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 [CITATION].

In-memory orthogonal range tree implementations have been able to achieve significant query time speedup [CITATION, Tim's paper]. 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 [BUZHUG CITATION].

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 Introduction

Goals: I/O Optimization

4 Implementation

In python serializer

5 Optimization methods

5.1 B trees

5.2 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.

5.3 Tree serialization

6 Results

We found and formatted two major data sets of two different sizes. The first data set corresponds to New York City parking garages, and the second data set contains entries for every documented instance of crime in the city of Chicago since the year 2001. The second data set is two orders of magnitude larger than the first data set. Both data sets contain multiple numeric fields apart from location on which range queries may

be performed. We analyzed the performance of the altered Buzhug database by performing a wide variety of range queries and examining the performance gains.

7 Analysis

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.

7.1 Total Time

7.2 Disk Reads

8 Conclusion

9 References

item