CS 581

PROJECT REPORT

INDEXING

SUBMITTED BY
Juhong Liu, Gent Panajoti, Huiyong Xiao

Table of Contents Page

A Report on Indexing by Juhong Liu, Gent Panajoti, and Huiyong Xiao

1. Introduction and Motivation

In order to efficiently process spatial queries, one needs specific access methods relying on a data structure called an index. These access methods accelerate the access to data; that is, they reduce the set of objects to be looked at when processing a query.

Processing a spatial query leads to the execution of complex and costly geometric operations. For operation such as point queries sequentially scanning and checking whether each object of a large collection contains a point involves a large number of disk accesses and the repeated expensive evaluation of geometric predicates. Hence, both the time-consuming geometric algorithm and the large volume of spatial collections stored on secondary storage motivate the design of efficient spatial access methods, which reduce the set of objects to be processed.

The most common access methods use B-trees and hash tables, which guarantee that the number of input/output operations (I/O.) to access data is respectively logarithmic and constant in the collection size, for exact search queries. A typical example of exact search is "Find County (whose name=) Cook. " Unfortunately, these access methods cannot be used in the context of spatial data. As an example, we take the B-tree. This structure indexes a collection on a key; that is, an attribute value of every object of the collection. The B-tree relies on a total order on the key domain, usually the order on natural numbers, or lexicographic order on strings of characters. Because of this order, interval queries are efficiently answered.

A convenient order for geometric objects in the plane is one that preserves object proximity. Two objects close in the plane should be close in the index structure. For example, objects inside a rectangle. They are close in the plane. It is desirable that their representation in the index also be close, so that a window query can be efficiently answered. Unfortunately, there is no such total order. The same limitation holds for hashing. Therefore, a large number of spatial access methods (indexes) were designed to try as much as possible to preserve object proximity.

The subject of our project is to study the performance of 2D R-tree, Quadtree, 3D R-tree, and Octree. All these indexes are used to efficiently process spatial queries for trajectories (in 2D case, the respective routes of the objects, without the timing information).

2. Our Goal

In this project we are to experimentally compare the running time(s) for: retrieval; insertion; deletion in 2D R-tree and Quadtree, and 3D R-tree variant and Octree.

The design of the experiment (i.e. bulk insertion, followed by "single"; or chunks of insertions…) will help us in order to get relevant results and have a valid interpretation.

3. Problem and Solution Approach

Our group focuses on the specific problem of comparing the running time performance of 2D R-tree and Quadtree, and 3D R-tree variant and Octree. The results of the experiments help us to decide which index is more efficient than the other(s) for different purposes. It depends on which operations are going to take place more in a particular database. This means that in some databases are performed more changes by inserting or deleting or changing information. We can see these operations to take place in databases where the movement of objects (cars, trucks etc) is stored time after time. The changing of the information takes place more in databases which have cellular telephone data, air-traffic data etc. Also, the amount of data to be inserted, deleted, or changed it depends on the purpose of a particular database.

We build insertion, deletion, and retrieval algorithms for each four indexes. Then we compare the running time for each particular algorithm in 2D R-tree and Quadtree, and 3D R-tree variant and Octree. The input data are trajectories or routes expressed by (x1, y1, t1), (x2, y2, t2)…or (x1, y1), (x2, y2)… We perform each operation with 5 groups of different data amounts: 180, 360, 540, 720, 900 for 3 dimensional data

(trajectories) and 200, 400, 600, 800, 1000 for 2 dimensional data (routes). This gives us a better perception on how fast each algorithm performs with different data amounts. Thus, we know which index to use for a particular database only by knowing the amount of data that database has to handle. Besides this, it is important the way that the insertion, deletion, or range query is performed for each group of data amounts in order to get relevant results and a valid interpretation. To achieve this we perform the insertion for each index by performing single insertions after bulk insertion for each group of data (first insert 95% of the data, then insert 5%). Also, we perform chunk insertions for each index by inserting 90 (for trajectories) or 100 (for routes) data every time. As for range query, we perform that for each index by using different range sizes (0.1%, 1%, and 10%). We perform the delete (remove) operation in the same way as we do for insertion by making single deletion (5% of each group of data amount), followed by bulk deletion (95% of data) and also by making chunk deletion (90 or 100 data every time).

Another issue of the experiment is the use of different sizes of leaf nodes. In our experiments we set the MAXLEAFNODE (maximum size of leaf node) in three different values: 16, 32 and 64.for trajectories and 32, 64 and 128 for routes.

4. Implementation Details

In our project we have implemented four indexing as we mentioned before. These indexing are Quadtree, Octree, 2D R-tree, and 3D R-tree variant. In this part of the report we explain the implementation of these indexing. Because the operations of insertion, deletion, and range search are similar, we explain their implementation in two groups. In the first group we treat 2D R-tree and 3D R-tree variant, while in the second group are Quadtree and Octree. Quadtree and 2D R-tree are indexes for two-dimensional objects, which are routes in this project. Octree and 3D R-tree are indexes for three-dimensional objects, which are trajectories in this project.

4.1. R-Tree Index Structure

R-tree is height-balanced tree similar to a B-tree with index records in its leaf nodes containing pointers to data object. Nodes correspond to disk pages if the index is disk-resident. Leaf nodes in a R-tree contain index record entries of the form

(I, tuple-identifier)

where tuple-identifier refers to a tuple in the database and I is an n-dimensional rectangle which is the bounding box of the spatial object indexed. Non-leaf nodes contain entries of the form

(I, child-pointer)

where child-pointer is the address of a lower node in the R-tree and I covers all rectangles in the lower node's entries. We let M be the maximum number of entries that will fit in one node, and let m = M/2 be a parameter specifying the minimum number of entries in a node.

The properties of an R-tree are:

- Every leaf node contains between *m* and *M* index records unless it is the root.
- For each index record (*I, tuple-identifier*) in al leaf node, *I* is the smallest rectangle the spatially contains the n-dimensional data object represented by the indicated tuple.
- Every non-leaf node has between *m* and *M* children unless it is the root.
- For each entry (*I, child-pointer*) in a non-leaf node, *I* is the smallest rectangle contains the rectangles in the child node.
- The root node has at least two children unless it is a leaf.
- All leaves appear on the same level.

4.1.1 Search Algorithm in an R-tree

- *Search subtrees* – If *N* (root node) is not a leaf, check each entry *E* to determine whether *E I* overlaps search rectangle *S*. For all overlapping entries, we invoke **Search** algorithm on the tree whose root node is pointed to by *Ep* [2].
- *Search leaf node* – If N is a leaf node, then we return all trajectories or routes of this node.

4.1.2 Insert Algorithm in an R-tree (see Fig. 1.a and Fig. 1.b for illustration)

- *Find position for new record* – Invoke **ChooseLeaf** algorithm to select a leaf node *L* in which to place *E*.
- *Add record to leaf node* – If *L* has room for another entry, install *E*. Otherwise invoke **SplitNode** algorithm to obtain *L* and *LL* containing *E* and all the old entries of *L*.
- *Propagate changes upward* – Invoke **AdjustTree** algorithm on *L*, also passing *LL* if a split was performed.
- *Grow tree taller* – If node split propagation caused the root to split, create a new root whose children are the two resulting nodes.

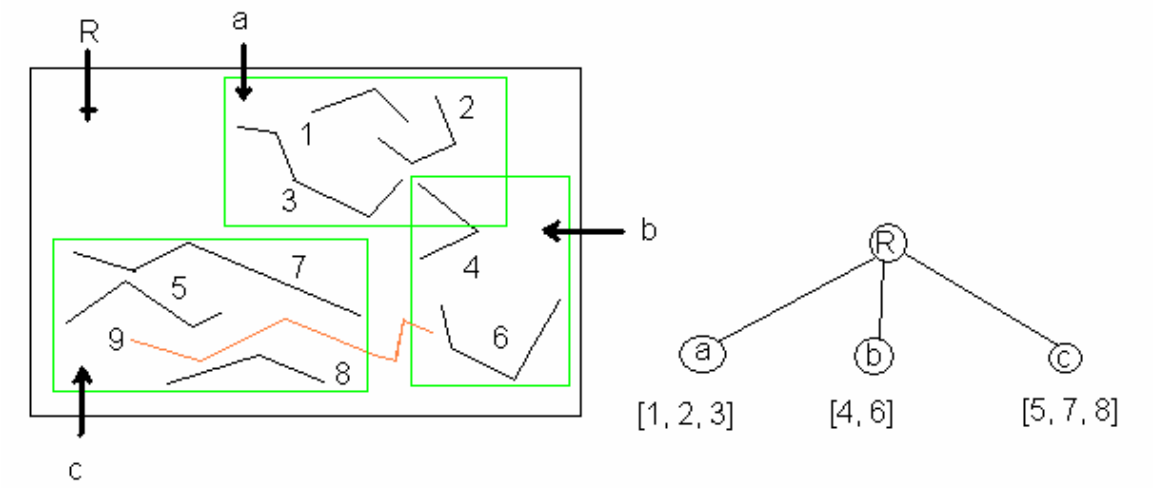


Fig. 1.a – Before the trajectory 9 is inserted in 2D R-tree

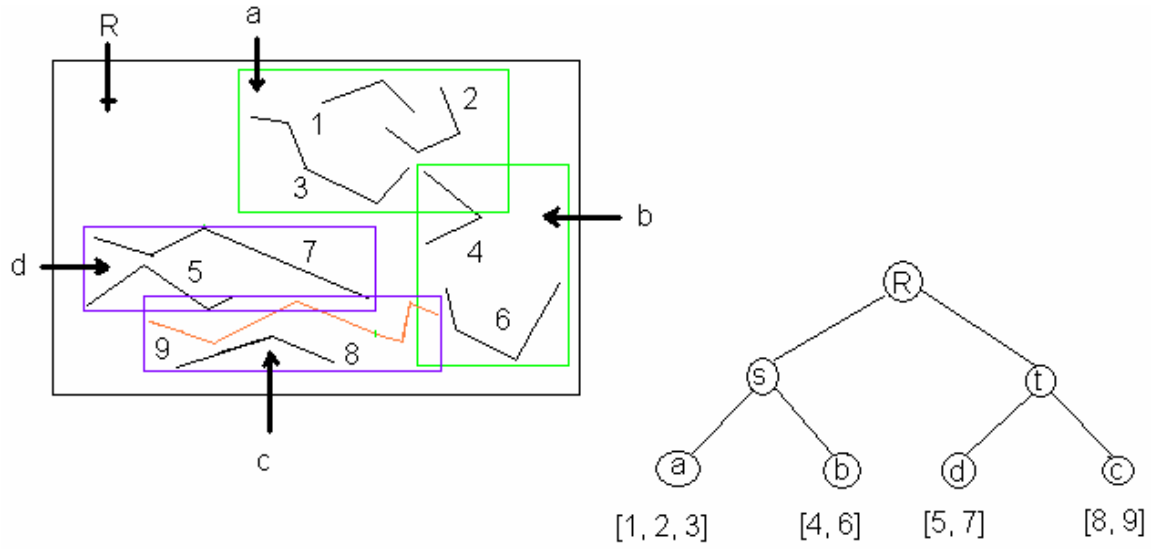


Fig. 1.b – After the insertion of the trajectory 9 in 2D R-tree
In the above illustration, the MAXLEAFSIZE is 3

4.1.3 ChooseLeaf Algorithm in an R-tree

- *Initialize* – Set *X* to be the root node
- *Leaf check* – If *X* is a leaf, return *X*.
- *Choose subtree* – If *X* is not a leaf, let *F* be the entry in *X* whose rectangle *F I* needs least enlargement to include *E I*. Resolve ties by choosing the entry with the rectangle of smallest area.
- *Descend until a leaf is reached* – Set *X* to be the child node pointed to by *Fp* and repeat from step 2 of **ChooseLeaf** algorithm.

4.1.4 AdjustTree Algorithm in an R-tree

- *Initialize* – Set *X = L* If *L* was split previously, set *XX* to be the resulting second node.
- *Check if done* – If *X* is the root, stop.
- *Adjust covering rectangle in parent entry* – Let *P* be the parent node of *X*, and let E_X be *X*'s entry in *P*. Adjust E_X *I* so that it tightly encloses all entry rectangles in *X*
- *Propagate node split upward* – If *X* has a partner *XX* resulting from an earlier split, create a new entry E_{XX} with $E_{XX}p$ pointing to *XX* and E_{XX} *I* enclosing all rectangles in *XX*. Add E_{XX} to *P* if there is room. Otherwise, invoke **SplitNode** algorithm to produce *P* and *PP* containing E_{XX} and *P*'s old entries.
- Move up to next level – Set *X = P* and set *XX = PP* if a split occurred. Repeat **AdjustTree** algorithm from step 2.

4.1.5 Deletion Algorithm in an R-tree

- Find nod containing record – Invoke FindLeaf to locate the leaf node L containing E. Stop if the record was not found.
- Delete record – Remove E from L
- Propagate changes – Invoke CondenseTree algorithm, passing L.

- Shorten tree – If the root node ahs only one child after the tree has been adjusted, make the child new root

4.1.6 FindLeaf Algorithm in an R-tree

- *Search subtrees* – If *N* is not a leaf, check each entry *F* in *N* to determine if *F I* overlaps *E I.* For each such entry invoke **FindLeaf** algorithm on the tree whose root is pointed to by *Fp* until *E* is found or all entries have been checked.
- *Search leaf node for record* – If *N* is a leaf, check each entry for record. If *N* is a leaf, check each entry to see if it matches *E*. If *E* is found return *N*.

4.1.7 CondenseTree Algorithm in an R-tree

- *Initialize* – Set $X = L$. Set *Q*, the set of eliminated nodes, to be empty.
- *Find parent entry* – If *X* is the root, we go to step 6 of this algorithm. Otherwise let *P* be the parent of *X*, and let E_X be the *X*'s entry in *P*.
- *Eleminate under-full node* – If *X* has fewer than *m* entries, delete E_X from *P* and add *X* to set *Q*.
- *Adjust covering rectangle* – If *X* has not been eliminated adjust E_X *I* to tightly contain all entries in *X*.
- *Move up one level in tree* – Set $X = P$ and repeat this algorithm from step 2
- *Re-insert orphaned entries* – Re-insert all entries of nodes in set *Q*. Entries from eliminated leaf nodes are re-inserted in tree leaves as described in algorithm **Insert**, but entries from higher-level nodes must be place higher in the tree, so that leaves of their dependent subtrees will be on the same level as leaves of the main tree.

4.2. Quadtree and Octree Index Structure

Trajectories or routes to be indexed are mapped to cells obtained by a recursive decomposition of the space into quadrants, known as quadtree decomposition. The space is recursively decomposed into quadrants until the number of trajectories or routes overlapping each quadrant are less than the page capacity. The index is represented as a

quaternary tree (each internal node has four children in Quadtree and eight children in Octree, one per quadrant). With each leaf is associated a disk page, which stores the index entries. A trajectory or route appears in as many leaf quadrants as it overlaps.

4.2.1 Insert Algorithm (see Fig. 2.1, 2.2, and 2.3 for illustration)

- Get the root N
- Check for overlapping of the trajectory or the route with the node N. If there is not overlapping we stop.
- If N is a internal node, repeat step 2 of this algorithm for each child of node N.
- N is a leaf node we insert the trajectory or route.
- If the number of trajectories or routes is more than MAXLEAFSIZE then we split that leaf node into four (for quad-tree) or eight (for oct-tree) leaf nodes. Then insert the trajectories or routes in original leaf node to the new leaf nodes.

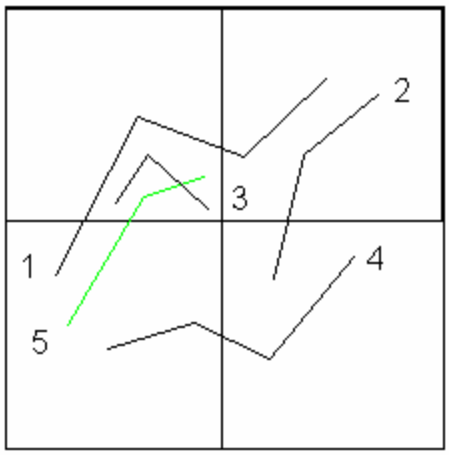


Fig. 2.1 Before inserting trajectory five

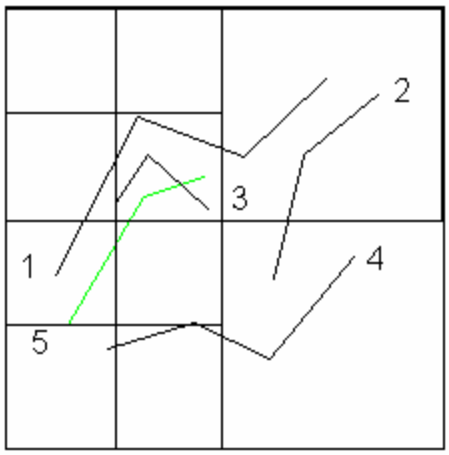


Fig. 2.2 Start splitting the quadrants

.

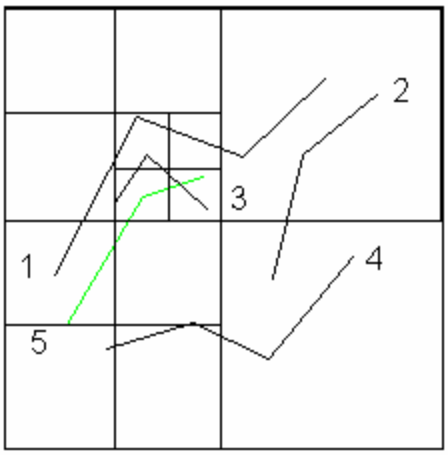


Fig. 2.3 Continue splitting the quadrants

In the above illustration, the MAXLEAFSIZE is 2.

4.2.2 Search Algorithm

- Get the root N
- Check for overlapping of query range and the range of N
- If the answer from the second step is NOT, we return
- If N is internal node, we go to the children of node N and repeat step 2 for each N's child.
- If the node is leaf node then we return all trajectories or routes of this node

4.2.3 Deletion Algorithm

- Get the root
- Check for overlapping of trajectory or route and the node range
- If the answer is NOT, we return.
- If the answer is YES and the node N is internal node, we go to the children of N and repeat step 2 for each child.
- If N is leaf node, we delete the trajectories or routes from N and check the father for the MINNODESIZE.
- If the number of trajectories or routes is not less than the MINNODESIZE, we return.

- If the number of trajectories or routes is less than the MINNODESIZE, we delete all the children of the father and change the father into a leaf node.
- For the father of node N, go step 6

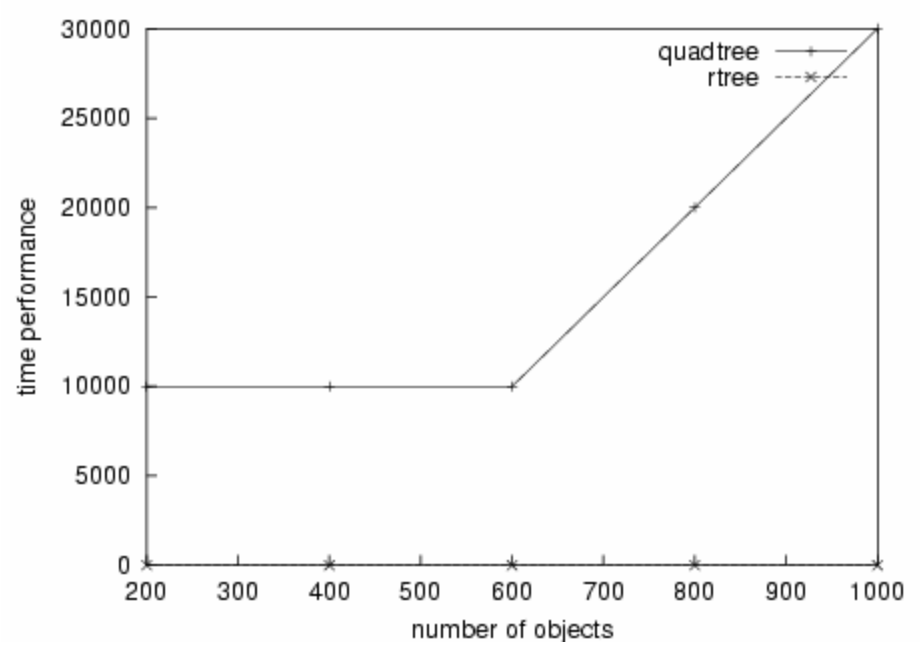
5. Experimental Settings

The algorithms were implemented in the C++ language under the SunOS operating system on the Sun Microsystems Ultra 5 machines in the Computer Science laboratory. Experiments were compared by the amount of time they take on these computers using clock() function in the C++ programming language. This function returns the number of clock ticks (the CPU time taken) the program has taken. We ran all four algorithms on five different groups of trajectories or routes (180, 360, 540, 720, and 900 trajectories or 200, 400, 600, 800 and 1000 routes) and we measure the running time for insertion, remove, and range search as we describe at Problem and Solution Approach part of this report.

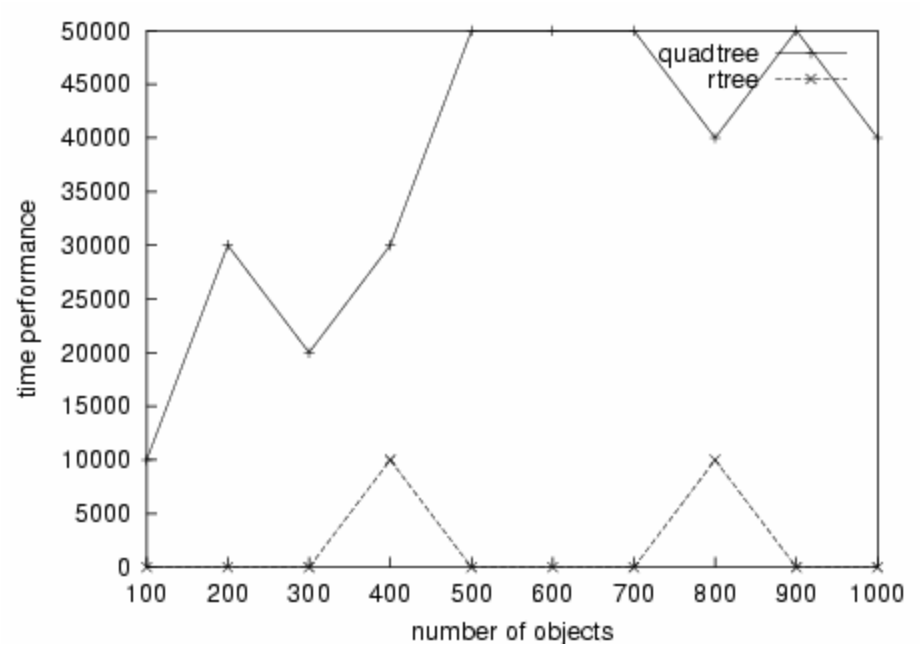
The second phase of our experiment is to compare the running time results that we obtain by our algorithms. That is; we compare the running time of Quadtree and 2D R-tree, Octree and 3D R-tree variant. For all insertions (single and chunk), range searches (0.1 %, 1 %, and 10 %), and removes (single and chunk) we build graphs where we see the performance of Quadtree algorithm compare with 2d R-tree, and Octree algorithm compare with 3D R-tree variant. All the results of these comparisons are included in Results and Discussion part of this report.

6. Results and Discussion

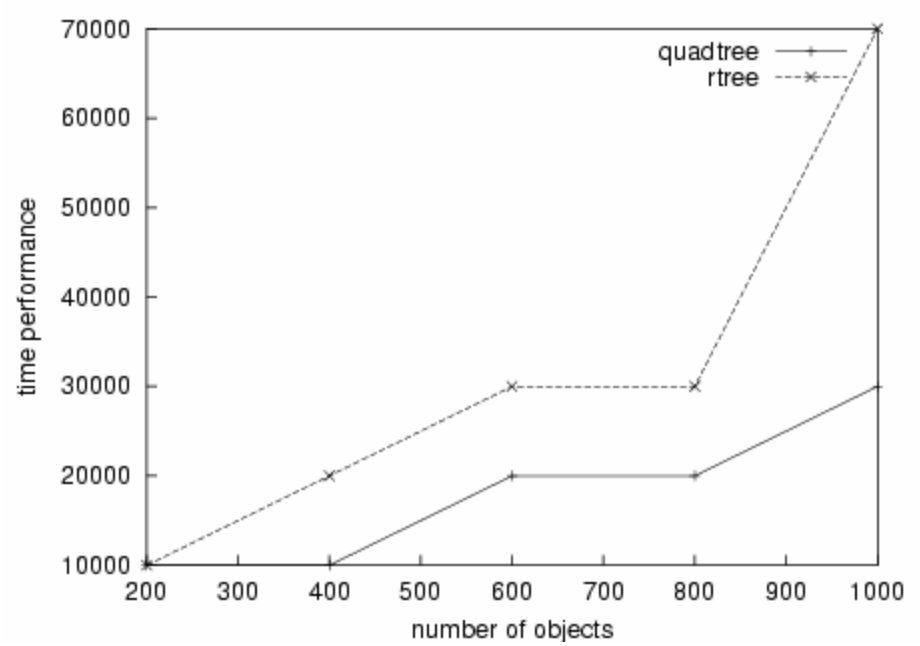
2 Dimensional with Leaf Size 32



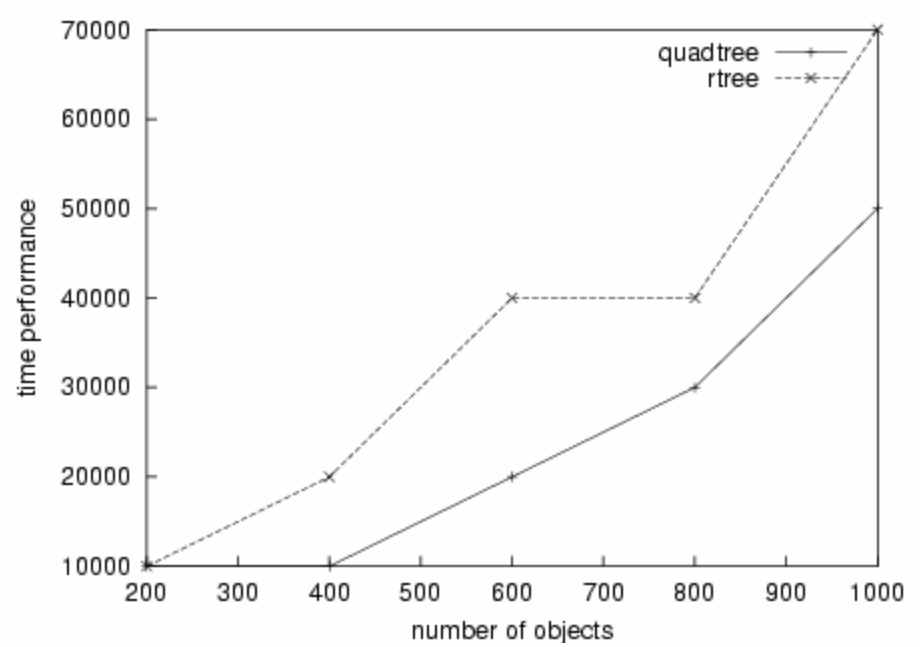
2d_single_insert_32



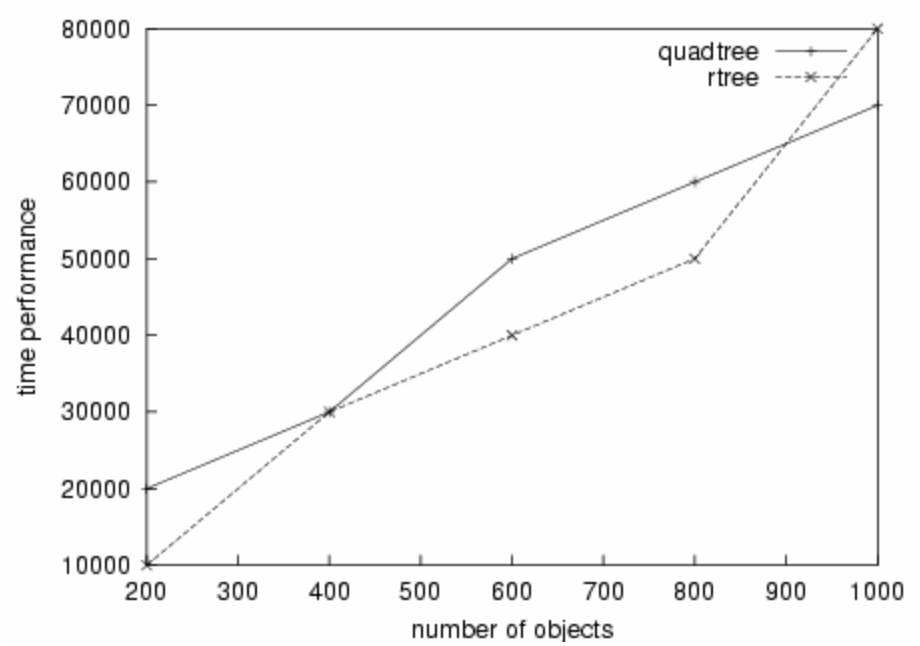
2d_chunk_insert_32



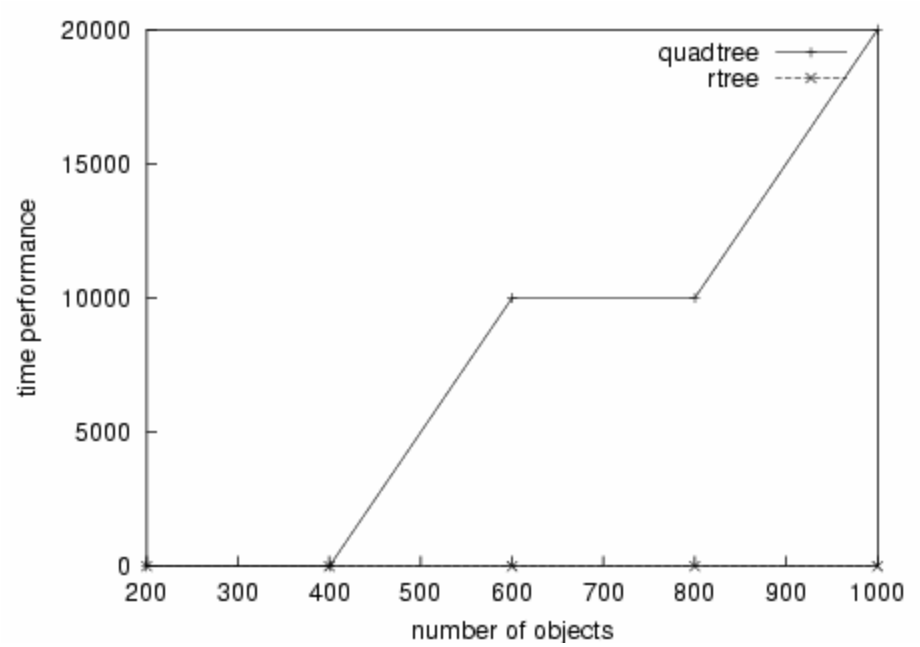
2d_search_0.1%_32



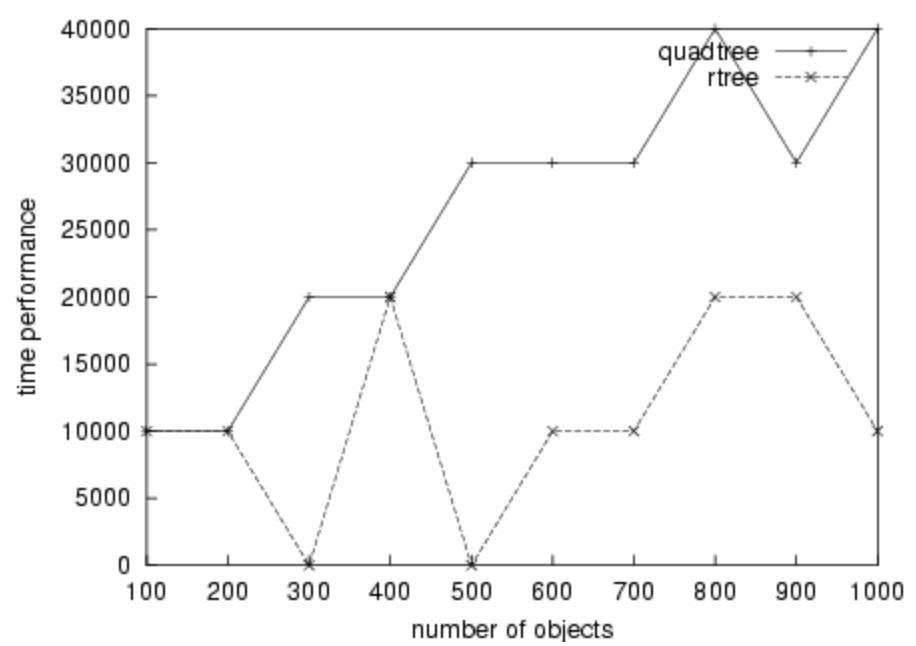
2d_search_1%_32



2d_search_10%_32

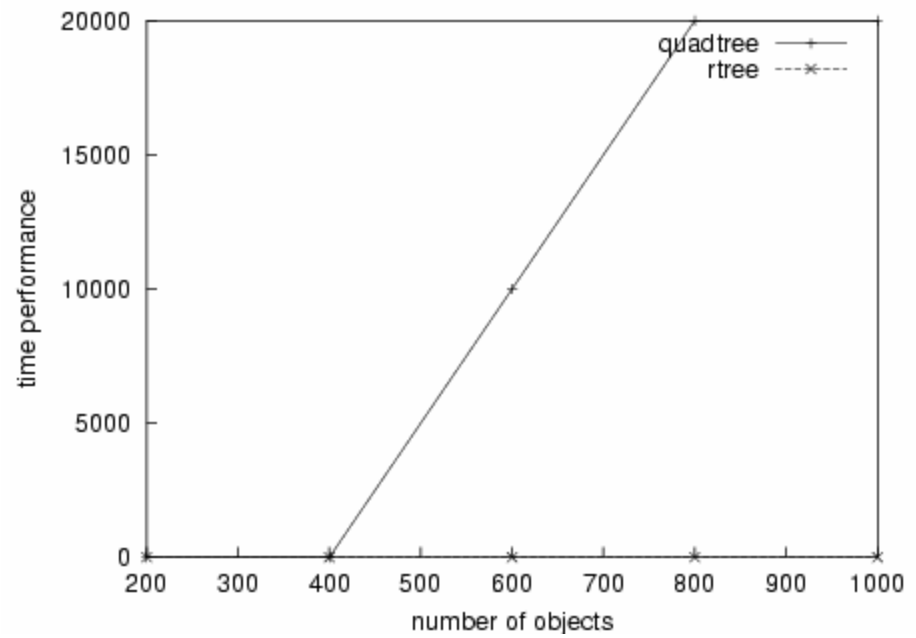


2d_single_remove_32

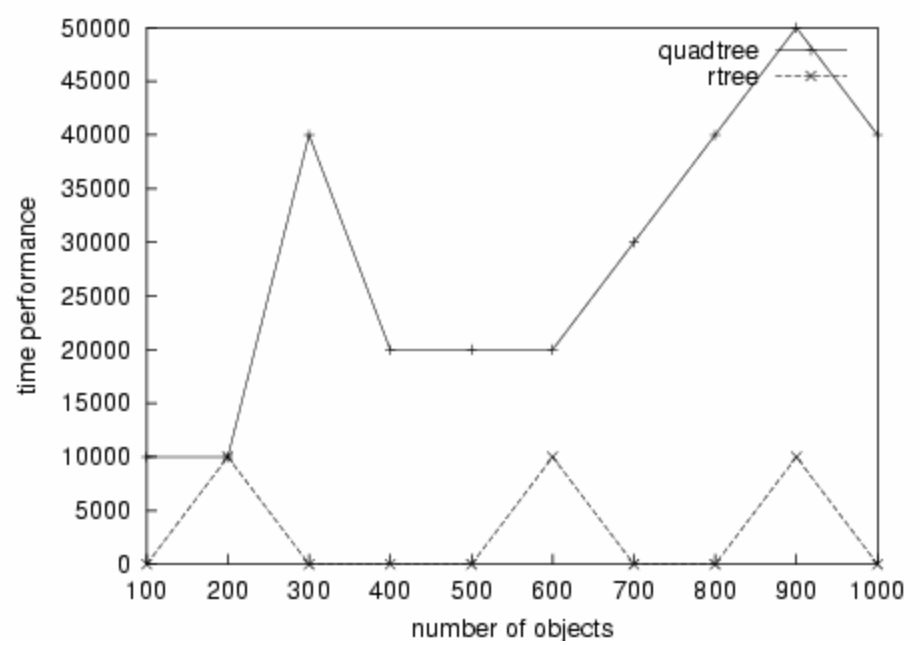


2d_chunk_remove_32

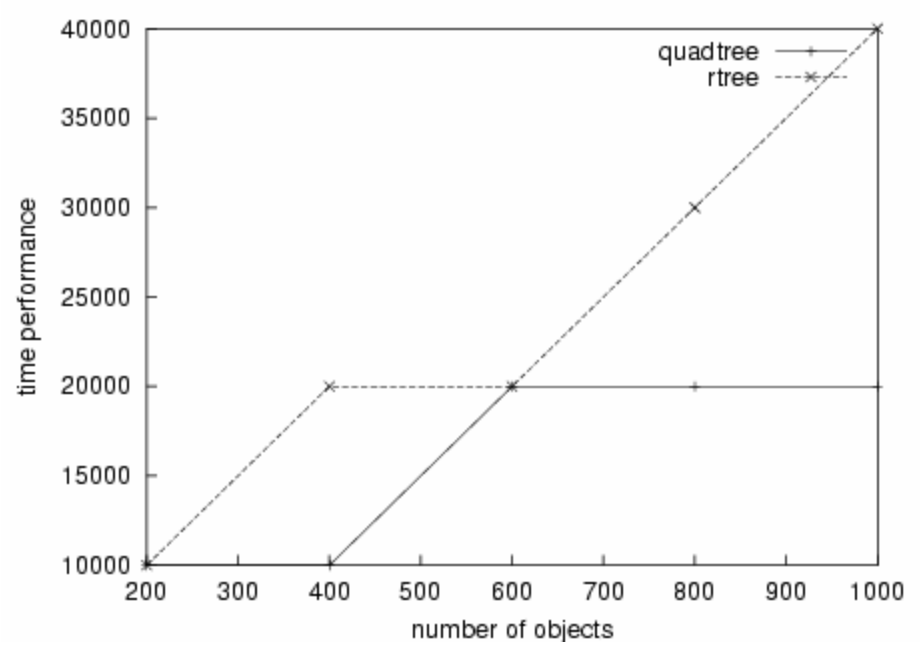
2 Dimensional with Leaf Size 64



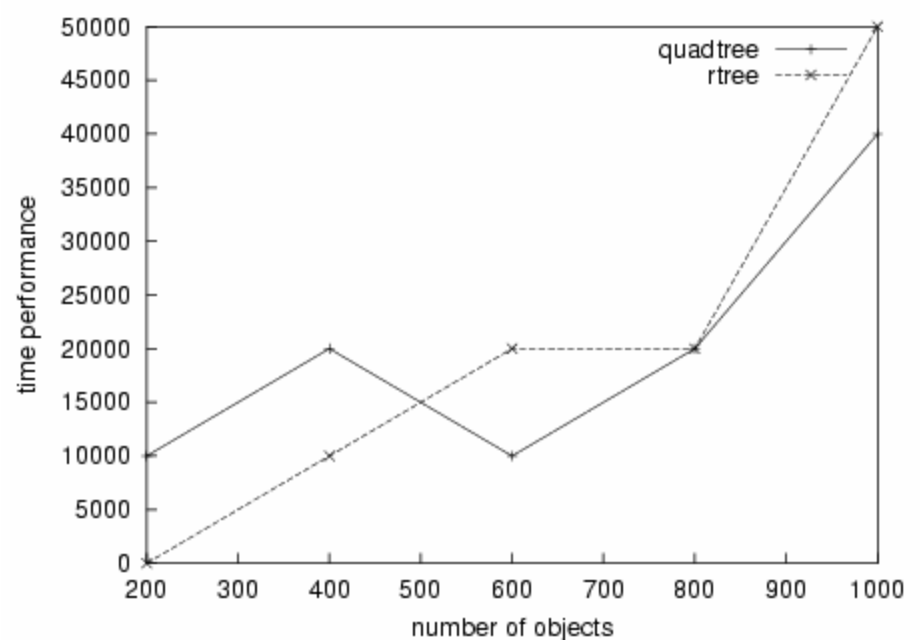
2d_single_insert_64



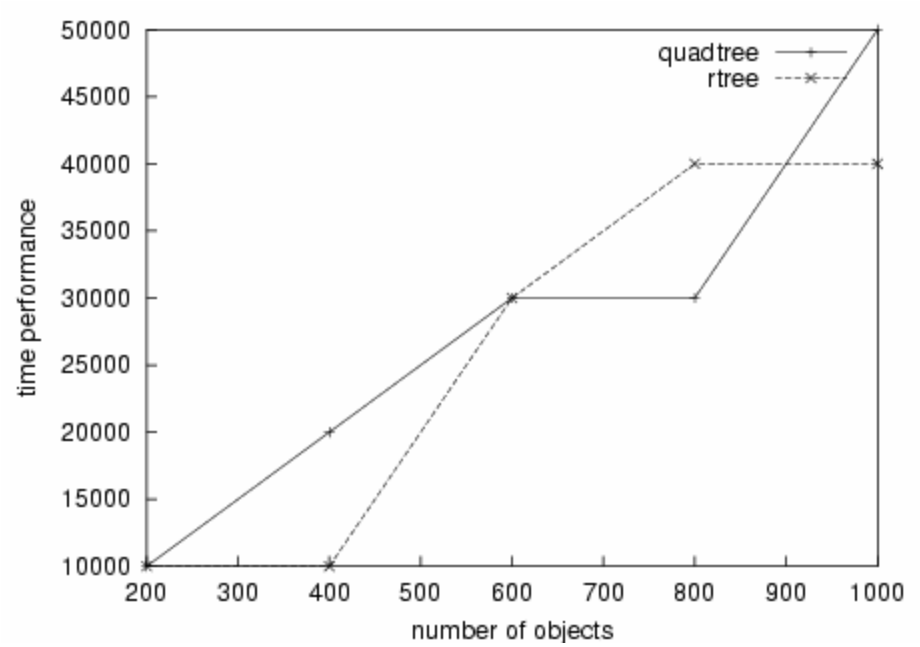
2d_chunk_insert_64



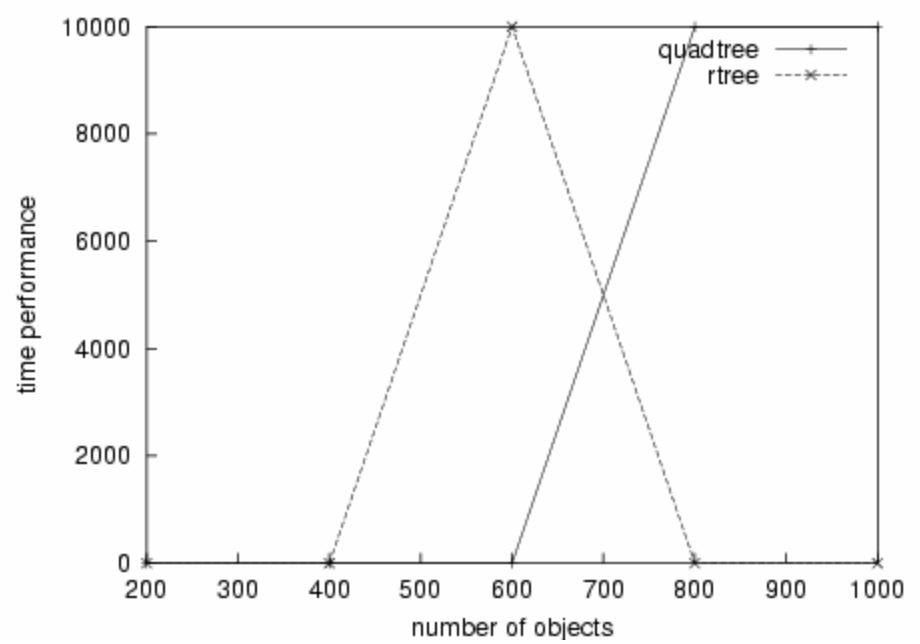
2d_search_0.1%_64



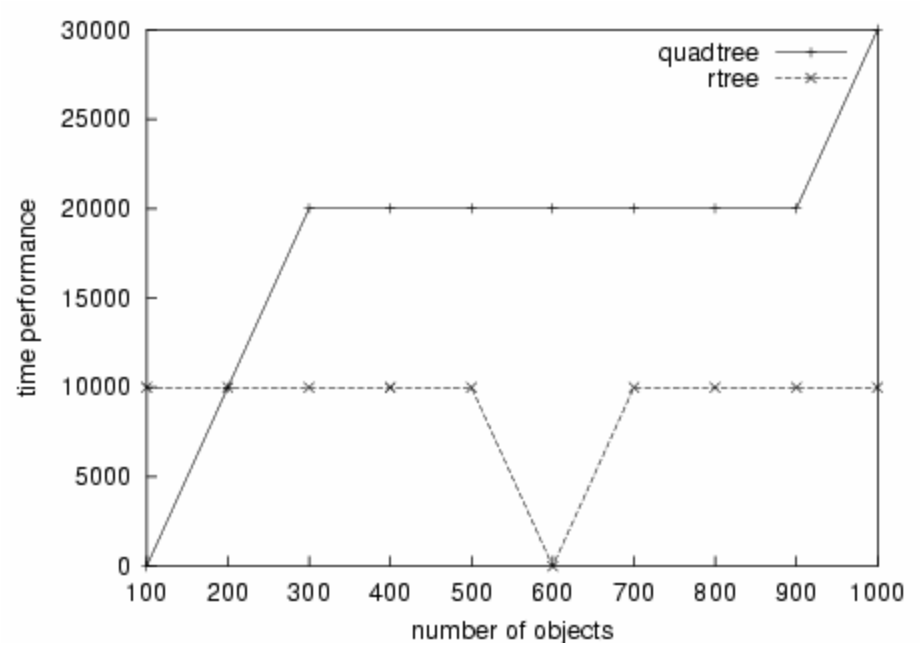
2d_search_1%_64



2d_search_10%_64

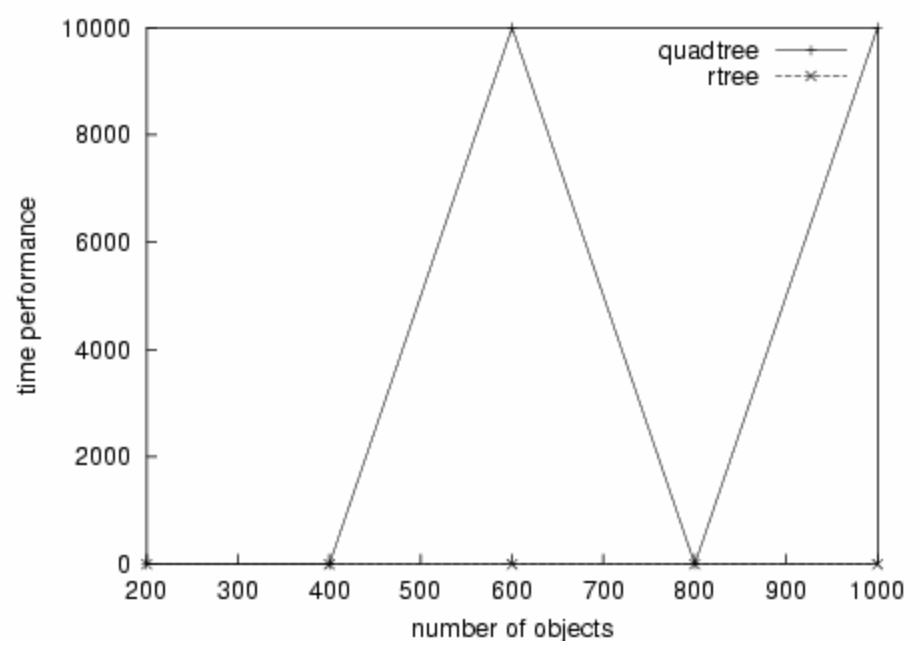


2d_single_remove_64

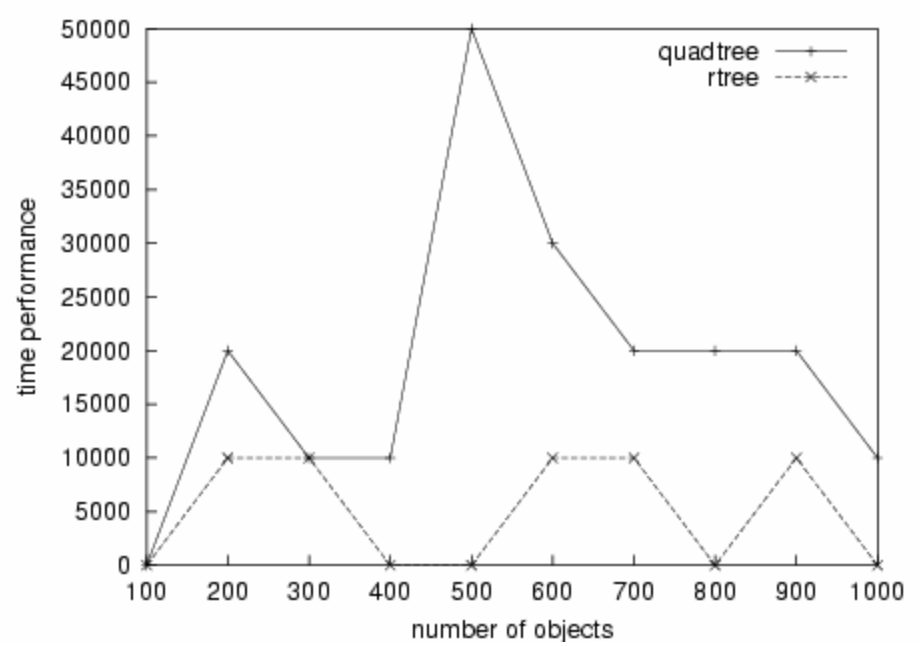


2d_chunk_remove_64

2 Dimensional with Leaf Size 128



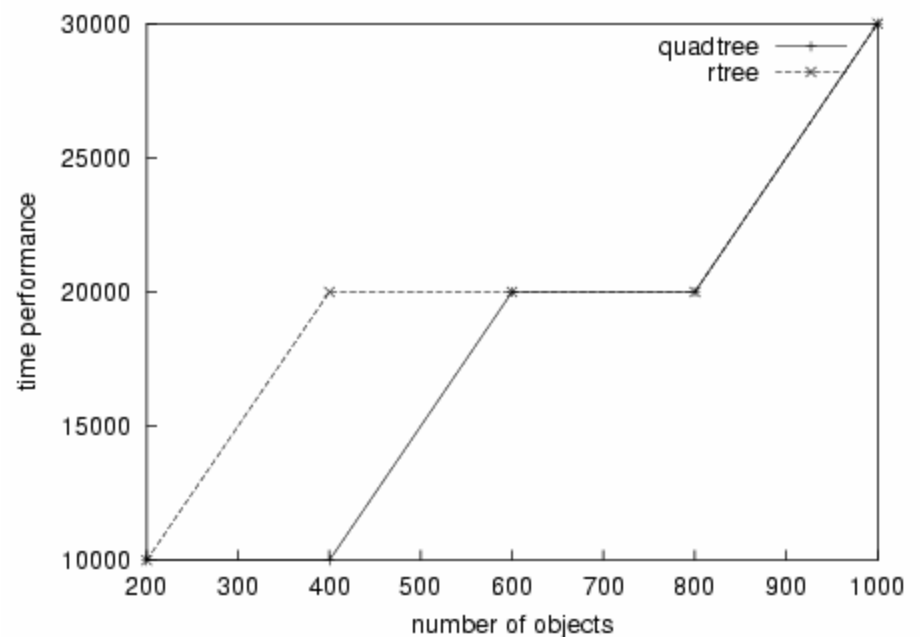
2d_single_insert_128



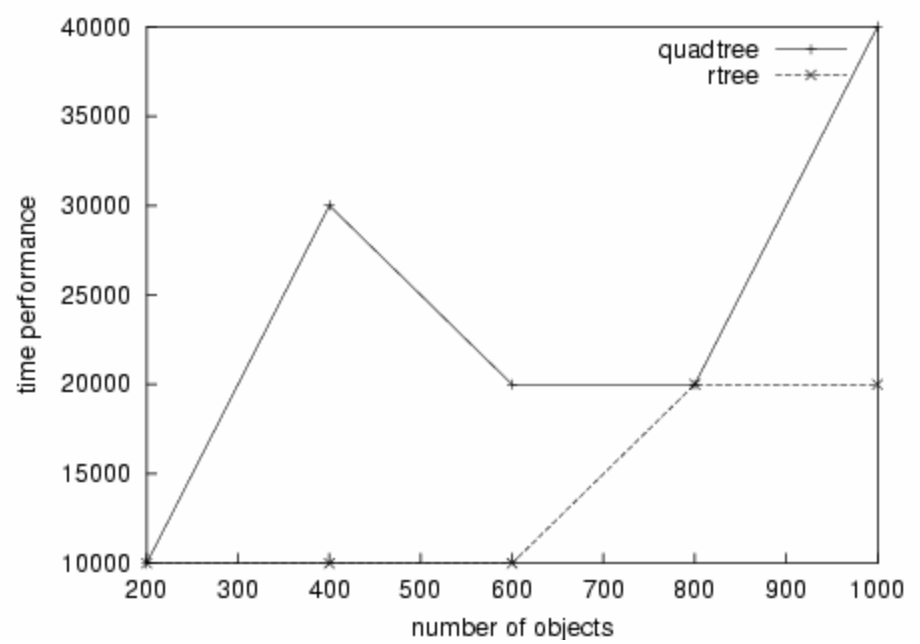
2d_chunk_insert_128

In above plot, when data number is 500, the time performance of quad tree for chunk insertion is much longer than other chunk insertion. We think when data number is

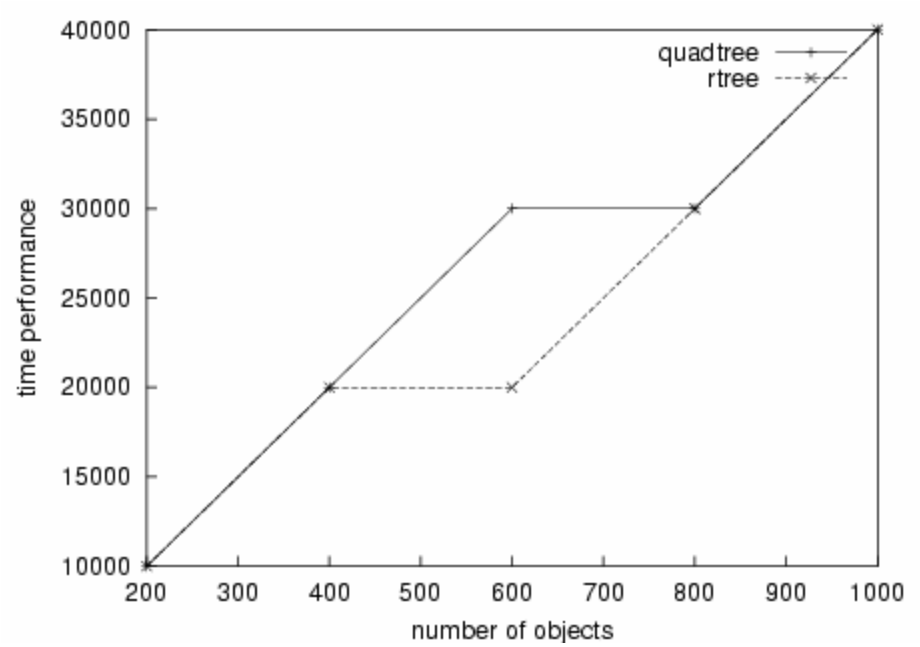
400, the leaf nodes of the quad tree are almost full, so when another 100 data insert, many leaf nodes split, this operation takes much time. After these nodes split, many of the leaf nodes are far from full, so when another 100 data insert, it takes less time, since few of them split.



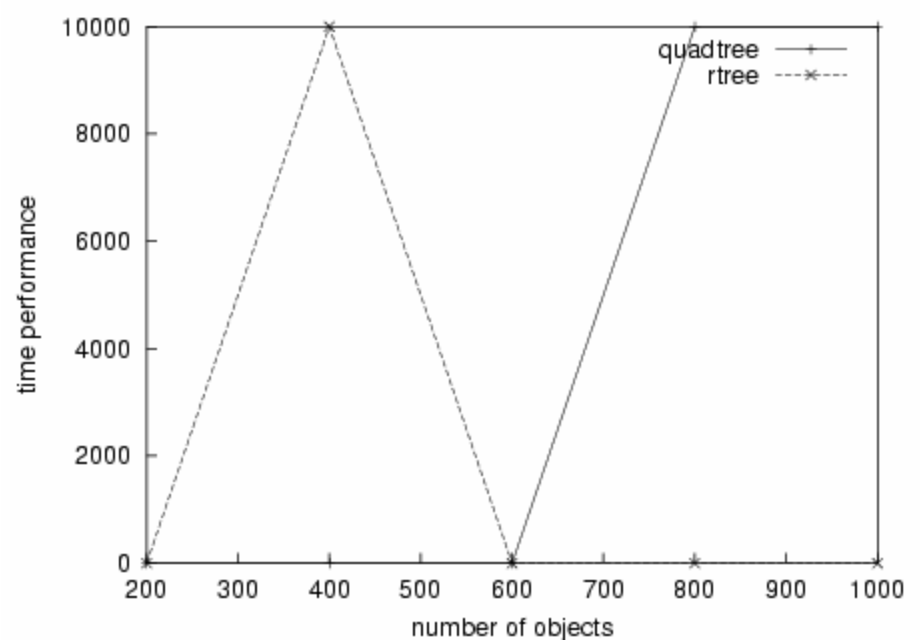
2d_search_0.1%_128



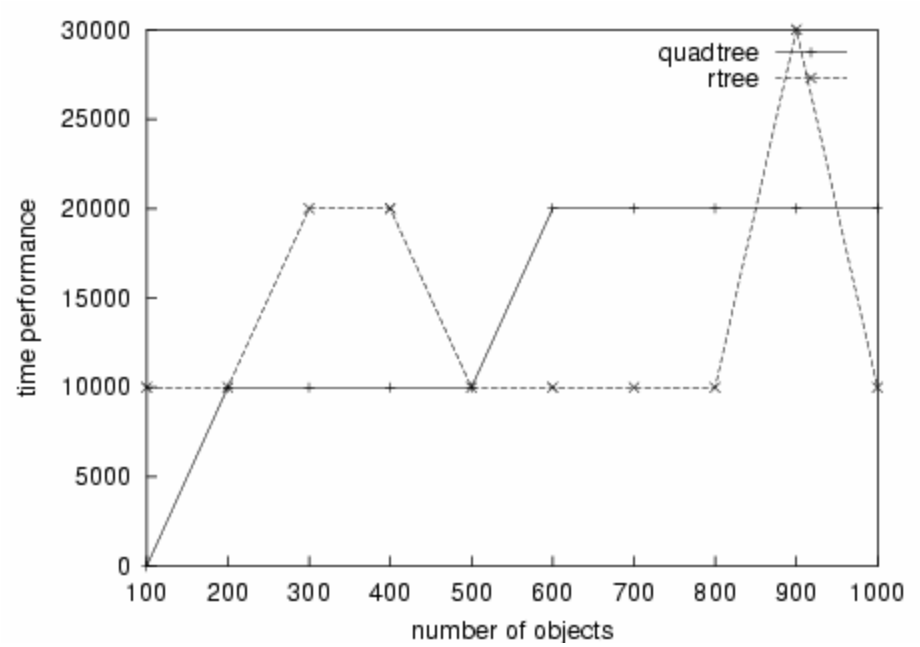
2d_search_1%_128



2d_search_10%_128



2d_single_remove_128

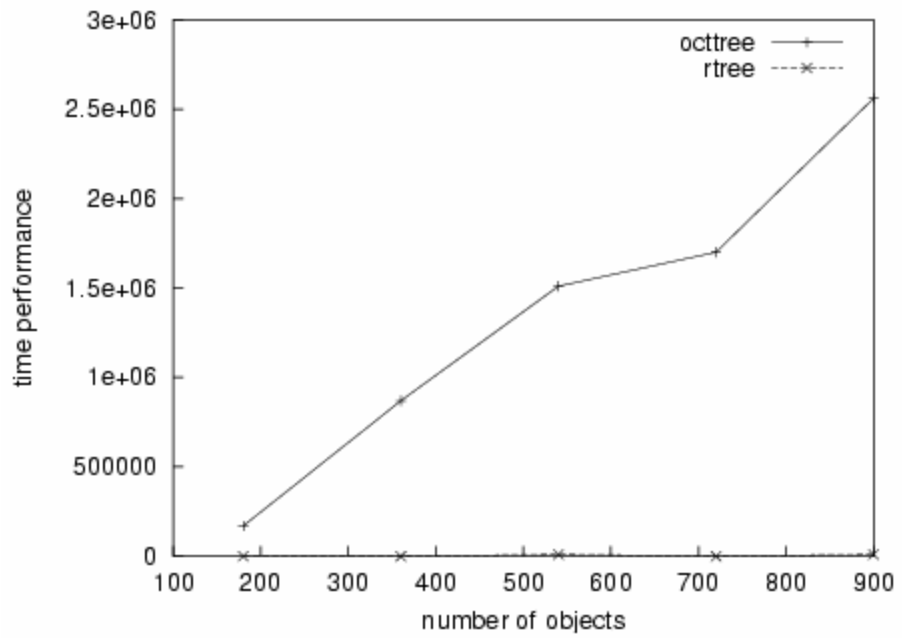


2d_chunk_remove_128

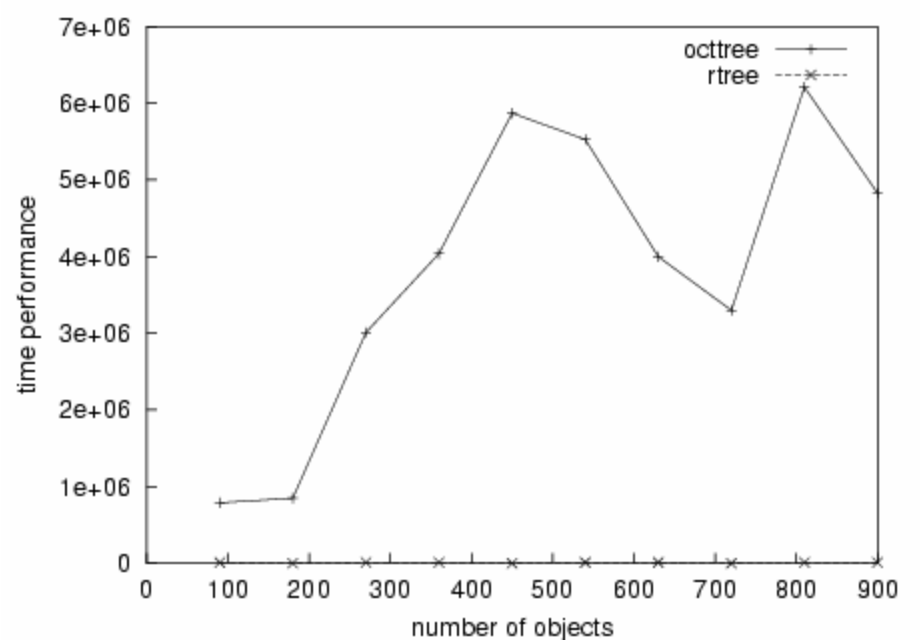
Similar as we discuss for the figure 2d_chunk _insert_128, the reason for the peak of R-tree in data equal to 900 is that the leaf nodes are nearly empty when the number of

data is equal to 1000. So, another 100 data remove makes many leaf nodes removed and some internal nodes change to leaf nodes.

3 Dimensional with Leaf Size 16

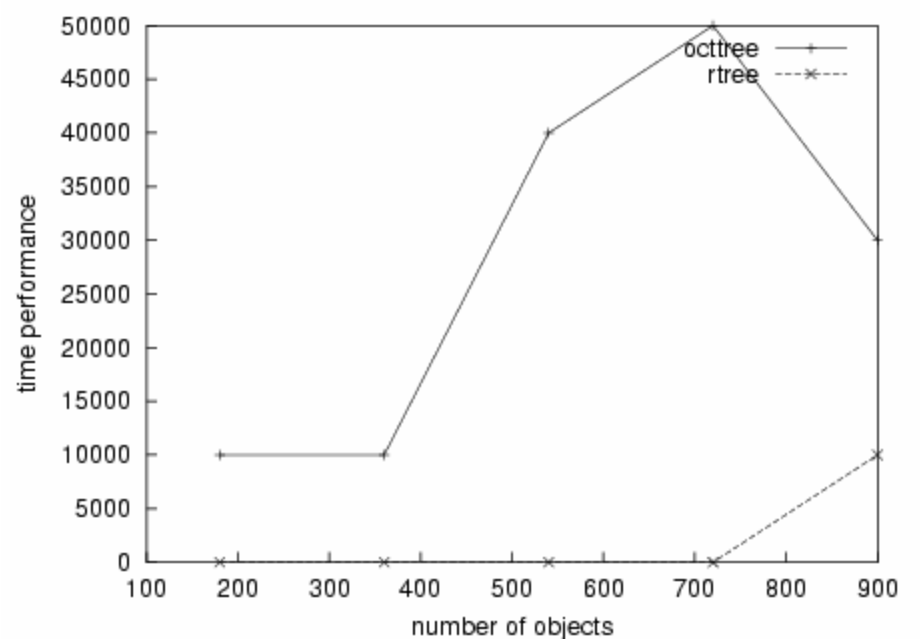


3d_single_insert_16

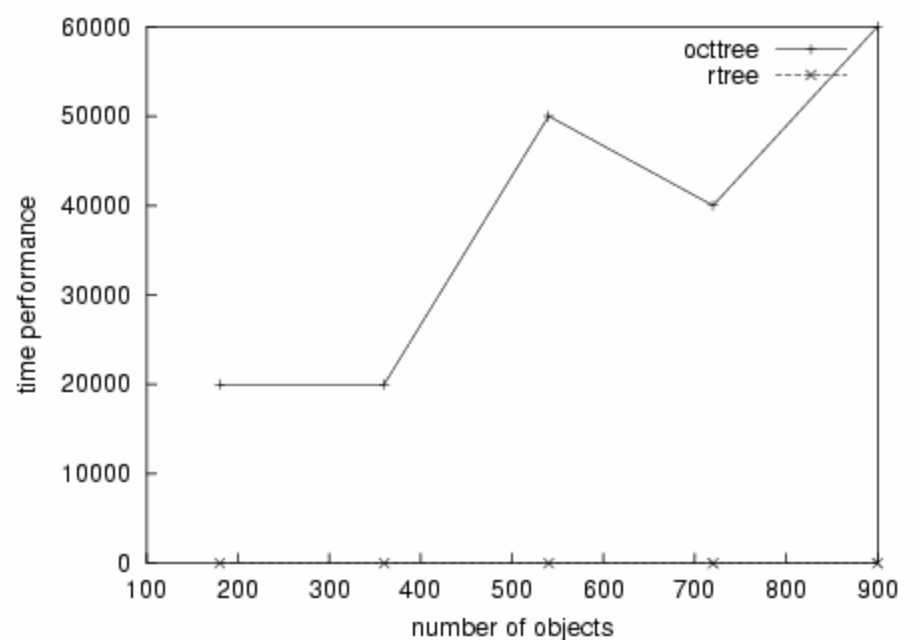


3d_chunk_insert_16

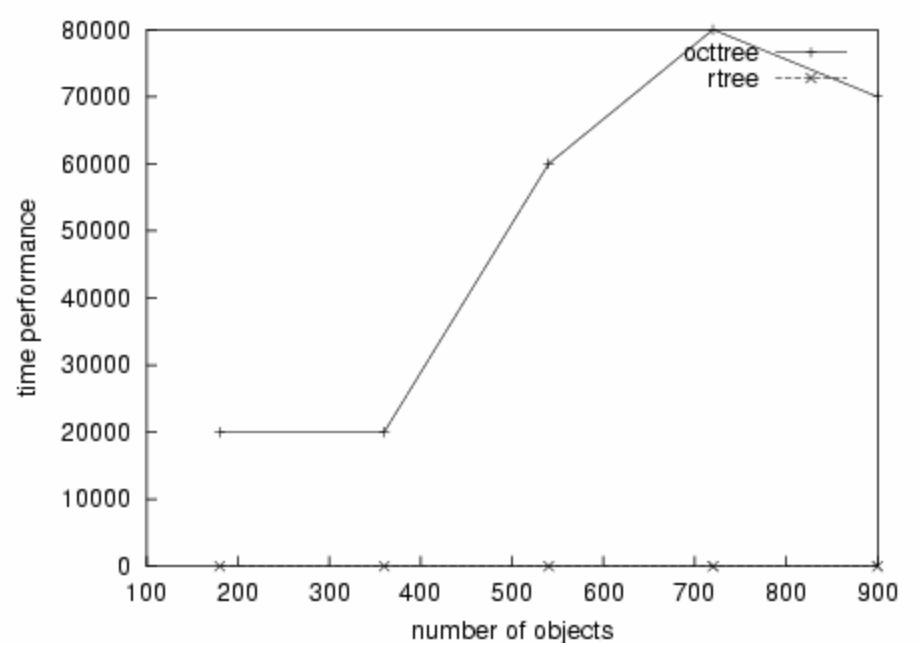
The same reason as we discussed in figure 2d_chunk_insertion_128 leads to the two peaks when data is 450 and 810.



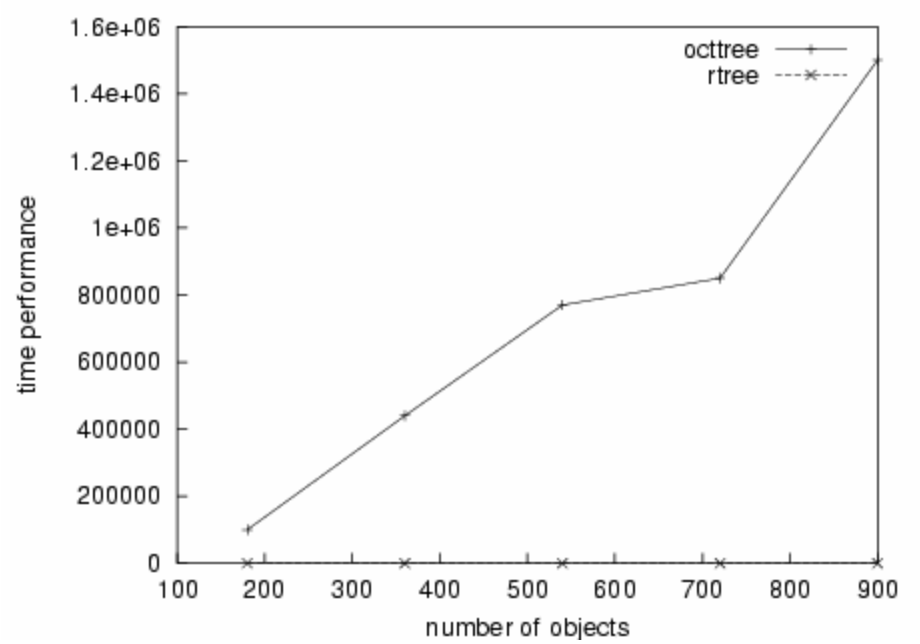
3d_search_0.1%_16



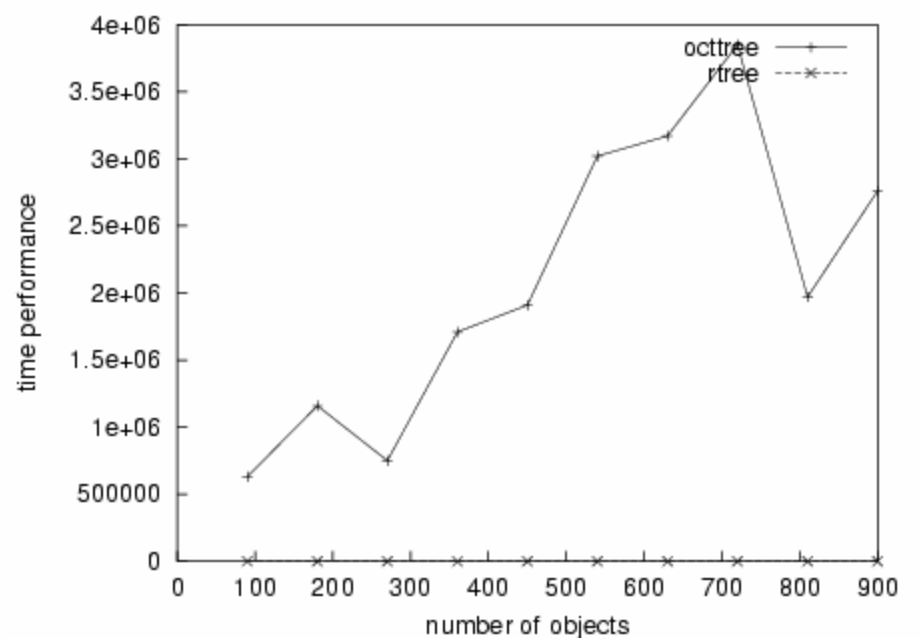
3d_search_1%_16



3d_search_10%_16



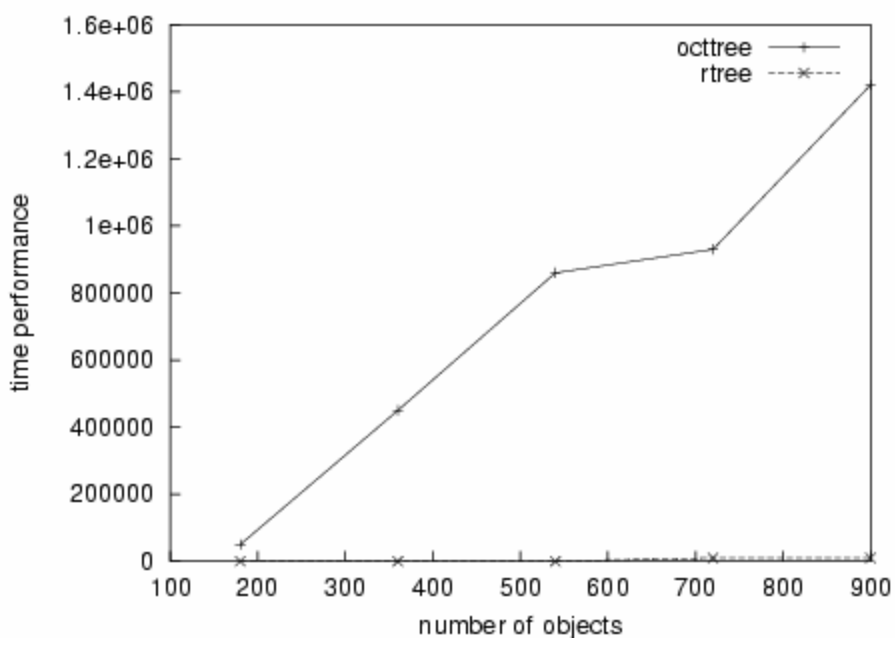
3d_single_remove_16



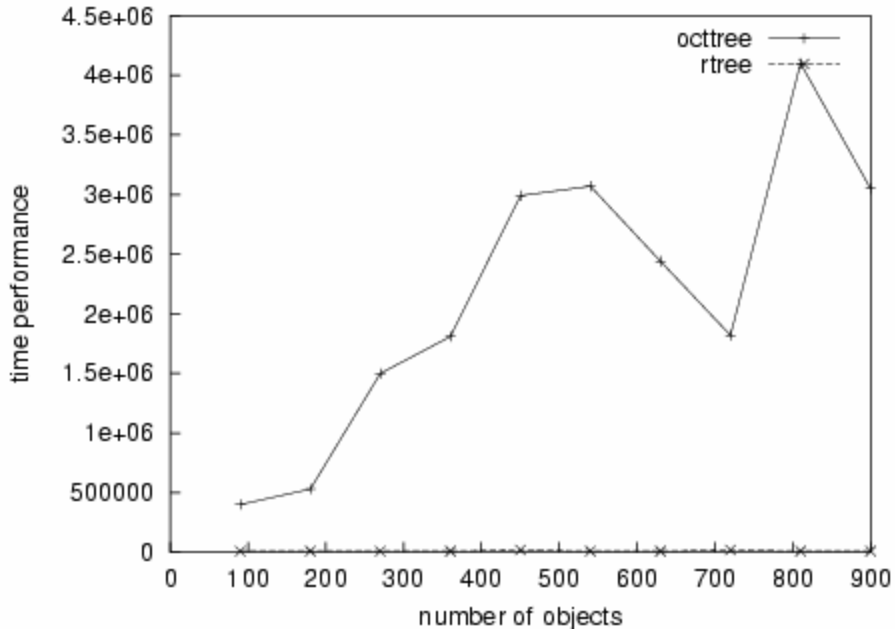
3d_chunk_remove_16

The same reason as we discussed in figure 2d_chunk_remove_128 leads to the two peaks when data is 720.

3 Dimensional with Leaf Size 32

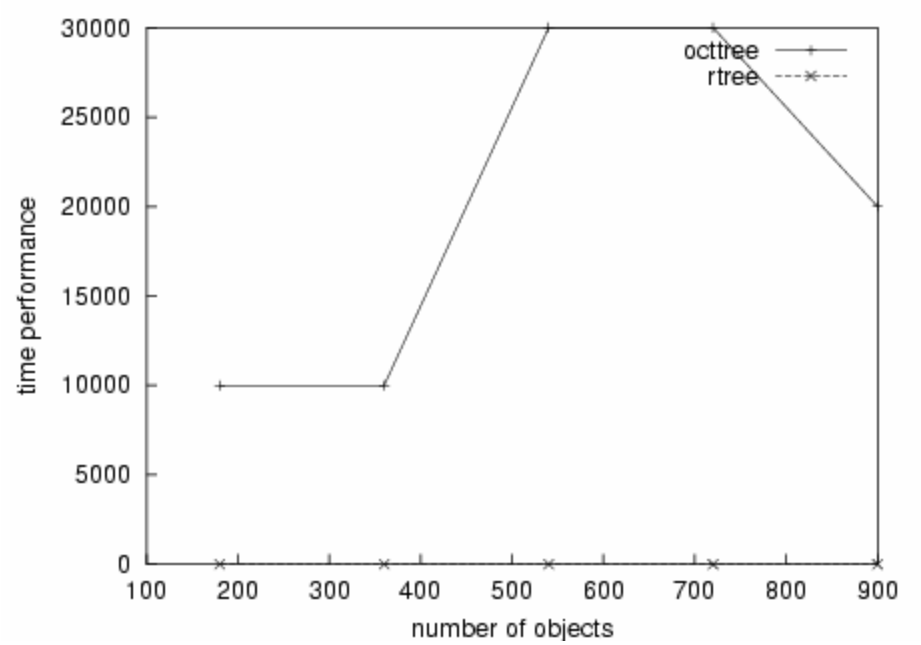


3d_single_insert_32

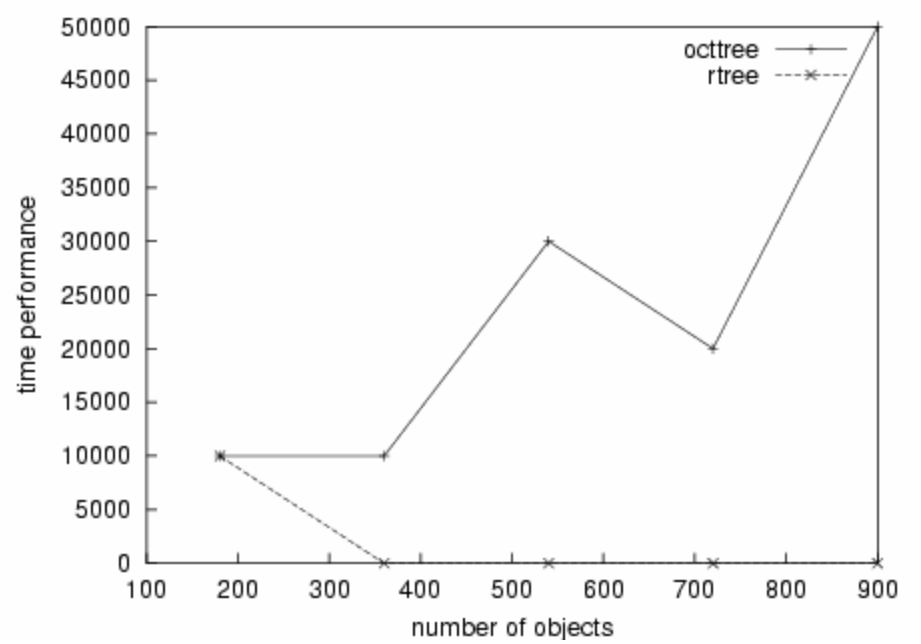


3d_chunk_insert_32

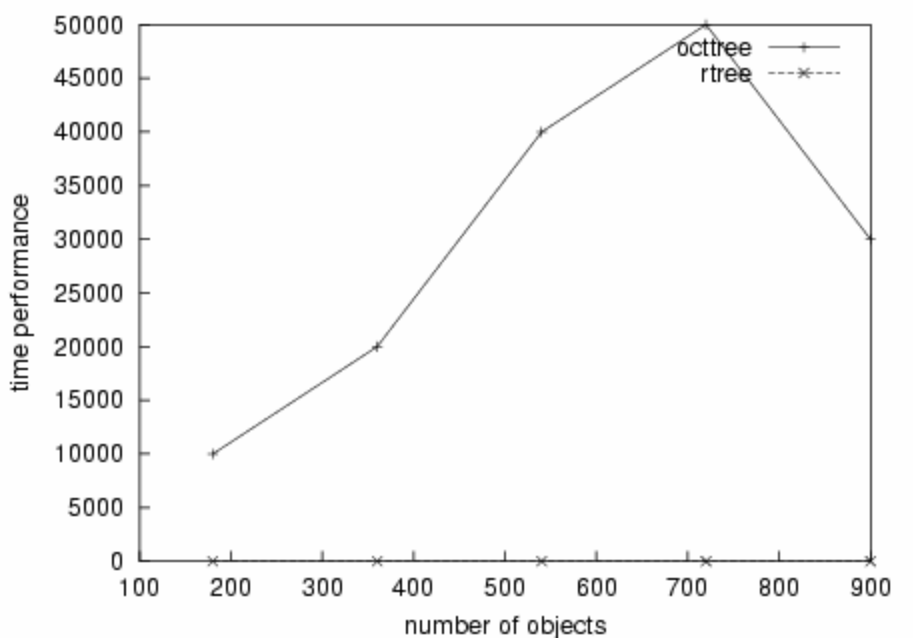
The same reason as we discussed in figure 2d_chunk_insertion_128 leads to the two peaks when data is 540 and 810.



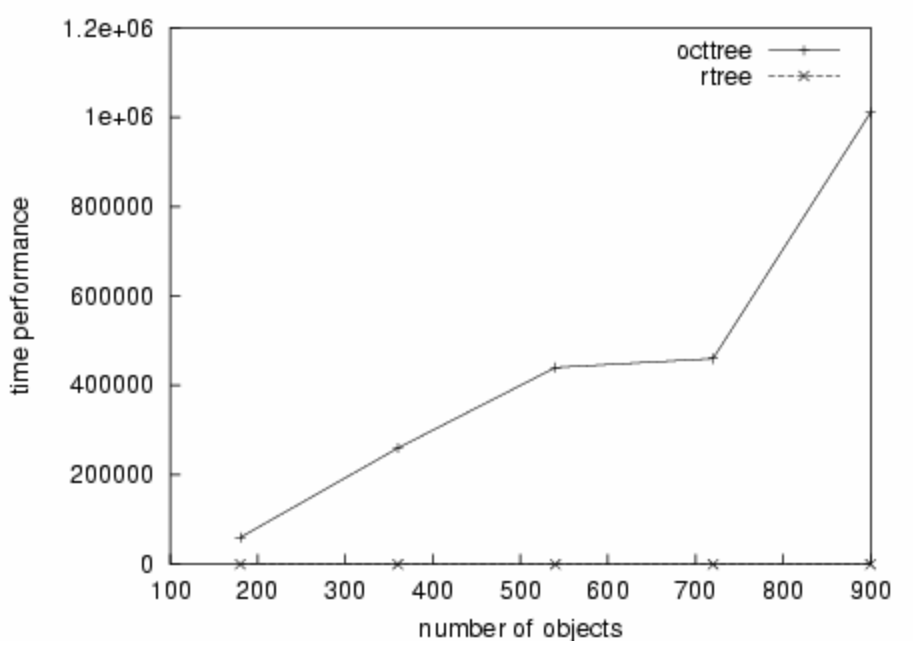
3d_search_0.1%_32



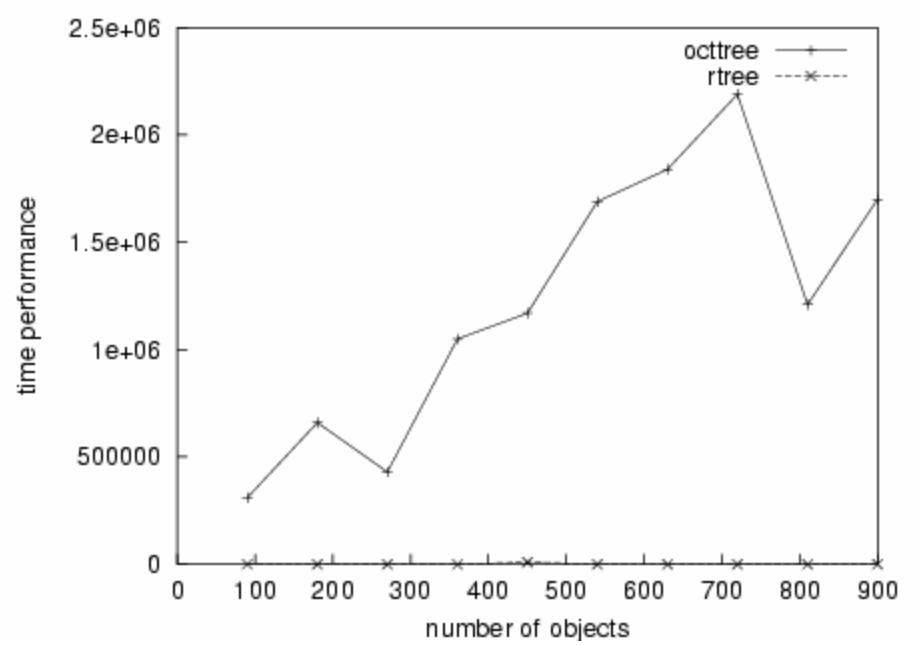
3d_search_1%_32



3d_search_10%_32



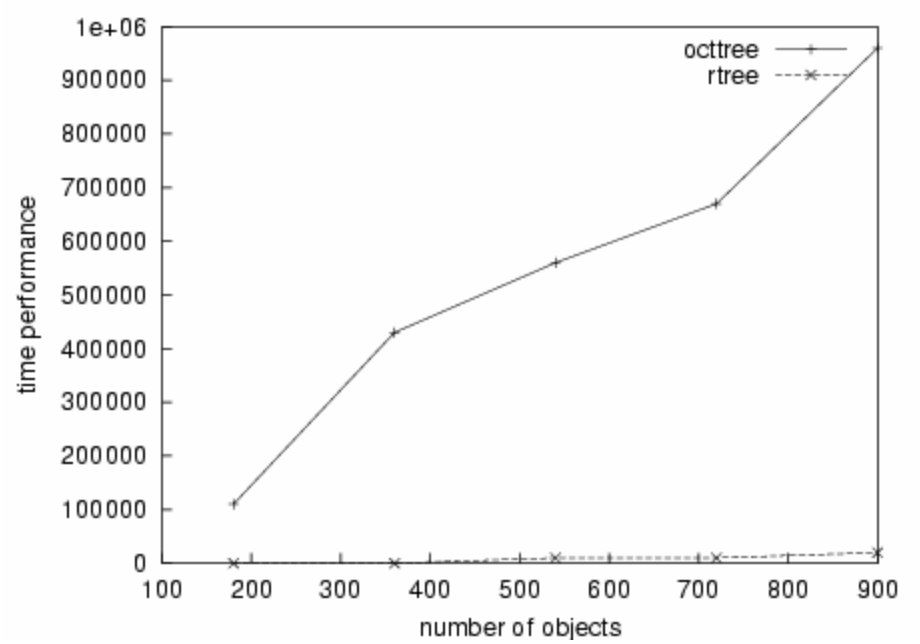
3d_single_remove_32



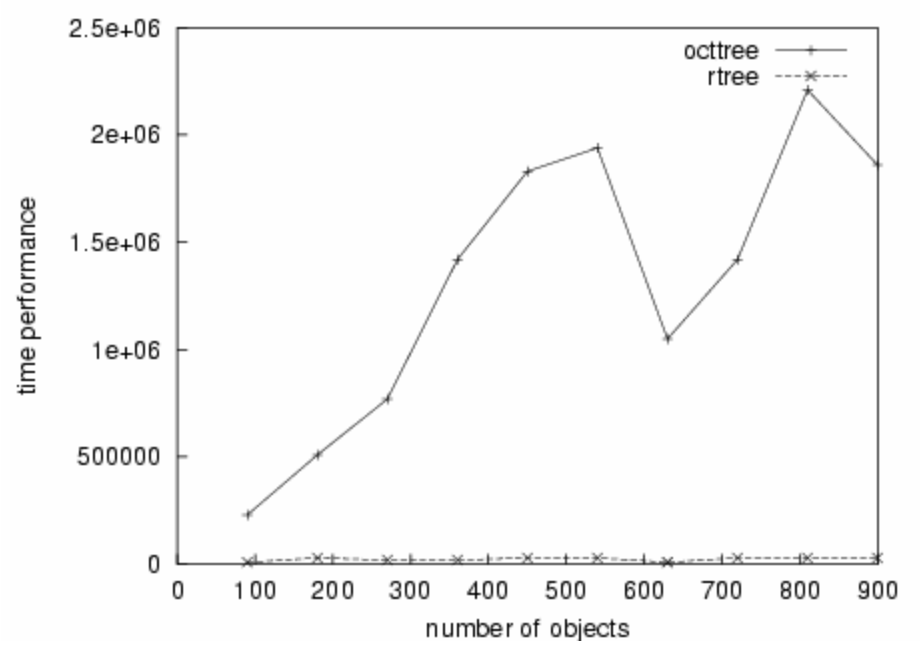
3d_chunk_remove_32

The same reason as we discussed in figure 2d_chunk_insertion_128 leads to the two peaks when data is 720.

3 Dimensional with Leaf Size 64

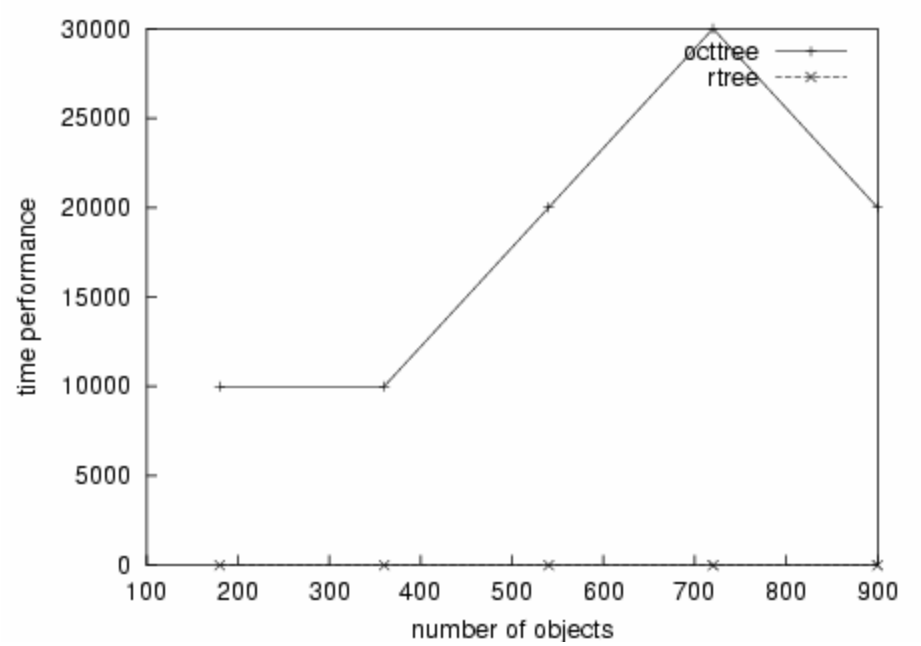


3d_single_insert_64

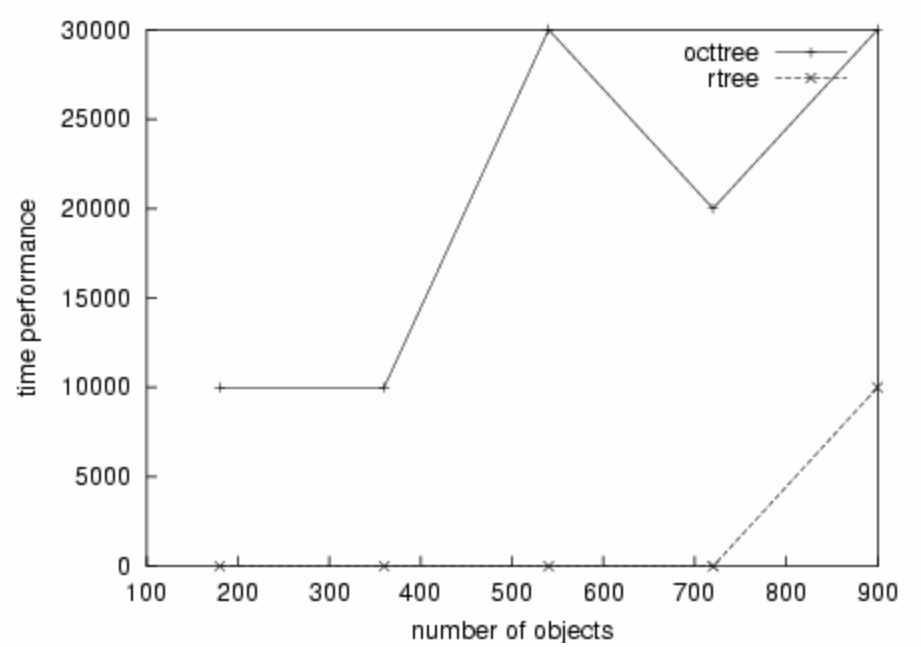


3d_chunk_insert_64

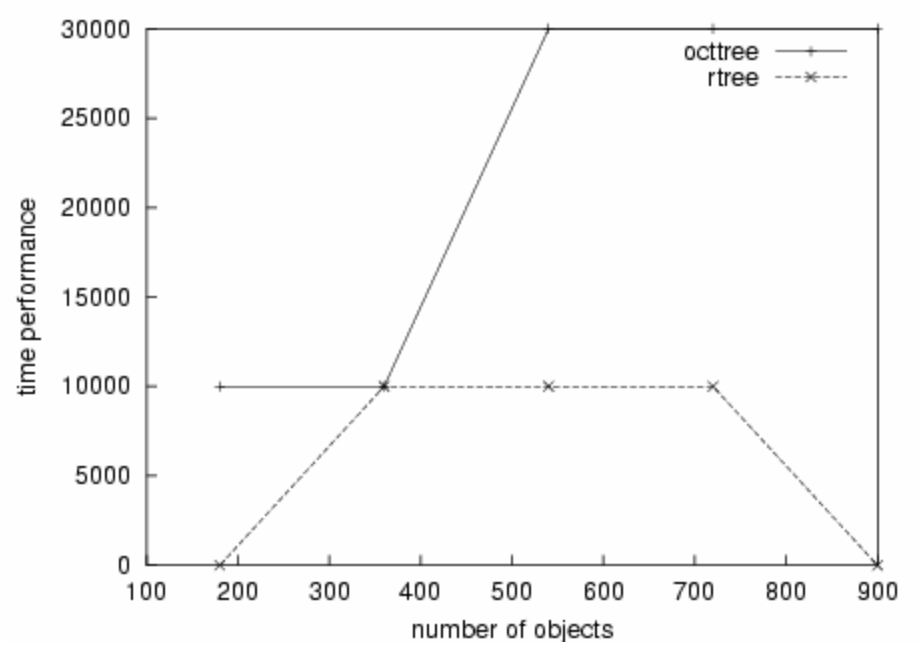
The same reason as we discussed in figure 2d_chunk_insertion_128 leads to the two peaks when data is 540 and 810.



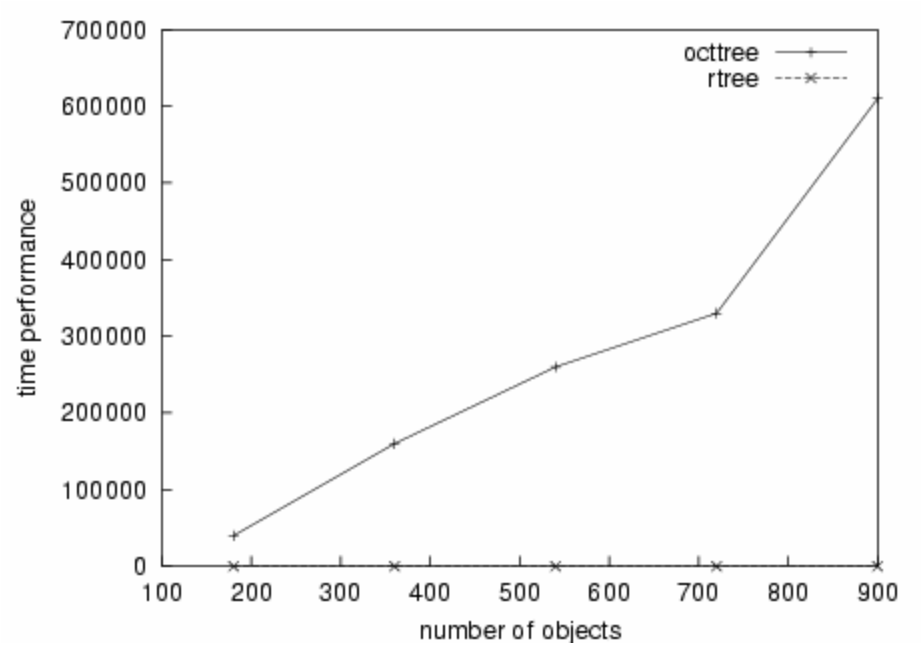
3d_search_0.1%_64



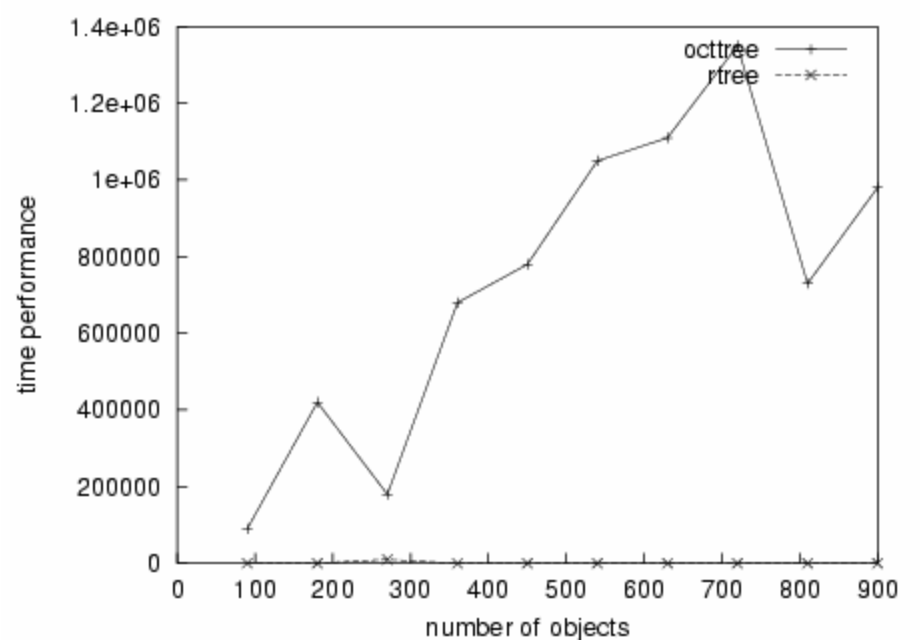
3d_search_1%_64



3d_search_10%_64



3d_single_remove_64



3d_chunk_remove_64

The same reason as we discussed in figure 2d_chunk_insertion_128 leads to the two peaks when data is 720.

6.1 Analysis of algorithm:

From the description of the algorithm in implementation details of our report, it is obvious that the more the data are, the longer the time performance for operations (insertion, range search and remove) is. Generally our experiment result reflected this except some noise.

6.2 Noise and solution:

There are two main sources of noise:

- We use the function clock() measure the time it takes the index tree to do operation. The granularity of result returned by this function is 10,000. So, the results we get are something like 0, 10,000, 20,000….It is not accurate enough.
- We did our experiment in bert.cs.uic.edu. Many students share the machine at the same time, this makes the measure of time not accurate.

For the second source of noise, we tried to do our experiment when there are few students use bert, such as at the holiday mornings.

For the first source of noise, we tried to do some operations, such as range search, for many times, so we can get more accurate result. But for the insertion or remove operations, we cannot use the same method, since such operation changes the tree.

Additionally, we generated ten ranges for the each range size (0.1%, 1% and 10%) to do the search experiment. We hope this can give us the average time for the search of the range size. But sometimes, we were in bad luck. In the figure 3d_search_10%_32, the time for oct-tree for the search operation of 900 data is less than that of 720 data. We think most of the ten search ranges are in sparse area of the special data.

The goal of our project is to compare the time performance of quad-tree (oct-tree) and R-tree. We don't mind the inaccurate result, as long as we can get conclusion from the results.

7. Conclusions

Based on the above experiments results, we can get such conclusion as: For the operation of insertion or remove, the time performance of 2D R-tree is better than the one of Quadtree. For the range search operation, they are similar. Sometimes, quad-tree performs better than R-tree. Sometimes, R-tree performs better than quad-tree. The time performance of 3D R-tree is better than the one of Octree. The reasons for such conclusion are:

Quadtree and Octree are space-driven structures, which are based on partitioning of the embedding space into rectangular cells, independently of the distribution of the objects (trajectories or routes). Whereas, 2D R-tree and 3D R-tree variant are data driven structures, which are organized by partitioning the set of objects, as opposed to the embedding space. The partitioning adapts to the objects' distribution in the embedding space. Such difference between the natures of these two indexing methods leads to the following facts:

- In R-trees, we use an MBR (minimal boundary range) to contain and represent a route or trajectory, whereas in quad-tree and oct-tree, we don't use MBRs. So when we perform insertion and deletion in a quad-tree or oct-tree, more time has to be cost

on calculating if a route or trajectory intersects with the existing cells. The reason is that for quad-ree or oct-tree we must calculate if every line segment of a route or trajectory intersects with the cells.

- Every internal node of R-tree has MAXLEAFNODE (In our experiment, MAXLEAFNODE can be 16, 32, 64 or 128) children. Whereas, each intern node has 4 children in quad-tree, and 8 children in oct-tree. So the depth of R-tree may be much lower than the depth of quad-tree or oct-tree, which leads to the inefficiency of quad-tree and oct-tree.
- In quad-tree or oct-tree, a trajectory or a route can be duplicated in neighbor cells. This increases the index size, decreases the access performance, and leads to a possibly expensive sort of the result for duplicate removal. While in R-tree, a trajectory or a route can only appear in a single leaf node.
- The structure of R-trees (2D and 3D variants), while keeping the tree balanced, adapts to the skewness of a data distribution. A region of the search space populated with a large number of objects generates a large number of neighbor tree leaves. In the case of quad-tree or oct-tree, mapping those spatial objects to a one-dimensional order (B^+-Tree) causes the clustering loss, i.e., some branches in the tree might be long, corresponding to regions with a high density of trajectories or routes, whereas others might be short. So, if a range searching is going along a shorter branch in a quad-tree or oct-tree than in an R-tree, it will run faster.

8. Acknowledgments

We thank Professor Trajcevski and Professor Wolfson for their valuable help, advice, and enriching presentations, which made it possible for us to implement the different algorithms.

9. References:

1. Spatial Access Methods, *Chapter 6 of the book Spatial Databases with application to GIS by P. Rigaux, M. School, and A. Voisard, published by Morgan Kauffman, 2002*
2. R-Trees A Dynamic Index Structure For Spatial Searching, *Research Paper by Antonin Guttman, University of California, Berkeley, 1984*
3. A Quadtree-Based Dynamic Attribute Indexing Method *by Jamel Tayeb, Ozgur Ulusoy and Ouri Wolfson, The Computer Journal, Vol. 41, No. 3, 1998*
4. Moving object databases: issues and solutions, *by O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang, International Conference on Scientific and Statistical Database Management, Proceedings / International Conference on Scientific and Statistical Database, 1998*
5. Quadtree and R-tree indexes in Oracle spatial: a comparison using GIS data *by Ravi Kanth V Kothuri, Siva Ravada, and Danial Abugov, ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA, 2002*
6. Specifications of efficient indexing in Spatiotemporal Databases *by Yannis Theodoridis, Timos Sellis, Apostolos N. Papadopoulos and Yannis Manolopoulos, ICDE, 1998.*
7. Storage and Retrieval of Moving Objects *by Hae Don Chon, Divyakant Agrawal, Amr El Abbadi, CIKM, 2000*