Building a Balanced k-d Tree with MapReduce

Russell A. Brown

ABSTRACT

The original description of the k-d tree recognized that re-balancing 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 then to build the tree via recursive subdivision of the data. One algorithm for building a balanced k-d tree finds the median of the data for each recursive subdivision of the data and builds the tree in $O\left(n \log n\right)$ time. A new algorithm builds a balanced k-d tree by presorting the data in each of k dimensions prior to building the tree, then preserves the order of the k presorts during recursive subdivision of the data and builds the tree in $O\left(kn \log n\right)$ 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 [4]. Like a standard binary tree, the k-d tree subdivides data at each recursive 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 [3, 10], 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 $O\left(n\right)$ time, it would be possible to build a depth-balanced k-d tree in $O\left(n \log n\right)$ time. However, algorithms that find the median in guaranteed $O\left(n\right)$ time are somewhat complicated [5, 7]. An alternative approach to building a balanced k-d tree presorts the data in each of k dimensions prior to building the tree using merge sort [9, 16, 17] or heap sort [22]. The order of the k presorts is then maintained when building a balanced tree and thereby permits a worst-case complexity of $O\left(kn \log n\right)$ for building the tree [6].

The remainder of this article will use as an example a k-d tree that sorts rectangular bounding boxes [19]. 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 in 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 [18].

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.
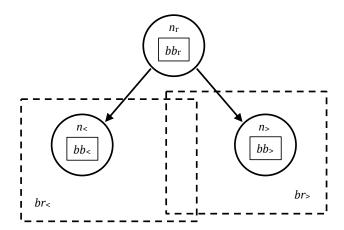

Figure 1: The root node n_{r} of the k-d tree and its two children $n_<$ and $n_>$ are depicted by circles. The root node stores the bounding box bb_{r} and the bounding regions $br_<$ and $br_>$ that are depicted by dashed rectangles whose sizes and locations are not represented accurately in this figure.

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 [6].

First, the bounding boxes are presorted independently in x_{min} and y_{min} prior to building the k-d tree to create two presorted 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 n_i is assigned to each bounding box and used to form the super keys $x_{min}:n_i$ and $y_{min}:n_i$ for use in the presorts, where the colon represents the catenation operator.

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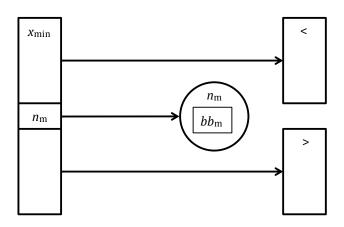

Figure 2: The $x_{min}:n_i$-sorted array (x_{min}) is partitioned trivially at its median element (n_m) to obtain the "less than" $x_{min}:n_i$-sorted subarray ($<$) and the "greater than" $x_{min}:n_i$-sorted subarray ($>$). The bounding box bb_m of the median element is stored at the root of the tree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

This median element also partitions the $y_{min}:n_i$-sorted array, albeit non-trivially, via a "sweep and partition" algorithm as shown in Figure 3 [21, 6]. In order to partition the $y_{min}:n_i$-sorted array, the array is swept from lowest to highest address, $i.e.$, swept in order from lowest to highest $y_{min}:n_i$ super key, and the elements of this array are partitioned into a "less than" $y_{min}:n_i$-sorted subarray and a "greater than" $y_{min}:n_i$-sorted subarray as follows. Each element's $x_{min}:n_i$ super key is compared to the $x_{min}:n_m$ super key of the median of the $x_{min}:n_i$-sorted array whose bounding box is stored at the root of the tree. If the element's super key is less than the root's super key, the element is partitioned into the "less than" $y_{min}:n_i$-sorted array. If the element's super key is greater than the root's super key, the element is partitioned into the "greater than" $y_{min}:n_i$-sorted

array. If the element's super key equals the root's super key, the element is ignored. In this manner, the $y_{min}:n_i$-sorted array is partitioned by the $x_{min}:n_m$ super key of the root of the tree to create "less than" and "greater than" $y_{min}:n_i$-sorted subarrays that each preserve the relative $y_{min}:n_i$-sorted order from the initial presort.

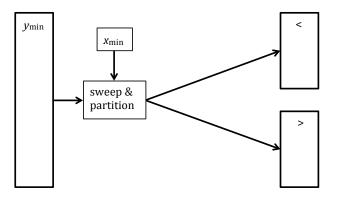**Figure 3: The $y_{min}:n_i$-sorted array (y_{min}) is swept from lowest address to highest address and partitioned by the x_{min} of the median element n_m (see Figure 2) of the $x_{min}:n_i$-sorted array to obtain the "less than" $y_{min}:n_i$-sorted subarray ($<$) and the "greater than" $y_{min}:n_i$-sorted subarray ($>$).**

The "less than" $x_{min}:n_i$-sorted and $y_{min}:n_i$-sorted subarrays are used to build recursively the "less than" subtree at the next level of the nascent k-d tree. The "greater than" $x_{min}:n_i$-sorted and $y_{min}:n_i$-sorted subarrays are used to build recursively the "greater than" subtree at the next level of the nascent k-d tree. At the root of each of these two subtrees, the $y_{min}:n_i$-sorted subarray is partitioned trivially. The $x_{min}:n_i$-sorted subarray is partitioned non-trivially via the "sweep and partition" algorithm that compares $y_{min}:n_i$ super keys. The alternation of trivial and non-trivial partitioning in x_{min} and y_{min} continues at each level of the k-d tree until the subarrays are exhausted at the leaf nodes of the tree.

As the recursion unwinds, pointers to the node's children are returned to each node of the nascent k-d tree. In addition, the bounding region for each node is calculated as depicted in Figure 4. For a leaf node, the dimensions of the bounding region are identical to those of the bounding box. 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_{min}, the minimum of the y_{min}, the maximum of the x_{max} and the maximum of the y_{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 elements
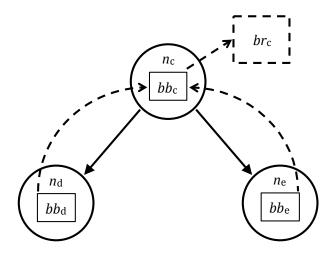
Figure 4: The bounding boxes bb_d and bb_e of the respective leaf nodes n_d and n_e are identical to the bounding regions of these nodes. The bounding regions of the leaf nodes are combined with the bounding box bb_c of their parent node n_c as indicated by the curved, dashed arrows to calculate the bounding region br_c of the parent node that is depicted by the dashed rectangle. Neither the size nor the location of br_c is represented accurately in this figure.

Figure 5: The y_{min}:n_i-sorted RDD (y_{min}) is filtered by the x_{min} of the median element n_m (see Figure 6) of the x_{min}:n_i-sorted RDD to obtain the "less than" y_{min}:n_i-sorted RDD ($<$) and the "greater than" y_{min}:n_i-sorted RDD ($>$).

that are distributed across multiple compute nodes of a cluster and hence may be processed in parallel [12]. Spark provides many methods for processing RDDs in parallel. Of particular interest are the sortByKey method [15] that may be used to presort an RDD prior to building the k-d tree and the filter method [13] that may be used to "sweep and partition" an RDD as shown in Figure 5.

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 [8]. 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 [23] and thereby determine in which partition the median element lies and then obtain the median element of the RDD. This median element is 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 [18]. 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
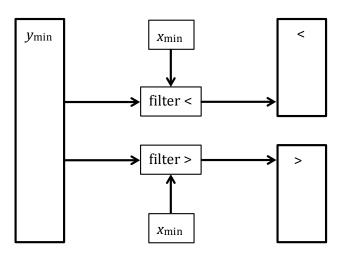
$$(n_r, (bb_r, n_<, br_<, n_>, br_>))$$

where n_r represents the unique name of the root node of the tree, bb_r 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* [20] and in particular, the outer set of parentheses encloses a two-element tuple or pair whose first element is n_r and whose second element is the tuple $(bb_r, n_<, br_<, n_>, br_>)$. 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 is not performed in a recursive, depth-first manner such as is used to search a memory-resident tree. Instead, an iterative, bread-first search is conducted in parallel by multiple bounding boxes via MapReduce, similar to the manner in which MapReduce executes
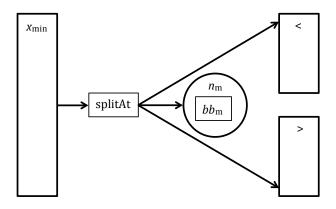
Figure 6: The splitAt method finds the median element (n_{m}) of the $x_{\min}$:n_{i}-sorted RDD ($x_{\min}$) then partitions the RDD at the median element to obtain the "less than" $x_{\min}$:n_{i}-sorted RDD ($<$) and the "greater than" $x_{\min}$:n_{i}-sorted RDD ($>$). The bounding box bb_{m} of the median element is stored at the root of the tree that is depicted by the circle.

Dijkstra's algorithm [18]. 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 $(n_{\mathrm{i}}, bb_{\mathrm{i}})$, where bb_{i} represents the bounding boxes that will search the tree for intersections and n_{i} 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 Spark map method [13] to create a **query RDD** whose elements are $(n_{\mathrm{r}}, (n_{\mathrm{i}}, bb_{\mathrm{i}}))$. For each element of this **query RDD**, the key is n_{r} 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 Spark join method [15] that performs an *inner join* and returns a **visit RDD** that contains the elements

$$(n_{\mathrm{r}}, ((n_{\mathrm{i}}, bb_{\mathrm{i}}), (bb_{\mathrm{r}}, n_{<}, br_{<}, n_{>}, br_{>})))$$

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 Spark flatMapValues method [15] in order to check for intersection between each bb_{i} and bb_{r}. Each intersection creates an element $(n_{\mathrm{j}}, n_{\mathrm{r}})$ in an **intersection RDD** that represents the fact that the bounding boxes bb_{j} intersect the bounding box bb_{r}, where bb_{j} is a subset of bb_{i}.

The **visit RDD** is processed once again using the flatMapValues method in order to check for intersection between each bb_{i} and $br_{<}$ as well as to check for intersection between each bb_{i} and $br_{>}$. Each intersection between bb_{i} and $br_{<}$ creates an element $(n_{<}, (n_{\mathrm{k}}, bb_{\mathrm{k}}))$ in the **next query RDD**. Similarly, each intersection between bb_{i} and $br_{>}$ creates an element $(n_{>}, (n_{\mathrm{l}}, bb_{\mathrm{l}}))$ in the **next query RDD**. The bounding boxes bb_{k} and bb_{l} are the subsets of bb_{i} 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_{i} 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 Spark union method [15]. Following the final iteration of the algorithm, the $(n_{\mathrm{s}}, n_{\mathrm{t}})$ pairs of the **cumulative intersection RDD** are reorganized using the Spark groupByKey method [15] to create the pairs $(n_{\mathrm{s}}, [n_{\mathrm{u}}...n_{\mathrm{w}}])$, where for each pair, $[n_{\mathrm{u}}...n_{\mathrm{w}}]$ specifies the list of bounding boxes $[bb_{\mathrm{u}}...bb_{\mathrm{w}}]$ that are a subset of bb_{t} and that intersect bb_{s}.

A deeper understanding of the distributed k-d tree search algorithm may be obtained by studying the Scala source code and embedded comments that are included with this article.

5. PERFORMANCE

Under the Spark execution model, a *master* distributes data and computation across multiple compute nodes known as *workers*. The master and workers may comprise individual compute nodes of a cluster or they may comprise separate cores of a CPU. For the experimental results reported below, the master and workers comprise separate cores of a 2.3 GHz Intel quad-core i7 CPU that supports concurrent execution of two threads per core.

A test data set comprises 16 rectangles contained within a square; nine of the 16 rectangles intersect one another. The square is replicated and translated in the (x, y) plane to produce a specified number of adjacent but non-overlapping squares, each of which contains 16 rectangles. The complete set of rectangles from all of the squares is used to construct a distributed k-d tree, then the complete set of squares is used to search the tree for intersecting rectangles. Because the squares themselves do not overlap, intersections between rectangles occur only within each square but do not extend to adjacent squares, so the expected number of intersecting rectangles is 9 times the number of squares. This expected number is used to verify the correctness of the search result.

Figure 7 shows the execution time in seconds that is required by four workers to build a balanced k-d tree, plotted versus $n \log_2(n)$ for $2^4 \le n \le 2^{12}$ rectangles. The dashed line of Figure 7 shows the least-squares fit of the time t to the function $t = mn \log_2(n) + t_{\mathrm{S}}$ where m is the slope of the line and the intercept t_{S} is a serial component of the execution time that is independent of n. The correlation coefficient $r = 0.9988$ indicates an adequate least-squares fit; hence, the execution time is proportional to $n \log_2(n)$.

Figure 8 shows the execution time in seconds that is required by four workers to search the balanced k-d tree for intersections between rectangles, plotted versus $n \log_2(n)$ for $2^4 \le n \le 2^{12}$ rectangles. The dashed line of Figure 8 shows the least-squares fit of the time t to the function $t = mn \log_2(n) + t_{\mathrm{S}}$ where m is the slope of the line and the intercept t_{S} is the serial component of the execution time. The correlation coefficient $r = 0.9702$ indicates a less-than-adequate least-squares fit; hence, the execution time may not be strictly proportional to $n \log_2(n)$.

In Figures 7 and 8, the scales of the y-axes are 120 seconds and 3 seconds, respectively. Thus, the deviation from $O(n \log n)$ computational complexity that is apparent in Figure 8 may be due to statistical variation that is present in both figures but more obvious in Figure 8 than in Figure 7.
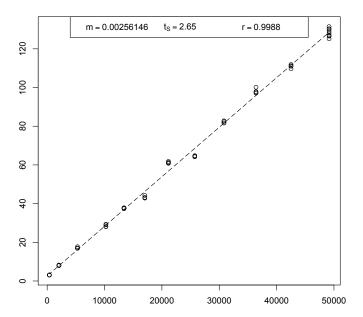
Figure 7: The *k*-d tree building time (seconds) is plotted vs. $n \log_2(n)$ for $2^4 \leq n \leq 2^{12}$ rectangles.

Figure 8: The *k*-d tree search time (seconds) is plotted vs. $n \log_2(n)$ for $2^4 \leq n \leq 2^{12}$ rectangles.

The *k*-d tree building and search algorithms were executed in parallel and their performance was measured for one to eight workers using a 2.3 GHz Intel quad-core i7 processor. Figures 9 and 10 show, respectively, the execution time in seconds that is required to build and search the *k*-d tree, plotted versus the number of workers w for $n = 2^{12}$ rectangles. The dashed curves in these figures show the least-squares fits of the execution time t to the model

$$t = t_{\mathrm{s}} + \frac{t_{\mathrm{p}}}{w} + m_{\mathrm{c}}(w - 1) \tag{1}$$

In this equation, w is the number of workers, t_{s} represents the time required to execute the serial or non-parallelizable portion of the algorithm, t_{p} represents the time required to execute the parallelizable portion of the algorithm via one worker, and $m_{\mathrm{c}}(w - 1)$ models a limitation to parallel execution that the Amdahl model [2] fails to capture [11] and that likely arises due to cache misses [6]. The correlation coefficients for the least-squares fits are 0.9968 and 0.9935 for Figures 9 and 10, respectively, and indicate adequate least-squares fits.

6. DISCUSSION

Figure 9 shows that one worker requires 200 seconds to build a distributed *k*-d tree by subdividing sorted RDDs that contain 2^{12} rectangles. In contrast, one worker requires only 122 ± 3 milliseconds to build a *k*-d tree by subdividing sorted arrays that contain 2^{12} rectangles (based on 20 measurements of the execution time).

Given this disparity in performance, a reasonable strategy for building a distributed *k*-d tree is to subdivide an RDD via workers but only to the point that each resulting subdivided RDD becomes small enough to fit into the memory of a compute node. At that point, the *k*-d tree building algorithm stops subdividing RDDs and begins subdividing arrays as follows.
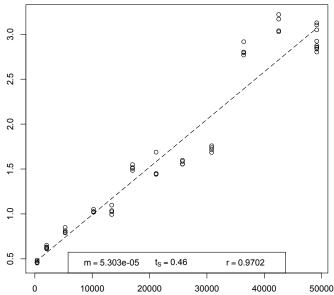
A subdivided RDD contains all of the rectangles that are required for building a subtree, so the subdivided RDD is converted to an array that is returned to the master via the Spark collect method [14]. Then the master constructs a subtree by subdividing the array. Each node of the resulting subtree is then appended to the **tree RDD**.

Potentially, multiple subdivided RDDs would be produced, each of which would be converted to an array that is small enough to fit into the memory of a compute node. Therefore, it is tempting to consider an execution model wherein the master would send an array representing the subdivided RDD to a worker via a remote procedure call (RPC). The worker would build a subtree by subdividing the array then return the nodes of the subtree to the master that would append the nodes to the **tree RDD**. Multiple workers would be available, so the master could send different arrays to different workers that could build different subtrees in parallel.

Because, for the performance measurements that are reported above, the master and workers comprise separate cores of an Intel quad-core i7 CPU, an effect similar to that of multiple RPCs may be achieved via execution of multiple threads that are provided by a Scala ExecutionContext. This approach was implemented and shows that the execution time required to build a *k*-d tree is independent of the number of threads dedicated to building subtrees by subdividing arrays. This result is consistent with the fact that building a tree from 2^{12} rectangles by subdividing arrays occurs several thousand times more rapidly than building that tree by subdividing RDDs. Hence, because an array is subdivided simultaneously to the subdivision of an RDD, subdivision of the array has completed long before subdivision of the RDD. However, it is not always the case that building a subtree by subdividing arrays occurs more rapidly than subdividing RDDs because the time required for either depends on the depth in the tree.

At any particular level of the nascent *k*-d tree, the time required to execute the Spark filter and splitAt methods in
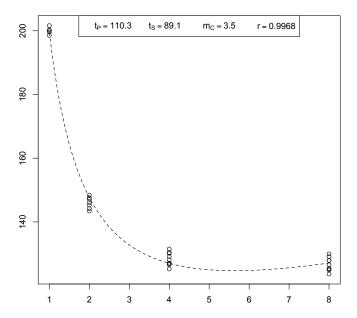
Figure 9: The *k*-d tree building time (seconds) is plotted vs. the number of workers for 2^{12} rectangles.

Figure 10: The *k*-d tree search time (seconds) is plotted vs. the number of workers for 2^{12} rectangles.

order to subdivide an RDD is directly proportional to the number of elements in the RDD, as evidenced by the fact that the time required to build the tree is proportional to $n \log_2 n$ (see Figure 7). The time required to execute these methods is inversely proportional to the number of workers because these methods necessitate no communication between workers. Hence, the time t required to subdivide the $x_{\min}$:n_i-sorted and $y_{\min}$:n_i-sorted RDDs may be modeled as

$$t = \frac{c_r n}{2^d w} \tag{2}$$

In this equation, n is the number of rectangles, d is the depth in the tree (where $d = 0$ at the root), w is the number of workers and c_r is a proportionality constant that is equal to the time required to build a k-d tree from n rectangles by subdividing RDDs, divided by $n \log_2 n$.

At any particular level of the nascent tree, the time required to build a subtree via subdivision of arrays may be modeled as

$$t = c_a 2 \frac{n}{2^{d+1}} \log_2 \left(\frac{n}{2^{d+1}} \right) \tag{3}$$

In this equation, n is the number of rectangles, d is the depth in the tree and c_a is a proportionality constant that is equal to the time required to build a k-d tree from n rectangles by subdividing arrays, divided by $n \log_2 n$.

The master can build a subtree via subdivision of arrays in less time than the time required for the workers to subdivide the $x_{\min}$:n_i-sorted and $y_{\min}$:n_i-sorted RDDs, so long as

$$c_a 2 \frac{n}{2^{d+1}} \log_2 \left(\frac{n}{2^{d+1}} \right) < \frac{c_r n}{2^d w} \tag{4}$$

This equation can be simplified to obtain

$$d > \log_2 (n) - \frac{c_r}{c_a w} - 1 \tag{5}$$

The physical interpretation of Equation 5 is that for a given number of rectangles n, increasing the ratio $c_r/c_a w$ means that for a smaller depth d in the tree, the master can
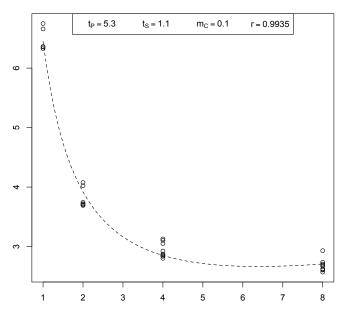
build a subtree via subdivision of arrays in less time than the time required for the workers to subdivide the $x_{\min}$:n_i-sorted and $y_{\min}$:n_i-sorted RDDs.

The k-d tree building algorithm that was discussed in Sections 2 and 3 of this article is not suitable for building a k-d tree wherein RDDs are collected to arrays that are subsequently processed by asynchronous threads or RPCs. That algorithm creates a bounding region as the recursion unwinds and returns the bounding region to the next higher level of the tree. However, when the master launches an asynchronous thread to build a subtree, that thread cannot return the bounding region of the subtree to the master because the master does not wait for the thread to finish building the subtree. Instead, the master subdivides the next pair of $x_{\min}$:n_i-sorted and $y_{\min}$:n_i-sorted RDDs while the thread builds the subtree.

For this reason, an alternate k-d tree building algorithm is implemented for building the upper portion of the nascent tree via subdivision of RDDs. This algorithm subdivides $x_{\min}$:n_i-sorted, $y_{\min}$:n_i-sorted, $x_{\max}$:n_i-sorted and $y_{\max}$:n_i-sorted RDDs. It is possible to obtain the bounding region for a node in the k-d tree from these four RDDs. Specifically, for a given level of subdivision, the bounding region is constructed from the first element of the $x_{\min}$:n_i-sorted RDD, the first element of the $y_{\min}$:n_i-sorted RDD, the last element of the $x_{\max}$:n_i-sorted RDD and the last element of the $y_{\max}$:n_i-sorted RDD.

Because it is possible to construct the bounding region for a node from these four RDDs, there is no requirement to return the bounding region of a node as the recursion unwinds. Hence, this algorithm for building a k-d tree is well-suited to building the upper portion of the nascent tree. For the lower portion of the nascent tree that is build via subdivision of arrays, the algorithm that was discussed in Sections 2 and 3 of this article suffices.

7. CONCLUSION

This article presents a new k-d tree building algorithm that builds a tree with MapReduce in time proportional to $n \log n$. The tree is represented as a distributed graph that may be searched via a new k-d tree search algorithm with MapReduce, possibly in time proportional to $n \log n$.

Source Code

The k-d tree building and search algorithms have been implemented in Scala and execute in parallel via Spark. The source code for this implementation includes the BSD 3-Clause License and is available for download.

Acknowledgements

8. REFERENCES

[1] G. Adelson-Velskii and E. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962.

[2] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies (AFIPS) Conference Proceedings*, pages 483–485, Reston, VA, 1967. AFIPS Press.

[3] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[4] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[5] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[6] R. Brown. Building a balanced k-d tree in $O\left(kn \log n\right)$ time. *Journal of Computer Graphics Techniques*, 4:50–68, 2015.

[7] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Selection in worst-case linear time. In *Introduction to Algorithms*, pages 220–222. MIT Press, Cambridge, MA, third edition, 2009.

[8] E. Erlandson. Some implications of supporting the Scala drop method for spark RDDs. 2014.

[9] H. Goldstine and J. von Neumann. Coding of some combinatorial (sorting) problems. In A. Taub, editor, *John von Neumann Collected Works: Design of Computers, Theory of Automata and Numerical Analysis*, volume 5, pages 196–214. Pergamon Press Ltd. and the Macmillan Company, New York, NY, 1963.

[10] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science.*, pages 8–21. IEEE Press, 1978.

[11] N. J. Gunther. Scalability - a quantitative approach. In *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services*, pages 41–70. Springer-Verlag, Berlin, first edition, 2007.

[12] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. Introduction to data analysis with spark. In *Learning Spark*, pages 1–8. O'Reilly Media, Sebastopol, CA, first edition, 2015.

[13] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. Programming with RDDs. In *Learning Spark*, pages 23–46. O'Reilly Media, Sebastopol, CA, first edition, 2015.

[14] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. Tuning and debugging spark. In *Learning Spark*, pages 141–160. O'Reilly Media, Sebastopol, CA, first edition, 2015.

[15] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. Working with key/value pairs. In *Learning Spark*, pages 47–70. O'Reilly Media, Sebastopol, CA, first edition, 2015.

[16] D. Knuth. Sorting by merging. In *The Art of Computer Programming: Sorting and Searching*, volume 3, page 161. Addison-Wesley, Reading, MA, 1973.

[17] D. Knuth. Summary, history, and bibliography. In *The Art of Computer Programming: Sorting and Searching*, volume 3, page 384. Addison-Wesley, Reading, MA, 1973.

[18] J. Lin and C. Dyer. Graph algorithms. In *Data-Intensive Text Processing with MapReduce*, pages 85–103. Morgan and Claypool, San Rafael, CA, first edition, 2010.

[19] W. Newman and R. Sproull. Overlap tests. In *Principles of Interactive Computer Graphics*, pages 371–372. McGraw-Hill, New York, second edition, 1979.

[20] M. Odersky, L. Spoon, and B. Venners. Tuples. In *Programming in Scala*, pages 396–398. Artima, Walnut Creek, CA, second edition, 2010.

[21] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 Institute of Electrical and Electronics Engineers (IEEE) Symposium on Interactive Ray Tracing*, pages 61–69. IEEE Press, 2006.

[22] J. Williams. Heapsort (algorithm 232). *Communications of the ACM*, 7:347–348, 1964.

[23] P. Zečević. How to get the number of elements in (a) partition? 2015.