HODOR: HODOR On-Disk Orthogonal Range-trees

Stephanie Wang (swang93@mit.edu)
Bennett Cyphers (bcyphers@mit.edu)
Katie Siegel (ksiegel@mit.edu)

6.851 Final Report

May 18, 2014

Abstract

Orthogonal range trees are simple to implement in memory, but implementing them as cache aware, on-disk data structures is a more involved task. In this paper, we describe HODOR, a system which allows efficient range queries on static data sets indexed with comparable values in an arbitrary number of dimensions. Since disk reads are costly, we analyzed our structure in terms of the number of reads per query. In our analysis model, forward-seeking reads are faster than backward-seeking reads, so we attempt to minimize the latter. HODOR achieves range queries with a polylogarithmic number of disk reads, and never has to seek backwards on disk during a single range query. We successfully implemented a version of HODOR in Python, and have posted the code for public use on Github.

1 Introduction

Web applications today often require batch queries on large datasets in two or more dimensions. Databases typically support efficient range querying on one primary key at a time. Multi-dimensional range queries, however, are expensive, and common implementations require $O(n)$ time.

As we saw in class, orthogonal range trees (ORTs) can achieve range queries on d dimensions in $O(\log^d n + k)$ time, where k is the size of the result set [1]. These trees have been implemented in memory with success, including as an extension to the popular database Redis [2]. In-memory databases are useful for smaller datasets, or for companies with enough resources to operate expansive data centers. However, for data sets which are too large to fit in main memory, magnetic hard disks remain the most economical long-term storage option. On-disk data structures present a unique set of challenges requiring a unique solution. To this end, we designed HODOR, an on-disk implementation of ORTs.

The main obstacle to an on-disk implementation of ORTs is the added overhead in disk access time. Therefore, in this paper, we explore different methods of optimizing disk I/O accesses, in terms of disk read frequency and overall read time. First, we present our overall design, and specific modifications we made to the standard ORT structure. Second, we describe our serialization protocols for the data structure. Third, we analyze each query in terms of both total disk accesses and a new metric, *back seeks*, which we introduce here. We describe how to achieve faster runtime by eliminating back seeks from range queries. Finally, we discuss a Python implentation of HODOR, and discuss how our findings from this project may contribute to future work.

2 Design

In this section, we detail the design of HODOR, and describe the decisions we made in the process.

2.1 Goals

The primary goal of HODOR is to allow efficient range queries of very large datasets in an arbitrary number of dimensions. We designed the structure to perform on machines with a limited amount of memory and a large amount of storage on a relatively slow hard disk. As such, we wanted to minimize the time spent reading data from disk, but were generally unconcerned with in-memory comparison operations. We optimized for this in two ways: first, by minimizing total disk reads, and second, by chaining consecutive disk accesses so that they may be executed as efficiently as possible.

HODOR requires a relatively long time to construct its tree and serialize it to disk, so the results presented in this paper are for queries on static trees only. In addition, there are well-documented techniques, such as fractional cascading, which can augment range trees to speed up queries by a logarithmic factor. However, the goal of this project was more to determine whether we could minimize the overhead in disk reads during a query than to build the fastest structure possible.

2.2 Range B-Trees

The standard data structures used for implementing many table-style databases are the B Tree, and a modified version, the B+ Tree. As a quick overview, a B Tree is a search tree with a branching factor of B, i.e., each node points to at most B children. A B+ Tree is an extension of the B Tree in which all data is stored at the leaves of the tree, and each leaf has a pointer to its immediate predecessor leaf, `prev`. Each leaf stores B data points instead of node pointers. The logic behind the B Tree's design is that, since the act of loading children from disk into memory is often the bottleneck on database search performance, each node should load as many children as it can into memory at once. B can then be tweaked to optimize for cache sizes on different machines. The predecessor pointers in a B+ tree allow easy range queries in one dimension: once we find the leaf with the maximum value in the range, we can walk backward along the predecessor leaves until we reach the one containing the minimum value, and return all leaves touched on the walk.

The orthogonal range tree is a structure covered in 6.851 which allows $O(\log^d n + k)$ range queries on d dimensions. For HODOR, we extend the B+ Tree structure with orthogonal, recursive linked trees to create B+ Orthogonal Range Trees (BORTs). Each node n in a BORT has a dimension d, a set of B child pointers sorted in dimension d, and a pointer to another BORT containing all of the data points contained in n's subtree, known as n's *linked tree*. When n is loaded from disk into memory, it brings with it all B child pointers and one link pointer. However, as we will describe below, not all disk accesses are equally costly, and we can improve runtimes in practice by laying out serialized nodes intelligently on disk.

2.3 Node Serialization

In order to store BORTs on disk, we require an efficient method to serialize and deserialize the data structure. In the Python implementation of HODOR, we represent a `RangeTree` instance as a list of serialized nodes, which may be instances of `RangeNode` or `RangeLeaf`. This list is stored in a *tree file*, which can be written to and read by a `Serializer` class.

During preprocessing, when building the tree from a set of datapoints, `Serializer` is initialized in write mode to build the tree file. In write mode, `Serializer` exposes a `dumps(node)` method that accepts a node instance, serializes the node, and adds it to the top of a *node stack*, a temporary data structure which holds serialized nodes before the whole tree is flushed to disk. This method also assigns the node a pointer into the tree file, equal to the number of nodes which have already been serialized. As the size of the node stack is likely to exceed the total amount of memory available, the stack is stored on disk in a temporary file while building the tree.

Once every node has been serialized and placed on the stack, we call `Serializer.flush()`. This method writes the serialized nodes to disk in the reverse order that they were created with `dumps`, by popping each element off of the node stack and appending it to a tree file. Once tree serialization is complete, the temporary stack file can be deleted. From this point onwards, `Serializer` is in read mode, and can only be used to read from the tree.

In read mode, `Serializer.loads(pointer)` can be called to deserialize a single node into a Python node instance, given its pointer into the tree file. So, to give a parent access to its child, we simply store the child's tree file pointer as an attribute of the parent. `loads` is implemented by seeking to the pointer in the tree file and reading out a single node's worth of bytes.

The seeking method varies by serializer. In HODOR, we test two main node serialization methods. The first is to use a delimiting character, such as a newline, between nodes. The second is to pack nodes into fixed-length strings, called *blocks*.

Delimited strings are convenient because they support arbitrarily sized datapoint values. However, seeks may be costly in performance even with Python optimizations like the line iterator or the `linecache` module [3].

On the other hand, when nodes are stored as blocks, a node's exact byte position can be calculated from its file pointer and jumped to directly, yielding a lower performance cost for `loads`. However, fixed-length blocks limit the type and size of data that can be stored. This can be solved by storing the data points themselves as delimited strings in a separate *data file* and storing pointers to this data in the leaves of the BORT. A range query can return a set of pointers into the data file that can be sorted and used to access the actual data sequentially from the file. This method requires one extra disk read for each datapoint in the result set, but no extra back seeks. Our final implentation of HODOR uses the block serialization method.

2.4 Tree Serialization

Tree serialization is executed while preprocessing the datapoints into a tree, so that the order of nodes in the tree file is the same as the order in which HODOR preprocesses them. The tree is built recursively, from the bottom up. HODOR starts with a set of children. If these are leaves, then HODOR sorts them in the primary dimension of the current tree. If the full set of leaves is too large to fit in memory, the algorithm can use funnel sort to arrange them on disk [4]. Next, HODOR partitions the children into clusters of at most *B* items. For each of these clusters, HODOR creates a parent node and recurses to build the parent's linked tree in the next dimension, if any dimensions remain. If there is only one parent, then the parent is the root of the tree, and HODOR is done. Otherwise, HODOR recurses on the parents.

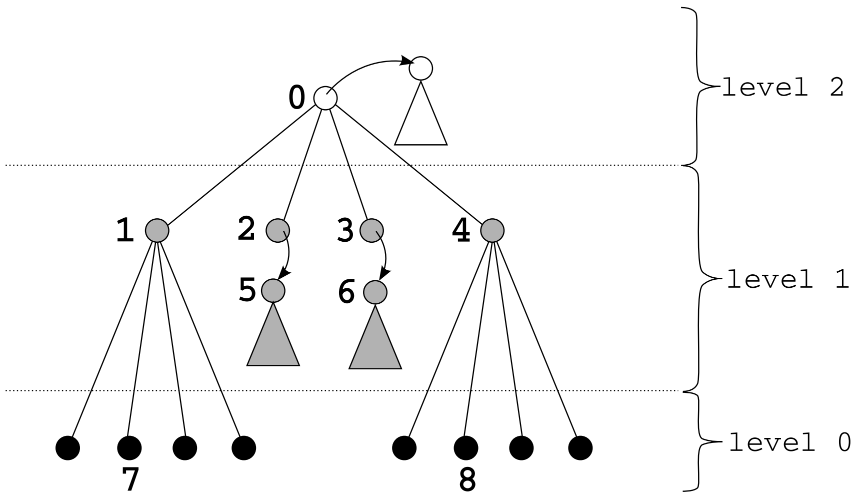

Figure 1: The levels structure of the serialization order in the context of an ORT. The nodes of the tree in the first dimension of this ORT are serialized in three levels, each level in order of min to max or left to right. First, level 0 is made up of the leaves in the first dimension. Second, level 1 is made up of the leaves' parents and each of the parent's linked subtrees in the next dimension. Third, level 2 is made up of the root and its linked subtree, which contains all *n* values. The levels are serialized in this order as the tree is built, but are flushed to disk in reverse order (see Figure 2).

HODOR serializes the nodes in the same order that they are created. In the recursion for a set of children, all children are serialized first, followed by the linked subtree for each parent node, followed by the parent node itself. Each child is serialized before its parent, and each parent is serialized immediately after its linked subtree in the next dimension. This forms a series of levels [5] for each tree (see Figure 1). Consider a BORT t. Each node n in t falls in the same level as all of n's siblings. The entirety of n's linked subtree, which is indexed in the next dimension, also falls in this same level of t. All of n's children and their linked subtrees fall in the level below n's, and n's parent falls in the level above.

All the leaves of the tree in the first dimension are in that tree's lowest level, and are therefore serialized first. The root of the same tree is in the tree's highest level, and is therefore serialized last. The leaves and root of each linked subtree follow the same pattern.

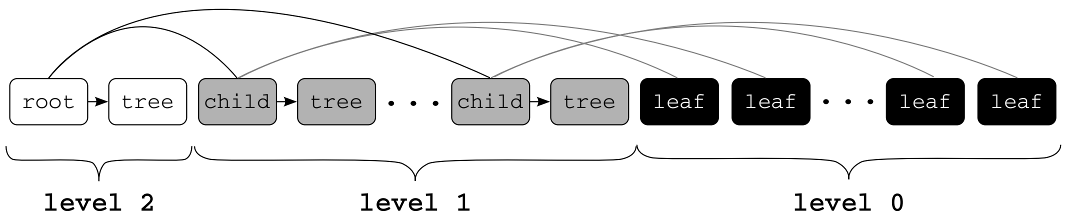

Figure 2: The final disk layout of the same ORT shown in Figure 1 after flushing to disk. The nodes that were pushed onto the node stack in the order given by figure 1 are popped off and appended to the final tree file in the reverse order, shown here. Level 2, the root in figure 1, is now at the beginning of the file, followed by level 1, and finally level 0, the leaves in figure 1. Each of the levels is now in reverse order, from max to min, and each node is immediately followed by its linked subtree in the next dimension.

Once we finish building the tree, we flush the nodes in the stack to the tree file on disk in reverse order of serialization, as described above. This way, each node will fall after all of its ancestors (see Figure 2). In the final disk layout, the root of the tree in the first dimension is the first node in the file, and the leaves of the same tree are last in the file. This layout allows us to always read forward through the file as we traverse downwards from any node in a range query. All of the siblings in one level were originally pushed onto the stack in order, so when they are popped from the stack, they will be written in reverse order. Then, within each level, the maximum-valued sibling now comes first and the minimum-valued sibling now comes last.

3 Analysis

In this section, we analyze the performance of queries in HODOR with respect to disk reads. A disk read is needed to load any serialized node into memory. Therefore, the algorithm requires one disk read to access any node it has not previously seen.

In our analysis, we focus on measuring disk reads in two categories. Assuming we are reading blocks from one continuous file on disk, *forward seeks* are defined as reads from a location after the current file pointer. *Back seeks* are reads from a location before the current pointer. In our disk-memory model, forward seeks are, in general, much faster than back seeks. When accessing file blocks in sequential order, the blocks can be loaded into memory at speeds approaching the disk's maximum read speed. However, if a file block at position $i-1$ is accessed immediately after the file block at i, the disk must make a complete revolution before block $i-1$ can be loaded. On a typical drive spinning at 7200 RPM, this revolution takes over 8 milliseconds.

In addition to minimizing total disk reads, we tried to eliminate back seeks wherever possible when designing HODOR. We will analyze our data structures in terms of both total reads and total back seeks in the sections below.

3.1 Disk Read Analysis

We will now examine the complexity of a search on a fully formed BORT, in terms of total disk reads. The logic for performing a range query on a (non-leaf) node is shown in Algorithm 1.

Algorithm 1 Recursively find all elements in a multidimensional range from a node's subtree.

Precondition: RANGES is a dictionary of (start, end) range tuples indexed by dimension, n is a node, and d is an integer representation of the dimension by which n is indexed. 0 is the last dimension.

```
function RangeQuery(Ranges, $n$, $d$)
    $start, end \leftarrow$ Ranges[$d$]
    if $d = 0$ then
        $leaf_{end} \leftarrow$ GetLeafContaining($end$)    ▷ Search for the leaf containing $end$
        return GetRange($leaf_{end}$, $start$)    ▷ Get all leaves in this dimension's range

    $r \leftarrow$ Null
    $c_L \leftarrow$ GetChildContaining($start$)
    $c_R \leftarrow$ GetChildContaining($end$)
    Load($c_R$)

    for all $c_i$ in ChildrenInRange($n$, $end$, $start$) do
        Load($c_i$)
        $l_i \leftarrow$ LinkedTree($c_i$)    ▷ Get the child's linked subtree
        $r \leftarrow r$ + RangeQuery(Ranges, $l_i$, $d-1$)

    Load($c_L$)
    $r \leftarrow r$ + RangeQuery($c_R$)
    $r \leftarrow r$ + RangeQuery($c_L$)
    return $r$
```

Let m be the total number of elements in the subtree of node n. In the base case, the node is in the last dimension, and the function is reduced to a search and a GETRANGE call. GETRANGE is a function, shown in Algorithm 2, which returns all leaves in a range from this node's subtree. It is accomplished by starting from the leaf containing the maximum value and walking backwards along the leaf's `prev` pointers until the leaf containing the minimum value is found. This entire operation takes $O(\log_B m + k)$ reads: the search down the tree touches $\log_B m$ nodes, and the walk touches k leaves.

Algorithm 2 Find all elements in a multidimensional range from a tree leaf.

```
Precondition: $n$ is a leaf in the last dimension and has data representing $B$ elements of the
    tree. $start$ is the startpoint of the queried interval in the last dimension.
1: function GETRANGE($n$, $start$)
2:     $data \leftarrow$ data at $n$ that is greater than or equal to $start$
3:     if $n.min \geq start$ then
4:         $data \leftarrow data$ + GETRANGE($n.prev()$, $start$)        ▷ Load the leaf previous to $n$
5:     return $data$
```

The algorithm makes two different types of recursive calls. First, the $O(B)$ nodes which are fully contained in the range (start, end) are loaded into memory, and the algorithm recurses on the linked tree of each one. These *linked-tree recursions* can happen at most d times in any query before the last dimension is reached. Second, the algorithm recurses on a constant number of its children (at most two) which contain the start and end values of the range. Each of these *child node recursions* operates on $O(\frac{m}{B})$ elements.

Once RANGEQUERY is called on a leaf, the leaf will either return some subset of its data, which is already loaded, or recurse on a linked leaf in the next dimension. This requires one disk read per dimension, for $O(d)$ reads in total.

Let the disk-read complexity of the algorithm with m nodes and d dimensions equal $\mathrm{T}(m,d)$. Each of the constant number of child node recursions operates on $\frac{m}{B}$ elements, and takes $\mathrm{T}(\frac{m}{B},d)$ reads. Using analysis similar to that used in Lecture 3, this evaluates to $O(\log_B m)$ reads total for any tree [1]. In the last dimension, there is also the possibility of a leaf-to-leaf walk, so the total complexity becomes $O(\log_B m + k)$ reads.

Each one of the linked-tree node recursions operates on $O(\frac{m}{B})$ elements, on $d-1$ dimensions, so the total reads from all $O(B)$ of these operations is at most $B\cdot\ \mathrm{T}(\frac{m}{B}, d-1)$. Therefore, again using logic similar to the runtime analysis in lecture 3, we see that running the entire algorithm from the root node of a BORT makes $O(B^{d-1}\cdot\log_B^d m + k)$ reads from the disk. In the sections below, we will analyze how we can bound the *types* of disk reads we have to make.

3.2 Back Seek Analysis

We want to order the node accesses during a range query so that we do as many consecutive forward seeks as possible, while eliminating back seeks. Here we analyze the worst-case back seek count for the given range query algorithm by counting the number of back seeks needed for each recursion, using figure 1 as a reference.

As discussed in section 2.4, the nodes are structured into recursive levels on disk, where each level contains a node, all of its siblings, and its linked subtree in the next dimension. The number of back seeks we do during each recursion depends on the order in which nodes of the same level are accessed and the order in which the levels themselves are accessed. Each level is sorted in reverse order, so if a node is accessed after its left sibling, it will cost one back seek. For each tree, the levels themselves are sorted in order from “highest” (root) to “lowest” (leaves), so if a node’s ancestor is accessed after the node itself, it will also cost one back seek.

The range query recursion starts at the root of a subtree, illustrated by node 0 in figure 1. Because a node and all of its siblings lie in the same level, when the range query accesses the root’s children, all work is done in the same level. This is the level after the root’s, or level 1 in figure 1. Each level is sorted in reverse order in the file, so if the range query accesses the children from max to min (right to left in the figure), then no back seeks are needed to load the root’s children. Because the range query always traverses downwards, we never return to the root’s level (level 2) once we access the children’s level (level 1).

We analyze the back seeks needed for the two types of recursive calls on a child by examining the node and level access order. The linked-tree recursion case is illustrated with node 2. Suppose node 2’s entire range is contained in the queried interval, so we recurse on its linked node, node 5. Note that the linked subtree at node 5 falls in the same level as node 2, so the recursive call on node 5 stays within level 1. Also, since a node occurs immediately before its linked subtree in the disk layout, we can make the recursive call to node 5 directly after loading node 2 in order to avoid back seeks within the same level. Thus, a linked-tree recursion costs nothing in back seeks.

The child node recursion case can be illustrated with node 1, in level 1. Suppose that node 1 contains `start`, so we recurse on node 1 and load its children in level 0. Level 1 comes before level 0 on disk, so we can load node 1’s children by seeking forward. However, if the range query ever has to access level 1 again, a back seek will be needed to return from level 0. This happens whenever the recursive call on a node is followed by a recursive call on one of that node’s siblings. For example, the recursive call on node 1 will traverse down the tree, ending at some level below 1 in the tree file. If we then recurse on one of the nodes 2-4, it will cost one back seek to return back to level 1. In other words, during the recursion on node 0, if more than one child node recursive call is made, then every call but the last will contribute one back seek.

Lemma. *According to the given algorithm, there will be $O(\log_B^{d-1} n)$ back seeks, where d is the number of dimensions and B is the branching factor.*

Proof. As we noted above, during each recursion step, any child node recursive call that is followed by another recursive call in the same step contributes one back seek. A child node recursive call can only be called on a child whose range is not completely contained within the queried interval, and whose range contains the `start` or `end` point. So, for each recursion, there will be at most two such recursive calls. If there are zero or one such calls, then there are no back seeks. If there are two, there is exactly one back seek.

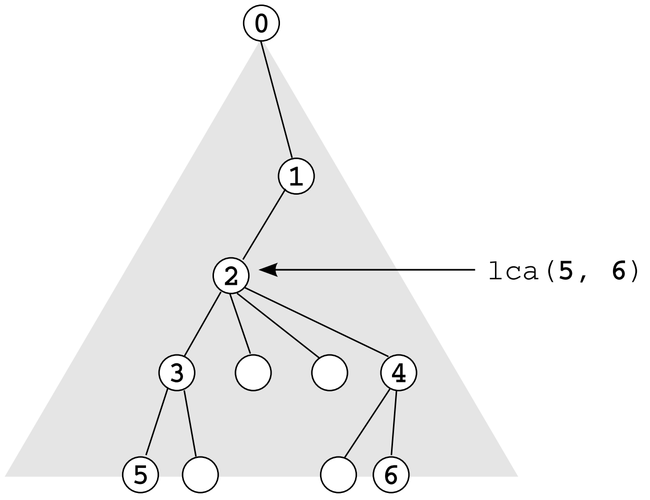

Figure 3: A range query on one tree layer of the entire BORT can be modeled as a search for the predecessor of *start*, leaf 5, and the successor of *end*, leaf 6 [1]. The unlabeled nodes are those whose linked subtrees must be queried. The least common ancestor of leaves 5 and 6 is node 2. During the RANGEQUERY recursion on node 2, the algorithm makes recursive calls on two of node 2's children, one on node 3 and the other on node 4. The RANGEQUERY recursions on all other nodes in this layer, including 0, 1, 3 and 4, require only one child node recursive call.

For a range query on a tree, there is at most one recursive step that requires making two such recursive calls. This happens when the root node of the recursion is the least common ancestor (`lca`) of the predecessor of `start` and the successor of `end` [1]. This is illustrated by node 2, the `lca` of nodes 5 and 6, in figure 3. By definition, `lca` has one child (node 3) containing `start` and a different child (node 4) containing `end`, each requiring one recursive call. Each node that is an ancestor of `lca` (nodes 0 and 1) covers `lca`'s range, so it has a child that contains both `start` and `end`. Only one recursive call on this child is needed. Each node that is a descendant of `lca` (nodes 3 and 4) can have at most one child that contains either `start` or `end`, not both, since `start` and `end` are contained in different children of `lca`.

Recursing on a least common ancestor can happen at most once during a range query on a linked tree. Therefore, the total number of back seeks performed by a range query is bounded above by the number of distinct linked trees that are queried. Using a similar analysis to the one given in lecture 3, this is $O(1)$ for 1 dimension, since there is only one tree. It is $O(\log_B n)$ for 2 dimensions, since we recurse on at most $O(\log_B n)$ linked subtrees in the second dimension. Recursing on d, we get $O(\log_B^{d-1} n)$ for d dimensions.

□

3.3 Eliminating Back Seeks

The given range query algorithm requires back seeks because of accesses to a level that we have already traversed past. We can eliminate these accesses by ensuring that by the time we traverse downward from some node, all work that will ever be done in that node's level has already finished. The work done on that node's level is the work involving that node, its children, and its linked subtree in the next dimension. Here, we propose a modified algorithm that accounts for the level structure and analyze its complexity in back seeks.

Like the naïve algorithm, the modified algorithm starts at the root of the tree in the first dimension. However, rather than recursing on one node at a time, the modified algorithm recurses on a set of nodes. Similar to before, the algorithm determines which of the current set of nodes cover ranges that fall completely within the queried interval. For these nodes, the algorithm makes a recursive call to the linked nodes, as in the naïve algorithm. However, for a child node recursive call, when a node in the current set has a range containing `start` or `end`, the algorithm does not make the call immediately. Instead, the algorithm collects the children of each such node in the current set, and recurses on the union of the children.

To show that the modified algorithm does not require any back seeks, we first note that for any recursive call, the nodes in the current set are all in the same level. Then, if we do our work for each node in order of max to min, the order in which the siblings are laid out on disk, we will avoid all back seeks between nodes in the current set.

Since we do not require any back seeks between nodes in the current set, the only other possible source of back seeks is from the recursive calls. The case when we do a linked-node recursive call is then the same as in the naïve algorithm; these do not cost any back seeks since we remain in the same level and can do all the recursive calls in order of disk layout.

The other case is when we do a recursive call on the union of a few nodes' children. In this case, we access the level after the current set's level. If we perform this recursive call after all linked-node recursive calls, we will never return to the current set's level, and will therefore not need any back seeks. Thus, each recursion, and therefore the modified algorithm as a whole, does not require any back seeks.

3.4 Results

We implemented HODOR in Python, and tested it with random data sets. We tracked forward seeks and back seeks during testing to see whether our data structure performed as expected in practice. After several iterations of the range query algorithm, we were able to achieve range queries without any back seeks, verifying our analysis above.

It would be interesting to analyze the performance of HODOR against that of range queries in traditional databases. However, we did not have a chance to properly optimize for speed or memory usage in our implementation. In addition, Python is inherently less suited to low-level tasks like serialization and disk I/O than languages like C. Working around, or controlling for, Python's limitations in order to generate valid comparison data was outside the scope of this project.

The code for HODOR is available to download from Github at `https://github.com/stephanie-wang/hodor`.

4 Conclusion

In our work on HODOR, we introduce a performance model to reflect the current state of hard disk technology. Our analysis model assumes that forward seeks are significantly faster than back seeks. Under this model, it is therefore critical to consider the number of back seeks required during on-disk data structure operations.

With a naïve recursive range query implementation, it is possible to achieve $O(\log_B^d n)$ back seeks by using a similarly recursive disk layout. With this disk layout, it is possible to perform queries without back seeks by adjusting the range query implementation to better fit the levels structure in the disk layout. Therefore, our work in HODOR shows that we can achieve better performance with an operation-aware serialization procedure and serialization-aware operations.

Orthogonal range-trees can in theory achieve significant speedup over traditional databases, making it desirable to implement them efficiently on-disk. They are just one of many data structures that are useful for storing datasets too large for memory. We believe that the back seek model used to analyze HODOR's performance would be effective for analysis of other such on-disk data structure implementations. Thus, we are hopeful that the results from HODOR can be used as a basis for optimizing future implementations of on-disk data structures.

References

[1] Demaine, Erik. “Lecture 3,” February 23, 2012. MIT: http://courses.csail.mit.edu/6.851/spring14/scribe/lec3.pdf, n.d. PDF.

[2] Kaler, Tim, and Oscar Moll. “Spatial Data Structures - Performance Comparision.” N.p.: http://tfk.mit.edu/pdf/2drangesearch_datastructures.pdf, 25 May 2012. PDF.

[3] Python linecache module documentation. https://docs.python.org/2/library/linecache.html

[4] Demaine, Erik. “Lecture 9,” March 15, 2012. MIT: http://courses.csail.mit.edu/6.851/spring14/scribe/lec9.pdf, n.d. PDF.

[5] Cherones, Tom. “The Pony Remark.” *Seinfeld.* NBC. 30 January 1991. Television.