Comparison of R Tree, MVR Tree and TPR Tree

Anand Murugappan

College of Computing,
Georgia Institute of Technology, Atlanta, USA
E-mail: anandm@cc.gatech.edu

Abstract

Naive B or B+ trees are very good and efficient for indexing single dimensional data but not well suited for multidimensional data. Antonin Guttman of University of California, Berkeley proposed a novel scheme for indexing multidimensional data called R Trees. Since then several variants of the R tree have been developed for for different situations and work loads. R tree and its variants have several applications ranging from CAD Drawings to geographical databases to mobile computing. This report summarizes the background and results of the comparison experiments performed on R Tree, MVR (Multi-Version R) Tree and the TPR (Time Parameterized R) Tree as part of the CS6400 project.

1 Introduction

There are several applications which rely on efficiently managing multidimensional data. In the past people have mapped the multidimensional information into a single dimension and used the popular B+ trees and Hash tables for indexing such information. However B+ trees and ISAM are not generally efficient for multidimensional data and hash tables are not good for range queries. Antonnin Guttman proposed a novel spatial data structure called R Tree [1] which was meant to do exactly this task.

R Tree even though was very efficient for Spatial indexing was very generic and there was scope for it to be fine tuned based on the nature of the application domain. A whole suite of R Tree variants have come up ever since. For instance MVR Tree has been proposed keeping in mind multi-version drawings which are common in CAD systems [4], TPR tree has been proposed to address the scenario where the indexed objects are on the move, changing location continuously [5]. A survey of spatio-temporal access data structures can be found in [3].

The Figure 1 from [3] shows the timeline of various spatio-temporal access methods that have been discussed in literature. The diagram has been modified to include the MVR Tree and its variants. The red boxes have been added to indicate the data structures that are being compared in this report.

In this report the necessary background for the R Tree, MVR Tree and TPR Tree are described in Section 2. Then in Section 3 the design of the open source implementations of R Tree and its variants called SaIL (A Spatial Index Library for Efficient Application Integration) [2] which has been used for the comparisons is discussed. Section 4 then describes how to setup and run the experiments. Finally Section 5 summarizes the results - tests were performed on creation of the tree, query operations and storage.

2 Background

In this section the required background for understanding the results is presented. Each of the subsections cover important aspects of the spatial data structures and address questions about creating, querying, maintaining and applications of the tree.

2.1 R Tree

The R tree [1] was an innovative data structure built on top of the idea of the B+ tree. The R Tree represents data as intervals in several dimensions. Nodes in an RTree are represented as (I,tuple_identifier), where $I = (I_0, I_1, I_2 ... I_{n-1})$. I_i represents the span of the bounding box for the object in dimension i. Figure 2 represents the R Tree corresponding to the spatial distribution of objects (solid rectangles in this case) below it.

2.1.1 Applications

The R Tree can be easily applied to any spatial scenario (generally in 2D and 3D).

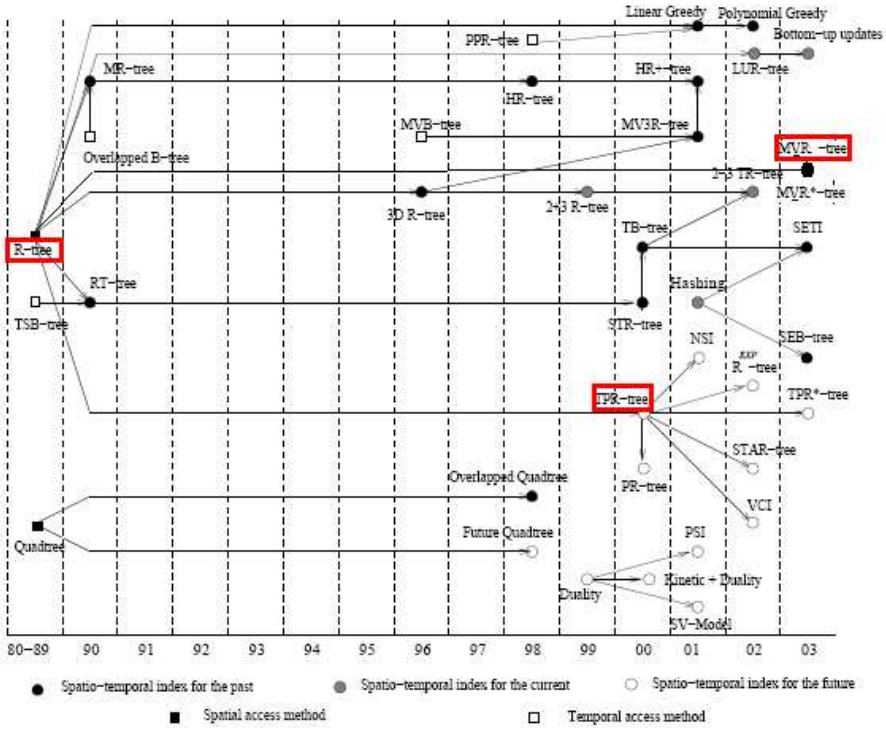

Figure 1: Survey of Spatio-temporal Access Methods [3]

2.1.2 Salient features

The following are some of the salient features of the R Tree:

- The R Tree is height balanced.
- Inserts/Deletes/Query operations can be intermixed
- If 'M' is the maximum number of entries in a R Tree then m $\leq$ M/2, where m is the minimum number of entries.
- Height of the R Tree is $\lfloor \log_m N \rfloor$ - 1
- Maximum number of nodes is $\lceil \frac{N}{m} \rceil + \lceil \frac{N}{m^2} \rceil + ... + 1$

2.1.3 Insert

The following summarizes the algorithm to insert a node into an R Tree.

```
------------------------------
Insert (Record E)
{
 Node L = ChooseLeaf(E);
 if ( L has enough space )
 {
  insert E into L
 }
 else
 {
  LL = SplitNode(L);
  insert E into L or LL
 }

 // Propagate changes upward
 AdjustTree(L,[LL]);
 if (root got split)
 {
  // Grow tree
  Add a new root and make it
  point to the 2 subtrees
 }
}
------------------------------
Node ChooseLeaf(Record E)
{
 Node N = root;
 while (N is not leaf)
 {
   Choose entry F such that
   enlargement of rectangle
   needed in minimum
   N = F.node;
 }
 return N;
}
------------------------------
AdjustTree(Node first, Node second)
{
 Node N = first;
 Node NN = second;

 while ( N is not the root)
 {
   Let P be the parent node of N
   Adjust N's entry in P to tightly
   enclose the rectangles in N

   If NN exists (that is there
   was a split before) try to
```

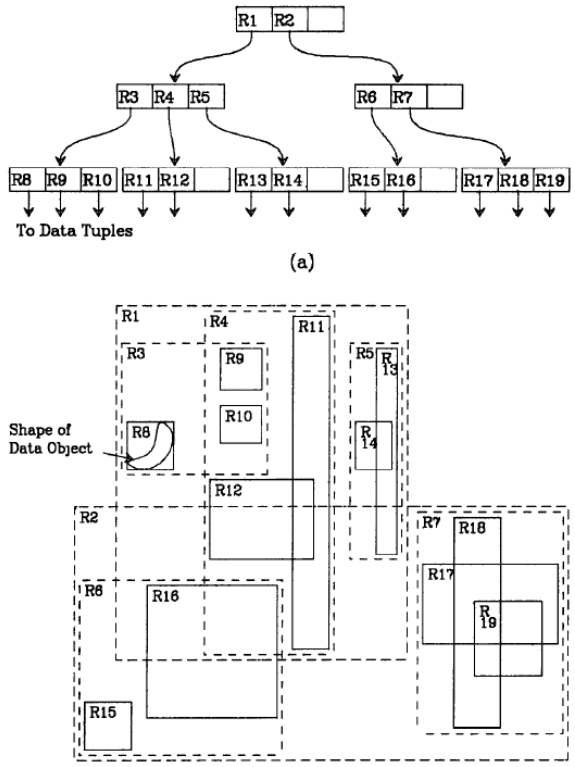

Figure 2: R Tree [1]

```
   accommodate an entry for NN into
   P.

   If there is not enough space
   call SplitNode();
 }
}
------------------------------
// The node should be split such that
// it would be highly unlikely that
// both the new nodes would be searched
// on subsequent queries
SplitNode()
{
 // There are 3 algorithms available
 // for splitting node
 a. Exhaustive algorithm
 b. Quadratic algorithm
 c. Linear algorithm
}
------------------------------
```

Details about each of these algorithms can be found in [1].

2.1.4 Delete

The following summarizes the delete algorithms.

```
------------------------------
delete(Node E)
{
 // R is the root node
 Node L = FindLeaf(R,E);
 Remove E from L;
 CondenseTree();

 If root node has only
 one child then child is
 the new root.
}
------------------------------
Node FindLeaf(Node T, Record E) {
 if (T is a leaf)
 {
   check every entry in T to see
   if it matches with E.
   If match found return it
 }
 else
 {
   For every entry in T see if
   the area of E overlaps with it.
   If so return FindLeaf(T.entry,E)
 }
}
------------------------------
```

2.1.5 Query

Querying is rather straightforward. Given an area E the idea is to start from the root and look at every entry such that the area intersects. If so repeat the same task with the child pointed to by the entry as the root till a leaf node is hit. The leaf node entries are then compared with the area to identify whether there is match or not.

In general several types of queries are possible. The most common ones are:

- Intersection Queries
- K-Nearest Neighbor queries
- Self join queries

2.1.6 Update

The simplest update procedure is to delete the updated item from the R tree, modify and then reinsert.

2.2 MVR Tree

The concept of MVR Tree was proposed by researchers interested in applying the R Trees to CAD drawings. In a CAD system several versions of the drawing exist at the same time. These drawings differ to a very small extent from one another. R Trees are generally used to represent these drawing but since they differ only to a small extent there is a huge amount of duplication. The use of R Tree to store is the drawings is as shown in the Figure 3. The motivation behind MVR Trees (Multi-Version R Trees) is to eliminate this storage of duplicates and hence conserve space.

The crux of the idea of MVR tree is to share as much common nodes as possible between versions. The following insert and delete algorithms illustrate in greater detail how this is achieved.

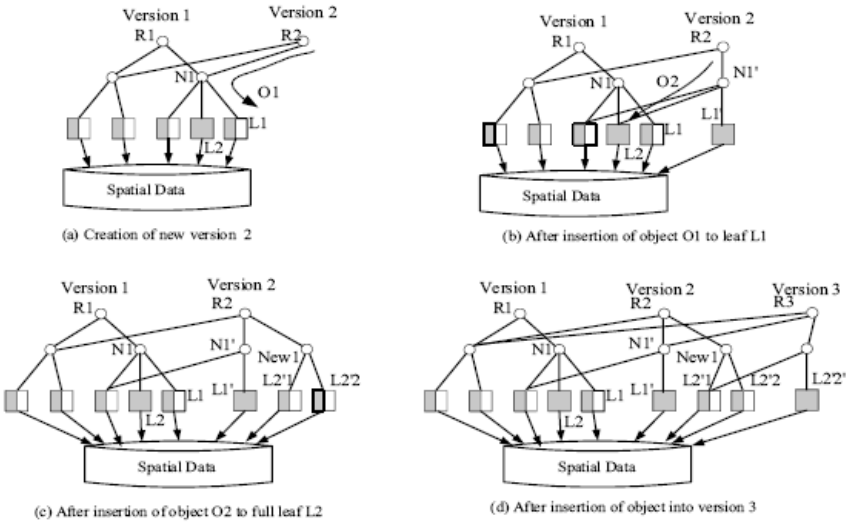

Figure 4: MVR Tree insert [4]

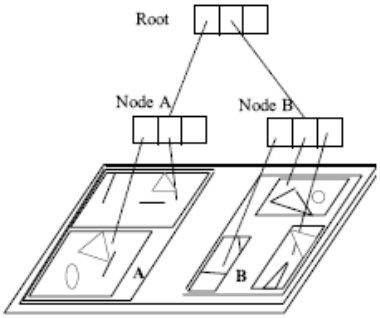

Figure 3: R Tree used to represent drawings [4]

2.2.1 insert

```
------------------------------
CreateNewVersionFrom(Vk)
{
 Create new root Rn corresponding to
 new version

 Set the child pointers of Vk's root
 to Rn

 Insert(Object);
}
------------------------------
Insert(Vn,Ox)
{
 Let R be the root of Vn

 Tracing from R find the Leaf L
 where Ox should be inserted

 Copy the nodes along the path
 into Vn.

 Insert Ox into Leaf L.

 If L is not full DONE

 If L is full then create L2 and
 move objects from L1 into L2.

 Adjust tree // same as with R Tree
}
------------------------------
```

2.2.2 Delete

```
------------------------------
Delete(Ox)
{
 Let R be the root corresponding
 to version Vn.

 Start at Vn and find the Leaf L
 which contains Ox.

 if (L is not shared)
 {
   if (underflow will not result on
       deletion)
   {
    Remove Ox;
   }
   else
   {
    Remove Ox;
    Remove all entries of L;
    Reinsert all entries of L;
   }
 }
 else
 {
   Copy all nodes along path into Vn

   Perform deletion as if L were not
   shared
 }

 AdjustTree();
}
------------------------------
```

2.2.3 Query

Querying is exactly same as that of an R Tree except that one has to just start at the root corresponding to the version.

2.3 TPR Tree

The idea of TPR (Time Parameterized R) Trees was proposed to answer queries on moving objects [5]. TPR trees are very well suited for mobile systems where queries on moving objects are common. In fact TPR tree brings into picture a new type of queries called moving queries. Moving queries are queries where the range of the queries changes with time. The TPR tree specifically is designed to answer present and future queries. The crux of the idea of TPR tree is to use MBR (Minimum Bounding Rectangles)to enclose objects of interest with time as a parameter. Moreover what objects are to be kept together in the index is not decided just based on their current distances (like in the case of R Tree) but instead based on the cumulative distance over a period of time. This makes sure objects traveling with largely different velocities don't end up together. This is very important because if the size of the MBR becomes very large there will be a greater likelihood of region overlaps which would greatly increase the query time.

The index structure of TPR trees is as follows:

- Leaf Node - The leaf nodes contain the position of the object (represented in terms of a reference position and velocity vector) and pointer to the moving object
- Index Node - The index node contains the minimum bounding rectangle (MBR) and pointer to the subtree.

The Figure 5 illustrates a case where the R Tree would perform very bad compared to a TPR tree.

The future distance is determined using the following formula:
$x = x(t_{ref}) + v(t - t_{ref})$
where,
t_{ref} represents the time at the time of reference.
v represents the velocity of the object
$x(t_{ref})$ represents the location along the dimension of the object at time t_{ref}

2.3.1 Insert

Insertion into the tree is very similar to the R* tree insertion algorithm but instead of considering only the area of the MBR the following integral is used. The integral represents the sum total of all the area of the MBR for the time interval for which the tree is likely to be queried.
$\int_{t_1}^{t_1+H} A(t)dt$

By using this formula the insertion algorithm does not put objects that are likely to move very far from each other in the future into the same node there by being in a position to efficiently answer future queries.

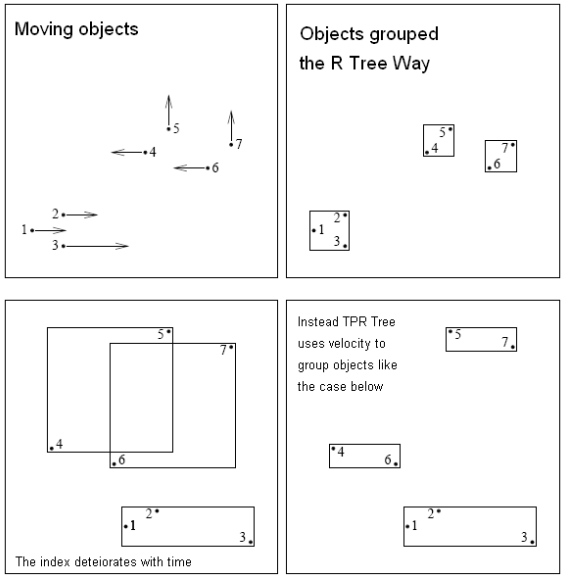

Figure 5: R Tree vs TPR Tree for indexing moving objects [5]

2.3.2 Delete

Deletions are performed exactly like in the case of R* tree. If a node gets under full it is delete and its elements reinserted.

2.3.3 Update

Updating the tree is as simple as delete the node and reinsert. However an interesting thing in the TPR tree proposed in [5] is that one can opportunistically use the update operation to update the tree. One must note that the MBRs in TPR trees are such that they never shrink so the rectangles can get really large resulting in inefficiencies. These bounding rectangles that are updated during update of a data object are referred to as "Update time bounding rectangles". Figure 6 illustrates the MBR at load time and update time.

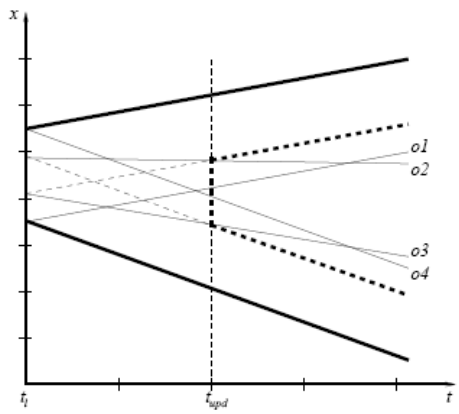

Figure 6: Load time(bold) and Update time (dashed) bounding intervals for 4 objects [5]

2.3.4 Query

Queries in a TPR tree can be either:

- Time Slice Queries
- Window Query
- Moving queries

Moving queries are the most generic form with the first and second being special cases of Moving Queries.

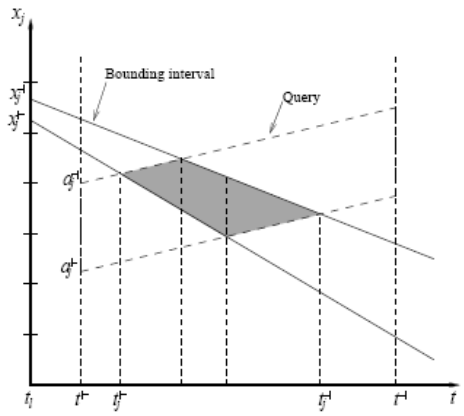

Figure 7: Querying in a TPR tree [5]

Figure 7 illustrates how the objects matching the query are found. The area of intersection represents portion of the results for the query.

3 Open Source Implementations used for the experiments

SaIL (A Spatial Index Library for Efficient Application Integration) [2] is an open source framework for implementing R Tree and its variants. Marios et al. built SaIL with the objective that programmers should be able to use a consistent interface to experiment with different R Tree implementations. The c++ version of SaIL currently has implementations of the R Tree, MVR Tree and the TPR tree. All the results listed out in this report are based on the experiments conducted on this system.

The Figure 8 gives an overview of the design of SaIL. The following are the 4 main components of SaIL.

- Core Library toolkit
- Storage manager toolkit
- Spatial Index toolkit
- Concrete implementations - R Tree, MVR Tree and TPR tree

One can find the finer details in the paper [2].

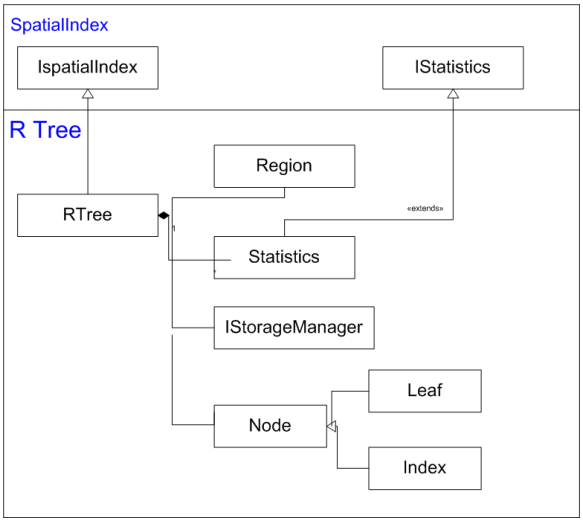

Figure 9: Implementation of R Tree on SaIL

The class diagram for implementation of the R Tree can be as seen in Figure 9.

To visualize R Tree Marios et al. have also built what they call the R Tree visualizer. Snapshots from the R Tree visualizer can be as seen in Figure 10.

4 Installing and running

In this section describes how to install and run the system to reproduce the results mentioned in the section below:

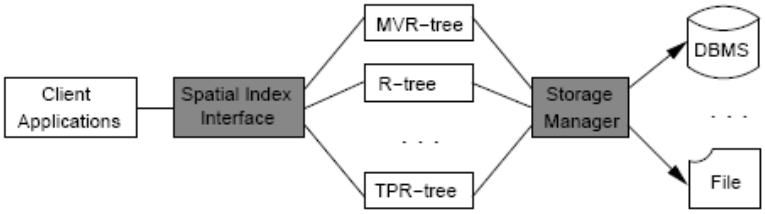

Figure 8: SaIL Design Overview [2]

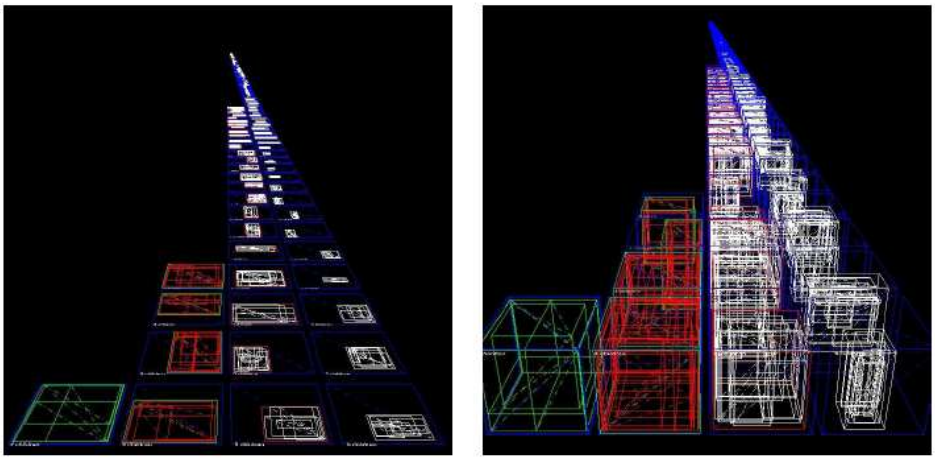

Figure 10: Visualizing 2D and 3D R trees [2]

- Visit http://www.cs.ucr.edu/ñarioh/spatialindex/ and install the latest C++ version of the SaIL code.
- Follow the instructions in the README and install the SaIL code
- Next extract the experiment scripts (contents in the .tar.gz file) that one should be able to find with this document into the root folder. The directory should be at the same level as the ”src” and ”regression-test” folders.
- Now browse through the subdirectories of the ”results” folder and run the scripts. Each folder has a perl script called ”getStats.pl”. Just type ”perl getStats.pl” (make sure you have perl installed) and one should see the results in for various sizes of inputs in a few minutes. To understand what the columns are just take a quick look at the perl script. Plot them along with results from another test on a graph and one should see the results explained in the next section.

  All tests were run on the jedi cluster at the college of computing, Georgia Tech. If one has access it is highly recommended the tests be run on the same server.

  The following are the available results that can be collected:

  – R Tree
    * bulkload
    * insert
    * query-10NN
    * query-intersection
    * query-selfjoin
    * storage
  – MVR Tree
    * insert
    * query-intersection
    * storage
  – TPR Tree
    * insert
    * query-10NN
    * storage

  One might be wondering why some of the test cases (esp. in the case of query) are not present uniformly under all the trees. This is the case because the current implementation of SaIL does not fully support MVR Tree

and the TPR tree. For instance, self join is yet to be implemented for MVR Trees. However for most practical purposes and for the task of this project this implementation is by far the most robust one found on the internet.

5 Results

5.1 Test Scripts

As explained in the section above the wrapper scripts called getStats.pl were written for each of the properties of the trees and then plotted using Microsoft Excel. The results are presented here with the conclusion section discussing about when to use the different trees.

Test Environment: All tests were run on the jedi cluster at the college of computing, Georgia Institute of Technology.

The SaIL library (See Class diagram) has features to collect statistics for the various operations. The wrapper scripts mentioned above call the various implementations (like RTreeLoad, TPRTreeLoad etc) with different parameters and extract out the statistics at the end. Finally the statistics are plotted.

In most cases the statistics includes I/O operations or number of nodes which are independent of the processing power or memory of the machine on which the tests were run. So one should be able to run it on any other machine and irrespective of the hardware should be able to reproduce results similar to the ones shown below.

Parameters: To maintain consistency and fairness in the tests all tests were run with node capacity 20 records and fill factor 80%. The variation of performance with number of records per node is considered at the end of the results section. The variations in these parameters reflect in the absolute values but the trends continue to be the same.

5.2 Data Sets

The implementation of SaIL also ships with a regression test suite which contains random data generators. The regression test suite was designed to verify that the implementation is indeed correct. The tests were conduced using the data sets generated by the in built generators. All spatial generators generate operations and number of objects with coordinates between 0.0 to 1.0. The following portion of this subsection is dedicated to give the reader an idea of the kind of data sets used.

Simple Spatial Data Set: The sample simple spatial data set generated is as shown below.

```
1 913 0.685875 0.283454 0.7749 0.333149
0 144 0.32959 0.79233 0.400701 0.873002
1 144 0.512394 0.935941 0.514928 0.965057
0 322 0.478646 0.202452 0.54537 0.208425
1 322 0.251812 0.108343 0.343164 0.142574
0 389 0.109943 0.416122 0.190364 0.488834
1 389 0.212017 0.250636 0.275513 0.27918
0 9 0.429878 0.43118 0.492774 0.474289
1 9 0.15937 0.315774 0.25139 0.352855
0 451 0.093956 0.310191 0.113388 0.343034
1 451 0.308383 0.237348 0.33494 0.323339
0 434 0.378524 0.190233 0.473331 0.19241
1 434 0.310807 0.959035 0.328547 1.03451
2 9999999 0.741349 0.790037 0.751349 0.800037
```

The first column represents the type of operation. 0 for delete, 1 for Insert and 2 for query. The second column can be ignored for all practical purposes. The rest of the 4 values indicate the rectangle enclosing the data object (x1,y1) and (x2,y2).

Multi Version Data Set: The Multi version data set is as shown below:

```
6 1 1294 0.978898 0.324339 1.01066 0.414037
6 0 1231 0.927296 0.985838 1.00616 0.993266
6 1 1295 0.386384 0.548925 0.443156 0.577718
6 0 0 0.675697 0.484229 0.764176 0.581714
6 1 1296 0.442786 0.643559 0.505081 0.686119
6 0 790 0.86099 0.793307 0.88683 0.848145
6 1 1297 0.16896 0.355582 0.203154 0.437757
6 0 1000 0.339026 0.782304 0.357765 0.841281
6 1 1298 0.297875 0.804416 0.314175 0.868983
6 0 325 0.0753029 0.62399 0.148757 0.679696
6 1 1299 0.920791 0.34508 0.967233 0.34518
6 2 9999999 0.0509074 0.559448 0.0609074 0.569448 1 3
6 2 9999999 0.437958 0.195006 0.447958 0.205006 4 6
6 2 9999999 0.387668 0.421674 0.397668 0.431674 1 3
7 0 1003 0.171737 0.980143 0.239408 0.994463
7 1 1300 0.759219 0.77254 0.846131 0.774028
7 0 602 0.634504 0.151635 0.724173 0.221382
7 1 1301 0.917315 0.981795 1.00494 1.0629
```

This data set is very similar to the normal data set shown previously. The only difference however is that there is an additional column (the very first one) which represents the version to which this operation and data object should be applied to. So in the example the last line represents insertion (1) into the 7th version of the data (diagram) an object represented by the rectangle (0.917315,0.981795) (1.00494,1.0629). The query operation however has a few additional entries at the end indicating additional versions on which the query is to be performed.

This data set is generally best suited for the MVR Tree.

Moving Objects Data Set: A sample moving objects data set is as shown below:

```
687 1 9 9 1 0.933175 0.00908294 1 0.690947 0.00946691
897 0 9 1 1 0.340114 0.00355914 1 0.96442 0.00493187
897 1 9 9 1 0.233717 -0.000566615 1 0.207758 0.0110022
75 0 9 0 1 0.389812 -0.00624552 1 0.54287 0.0136691
75 1 9 9 1 0.333602 -0.0053619 1 0.665892 0.0036609
767 0 9 0 1 0.72681 0.0126859 1 0.690544 0.00877249
767 1 9 9 1 0.840983 -0.0004696 1 0.769497 0.0154046
9999999 2 20 22 1 -0.0850889 0.100428 1 0.419823 0.55049
9999999 2 17 23 1 0.304962 0.469942 1 0.209014 0.324743
```

The important fields for insert/delete operations are Object id - first field, Operation - second field (0-Delete,1-Insert), Location (x) - fifth field, Velocity (x) - sixth field, Location (y) - Eight field, Velocity (y) - Ninth field.

The important fields for the query operations are Operation (always 2 for Query) - second field, time range - third and fourth fields, x dimension ranges - sixth and seventh field and finally the y dimension ranges - ninth and tenth field.

This data set is best suited for the TPR tree.

5.3 R Tree Visualization

Operation	ID	x1	y1	x2	y2
1	818	0.1	0.1	0.11	0.11
1	818	0.2	0.2	0.21	0.21
1	818	0.7	0.2	0.71	0.21
1	818	0.5	0.5	0.51	0.51
1	818	0.2	0.7	0.21	0.71
1	818	0.8	0.8	0.81	0.81
1	818	0.4	0.8	0.41	0.81
1	818	0.2	0.9	0.21	0.91

Table 1: R Tree visualizer input

Figure 11 shows the results of visualizing the R Tree generated with input in Table 1. Operation value 1 means INSERT and 818 for ID can be ignored. The objects are enclosed within a rectangle whose upper left coordinates are (x1,y1) and lower right coordinates are (x2,y2). Coordinate values can vary from 0.0 to 1.0.

Note: The RTree Visualizer is also available at the same URL as the SaIL library. However the C++ SaIL library and the RTree Visualizer are not compatible. To use the Visualizer feed the same input to the Java version of the library and then pick up the tree.idx and tree.dat files to be used with the Visualizer.

As can be seen from the figure, there are 2 levels. The first level (root) has 2 children (represented by 2 boxes) which point to the elements in the next level. Each of the children have 4 pointers (represented by the tiny boxes within the big white rectangle). The pointers point to the actual data objects mentioned in Table 1.

5.4 Insert

Figure 12 compares the R Tree, MVR Tree and TPR Tree based on the number of I/O operations needed to insert objects. As can be seen it is fairly obvious that with increase in the number of inserts the number of I/O operations increase. The inserts in the MVR Tree are least expensive. This is because the MVR Tree does not just represents 1 tree but multiple trees [See Figure 4]. As a result the MVR Tree spans horizontally

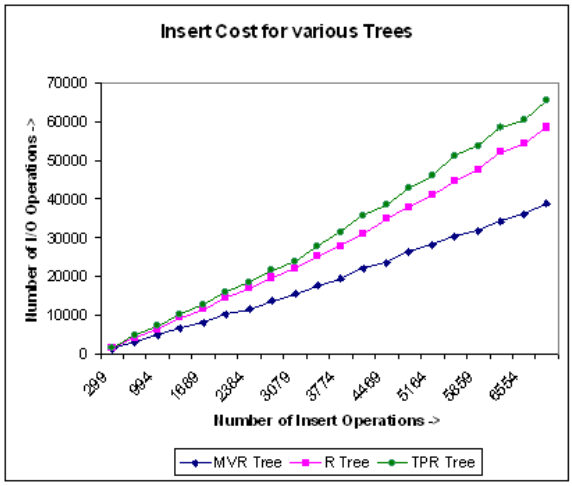

Figure 12: Comparison of insert costs

rather than vertically like the other trees and hence the insert cost is reduced for the same number of insert operations. The TPR tree however costs a little bit more because it is not always possible to combine the different objects together because of the modified grouping technique. Even though the R Tree seems to perform slightly better in terms of insertion it does not necessarily imply that R Tree is better because this insertion experiment represents insertion only at the time t_{ref}.

5.5 Creation of R Tree

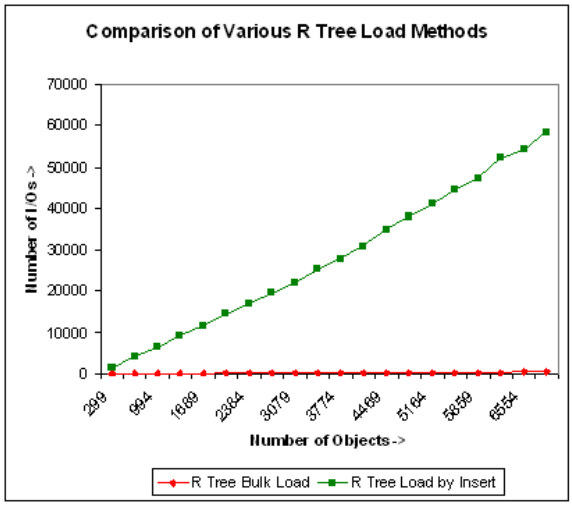

Figure 13: Comparison of naive loading using inserts vs bulk loading in a R Tree

Figure 13 is not a comparison of the 3 trees in any way but has been presented because it highlights the significance of bulk loading. It compares the I/O costs for

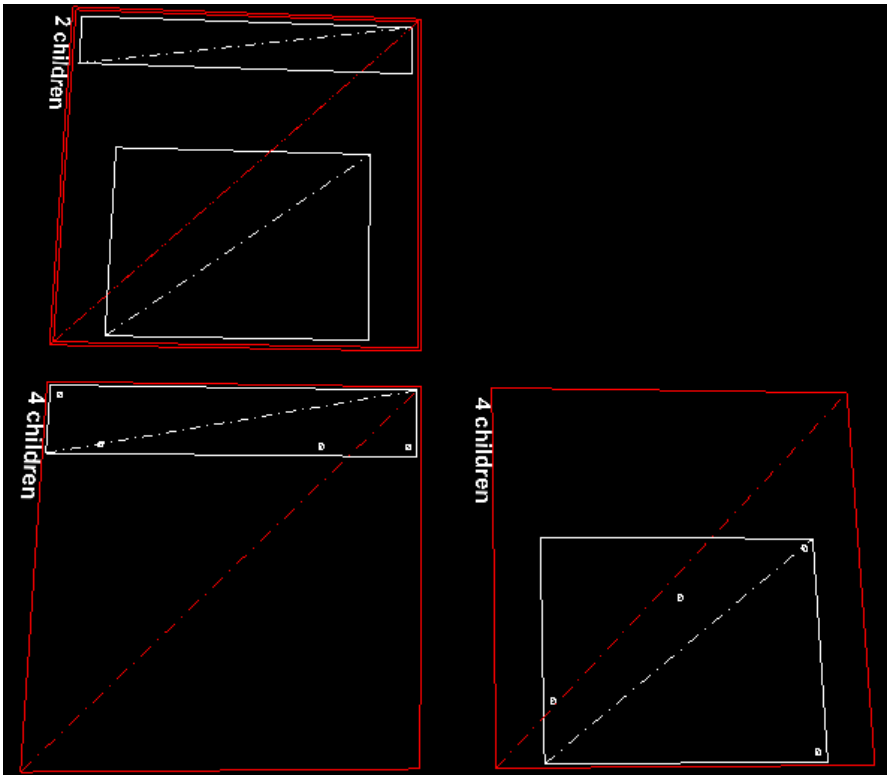

Figure 11: R Tree visualization for a simple input

Bulk loading an R Tree vs loading the R Tree with a sequence of insert operations [Simple Spatial Data Set]. As expected the bulk load performs way better than the load via insert. Load via insert increases linearly with very high slope as the number of objects increase but bulk load on the other hand hardly costs anything. The worm almost overlaps with the X axis emphasizing the importance of exploiting the concept of bulk load.

Note: This should apply not just to the R Tree but to MVR and TPR tree also. Unfortunately the current implementation of SaIL does not support bulkloading for MVR and TPR trees so at this moment it is impossible to verify this.

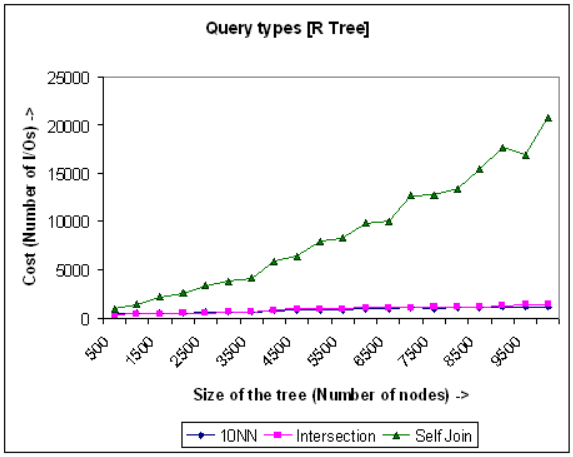

Figure 14: Comparison of various queries on the R Tree

5.6 Query Comparison on R Tree

The Figure 14 compares the costs of the different types of queries implemented for the R Tree. This graph again is not a comparison the 3 types of trees but considers only the R Tree. The interesting thing to note is that in this particular implementation the cost of 10NN query is similar to the to the cost of the intersection query. However the selfjoin query is a lot more expensive.

5.7 Storage Cost

The Figure 15 compares the storage cost of the MVR Tree and the R Tree [Multi Version Data Set]. The results show that the MVR tree lives up to what it was designed for. The storage cost (measured in terms of number of nodes) is way lesser for the MVR Tree. The storage cost of the R Tree increases linearly with a rel-

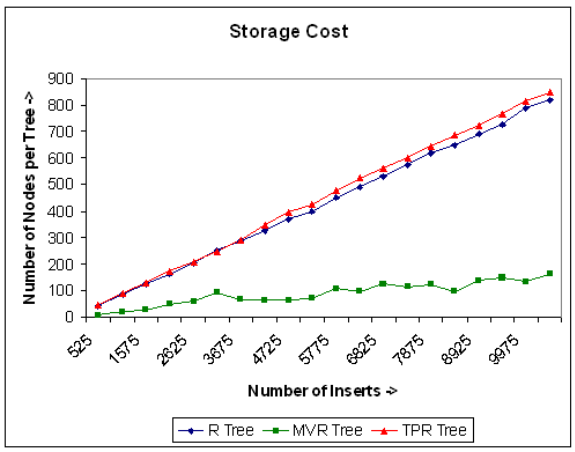

Figure 15: Comparison of storage cost as nodes per tree

atively larger slope. The MVR tree also grows linearly with increase in objects being inserted but the slope is gradual. Another interesting thing to note is that the TPR tree storage cost almost overlaps with the R Tree. So using a TPR tree does not help reduce storage. The TPR tree's claim to fame is not storage cost but performance and hence this is as expected.

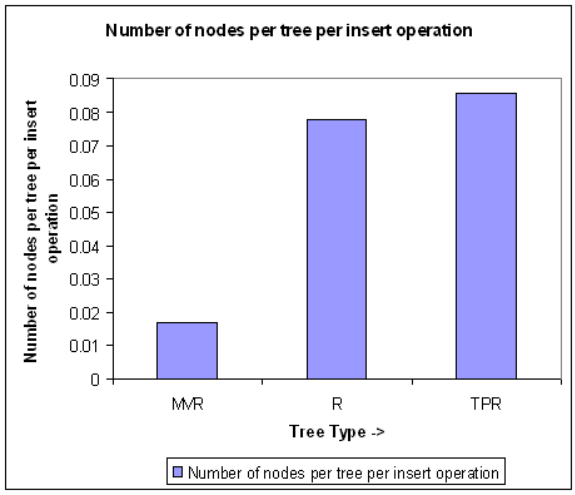

Figure 16: Comparison of storage cost as nodes per tree per operation

The Figure 16 is similar to the Figure 15 but instead represents the cost per operation. The values here are representative of the slopes of the worms in the Figure 15.

5.8 Queries [10NN]

The SaIL package implements 10NN (10 Nearest Neighbor) queries for the R Tree and the TPR Tree

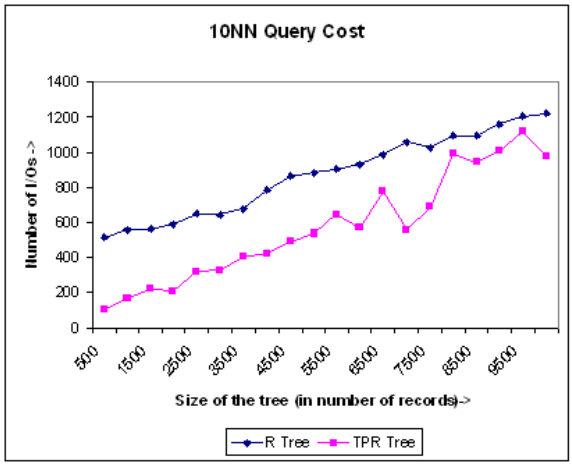

Figure 17: Comparison of 10 Nearest Neighbor queries on R Tree and TPR Tree

[Moving Object Data Set]. The implementation also exists for MVR Tree but in my opinion it does not work. For all requests it returned a read hit of 1 (which is definitely not correct). So the graph in Figure 17 compares the query costs for the TPR vs R Tree. As can be seen the TPR Tree out performs the R Tree. The TPR tree worm is a bit wavy. This is because the results were taken on only one run. Averaging it out a few extra runs should be able to produce a smooth worm but in any case it is very evident that the TPR tree performs way better than the R Tree in this aspect for this particular data set.

5.9 Query [Intersection]

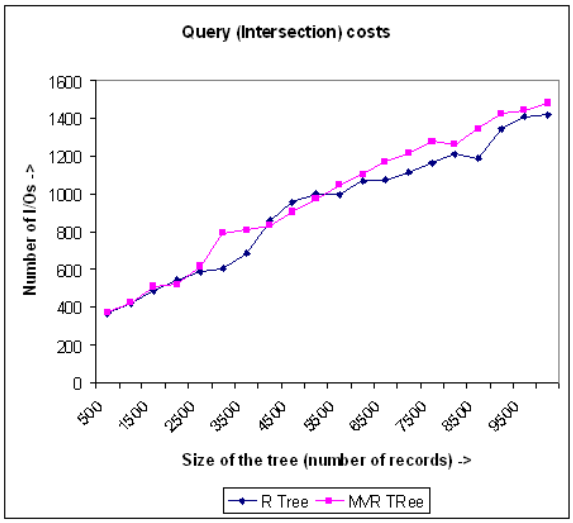

Figure 18: Comparison of Intersection queries on R Tree and MVR Tree

The Figure 18 compares the number of IO operations to answer intersection queries on the R Tree vs MVR Tree. The 2 worms are almost overlapping. This indicates that the MVR structure does not necessarily benefit query operations when compared to the R Tree approach.

5.10 Effect of Changing the number of records per node

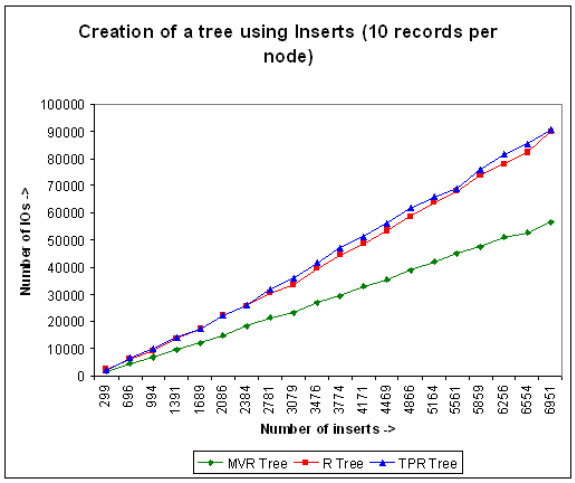

Figure 19: Creation of R Tree using inserts (10 records per node)

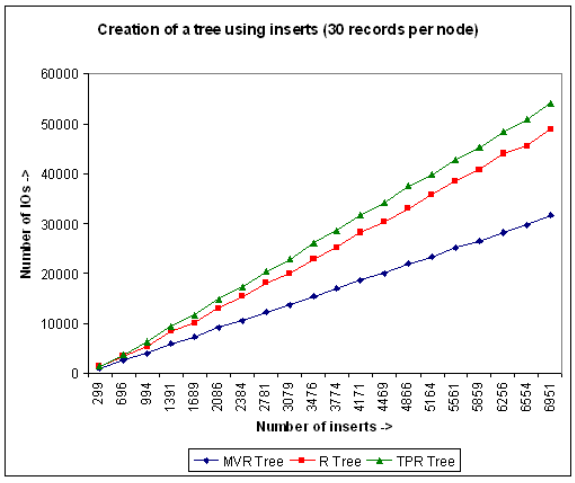

Figure 20: Creation of R Tree using inserts (30 records per node)

Figure 12 compares the creation of the 3 types of the trees via insertion with 20 records per node. However Figure 19 and 20 compare the same task but with 10 records per node and 30 records per node respectively. As can be seen the trends look the same in all the 3 cases. However the absolute values are different. With 30 records per node the number of IO operations reduces. So more records per node will reduce the IO operations but the trends will still be the same. TPR Tree and R Tree creation via insert are considerably worse than the MVR Tree in this implementation.

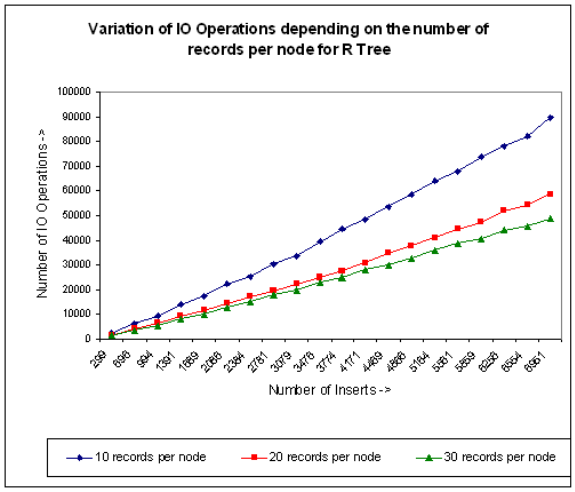

Figure 21: Effect of number of records per node in creation of an R Tree

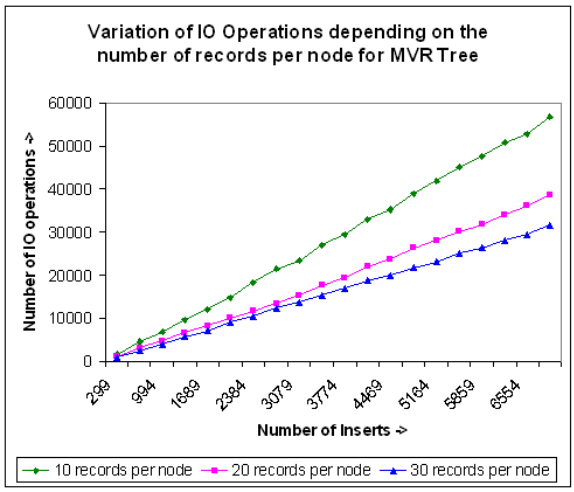

Figure 22: Effect of number of records per node in creation of an MVR Tree

Figures 21, 22 and 23 compare the relative costs of varying the number of records per node on the number of IO operations on the creation of the tree. As has been discussed above the growth in still linear but the cost reduces with increased number of records per node.

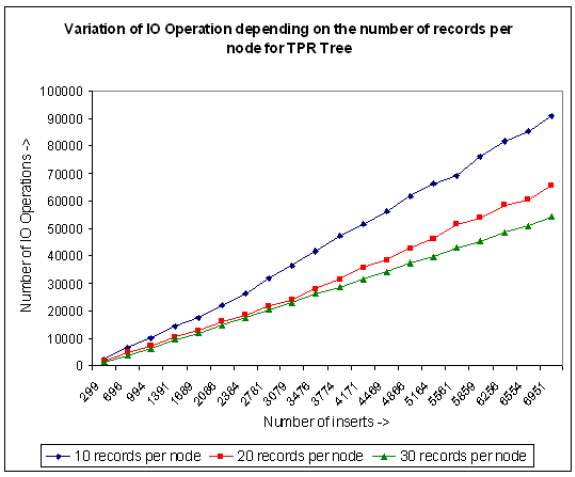

Figure 23: Effect of number of records per node in creation of an TPR Tree

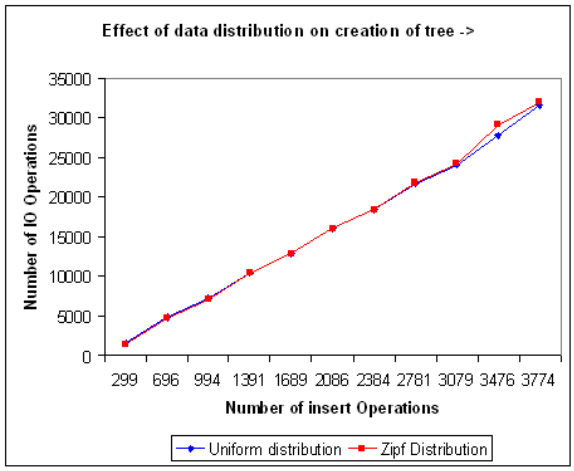

Figure 24: Effect of the initial distribution for the creation of TPR Tree

5.11 Effect of the initial distribution for the creation of TPR Tree

Figure 24 compares the cost difference with changes in initial distribution of the data used to create a TPR tree. Uniform and Zipf distribution were considered. As can be seen the 2 worms almost overlap indicating that the initial distribution does not make a difference in the creation cost.

6 Conclusion

As can be seen from the preceding sections the R Tree is a simple and elegant data structure for indexing spatial or multi-dimensional data. The variants of R Tree - MVR Tree and TPR tree have been built for very specific scenarios and they out perform the R Tree when fed with appropriate data sets. MVR Tree especially is very good for reducing storage space compared to an R Tree when used in an environment that indexes multiple versions of the same data. One such application is the CAD system where several versions of the drawing are stored. In such a scenario the MVR Tree should be an obvious choice. However in other scenarios where multiple versions are not common there is no advantage is using MVR Trees. Similarly TPR trees are very good with moving objects data set. They are well suited for mobile computing since the objects involved are on the move. Using TPR trees with a data set involving mobile objects instead of R trees helps improve on the query latency. However when the data set is static TPR trees offer no advantage.

In this report a brief overview of R Tree, MVR Tree and TPR Tree data structure was provided. The SaIL library was used to demonstrate the scenarios in which the MVR Tree and TPR Tree outperform the R Tree and conclusions were drawn based on the results of the experiments conducted.

References

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

[2] E. H. Marios Hadjieleftheriou and V. J. Tsotras. Sail: A spatial index library for efficient application integration.

[3] W. G. A. Mohamed F. Mokbel, Thanaa M. Ghanem. Spatio-temporal access methods.

[4] Y. Nakamura and H. Dekihara. Spatial data structures for version management of engineering drawings in cad database. In *ICIAP '03: Proceedings of the 12th International Conference on Image Analysis and Processing*, page 219, Washington, DC, USA, 2003. IEEE Computer Society.

[5] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 331–342, New York, NY, USA, 2000. ACM Press.