Building a Balanced k-d Tree with MapReduce

Russell A. Brown

ABSTRACT

The original description of the k-d tree recognized that rebalancing techniques, such as are used to build an AVL tree or a red-black tree, are not applicable to a k-d tree. Hence, in order to build a balanced k-d tree, it is necessary to obtain all of the data prior to building the tree since it is necessary to store the tree. A new algorithm builds a balanced k-d tree by presorting the data in each of k dimensions to build the tree in $O(n \log n)$ time. This new algorithm is amenable to execution via MapReduce and permits building and searching a k-d tree that is represented as a distributed graph.

1. INTRODUCTION

Bentley introduced the k-d tree as a binary tree that stores k-dimensional data [3]. Like a standard binary tree, the k-d tree subdivides data at each level of the tree. Unlike a standard binary tree that uses only one key for all levels of the tree, the k-d tree uses k keys and cycles through these keys for successive levels of the tree. For example, to build a k-d tree from two-dimensional points that comprise $x, y$ coordinates, the keys would be cycled as $x, y, x, y, x, y...$ for alternate levels of the k-d tree.

Due to the use of different keys at successive levels of the tree, it is not possible to employ rebalancing techniques, such as are used to build an AVL tree [1] or a red-black tree [2, 9], when building a k-d tree. Hence, the typical approach to building a balanced k-d tree finds the median of the data for each recursive subdivision of those data. Bentley showed that if the median of $n$ elements were found in $O(n)$ time, it would be possible to build a depth-balanced k-d tree in $O(n \log n)$ time. However, algorithms that find the median in guaranteed $O(n)$ time are somewhat complicated [3, 6]. An alternative approach to building a balanced k-d tree pre-sorts the data in each of $k$ dimensions prior to building the tree using merge sort [8, 13, 14] or heap sort [15]. The order of the $k$ presorts is then maintained when building a balanced tree and thereby permits a worst-case complexity of $O(kn \log n)$ for building the tree [5].

The remainder of this article will use as an example a k-d tree that sorts rectangular bounding boxes [16]. This k-d tree permits a directed search of the tree by a query bounding box in order to determine which of the bounding boxes from the tree the query bounding box intersects. The principles of building and searching such a k-d tree will be discussed initially for a tree that resides in memory, then extended to a k-d tree that is represented as a graph whose elements are distributed across multiple MapReduce compute nodes [15].

2. BUILDING A MEMORY-RESIDENT TREE

The k-d tree will be used to search for bounding boxes that intersect a query bounding box. Hence, each node of the tree must store the $(x_{min}, y_{min}, x_{max}, y_{max})$ coordinates of a bounding box to facilitate an intersection test of that bounding box against the query bounding box, as shown for the root node in Figure 1. In addition, each node of the tree must permit a determination of which of the node’s two subtrees the query bounding box will search recursively. This determination is facilitated by storing, for each subtree, the $(x_{min}, y_{min}, x_{max}, y_{max})$ coordinates of a rectangular bounding region that just encloses all of the bounding boxes in the subtree. If the query bounding box intersects a bounding region, the query bounding box must search the associated subtree.

Given the above requirements for each node of the k-d tree, the tree may be constructed recursively as summarized below. A detailed description of the k-d tree building algorithm has been published previously [5].

First, the bounding boxes are presorted independently in $x_{min}$ and $y_{min}$ prior to building the k-d tree to create two pre-sorted arrays of bounding boxes. In order that each bounding box may have a unique sorting key in each of $x_{min}$ and $y_{min}$, a unique name $ni$ is assigned to each bounding box and used to form the super keys $x_{min}ni$ and $y_{min}ni$ for use in the presorts, where the colon represents the concatenation operator.

Next, the presorted arrays are partitioned in $x_{min}$ at the root of the tree as follows. The bounding box $bbni$ at the median element $n_{ni}$ of the array that was presorted by the $x_{min}ni$ super key is stored at the root of the tree. This median element trivially partitions the $x_{min}ni$-sorted array as shown in Figure 2. The elements of the $x_{min}ni$-sorted array at all addresses below the median address form a “less than” $x_{min}ni$-sorted subarray. The elements of the $x_{min}ni$-sorted array at all addresses above the median address form a “greater than” $x_{min}ni$-sorted subarray.

This median element also partitions the $y_{min}ni$-sorted array, albeit non-trivially, via a “sweep and partition” algo-
Figure 1: The root node $n_1$ of the $k$-d tree and its two children $n_<$ and $n_>$ are depicted by circles. The root node stores the bounding box $bb_r$ and the bounding regions $br_<$ and $br_>$ that are depicted by dashed rectangles whose sizes and locations are not represented accurately in this figure.

Figure 2: The $x_{\text{min}}:n_1$-sorted array ($x_{\text{min}}$) is partitioned trivially at its median element ($n_m$) to obtain the “less than” $x_{\text{min}}:n_1$-sorted subarray ($<$) and the “greater than” $x_{\text{min}}:n_1$-sorted subarray ($>$). The bounding box $bb_m$ of the median element is stored at the root of the tree that is depicted by the circle.

For a non-leaf node that has one or two children, the bounding region is the region that just encloses the bounding box of the node and the bounding regions of the children. This bounding region is calculated as the minimum of the $x_{\text{min}}$, the minimum of the $x_{\text{max}}$ and the maximum of the $y_{\text{max}}$ of the node and of the node’s children.

3. BUILDING A DISTRIBUTED TREE

A distributed $k$-d tree may be constructed with MapReduce using Spark. The construction proceeds recursively in a similar manner to the construction of a memory-resident $k$-d tree; however, arrays are not used to store the presorted bounding boxes. Instead, Spark uses a resilient distributed dataset (RDD), which represents a collection of items that are distributed across multiple compute nodes and hence may be processed in parallel. Spark provides many methods for processing RDDs in parallel. Of particular interest are the sortByKey method that may be used to “sweep and partition” an RDD as shown in Figure 6.

Unfortunately, Spark provides neither a method that obtains the median element of a sorted RDD nor a method that splits a sorted RDD about its median element to create “less than” and “greater than” RDDs. However, a proposed extension to Spark that implements the drop method, which deletes a specified number of elements from an RDD, provides insight into how the necessary splitting method may be implemented. This splitting method, named splitAt, has been implemented to iterate over the partitions of an RDD in parallel in order to count the elements in each partition and thereby obtain the median element of the RDD. This median element is then used to split the RDD to create “less than” and “greater than” RDDs, as shown in Figure 6.

The Spark sortByKey, filter and splitAt methods permit
the construction of a distributed k-d tree with MapReduce. In contrast to a memory-resident tree, a distributed tree does not use pointers to link each k-d node to its children. Instead, the tree is represented as a graph whose nodes are distributed across multiple compute nodes as an RDD [15]. For example, the root node of the k-d tree is represented as the following element of an RDD that contains all the nodes of the k-d tree

\[(n_r, (bb_r, n_<, br_<, n_>, br_>)\]

where \(n\) represents the unique name of the root node of the tree, \(bb\) represents the bounding box that is stored at the root node of the tree, \(n_<\) represents the unique name of the “less than” child of the root node, \(br_<\) represents the bounding region that encloses the “less than” subtree, \(n_>\) represents the unique name of the “greater than” child of the root node, and \(br_>\) represents the bounding region that encloses the “greater than” subtree. This RDD will be named the **tree RDD** (in the following discussion, the name of a specific RDD will be designated in bold face text).

The parentheses in the above representation of the root node of the k-d tree enclose **tuples** [17] and in particular, the outer set of parentheses encloses a two-element tuple or pair whose first element is \(n\) and whose second element is the tuple \((bb, n_<, br_<, n_>, br_>)\). An RDD that comprises pairs is a **pair RDD** and has the special property that the two elements of each pair function as a (key, value) pair. In the case of the above-described root node of the k-d tree, the unique name \(n_r\) is the key and the tuple \((bb_r, n_<, br_<, n_>, br_>)\) is the value. The (key, value) properties of a pair RDD permit Spark to retrieve a particular node of the k-d tree from the **tree RDD** using the unique name of that node, analogous to the manner in which a pointer to a particular node of a memory-resident k-d tree permits the retrieval of that node from the tree.

4. **SEARCHING A DISTRIBUTED TREE**

Searching a distributed k-d tree is not performed in a recursive, depth-first manner such as is used to search a memory-resident tree. Instead, an iterative, breadth-first search is conducted in parallel by multiple bounding boxes via MapReduce, similar to the manner in which MapReduce executes Dijkstra’s algorithm [15]. An overview of the search algorithm is presented below.

A search of the k-d tree assumes the existence of a **search RDD** that contains the elements \((n_1, bb_1)\), where \(bb_1\) represents the bounding boxes that will search the tree for intersections and \(n_1\) represents the unique names of the bounding boxes. In preparation for the first iteration of the search algorithm, the **search RDD** is processed using the map method [11] to create a **query RDD** whose elements are \((n_i, (n_i, bb_i))\). For each element of this **query RDD**, the key is \(n_i\) and represents the unique name of the root node of the k-d tree. This key specifies that each element of the **search RDD** will visit the root of the k-d tree. The **query RDD** is joined to the **tree RDD** using the join method [12] that performs an **inner join** and returns a **visit RDD** that contains the elements

\[(n_i, ((n_i, bb_i), (bb_i, n_<, br_<, n_>, br_>)))\]

The **visit RDD** represents the fact that each element of the **search RDD** visits the root of the k-d tree.

The **visit RDD** is subsequently processed using the flatMapValues method [12] in order to check for intersection between each \(bb_i\) and \(bb_i\). Each intersection creates an element \((n_j, n_j)\) in an **intersection RDD** that represents the fact that the bounding boxes \(bb_i\) intersect the bounding box \(bb_i\) where \(bb_i\) is a subset of \(bb_i\).

The **visit RDD** is processed once again using the flatMapValues method in order to check for intersection between
Figure 5: The $y_{\min}$-sorted RDD ($y_{\min}$) is filtered by the $x_{\min}$ of the median element $n_m$ (see Figure [6]) of the $x_{\min}$-$n_\ell$-sorted RDD to obtain the “less than” $y_{\min}$-$n_\ell$-sorted RDD ($<$) and the “greater than” $y_{\min}$-$n_\ell$-sorted RDD ($>$).

Each $bb_k$ and $br_{<}$ as well as to check for intersection between each $bb_k$ and $br_{>}$. Each intersection between $bb_k$ and $br_{<}$ creates an element $(n_{<}, (n_k, bb_k))$ in the next query RDD. Similarly, each intersection between $bb_k$ and $br_{>}$ creates an element $(n_{>}, (n_k, bb_k))$ in the next query RDD. The bounding boxes $bb_k$ and $bb_0$ are the subsets of $bb_k$ that will visit the “less than” and “greater than” subtrees, respectively, during the next iteration of the search algorithm. The iterative search algorithm terminates when the next visit RDD is empty, as will be the case when no intersections between $bb_k$ and $br_{<}$ or $br_{>}$ are detected.

The intersection RDD from each iteration of the search algorithm is accumulated to a cumulative intersection RDD using the union method \[^{12}\]. Following the final iteration of the algorithm, the $(n_{<}, n_{>})$ pairs of the cumulative intersection RDD are reorganized using the groupByKey method \[^{12}\] to create the pairs $(n_{<}, [n_u...n_w])$, where for each pair, $[n_u...n_w]$ specifies the list of bounding boxes $[bb_u...bb_w]$ that are a subset of $bb_k$ and that intersect $bb_k$.

5. PERFORMANCE

The $k$-d tree building and search algorithms have been executed on only a local Spark node, so their performance and scalability on a cluster of Spark compute nodes has not yet been measured.

6. CONCLUSION

This article presents new $k$-d tree building and search algorithms that are a generalization of algorithms that were published previously \[^{5}\]. In contrast to those prior algorithms that build and search a memory-resident $k$-d tree, the new algorithms build and search a $k$-d tree that is represented as a distributed graph.

7. REFERENCES


