JACOBS
UNIVERSITY

Space Filling Curve Index

by

Alexandru Valentin Toader

Database and Web Applications Lab

Date of Submission: January 31, 2015

Jacobs University — School of Engineering and Science

Contents

1 Introduction

Array database management systems (DBMS) offer scalable, flexible storage, and fast manipulation of n-dimensional homogeneous collections of items (array/raster data). Often, arrays are used to represent sensor, simulation, image or statistics data.

There are many applications that benefit from efficient processing of array data such as: remote sensing, geology, oceanography, business data and gene data. These applications pose high requirements concerning the data and the operations that need to be supported, and therefore new techniques and tools are needed for increased processing efficiency.

One of the main issues that arise when working with high dimensional, terabyte objects is how to store them in the underlying system while maintaining a high level of performance. Due to their size, raster objects need to be divided into sub-arrays that can be easily managed by the underlying storage provider. In order to attain this, a system for indexing and retrieving fragments of n-dimensional objects must be provided.

Rasdaman is the leading array DMBS and has been used in applications involving data ranging from 1-D measurements to 4-D ocean and climate data. The rasdaman system partitions raster objects into tiles [1] which are indexed using a mechanism based on the R^+-tree . The individual tiles can be considered n-dimensional spatial objects (rectangles).

The purpose of this project was to design and implement an indexing mechanism for multidimensional data objects using the Hilbert R-Tree as the underlying data structure.

The B+ tree [2] has found wide adoption in the field of relational databases and represents the standard data structure for database index development. The task of storing, indexing and querying n-dimensional data is not well supported by indexes based on the B+ tree.

The R-tree, proposed by Antonin Guttman [3], is the preferred method for indexing spatial data. The key idea is to group objects using their minimum bounding rectangle(MBR). Objects are added to a MBR within the node that will lead to the smallest increase in its size. The performance of R-trees depends on the quality of the algorithm that clusters the data rectangles on a node.

The R^+-tree is a compromise between R-trees and kd-trees: it avoids overlapping of internal nodes by inserting an object into multiple leaves if necessary. The R^+-tree offers increased point query performance over the original R-tree at the cost of a higher space utilization.

The original R-tree, proposed by Guttman, is the basis of a number of variations and improvements [4]. The R^*-tree is considered the most robust variant, and has found numerous applications, in both research and commercial systems. However, the empirical study performed in [5] has shown that the Hilbert R-Tree can perform better than the other variants in some cases.

Hilbert R-trees use the Hilbert space-filling curve to impose a linear ordering on the data rectangles with the goal of obtaining an improved clustering of the spatial objects.

To the extent of my knowledge, there is no open-source C++ implementation of the Hilbert R-tree data structure and providing one will be of great value to the community.

The initial specification for the project outlined the following objectives:

- Implement the Hilbert R-tree data structure
- Create a benchmark to compare the performance of the indexing mechanism of the rasdaman system with a new Hilbert R-tree based implementation
- Integrate the Hilbert R-tree data structure into the rasdaman database system.

The first two tasks have been successfully completed, while the third and last task has been left for future research.

The remainder of this document is structured as follows: section 2 discusses the theoretical foundations and algorithms necessary for the implementation of the Hilbert R-tree while also shortly describing the original R-tree implementation and the R^+-tree based implementation used in rasdaman, section 3 describes in detail the technical details of the Hilbert R-tree implementation, followed by a technical description of the benchmark procedure in section 4.

The report ends with a discussion of the results and future work.

2 Related work

In this section, I will first describe the original R-tree proposed by Guttman, as it lays the foundation for the R^+-tree data structure used in rasdaman and the Hilbert R-tree that is the subject of this project, followed by a short description of the R^+-tree variant of the R-tree used in rasdaman, a presentation of the Hilbert curve and, finally, a description of the Hilbert R-tree algorithm.

2.1 R-tree

The R-tree data structure was developed by Antonin Guttmann [3] and is a hierarchical data structure based on the B^+-tree . It is used for the dynamic organization of a set of n-dimensional geometric objects represented by their n-dimensional minimum bounding rectangle.

An R-tree of order (m, M) has the following characteristics:

- Each leaf node (unless it is the root) can host at most M and at least $m \leq \frac{M}{2}$ entries. Each entry is of the form (mbr, oid), such that mbr is the MBR that spatially contains the object and oid is the object's identifier.
- Each non-leaf node can store at most M and at least $m \leq \frac{M}{2}$ entries. Each entry is of the form (mbr, p) where p is a pointer to a child of the node and mbr is the MBR that spatially contains the minimum bounding rectangles contained in this child.
- The minimum allowed number of entries in the root node is 2, unless it is a leaf (in this case, it may contain zero or a single entry)
- All leaves of the R-tree are at the same level.

From the above definition, an R-tree is a height-balanced tree with a maximum height of $h_{max} = [log_m N] - 1$ where N is the number of data rectangles contained. The data structure is dynamic i.e. global reorganization is not required to handle insertions or deletions as we will see in the sub-section covering the insertion and deletion algorithms.

It is worth mentioning that the MBRs of different nodes may overlap each other and a MBR can be contained (in the geometrical sense) in many nodes, but it can be associated to only one of them. This means that a query may visit many nodes before confirming the existence of a given MBR.

Two terms that are important for the performance of an R-tree are *coverage* and *overlap*. The coverage of a level of an R-tree is defined as the total area of all the rectangles associated with the nodes of that level. The overlap of a level of an R-tree is defined as the total area contained within two or more nodes. Efficient R-tree searching demands that both overlap and coverage be minimized. Data structures based on the R-tree as the R^+-tree and the R^*-tree use methods that improve these two metrics.

The R-tree supports the following operations: searching for objects whose MBR intersects a query rectangle Q, inserting new objects, and deleting existing objects.

Search

Given a rectangle, Q, we can find all data rectangles that are intersected by this rectangle. The search starts from the root node of the tree. For every entry (mbr, p) in a non-leaf node, if mbr intersects Q, the search continues on the node that p points to. When a leaf node is reached, Q is tested for intersection with each MBR of each leaf entry, and the entries that are intersected are returned.

Insertion

Insertions in an R-tree are handled similarly to insertions in a B^+-tree. The R-tree is traversed to locate an appropriate leaf to insert the new entry. At each step, the rectangles in the current node are examined, and a candidate is chosen using a heuristic such as choosing the rectangle which requires the least enlargement. The search descends into the tree until a leaf node is found.

The entry is inserted in the found leaf and all nodes on the path from the root to that leaf are updated accordingly. If the leaf cannot accommodate the new entry because it is full, then it is split into two nodes.

In case of a split, the redistribution of the leaf entries is done following a heuristic which impacts the performance of the tree. The objective of the split algorithm is to minimize the probability of invoking both created nodes for the same query.

Three different split algorithms have been proposed by Guttman: linear split, quadratic split and exponential split. Each algorithm trades efficiency of the split for time-complexity. The quadratic split is considered a good compromise between the quality of the split and the time-complexity of the algorithm.

Deletion

In order to delete an entry from the R-tree, the search algorithm is used to find the entry which is then removed from the containing node. If the node does not underflow after the deletion, its parent nodes will be updated and the algorithm finishes. If an underflow takes place, the node in which the deletion took place is dissolved and its entries are reinserted.

2.2 R$^+$-tree

The standard R-Trees dynamic insertion algorithms might lead to a structure that has excessive space overlap and "dead-space" in the nodes which results in bad performance. Efficient R-tree searching demands that both overlap and coverage be minimized. It has been shown that zero overlap and coverage is only achievable for data points that are known in advance and, that using a packing technique for R-trees search is dramatically improved [6].

The R$^+$-tree [7] is a variation of Guttman's R-trees that avoids overlapping rectangles in intermediate nodes of the tree by allowing partitions to split rectangles. The paper introducing the data structure shows that the R$^+$-tree achieves up to 50% savings in disk accesses compared to an R-tree when searching files of thousands of rectangles.

The entries of the R$^+$-tree are identical to the ones of the original R-Tree with the following extensions [7]:

- For each entry (p, mbr) in an intermediate node, the subtree rooted at the node pointed to by p contains a rectangle R if and only if R is covered by mbr. The only exception is when R is a rectangle at a leaf node; in that case R must just overlap with mbr.
- For any two entries (p_1, mbr_1) and (p_2, mbr_2) of an intermediate node, the overlap between mbr_1 and mbr_2 is zero.

One property satisfied by an R-tree but not by an R$^+$-tree is that in the former, every node contains between $M/2$ and M entries unless it is the root. In consequence, the R$^+$-tree achieves better search performance at the expense of space utilization. In the case of point queries, the R$^+$-tree can achieve even more than 50% savings in disk accesses compared to the R-tree. The R-tree suffers in the case of few, large data objects, which force a lot of "forking" during the search while the R$^+$-tree handles these cases, by splitting these large data objects into smaller ones.

A full discussion of the R$^+$-tree algorithms and a performance comparison to the R-tree is available in [7].

2.3 Hilbert curve and compact Hilbert index

A space-filling curve is a parametrized, injective function which maps a unit line segment to a continuous curve in a hypercube, which gets arbitrarily close to a given point in the hypercube as the parameter increases. [8]. In essence, this means that space

filling curves can be used for dimension reduction i.e. by using the inverse of the curve generating function, one can map a multidimensional point to a point on a line.

Space filling curves that are Lebesgue measure-preserving have the additional property that equal length intervals map to equal volumes i.e. points that are close together in 1-dimensional space will map to objects that are close together in multidimensional space.

The Z-order curve, the Hilbert curve, and the Gray-code curve [9] are examples of space filling curves. In [10], it was shown experimentally that, given a set of n-dimensional rectangles, the Hilbert curve achieves the best clustering among the three methods above i.e. the rectangles that are close together in n-dimensional space will be mapped to points that are close together in 1-dimensional space.

The geometric construction of the Hilbert curve starts with the ⊓ shape as shown in the first image of figure 1. This is known as an order 1 Hilbert curve. Given an order k curve, one can obtain an order k+1 curve as follows:

1. Place a copy of the curve, rotated 90° counter clock-wise, in the lower right.
2. Place a copy of the curve, rotated 90° clockwise, in the lower left.
3. Place a copy of the curve in the upper right and one in the upper left.
4. Connect the four disjoint curves.

The first six orders of the Hilbert curve can be seen in figure 1.

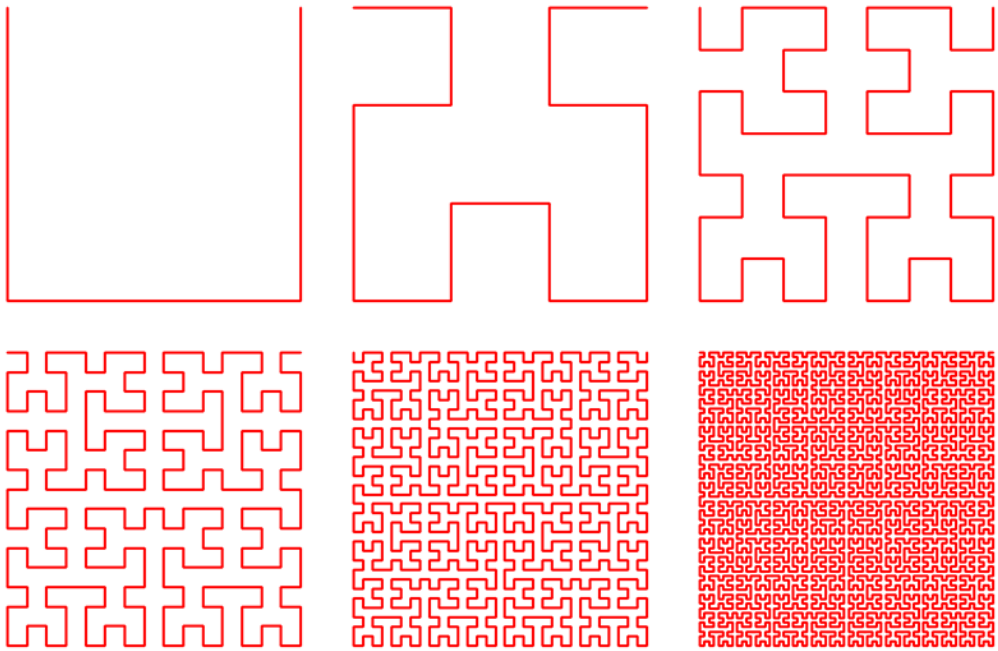

Figure 1: "Hilbert curve". Licensed under CC BY-SA 3.0 via Wikipedia add - `http://en.wikipedia.org/wiki/File:Hilbert_curve.svg#mediaviewer/File:Hilbert_curve.svg`

When the order of the curve tends to infinity, the resulting curve is a fractal, with a fractal dimension of 2. The Hilbert curve can be generalized for higher dimensions. The path of a space filling curve imposes a linear ordering on the grid points. Figure 2 shows such

an ordering for an order 2 curve. For example the point (0,0) on the curve has a Hilbert value of 0, while the point (1,1) has a Hilbert value of 2.

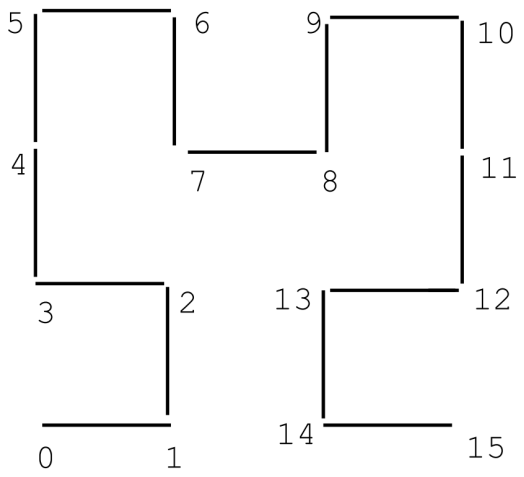

Figure 2: Ordering imposed by the Hilbert curve [5]

A method for efficiently computing the Hilbert coordinates of a n-dimensional point is described by Hamilton and Rau-Chaplin in [11]. A C++ library based on this method is offered by Chris Hamilton at `https://web.cs.dal.ca/~chamilto/hilbert/`. Using this library I have implemented the Hilbert R-tree data structure described in the following subsection.

2.4 Hilbert R-Tree

A standard R-tree splits a node on overflow, turning 1 node into 2. The Hilbert R-Tree is a data structure, based on the original R-Tree, that offers the possibility of deferred splitting of overflowing nodes. This leads to better space utilization and better search performance.

A Hilbert R-tree of order (m, M) has the following structure. A leaf node contains at least m and at most M entries of the form (mbr, obj_id, h) where mbr and obj_id are the same as in the standard R-tree and h is the Hilbert value associated with the center of mbr. A non-leaf node contains at least m and at most M entries of the form (mbr, ptr, LHV) where mbr and ptr are the same as in the standard R-tree and LHV is the largest Hilbert value among the data rectangles enclosed by mbr.

The Hilbert value of the entries is used to create an ordering on the nodes and leads to the concept of sibling nodes. As discussed in the previous subsection, this ordering groups similar data rectangles together, minimizing the area and the perimeter of the resulting minimum bounding rectangles.

By using the ordering imposed on the nodes of the Hilbert R-tree, instead of splitting a node that overflows into two nodes, we can implement an s to $s + 1$ splitting policy. In other words, when a node is overflowing, it tries to push some of its entries to $s - 1$ of its siblings. If all the siblings are full, a new node is created and the entries of the s nodes is

distributed among the $s + 1$ siblings. By adjusting the split policy, space utilization of the tree's nodes can be driven arbitrarily close to 100%.

I will briefly describe the search, insert and delete procedures while referring the reader to the original paper on the Hilbert R-tree by Kamel and Faloutsos [5] for a detailed explanation.

Search

Given a query rectangle Q, the search procedure starts from the root and descends the tree examining all nodes that intersect the query rectangle. When the algorithm reaches a leaf, it reports all entries that intersect the query window Q as results.

Insertion

To insert a new rectangle r in the Hilbert R-tree, the Hilbert value h of the center of the new entry is used as key.

The algorithm initially descends the tree to find a suitable leaf for the new entry. At each level, the node with the minimum LHV greater than the one of the new entry is chosen. When a leaf node is reached, the rectangle r is inserted in the correct position according to h.

If the leaf is about to overflow, the entries are first distributed among the leaf's siblings. If all the s cooperating siblings are about to overflow, a new node is created and the entries of the s cooperating siblings are divided among the $s + 1$ nodes. The LHV and MBR of the nodes directly affected by the insert are updated.

Finally, the changes are propagated upwards by adjusting the LHV and MBR of all the affected parent nodes.

Deletion

In the Hilbert R-tree orphaned nodes do not need to be reinserted whenever a father node underflows. Instead of reinserting the node, we borrow keys from its cooperating siblings or, if all the siblings are about to underflow, the $s + 1$ siblings are merged into s siblings. Finally, the changes are propagated upwards towards the root.

The Hilbert R-tree provides a better space utilization and greater search performance than the other R-tree variants at the cost of a higher insertion cost. In the benchmarks displayed in [5], the Hilbert R-tree outperforms the R*-tree and the standard R-tree in all tests.

In the next section I will give details about the C++ implementation of the Hilbert R-tree that was done for this project.

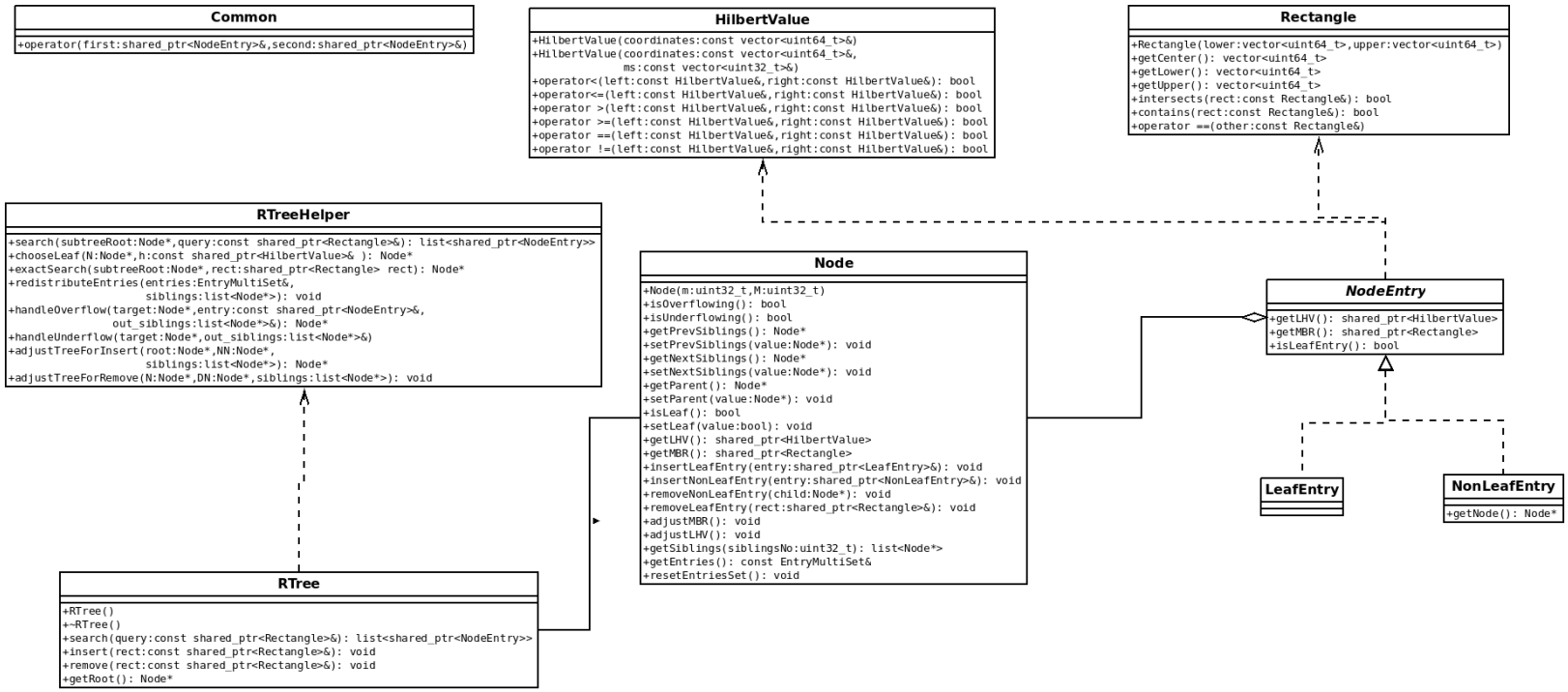

Figure 3: Hilbert R-tree class diagram

3 Hilbert R-tree implementation

The implementation of the Hilbert R-tree was performed in C++ using the Compact Hilbert Index library [12] implemented by Chris Hamilton and the Boost library [13].

The code was written following a test-driven development process. This lead to a large number of tests which also act as a secondary developer documentation for the code. The class structure of the software can be seen in figure 3.

To the extent of my knowledge, there is no implementation of the Hilbert R-tree for n-dimensional data available on the internet. The code written in this step is a basis for integration into the rasdaman index manager.

The implementation if very well tested and documented, and can be found here `https://github.com/atoader/HilbertRTree` together with instructions on how to build and run it.

4 Benchmark implementation

The initial goal was to compare the performance of rasdaman's current index implementation with an implementation based on the Hilbert R-tree. Because this could not be achieved, a benchmark was devised that compares the performance of the Hilbert R-tree implementation with an R^+-tree implementation. The results of this set of performance tests would decide if integrating the Hilbert R-tree into rasdaman would be of value to the community.

The benchmark is comprised of a set of test cases which have been developed considering how the indexing mechanism in rasdaman is used. In rasdaman, raster objects are maintained in a standard relational database, based on the partitioning of an array object into tiles [1] 4. Tiles are stored as BLOBs. The indexing mechanism employed by rasdaman, which is the subject of a possible improvement through this research, is the component that manages these BLOBs.

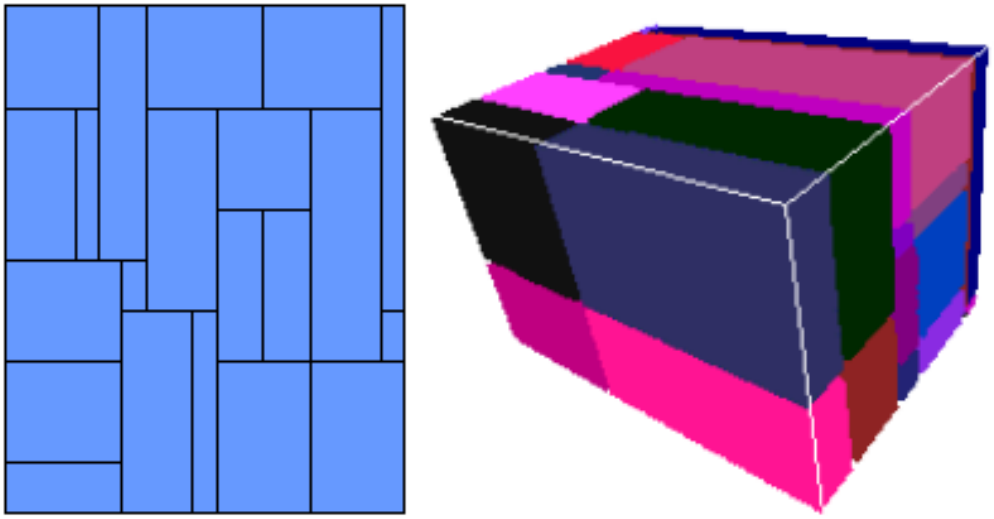

Figure 4: "Sample tiling of an array for storage in rasdaman" by Pebau.grandauer - Licensed under CC BY-SA 3.0 via Wikimedia Commons -http://commons.wikimedia.org/wiki/File:Sample_tiling_of_an_array_for_storage_in_rasdaman.png#mediaviewer/File:Sample_tiling_of_an_array_for_storage_in_rasdaman.png

Rasdaman offers a set of options that can be used to tune the tile storage to the user's requirements when inserting a new collection into the database:

```
INSERT INTO collName VALUES ...
[ TILING tilingName tilingOptions [ TILE SIZE tileSize ] ]
[ INDEX indexName ]
```

These options have an impact on the performance of subsequent queries on the inserted data. The *indexName* parameter is used to select the type of index. The *tileSize* parameter determines the size of a tile. The size of the tile determines the overall number of tiles and also influences how the data is stored on disk. The *tilingName* option allows the user to select from a list of available tiling schemes.

Because this benchmark assesses the performance of the component that indexes tiles and not of the tiling scheme, a regular tiling will be used. The tile size only influences the number of tiles, so it will be fixed in this case.

The trials described below will measure the performance of inserting new data into the index and searching for data in the index. Removing data from the index is not tested because it is not relevant to the use case described above. The tests have been separated into two subsections: Insertion and Search. Multidimensional rectangles of fixed size will be used to simulate the insertion and retrieval of tiles through rasdaman's indexing mechanism.

Insertion

In order to test the performance of insertion, I will insert collections of $[items\_no^{\frac{1}{N}} + 1]^N$ rectangles where $items\_no$ takes values 1, 10, 100, 1000, 10000, and 100000 and N is the dimension of the space and takes values from 2 to 6.

As can be seen from figures 5, 6, 7, 8 and 9 [1], the performance of inserting rectangles into the Hilbert tree takes more time than inserting rectangles into the R^+-tree. This was expected, as the algorithm for inserting an item into the Hilbert R-tree is more expensive than the one for inserting an item into the R^+-tree.

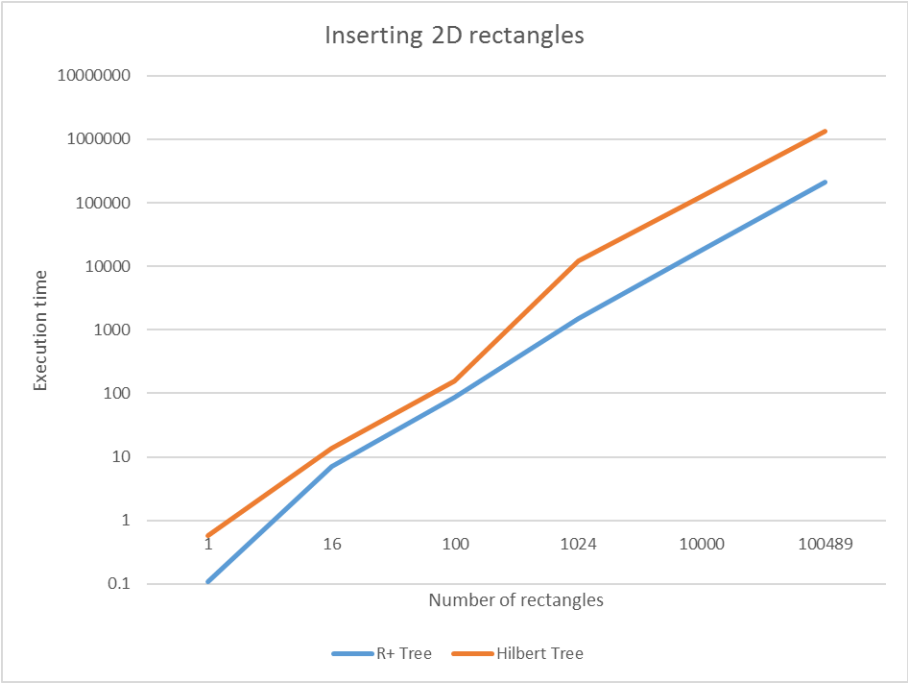

Figure 5: Performance of 2D rectangle insertion

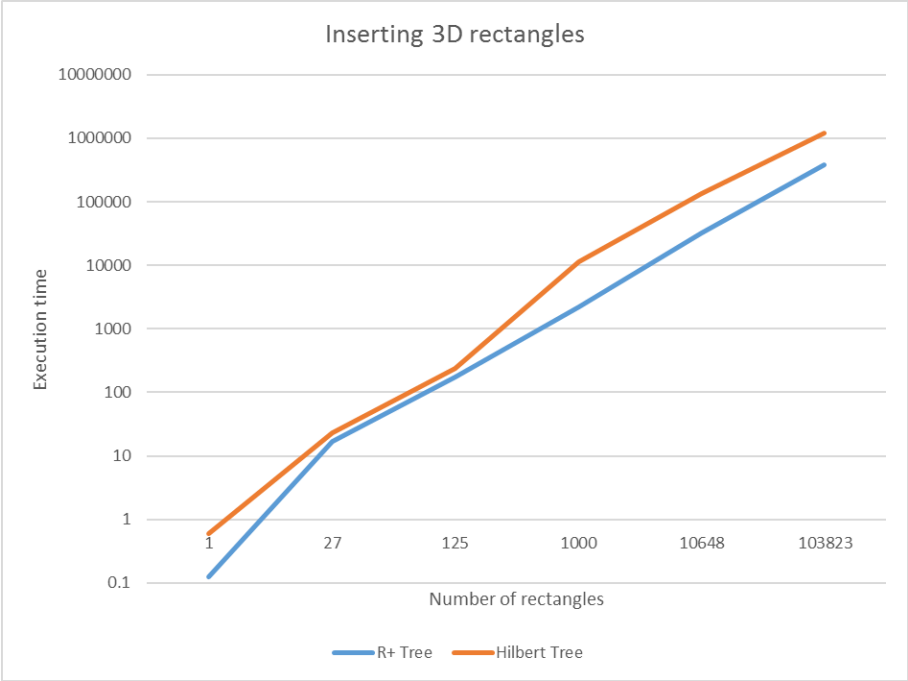

Figure 6: Performance of 3D rectangle insertion

[1] The left vertical axis of the figures uses logarithmic scale.

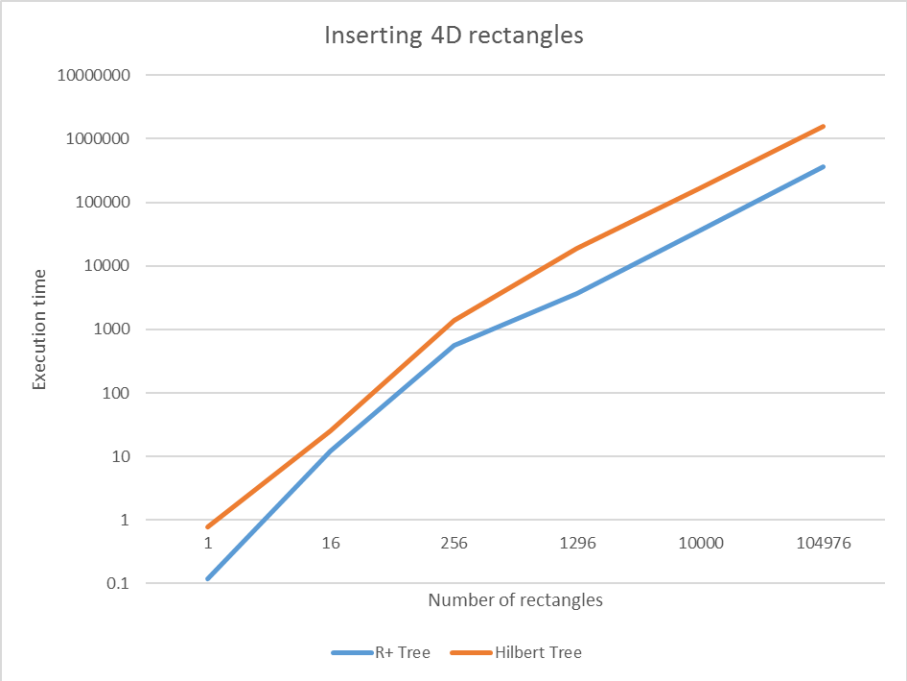

Figure 7: Performance of 4D rectangle insertion

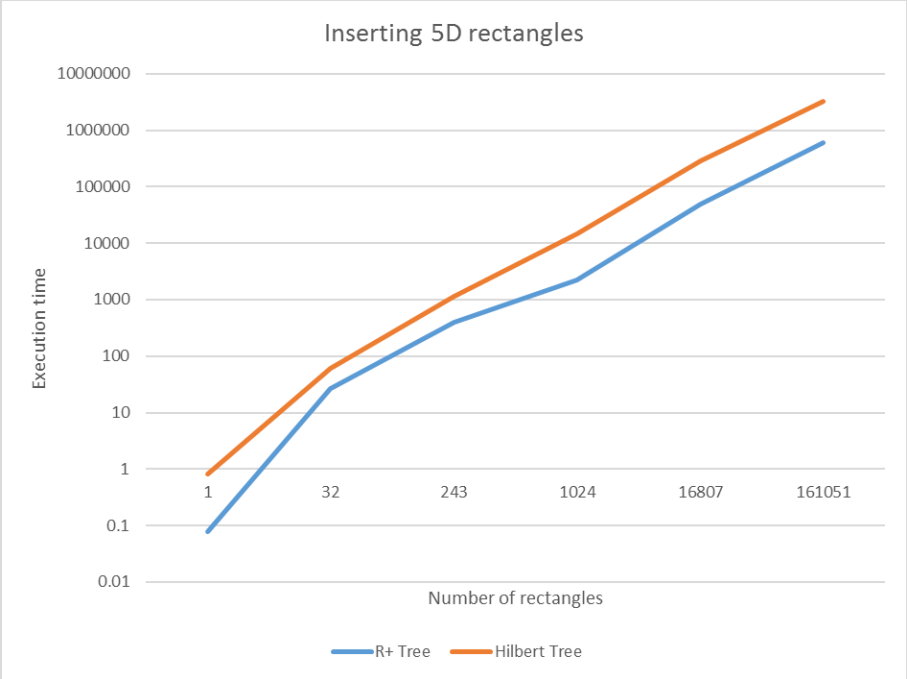

Figure 8: Performance of 5D rectangle insertion

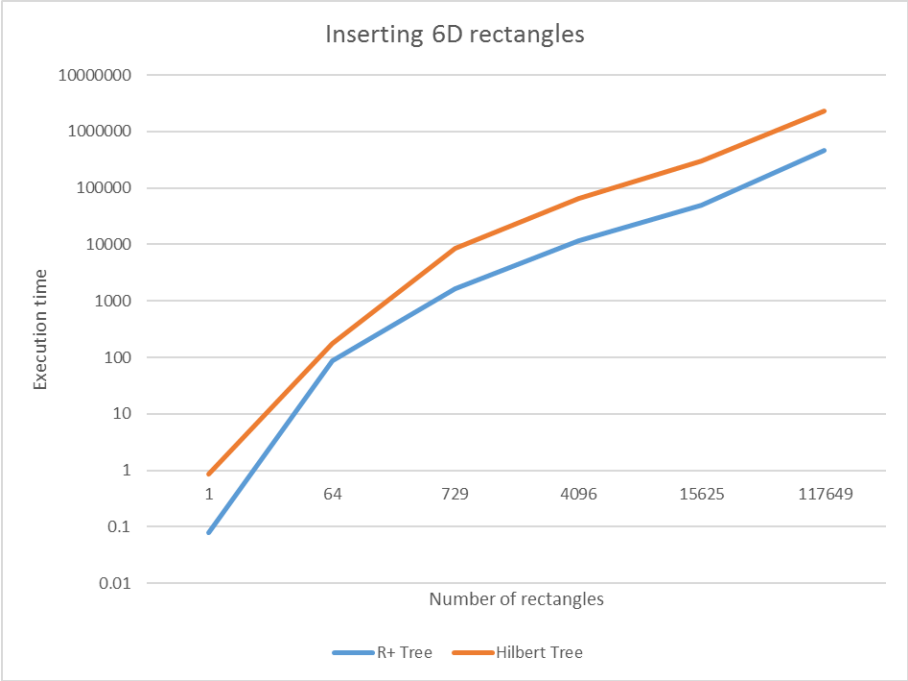

Figure 9: Performance of 6D rectangle insertion

Search

In order to test the performance of search, for each of the collections inserted in the previous step, I have performed the following operations:

1. Retrieve slices of the same dimension, positioned at sequential coordinates through the collection. An example for the 2-D case can be seen in figure 10. The results can be seen in figures 11, 12, 13, 14 and 15.

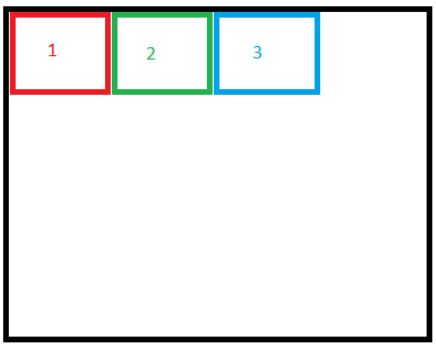

Figure 10: Sequential retrieval of consecutive slices

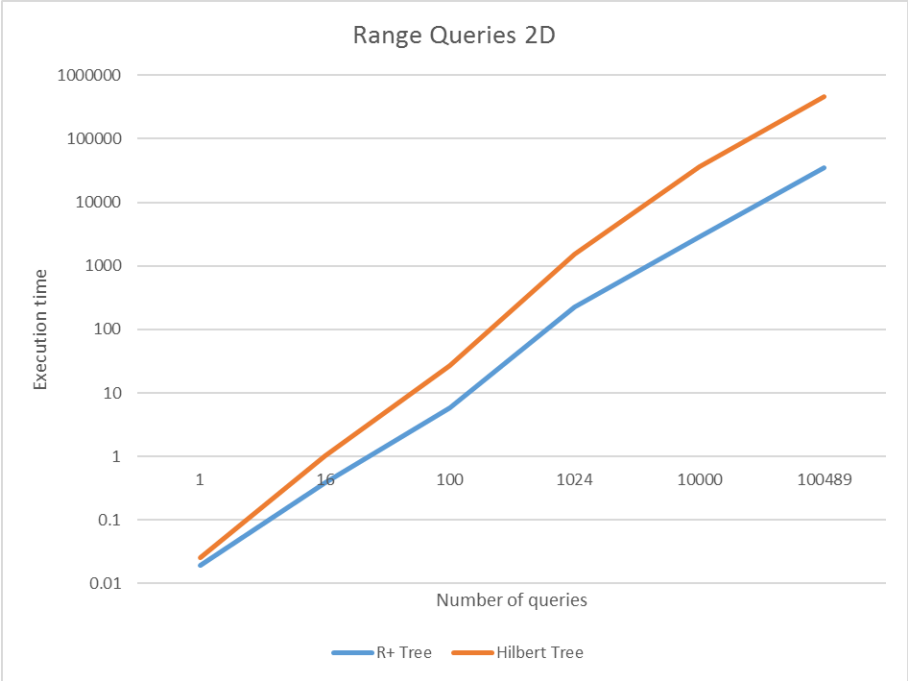

Figure 11: Performance of sequential range queries 2D

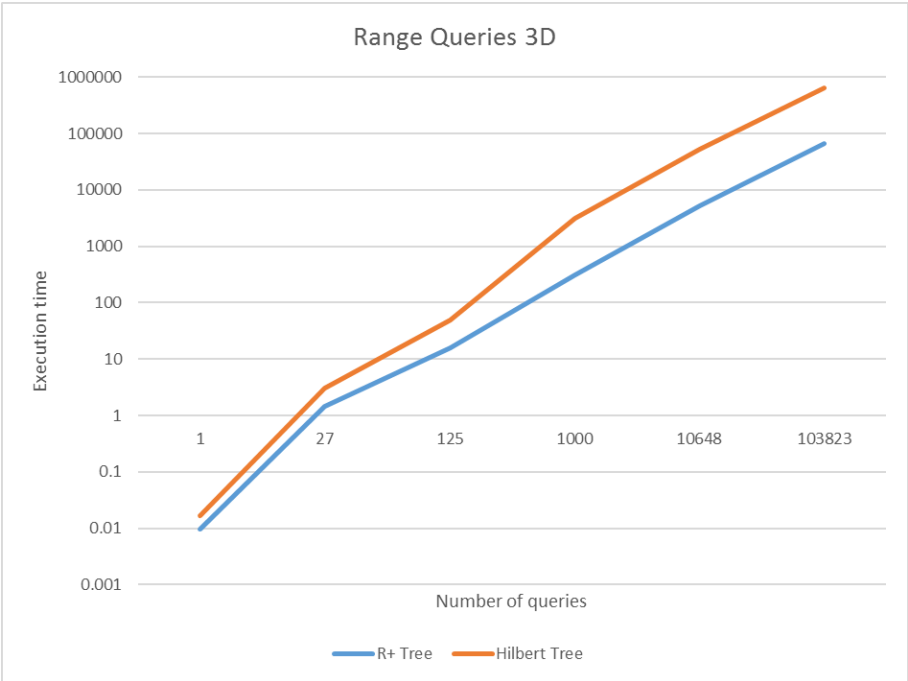

Figure 12: Performance of sequential range queries 3D

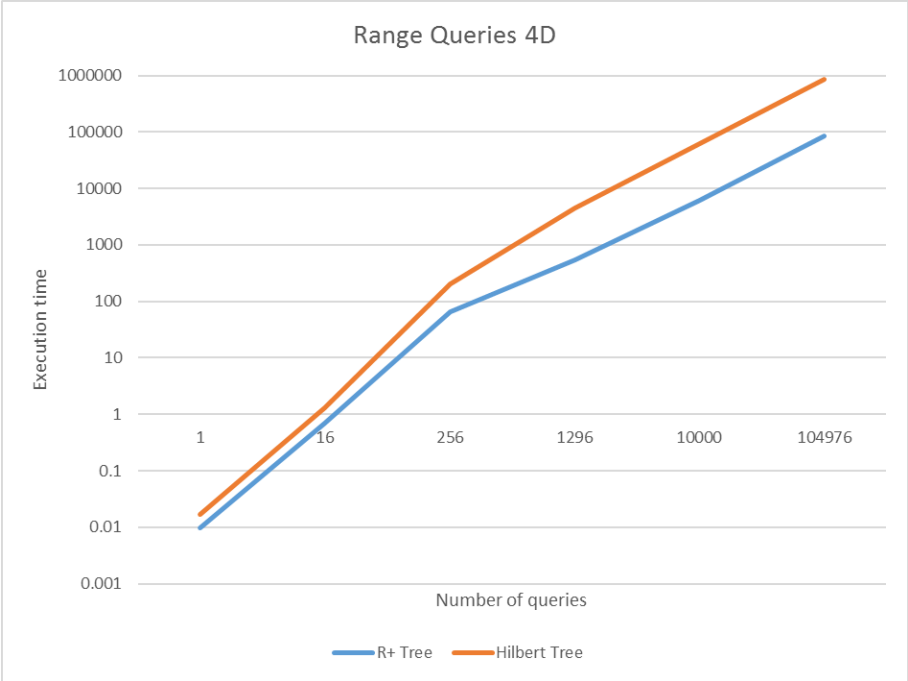

Figure 13: Performance of sequential range queries 4D

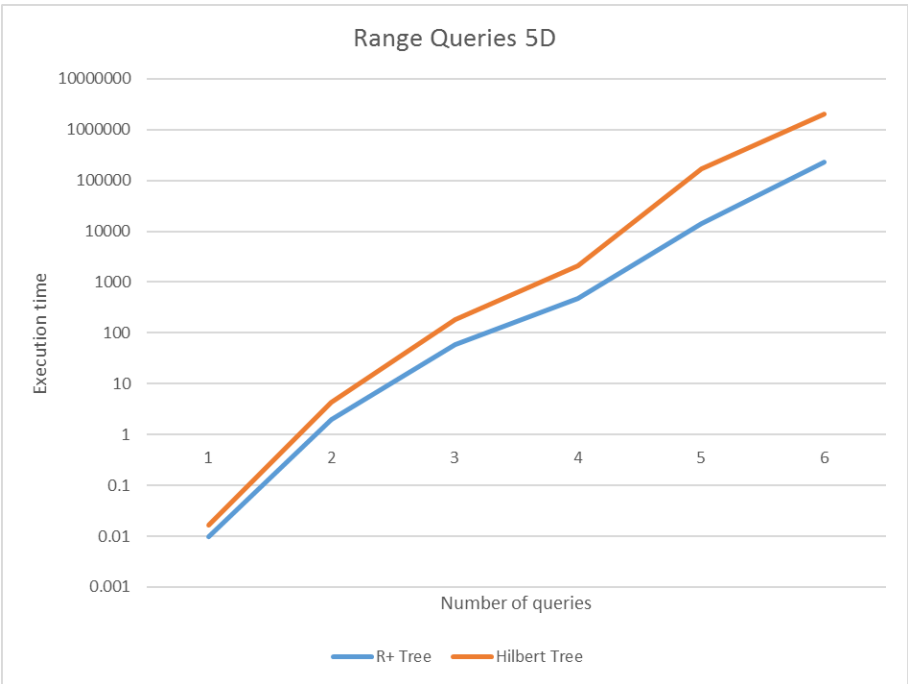

Figure 14: Performance of sequential range queries 5D

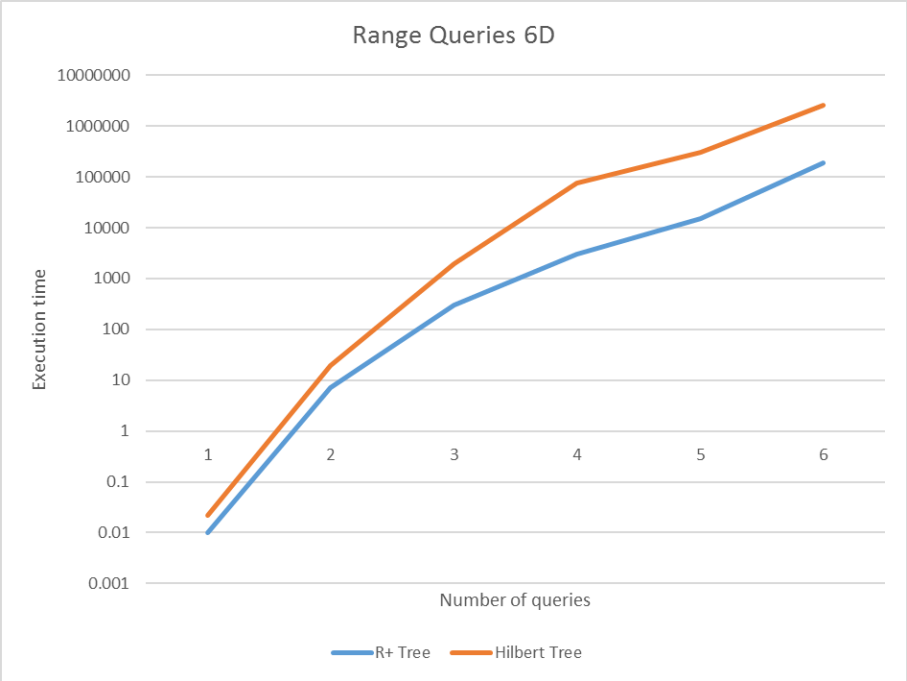

Figure 15: Performance of sequential range queries 6D

2. Retrieve slices of the collection, centered at the upper-left corner and of increasing size in every dimension. This will test the performance of the indexing structure when a large number of nodes are processed by the search algorithm. An example for the 2-D case can be seen in figure 16. The results can be seen in figures 17, 18, 19, 20 and 21.

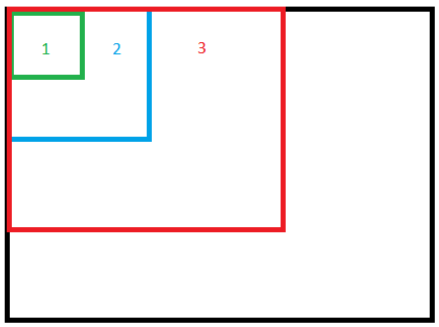

Figure 16: Sequential retrieval of increasing subsets of a collection

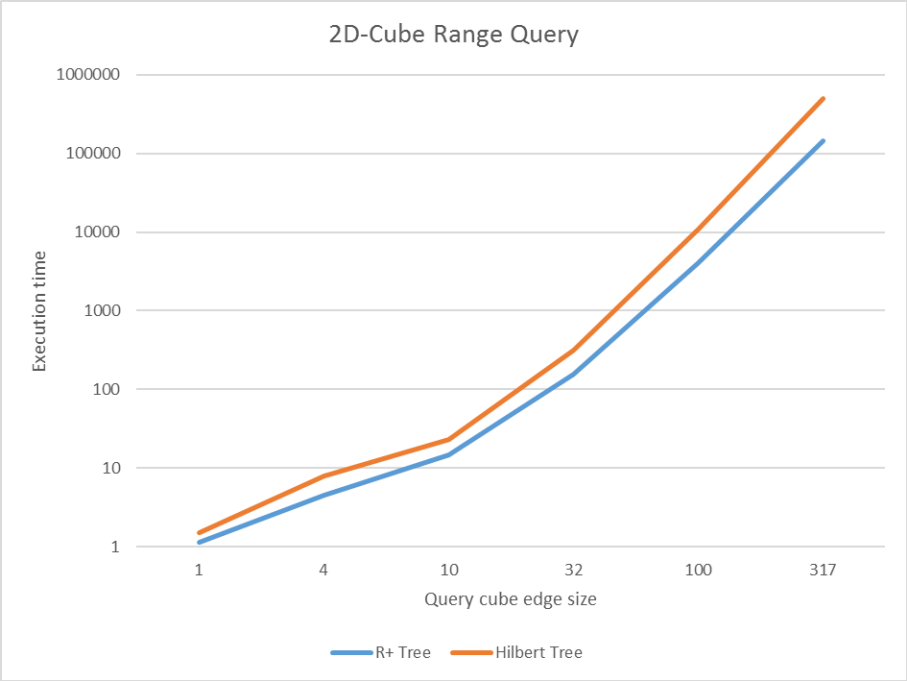

Figure 17: Performance of 2D cube range query

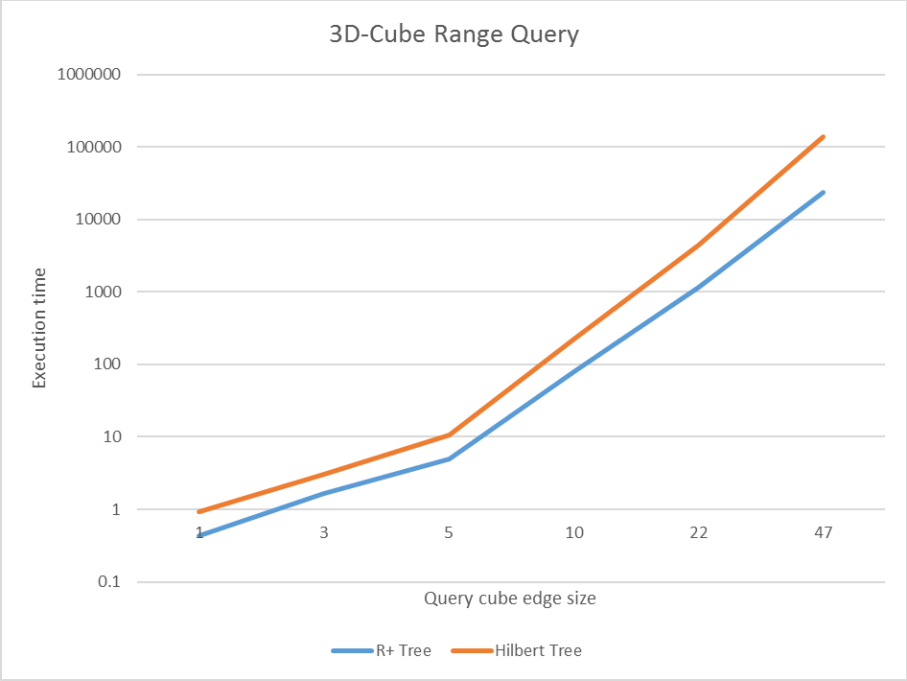

Figure 18: Performance of 3D cube range query

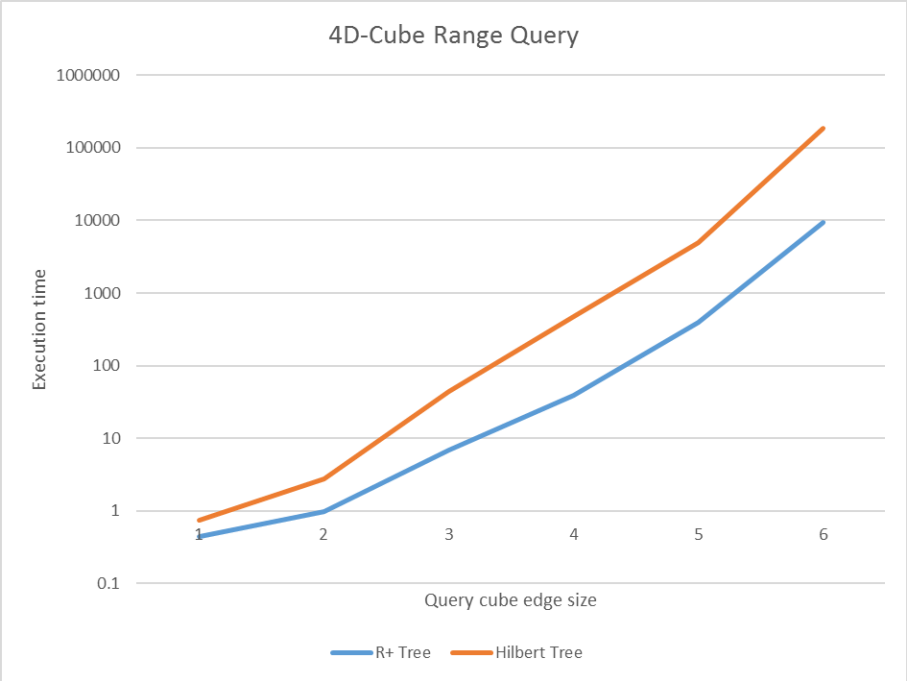

Figure 19: Performance of 4D cube range query

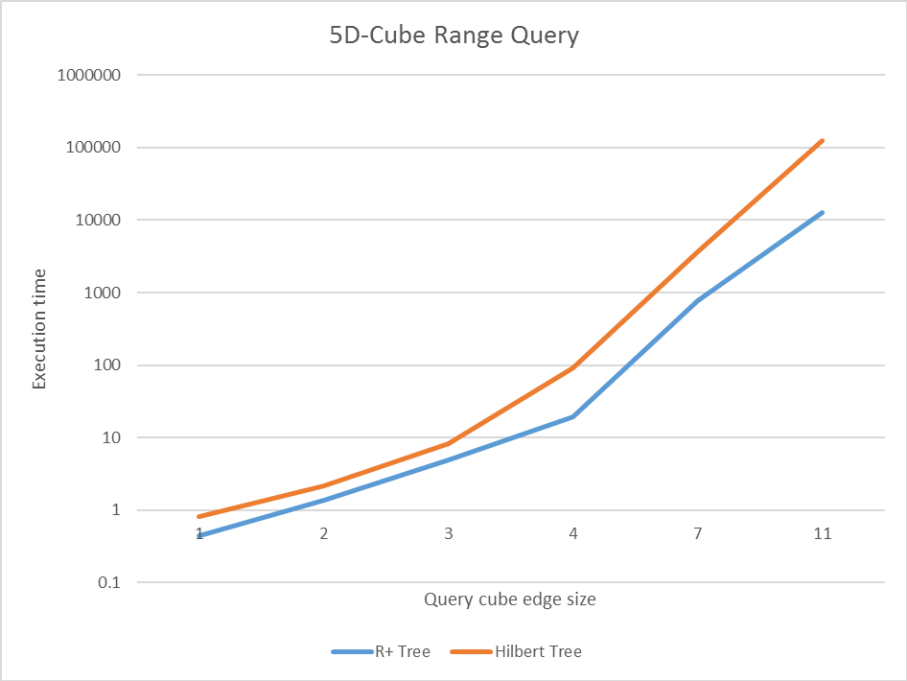

Figure 20: Performance of 5D cube range query

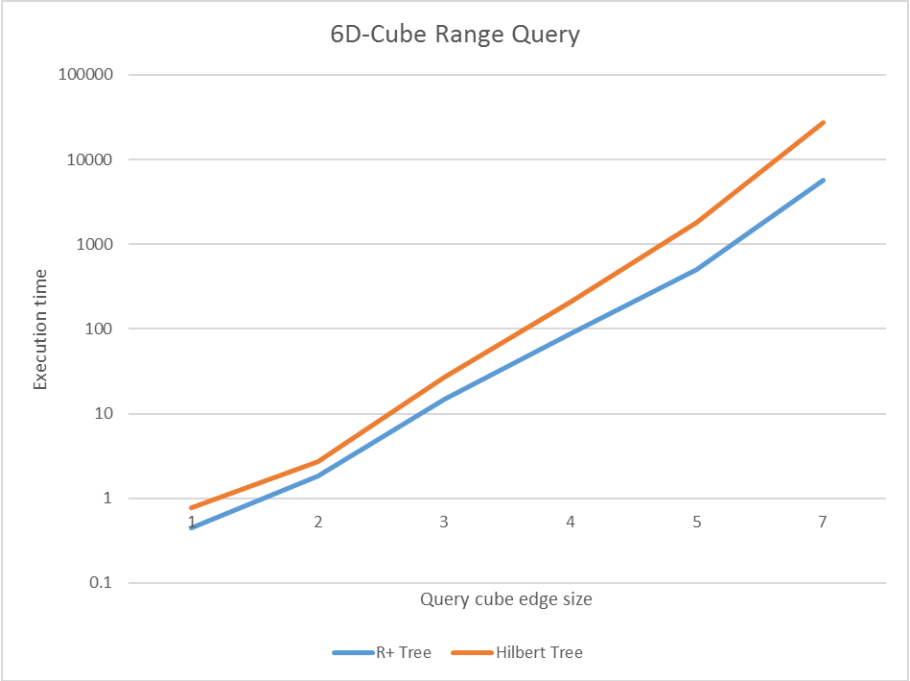

Figure 21: Performance of 6D cube range query

3. Perform point queries targeting each rectangle that is part the N-dimensional cube. The results can be seen in figures 22, 23, 24, 25 and 26.

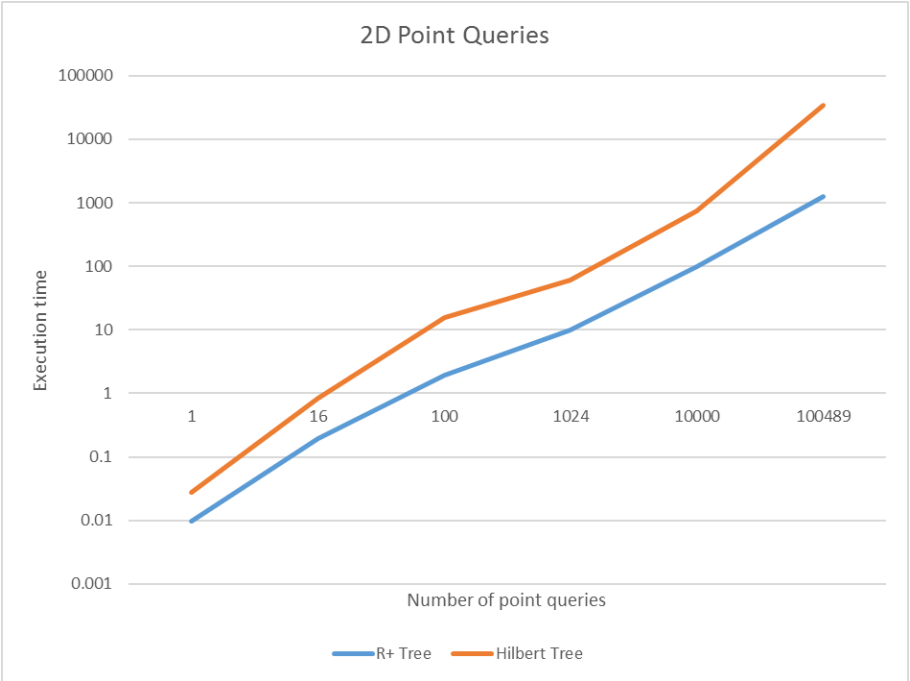

Figure 22: Performance of 2D point queries

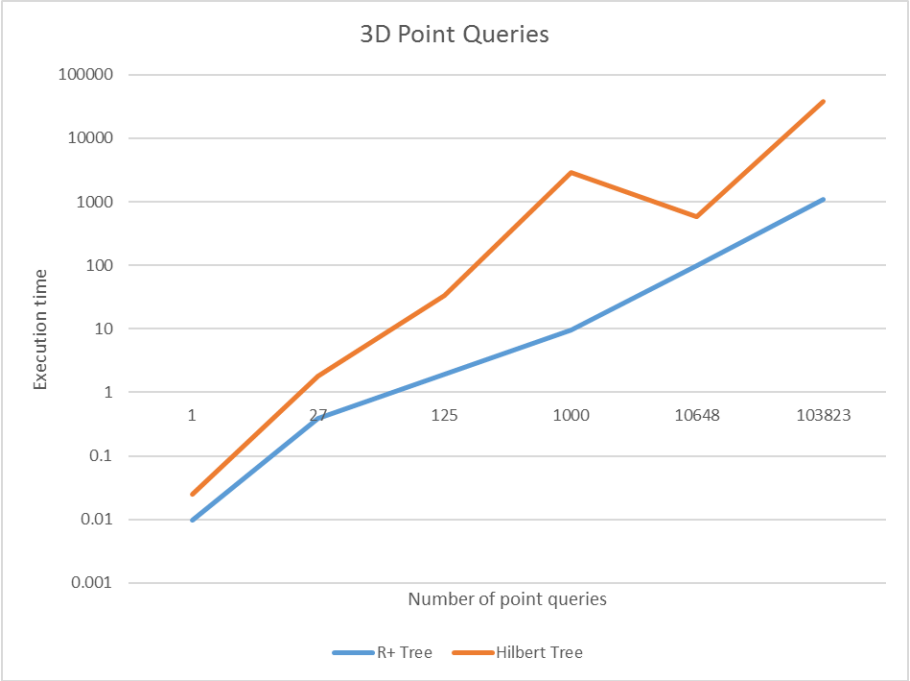

Figure 23: Performance of 3D point queries

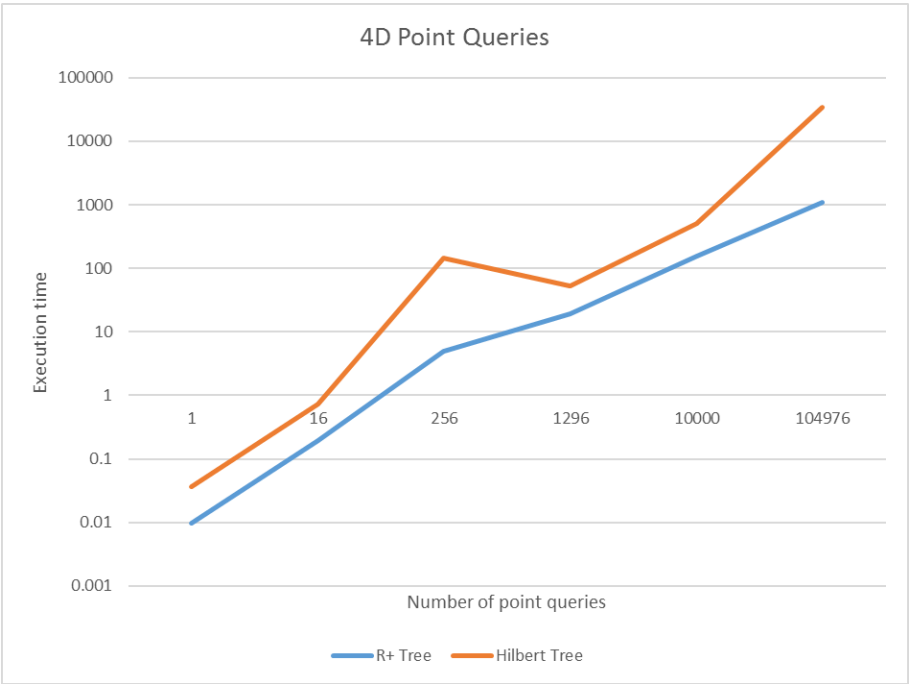

Figure 24: Performance of 4D point queries

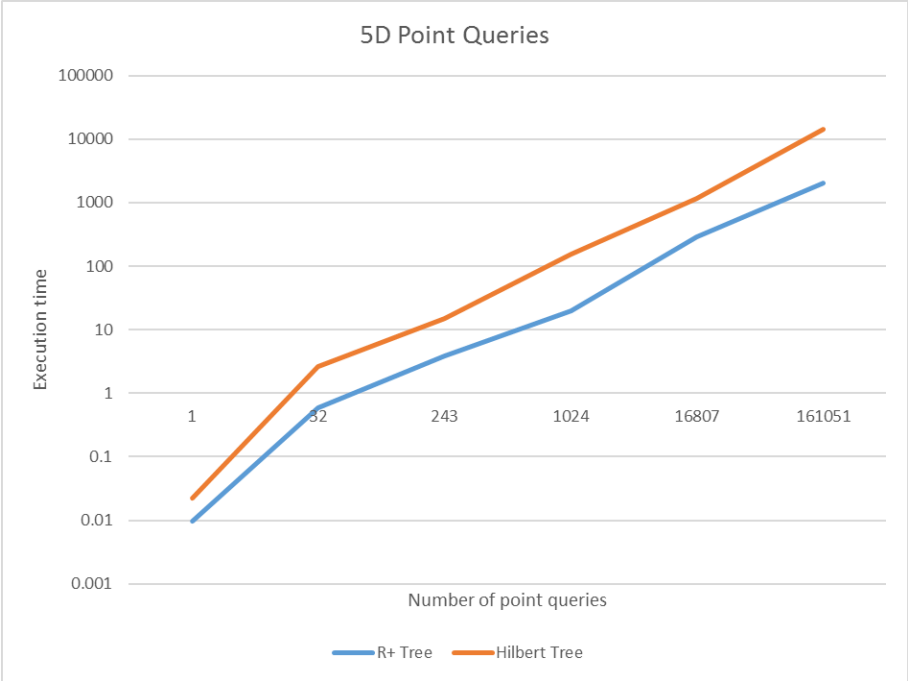

Figure 25: Performance of 5D point queries

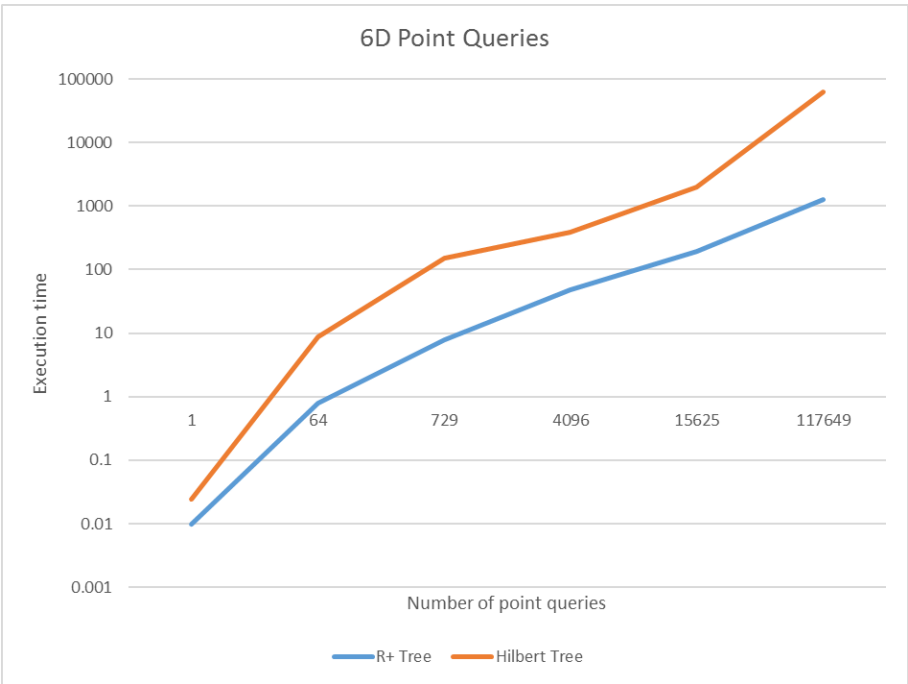

Figure 26: Performance of 6D point queries

For all the tests, the Hilbert R-tree performed, on average, worse than the R^+-tree. This allows me to conclude that, in this case, the Hilbert R-tree is not a viable option for integration into rasdaman's indexing mechanism.

5 Conclusions and future work

As stated in the initial project plan, the Hilbert R-tree data structure was implemented in the C++ language with support for n-dimensional objects. This implementation should prove valuable to the community as a whole, and act as a starting point for the integration into rasdaman's indexing mechanism, given that the performance is improved.

The next step was to develop a benchmark for assessing the performance of the Hilbert R-tree compared to rasdaman's current implementation. Because the Hilbert R-tree could not be integrated into rasdaman, an alternative solution was provided which compares the performance of the Hilbert R-tree implementation with an R^+-tree implementation. The results of this benchmark show that the performance of the Hilbert R-tree is worse than that of the R^+-tree , and integrating the Hilbert R-tree will not add any value to the rasdaman DBMS.

Unfortunately, the research and implementation of the Hilbert R-tree for n-dimensions proved quite involved and accounted for more than 80% of the time allocated to this project. This lead to the impossibility of successfully achieving the last goal of the project, integrating the Hilbert R-tree into rasdaman. This last task, together with improving the performance of the Hilbert R-tree implementation, is left for future research.

References

[1] Paula Furtado and Peter Baumann. Storage of Multidimensional Arrays Based on Arbitrary Tiling. In *Proceedings of the 15th International Conference on Data Engineering, Sydney, Austrialia, March 23-26, 1999*, pages 480–489, 1999.

[2] Douglas Comer. Ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[3] Antonin Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[4] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications (Advanced Information and Knowledge Processing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[5] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 500–509, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[6] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed r-trees. In *Proc. ACM SIGMOD*, 1985.

[7] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[8] Hans Sagan. *Space-filling curves*. Universitext. Springer-Verlag, New York, 1994.

[9] C. Faloutsos. Gray Codes for Partial Match and Range Queries. *IEEE Trans. Softw. Eng.*, 14(10):1381–1393, October 1988.

[10] C. Faloutsos and S. Roseman. Fractals for Secondary Key Retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '89, pages 247–252, New York, NY, USA, 1989. ACM.

[11] Chris H. Hamilton and Andrew Rau-Chaplin. Compact Hilbert Indices: Space-filling Curves for Domains with Unequal Side Lengths. *Inf. Process. Lett.*, 105(5):155–163, February 2008.

[12] Chris Hamilton. Compact Hilbert Index Library, 2015. Retrieved December 6, 2014 `https://web.cs.dal.ca/~chamilto/hilbert/`.

[13] Boost Community. Boost library, 2015. Retrieved December 6, 2014 `http://www.boost.org/`.