Tim Kaler, Oscar Moll May 25, 2012

Spatial Data Structures - Performance Comparision

1 Executive Summary

We evaluated the use of five different data structures to allow Redis, an in-memory database, to support two dimensional indices that can perform range queries efficiently. Through the use of a benchmark suite, utilizing both synthetic and real world data, we compare the performance of these five data structures under different workloads considering both space usage and query time. We implement a simple query optimizer which identifies the best data structure to use for a given 2D range query, and demonstrate its effectiveness on certain query sets.

2 Implementation

We implemented each of our data structures in C within an existing database system called Redis. Redis is a single threaded in-memory database written in C that dubs itself a "datastructure server." It is essentially a key, value store in which the values are either strings, sets, sorted sets, or lists. While Redis supports sorted sets, which can serve as indices into other data structures, it does not support efficient indexing of spatial data. Our goal was to add support for spatial indices in Redis. In order to empirically determine the best approach, we implemented several data structures which could be used to support range queries and evaluated their performance.

3 Data structures

We implemented five different data structures capable of supporting two dimensional rage queries in Redis.

3.1 Filtered Range Search

The simplest way to perform a two dimensional range query is to perform a one dimensional range search and then filter out all points whose y coordinate lies outside the range. A refinement of this approach maintains two copies of the data sorted on each coordinate,

and then determines at runtime whether to perform the initial range query on x or y. For uniformly distributed data points, performing the initial range query on the more restricted coordinate is optimal. A more sophisticated approach may estimate the best dimension to use for the range query using a pre-computed histogram. This method is similar to those used by mainstream database systems which do not specialize in storing spatial data: such as mySQL and PostgreSQL.

3.2 KD Tree

KD trees are a spatial partitioning data structure which are represented as a balanced binary tree in which each node corresponds to a box in space. A KD tree is constructed recursively by partitioning the points within its box on one coordinate, and then constructing two KD trees to contain the points in each partition. The coordinate used for partitioning alternates with depth in the tree, so that the range of each coordinate is partitioned equally in space. Given a range, a KD tree can report all points within that range by performing a range query using each child which overlaps with the range. The recursion bottoms out at leaf nodes, which scan an array of points and report those which are contained within the queried range.

3.3 Layered Range Trees

A layered range tree is a balanced two level binary tree in which each node of the first level points to another tree in the second level. The first level of the tree is sorted on the x coordinate, and the second level of the tree is sorted on the y coordinate. The second level tree contains all points within the x range of the associated first level node. Since the tree is balanced, each point appears in $O(\lg n)$ second level trees bounding the space used by $O(n \lg n)$. To perform a range query in a layered range tree, we check if the range is entirely contained within the x range represented by the current node. If it is, then we perform a range query on the y coordinate using the second level tree. Otherwise, we recursively query each first level child that overlaps with the range. At the leaf nodes, at both the first and second level trees, we simply scan the elements and report those within the range. In the worst case we will perform $2 \lg n$ range queries using the second level trees, leading to an $O(\lg^2 n + k)$ query time.

3.4 Layered Range Trees with Cascading

A refinement of layered range trees eliminates the need to repeatedly perform range searches in the second level data structure. We replace the second level of trees with a second level

of arrays sorted on the y coordinate. Each element in an array contains two pointers to its successor in the arrays of the left and right child node. Since each array is the union of its two children, one of these pointers will point to a copy of the element. To perform a two dimensional range query we first do a binary search on the root node's array sorted by y, and find the successor of the range's smaller y coordinate. We then perform our range search recursively as with layered range trees. By following the successor pointers when we visit a child node from a parent, we will always know the successor of the range's smaller y coordinate in the array of each node we visit. When we reach a node with an x range entirely within the query's x range, we use the second level array at that node to report the points within the box. Since we know the successor of the range's smaller y coordinate in this array, we can perform a scan starting at that element which ends as soon as we reach a point with a y coordinate outside of the range. This approach reduces the asymptotic running time of a two dimensional range query to $O(\lg n + k)$.

3.5 Layered Range Trees with Cascading and Varied Branching Factor

One shortcoming of layered range trees is the $O(n \lg n)$ space requirement which can pose a significant overhead when working with large datasets. One way to reduce this overhead is by increasing the branching factor of the tree. For example, utilizing a branching factor of 4 instead of 2 cuts the height of the first level tree in half which also cuts the total space usage in half. For large data sets, this space optimization can have a large impact. Further, we could increase the branching factor further if additional space reduction was needed.

In order to empirically evaluate the effect of this modification on query times, we implemented a version of layered range trees with cascading that used a branching factor of four. For many tree based data structures used in databases, the branching factor is an important parameter that is normally selected to fill an entire page of memory. By comparing these two data structures utilizing branching factors of 2 and 4 we will better understand what, if any, impact the change has on average query times.

structure	worst case range runtime
filtered 1-D range search	$\lg n + n$
kd-tree	$\sqrt{n} + k$
layered range tree	$\lg^2 n + k$
range tree (cascaded)	$\lg n + k$
range tree (cascaded with branching factor 4)	$\lg n/2 + k$

Figure 1: Range query performance summary

4 Parameter Tuning

4.1 Leaf node size

For each data structure points are stored only in leaf nodes. We coarsen each structure so that leaf nodes contain multiple elements. The number of elements stored in each leaf node is a parameter, L, which we tuned using random data points and query rectangles generated by our Random Rectangle benchmark with 2^{18} query points, and 1 thousand queries. We varied L, testing the performance of our four tree based data structures for each power of two between 16 and 512. The results appear in Figure 2.

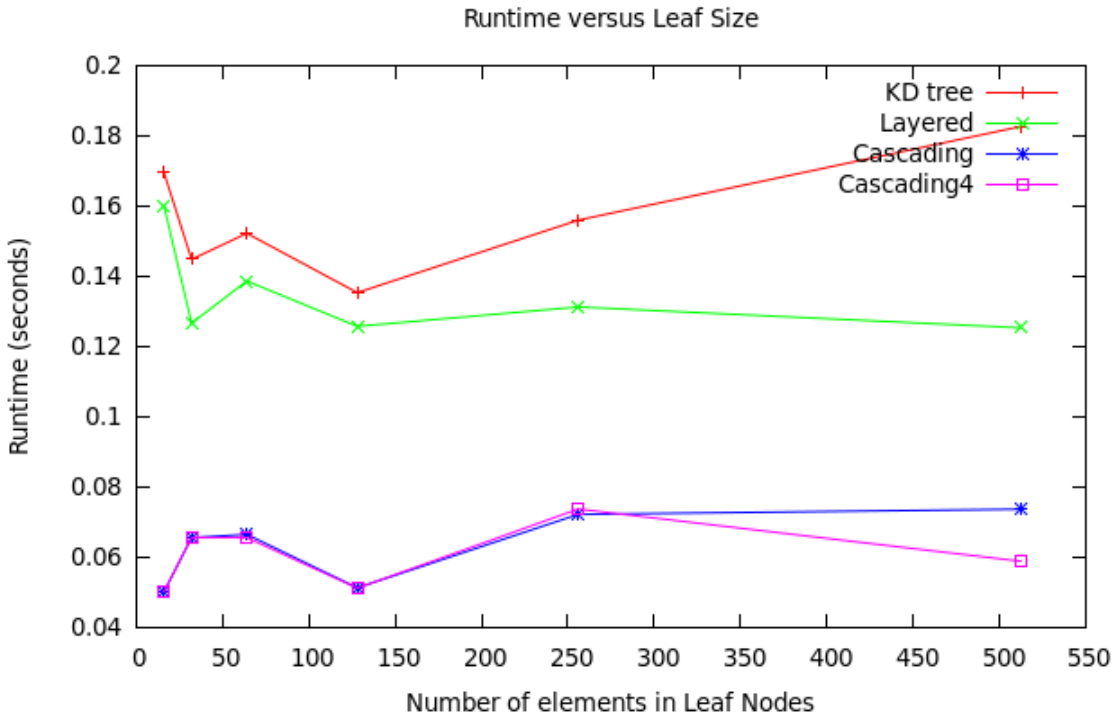

Figure 2: Tuning each of the data structures for optimal leaf size

Although there does exist some noise in the data, it appears as though a leaf node size of 128 is near optimal for all of our tree based data structures. Choosing this leaf node size has the added benefit of making our future performance comparisons more direct. This choice is also consistent with our experiments with our other, larger, benchmarks which has confirmed that 128 is as much as a factor of 2 better than smaller leaf sizes.

5 Benchmark Suite

We tested the five different data structures using a set of benchmarks consisting of both synthetic and real data. We chose our synthetic benchmarks to capture cases which we thought would give us insight into the different performance characteristics of our data structures. Our real benchmarks were obtained from the CarTel [3] project, and consist of latitude, longitude data centered mostly around the boston area.

5.1 Random Rectangles

This benchmark measures the performance of each of our data structures on N random points and R random query rectangles. We pick points uniformly at random within a 100 by 100 box, and generate random queries by picking two points uniformly at random to be two opposite corners of a rectangular query. For this benchmark, we fixed $R = 1000$ and varied N to see how well our data structures scaled as we increased the size of the indexed data set. The results of the benchmark are summarized in Figure 3.

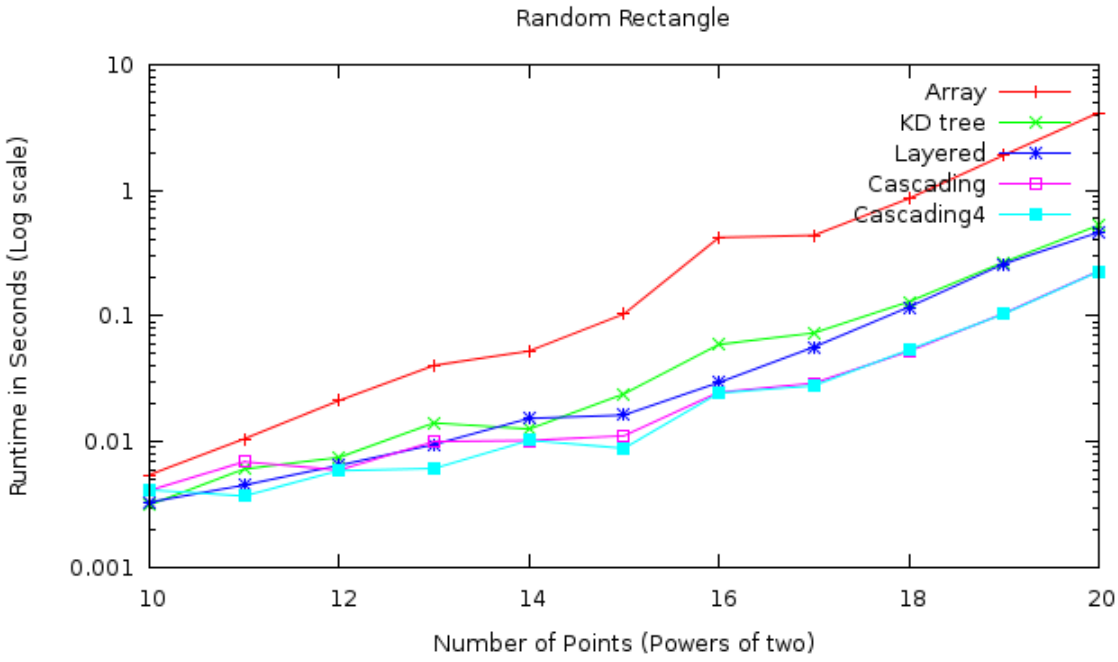

Figure 3: Results of the random rectangle benchmark

The results of this benchmark demonstrate that for random data, and random queries successively more sophisticated data structures yield superior performance. The naive filtering

approach is about an order of magnitude slower than all other data structures. Layered range trees generally outperform KD trees, but the performance gap is often not very wide. Layered range trees with cascading are about a factor of two faster than KD trees and layered range trees without cascading. Finally, we note that adjusting the branching factor of layered range trees with cascading appears to have only a small impact on query time. This suggests that space usage may be reduced, with limited effect on query performance, by adjusting the branching factor of layered range trees with cascading.

5.2 Rectangle Height/Length Ratio Benchmark

This benchmark aims to evaluate the effects of query rectangle shape on the effectiveness of each data structure.

Suppose our dataset consists of n points scattered uniformly at random on the square $[0,1] \times [0,1]$, and that we wish to perform a rectangular query of size $\Delta_y \times \Delta_x$. Let k be the result set size. On average $k = n\Delta_y/\Delta_x$. With r defined as before, we can replace all occurrences of Δ_y and Δ_x with expressions only in terms of k and r.

Then, the cost of the 1-D range tree with filtering becomes

$$\lg n + \min(\Delta_x, \Delta_y)n$$

We take the minimum of the two dimensions, because our filtering datastructure chooses to perform the 1-D range query on the more restrictive dimension. For uniformly distributed data, this will be the dimension with the smaller range.

Suppose we use an alternate data structure with theoretical query time

$$f(n) + (\Delta_x \Delta_y)n$$

For a kd-tree $f(n)$ is $\sqrt{n}$, for a layered range tree $f(n) = \lg^2 n$, etc.

Using this model we can explicitly analyze and predict query performance for our datastructures as we vary the shape of the rectangle. We wish to de-couple the effect of changing the shape of the query rectangle from the possibility of increasing the result size. Therefore, we choose to vary only the ratio of height to length, $r = \Delta_y/\Delta_x$, while keeping the area, $\Delta_y\Delta_x$ fixed. Since $k = \Delta_y\Delta_x n$, we have that

$$\Delta_x = \sqrt{k/nr}$$

and

$$\Delta_y = \sqrt{kr/n}$$

.

So, $n \min(\Delta_x, \Delta_y) = n\sqrt{k/rn}$, and the linear scan then has a runtime of

$$\lg n + \sqrt{nk/r}$$

This suggests that the 1-D range tree followed by a linear scan can perform better than the other structures when

$$\lg n + \sqrt{nk/r} \leq f(n) + k$$

The range of r for which this is possible can be found by solving r:

$$\begin{aligned} \sqrt{nk/r} &\leq f(n) + k - \lg n \\ nk/r &\leq (f(n) + k - \lg n)^2 \\ r &\geq \frac{nk}{(f(n) + k - \lg n)^2} \end{aligned}$$

The result from 1 shows that given a fixed size of a data set n and result set k, we can always find a rectangle tall enough that it becomes better to simply use a 1-D range query. This agrees with intuition, the taller a rectangle, the more 1-D the result set, and thus few points will be filtered out after the 1-D range search.

To test this, we designed a benchmark for this scenario. We scatter $n = 10^4$ points on a 100×100 square, and vary the rectangle shape while keeping the area constant. We work with rectangles of total area 100, and shapes ranging from $r = 1$ (a 10×10 square) to $r = 100$ (a narrow 100×1 slab). For each ratio we perform 2000 random range queries uniformly distributed within the universe. The results are summarized in 4.

As we can see, the cost of the filtering 1-D range query decreases in a shape similar to $1/r$.as we increase r, which is what we expected from the model presented before. Interestingly, the empirical performance of the other data structures also changes with r in this scenario, for the kd-tree performance actually worsens, whereas we see some improvement in all three variations of layered range trees.

Moreover, from the equation in 1 with the parameters we used for the benchmark, we let $n = 10^4$ and $k = 100$ as in the model, and $f(n) = \lg^2 n$ for the layered range trees. The range at which r is large enough so that filtering works better is then predicted to be $r \geq \frac{10^4 * 100}{(\lg^2 10^4 + 100 - \lg 10^4)^2} \sim 10^6/4(10^4) \sim 25$. We get a similar result for $f(n) = \sqrt{n}$. These equations were not meant to be accurate at the level of constants in the first place, but notably, the constants are reasonable enough that the estimate is within an order of magnitude of the empirical one.

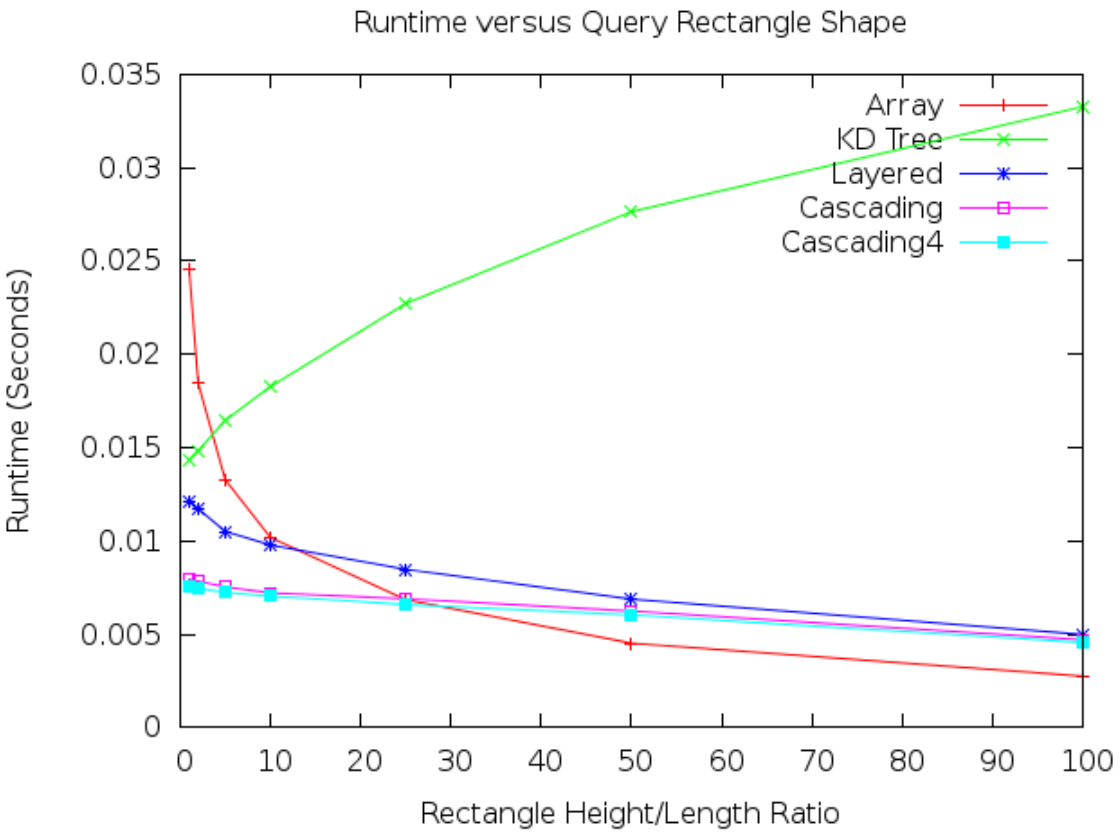

Figure 4: Variation of Height /Length ratio and effects on query time for our data structures

5.3 CarTel Synthetic Benchmark

We obtained a synthetic benchmark used by the CarTel group to measure the performance of range queries performed on their database of latitude, longitude coordinates. It is generated by picking a box according to some ordered distribution, and then filling that box with uniformly distributed points. The dataset consists of about 1.1 million data points. The following plots show the macro and micro structure of the data set. The queries provided to us represent a sample of the types of range queries performed on the CarTel databases of lat/long coordinates. We used a query trace consisting of approximately $3,000$ queries, also provided to us by CarTel, to run on this dataset.

We compared the performance of our five data structures on the query traces over this data set.

structure	runtime
filtered 1-D range search	4.483 seconds
kd-tree	3.027 seconds
layered range tree	2.154 seconds
range tree (cascaded)	0.764 seconds
range tree (cascaded with branching factor 4)	0.744 seconds

Figure 5: CarTel synthetic benchmark results

5.4 CarTel Real Benchmark

We obtained a dataset of approximately $200,000$ latitude, longitude points obtained from vehicles participating in a data collection program for CarTel. Data was collected from taxi cabs which were fitted with gps units, as well as by graduate students driving around boston. The data set, shown in Figure 6, is composed of points clustered in urban centers, and aligned along major roads. We utilized the same query trace of approximately $3,000$ queries as used in the synthetic benchmark. The synthetic benchmark was designed to cluster points in the boston area, and thus the query traces are appropriate for use in both benchmarks. The results are summarized in Figure 7.

6 Query Optimizer

Based on our experience with the model from subsection 5.2, we felt given some tuning we could automate the decision on which access path to use for a given 2-D range query. We implemented a rudimentary query optimizer based upon the performance characteristics we observed while conducting our benchmarks. The basic idea is that if using the model from that section, we can tell based on the ratio r, the number n and an estimate of the result set size k, which data structure to use. In summary, we concluded that layered range trees with cascading work best under most circumstances, but as revealed by our Rectangle Height/Length benchmark, as the 2D range query becomes more and more like an axis parallel slab, it is preferable to perform a 1 dimensional range query followed by a naive filtering step. The results of running the optimizer are shown in Figure 8. We note that our optimizer would need significant additional work in order to be applicable to arbitrary data sets. However, our basic experiments with the concept show that the idea may be useful under some circumstances.

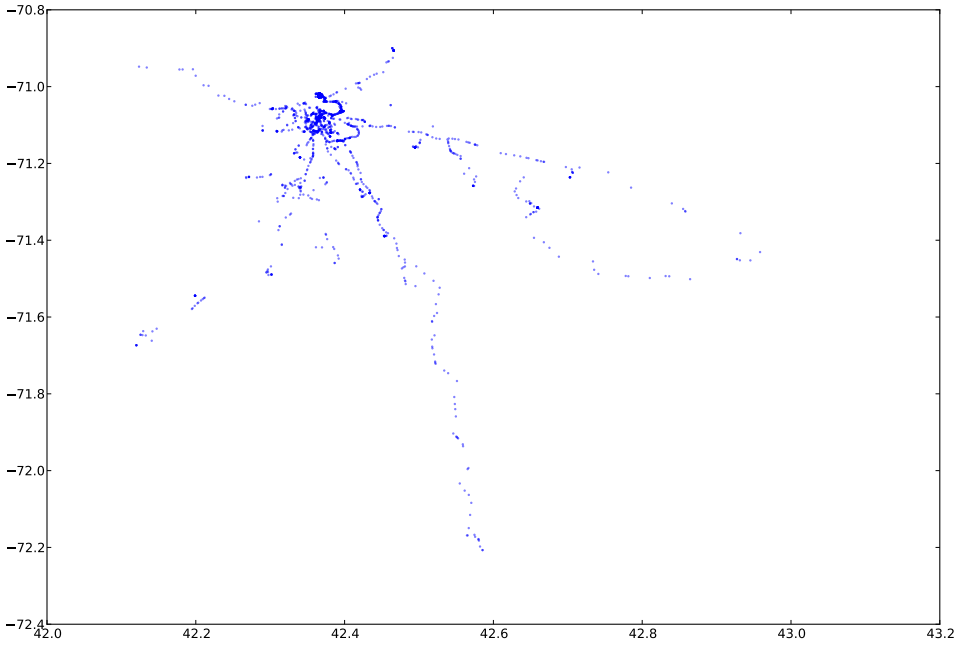

Figure 6: Dataset from the real cartel benchmark

structure	runtime
range	0.547 seconds
kd-tree	0.329 seconds
layered range tree	0.297 seconds
range tree (cascaded)	0.142 seconds
range tree (cascaded with branching factor 4)	0.143 seconds

Figure 7: CarTel vehicle position dataset results

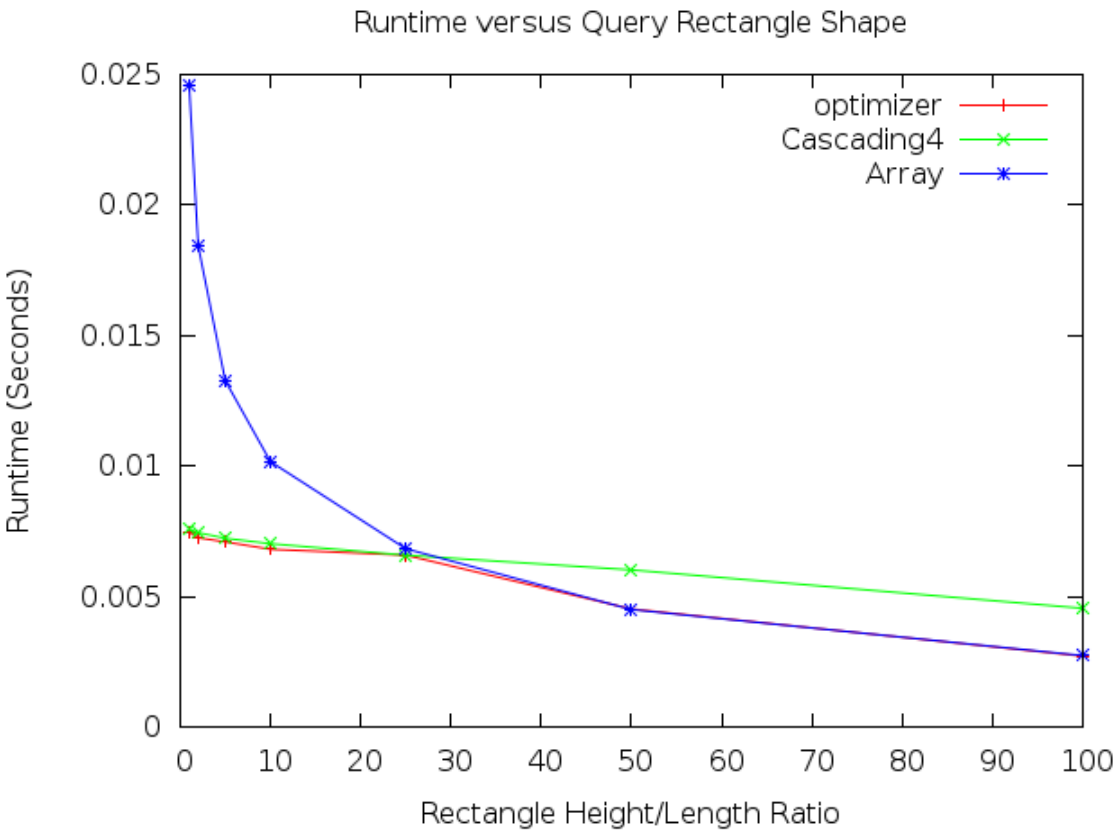

Figure 8: Optimizer choosing access method based on model

7 Conclusion

We conclude that the best candidate data structure for adding support for spatial queries in Redis is the layered range tree with cascading. This data structure has consistently good performance on all of our benchmarks. The data structure's space usage may be a concern, especially for Redis which needs to run in-memory in order to get good performance. For this reason it would be necessary to further investigate the impact of adjusting the branching factor of the tree to larger values. However, our results comparing the implementation with branching factor 2 against the implementation with branching factor 4 lead us to believe that the trade-off between space and time will be modest. Our experience attempting to implement a query optimizer to select, at runtime, the best data struture to use for a given range query was interesting and may improve performance under certain skewed workloads. By maintaining aggregate statistics on the data set, perhaps as a histogram, it may be possible to efficiently estimate and compare the cost of a 1-D range search followed by filtering for datasets with arbitrary distributions. Our code may be viewed on GitHub at: http://github.com/omoll/redis

References

[1] Multiple-column indexes. MySQL 5.0 Reference Manual. `http://dev.mysql.com/doc/refman/5.0/en/multiple-column-indexes.html`.

[2] M. de Berg. *Computational Geometry: Algorithms and Applications*. Springer, 1997.

[3] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen K. Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. CarTel: A Distributed Mobile Sensor Computing System. In *4th ACM SenSys*, Boulder, CO, November 2006.