data MBR (which does not happen in most cases) in addition to regular intersection. Such optimizations are not easily extended to Quadtrees since the exterior of the query/data is split over multiple tiles. For overlap masks ("overlapbdydisjoint" and "overlapbdyintersect"), both R-trees and quadtrees perform nearly the same. Similar results are also obtained for the ABI dataset. Next, we examine the performance of the two structures for convex query windows.

Query mask	Quadtree(s)	Rtree(s)
anyinteract	0.81	0.49
inside	0.80	0.28
contains	0.85	0.04
touch	1.52	1.13
coveredby	0.88	0.66
covers	0.44	0.05
equal	1.53	0.04
overlapbdydisjoint	1.77	1.41
overlapbdyintersect	1.53	1.41

Table 4: Comparison of Average Query times for Quadtree and R-tree indexes on USBG data: Queries are from counties dataset. R-tree is faster for all masks — anyinteract, R-tree faster by 35%; inside by 65%.

Figure 3 (a) shows the performance of R-tree and Quadtree indexes for the most common type: "anyinteract" queries on the ABI dataset. The x-axis plots the query radius in miles and the y-axis the response time for each index. Both axes are on a logarithmic scale. As the query radius increases, the response times also increase. R-tree consistently outperforms Quadtree for small-medium radii: for small radii of 0.25 miles, R-trees are 30% faster. As the radius increases to 100 miles, the number of retrieved geometries increases (to order of .5M) and the geometry-retrieval cost dominates the index cost. In such cases, both indexes perform the same. Figure 3 (b) shows similar results for the USBG dataset for small query radii. R-trees dominate quadtrees for small to medium query radii.

Figure 4 compares the performance of R-tree and Quadtree for the next most popular query: the inside query. Figure 4(a) plots the results for the ABI dataset and Figure 4(b) plots the results for the USBG dataset. Once again, the x-axis plots the query radius in miles and the y-axis the response time for each index. As the query radius increases, the response times also increase. R-tree consistently outperforms Quadtree for all radii: for small radii of 0.25 miles, R-trees are nearly 3-4 times faster than Quadtree; whereas for large radii of 100 miles R-trees are 2 times faster than Quadtrees.

In Figures 3 and 4, R-trees obtain better performance due to capturing a lot of interior using interior rectangles/tiles. However, in "touch"-type of queries, the boundaries of query and data items are checked for touch-type of interaction. In this case, the boundary area for an R-tree query is the MBR approximation minus the interior approximations of R-tree. This boundary for an R-tree covers more area than the "boundary" tiles of a Quadtree (as the MBR is a coarser approximation than quadtree tiles). As a result, the number of data exclusively intersecting the boundary area is more with an R-tree than with a Quadtree. This is especially evident (1) for large query windows, and (2) more predominant for point layers (since more points can fit in the boundary area than non-point data). Since the number of candidate data resulting from the index determines the number of secondary filter comparisons and the resulting query performance, R-tree performance could be worse than that of Quadtree for these queries. Figure 5 (a) and (b) illustrates this for the ABI dataset (a point dataset). Figure 5(a) plots the touch-query response time for both R-tree and Quadtree whereas Figure 5(b) plots the number of candidates passed on to the secondary filter from the primary filter (spatial index). For small query windows the R-tree is faster but as the query size increases, the boundary tiles of the query window in a quadtree intersect fewer data points in contrast to the R-tree. As shown in Figure 6(a), for non-point data of the USBG dataset this

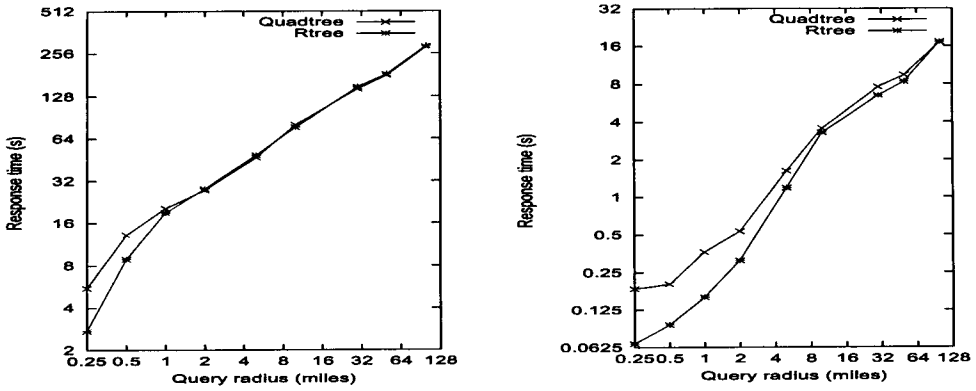

Figure 3: Performance for Anyinteract queries: (a) ABI dataset and (b) USBG dataset.

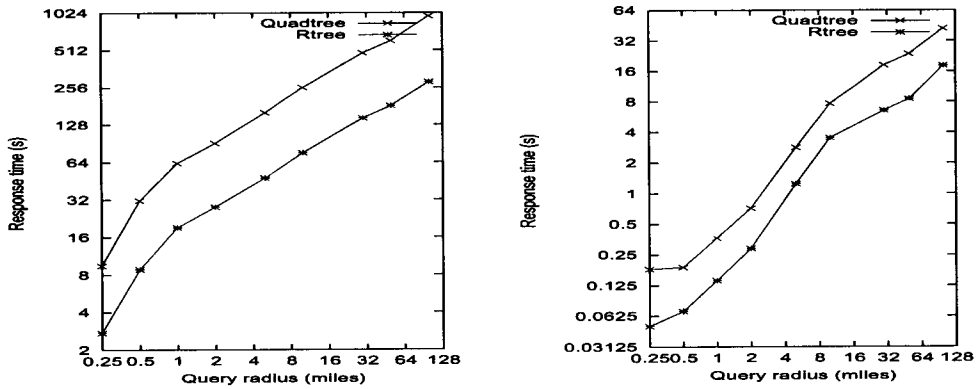

Figure 4: Performance for Inside queries: (a) ABI dataset and (b) USBG dataset.

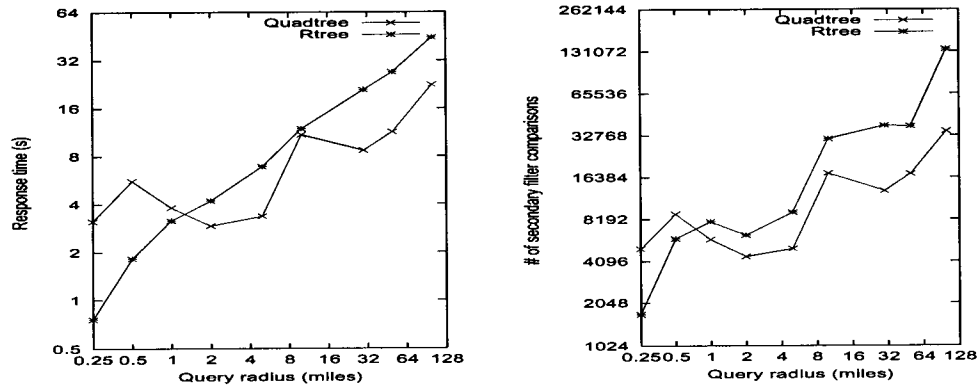

Figure 5: Performance for Touch queries on ABI dataset: (a) Query times, and (b) Number of secondary filter comparisons. Quadtrees are faster because they pass in fewer candidates to secondary filter. This is because the "boundary" tiles of quadtrees are more effective in filtering candidate point data than the "boundary" area of the query in R-tree (query MBR minus interior approximations).

phenomena does not occur until the query window becomes relatively much larger than the data, i.e, for large query windows. Similar elimination strategy also applies to "coveredby" and "overlap" masks. These results for the USBG dataset are shown in Figures 6(b) and 7. These masks are not applicable for point data (all "coveredby" and "overlap" mask queries on ABI dataset return 0 results right away). To summarize the results for "touch", "overlap", and "coveredby" queries that use boundary-area-based filtering, R-trees could be slower than Quadtrees for touch-type queries on point data layers. For non-point data, R-trees are faster for most query sizes.

Next we present the results for other masks using the USBG dataset. Figure 8 compares R-tree and Quadtree for "contains" and "covers"-type of queries. Since R-trees have only one exterior approximation for both query and data geometries, optimizations such as "traverse an index path only if the index/geometry MBR contains or covers the query MBR". Such an optimization is not easily enforceable in Quadtrees. As a result, the performance of Quadtrees degrades as the query radius increases although at most 2 objects are retrieved in the query. In contrast, R-tree response times stay flat with the increase in query radius for both "contains" as well as "covers"-type of queries. Similar results are also obtained for "equals" masks in Figure 8(c).

Next, we present the results for within_distance queries for USBG and ABI datasets. Figure 9(a) plots the results for the ABI dataset and Figure 9(b) plots the results for the USBG dataset. Once again, R-tree consistently outperforms Quadtree for all radii: for small radii of 0.25 miles, R-trees are nearly 3-4 times faster than Quadtree; whereas for large radii of 100 miles R-trees are 2 times faster than Quadtrees. This is because Quadtrees have an additional cost of creating buffers around the query center and tiling it whereas R-trees internally use distance-based computation between query center and index/data geometries to avoid expensive buffering.

5. CONCLUSIONS

In this paper, we compared Quadtree and R-tree indexes supported in Oracle Spatial. We briefly described their implementations along with some optimizations. We evaluated their relative performance on large GIS datasets using different query, insert, and index-creation operations and compared their storage requirement. For these datasets, R-trees consistently outperformed Quadtrees by 2-3 times for upto 10-mile radius windows for almost all query types. For larger query windows, the performance gap between the two indexes decreased and the gap even reversed for some masks such as *touch, overlapbdyintersect* and *overlapbdydisjoint* where Quadtrees performed better due to finer approximation of "boundary" area in a query window. For distance queries such as within-distance (and nearest-neighbor), R-trees consistently outperformed Quadtrees by a factor of 3. Update times for R-trees are almost linear in the number of updates, whereas for quadtrees they depend on the size and complexity of the geometry. Storage requirements for a Quadtree are nearly the same as those for R-trees when the data is *points* and more otherwise. In short, a quadtree could be recommended for update-intensive applications using simple polygon geometries, high concurrency update databases, or when specialized masks such as *touch* are frequently used in queries. However, users have to fine-tune the tiling level to obtain best performance. R-trees, which do not require any such tuning, could be used in all other cases to obtain nearly equivalent or better performance.

6. REFERENCES

[1] W. M. Badaway and W. Aref. On local heuristics to speed up polygon-polygon intersection tests. In *Proceedings of ACM GIS International Conference*, pages 97–102, 1999.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R* tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, 1990.

[3] S. Berchtold, D. A. Keim, and H. P. Kreigel. The X-tree: An index structure for high dimensional data. *Procóf the Int. Conf. on Very Large Data Bases*, 1996.

[4] S. Berchtold, D. A. Keim, H.-P. Kriegel, and T. Seidl. A new technique for nearest neighbor search in high-dimensional space. *IEEE Trans. on Knowledge and Data Engineering*, 12(1):45–57, 2000.

[5] T. Brinkhoff, H. Horn, H. P. Kriegel, and R. Schneider. A storage and access architecture for efficient query processing in spatial database systems. In *Symposium on Large Spatial Databases (SSD'93), LNCS 692*, 1993.

[6] S. Defazio, A. Daoud, L. A. Smith, and J. Srinivasan. Integrating ir and rdbms using cooperative indexing. In *Proc. of ACM SIGIR Conf. on Information Retrieval*, pages 84–92, 1995.

[7] M. J. Egenhofer. Reasoning aobout binary topological relations. In *Symposium on Spatial Databases*, pages 271–289, 1991.

[8] M. J. Egenhofer, A. U. Frank, and J. P. Jackson. A topological data model for spatial databases. In *Symposium on Spatial Databases (SSD)*, pages 271–289, 1989.

[9] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. E. Abbadi. Approximate nearest neighbor searching in multimedia databases. In *Proc. Int. Conf. on Data Engineering*, pages 503–511, 2001.

[10] P. Fischer and K. U. Hoffgen. Computing a maximum axis-aligned rectangle in a convex polygon. In *Information Processing Letters, 51*, pages 189–194, 1994.

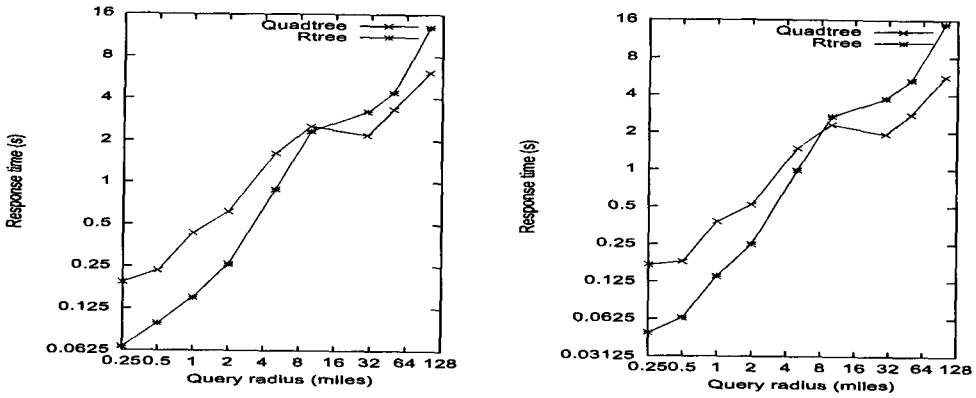

Figure 6: Performance for USBG dataset: (a) Touch Queries and (b) Coveredby Queries. R-trees are faster for query windows of 10 mile radii. Quadtrees are faster for larger windows.

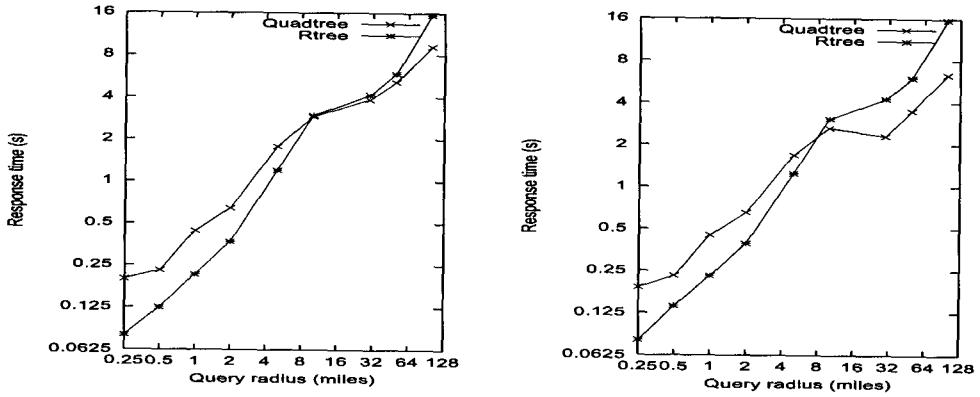

Figure 7: Performance for USBG dataset: (a) Overlapbdydisjoint queries and (b) Overlapbdyintersect queries. R-trees are faster for query windows of 10 mile radii. Quadtrees are faster for larger windows.

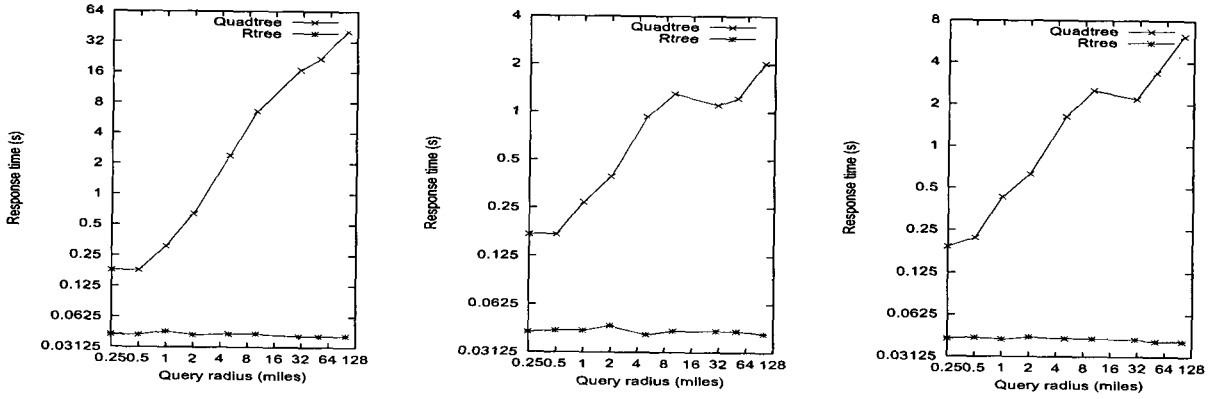

Figure 8: Performance for USBG dataset: (a) Contains queries, (b) Covers queries, and (c) Equals queries. R-trees are faster than Quadtrees by at least a factor of 2.

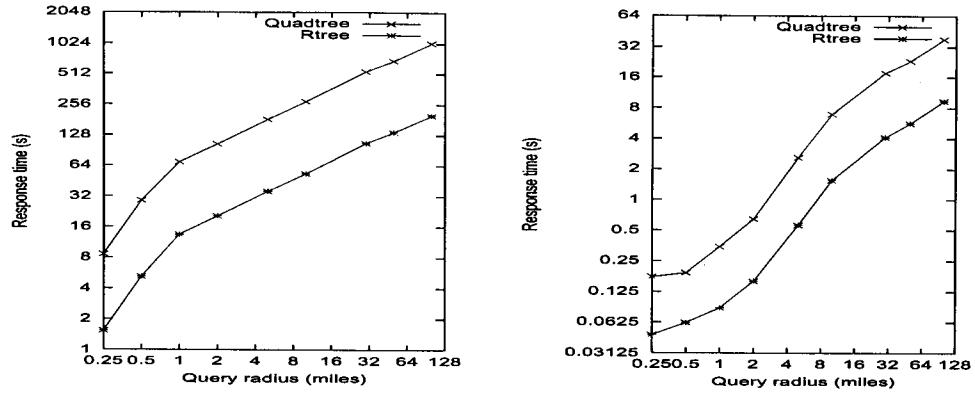

Figure 9: Performance for Within-distance queries: (a) ABI dataset and (b) USBG dataset. R-trees consistently outperform Quadtrees by at least a factor of 2. This is because Quadtrees use buffering around the query center and R-trees use use distance-based computation between query center and index/data geometries to avoid expensive buffering.

[11] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2), 1998.

[12] Y. J. Garcia, S. T. Leutenegger, and M. A. Lopez. A greedy algorithm for bulk loading R-trees. In *Proc. of ACM GIS*, 1998.

[13] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, 1984.

[14] G. Hjaltson and H. Samet. Ranking in spatial databases. In *Symposium on Spatial Databases (SSD)*, 1995.

[15] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest-neighbor queries. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 369–380, May 1997.

[16] M. Kornacker, C. Mohan, and J. Hellerstein. Concurrency and recovery in GiST. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 62–72, Tucson, Arizon, June 1997.

[17] S. T. Leutenegger, M. A. Lopez, and J. M. Edgington. STR: A simple and efficient algorithm for R-tree packing. In *Proc. Int. Conf. on Data Engineering*, 1997.

[18] K.-I. Lin, H. V. Jagdish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal*, 3:517–542, 1994.

[19] D. B. Lomet and B. Salzberg. The hB-tree: A multi-attribute indexing method with good guaranteed performance. *Proc. ACM Symp. on Transactions of Database Systems*, 15(4):625–658, December 1990.

[20] B. C. Ooi, C. Yu, K. L. Tan, and H. V. Jagadish. Indexing the distance: an efficient method to knn processing. In *Procöf the Int. Conf. on Very Large Data Bases*, 2001.

[21] D. Papadis, T. Sellis, Y. Theodoridis, and M. Egenhofer. Topological relations in the world of minimum bounding rectangles: a study with r-trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 92–103, 1995.

[22] K. V. Ravi Kanth, D. Agrawal, Amr El Abbadi, and Ambuj K. Singh. Dimensionality reduction for similarity searching in dynamic databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1998.

[23] K. V. Ravi Kanth and Siva Ravada. Efficient processing of large spatial queries using interior approximations. In *Symposium on Spatial and Temporal Databases (SSTD)*, 2001.

[24] K. V. Ravi Kanth, Siva Ravada, J. Sharma, and J. Banerjee. Indexing medium-dimensionality data in oracle. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1999.

[25] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 71–79, May 1995.

[26] H. Samet. Recent developments in linear quadtree-based geographic information systems. *Image and Vision Computing*, 5(3):187–197, Aug. 1987.

[27] H. Samet. *The design and analysis of spatial data structures.* Addison-Wesley Publishing Co., 1989.

[28] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r^+-tree: A dynamic index for multi-dimensional objects. *Procöf the Int. Conf. on Very Large Data Bases*, 13:507–518, 1988.

[29] Y. Theodoridis and T. K. Sellis. Optimization issues in r-tree construction. In *Geographic Information Systems (IGIS)*, pages 270–273, 1994.

[30] Y. Theodoridis and T. K. Sellis. A model for the prediction of r-tree performance. In *Proc. ACM Symp. on Principles of Database Systems*, 1996.

[31] F. Wang. Relational-linear quadtree approach for two-dimensional spatial representation and manipulation. *IEEE Trans. on Knowledge and Data Engineering*, 3(1):118–122, Mar. 1991.

[32] D. White and R. Jain. Algorithms and strategies for similarity retrieval. *Proc. of the SPIE Conference*, 1996.

[33] D. White and R. Jain. Similarity indexing with the SS-tree. *Proc. Int. Conf. on Data Engineering*, pages 516–523, 1996.