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 14, 2014

1 Introduction

Many web applications today require data queries in two or more dimensions. For instance, a point of interest on a map has latitude and longitude coordinates, and may be indexed by other comparable values. These include start and end times for an event, price ranges for a business, or average user rating for a service. Databases typically support efficient range querying on one primary key at a time. Multi-dimensional range queries, however, are expensive and require several linear passes of the dataset in most implementations. On the other hand, orthogonal range trees with fractional cascading can improve range queries in d dimensions to $O(\log^{d-1} n + k)$ time, where k is the size of the result set [1].

In-memory orthogonal range tree implementations have been able to achieve significant query time speedup [2]. These can be used to extend a database for smaller datasets. Most datasets, however, are too large to fit in memory, and therefore require an on-disk implementation. To this end, we present HODOR, an on-disk Python implementation of range-trees.

The main obstacle in an on-disk implementation of orthogonal range trees is the added overhead in disk access time. Therefore, in this paper, we explore different methods of optimizing disk I/O accesses. First, we present optimizations on the range tree structure itself. Second, we propose methods of optimizing individual node serialization. Third, we explore optimizing the serialization of the entire tree. Finally, we present results from benchmarking these optimizations against a Python database [3].

2 Data Structure

Ordinarily, multidimensional range queries are expensive, with most methods taking $O(n)$ time. With large databases, a runtime of $O(n)$ is highly costly. To allow for polylog time range queries, we implemented an orthogonal range tree, generalized to d dimensions. We used a B+ tree for our implementation, as B+ trees are more efficient given the architecture of our machine. In this section, we discuss these data structures and the resulting speedup.

2.1 B+ Tree-Based Orthogonal Range Tree

B trees are commonly used for databases, they optimize for the number of disk accesses made for queries. B+ trees are extensions to B trees in which all data is stored in the leaves, and each leaf keeps a pointer to its successor. B+ trees maintain a search time of $O(\log_B n)$. We extend this B+ tree structure to form the orthogonal range tree.

An orthogonal range tree is an extension to a B+ tree in which each node in the tree points to a tree containing the contents of the node's subtree sorted in the next dimension. Dimensions are ordered hierarchically, with the "top level" dimension corresponding to the first-level tree, the second dimension's information held in subtrees of the nodes in that tree, and so on. B trees support queries in time $O(\log_B^d n + k)$, where d is the number of dimensions of the data. Because we optimize for query time, orthogonal range trees do add a factor of overhead for additional data storage; this space overhead factor is $O(\log_B^{d-1} n)$.

2.2 Using the data structure

The tree is constructed in preprocessing, which takes $O(n \cdot \log_B^{d-1} n)$ time. The user can specify any d dimensions for the dataset to be indexed on. A dimension can be any continuous, comparable data type which can be mapped to a number. Searches are performed on ranges for any subset of the d dimensions, by passing the range tree a set of dimension: (range start, range end) pairs. Open-ended range queries are not explicitly supported, but can be performed in practice by setting one of the range bounds to $\pm\infty$.

3 Design

The main goal of HODOR is to allow efficient range queries of very large datasets with an arbitrary number of dimensions. We designed the structure to perform on machines with a limited amount of very fast memory, and a large amount of memory on a comparatively slow hard disk. As such, we wanted 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 can 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 data only. In addition, although there have been results for orthogonal range trees using fractional cascading which achieve $O(\log^{d-2} n)$ query time, we were most concerned with implementing a working, testable data structure within the time we had for the project, so we did not attempt to include those techniques in HODOR.

3.1 Optimization Methods

3.2 Range B-Trees

The standard data structures used for implementing many table-style databases are the B Tree, and its cousin, 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. 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 be tweaked to optimize for cache sizes on different machines. A B-Plus tree is an extension of the B tree in which all data is stored at the lowest level of the tree (on the "leaves") and each leaf has a pointer to its immediate successor leaf.

Orthogonal range trees were a structure covered in 6.851 which allow $O(\log^d n + k)$ range queries on d dimensions, where k is the number of data items returned by the For HODOR, we extend the B-plus tree structure with orthogonal, recursive linked trees to create a B-plus Orthogonal Range Trees (BORT). Each node n in a BORT has a dimension d, a set of B child pointers sorted in 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 into memory, it brings with it all B child pointers and one link pointer. Therefore, a search on n data points indexed in d dimensions touches $O(log_B^d n)$ nodes, which requires loading $O(log_B^d n)$ chunks from disk. However, as we will describe below, not all disk accesses are equal, and we can achieve better in-practice running times by laying out serialized nodes intelligently on disk.

3.3 Node serialization

In order to store range trees on disk, we require an efficient method to serialize and deserialize the data structure. In Hodor, we represent a Python `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 datapoints, `Serializer` is initialized in write mode to build the tree file. `Serializer` in write mode exposes a `dumps(node)` method that takes in a node instance, serializes the node, and appends it to the tree file. This method also assigns the node instance a pointer into the tree file, which is the number of nodes appended previously to it.

Once the tree is fully built, we call `Serializer.flush()`, which flushes the tree file to disk. From this point onwards, `Serializer` can be used to read 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 load the child of a node instance, we simply store the child's tree file pointer as an attribute of the parent. `loads` is implemented by seeking 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 using a delimiter, such as a newline character, between nodes. This is convenient because it allows us to support arbitrarily sized datapoint values. In addition, Python is able to perform specific line reads quickly, by using either an iterator on the file or a linecache [CITATION, f.lines() iterator, linecache].

The second method we use is to pack nodes into fixed-length strings, which we call "blocks". Each node in a range tree stores a constant number of values, including min, max, and child pointers. This allows us to serialize each node as a block, as long as we know the maximum sizes of these values. The block method makes seeking efficient because it allows us to calculate exact offsets between `Serializer`'s current byte position in the file and a given node's pointer. However, this method also limits the amount and type of data that can be stored at each node.

3.4 Tree serialization

4 Results

4.1 Optimality of Serialization

We will now examine the complexity of a search on a fully formed BORT, in terms of disk reads. The algorithm for searching in a node is as follows, in pseudocode. Here, `ranges` is a dictionary of the ranges to be sorted, indexed by the name of each range's dimension.

```
def range_query(ranges):
    start, end = ranges[self.dimension]
```

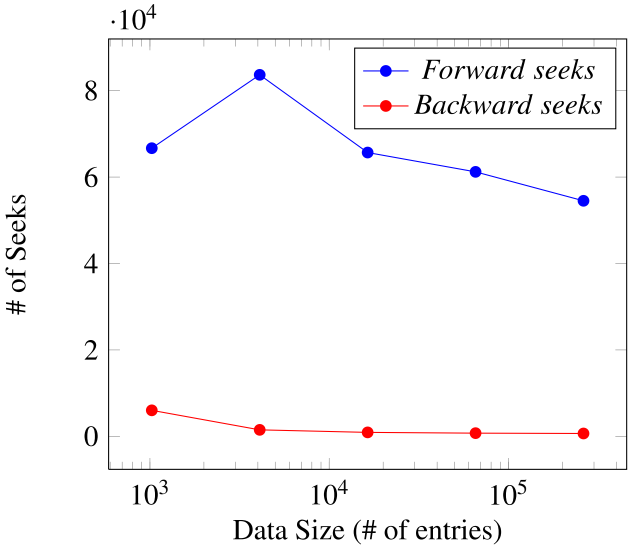

Figure 1: The number of forwards and backwards seeks involved with range queries varies with the size of the data stored in the orthogonal range tree. While backwards seeks are far more costly (10x) than forward seeks, they are far less in number, and decrease with larger data size.

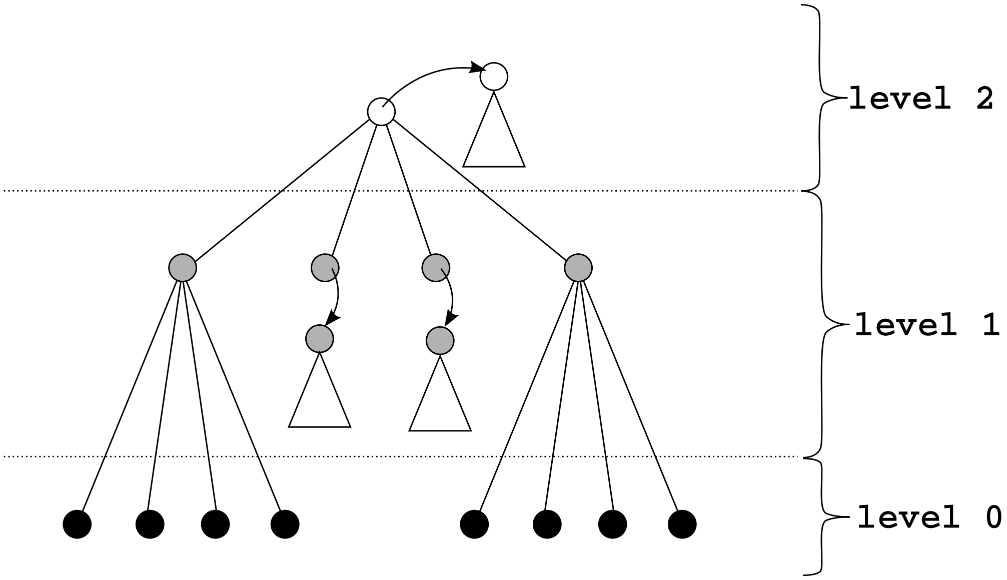

Figure 2: Awesome Image

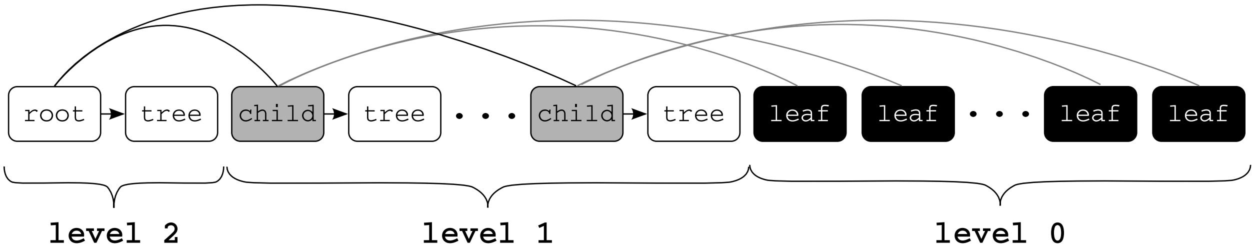

Figure 3: Awesome Image

5 Results

5.1 Optimality of Serialization

We chose to implement HODOR in Python–namely, in conjunction with the Buzhug Python database. Our initial decision allowed for the rapid implementation of the on-disk orthogonal range tree; however, the performance of our database was hindered by our choice of language. As a high-level scripting language, Python does not allow for low-level memory management of data, in contrast to a language such as C. Additionally, inherent aspects of Python such as dynamic typing and a lack of pointer use result in additional overhead. We hypothesize that an implementation of the orthogonal range tree in a low-level language such as C or C++ that allows memory management would allow for faster tree construction and query times.

A low level language would allow us to specify the exact block size used by the B+ tree on which the orthogonal range tree is built. This block size could be attenuated to the cache block size on the particular machine on which the orthogonal range tree is stored. As such, when a new block is moved into the cache, no space is wasted. Given that queries on the orthogonal range tree move mostly in the forward direction, the proportion of cache hits would increase. Furthermore, nodes could have the same size as the block size optimal for the machine. When the orthogonal range tree is in serialized format, exactly one entire node will be read into the cache at a time allowing for more efficient traversal of the tree structure.

5.2 BORTs

5.3 Optimality of Serialization

5.4 Application to Buzhug Database

An initial goal of our Pythonic implementation of the orthogonal range tree was to integrate the structure seamlessly with an existing Python database–namely, Buzhug. Buzhug is lightweight and stores its data on-disk in in a hierarchical structure. Initial testing of this integration on generated data sets showed that orthogonal range trees are impractical on small data sets. The overhead of generating the tree outweighs any possible performance benefits; ordinary queries are also frequently slower on orthogonal range trees built over small data sets than on the original list of data. However, HODOR is asymptotically less costly than a simple linear search; with larger data sets, such as those found in large data centers, HODOR would likely function comparatively more quickly. Integration of HODOR into large databases–feasible because HODOR stores the orthogonal range tree on disk–would thus be a highly practical implementation scenario.

5.5 Total Time

5.6 Disk Reads

6 Conclusion

7 References

1. Demaine, Erik. Lecture 3, February 23, 2012. N.p.:
   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, n.d. PDF.

3. ”Buzhug, a Pure-Python Database Engine.” SourceForge. N.p., n.d. Web. 14 May 2014. <http://buzhug.sourceforge.net/>.