HAL
open science

Sequential and MapReduce-based Algorithms for Constructing an In-Place Multidimensional Quad-Tree Index for Answering Fixed-Radius Nearest Neighbor Queries

Mugurel Ionut Andreica, Nicolae Tapus

To cite this version:

Mugurel Ionut Andreica, Nicolae Tapus. 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 (ISSN: 1582-5329), 2012, 29, pp.131-151. hal-00787796

HAL Id: hal-00787796

https://hal.science/hal-00787796v1

Submitted on 13 Feb 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Acta Universitatis Apulensis No. 29/2012
ISSN: 1582-5329 pp.00-00

SEQUENTIAL AND MAPREDUCE-BASED ALGORITHMS FOR CONSTRUCTING AN IN-PLACE MULTIDIMENSIONAL QUAD-TREE INDEX FOR ANSWERING FIXED-RADIUS NEAREST NEIGHBOR QUERIES

Mugurel Ionuţ Andreica, Nicolae Ţăpuş

Abstract. Answering fixed-radius nearest neighbor queries constitutes an important problem in many areas, ranging from geographic systems to similarity searching in object databases (e.g. image and video databases). The usual approach in order to efficiently answer such queries is to construct an index. In this paper we present algorithms for constructing a multidimensional quad-tree index. We start with well-known sequential algorithms and then adapt them to the MapReduce computation model, in order to be able to handle large amounts of data. In all the algorithms the objects are indexed in association with quad-tree cells (or nodes) which they intersect (plus possibly a few other nearby cells). When processing a query, multiple quad-tree cells may be searched in order to find the answer.

2010 *Mathematics Subject Classification*:68P05, 68P10, 68P15, 68P20, 68U35, 68W10, 68W15.

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, restaurant, etc.) located at a distance at most R from him. 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 present algorithms for constructing a quad-tree-based index for answering fixed-radius nearest neighbor queries. We start with well-known top-down sequential algorithms, then we adapt them to a bottom-up approach and, finally, we provide algorithms based on the MapReduce computation model [1], which can handle large amounts of data. 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 our 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 Section 5 we present the sequential top-down and bottom-up indexing algorithms. In Section 6 we present the MapReduce-based indexing algorithm, which is based on the bottom-up sequential algorithm presented in Section 5. In Section 7 we discuss distributed query processing based on the multidimensional quad-tree index constructed by the presented algorithms. In Section 8 we discuss related work and in Section 9 we conclude and present 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 ($1 \leq p \leq +\infty$)). We are interested in efficiently answering the following types of queries: Given a point P in the D-dimensional space and a distance threshold R, return the object O closest to P, located at distance at most R from P.

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 $dist(P, O)$ 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.

3. Choice of the Index Data Structure

Since the number of objects N is large, a brute force approach which, for each query considers every object, is out of the question. Thus, we need to construct an index over the objects as part of a preprocessing stage, which will help us efficiently

answer queries. Many data structures have been proposed in order to solve the fixed-radius nearest neighbor problem, like R-trees, kd-trees, (multidimensional) quad-trees, cell-based space division methods, etc. All of the data structures construct a subdivision of the space, consisting of multiple regions. We will classify the existing data structures in two categories, according to the relation between their regions and the objects:

1. the regions (cells) are chosen from a predefined set SR: we can choose any regions from SR based on the objects' data, but we cannot use regions outside of SR within the data structures; examples of data structures from this category are (multidimensional) region quad-trees and space divisions into grids of cells of fixed sizes.

2. the coordinates of the regions depend on the objects' coordinates; examples of data structures from this category are R-trees and kd-trees.

Since we want our solution to be based on the MapReduce computation model (for which strong infrastructure support already exists), we decided to exclude data structures from the second category. In the MapReduce model the workers do not interact with each other and we feel that a non-negligible amount of interaction would be required if we tried to adapt data structures from the second category to the MapReduce model (because of the need to cluster the objects together). Note that this was a very early decision and, thus, the rejected alternatives were not explored any further. It might be possible to adapt the construction of data structures from the second category to the MapReduce computation model in an efficient manner [17], but we do not make any such attempts in this paper.

At the same time, we do want our index data structure to adapt to the objects' space distribution - thus, solutions based on fixed size grids have also been excluded. We decided that a (multidimensional) quad-tree is the most appropriate solution, because:

1. the set of potential regions 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.

Another requirement was for the data structures to be built in a bottom-up manner. Most of the data structures we mentioned have top-down construction algorithms (based on inserting the objects iteratively), but only some of them have bottom-up construction algorithms.

Our choice of the data structures 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.

The considered indexing strategy was as follows: Every object is indexed only in cells which it intersects (plus, possibly, a few other neaby cells). Then, at query time, we will have to search multiple cells in order to find the answer (we call this the *"in-place" indexing, "out-of-place" searching* method).

4. Main Prerequisites, Assumptions and Terms

In this section we describe the main assumptions and prerequisites which will be considered during the indexing process and at query time. 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 $(ar_1, \ldots, ar_{D-1})$ 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 $L_i/L_D = ar_i$ (for $1 \leq i \leq D-1$). $(ar_1, \ldots, ar_{D-1})$ are constant parameters of the tree. The same holds for another constant $K \geq 2$, which describe 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 $Cell(Q)$. 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: $Level(Q)$. $Level(root) = 1$ and $Level(Q \neq root) = Level(Parent(Q)) + 1$, where $root$ is the root node of the tree and $Parent(Q)$ is the parent node of Q (every node except the root has a parent). 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. $(ar_1, \ldots, ar_{D-1})$ 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 $Parent(Q') = Parent(Q)$; this function can be implemented as:
  - $Sibling(root) = \{\}$
  - $Sibling(Q \neq root) = Children(Parent(Q)) \setminus \{Q\}$
- $ExtNeighbors(Q)$: returns the set consisting of node Q's neighbors and siblings; the function can be implemented as: $ExtNeighbors(Q) = Neighbors(Q) \cup Siblings(Q)$.
- $Descendants(Q, dlevel)$: returns the identifiers of all the descendants of Q located at the level $dlevel$; the function can be implemented as follows:
  - $Descendants(Q, dlevel < Level(Q)) = \{\}$
  - $Descendants(Q, dlevel = Level(Q)) = \{Q\}$
  - $Descendants(Q, dlevel > Level(Q)) =$ $\cup\{Descendants(Q', dlevel) | Q' \in Children(Q)\}$
- $Ancestor(Q, alevel)$: returns the ancestor of the node Q located at the level $alevel$; we can implement this function as follows:
  - $Ancestor(Q, alevel > Level(Q))$ is not defined
  - $Ancestor(Q, Level(Q)) = Q$
  - $Ancestor(Q, alevel < Level(Q)) = Ancestor(Parent(Q), alevel)$
- $Ancestors(Q, alevel)$: returns the set of ancestor of the node Q located at or below the level $alevel$; we can implement this function as follows:
  - $Ancestors(Q, alevel > Level(Q)) = \{\}$
  - $Ancestors(Q, Level(Q)) = \{Q\}$
  - $Ancestors(Q, alevel < Level(Q)) = \{Q\} \cup$ $Ancestors(Parent(Q), alevel)$

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 cells of a multidimensional quad-tree helps us to easily implement all the functions mentioned above. For simplicity, let's assume that, in our quad-tree, we have $K = 2^H$. At each level, the cells of the quad-tree form a D-dimensional grid with the same number of cells in each dimension. The cell located at position $(c(1), \ldots, c(D))$ ($c(i) \geq 0$, $1 \leq i \leq D$) in this grid has an identifier equal to the bit interleaving of the bit representations of $c(1), \ldots, c(D)$ (i.e. we take the first bit of each number, in order, from 1 to D, then the second bit of each number, in the same order, and so on), to which we add an encoding of the cell's level in the tree. The positions of the level $L + 1$ children of a cell located at position $(c(1), \ldots, c(D))$ in the grid at level L will be obtained by appending H bits to each number $c(1), \ldots, c(D)$, thus obtaining $2^{H \cdot D} = K^D$ new positions (from which we can compute the corresponsing identifiers). Computing the position of a cell from its identifier is also very easy, by reversing the encoding algorithm. If K is not a power of 2, then we will use $H = \lceil log_2 K \rceil$. Other encoding schemes with similar properties are also possible [19].

Figure 1 presents the positions of the children of the level 2 node $(01_2, 11_2)$, when $D = 2$ and $K = 4$. The identifier of the node is $(0111_2.2)$, while the identifier of the child $(0101_2, 1101_2)$ is $(01110011_2.3)$.

In our geometric interpretations, we will mainly use the following terms, with the specified meanings:

- *Object*: one of the N objects from the object database
- *Cell*: the region associated to a node from the tree
- *Polyhedron*: this has the usual meaning; note that a *cell* is a polyhedron, as well as the faces and borders of each cell
- *Figure*: a geometric shape, which may be either an *object* or a *polyhedron*

Another set of functions which we require are the following:

- $Distance(F, P)$: returns the distance between a D-dimensional polyhedron P and a figure F (P may be degenerate, in the sense that it may be a D'-dimensional polyhedron, placed in a D-dimensional space, where $D' < D$; think, for instance, of a $2D$ polygon placed in a $3D$-space).
- $Cover(F, clevel)$: returns the set of all the node identifiers Q such that $Level(Q) = clevel$ and the figure F intersects $Cell(Q)$.

$(0100_2, 1100_2)$	$(0100_2, 1101_2)$	$(0100_2, 1110_2)$	$(0100_2, 1111_2)$
$(0101_2, 1100_2)$	$(0101_2, 1101_2)$	$(0101_2, 1110_2)$	$(0101_2, 1111_2)$
$(0110_2, 1100_2)$	$(0110_2, 1101_2)$	$(0110_2, 1110_2)$	$(0110_2, 1111_2)$
$(0111_2, 1100_2)$	$(0111_2, 1101_2)$	$(0111_2, 1110_2)$	$(0111_2, 1111_2)$

Figure 1: Positions of the children of the level 2 node $(01_2, 11_2)$. We consider $D = 2$ and $K = 4$.

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 cell at level *clevel* containing the point P). This can be easily achieved, by computing the position of this cell in the level *clevel* grid of cells (we only need to divide the coordinates of P to the side lengths of the level *clevel* cells in each dimension). Then, we will perform a breadth-first search traversal starting from that cell. We will visit all the level *clevel* cells starting from $Cell(Q)$ which are intersected by the figure F (once a cell is visited, we add it to a queue; when we extract a cell 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 nec-

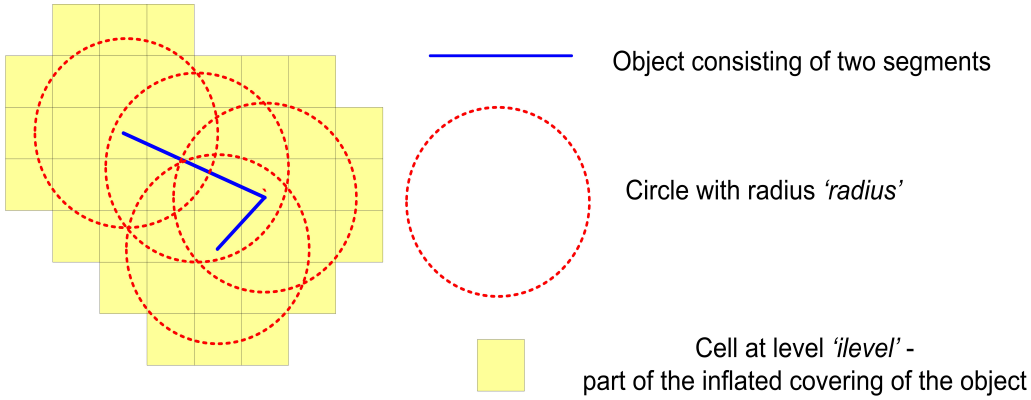

Figure 2: Inflated covering example.

essarily assume the previous equivalence and we will consider that the intersection between an object and a cell is computed geometrically.

Based on the $Distance$ function above we can compute the distance between the cells of two nodes A and B of the tree: $Distance(Cell(A), Cell(B))$, which is 0 if one of the cells is included in the other (i.e. if one of the two nodes is an ancestor of the other one). 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 borders 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 cells are at distance at most $radius$ from the geometric figure F. We will refer to the set of cells of the identifiers from this set as a "covering". See Fig. 2 for an example, where the covered figure (object) F consists of two line segments in $2D$. See also Fig. 3 for an example with two objects.

Actually, we will define a more general function, $ExtInflate(F, ilevel, radius, fraction)$. This function can be implemented as in Algorithm 1. 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 cell $Cell(Q)$ if it is adjacent to a same-level cell $Cell(Q')$ intersecting the figure F and the distance between the figure F and $Cell(Q)$ does not exceed $fraction \cdot Diameter(Cell(Q'))$ (note

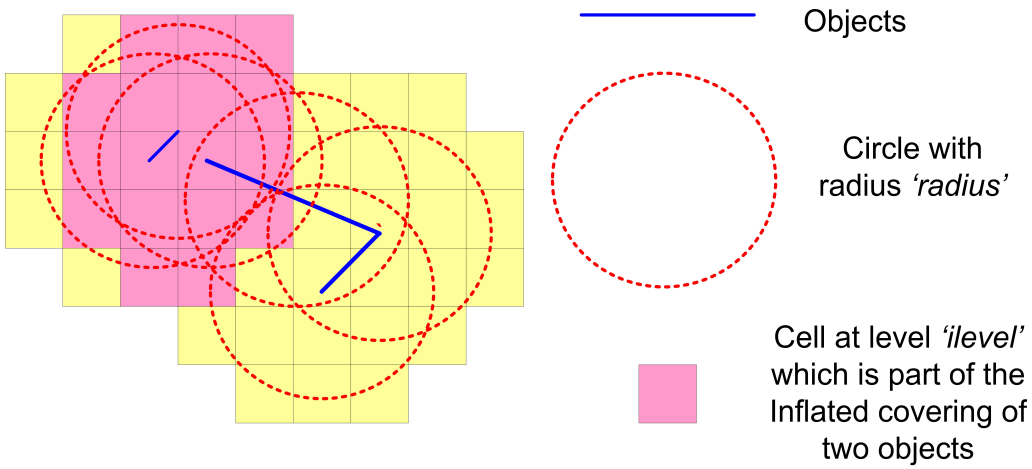

Figure 3: Inflated covering example with two objects.

that since all the cells at the same level are identical, we have $Diameter(Cell(Q)) = Diameter(Cell(Q'))$. The $radius$ parameter can be used, for instance, as a tolerance level. Even if the intersection function decides that a figure F does not intersect a cell $Cell(Q)$, but it is, however, very close to it, we may decide to include $Cell(Q)$ in the covering. The $fraction$ parameter may be used to insert in the covering cells which are "proportionally" close to the figure (where the proportion depends on the cell's size, i.e. its diameter). Normally, $fraction$ would be set to a value very close to 0 (or even 0).

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 quad-tree. 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 Algorithm 2).

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.

Algorithm 1 $ExtInflate(F, ilevel, radius, fraction)$

```
SCover = Cover(F, ilevel)
Queue Qu {Add all the nodes from SCover into the queue Qu}
for C ∈ SCover do
  Qu.enqueue(C)
end for
SNeighbors = {}
while not Qu.isEmpty() do
  C = Qu.dequeue()
  if C ∈ SCover then
    threshold = max{radius, fraction · Diameter(Cell(C))}
  else
    threshold = radius
  end if
  for C' ∈ Neighbors(C) do
    if (C' ∉ SNeighbors) and
    (C' ∉ SCover) and
    (Distance(F, Cell(C')) ≤ threshold) then
      SNeighbors ← SNeighbors ∪ {C'}
      Qu.enqueue(C')
    end if
  end for
end while
return SCover ∪ SNeighbors
```

Algorithm 2 $SplitAtLevel(Q, slevel)$

for $Q' \in Descendants(Q, slevel)$ **do**
  $Lid(Q') = Lid(Q)$ {Optionally, if needed, we may also set $Lobj(Q') = Lobj(Q)$}
  Add Q' as leaf in T
end for
$Lobj(Q) = \{\}$
$Lid(Q) = \{\}$ {Q is now no longer a leaf in T}

5. Sequential "In-Place" Indexing and "Out-of-Place" Searching

Every object O is indexed only in a set of cells which it intersects (plus, possibly, a few other nearby cells). We present both the top-down and the bottom-up methods of implementing this approach.

5.1. The Top-Down Method

5.1.1. Constructing the Index

We start with a sub-tree T consisting only of the root node and we insert the objects one at a time, in an arbitrary order. Algorithm 3 describes a recursive insertion procedure. The initial call is $InsertTopDown(root, O, radius, fraction)$, where O is the current object and $radius$ and $fraction$ have the same meaning as in Algorithm 1.

5.1.2. Answering a Query

A query specifies a point P and a distance threshold R. We define Algorithm 6, in which P is an arbitrary polyhedron and R is a threshold (note that a point is a particular case of polyhedron). The algorithm must be called for the root of the tree and returns a list of object identifiers from the leaves whose cells are located at distance at most R from P. Thus, the list of identifiers will contain the identifiers of all the objects located at distance at most R from the query point P, plus, possibly, a few others. The closest object with the identifier in this list will be the final answer.

5.2. The Bottom-Up Method

The bottom-up method constructs the tree starting from the leaves. The objects are still considered sequentially, one at a time. We will store the identifiers of the leaf nodes of T into a hash table, because we need to be able to efficiently test if a given node is a leaf of T or not. We denote the set of leaves of T by $Leaves_T$. Initially, only the root is a leaf.

Algorithm 3 $InsertTopDown(Q, O, Dmax, Frac)$

```
if Q is a leaf in T then
  AddObjectToLeaf(Q, O, Dmax, Frac)
else
  InsertedChildrenSet = {}
  for Q' ∈ Children(Q) do
    if (Distance(O, Cell(Q') ≤ Dmax) or
    (O intersects Cell(Q')) then
      if Q' ∉ Children_T(Q) then
        Add Q' as a child of Q in T
      end if
      InsertedChildrenSet ← InsertedChildrenSet ∪ {Q'}
      InsertTopDown(Q', O, Dmax, Frac)
    end if
  end for
  for Q' ∈ Children_T(Q) do
    if O intersects Cell(Q') then
      for Q'' ∈ Neighbors(Q') do
        if (Q'' ∈ Children(Q)) and
        (Q'' ∉ InsertedChildrenSet) and
        (Distance(O, Cell(Q'')) ≤ Frac·
        Diameter(Cell(Q'))) then
          if Q'' ∉ Children_T(Q) then
            Add Q'' as a child of Q in T
          end if
          InsertedChildrenSet ← InsertedChildrenSet ∪ {Q''}
          InsertTopDown(Q'', O, Dmax, Frac)
        end if
      end for
    end if
  end for
end if
```

Algorithm 4 $AddObjectToLeaf(Q, O, Dmax, Frac)$

```
Add O to Lobj(Q)
WL(Q) ← aggf(WL(Q), W(O))
if WL(Q) > IWT and Level(Q) < MaxLevel then
  SplitLeaf(Q, Dmax, Frac)
end if
```

Algorithm 5 $SplitLeaf(Q, Dmax, Frac)$

```
if Level(Q) = MaxLevel then
  return
end if
for Q' ∈ Children(Q) do
  Lobj(Q') = {}
  WL(Q') = 0 (we denote by 0 the neutral element for the aggregation function)
  for O ∈ Lobj(Q) do
    if (O intersectes Cell(Q')) or (Distance(O, Cell(Q')) ≤ Dmax) or
    (Distance(O, Cell(Q')) ≤ Frac · Diameter(Cell(Q')) and Q' has a same-
    level neighbor Q'' such that O intersects Cell(Q'')) then
      Lobj(Q') ← Lobj(Q') ∪ {O}
      WL(Q') ← aggf(WL(Q'), W(O))
    end if
  end for
  if |Lobj(Q')| > 0 then
    Add Q' as a child of Q in T
    if WL(Q') > IWT then
      SplitLeaf(Q', Dmax, Frac)
    end if
  end if
end for{Altenatively, we could first compute EI(O) = ExtInflate(O, Level(Q)+
1, Dmax, Frac) for each object O and only add O to the lists Lobj(Q') of those
nodes Q' ∈ (EI(O) ∩ Children(Q))}
Clear Lobj(Q)
```

Algorithm 6 $TopDownQuery(Q, P, R)$

```
if Q is a leaf then
  return Lid(Q)
else
  result = {}
  for Q' ∈ Children_T(Q) do
    if P is located inside Cell(Q') or Distance(Cell(Q'), P) ≤ R then
      result ← result ∪ TopDownQuery(Q', P, R)
    end if
  end for
  return result
end if
```

Algorithm 7 $BottomUpQuery(T, P, R)$

```
SC = Ancestors(Inflate(P, R, MaxLevel), MinLevel)
result = {}
for Q ∈ SC do
  if Q is a leaf in T then
    result ← result ∪ Lid(Q)
  end if
end for
return result
```

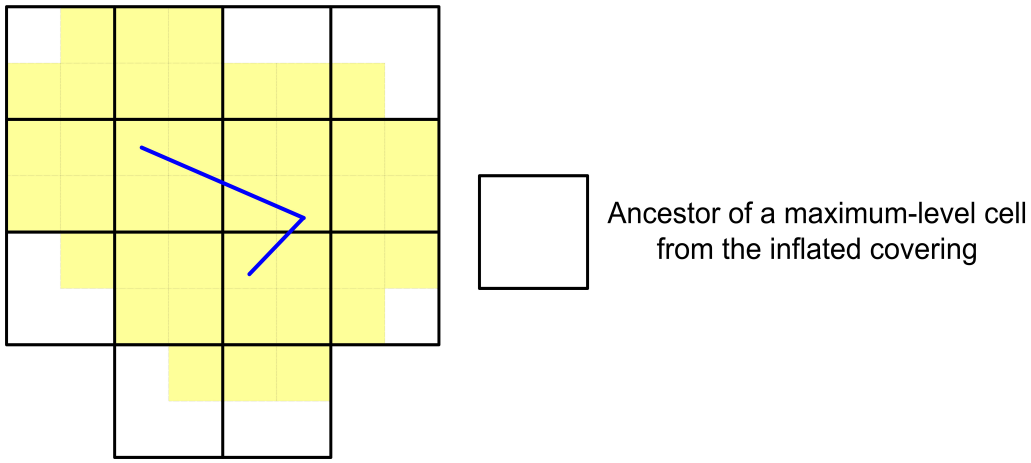

Figure 4: Example of ancestors of an inflated covering.

5.2.1. Constructing the Index

For each object O we first compute the set $ExtInflate(O, MaxLevel, radius, fraction)$. Then, for each of the nodes $Q \in (Ancestors(ExtInflate(O, MaxLevel, radius, fraction), MinLevel) \cap Leaves_T)$, we call the function $AddObjectToLeaf(Q, O, radius, fraction)$. Basically, this part is implemented by computing the mentioned set first (see Fig. 4) and then, for each node Q in the set, look it up in the hash table in order to check if it is also a leaf of T.

5.2.2. Answering a Query

A query specifies a point P and the distance threshold R. Then, similarly to the previous case, we compute the list of object identifiers, this time by calling $BottomUpQuery(root, P, R)$. Then, as before, the closest object whose identifier is in this list is returned as the final answer.

6. "In-Place" Indexing using MapReduce

The MapReduce computation model is based on the existence of two functions, Map and Reduce, which are used for processing input data. Let's assume that the input data consists of M records. The Map function is called independently for processing each record (the record to be processed will be its argument), possibly on separate machines. Each Map function may emit zero, one or more $(key, value)$ pairs. After the Map stage is over (i.e. all the Map functions finished their execution), all the emitted $(key, value)$ pairs are sorted and grouped according to the key (the shuffle stage). Basically, the Shuffle stage computes for each emitted key Key the list of all its values $Lvalues(Key)$ (multiple equal values are preserved in the list). Then, for each pair $(Key, Lvalues(Key))$, the Reduce function is called (the two input arguments to the Reduce function are the key and its list of values). Each Reduce function call may emit zero, one or more *output values* (they can be anything, not necessarily values from the lists of values associated to the keys). The output values of all the Reduce function calls constitute the output of the MapReduce operation.

In this section we describe a chain of 3 MapReduce phases which can construct the "in-place" multidimensional quad-tree index.

6.1. Generation of candidate cells

The input to this MapReduce phase is the set of objects and the output is a set of candidate cells (or nodes). The Map function is presented in Algorithm 8 and the *Reduce* function is presented in Algorithm 9. The $MaxLevel$, $MinLevel$, $radius$ and $fraction$ parameters must be known by the Mappers and Reducers, when they are needed (they can be either constant values or given as side inputs, where needed).

Algorithm 8 $IPI - GenCandCells - Map(O)$

for $Q \in Ancestors(ExtInflate(O, MaxLevel, radius, fraction), MinLevel)$ **do**
  $Emit(key = Q, value = W(O))$
end for

6.2. Filtering candidate cells

In this sub-section we introduce a generic MapReduce phase for filtering cells. The side parameter of the Map function is the boolean value $ShouldOutputCell$, which will take different values, depending on the set of cells being filtered. Algorithm 10 presents the Map function of this phase and Algorithm 11 presents the

Algorithm 9 $IPI - GenCandCells - Reduce(Q, Lvalues)$

$AggWeight = 0$
for $value \in Lvalues$ **do**
  $AggWeight \leftarrow aggf(AggWeight, value)$
end for
if $AggWeight \leq IWT$ or $Level(Q) == MaxLevel$ **then**
  $Emit(Q)$
end if

Reduce function. The input to the Map function is a candidate cell and the output of the *Reduce* function is the set of candidate cells which were kept (the others were filtered out). The side parameter $ShouldOutputCell$ is set to $true$ in this case. The main rule used for filtering the candidate cells is also depicted in Fig. 5.

Algorithm 10 $FilterCandCells - Map(Q)$

if $ShouldOutputCell$ **then**
  Emit $(key = Q, value = CandidateCell)$
end if
for $Q' \in Children(Q)$ **do**
  $Emit(key = Q', value = NonCandidateCell)$
end for

Algorithm 11 $FilterCandCells - Reduce(Q, Lvalues)$

if $Lvalues$ contains only one value V and $V = CandidateCell$ **then**
  $Emit(Q)$
end if

6.3. Computing the lists of object ids for the cells

The input to the Mappers of this phase is the set of N objects and the set of filtered candidate cells. Basically, we perform a MapReduce join between two inputs:

- the set of N objects
- the set of filtered candidate cells

There are many ways of performing the join. The "standard" MapReduce way would be to use two Map functions, one for each input, and let the Reducer perform

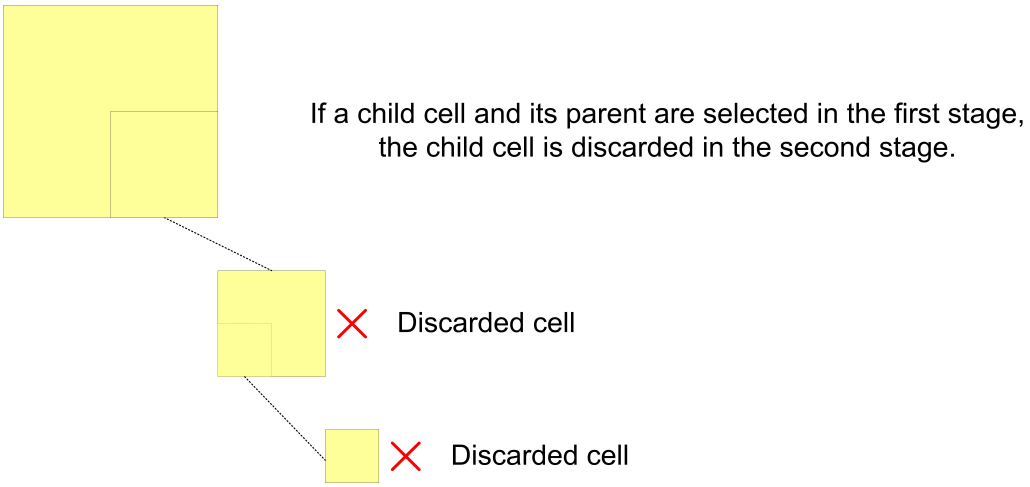

Figure 5: The main rule used for filtering candidate cells.

Algorithm 12 $IPI - ComputeListObjIds - 1 - Map(O)$

for $Q \in (Ancestors(ExtInflate(O, MaxLevel, radius, fraction), MinLevel)$ **do**
  $Emit(key = Q, value = (ObjectCell, id(O)))$
end for

Algorithm 13 $IPI - ComputeListObjIds - 2 - Map(Q)$

$Emit(key = Q, value = (SelectedCell, -))$

Algorithm 14 $IPI - ComputeListObjIds - 1 - Reduce(Q, Lvalues)$

if $Lvalues$ contains at least one pair $(ObjectCell, *)$ and one pair $(SelectedCell, -)$ **then**
  $Lid(Q) = \{id(O) | (ObjectCell, id(O)) \in Lvalues\}$
  $Emit((Q, Lid(Q)))$
end if

the actual join. Algorithms 12, 13 and 14 present the two Map functions and the Reduce function of a standard join for this phase.

When the set of filtered cells is not too large, it may be more efficient for each Mapper to read the whole set of filtered cells in memory and then simply output only those pairs for which the key exists in the set of filtered cells. Let's assume that each Mapper read the set of filtered cells into a variable CC. Algorithms 15 and 16 present the Map and Reduce functions for this case. The inputs for the Mappers are the N objects - each Map function call processes a different object. Note that care must be taken when implementing this approach. If the set of filtered cells is read by each Map function call, then this would be very inefficient. Instead, each Mapper process or thread should read the set of filtered cells in memory. Each Mapper will be used for processing multiple Map function calls - thus, the set of filtered cells will not be read at each Map function call, but rather only once for each Mapper process/thread (since it is most common for each Mapper to run on a separate machine, this means that the set of cells will be read once for each machine used during the Map stage).

Algorithm 15 $IPI - ComputeListObjIds - Map(O)$

for $Q \in (Ancestors(ExtInflate(O, MaxLevel, radius, fraction),\ MinLevel) \cap CC)$ **do**
  $Emit(key = Q, value = id(O))$
end for

Algorithm 16 $IPI - ComputeListObjIds - Reduce(Q, Lids)$

$Lid(Q) = \{id(O) | id(O) \in Lids\}$
$Emit((Q, Lid(Q)))$

7. Distributed Query Processing

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 cells SC 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 returned by each machine, each object is retrieved independently and we

compute the distance from the query point to it. The objects may also be distributed over the same set of machines (or over another set of machines). Unlike the lists of object ids, an object is fully stored on a single machine (it wouldn't make sense to split an object over multiple machines). A good idea might be to store an object O together with an inverted index $IID(O)$, consisting of the set of nodes Q for which $id(O) \in Lid(Q)$, on the same machine. An inverted index can be computed easily [1] from the lists $Lid(*)$ (considering every list to be a *document* and every object id from the list as a *token*). This distributed query processing model is similar to the query processing model used by Google [20].

8. Related Work

The fixed-radius nearest neighbor problem has been addressed before in several research papers (e.g. [2-5]) 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 cell 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].

9. Conclusions and Future Work

In this paper we presented several methods for constructing an in-place multidimensional quad-tree index over a set of (arbitrary) geometric objects, which can speed up the computation of answers for fixed-radius nearest neighbors queries. We started by presenting the top-down and bottom-up sequential implementations and then adapted the bottom-up indexing algorithm to the MapReduce computation model. As future work, we intend to implement the presented MapReduce-based algorithm using the Hadoop framework [21] and assess its performance experimentally. Moreover, we want to explore other types of nearest-neighbor problems and develop new MapReduce-based indexing algorithms for these problems. An "out-of-place"

indexing method (in which objects may also be indexed outside of the cells they intersect) would be particularly interesting, especially if it could be coupled with an "in-place" searching method. We 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 they may provide.

Acknowledgements

The work presented in this paper was partially funded by the Romanian National Council for Scientific Research (CNCS)-UEFISCDI, under research grants ID_1679/2008 (contract no. 736/2009) from the PN II - IDEI program, and 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 6th 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 11th 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 Cell 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 14th 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 35th 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 21st 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 39th 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] L. A. Barroso, J. Dean, U. Holzle, Web Search for a Planet: The Google Cluster Architecture, IEEE Micro, vol. 23, no. 2, (2003), pp. 22-28.

[21] T. White, Hadoop: The Definitive Guide, O'Reilly Media, (2010).

Mugurel Ionuţ Andreica, Nicolae Ţăpuş
Department of Computer Science and Engineering
Politehnica University of Bucharest
Splaiul Independenţei 313, sector 6, Bucharest, Romania
email:{*mugurel.andreica, nicolae.tapus*}*@cs.pub.ro*