Benchmarking the Cache-Oblivious 2D Orthogonal Range Search Data Structure

Kuo-An Andy Wei, Kai Yuanqing Xiao

December 14, 2017

1 Introduction

Today's computers have a complex, hierarchically structured memory system consisting of disk, main memory, and several layers of cache. Accessing data from different parts of the memory hierarchy requires vastly different amounts of time—it is orders of magnitude slower to access data from disk than from the L1 cache level. Thus, the performance bottleneck of many algorithms in practice is not the theoretical runtime of the algorithm, but the number of disk accesses necessary for that algorithm to run.

As a result, the IO model was developed as another method of analyzing the efficiency of algorithms. An efficient algorithm in the IO model seeks to minimize the number of disk accesses needed by the algorithm. Various algorithms have been designed and analyzed in the *external memory* setting, in which a 2-layer memory hierarchy consisting solely of the disk and cache exists and both the size of the cache and the block size of chunks of memory are known. However, not as much work has been done to analyze multi-level memory architectures, which is closer to the reality of modern computers.

In recent years, another important line of study has been *cache-oblivious* algorithms, which are designed to work efficiently without knowing the size of caches or the size of individual blocks. These algorithms are useful when memory configurations are unknown to the algorithm, and they are especially useful across multilevel memories. This is because these algorithms are designed to be efficient for arbitrary memory configurations, so they will work regardless of the specific memory configurations at each level of a memory hierarchy.

Coors2D is a cache-oblivious data structure designed to solve the *2D Orthogonal Range Search* problem[1], or the problem of finding all points inside a given query box $[x_1, x_2] \times [y_1, y_2]$. The Orthogonal Range Search problem arises frequently in the fields of graphics, computer aided design, and databases[2].

The goal of the data structure is to be able to perform queries in $O(\log_B N + k/B)$ disk accesses and to use $O(N\text{polylog}N)$ space, where N is the total number of points, k is the size of the output of the query, and B is the block size.

In our project we implemented the Cache-Oblivious 2D Orthogonal Range Search Data Structure in Python 3 and compared its practical performance with a variety of simpler baselines data structures.[1]

[1]The code is available at https://github.com/6851-2017/Coors2D

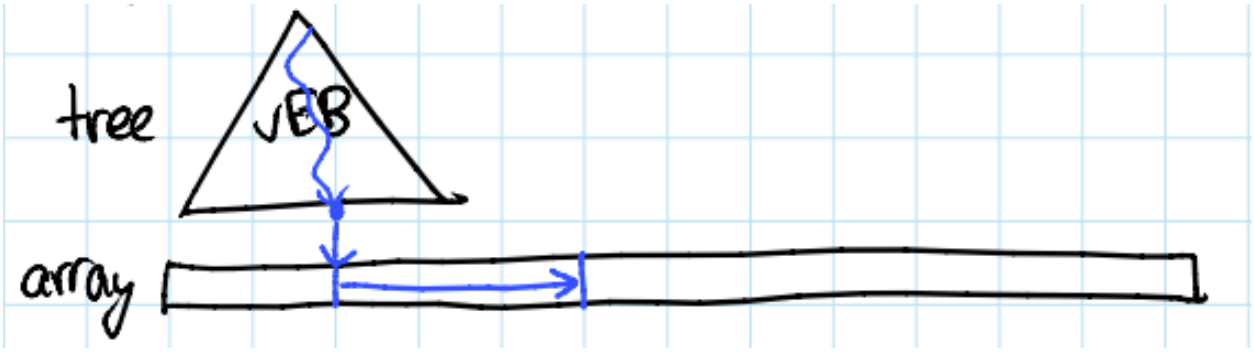

Figure 1: A schematic diagram of query algorithm for Coors2D[3].

2 Coors2D Data Structure

The Coors2D Data Structure is built up in a manner similar to regular orthogonal range trees. First, a structure capable of supporting 2-sided orthogonal range queries of the form $(\leq x, \leq y)$ is constructed. Next, a structure capable of supporting 3-sided queries is built by forming a search trees where each node n of the tree contains two mirrored versions of the 2-sided structure built on the points of the subtree with root n. The final 4-sided query structure is built from the 3-sided query structures in a similar fashion. We provide a brief overview of the Coors2D data structure and our implementation details in the following sections.

2.1 Coors 2-Sided

The Coors 2-Sided structure stores N points and supports orthogonal queries of the form $(\leq x, \leq y)$ in $O(\log_B N + \lceil k/B \rceil)$ time. It consists of a static VEB-layout binary search tree (which we refer to as the VEBTree) on the set of points keyed by its y-coordinate, and an preprocessed array (which we refer to as the XArray) consisting of "chunks" where points are monotonically ordered by x-coordinate within each chunk. The XArray may contain duplicates of certain points, but its size will not be significantly bigger than N[3].

The primary idea is that for a query $(\leq x^*, \leq y^*)$, the data structure will first find the the successor of the coordinate y^* in the VEB tree, then follow a pointer from this y^* to the start of the corresponding chunk in the XArray.

Then, for any x, the XArray will ensure that the query $(\leq x, \leq y^*)$ will be "dense" in the XArray chunk. Here, we use the word "dense" to mean that at least $\frac{1}{\alpha}$ of the points that would be outputted from the query $(\leq x, *)$ must be in the query $(\leq x, \leq y^*)$ for some constant α.

2.1.1 VEBTree

The VEBTree is a static perfect binary search tree on the set of points keyed by the y-coordinate, stored in memory in the recursive VEB layout. This allows for cache-oblivious $O(\log_B N)$ predecessor and successor search. In our implementation, we simply convert the

perfect BST into a list of the nodes in VEB order, and store the BST nodes in VEB order in our external memory setup.

2.1.2 XArray

The other component of the Coors 2-Sided structure is the XArray.

The XArray is constructed iteratively. First, the set S_0 is defined to be the set of all points. Then, XArray will iteratively find the maximal value of y such that there exists an x where the query $(\leq x, \leq y)$ is "sparse" in S_{i-1}. Here, "sparse" just means "not dense." Call this value y_i. Also, let x_i denote the largest x such that $(\leq x, \leq y_i)$ is sparse in S_{i-1}.

Then, the following updates will be performed.

- $P_{i-1} \leftarrow S_{i-1} \cap (\leq x_i, *)$
- $S_i \leftarrow S_{i-1} \cap ((*, \leq y_i) \cup (> x_i, *))$
- $XArray \leftarrow XArray + P_{i-1}$

This procedure is repeated until S_i reaches a length that is less than some constant $B = O(1)$, and the final S_i is appended to the end of XArray. B is the base case size. Thus, the final XArray is simply the concatenation of the sets $P_0, P_1, \ldots, P_{i-1}, S_i$.

This construction is useful because within each chunk P_i, any query is guaranteed to be dense. If we let the number of points in the output of the query $(\leq x, \leq y)$ be k, then we will read $O(k)$ points in the chunk of the XArray that we end up in. Scanning through the XArray will take $O(\lceil k/B \rceil)$ cache-oblivious disk reads. The constant factor depends on the parameters α and B. In our case, we chose $\alpha = 2$ and $B = 10$.

We implemented a more basic version of the data structure construction procedure that takes $O(N^2)$ time. In our construction, we loop through every point when looking for the maximal value x_i given a potential value of y_i. We include an optimization where we stop the loop when we know that there are not enough "bad" points left to cause any larger values of x to be sparse queries for a given y_i. For example, when y_i is the second highest value of y over all points, there can be at most 1 "bad" point (which is the point with the maximal y value). Then, we know that all queries $(\leq x, \leq y_i)$ are dense if the first point (x_1, y_1) we examine is not the point with the maximal y value.

It is possible to construct the XArray in $O(N \log_B N)$ disk accesses and $O(N \log N)$ time overall [4]. We did not implement that in this project.

2.1.3 Space Usage

As both the VEBTree and XArray take $O(N)$ space, the 2-sided Coors data structure takes $O(N)$ space total.

2.2 Coors 3-Sided

For the 3-sided case, we again store the points in a perfect BST in VEB order, but this time with all the data stored at the leaves. We set the key of each internal node to the min of

its right subtree, and store at each internal node pointers to two Coors 2-sided structures of mirrored orientations on the leaves of its subtree. Since each 2-sided structure at the internal nodes takes $O(\text{size of subtree})$ space and each leaf node belongs to $O(\lg N)$ subtrees, the total space complexity of the 3-sided case is $O(N \lg N)$.

2.3 Coors 4-Sided

Our implementation of the 4-sided case is very similar to that of the three sided case, where we construct a perfect BST with data at leaves and pointers from internal nodes to two mirrored 3-sided structures on its subtree data. It's straightforward can show that the space complexity of the 4-sided Coors structure is $O(N \lg^2 N)$.

However, it is worthwhile to note that the original paper gave a construction of the 4-sided Coors structure that shaves off an extra $O(\lg \lg N)$ factor in the space usage [1], so their final Coors2D data structure uses $O(N \lg^2 N / \lg \lg N)$ space. This can be achieved by contracting internal subtrees of the perfect BST such that each internal node has degree $\sqrt{\lg N}$. Then, the height of our search tree in the last layer decreases by a $O(\lg \lg N)$ factor, which translates to a $O(\lg \lg N)$ factor reduction in space.

We opted for the more straightforward implementation that stores two 3-sided structures at each internal node for simplicity's sake.

3 Benchmark Baselines

We implemented 5 different data structures as baselines to compare our implementation of Coors2D against, which we list below:

1. Naive Linear Scan
2. 1D Range Tree
3. 1D Range Tree with VEB layout
4. 2D Range Tree
5. 2D Range Tree with VEB layout

We provide a brief overview on the implementation and asymptotic performances of these baseline data structures in the following sections.

3.1 Naive Linear Scan

We store all the points in an array. To query for points in an orthogonal range, we scan through all the points in the array, and filter for the points that are within the query box. The data structure takes $O(N)$ space and $O(\lceil N/B \rceil)$ memory transfers per query.

3.2 1D Range Tree

This data structure is represented by a binary search tree and an array on the set of points. We store all the points in a perfect BST keyed by the x-coordinate, and store an additional copy of each point in an array sorted by x. We also store pointers from each node in the BST to its copy in the array. To query for points in a box, we first search for the lower x boundary in the BST, follow its pointer to the array of copies, and scan through the array until we reach a point whose x value is larger than the upper x boundary. As we scan through the array, we filter for points whose y value lies also within the query box, and output them as our solution. Later on, we will refer to points with y values outside the query box as **incorrect points**.

We implemented two versions of this data structure, one with the binary search tree stored in VEB order, and one with the binary search tree stored in x-sorted order. While this data structure has a worst case of $O(\lceil N/B \rceil)$ memory transfers per query, its actual performance depends on the number of incorrect points we have to filter through as we scan through the copy array (note that we are still able to take advantage of the $1/B$ reduction factor from linear scans). Meanwhile, the search step has a fixed asymptotic performance of $O(\log_B N)$ memory transfers for the cache-oblivious case, or $O(\lg N)$ memory transfers otherwise. As the BST and the array both cost $O(N)$ space, this data structure takes $O(N)$ space total.

3.3 2D Range Tree

Our implementation follows that of the standard 2D range tree with $O(\lg^2 N + k)$ query time. We first construct a perfect BST keyed by the y-coordinate with data stored at leaves, where the key of each internal node is set to the min of its right subtree. At each internal node, we store a 1D range tree on the points in its corresponding subtree.

Then, to perform an orthogonal range query, we search for the lower and upper y boundaries of the box, and perform a 1D range search in the x coordinate on the relevant subtrees that hang off the two root to leaf paths. As there are $O(\lg N)$ subtrees that hang off the root to leaf path, and performing a 1D range querying on each subtree takes $O(\log_B N + \lceil k/B \rceil)$ memory transfers in the cache-oblivious case, or $O(\lg N + \lceil k/B \rceil)$ memory transfers otherwise. Hence, the asymptotic performance of this data structure is $O(\lg N \log_B N + \lceil k/B \rceil)$ memory transfers per query in the cache-oblivious case, or $O(\lg^2 N + \lceil k/B \rceil)$ memory transfers otherwise.

4 External Memory Setup

To test the performance of our Python implementation of the Coors2D data structure, we implemented a simple LRU cache (with variable dimensions) and a disk array to mimic the external memory model. By integrating our data structure implementations with the external memory setup, we can artificially count the number of memory transfers incurred during each query step. This simple model allows us to capture the cache efficiency of our implementation of the Coors2D data structure relative to other baselines.

Our external memory setup is specified by two parameters, the block size B and the cache size M. We represent the disk as a dynamic array of unbounded size, and each cell in

the disk array represents one word in memory. In our implementation, we require that each cell store a node object, which contains $O(1)$ attributes and pointers (e.g. left/right child pointers, key/value, etc). Hence, we are assuming that every node object can be completely contained in one machine word in our model.

We represent the cache as a hashtable of memory blocks with capacity M/B, augmented with block priorities based on last touch time. Whenever we access a node object, we first check if its containing block is in the cache. If yes, we simply return the node object; otherwise, we evict the least-recently-used cache block if the cache is full, insert the containing block, and return the node object. This external memory setup allows us to actively keep track of the number of cache misses, as well as the total number of cell reads, when we perform orthogonal range queries.

As a technical implementation detail, we assume 2 extra memory cells in the cache in addition to the M/B cache blocks, which are implicitly used for temporary storage in our query algorithms (one can also treat this as an additional $O(1)$ storage in the CPU register).

Our main objective is to test the cache efficiency of the Coors2D data structure compared to baselines under our simplified memory model, which is captured by the memory transfer counts that we keep track of in our external memory setup.

5 Experiments

We designed various experiments to empirically measure the performance of the Coors2D.

In general, our methodology was to create a point set, and generate a copy of each of the 5 baselines data structures and one copy of Coors for that point set. Then, we would generate many "big box" queries and also many "small box" queries and call the same queries on all 6 data structures. We would measure the performance of each of the 6 data structures in terms of disk accesses and cell probes.

Our main evaluation metric was the number of disk accesses. These are tracked by our memory model on all node reads.

We also looked at the number of cell probes required by each algorithm, which aims to count the number of pointer operations and write operations that are needed to access and output all the nodes in a given query. This gives some sense of the runtime of the algorithm if we ignore disk accesses.

We did not evaluate actual wall-clock time of these algorithms.

Specifically, points in each point set are generated uniformly at random inside the box $[-10000, 10000] \times [-10000, 10000]$. "Big box" queries have their upper and lower bounds in each coordinate chosen uniformly from $[-12000, 12000]$. "Small box" queries have their upper and lower bounds in each coordinate chosen uniformly from $[-100, 100]$, but then the center of the box itself is translated uniformly at random by some amount x, y where $x, y \in [-9900, 9900]$. The final results show the average performance over 1000 queries.

To test the relative efficacy of Coors with other data structures in different settings, we ran two primary sets of experiments. First, we ran varying memory experiments, in which we fixed the number of points to be 2000. Then, we varied the memory configurations. We chose 5 different memory configurations (cache size, block size), with the block size being roughly the square root of the cache size. The configurations we chose are

$(32, 8), (64, 8), (128, 8), (256, 16), (1024, 32)$.

Next, we considered the effect of changing the number of points that each data structure stored. We fixed the memory configuration to be $(32, 8)$.

We chose 5 different numbers of points - $1000, 2000, 5000, 10000, 20000$. We were limited from testing significantly larger numbers of points due to the slow construction time of the Coors data structure.

6 Evaluation Results

We compared the relative efficiency of Coors2D compared to our benchmark baselines on the above metrics. As a note, "XBST" refers to the 1D Range Tree with VEB layout data structure described above, while "Range Tree" refers to the 2D Range Tree with VEB layout.

6.1 Varying Memory Configurations

The number of cell probes required didn't vary much for different memory configurations, so we omit the plots for those results. Instead, we focus on the number of disk accesses required.

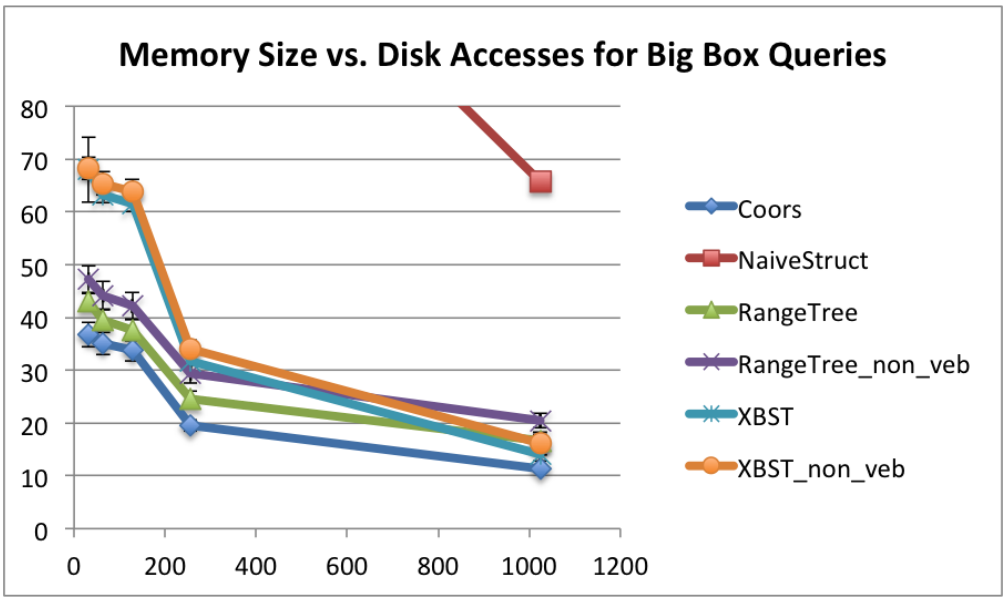

Figure 2: Coors is the best data structure for every possible setting of memory in this case. (Note that the first few data points for our naive data structure are not shown in this range as it performs much worse than all the other data structures.)

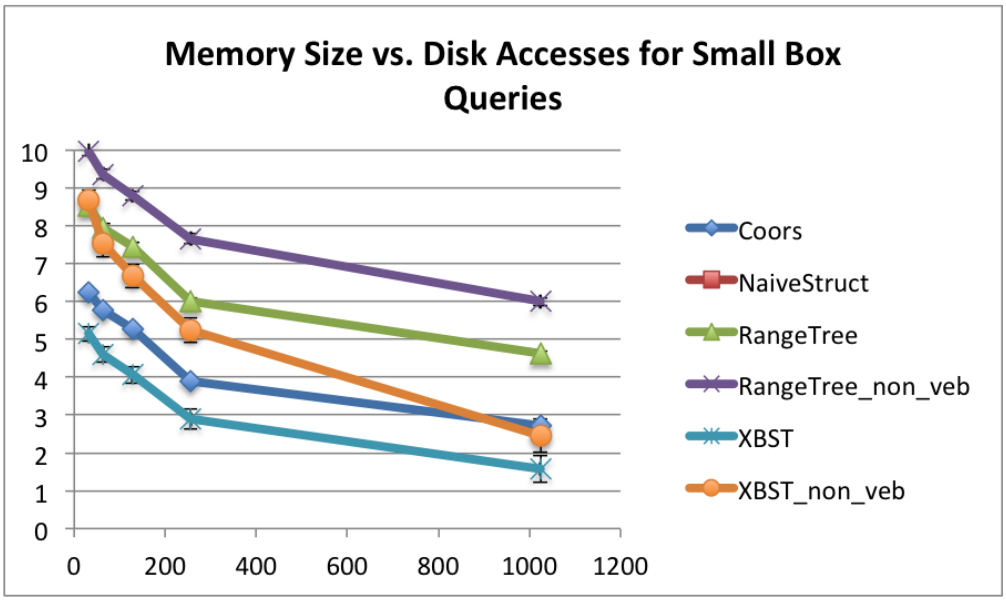

Figure 3: Coors is close to the second best data structure for every memory configuration.

6.2 Varying Number of Points

Here, we plot all the results, starting with results for cell probes.

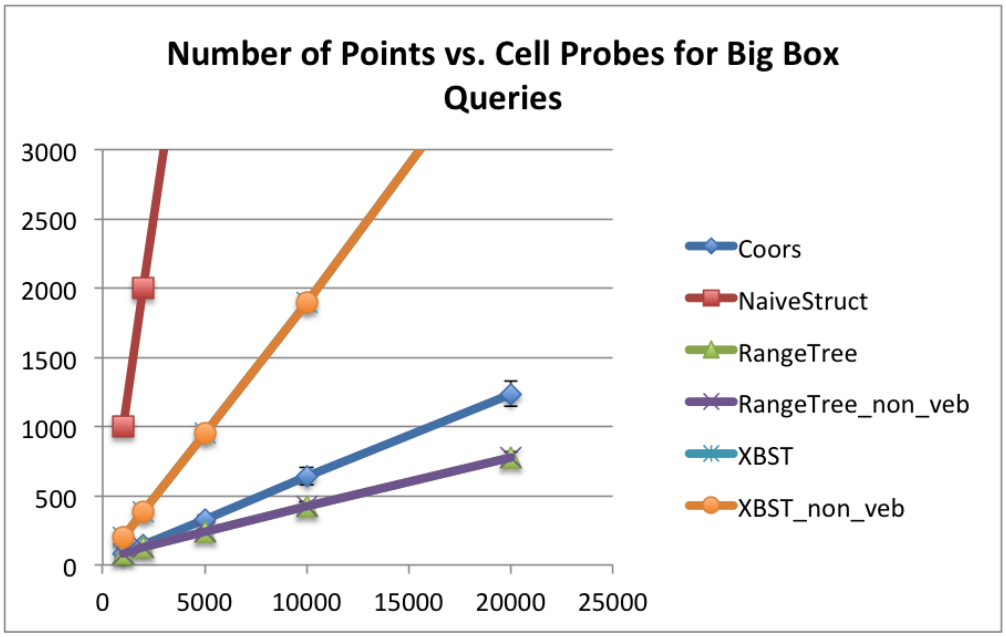

Figure 4: The number of cell probes for big box queries scales roughly linearly with the number of points, as expected (since the size of output term dominates). Coors seems to have a slightly higher constant factor in front of the output size than 2D Range Trees and performs slightly worse in terms of cell probes.

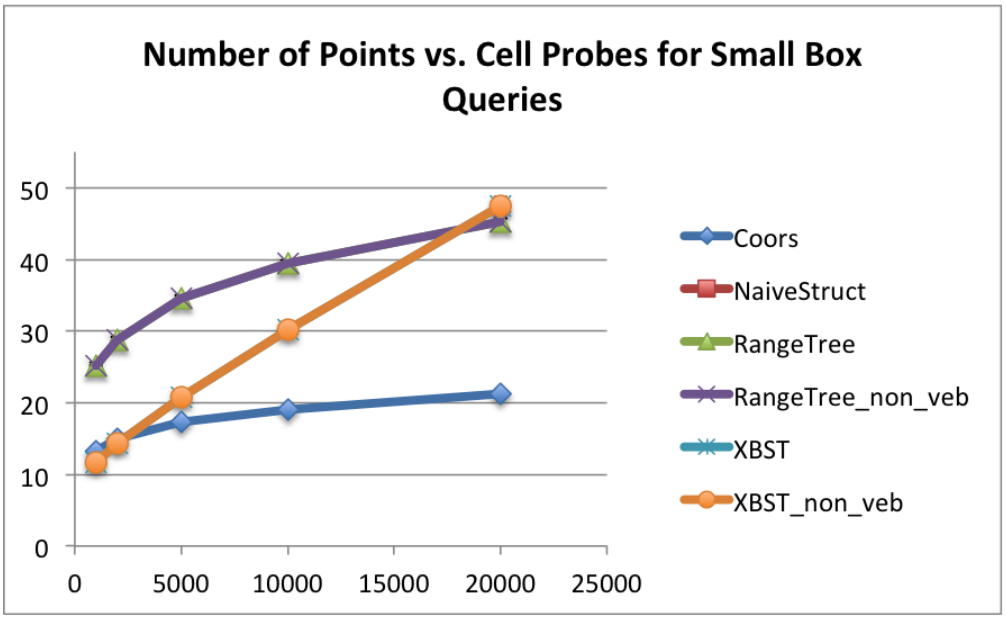

Figure 5: The number of cell probes is dominated by the log term for small box queries as opposed to the output size. 2D Range Trees have a $\log^2$ term, and thus Coors, which only has a log term, outperforms it. 1D Range Trees are good for a smaller number of points, but once the number of **incorrect points** becomes too big, its performance degrades linearly with the number of points.

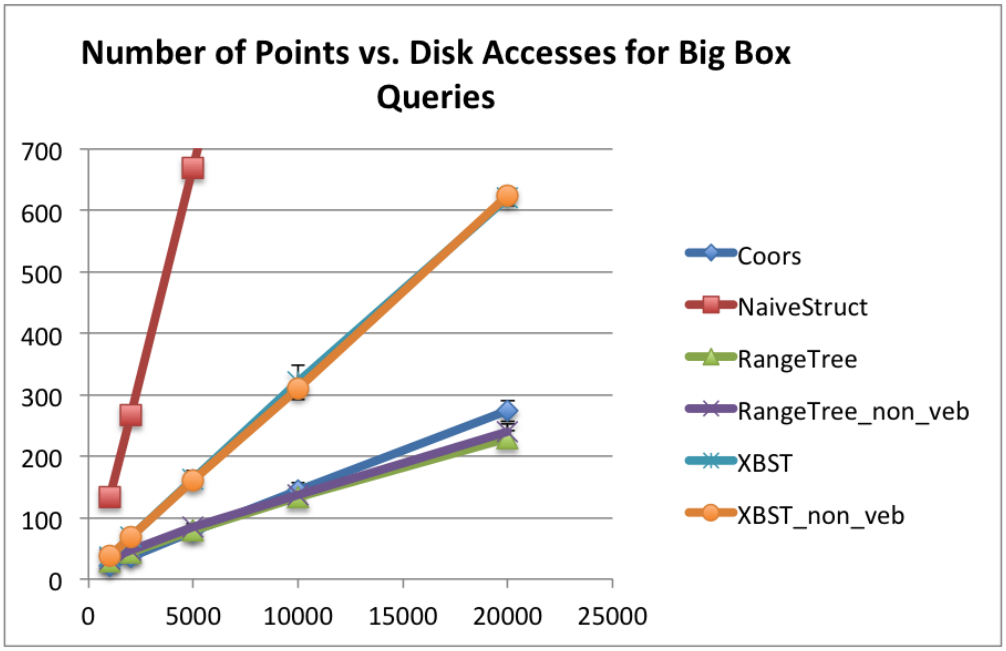

Figure 6: The number of disk accesses for big box queries scales roughly linearly with the number of points, as expected. Coors has a slightly higher constant factor in front of the output size than 2D Range Trees and performs slightly worse in terms of disk accesses.

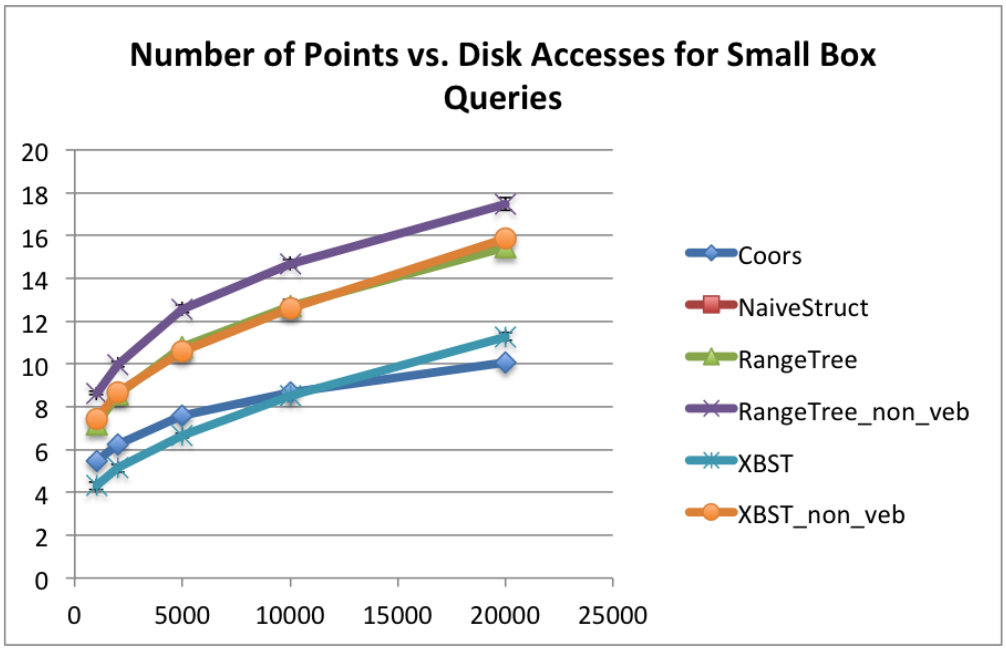

Figure 7: The number of disk accesses is dominated by the log term for small box queries as opposed to the output size. 2D Range trees have a $\log^2$ term, and thus Coors, which only has a log term, outperforms it. 1D Range Trees is good for a smaller number of points, but once the number of **incorrect points** becomes too big, its performance drops off. The decrease in performance is not as noticeable as in the case of cell probes, as the linear increase is divided by the block size $B = 8$.

7 Future Work

Below we outline some areas that we would like to work on in the future as an extension to this project:

1. Implement $O(N \lg N)$ time algorithm for constructing the XArray in Coors 2-sided, as outlined in paper[4].
2. Implement Coors 4-sided data structure as outlined in the original paper[1], which uses $O(N \lg^2 N / \lg \lg N)$ space.
3. Run experiments with different values of α (query density threshold) and base case size for constructing the XArray in Coors 2-sided.
4. Run experiments with larger number of points and queries.
5. Run experiments on memory setups with multiple levels of cache.
6. Measure empirical wall-clock query times of each structure in our experiments.
7. Measure space usage of each data structure in our experiments.

8. Refactor parts of our code and design to improve the integrity of our abstracted memory model.

References

[1] L. Arge, G. S. Brodal, R. Fagerberg, and M. Laustsen, "Cache-oblivious planar orthogonal range searching and counting," in *Proceedings of the Twenty-first Annual Symposium on Computational Geometry*, SCG '05, (New York, NY, USA), pp. 160–169, ACM, 2005.

[2] M. de Berg, M. van Kreveld, M. Overmars, and O. C. Schwarzkopf, *Orthogonal Range Searching*, pp. 95–120. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000.

[3] E. Demaine, "Lecture notes for 6.851 advanced data structures," 2012.

[4] L. Arge and N. Zeh, "Simple and semi-dynamic structures for cache-oblivious planar orthogonal range searching," in *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*, SCG '06, (New York, NY, USA), pp. 158–166, ACM, 2006.