An Investigation of Grid-enabled Tree Indexes for Spatial Query Processing

Jaewoo Shin
shin152@purdue.edu
Purdue University
West Lafayette, Indiana

Ahmed R. Mahmood
amahmoo@google.com
Google Inc.
Madison, Wisconsin

Walid G. Aref
aref@purdue.edu
Purdue University
West Lafayette, Indiana

ABSTRACT

Two-dimensional tree-based spatial indexes (e.g., the quad tree or the k-d tree) are commonly used for indexing spatial data. However, both types of indexes have limitations. Although two-dimensional trees can handle skewed data, index traversal and tree maintenance can be expensive. In contrast, a spatial grid has low update overhead, but is not suitable for skewed data. In this paper, we investigate the augmentation of a grid into tree-based indexing for spatial query processing. We introduce the Grid-Enabled Tree index (the GE-Tree, for short); a hybrid spatial index that augments a grid into two-dimensional tree indexes. In particular, we investigate the use of a grid at the leaf level of a quadtree to facilitate tree navigation and maintenance. At the expense of the extra storage, the GE-Tree achieves constant-time tree search, insert, update, and leaf node neighbor finding operations, in contrast to the log time in conventional two-dimensional trees, e.g., $O(\log S)$ in $S \times S$ space as in the case of the quadtree. Also, we illustrate how the range and k-nearest-neighbor operations can be facilitated using grid-enabled trees. Experimental results using real spatial data highlight the tradeoffs of when using grid-enabled trees pay off in contrast to regular space-partitioning trees for range and k-NN operation. Also, the GE-Tree outperforms conventional tree or grid-only indexes by up to two times for leaf node access operations, e.g., as in point-location search, neighbor-finding search operations, and the k-nearest neighbor search.

CCS CONCEPTS

• **Information systems** → **Data structures**; *Point lookups*; *Multidimensional range search.*

SIGSPATIAL '19, November 5–8, 2019, Chicago, IL, USA

ACM ISBN 978-1-4503-6909-1/19/11…$15.00
https://doi.org/10.1145/3347146.3359384

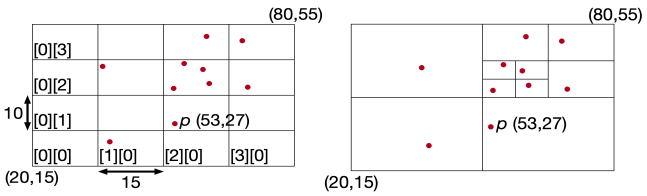
KEYWORDS

Spatial index, Query processing, Grid-based spatial index, Quadtree

ACM Reference Format:
Jaewoo Shin, Ahmed R. Mahmood, and Walid G. Aref. 2019. An Investigation of Grid-enabled Tree Indexes for Spatial Query Processing. In *27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '19), November 5–8, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3347146.3359384

1 INTRODUCTION

Many mobile applications generate large volumes of spatial data, e.g., social media applications, ride-sharing applications, and weather alert services. Spatial indexing is important to ensure low latency and fast response times for these services and applications. In this paper, we investigate the augmentation of a grid into a tree-based index for efficient spatial query processing.



(a) grid **(b) quadtree**

Figure 1: Sample objects in (a) A grid index with w=15, h=10 and $min(X,Y)$=(20,15) and (b) a quadtree with $threshold$=2.

On the one hand, using only a two-dimensional uniform grid allows for rapid access to data given location as input. However, a uniform grid does not handle skewed datasets efficiently, and does not enable efficient query processing (e.g., the range or the k-nearest-neighbor operation) due to load imbalance. On the other hand, space-partitioning trees (e.g., a quadtree or a k-d tree) are broadly used in applications (e.g., as in [1, 8, 10, 13]) because of their ability to handle skewed datasets and queries. However, a tree index causes a bottleneck due to the tree traversal time from the root to the leaf nodes. This is especially true in time-sensitive streaming applications that require low latency.

In this paper, we investigate grid-enabled trees, the GE-tree, for short, where we augment a spatial tree-based index with a grid structure, and study the trade-offs in query processing efficiency at the expense of the extra space that a grid structure consumes. Throughout the paper, the GE-Tree minimizes tree traversal and maintenance times, and supports the range and the k-NN operations efficiently. The GE-Tree is able to achieve constant-time tree operations in contrast to logarithmic time in conventional space-partitioning trees. Also, we study how to perform the range or the k-nearest-neighbor (or k-NN, for short) operations using the GE-Tree. Our experimental results demonstrate that the GE-Tree outperforms conventional tree-only or grid-only indexes up to two times in tree building time and up to six times in performing the range and k-NN operations.

The contributions of this paper are as follows:

- We introduce grid-enabled trees, the GE-tree, a hybrid spatial data structure for minimizing tree maintenance time and enhancing query processing time.
- We study the structure of the GE-Tree, and develop constant-time operations for search, insert, update, and neighbor finding. In addition, we present algorithms for the range and k-NN operations using the GE-tree for efficient spatial query processing.
- We conduct extensive experiments using real datasets. The experimental results demonstrate that the GE-Tree outperforms a conventional tree-only structure for various spatial operations. Also, we compare against a grid-only structure and highlight the cases when the GE-Tree can outperform the grid-only structure and vice versa.

The rest of this paper proceeds as follows. Section 2 provides background material. Section 3 presents the GE-Tree structure and introduces mechanisms for constant-time node operations. Section 4 presents new algorithms for range and k-NN query processing. Finally, Section 5 presents the results of the performance of the GE-Tree in comparison to the grid index and other conventional spatial tree indexes.

2 BACKGROUND

We present an overview of space-driven data structures, the spatial grid, and space-partitioning trees that are core and basic methods for spatial indexing in traditional systems.

2.1 The Grid Structure

The grid-based spatial index is a simple yet effective structure for organizing and searching for spatial objects in the appropriate grid cells. Spatial data processing systems, e.g., SpatialHadoop [5] and LoacationSpark [10] use a spatial grid as a global index. In the 2D space, rectangular grid cells have fixed locations and dimensions, i.e., width w and height h. This allows for constant-time calculations to specify the index numbers, I_X and I_Y, for the grid cells as follows. Given a point, say (x, y), then the grid cell index numbers for it are:

$$I_X = \left\lfloor \frac{x - min(X)}{w} \right\rfloor, I_Y = \left\lfloor \frac{y - min(Y)}{h} \right\rfloor \quad (1)$$

where $(min(X), min(Y))$ is the minimum value of the coordinate space, w is the width and h is the height of the grid cell. Refer to Figure 1(a) for illustration. Assuming that the grid-cell width and height are 15 and 10, respectively, then the index numbers for Point p are calculated as follows:

$$[I_X][I_Y] = [\left\lfloor \frac{53 - 20}{15} \right\rfloor][\left\lfloor \frac{27 - 15}{10} \right\rfloor] = [2][1]$$

Each grid cell contains a pointer to a bucket that holds the details of each data object located in the grid cell. Thus, accessing a bucket requires constant time by calculating (1). Also, finding a neighboring cell for a given cell is simple and requires a constant time to increase/decrease the index numbers by 1. For example, the right side cell for cell[2][3] is cell[3][3]. The main disadvantage of the grid structure is the load imbalance among the grid buckets for data with skewed distributions. In the real world, the locations of the objects (e.g., the GPS tracks or Tweets data) are not randomly generated, but are closely related to other parameters, e.g., the time zone or the geographical traits. With the grid structure as a spatial index, it is likely that only a small number of the cells is active (i.e., form hot-spots) and the remaining cells are redundant (i.e., form cold-spots).

2.2 Space-Partitioning Trees

To deal with data skew, space-partitioning trees (e.g., the quadtree [6] or the k-d tree [2]) are commonly used in real-world applications, e.g., [1, 8, 10, 13]. The quadtree [6] is one of the prominent data structures due to its ability to adaptively subdivide a densely populated node recursively into four equal-sized quadrants, and re-distribute the node's objects into the quadrants until each node holds objects less than a specific threshold, e.g., a certain bucket size.

The k-d tree [2] also decomposes its densely populated nodes recursively. A line parallel to one of the two-dimensional axes is picked. The location of this line is determined so that the line splits the objects in the original node into two mostly equal groups of objects. Each group of objects is pushed into one of two child nodes. In the two-dimensional space, the root node of the tree would be divided by a median x value of objects in the root. Then, the two child nodes of the root would be divided by the median y values of the objects in each node.

In both the quadtree and the k-d tree, all data objects are typically stored at the leaf-level nodes. Searching for objects in the index involves a traversal from the root to the leaf

nodes. For each intermediate non-leaf node, the next child node to be visited is decided by the pivot value, i.e., a center point or splitting line of a node in the quadtree or the k-d tree, respectively. The time complexity of the tree traversal depends on the height of the tree, which is proportional to a log function of the length of the space or the number of objects in the case of the k-d tree.

Finding a neighbor node of a given node requires a tree traversal as well. Given a leaf node , say a, in a quadtree, its neighbor node, say b (in the right, left, upper, or bottom directions) can be found by moving upward in the tree until we find a common ancestor of a and b then moving downward by mirrored order [9]. The k-d tree requires a tree traversal to find its neighbors node as well.

3 THE GRID-ENABLED TREE

In this section, we introduce the Grid-Enabled Tree, a grid-based tree structure for efficient spatial data indexing and query processing. As the name indicates, the GE-Tree is a hybrid indexing approach that combines the strengths and conceal the weaknesses of both the grid and the tree indexing. The main purpose of studying the GE-Tree is to investigate what benefits, if any, that may result from augmenting a grid to the leaf level of a tree-based index.

For a given point object and a space partitioning tree structure, finding an appropriate leaf node requires a tree traversal from the root to the leaf nodes. Tree traversal takes logarithmic time, as discussed above, and those traversals can cause bottlenecks when the tree is used as a spatial index in a system that requires minimal latency.

Notice that several data structures use a grid as part of their structures, e.g., the pyramid data structure [11] that is a multi-resolution data structure where the root of the pyramid is one node that covers the entire space. The children nodes of the root collectively cover the same space as that of the root but at a finer resolution. Similarly, each consecutive level of the pyramid covers the same space but at a finer resolution. The leaf nodes of the pyramid form the finest resolution, and collectively they form a grid structure that covers the entire space. The difference between the GE-tree and the pyramid data structure is as follows. While both structures share the finest resolution grid, in contrast to the pyramid, in the GE-tree, a leaf node of the tree does not have finer resolution child nodes underneath except for the grid.

One of the potential goals of the GE-Tree is to eliminate tree traversal in order to achieve constant-time access to get to a leaf node as well as to take full advantage of a space partitioning tree. In other words, the GE-Tree should not require logarithmic time for the tree traversal even though it maintains a hierarchical structure. Various types of space-partitioning trees can be used within the GE-Tree. For a tree to be used within the GE-Tree, it has to satisfy following requirements: 1) The entire region that is associated with the parent node needs to be fully decomposed to child sub-regions, and 2) Nodes at the same level should be disjointed from each other. A quadtree is one tree that meets these requirements. Thus, in the following sections, we use the quadtree as our driving structure for our investigation and study of the GE-Tree. We term the resulting tree the Grid-enabled Quadtree, or GQT, for short. As will be explained in greater detail below, each grid cell in GQT will hold a pointer to a corresponding leaf node, and leaf nodes will contain spatial objects.

3.1 Node Operations in the GE-Tree

In this section, we examine the main index operations: search, insert, update, and neighbor finding in the GE-Tree. We demonstrate how to process each of them in constant time in the context of GQT, i.e., the Grid-Enabled Quadtree structure.

3.1.1 The GE-Tree Index Structure. In order to define the grid where each cell will hold a pointer to a leaf node, we need to define two parameters: the size of the entire space, and the number of cells. The size of the cell and the number of cells in the grid can be calculated from these two parameters. For example, if we want to build GE-Tree for the geographic coordinates system on Earth, the range of the space would be (lon: -180, lat: -90) to (lon: 180, lat: 90) so the size of the entire space is 360/180, which will be width/height of the grid. By dividing the space size by the desired number of cells, we get the cell size in the grid. If we want a GE-Tree with 1024 X 1024 grid cells, the width and height of each cell will be 360/1024, 180/1024 = 0.3515625, 0.17578125. Thus, $min(X)$, $min(Y)$, w, and h in (1) will be -180, -90, 0.3515625 and 0.17578125, respectively.

Once we have the grid with the above parameters, we build the space partitioning tree on top of it. At the very first time, we will have only a root node of the tree so all of the grid cells have pointers to the root node. As objects are inserted into the tree and the number of objects in the root node hits a threshold, the root node of the tree will be split and the corresponding grid cells will need to update their pointers to point to the appropriate (newly created) leaf nodes. Notice that the pointer updates can be performed at the time of the split or lazily on-demand as discussed in Section 3.2 (Lazy Maintenance).

Figure 2 illustrates GQT using a two-dimensional 8×8 grid, and a quadtree on top of it. We place the grid at the leaf level so that every cell of the grid has a pointer to the corresponding leaf node of the quadtree that covers this cell in space. For example, all sixteen cells from grid[0][0] to grid[3][3] are pointing to the same Quadtree Leaf Node '0'. Also, the green, red, and blue cells are pointing to Nodes '22', '201' and '0', respectively. Notice that Nodes 0, 1, 2,

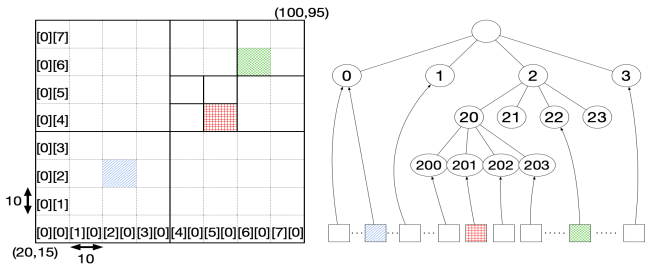


Figure 2: Grid-Enabled Quad tree

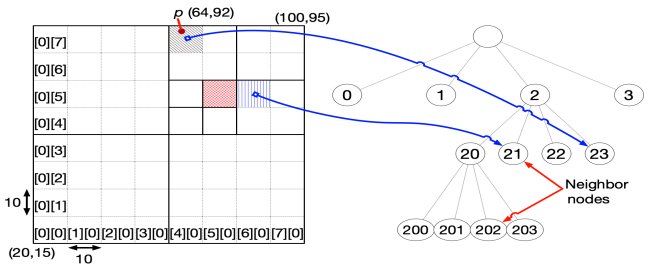
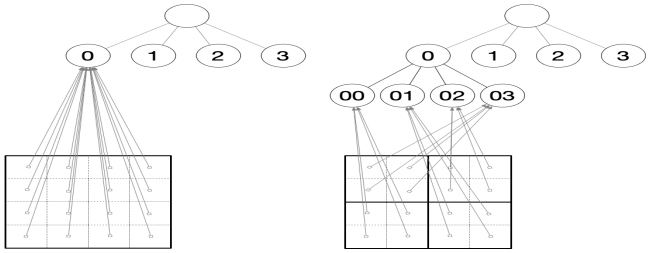
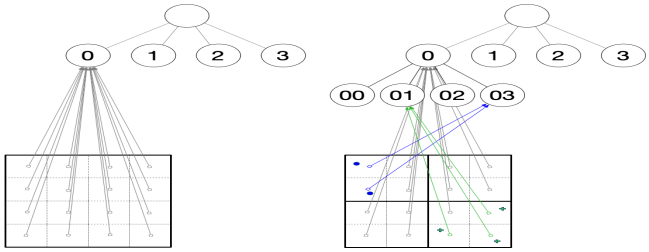


Figure 3: Example of node operations



(a) GE-Tree without Lazy maintenance



(b) GE-Tree with Lazy maintenance

Figure 4: Before/after node split (a)without or (b)with lazy maintenance of grid cells in GE-Tree

and 3 are assigned to the SW, SE, NE and NW child nodes, respectively. Below, we demonstrate the advantages of this hybrid structure.

3.1.2 Constant-Time Access to a Leaf Node. Searching for, inserting, or updating a spatial object involves accessing the leaf node that contains the object. In a conventional quadtree search algorithm, the search starts at the root node, then descends next to the child node that overlaps in space with the object being searched. The higher the depth of the leaf node that is being accessed, the longer it takes to reach this node. Moreover, to insert an object into the quadtree, first we need to search for the correct leaf node that should contain the object. If there are frequent insertions into the tree, the same effort for searching the tree is required each time. Similarly, updating an object involves two searches: one for finding and removing the existing object from the *old* node, and the other for inserting into the *new* node.

With the help of the grid, the GE-Tree search mechanism reduces the time it takes to get to a leaf node. As discussed in Section 2.1, Index Numbers I_X and I_Y for an object are calculated using Equation (1). We use the grid cell index numbers to follow the one pointer in Grid Cell$[I_X][I_Y]$, and access the leaf node without the need for a tree traversal. For example, in Figure 3, the corresponding leaf node for Point p is Node '23' by solving (1):

$$[I_X][I_Y] = [\left\lfloor \frac{64-20}{10} \right\rfloor][\left\lfloor \frac{92-15}{10} \right\rfloor] = [4][7]$$

Note that $w = h = 10$ in this example. Grid Cell[4][7] holds the pointer to Leaf Node '23'. This obviously requires constant time and does not traverse the tree to get the leaf node.

3.1.3 Constant-Time Access to a Neighboring Node. Two nodes that are adjacent to each other are termed neighbor nodes. In the context of a 2D quadtree, neighbor finding refers to the following basic operation: Given a node in the quadtree, find its neighboring node(s) in any one of the cardinal(N, E, S or W) or inter-cardinal(NE, SE, SW or NW) directions. The neighbor finding operation is used in the k-NN query as a building block. For instance, if the k-NN operation has found $< k$ objects, and there are no more candidates in a given node, the k-NN operation expends the searching space by examining the neighboring nodes to the current node. Finding a neighbor node in a conventional quadtree requires tree traversal as discussed in Section 2.2.

In the GE-Tree structure, neighbor finding is performed in constant time without the need for traversal. Suppose that we want to find a right-side (East) node of Node '202' in Figure 3. First, we need to get the coordinates of the bottom-right corner of Node '202' to find out which cell in the grid is the rightmost cell that is pointing to Node '202'. We calculate the index numbers for the point at the bottom-right corner of Node '202', and we find them to be [5][5]. Then, the right-side neighboring node for Node '202' is pointed by a grid cell that is to the right of Grid Cell[5][5], so that we can get the pointer indicating the correct neighbor node from Grid Cell[5+1][5], which is Node '21'. A node in any direction can be accessed analogously. Again, we achieve constant time access to find a neighbor node without traversing the tree.

n_{isLeaf}	$n_{isSplit}$	node's status
true	-	leaf node
false	*true*	merged to parent
false	*false*	divided to children

Table 1: Node's status for lazy maintenance

3.2 Lazy Maintenance of the Grid Cells

If not done right, maintenance of the grid in the GE-Tree structure would be costly. The reason is that each grid cell contains a pointer to the corresponding tree node. Splitting or merging of a tree node involves a costly update to the grid cells because the former node is no longer a leaf node after the splitting/merging and the cells pointing to the node being split/merged do not hold valid pointers to a leaf node anymore. If node splitting or merging occurs frequently, the overhead of reassigning pointers of the affected cells becomes expensive. To resolve this issue, we adopt a *lazy maintenance* technique to update the pointers on-demand, only as needed. This lazy maintenance approach is explained next.

In the GE-Tree structure, each node, say p, has two flags, n_{isLeaf} and $n_{isMerged}$, to denote the current status of the node. Once Leaf Node p gets split into its children, we set p's flags to be $n_{isLeaf} = false$, $n_{isMerged} = false$, and set the newly created child nodes' flags as $n_{isLeaf} = true$, $n_{isMerged} = false$. Likewise, if leaf nodes are merged into their parent node, then the $isLeaf$ and $isMerged$ flags for the leaf nodes are set to $false$, and $true$, respectively, and the parent node's $isLeaf$ flag to $true$. Table 1 gives the status of a node for the various combinations of values for the n_{isLeaf} and $n_{isMerged}$ flags.

Using the flags in a node, a leaf node pointer in a cell is updated only when the $isLeaf$ flag is true. For example, in Figure 4(a), when Node '0' is divided to four child nodes without *Lazy Maintenance*, all of the grid cells for Node '0' (i.e., all 4×4 cells) need to be updated to point to the corresponding child nodes, '00', '01', '02', or '03', based on the location of each of the 16 grid cell. This would result in massive overhead, especially when there are frequent node splitting and merging in the GE-Tree. The GE-tree applies a *Lazy Maintenance* technique. Assume that we access a grid cell, say c. We check the current status of the node, say p, that c points to. If p's flags are $n_{isLeaf} = false$ and $n_{isMerged} = true$, then p is not a leaf anymore as p has been merged in a previous tree update step, and one of p's ancestor nodes, say Node q is supposed to be the new leaf that c should be pointing to. Thus, c needs to be updated to point to q; the correct leaf node. To locate q, we start from Node p and traverse upward to each of the ancestors of p. For each visited node, say s, we check the n_{isLeaf} flag for Node s. If s's n_{isLeaf} flag is false, we set s to s's parent and repeat this step. We stop when we reach a node s with $n_{isLeaf} = true$ flag. The second case that we need to consider is when Grid Cell c is pointing to a node, say p, with the flag settings $n_{isLeaf} = false$ and $n_{isMerged} = false$. This setting corresponds to the case when Node p got split one or more time during a former tree update operation. Thus, to fix Grid Cell c's pointer to point to the correct leaf node, we need to descend the tree under p where the child nodes that we descend have to contain Grid Cell c. This descending step is repeated as long as the $isLeaf$ flag for the visited child node is set to false. Once one of the descendant nodes of p, say q that spatially contain Grid Cell c has its $isLeaf$ flag with true, we stop descending and set c's pointer to q, which is the current leaf node that contains c. As in Figure 4(b), grid cells in a hot-spot area will indicate correct leaf nodes as they are visited more often.

4 QUERY OPERATIONS

In this section, we study two query operations; the range search and the k-NN operations for the GE-Tree. The GE-Tree structure uses the grid to directly locate leaf nodes. Thus, we need to study how the range and k-NN operations benefit from the presence of the grid.

4.1 The Range Search Operation

Consider the case when a quadtree is used as the space-partitioning tree for the GE-Tree, i.e., what we term GQT. A conventional tree-based quadtree performs a range search operation by recursively descending the tree starting from the root node down as long as the visited nodes overlap the query range. Once the leaf nodes are reached, the objects in these leaf nodes that overlap the query range will be reported as output of the range search operation [3]. If the leaf node is fully contained in the range of the query, all of the objects in the node will be returned as output without further validation.

The way the GE-Tree structure accesses leaf nodes depends on the grid. This affects the algorithm for the range search operation as we illustrate below. Mainly, we examine how the range search operation is performed over GQT, and then discuss its correctness. Algorithm 1 and Figure 5 show how the new range query works in GQT with running examples for the query represented by the red rectangle. For the range search algorithm, we maintain two additional data structures to facilitate efficient processing, mainly (1) a First-In-First-Out (FIFO) queue to store the index coordinates of grid cells, and (2) a hash map to store the resulting leaf nodes.

The range search algorithm for the GE-Tree starts from the lower-left corner of the query range boundary, and moves towards the upper-right corner. In each node, we examine two adjacent nodes in the right and upper directions, as in Lines 6 and 12 in Algorithm 1 respectively, then insert the node

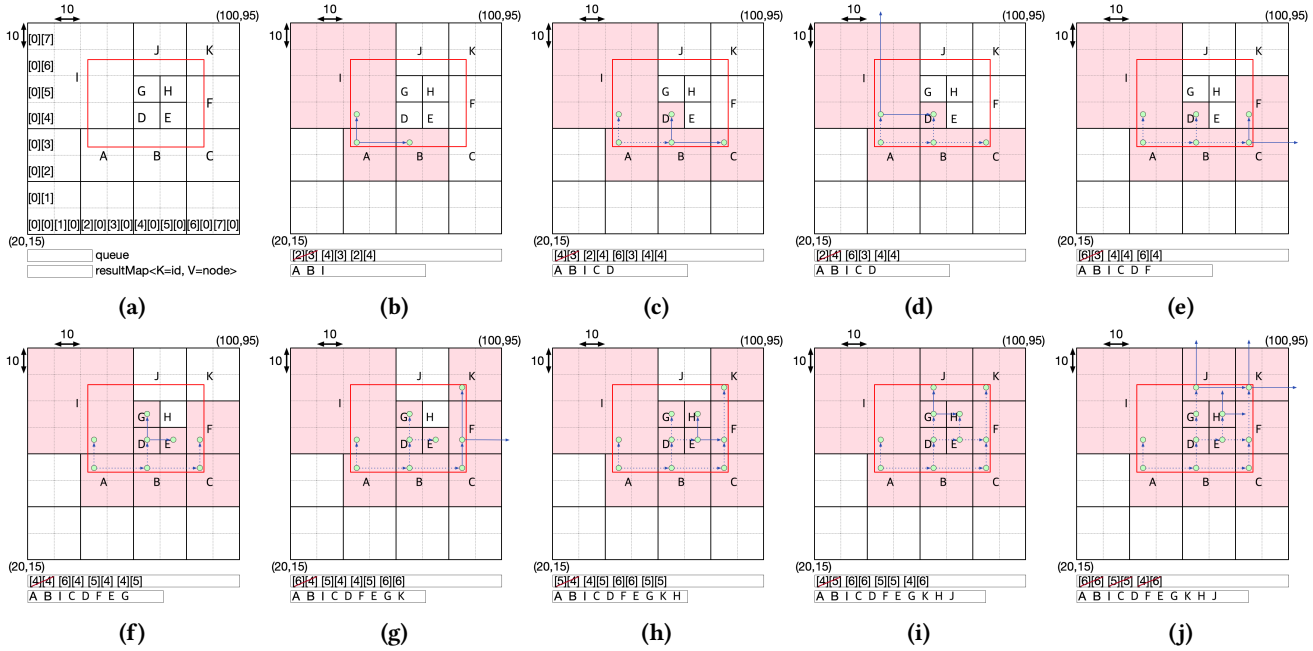


Figure 5: An example of range query on GQT

into the result hash map if the node has not been accessed before. In Line 7 of the algorithm, there are two cases to set the grid cell index of the node that will be enqueued based on the node's position within the search range. If the current node intersects with the bottom-side border of the query, we get $Index_X$ from the first column in the node and the same $Index_Y$ from the previous search step, i.e., the grid cell index of Node B in Figure 5(b) is [4][3] where $Index_X = 4$ is from the first column of Node B and $Index_Y = 3$ is from the previous grid cell index enqueued in Node A. Otherwise, if the current node is fully contained in the search range, we get the grid cell index from the lower-left corner cell in the node. Similarly, this strategy is applied to the upper-direction search in Line 13. If a node is under the left-side border of the query, it will get the same $Index_X$ from the previous search step and $Index_Y$ from first row of the node. This is illustrated by the following example.

In Figure 5, we start from the Grid Cell [2][3] and insert the index coordinates into the *queue*. The node indicated by the cell is Node 'A', and it is fetched in constant time as discussed in Section 3.1.2. Then, we insert Node 'A' into the *resultMap*. Next, we calculate the index numbers for the right-side node, i.e., Grid Cell[4][3] as described in Section 3.1.3. Grid Cell[4][3] overlaps the range of the query, but is not in the *resultMap*. Thus, we insert the *id* of the leaf node pointed by Grid Cell[4][3], i.e., Node 'B', into the *resultMap*, and enqueue Grid Cell [4][3]. In addition, we probe an upper-side neighbor, i.e., Node 'I', fetched by Grid Cell[2][4], and insert them into the *resultMap* and into the *queue*. In the

Algorithm 1: Range query algorithm using GQT

Input: query range
Output: *resultMap* having all leaf nodes' id intersected with the query range

```
1  cell ← bottom-left cell of the query range;
2  queue.enqueue(cell.index());
3  resultMap.put(cell.node());
4  while queue is not empty do
5      cell ← queue.dequeue();
6      rID ← cell.getRightSideNodeID();
7      rIndex(X,Y) ← cell index of the rID node;
8      if rIndex_X ≤ query range then
9          if not resultMap.containsKey(rID) then
10             resultMap.put(rID);
11             queue.enqueue(rIndex(X,Y));
12     uID ← cell.getUpperSideNodeID();
13     uIndex(X,Y) ← cell index of the uID node;
14     if uIndex_Y ≤ query range then
15         if not resultMap.containsKey(uID) then
16             resultMap.put(uID);
17             queue.enqueue(uIndex(X,Y));
18 return resultMap
```

next iteration, the head of the *queue* is Grid Cell[4][3]. Thus, we search Grid Cell[4][3]'s two neighbor nodes towards the

right and upper directions, i.e., [6][3]/'C' and [4][4]/'D' in Figure 5(c). In Figure 5(d), the right-side node of Grid Cell[2][4] is Node 'D' with the Grid Cell [4][4]. At this point, Node 'D' is no longer processed as Node 'D' is already in the *resultMap*. Grid Cell[4][4] will be searched and processed when this grid cell gets dequeued from the *queue* in future iterations. When searching toward the upper direction from Grid Cell[2][4], I_Y exceeds the query range, and thus we skip progressing in this direction. The next step is to process Grid Cell[6][3], as in Figure 5(e). We skip the right-direction procedures because it exceeds the query range and insert Grid Cell[6][4] (Node 'F') into the *queue* (*resultMap*), respectively. The next iterations are illustrated in Figure 5(f), 5(g) and 5(h), where we process the grid cells/Node pairs [5][4]/'E', [4][5]/'G', [6][6]/'K', and [5][5]/'H'. The remaining elements in the *queue* do not affect the *resultMap*. Once we iterate over all of the items in the *queue*, we get the complete set of nodes that overlaps the spatial range of the range search operation. The proof of correctness of the range search is omitted to conserve space.

4.2 The k-NN Operation

In this section, we present an algorithm for the k-nearest neighbor (k-NN, for short) operation in GQT, and investigate how k-NN can benefit from the presence of the grid, and how it retrieves correct results in an efficient way. Hjaltason and Samet [7] present the Distance Browsing algorithm, where they use a priority queue to select the nearest elements and to avoid repeated tree traversal in the context of a PMR quadtree while retrieving a next nearest neighbor. In addition to the nearest elements, the priority queue also stores and priorities all the intermediate tree nodes visited during the search path to the leaf node that contains the k-NN query point. A node will be examined when it reaches the head of the queue. The GE-Tree does not traverse the tree to get the leaf node so we need to adapt the Distance Browsing algorithm [7] to make use of the grid and process the k-NN query properly.

Algorithm 2 presents how the k-NN query is processed using GQT. The algorithm borrows the idea of a priority queue, where both the nodes and objects are inserted into the same queue and are ordered by the distance from the query point. Given a query point q, we can get a leaf node as discussed in Section 3.1.2 and insert the node into the priority queue (Lines 2-4). If the head of the queue is an object, we insert the object into the *resultArray*, and return the result if we have k nearest neighbors (Lines 7-10). If the head of the queue is a node, we first insert all the objects in the node into the priority queue to be ordered by distance from the query point (Lines 12-13). In addition, there is a case that the next nearest neighbor object could exist in an adjacent node, and not necessarily in the current node. Thus, we insert the nodes adjacent to the current node into the queue (Lines 14-16). Getting all adjacent nodes to a given node requires

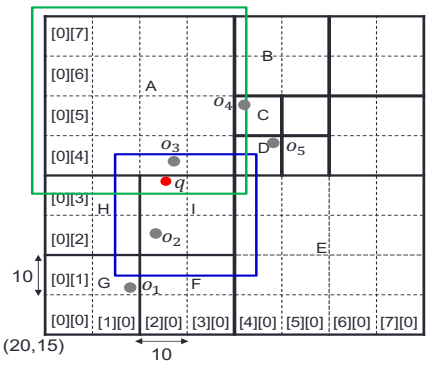


Figure 6: k-NN query example

multiple neighbor finding operations, as discussed in 3.1.3. The conditional statement in Line 15 is to filter out a node because if $Dist(q, node) < Dist(q, element)$, the *node* has to be processed in the past iteration. Notice that items in the priority queue are ordered by the distance from the query point. Lines 17-18 eliminate duplicate nodes from the queue. Lastly, Line 19 is to return the current result sets when the *queue* is empty before we find all the k objects. In this case, the total number of objects in the index is actually less then k, so all of the objects will need to be reported as output.

Refer to Figure 6 for illustration. Suppose that we want to process a k-NN query with $k = 2$ and the distances of objects and nodes from the query point $q = (45, 54)$ are as follows: { $o_1 : 27.2$, $o_2 : 14.1$, $o_3 : 4.12$, $o_4 : 25.49$, $o_5 : 25.63$ } and { $A : 1.00$, $B : 25.81$, $C : 18.60$, $D : 15.03$, $E : 15.00$, $F : 19.00$, $G : 19.65$, $H : 5.00$, $I : 0$ }. At the first iteration, we dequeue I (Line 6), enqueue the object o_2 (Lines 12-13) then enqueue neighbor nodes of the I, which are G, F, E, D, A, and H (Lines 14-16). At this point, the priority *queue* is holding [A, H, o_2, E, D, F, G] as ordered by the distance from q. In the next iteration, we dequeue A, enqueue o_3 and the neighbor nodes under the green line, i.e., Nodes H, E, D, C, and B. Notice that I is filtered out because the distance between I and q is 0 so that the *queue* has [o_3, H,H, o_2, E,E, D,D, C, F, G, B] at this moment. Next, we dequeue Object o_3 and insert it into the *resultArray* (Line 8). In the next step, we dequeue H, and enqueue G and F, so that the *queue* contains [H, o_2, E,E, D,D, C, F,F, G,G, B]. At this point, the head of the *queue* equals to the current *element*, H, so we dequeue head items until the head differs from the current *element*. Therefore, the *queue* becomes [o_2, E,E, D,D, C, F,F, G,G, B]. The dequeued element in the next iteration is o_2 so it is added to the *resultArray* then the loop is terminated and returns the output results.

The main difference between the Distance Browsing algorithm and the grid-enabled tree algorithm is in how to hold nodes in the priority queue. In the Distance Browsing

Algorithm 2: k-NN algorithm using GQT

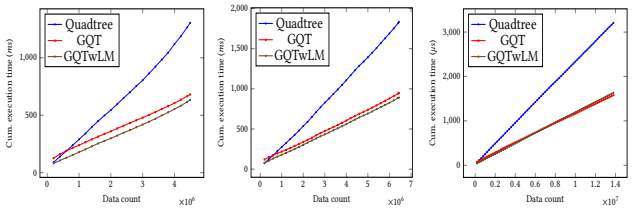
Input: q: query point, k: number of neighbors
Output: *resultArray* containing k nearest objects

```
1  resultArray ← NewArrayList();
2  leaf node ← getLeafNode(queryPoint);
3  queue ← NewPriorityQueue();
4  queue.enqueue(0, leafnode);
5  while queue is not empty do
6      element ← queue.dequeue();
7      if element is an object then
8          resultArray.add(element);
9          if resultArray.size() == k then
10             return resultArray
11     else // element is a leaf node
12         for each object in leaf node element do
13             queue.enqueue(Dist(q, object), object);
14         for each adjacent node of the node element do
15             if node not visited before then
16                 queue.enqueue(Dist(q, node), node);
17         while element == queue.head() do
18             queue.dequeue();
19 return resultArray
```

algorithm, the leaf node is accessed in a top-down manner, where the queue holds all intermediate nodes in the searching path from the root to the leaf node and examines the leaf node when it is at the head of the queue. However, the GE-Tree does not touch a non-leaf nodes so that the queue is maintained in a bottom-up fashion, i.e., the leaf nodes are enqueued gradually as the searching proceeds.

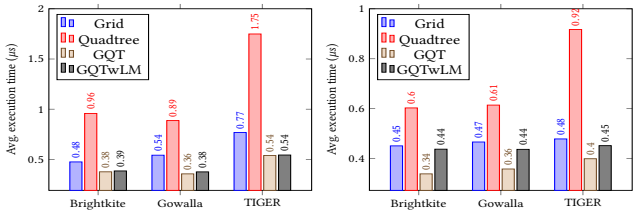
5 PERFORMANCE ANALYSIS

In this section, we evaluate the performance of GQT for tree building (insertions), node access, neighbor finding, range search, and k-NN query operations as compared with a conventional quadtree. All the experiments are conducted on a machine running Mac OS 10.13.4 on Intel Core i7 with 2.3 GHz and 16 GB of main memory. We use three datasets [4, 12] with millions of points: Brightkite(4.4m), Gowalla(6.4m) and TIGER(13.9m). In the Datasets Brightkite and Gowalla, points having no coordinates are excluded from the experiments. The TIGER data is a set of points that are sampled from all coordinates of roads in the United States. The performance metric that we use is the execution time of the operations. In all experiments, the resolution of the grid in GQT is set to $2^{12} \times 2^{12}$, and the node capacity of the tree is set to 100.



(a) Brightkite (b) Gowalla (c) TIGER

Figure 7: Performance of tree building (insertions)



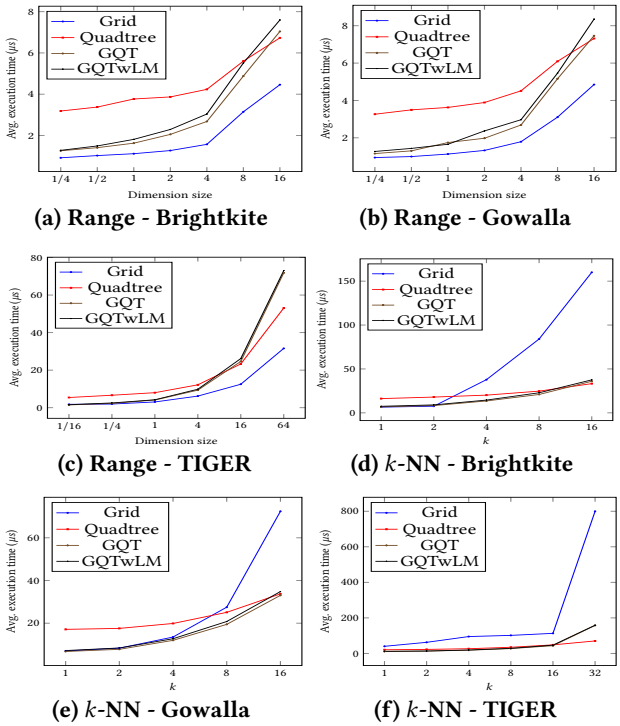
(a) Node access (b) Neighbor finding

Figure 8: Performance of node operations

5.1 The Performance of the Insert and Search Operations

First, we analyze the performance of the insert operation when building a GE-tree. We compare both GQT and GQT with Lazy Maintenance (GQTwLM, for short) against a conventional quadtree and a grid. As the quadtree grows by node splits, the time it takes to traverse the quadtree to reach to the leaf node increases. However, with the help of the grid, to access a leaf node, GQT reduces that time significantly. The three graphs in Figure 7 illustrate that the total execution time of tree construction in GQT achieves up to 2x speedup on all the tree datasets (Brightkite, Gowalla and TIGER).

Observe that GQTwLM performs better than the GQT in Datasets Brightkite and Gowalla due to the use of *Lazy Maintenance* but does not have an advantage in the case of the Tiger dataset. The reason is as follows. As discussed in Section 3.2, GWTwLM avoids unnecessary updates to the grid cells by correcting the node pointers only on demand in a lazy fashion and not eagerly as explained in Section 3.2. The data points in Datasets Brightkite and Gowalla are highly skewed. Thus, there exist many grid cells that do not need to be updated throughout the insertions during tree construction. Although there are few points at the very beginning of the tree building, GQT needs to update its grid cells (up to $2^{12} \times 2^{12}$) to point to the corresponding leaf nodes. On the other hand, the TIGER data has more balanced and uniformly distributed data points than the other two datasets. This implies that most of the grid cells in GQT and in GQTwLM are occupied, and need to be updated in the tree-building phase. Thus, in this case, for the Tiger dataset, the effect of the *Lazy Maintenance* is minimal.



(a) Range - Brightkite (b) Range - Gowalla

(c) Range - TIGER (d) k-NN - Brightkite

(e) k-NN - Gowalla (f) k-NN - TIGER

Figure 9: Performance of range and k-NN query

After the trees are constructed for each dataset, we perform micro-benchmarks to study the search performance on GQT using a testset. Each testset has 1,000 sampled points from each of the datasets and varied their coordinates with ±0.25 so the test points are still skewed in the search space. In the case of a grid-only structure, we initialize a fixed-size grid structure then insert data points into the grid. Note that the grid is a static structure so the grid cells are neither divided nor merged. After the insertions, we conduct the same procedures using testsets to test the search performance of the grid. First, we examine the access time to reach to the leaf node with the testset on each of the grid, quadtree, GQT, and GQTwLM. As illustrated in Figure 8(a), GQT and GQTwLM perform better than the quadtree in all cases. From this experiment, observe that GQT has shorter execution time than GQTwLM. The reason is that, at search time, all of the grid cells in GQT already have the correct pointers to leaf nodes, while this is not the case for GQTwLM. In GQTwLM, some of the node pointers in the grid cells are not up-to-date due to the splits during insertion. Therefore, GQTwLM takes a little more time to update its grid cells, which is due to the *Lazy Maintenance* saved during insertions. Another search operation is finding a neighboring leaf node from a given one. As discussed in Section 2.2, the conventional quadtree requires a tree traversal to find a neighboring node from a given node. In contrast, GQT avoids this traversal during neighbor finding as discussed in Section 3.1.3. We conduct experiments with similar testsets as discussed above to examine the neighbor finding execution time. Figure 8(b) demonstrate that the neighbor finding in GQT and GQTwLM performs better than the one in the quadtree by 250%.

Although the grid and GQT perform better than the conventional quadtree, it is interesting to observe that GQT performs better than the grid for node operations. One of the reasons could be due to memory management issues. In the grid for a skewed dataset, there could be lots of unused nodes wasting memory resources or hot-spot nodes containing excessive objects in its container leading to slower performance when using the grid.

5.2 Range Query Performance

In this section, we study the performance of the range query on the grid, the quadtree, GQT, and GQTwLM. As in Figure 5, a range query is defined by a location (x, y) and dimensions (w × h). We use the query location from the testsets described in the experiments above and vary the query dimensions to study the effect of the range size.

Although we achieve constant-time access to perform the neighbor finding operation, there exist maintenance overheads due to the FIFO queue and the hash map in Algorithm 1. In contrast, the conventional range query algorithm requires tree traversals but it has minor computational overhead at each level in the tree to find a node's child. The range search algorithm for GQT involves some computational overhead to "compute" the locations of the leaf nodes inside the search range. Notice that the amount of computational overhead in GQT is proportional to the number of leaf nodes inside the search range. Generally speaking, the smaller the search range, the smaller the computational overhead. Therefore, it is expected that GQT's range query algorithm would perform better as long as its computational overheads are smaller than the tree traversal time. In other words, Algorithm 1 performs better when the height of the tree is large and the size of the range query is small. In this experiment, we vary the size of the query range $sizeOfQueryRange = d \cdot cellSize$ by adjusting dimension value, d.

Figures 9(a), 9(b), and 9(c) give the performance results of the range query for the grid, the quadtree, GQT, and GQTwLM. For the datasets Brightkite and Gowalla, the computational overhead in either GQT and GQTwLM is smaller than the time for the tree traversals in the conventional quadtree when $d < 8$. For the TIGER dataset, the calculation overhead in both GQT and GQTwLM exceeds the tree traversal time when $d \geq 16$. For small sized search ranges, we achieve a speedup of up to 2.53, 2.82, 3.65 times in GQT and 2.49, 2.57, 3.41 times in GQTwLM on contrast to using the conventional quadtree for Brightkite, Gowalla and TIGER data. For the same reason discussed in the previous sections (i.e. *lazy maintenance*), GQT performs better than GQTwLM

in the skewed dataset cases. The grid structure beats all others in the range query. Although the grid needs to access much more cells than the quadtree or GQT, its overhead for range query is smaller than the other structures.

5.3 k-NN Query Performance

In this experiment, we examine the performance of the k-NN query using GQT and GQTwLM in comparison with the grid and the conventional quadtree. Similar to the range search experiments, we use the testsets for the query point and adjust k, the number of nearest neighbors. The same contrast between the computational overhead versus the tree traversal time exists in the k-NN query as above. Even though the k-NN algorithm for GQT provides a bottom-up approach for computing k-NN, the computational overheads exist during the process of finding the neighbor nodes (Lines 14-16 in Algorithm 2). Therefore, we expect that the performance of the k-NN performs better when the height of the tree is large or the number k is small and the points are clustered near each other, so that the computational overheads in GQT do not exceed the tree traversal time in the quadtree.

Figures 9(d), 9(e), and 9(f) give the performance of the k-NN query in the grid, the quadtree, GQT, and GQTwLM. The results demonstrate that the performance of the k-NN query is better for smaller k values, i.e., $k < 16$ for all datasets. Similar to the range query experiments, GQT performs better than GQTwLM due to the additional overhead due to *Lazy Maintenance* on the deprecated nodes in GQTwLM. As discussed earlier, the main disadvantage of the grid is the load imbalance with skewed data points. For the k-NN operation, as k increases, many adjacent cells need to be accessed to find all k points specifically when dealing with skewed datasets. If the skew is high, many of the cells accessed will be empty, and the searching space will need to be extended.

6 CONCLUSIONS

This paper investigates the tradeoffs for adding a grid along with a space-partitioning tree. It introduces the Grid-Enabled Tree (the GE-Tree); a hybrid data structure containing a grid and a space partitioning tree. The paper's focus is when the spatial tree is actually a quadtree (GQT). The GE-Tree handles skewed datasets, achieves constant-time node access operations, and provides *Lazy Maintenance* functionality to save time for updating unnecessary cells in GE-Tree. We also present new algorithms for the range and k-NN queries used in GQT. The performance study demonstrates there is a trade-off between the time it takes to traverse the tree and is affected by the height of the conventional quadtree, and the computational overhead it takes to "compute" the locations of the leaf nodes and directly accessing them in GQT. The performance study illustrates that GQT outperforms the conventional quadtree structure in the range and k-NN queries in specific circumstances (e.g., when the height of the tree is big and the search range is small in the case of the range search operation or when the value of k is small in the case of the k-NN operation). GQT outperforms the conventional quadtree for node access operations, e.g., when searching for a query spatial object or for the neighbor finding operation. Also, GQT performs better than the grid-only structure on node access operations, specifically on k-NN with large k.

ACKNOWLEDGEMENTS

Walid G. Is the Aref acknowledges the support of the NSF under Grant Numbers. III-1815796 and IIS-1910216.

REFERENCES

[1] Furqan Baig, Hoang Vo, Tahsin Kurc, Joel Saltz, and Fusheng Wang. 2017. Sparkgis: Resource aware efficient in-memory spatial query processing. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 28.

[2] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[3] Jon Louis Bently and Donald F. Stanat. 1975. Analysis of range searches in quad trees. *Inform. Process. Lett.* 3, 6 (1975), 170 – 173. https://doi.org/10.1016/0020-0190(75)90034-4

[4] Eunjoon Cho, Seth A Myers, and Jure Leskovec. 2011. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1082–1090.

[5] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st international conference on Data Engineering*. IEEE, 1352–1363.

[6] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4, 1 (1974), 1–9.

[7] Gísli R. Hjaltason and Hanan Samet. 1999. Distance Browsing in Spatial Databases. *ACM Trans. Database Syst.* 24, 2 (June 1999), 265–318. https://doi.org/10.1145/320248.320255

[8] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In *2011 IEEE 12th International Conference on Mobile Data Management*, Vol. 1. IEEE, 7–16.

[9] Hanan Samet. 1982. Neighbor finding techniques for images represented by quadtrees. 18, 1 (1982), 37–57.

[10] Mingjie Tang, Yongyang Yu, Qutaibah M Malluhi, Mourad Ouzzani, and Walid G Aref. 2016. Locationspark: A distributed in-memory data management system for big spatial data. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1565–1568.

[11] Steven Tanimoto and Theo Pavlidis. 1975. A hierarchical data structure for picture processing. *Computer graphics and image processing* 4, 2 (1975), 104–119.

[12] U.S. Census Bureau. 2017. TIGER/Line Shapefile. http://www.census.gov/geo/www/tiger/

[13] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 70.