Sequential Algorithms for Constructing an "Out-of-Place" Multidimensional Quad-Tree Index for Answering Exact and Approximate Fixed-Radius Nearest Neighbor Queries

Mugurel Ionut Andreica, Nicolae Tapus

Sequential Algorithms for Constructing an "Out-of-Place" Multidimensional Quad-Tree Index for Answering Exact and Approximate Fixed-Radius Nearest Neighbor Queries

Mugurel Ionuţ Andreica, Nicolae Ţăpuş
Computer Science Department
Politehnica University of Bucharest
Email:{mugurel.andreica, nicolae.tapus}@cs.pub.ro

Abstract

Answering nearest neighbor queries is an important problem in many areas, ranging from geographic systems to similarity searching in object data-bases (e.g. image and video databases). In order to answer such queries efficiently, an index data structure is usually constructed over the searched objects. In this paper we present novel top-down and bottom-up sequential algorithms for constructing a multidimensional quad-tree index. In all the algorithms the objects may be indexed in association with both quad-tree nodes which they intersect and those which they do not intersect. The common aspect of all the algorithms is that, in order to answer a query, only a single node of the index will need to be searched. The work presented in this paper continues the work presented in [21].

1 Introduction

The fixed-radius nearest neighbor problem is defined as follows. Given a set of objects in a D-dimensional space, a query point P and a distance R, find the closest object to the point P located at a distance at most equal to R. Usually, the set of objects is fixed and it requires pre-processing in order to answer multiple queries in an efficient manner. This problem has applications in many areas. The most obvious one is in the domain of geographic information systems (GIS). A user may send his coordinates to a GIS and receive back information about the closest *object of interest* (e.g. address, business, etc.) located at a distance at most R from him/her. Another application is given by similarity search queries. There are many services storing data objects (e.g. images, video clips) which can be described by the values of their features. The set of features forms the feature space. A query point in this space specifies a value for each feature and asks for the object whose features are most similar to those of the query point, but which is not too "far away" from the point.

In this paper we consider a *mixed* nearest neighbor problem with two distance thresholds: $Rmin$ and $Rmax$. We are interested in obtaining *exact* answers up to distance $Rmin$ and *approximate* answers up to distance $Rmax$. The rest of

this paper is structured as follows. In Section 2 we define the problem statement clearly. In Section 3 we discuss the choice of the index data structure. In Section 4 we introduce the main assumptions, prerequisites and we define the main notations used in the rest of the paper. In Sections 5 and 6 we present sequential versions of the "out-of-place" indexing, "in-place" searching algorithm (the definitions of "in-place" and "out-of-place" are given in Section 4), both using a top-down (Section 5) and a bottom-up (Section 6) approach. In Section 7 we discuss a few object filtering methods. In Section 8 we discuss distributed query processing based on the multidimensional quad-tree index constructed by the presented algorithms. In Section 9 we present the proof for the constant factor approximation of our algorithm in the case of the *approximate* requirement. In Section 10 we discuss related work and in Section 11 we conclude and discuss future work.

2 Problem Statement

The problem addressed in this paper is the following. We consider N objects in a D-dimensional space. The objects can be of any type (e.g. points, segments, polyhedra, unions of simpler objects, etc.), where both N and the total amount of data representing the objects are very large. We also consider a distance function over the D-dimensional space (e.g. one of the L_p norms ($1 \leq p \leq +\infty$)) and two distance thresholds: $Rmin$ and $Rmax$. We are interested in efficiently answering the following types of queries: Given a point P in the D-dimensional space, return an object O satisfying the following requirements:

1. if the distance between the point P and the nearest object is at most equal to $Rmin$, then O must be the closest object among all the objects O' located at distance at most $Rmin$ from the point P (the *exact requirement*).

2. if the distance between the point P and the nearest object exceeds $Rmin$ but is at most equal to $Rmax$, then O should be an *approximate* nearest object to P (the *approximate requirement*).

3. if the distance between P and its nearest object exceeds $Rmax$, then O can be any object, or even no object (the *don't care requirement*).

The distance between a point P and an object O is defined in the usual way, as the distance between P and the closest point Q to P, such that $Q \in O$. We will assume that a function which computes the distance between a point P and an object O is given. We assume a normal distance function, without additive or multiplicative weights. We also assume that a distance function computing the distance between two geometric objects exists (if the objects intersect, then the distance is 0; otherwise, the distance between them is equal to the minimum distance between a point of the first object and a point of the second object).

Obviously, satisfying the third requirement is trivial (e.g. just select a *default* object which is returned whenever we do not fit in the first two cases), so we will focus on the first two requirements next.

3 Choice of the Index Data Structure

The same reasons considered in [21] for choosing the index data structure are valid in this case, too. We decided that a (multidimensional) quad-tree is the most appropriate solution, because:

1. the set of potential regions (covered by the nodes of the tree) is decoupled from the set of objects.

2. the regions can be chosen at different "resolutions", thus adapting to the space distribution of the objects.

Our choice of the data structure affects the indexing algorithm to a large degree. However, other data structures can be used instead of the (multidimensional) region quad-tree, as long as they have similar properties.

We considered two possibilities for using the (multidimensional) quad-tree:

1. every object is indexed only in nodes which it intersects; then, at query time, we will have to search multiple tree nodes in order to find the answer (we call this the *"in-place" indexing, "out-of-place" searching* method) (this possibility was investigated in [21]).

2. every object is indexed both in nodes which it intersects, as well as in other (neighboring) nodes; then, at query time, we will only need to search one node containing the query point (we call this the *"out-of-place" indexing, "in-place" searching* method) (this possibility is described in this paper).

4 Main Prerequisites, Assumptions and Terms

The same prerequisites, assumptions and notations from [21] are used in this paper, too. For completeness, we include most of them in this section. First, we will describe in more detail what a multidimensional quad-tree is. Each node of the tree has a unique identifier and corresponds to a finite hyper-rectangular region of the D-dimensional space, having a pre-specified aspect ratio. To be more precise, let $(c_1, \ldots, c_{D-1})$ be a set of constant positive values and let $(L_1, \ldots, L_D)$ be the side lengths of a hyper-rectangle corresponding to any tree node. Then we must have $L_i/L_D = c_i$ (for $1 \leq i \leq D-1$). $(c_1, \ldots, c_{D-1})$ are constant parameters of the tree. The same holds for another constant $K \geq 2$, which describes how the regions corresponding to a node's sons are computed. Let's consider the hyper-rectangular region $Cell(Q)$ corresponding to a node Q (with Q being the node's identifier). The node has K^D children and their regions are computed as follows: For each dimension i ($1 \leq i \leq D$), divide the side length of $Cell(Q)$ in dimension i into K equal parts, by drawing $K-1$ equally-spaced hyper-planes. $(K-1) \cdot D$ hyper-planes drawn this way divide the interior of $Cell(Q)$ into K^D equal hyper-rectangles, each of them having the same aspect ratio as $Cell(Q)$ (the usual region quad-tree in $2D$ uses $K = 2$ and the region of each node is a square). Each of these hyper-rectangles corresponds to a child of Q. Note that we consider $Cell(Q)$ to contain all the points in its interior.

Each node Q of the tree has an associated level $Level(Q)$ ($Level(root) = 1$, where *root* is the root node of the tree) and every node Q except the root has

a parent, $Parent(Q)$. In theory, the tree can have an infinite number of nodes. Because of this, we will set a threshold $MaxLevel$ and we will consider that the nodes at the level $MaxLevel$ have no children.

Given the identifier Q of a node, the following functions must be computed efficiently, preferably based only on Q and the constant parameters of the tree (i.e. $(c_1, \ldots, c_{D-1})$ and K):

- $Level(Q)$: returns the level of the node.
- $Parent(Q)$: returns the identifier of the node's parent.
- $Cell(Q)$: returns the geometric representation of the hyper-rectangle (*cell*) corresponding to the node Q.
- $Children(Q)$: returns a set consisting of the identifiers of the node's children (if any); nodes at level $MaxLevel$ have no children and the result is not defined for $Level(Q) > MaxLevel$.
- $Neighbors(Q)$: returns a set consisting of the identifiers of the nodes Q' such that $Level(Q') = Level(Q)$ and $Cell(Q')$ touches $Cell(Q)$ in at least one point.

Based on these functions, we can define the following extra functions:

- $Siblings(Q)$: returns the set of identifiers of all the nodes Q' such that $Parent(\ Q') = Parent(Q)$
- $ExtNeighbors(Q)$: returns the set consisting of node Q's neighbors and siblings (together called extended neighbors)
- $Descendants(Q, dlevel)$: returns the identifiers of all the descendants of Q located at the level *dlevel*
- $Ancestor(Q, alevel)$: returns the ancestor of the node Q located at the level *alevel*
- $Ancestors(Q, alevel)$: returns the set of ancestor of the node Q located at or below the level *alevel*

The function $Ancestors(Q, alevel)$ can be extended to $Ancestors(S, alevel)$, where S is a set of nodes. In this case, $Ancestors(S, alevel)$ returns the union of the sets $Ancestors(Q, alevel)$ for $Q \in S$.

Using the Z-order (or Morton curve) [19] in order to assign identifiers to the nodes of a multidimensional quad-tree helps us to easily implement all the functions mentioned above. However, other encoding schemes with similar properties are also possible [19].

Another function which we require is:

- $Cover(F, clevel)$: returns the set of all the node identifiers Q such that $Level(Q) = clevel$ and the geometric figure F intersects $Cell(Q)$.

The $Cover(F, clevel)$ function can be implemented easily for connected figures F. One possible implementation is the following. First, we find a point $P \in F$ and compute the identifier Q such that $Level(Q) = clevel$ and $P \in Cell(Q)$ (i.e. we find the node at level *clevel* containing the point P). This can

be easily achieved, by computing the position of this node in the level *clevel* grid of nodes. Then, we will perform a breadth-first search traversal starting from that node. We will visit all the level *clevel* nodes starting from Q which are intersected by the figure F (once a node is visited, we add it to a queue; when we extract a node from the queue, we visit all of its non-visited neighbors intersected by the figure F). If F is disconnected, we can still use the same algorithm, as long as we know the coordinates of a point P from each connected component.

Note that, since the $Distance$ function does not include additive distance weights, we have the equivalence between:

- object O intersects $Cell(Q)$ **and**
- $Distance(O, Cell(Q)) = 0$

However, in the algorithms presented in the rest of this paper, we will not necessarily assume the previous equivalence and we will consider that the intersection between an object and a node is computed geometrically.

We will also make use of two other functions, which can be implemented easily:

- $Diameter(F)$ which returns the diameter of the figure F, i.e. the largest distance between any pair of points belonging to F.
- $Border(Q)$: returns a geometric representation of the hyper-rectangle $Cell(Q)$, but without the points in its interior (i.e. containing only the border of $Cell(Q)$).

Using the functions defined above, we can define a new function: $Inflate(F, ilevel, radius)$ which returns a set of identifiers of all the nodes at level *ilevel* whose nodes are at distance at most *radius* from the geometric figure F. We will refer to the set of nodes of the identifiers from this set as a "covering". Actually, we will use a more general function, $ExtInflate(F, ilevel, radius, fraction)$ (Algorithm 1 from [21]). Then, we can define $Inflate(F, ilevel, radius) = ExtInflate(F, ilevel, radius, 0)$. The $fraction$ parameter can be used in order to also include in the covering a node Q if it is adjacent to a same-level node Q' intersecting the figure F and the distance between the figure F and $Cell(Q)$ does not exceed $fraction \cdot Diameter(Cell(Q'))$ (note that since all the nodes at the same level are identical, we have $Diameter(Cell(Q)) = Diameter(Cell(Q'))$. We will use a constant $Frac$ for the value of the $fraction$ parameter.

The final assumption is that each of the N objects O has a unique identifier $id(O)$. This way, we will differentiate between the whole object O (which contains the object's geometry and, possibly, other information) and its identifier.

All the functions defined in this section will be used in the following sections, both at indexing and at query time.

The index consists of a sub-tree T of the complete multidimensional quadtree. During the indexing process, each leaf Q of T will have assigned a list $Lobj(Q)$ of objects which are indexed in association with Q. At the end of the indexing process, we will compute a list $Lid(Q)$ for each leaf Q, where $Lid(Q) = \{id(O) | O \in Lobj(Q)\}$. During our indexing process, we will also use a parameter $MinLevel$, meaning that we don't want to have leaves at a smaller level than $MinLevel$. Because of this, we will define the operation

$SplitAtLevel(Q, slevel)$, which replaces a leaf $Q \in T$ such that $Level(Q) < slevel$ by its descendants at the level *slevel* (see Algoritm 2 from [21]).

We will denote by $Children_T(Q)$ the set of identifiers of the nodes of T which are also children of Q. $Children_T(Q)$ is a subset of $Children(Q)$.

We will associate to each object O a non-negative weight $W(O)$. We provide guidelines as to how this weight should be chosen. The weight should be proportional to:

- the size of the object (i.e. the storage space it takes) and/or
- the duration of computing the distance from a query point to the object

For each leaf Q of the tree, we will maintain a value $WL(Q)$ representing the aggregate weight of the objects associated to Q. We will use an aggregation function $aggf$ (e.g. $aggf = addition$). We will use an indexing weight threshold IWT in order to decide when we need to split a leaf. If the aggregate weight of the objects associated to a leaf Q exceeds IWT and $Level(Q) < MaxLevel$, then we will need to split the leaf.

The time complexities of the algorithms presented in this paper depend both on the maximum depth of the (multidimensional) quad-tree ($MaxLevel$) and on the sizes of the indexed objects (or on the number of cells of the covers and inflated covers of the objects).

5 The Top-Down Method

5.1 Constructing the Index - Addressing the Exact Search Requirement

We insert the objects O one at a time, in an arbitrary order. We use the algorithm $InsertTopDown$ (Algorithm 3 from [21]); the initial call is $InsertTopDown($ $root, O, Rmin, Frac)$, where *root* is the root node of the tree. At the end of the insertion process, the list $Lobj(Q)$ of each leaf Q contains all the objects located at a distance at most equal to *Rmin* from $Cell(Q)$. However, not necessarily all these objects can be nearest object candidates for the points of $Cell(Q)$. Thus, we propose a filtering process, in order to reduce the sizes of the lists $Lobj(*)$. Let's consider that we have a function $Filter(Q, L)$ which takes as input a node id of a leaf in T and a list of objects L and returns a list L' obtained by removing from L (all or some of) the objects which are not nearest neighbors to any part of $Cell(Q)$. We discuss the implementation of the $Filter$ function in a separate section.

During the filtering phase we will use a second threshold FWT for the aggregate weight of the objects associated to any leaf Q and a second value for the maximum allowed level of a leaf: $FMaxLevel$ (with $FMaxLevel \geq MaxLevel$). Unlike the insertion procedure, this part of the indexing process does not need to be implemented sequentially. Thus, we will describe the algorithm using the *Replicated Workers* paradigm. We will use a pool of tasks $TaskPool$; a task is specified by the identifier Q of a leaf on which we need to perform the filtering process. Initially, we insert all the leaf identifiers into $TaskPool$. We will also maintain a counter $NumWaiting$, describing the number of workers waiting for extracting a task from $TaskPool$. Let $NumWorkers$ be the total number of

workers. Algorithm 1 describes the steps taken by each worker. The read/write accesses to $TaskPool$ and $NumWaiting$ are synchronized by a condition variable $Cond$.

Algorithm 1 $ReplicatedWorkersFilter(NumWorkers, TaskPool, NumWaiting, Cond)$

```
Cond.lock()
while (NumWaiting < NumWorkers − 1) or (TaskPool.size() > 0) do
  NumWaiting ← NumWaiting + 1
  while TaskPool.size() = 0 do
    Cond.wait()
  end while
  Q ← TaskPool.ExtractTask()
  NumWaiting ← NumWaiting − 1
  Cond.unlock()
  if Q = FINISH_TASK then
    return
  end if
  Lobj(Q) = Filter(Q, Lobj(Q))
  WL(Q) = 0
  for O ∈ Lobj(Q) do
    WL(Q) ← aggf(WL(Q), W(O))
  end for
  if (WL(Q) > FWT) and (Level(Q) < FMaxLevel) then
    Cond.lock()
    for Q' ∈ Children(Q) do
      Lobj(Q') = Lobj(Q)
      TaskPool.InsertTask(Q')
    end for
    Clear Lobj(Q)
    Cond.notifyAll()
    Cond.unlock()
  end if
end while
for i = 1 to NumWorkers − 1 do
  TaskPool.InsertTask(FINISH_TASK)
end for
Cond.notifyAll()
Cond.unlock()
```

5.2 Constructing the Index - Addressing the Approximate Search Requirement

The set of leaf nodes of T obtained so far corresponds to *dense nodes*, i.e. parts of the space which intersect some object or are close to an object. However, they do not cover the whole space (in particular, there are zones whose distance to the closest object is at most $Rmax$ which may not be covered by any dense node). In order to address the *approximate* search requirement, we will *fill in*

the gaps by adding a set of *non-dense nodes* as leaves of T.

First, we will construct the set NDN of *non-dense nodes*, by usng Algorithm 2. We need to run the algorithm ($ComputeNonDenseNodesTopDown(root)$), considering that, initially, $NDN = \{\}$ and is a global variable.

Algorithm 2 $ComputeNonDenseNodesTopDown(Q)$

if Q is a leaf in T **then**
  return
else
  for $Q' \in (Children(Q) \setminus Children_T(Q))$ **do**
    $NDN \leftarrow NDN \cup \{Q\}$
  end for
  for $Q' \in Children_T(Q)$ **do**
    $ComputeNonDenseNodesTopDown(Q')$
  end for
end if

The set NDN contains non-dense nodes which, together with the dense nodes, disjointly cover the whole space (the non-dense nodes "filled the gaps" left by the dense nodes). All that is left to do is associate some objects to the non-dense nodes. Unfortunately, we were unable to achieve a constant factor approximation in a top-down manner. Because of this, we will provide a bottom-up algorithm. Initially, $Lobj(QNDN) = \{\}$ and $Lid(QNDN) = \{\}$ for every non-dense node $QNDN$. The algorithm 3 is called once for every node Q corresponding to a dense node (i.e. for every leaf Q of T; note that the non-dense nodes were not added as leaves in T, yet). The second input parameter of the algorithm must be $Rmax$. The algorithm is based on the following property: each non-dense node is the neighbor of an ancestor of a dense node.

Algorithm 3 $ComputeObjsNonDenseNodesBottomUp(Q,\ Rmax)$

$SA = Ancestors(\{Q\}, 1)$
for $A \in SA$ **do**
  for $AN \in (ExtNeighbors(A) \cap NDN)$ **do**
    if $Distance(Cell(Q), Cell(AN)) \leq Rmax$ **then**
      Let O be any object from $Lobj(Q)$ {Or, alternatively, let $id(O)$ be any object id from $Lid(O)$} $Lid(AN) \leftarrow Lid(AN) \cup \{id(O)\}$
    end if
  end for
end for

Note that we compute the lists Lid directly for the non-dense nodes. Also note that the total number of object ids in the lists $Lid(*)$ of the non-dense nodes is bounded by $O(|Leaves_T| \cdot FMaxLevel)$. In Algorithm 3 we had to go all the way up to level 1, because non-dense nodes can be located at any level $lev \leq FMaxLevel$ (not just at levels lev for which $MinLevel \leq lev \leq FMaxLevel$). In order to use only nodes between the levels $MinLevel$ and $FMaxLevel$, at the end of the indexing process, we will call the function $SplitAtLevel(Q, MinLevel)$ for each non-dense node Q with $Level(Q) < MinLevel$ (see Algoritm 2 from [21]).

In Section 9 we present a full proof regarding why the simple algorithm described above provides a constant factor approximation when the query point is located in one of the non-dense nodes.

Here we will continue by improving the algorithms presented so far for non-dense nodes. The problem with the algorithm above is that although we have an upper bound on the total number of object ids in the lists $Lid(*)$ of the non-dense nodes, there is no upper bound on the cardinality of every individual list. The improvement described next will provide a somewhat better theoretical upper bound, but a much better practical upper bound. Let's consider the nodes Q and AN in Algorithm 3. Let $QAN(Q, AN)$ be the lowest ancestor of Q which has some extended neighbor EN such that AN is an ancestor of EN. A less formal explanation is that $QAN(Q, AN)$ is the lowest ancestor of Q which is an extended neighbor of some part of AN. The reason for computing this is that given a fixed node AN, all the objects O from dense nodes Q with the same value of $QAN(Q, AN)$ are "close" to each other and any such object can be selected as a representative (discarding the others). Moreover, if we have two nodes Q_1 and Q_2 such that $QAN(Q_1, AN)$ is an ancestor of $QAN(Q_2, AN)$, then Q_1 can be ignored, as any object whose id belongs to $Lid(Q_2)$ is at most a constant number of times further away from any point of AN than any object with an id from $Lid(Q_1)$. The Algorithm 4 presents the first step of this optimization, in which for every non-dense node AN we compute a list $Ltuple(AN)$ of tuples $(node, ancestor\_node, id(O))$. In the algorithm we will maintain a set $UsedQAN(QND)$ for every non-dense node QND (initially, each of these sets is empty). We need to run the algorithm for every dense node Q.

The second part of this optimization is presented in Algorithm 5. We need to run the algorithm for each non-dense node $AN \in NDN$.

The proof that this optimization still preserves the constant factor approximation is given in Section 9. Note that in the algorithms above, a non-dense node "received" objects only from dense nodes located at the same level or larger levels (i.e. lower in the tree). If required, we may also add objects from dense nodes at smaller levels (although this is not strictly needed for achieving a constant factor approximation). Algorithm 6 shows how this can be implemented (it must be called for each non-dense node Q).

As an observation, we could have computed the list of objects $Lobj(Q)$ for every non-dense node Q (and then, based, on $Lobj(Q)$, we would be able to compute $Lid(Q)$). This would have allowed us to filter useless objects from the list in the end (before computing $Lid(Q)$) and keep only some of the objects from the list. However, it is more time-efficient to compute $Lid(*)$ directly (and, thus, we will not perform any extra filtering, as in the case of the dense nodes).

5.3 Answering a Query

For the query part we use the algorithm 7. We will call $TopDownQuery(root, P)$ in order to obtain the list of candidate object ids. The closest object to the query point P whose identifier belongs to the set of candidate object ids is returned.

Algorithm 4 $ComputeObjsNonDenseNodesBottomUpOpt-1(Q,\ Rmax)$

$SA = Ancestors(\{Q\}, 1)$
QAN=an empty hash table
for $A \in SA$ (in decreasing order of the levels) **do**
  for $AN \in ExtNeighbors(A)$ **do**
    for $AAN \in Ancestors(AN, 1)$ **do**
      if QAN does not contain the key AAN **then**
        Insert in QAN the pair $(key = AAN, value = A)$
      end if
    end for
  end for
  for $AN \in (ExtNeighbors(A) \cap NDN)$ **do**
    if $Distance(Cell(Q), Cell(AN)) \leq Rmax$ **then**
      Let QA be the value associated to the key AN in the hash table QAN (this key always exists).
      if $UsedQAN(AN)$ does not contain QA **then**
        $UsedQAN(AN) \leftarrow UsedQAN(AN) \cup \{QA\}$
        Let O be any object from $Lobj(Q)$ {Or, alternatively, let $id(O)$ be any object id from $Lid(O)$}
        $Ltuple(AN) \leftarrow Ltuple(AN) \cup \{(Q, QA, id(O))\}$
      end if
    end if
  end for
end for

Algorithm 5 $ComputeObjsNonDenseNodesBottomUpOpt-2(AN, Rmax)$

$MarkedAncestors = \{\}$
for $(node, ancestor\_node, id(O)) \in Ltuple(AN)$ **do**
  $MarkedAncestors \leftarrow MarkedAncestors \cup (Ancestors(ancestor\_node, 1) \setminus \{AN\})$
end for
$Ltemp = \{\}$
for $(node, ancestor\_node, id(O)) \in Ltuple(AN)$ **do**
  if $ancestor\_node \notin MarkedAncestors$ **then**
    $Ltemp \leftarrow Ltemp \cup \{((node, id(O))\}$
  end if
end for
for $AN' \in Descendants(AN, max\{Level(AN), MinLevel\})$ **do**
  for $(node, id(O)) \in Ltemp$ **do**
    if $Distance(Cell(node), Cell(AN')) \leq Rmax$ **then**
      $Lid(AN') \leftarrow Lid(AN') \cup \{id(O)\}$
    end if
  end for
end for

Algorithm 6 $ComputeObjectsForNonDenseNodesFromLargerDenseNodes(Q, Rmax)$

$SA = Ancestors(\{Q\}, 1)$
for $A \in SA$ **do**
  for $AN \in (ExtNeighbors(A) \cap Leaves_T)$ **do**
    if $Distance(Cell(Q), Cell(AN)) \leq Rmax$ **then**
      Let O be the closest object to $Cell(Q)$ from $Lobj(AN)$ $Lid(Q) \leftarrow Lid(Q) \cup \{id(O)\}$
    end if
  end for
end for

Algorithm 7 $TopDownQuery(Q, P)$

if Q is a leaf **then**
  return $Lid(Q)$
else
  $result = \{\}$
  for $Q' \in Children_T(Q)$ **do**
    if P is located inside $Cell(Q')$ **then**
      $result \leftarrow result \cup TopDownQuery(Q', P)$
    end if
  end for
  return $result$
end if

6 The Bottom-Up Method

6.1 Constructing the Index - Addressing the Exact Search Requirement

Let's consider a fraction $Frac$. For each object O we first compute the set $SC = ExtInflate(O, MaxLevel, Rmin, Frac)$. Then, for each node $Q \in Ancestors(SC, MinLevel)$ which is also a leaf in T, we call $AddObjectToLeaf(Q, O, Rmin, Frac)$ (Algorithm 4 from [21]). At the end, we use Algorithm 1 in order to filter the objects associated to every tree leaf and, possibly, split the leaves up to the level $FMaxLevel$ (where $FMaxLevel \geq MaxLevel$).

6.2 Constructing the Index - Addressing the Approximate Search Requirement

The top-down method described previously actually makes use of many bottom-up functions, which will not be described here anymore. The only function which still needs to be defined in a bottom-up manner is the computation of the set of non-dense nodes. We will assume that the set NDN of non-dense nodes is initially empty. Then, we call $ComputeNonDenseNodesBottomUp$ for every dense node Q (see Algorithm 8).

After this step, we need to remove from NDN all the nodes whose parent also belongs to NDN. The simplest way of achieving this is to construct the

Algorithm 8 $ComputeNonDenseNodesBottomUp(Q)$

```
SA = Ancestors({Q}, 1)
for A ∈ SA do
  for AN ∈ (ExtNeighbors(A) \ Leaves_T) do
    NDN ← NDN ∪ {AN}
  end for
end for
```

set $NDNParents = \{Q' | Q' = Parent(Q)$ and $Q \in NDN\}$ and then to set $NDN = NDN \setminus NDNParents$.

6.3 Answering a Query

We will compute the set $SC = Ancestors(Cover(P, FMaxLevel), MinLevel)$, consisting of the node Q containing the point P such that $Level(Q) = FMaxLevel$, plus all of its ancestors up to the level $MinLevel$. Only at most one of the nodes from this set is a leaf in T: let this node be QL. Then, we consider all the objects O for which $id(O) \in Lid(QL)$ as candidate answers for the query. The closest candidate object to the query point P is returned.

7 Object Filtering Methods

The function $Filter(Q, Lobjects)$ returns a subset of objects O from $Lobjects$, such that those objects are nearest neighbors to at least one point of $Cell(Q)$. We propose two types of filtering:

- a coarse-grained filtering, based on subdividing (the borders of) $Cell(Q)$ and computing distances from the objects to the subdivisions.
- a fine-grained filtering, based on computing a (restricted) Voronoi diagram of the objects.

7.1 Filtering based on Distances to Nodes' Cells and their Borders

We can subdivide $Cell(Q)$ by using a D-dimensional grid (e.g. by dividing its side in dimension i in s_i equal parts by drawing $s_i - 1$ equally spaced hyperplanes perpendicular on the dimension i). Then, for each part P of the $s_1 \cdot \ldots \cdot s_D$ grid, we compute the following values:

- $Dmin(P) = min\{Distance(O, P) | O \in Lobjects\}$
- $Dmax(P) = Dmin(P) + Diameter(P)$

Since each part P is a hyper-rectangle, the same function which computes the diameter of a node can be used for computing the diameter of P. The result list will contain only those objects $O \in Lobjects$ for which $Distance(O, P) \leq Dmax(P)$ for at least one subdivided part P.

If the distance function returns 0 if an object intersects $Cell(Q)$, then we can reduce the dimensionality by 1. $Cell(Q)$ has $2 \cdot D$ "faces", each corresponding to

an extreme (lowest or highest) coordinate in each dimension. We will subdivide each "face" F using a $(D-1)$-dimensional grid. We can choose the numbers $s_1, \ldots, s_{D-1}$. Then, we renumber the dimensions $dim(1), \ldots, dim(D-1)$ in such a way that the dimension i where the length of F is 0 is excluded. Then, we subdivide the side of F in dimension $dim(i)$ into s_i equal parts, by drawing $s_i - 1$ equally spaced hyper-planes perpendicular on the dimension $dim(i)$. Thus, each face F is subdivided into $s_1 \cdot \ldots \cdot s_{D-1}$ parts. Then, for each part P (of some face F), we compute the same values as before (however, the parts are now restricted to the faces of the node only, and are not located in the node's interior).

The result list will contain only those objects $O \in Lobjects$ for which:

- $Distance(O, P) \leq Dmax(P)$ for at least one subdivided part P **or**
- $Distance(O, Cell(Q)) = 0$ (e.g. if O intersects $Cell(Q)$)

We denote this filtering function $FilterBasedOnDistancesToSubdivisions(Q, Lobjects)$.

7.2 Filtering based on Voronoi Diagrams

In order to maintain strictly only those objects $O \in Lobjects$ which can be nearest neighbors to some parts of $Cell(Q)$, we could compute the Voronoi diagram of the objects (using the distance function $Distance$) restricted to the interior and the borders of $Cell(Q)$. Then, only those objects O whose Voronoi cell intersects $Cell(Q)$ (or, equivalently, their Voronoi cells are non-empty when the diagram is restricted to $Cell(Q)$ only) would be added to the result list.

However, computing the Voronoi diagram of (not necessarily simple) geometric objects in multiple dimensions is a complex task (both time consuming and cumbersome to implement). If the distance function returns 0 for any object intersecting $Cell(Q)$, then we can reduce the dimensionality by 1. We will compute a Voronoi diagram restricted to each face F of $Cell(Q)$. Then, an object O is added to the result list if:

- it intersects $Cell(Q)$ **or**
- its Voronoi cell restricted to at least some face F of $Cell(Q)$ is non-empty.

In the common case of $D = 2$, the "faces" of each node are actually line segments. If the objects are points or line segments (or can be decomposed into a finite number of points and line segments) then the Voronoi diagram restricted to a segment of the border of a node consists of the lower envelope of the distance function from the objects to the segment. Algorithms for computing lower envelopes in the case we mentioned were discussed in [20, 22].

We will denote the function performing Voronoi diagram-based filtering as $Filter-UsingVoronoiDiagram(Q, Lobjects)$. An efficient way of combining the two types of filtering is to define a general filter function $Filter(Q, Lobjects) = FilterUsingVoronoiDiagram(Q, FilterBasedOnDistancesToSubdivisions(Q, Lobjects))$.

8 Distributed Query Processing

The possibility of processing queries in a distributed manner was discussed in [21]. For completeness, we include here the most important ideas. When multiple machines are available for answering a query, we can distribute the index over these machines. From the point of view of a leaf node Q, we may choose to store its list $Lid(Q)$ on a single machine, or have it distributed over the whole range of available machines. When a query is performed, we first compute the set of nodes SC (from the bottom-up solution) which may have the answer to the query. Then, this set is sent to each machine, which, in turn, returns a set of candidate object ids for the query (if it stores part of $Lid(Q)$ for some $Q \in SC$) or doesn't return anything. After computing the union of the sets of object ids, each object is retrieved independently and we compute the distance from the query point to it (we may consider the candidate objects sequentially or in parallel, using multiple threads and/or multiple machines).

9 Proof of Constant Factor Approximation for the Approximate Search Requirement

First, it should be obvious that, whenever the query point is at most a distance $Rmin$ away from the closest object, it will fall inside a dense node (because all the space up to distance $Rmin$ and possibly more is covered by dense nodes). Moreover, the list of object identifiers associated to a dense node will always contain the identifier of the closest object O to any point P inside the node, if $Distance(P, O) \leq Rmin$. That is to say, whenever a query point P lies inside a dense node of a node Q, the identifier of the closest object O will definitely be found within $Lid(Q)$, as long as the distance from the closest object to P does not exceed $Rmin$ (that's because $Q \in Ancestors(ExtInflate(O, MaxLevel, Rmin, Frac), MinLevel)$.

Thus, the *exact* search requirement is fulfilled by our algorithms. We will prove next that, when we fall within the approximate search requirement case, the returned object is at most a constant number of times further away from the query point than the real nearest neighbor. Our proof will need to handle two major cases:

1. Case 1: The query point P lies within a dense node, but its closest object is at a distance larger than $Rmin$
2. Case 2: The query point P lies within a non-dense node.

In each case, the proof will consist of computing a lower bound $Dmin$ of the minimum possible distance of an object to P and an upper bound $Dmax$ of the maximum possible distance of the returned object to P and showing that $\frac{Dmax}{Dmin}$ is upper bounded by a constant value. The proof will also use $Dmin(zone)$ as a lower bound of the minimum possible distance from any part of an object intersecting the zone $zone$ to the point P.

9.1 The query point lies within a dense node

Let's assume that the query point P lies in a dense node Q. We have $Dmin = max\{Rmin, Frac \cdot Diameter(Cell(Q))\}$ and $Dmax = Diameter(Cell(Q)) +$

$max\{\ Rmin, Frac \cdot Diameter(Cell(Q))\} = Diameter(Cell(Q)) + Dmin$. Thus, we have $\frac{Dmax}{Dmin} = 1 + \frac{Diameter(Cell(Q))}{Dmin}$. Since $Dmin \geq Frac \cdot Diameter(Cell(Q))$, we have $\frac{Diameter(Cell(Q))}{Dmin} \leq \frac{1}{Frac}$. Thus, the upper bound in this case is $1 + \frac{1}{Frac}$.

9.2 The query point lies within a non-dense node

First, it should be obvious that the non-dense nodes cover all the space not occupied by the dense nodes, at least up to distance $Rmax$ from the objects. Thus, if the query point lies neither in a dense node, nor in a non-dense node, then its distance from the closest object certainly exceeds $Rmax$.

We will start by proving that our initial, unoptimized algorithm for assigning object identifiers to non-dense nodes provides a constant factor approximation. Then, we will show how the optimized algorithm maintains the constant factor approximation.

9.2.1 The unoptimized version of the algorithm

The key to the proof is the following. Let's consider a query point P contained inside a non-dense node NDC. Then, we will show that, for every dense node DC, $L(NDC)$ contains the identifier of an object which is at most a constant number of times further away from P than any part of the intersection between an object and $Cell(DC)$.

Case 1: $Level(NDC) \geq Level(DC)$ (i.e. $Cell(NDC)$ is larger than $Cell(DC)$)

Subcase 1.1: $Level(NDC) = Level(DC)$ (this means that NDC is an extended neighbor of DC)

Some (random) object identifier $id(O) \in Lid(DC)$ also belongs to $Lid(NDC)$. Since no parts of NDC were indexed as dense nodes (otherwise NDC would not have been indexed as a non-dense node), then $Dmin(Cell(DC)) = max\{Rmin, Frac \cdot Diameter(Cell(DC)), Distance(Cell(DC), Cell(NDC))\}$. Let's consider the identifier $id(O)$ which was "thrown" from $Lid(DC)$ to $Lid(NDC)$. If O intersects $Cell(DC)$, then $Distance(O, Cell(NDC)) \leq Distance(Cell(DC), Cell(NDC)) + Diameter(Cell(DC))$; otherwise the upper bound is equal to $Distance(Cell(DC), Cell(NDC)) + Diameter(Cell(DC)) + max\{Rmin, Frac \cdot Diameter(Cell(DC))\}$. In this case, the ratio between the upper bound and the lower bound is again $O1Frac$.

Subcase 1.2: Let ADC be the lowest ancestor of DC which is an extended neighbor of NDC (thus, we have $Level(ADC) = Level(NDC)$).

Like in the previous subcase, some (random) object identifier $id(O) \in Lid(DC)$ also belongs to $Lid(NDC)$. We have the same lower and upper distance bounds as in the previous subcase.

Subcase 1.3: Let ADC be the ancestor of DC located at the same level as NDC.

Note that ADC and NDC are not extended neighbors (otherwise, we would get subcase 1.2). In this case, no object identifier from $Lid(DC)$ will be "thrown" into $Lid(NDC)$. However, this is not necessary. We have $Dmin(Cell(DC)) = Distance(Cell(DC), Cell(NDC))$. However, $Dmin(Cell(DC)) \geq max\{Rmin, Frac \cdot Diameter(DC), min\{L_i | 1 \leq i \leq D\}\}$ (where L_i is the length in dimension i of $Cell(NDC)$). In this case, there is some extended neighbor DC' of NDC which is an ancestor of a dense node (otherwise not NDC, but one of

its ancestors would have been indexed as a non-dense node). $Lid(NDC)$ contains an identifier $id(O) \in Lid(DC')$. The distance from O to any point within NDC is upper bounded by $K \cdot D \cdot Diameter(Cell(NDC)) + max\{Rmin, Frac \cdot Diameter(Cell(NDC))\}$. Thus, the ratio of the upper and lower bounds is equal to $K \cdot D \cdot \frac{Diameter(Cell(NDC))}{Dmin(DC)} + \frac{max\{Rmin, Frac \cdot Diameter(Cell(NDC))\}}{Dmin(DC)}$. Note that $Diameter(Cell(NDC))$ is a function only of the sizes of $Cell(NDC)$ in every dimension and $Dmin(Cell(DC))$ is larger than the minimunm value of such a size. Since all the nodes have the same aspect ratio, it is easy to prove that all the nodes have the same ratio between their diameter and their smallest size in a dimension (which we will denote by a constant DS). Thus, we obtain again a constant upper bound.

Case 2: $Level(NDC) > Level(DC)$ (i.e. $Cell(NDC)$ is smaller than $Cell(DC)$)

In this case, no identifier from $Lid(DC)$ is added to $Lid(NDC)$. However, none is actually required (as shown below).

We have $Dmin(Cell(DC)) = max\{Rmin, Frac \cdot Diameter(DC), Distance(Cell(DC), Cell(NDC))\}$. There exists an extended neighbor DC' of NDC which is an ancestor of a dense node and, thus, an object identifier $id(O) \in Lid(DC')$ which also belongs to $Lid(NDC)$. In this case, we have $Dmax = K \cdot D \cdot Diameter(NDC) + max\{Rmin, Frac \cdot Diameter(NDC)\}$. Thus, the ratio between the upper and lower bounds is bounded by $1 + \frac{K \cdot D}{Frac}$.

9.2.2 The optimized version of the algorithm

When considering the optimized version of computing the lists $Lid(*)$ of the non-dense nodes, we need to recompute the upper distance bounds in each of the subcases defined above (the new bounds will be higher than in the unoptimized case, but they will still provide a constant factor approximation).

Subcase 1.1: $Dmax = K \cdot D \cdot Diameter(Cell(NDC))$. We still have a constant factor approximation.

Subcase 1.2: We obtain the same lower and upper bounds as in subcase 1.1.

Subcase 1.3: We have the same lower and upper bounds as in the same subcase of the unoptimized version.

Case 2: We have the same lower and upper bounds as in the same subcase of the unoptimized version.

10 Related Work

The fixed-radius nearest neighbor problem has been addressed before in several research papers (e.g. [2, 3, 4, 5]) and many data structures for solving this problem or related problems have been proposed: R-trees [6, 16], kd-trees [7], quad-trees [8], fixed-size node subdivisions [9, 12] and many others [11]. Most of the proposed solutions assume that the index can be constructed in main memory or, at least, can be stored on the disk of a single machine. Thus, the proposed algorithms are sequential in nature (see, for instance, [10]).

More recently, parallel and distributed algorithms for constructing indices over geometrical data have been proposed. In [13], some parts of the construction of a hierarchical index are parallelized using the MapReduce computation model, but other parts were still implemented in a sequential manner. In [14],

a distributed algorithm for constructing octrees ($3D$ quad-trees) was presented. In [15], a MapReduce-based framework which can be used for constructing classification and regression trees in parallel has been proposed. Other attempts for processing spatial data using the MapReduce model for constructing an R-tree index have been made in [17]. A generic MapReduce framework for tree data structures has been proposed in [18].

The work presented in this paper is a natural continuation of the work we presented in [21].

11 Conclusions and Future Work

In this paper we presented novel methods for constructing an index over a set of (arbitrary) geometric objects, which can speed up the exact and approximate computation of answers for fixed-radius nearest neighbors queries. We presented novel, sequential, top-down and bottom-up, algorithms for "out-of-place" indexing and "in-place" searching. The model presented in this paper complements our previous work from [21], where "in-place" indexing and "out-of-place" searching algorithms were discussed. Unlike in [21], we did not provide a parallel or distributed "out-of-place" indexing method (e.g. based on the MapReduce computation model [1]) in this paper. Note, also, that there may also be other intermediate levels between "in-place" indexing plus "out-of-place" searching and "out-of-place" indexing plus "in-place" searching, which might be interesting to explore in order to better understand the trade-offs which they may provide.

12 Acknowledgements

The work presented in this paper was partially funded by the Romanian National Council for Scientific Research (CNCS)-UEFISCDI under research grant PD_240/2010 (AATOMMS - contract no. 33/28.07.2010), from the PN II - RU program, and by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of Labour, Family and Social Protection through the financial agreement POSDRU/89/1.5/S/62557.

References

[1] J. Dean, and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Proceedings of the 6^{th} International Symposium on Operating System Design and Implementation, (2004).

[2] J. L. Bentley, D. F. Stanat, and E. H. Williams, Jr., The Complexity of Finding Fixed-Radius Near Neighbors, Information Processing Letters, vol. 6, no. 6, (1977), pp. 209-212.

[3] W.G. Aref, D. Barbara, S. Johnson, and S. Mehrotra, Efficient Processing of Proximity Queries for Large Databases, Proceedings of the 11^{th} International Conference on Data Engineering, (1995), pp. 147-154.

[4] V. Castelli, Multidimensional Indexing Structures for Content-based Retrieval, IBM Research Report RC 22208 (98723), 2001.

[5] D. Kirkpatrick, Optimal Search in Planar Subdivisions, SIAM Journal of Computing, vol. 12, no. 1, (1983), pp. 28-35.

[6] A. Guttman, R-Trees - A Dynamic Index Structure for Spatial Searching, Proceedings of the ACM SIGMOD International Conference on Management of Data, (1984), pp. 47-57.

[7] J. L. Bentley, Multidimensional Binary Search Trees Used for Associative Searching, Communications of the ACM, vol. 18, no. 9, (1975), pp. 509-517.

[8] G. M. Hunter, and K. Steiglitz, Operations on Images using Quad Trees, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-1, no. 2, (1979), pp. 145-153.

[9] J.-Y. Lai, S.-H. Shu, and Y.-C. Huang, A node Subdivision Strategy for R-Nearest Neighbors Computation, Journal of the Chinese Institute of Engineers, vol. 29, no. 6, (2006), pp. 953-965.

[10] J. Dinis, and M. Mamede, A Sweep Line Algorithm for Nearest Neighbot Queries, Proceedings of the 14^{th} Canadian Conference on Computational Geometry, (2002), pp. 123-127.

[11] E. Chavez, and G. Navarro, A Compact Space Decomposition for Effective Metric Indexing, Pattern Recognition Letters, vol. 26, no. 9, (2005), pp. 1363-1376.

[12] V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami, Geometric Computing and Uniform Grid Technique, Computer-Aided Design, vol. 21, no. 7, (1989), pp. 410-420.

[13] T. Liu, C. Rosenberg, and H. A. Rowley, Clustering Billions of Images with Large Scale Nearest Neighbor Search, IEEE Workshop on Applications of Computer Vision, (2007).

[14] H. Sundar, R. H. Sampath, and G. Biros, Bottom-Up Construction and 2:1 Balance Refinement of Linear Octrees in Parallel, SIAM Journal on Scientific Computing, vol. 30, no. 5, (2008), pp. 2675-2708.

[15] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce, Proceedings of the 35^{th} International Conference on Very Large Data Bases, (2009).

[16] M. Sharifzadeh, and C. Shahabi, VoR-tree: R-trees with Voronoi diagrams for efficient processing of spatial nearest neighbor queries, Proceedings of the VLDB Endowment, vol. 3, no. 1-2, (2010), pp. 1231-1242.

[17] A. Cary, Z. Sun, V. Hristidis, and N. Rishe, Experiences on Processing Spatial Data with MapReduce, Proceedings of the 21^{st} International Conference on Scientific and Statistical Database Management, (2009), pp. 302-319.

[18] A. Sarje, S. Aluru, A MapReduce Style Framework for Computations on Trees, Proceedings of the 39^{th} International Conference on Parallel Processing, (2010), pp. 343-352.

[19] A. Kumar, A Study of Spatial Clustering Techniques, Lecture Notes in Computer Science, vol. 856, (1994), pp. 57-71.

[20] M. I. Andreica, E.-D. Tîrşa: Line-Constrained Geometric Server Placement, Metalurgia International, vol. 16, no. 11, (2011), pp. 106-110.

[21] M. I. Andreica, N. Tăpuş: Sequential and MapReduce-based Algorithms for Constructing an In-Place Multidimensional Quad-Tree Index for Answering Fixed-Radius Nearest Neighbor Queries, Acta Universitatis Apulensis - Mathematics-Informatics, no. 29, (2012), pp. 131-151.

[22] P. K. Agrawal, M. Sharir: Davenport-Schinzel Sequences and Their Geometric Applications, Cambridge University Press, (1995).