MRSLICE: Efficient RkNN Query Processing in SpatialHadoop

Francisco García-García[1], Antonio Corral[1(✉)], Luis Iribarne[1], and Michael Vassilakopoulos[2]

[1] Department of Informatics, University of Almeria, Almeria, Spain
{paco.garcia,acorral,liribarn}@ual.es
[2] Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece
mvasilako@uth.gr

Abstract. Nowadays, with the continuously increasing volume of spatial data, it is difficult to execute spatial queries efficiently in spatial data-intensive applications, because of the limited computational capability and storage resources of centralized environments. Due to that, shared-nothing spatial cloud infrastructures have received increasing attention in the last years. SpatialHadoop is a full-edged MapReduce framework with native support for spatial data. SpatialHadoop also supports spatial indexing on top of Hadoop to perform efficiently spatial queries (e.g., k-Nearest Neighbor search, spatial intersection join, etc.). The Reverse k-Nearest Neighbor (RkNN) problem, i.e., finding all objects in a dataset that have a given query point among their corresponding k-nearest neighbors, has been recently studied very thoroughly. *RkNN* queries are of particular interest in a wide range of applications, such as decision support systems, resource allocation, profile-based marketing, location-based services, etc. In this paper, we present the design and implementation of an *RkNN* query MapReduce algorithm, so-called *MRSLICE*, in SpatialHadoop. We have evaluated the performance of the *MRSLICE* algorithm on SpatialHadoop with big real-world datasets. The experiments have demonstrated the efficiency and scalability of our proposal in comparison with other *RkNNQ* MapReduce algorithms in SpatialHadoop.

Keywords: RNNQ · SpatialHadoop · MapReduce · Spatial data processing

1 Introduction

Large-scale data analysis and processing is currently the core of many scientific research groups and enterprises. Nowadays, with the development of modern mobile applications, the increase of the volume of available spatial data is huge world-wide. Recent developments of big spatial data systems have motivated the emergence of novel technologies for processing large-scale spatial data on clusters of computers in a distributed environment. Parallel and distributed computing

K.-D. Schewe and N. K. Singh (Eds.): MEDI 2019, LNCS 11815, pp. 235–250, 2019.
https://doi.org/10.1007/978-3-030-32065-2_17

using shared-nothing clusters on extreme-scale data is becoming a dominant trend in the context of data processing and analysis. MapReduce [3] is a framework for processing and managing large-scale datasets in a distributed cluster. MapReduce was introduced with the goal of supplying a simple yet powerful parallel and distributed computing paradigm, providing scalability mechanisms.

However, MapReduce has weaknesses related to efficiency when it needs to be applied to spatial data. A main deficiency is the lack of an indexing mechanism that would allow selective access to specific regions of spatial data, which would in turn yield more efficient spatial query processing algorithms. A recent solution to this problem is an extension of Mapreduce, called *SpatialHadoop* [4], which is a mature and robust framework that inherently supports spatial indexing on top of Hadoop. Moreover, the generated spatial indexes enable the design of efficient spatial query processing algorithms that access only part of the data and still return the correct result query. That is, SpatialHadoop is an efficient MapReduce distributed spatial query processing system that supports spatial indexing and allows to work on distributed spatial data without worrying about computation distribution and fault-tolerance.

A Reverse k-Nearest Neighbor query (*RkNNQ*) [8] returns the data objects that have the query object in the set of their k-nearest neighbors. It is the complementary problem to that of finding the k-Nearest Neighbors (kNN) of a query object. The goal of a *RkNNQ* is to identify the *influence* of a query object on the whole dataset; several real examples are shown in [8]. Although the *RkNN* problem is the complement of the kNN problem, the relation between *kNNQ* and *RkNNQ* is not symmetric and the number of the RkNNs of a query object is not known in advance. A naive solution to the *RkNN* problem requires $O(n^2)$ time, since the k-nearest neighbors of all of the n objects in the dataset have to be found [8]. Obviously, more efficient algorithms are required, and thus, the *RkNN* problem has been studied extensively in the past few years [8,9,13]. As shown in a recent experimental study [14], SLICE [15] is the state-of-the art *RkNN* algorithm for two dimensional location data, since it is the best algorithm in terms of CPU cost. Most of the research works in this topic have been devoted to improve the performance of this query by proposing efficient algorithms in centralized environments [14]. But, with the fast increase in the scale of the big input datasets, processing such datasets in parallel and distributed frameworks is becoming a popular practice. For this reason, parallel and distributed algorithms for *RkNNQ* [1,6,7] have been designed and implemented in MapReduce, and a naive approach [5] has been implemented in SpatialHadoop and LocationSpark.

Motivated by the above observations, in this paper, we propose a novel MapReduce version of the SLICE algorithm (the fastest *RkNNQ* algorithm) in SpatialHadoop, called **MRSLICE**. The most important contributions of this paper are the following:

– The design and implementation of a novel *RkNNQ* MapReduce algorithm, called *MRSLICE*, in SpatialHadoop for efficient parallel and distributed *RkNNQ* processing on big real-world spatial datasets.

– The execution of a set of experiments for examining the efficiency and the scalability of *MRSLICE*, against other *RkNNQ* MapReduce algorithms in SpatialHadoop. *MRSLICE* has shown an excellent performance for all considered performance parameters.

This paper is organized as follows. In Sect. 2, we review related work on *RkNNQ* algorithms and provide the motivation of this paper. In Sect. 3, we present preliminary concepts related to *RkNNQ*, SLICE algorithm and SpatialHadoop. In Sect. 4, the *MRSLICE*, in SpatialHadoop is proposed. In Sect. 5, we present the most representative results of the experiments that we have performed, using real-world datasets. Finally, in Sect. 6, we provide the conclusions arising from our work and discuss related future work directions.

2 Related Work and Motivation

Researchers, developers and practitioners worldwide have started to take advantage of the cluster-based systems and shared-nothing cloud infrastructures to support large-scale data processing. There exist several cluster-based systems that support spatial query processing over distributed spatial datasets. One of the most representative is *SpatialHadoop* [4], which is an extension of the Hadoop-MapReduce framework, with native support for spatial data.

RkNNQ processing has been actively investigated in centralized environments, and here we review the most relevant contributions. In [8], *RkNNQ* was first introduced. Processing is based on a pre-computation process (for each data point $p \in \mathbb{P}$ the k-Nearest Neighbor, *kNN(p)*, is pre-computed and its distance is denoted by *kNNdist(p)*) and has three phases: *pruning*, *containment* and *verification*. In the *pruning* phase, for each $p \in \mathbb{P}$ a circle centered at p with radius *kNNdist(p)* is drawn, and the space that cannot contain any *RkNN* is pruned by using the query point q. In the *containment* phase, the objects that lie within the unpruned space are the *RkNN* candidates. Finally, in the *verification* phase, a *range query* is issued for each candidate to check if the query point is one of its *kNN* or not. That is, for any query point q, determine all the circles *(p, kNNdist(p))* that contain q and return their centers p. In [11], the Six-Regions algorithm is presented, and the need for any pre-computation is eliminated by utilizing some interesting properties of *RkNN* retrieval. The authors solve *RkNNQ* by dividing the space around the query point into six equal partitions of 60° each (R_1 to R_6). In each partition R_i, the *k-th* nearest neighbor of the query point defines the pruned area. In [9] the multistep *SFT* algorithm is proposed. It: (1) finds (using an R-tree) the *kNNs* of the query point q, which constitute the initial candidates; (2) eliminates the points that are closer to some other candidate than q; and (3) applies *boolean range queries* on the remaining candidates to determine the actual *RNNs*. In [12], the *TPL* algorithm which uses the property of perpendicular bisectors located between the query point for facilitating pruning the search space is presented. In the containment phase, TPL retrieves the objects that lie in the area not pruned

by any combination of k bisectors. Therefore, TPL has to consider each combination of k bisectors. To overcome the shortcomings of this algorithm, a new method named *FINCH* is proposed in [13]. Instead of using bisectors to prune the objects, the authors use a convex polygon that approximates the unpruned area. *Influence Zone* [2] is a half-space based technique proposed for *RkNNQ*, which uses the concept of *influence zone* to significantly improve the verification phase. Influence zone is the area such that a point p is a *RkNN* of q if and only if p lies inside it. Once influence zone is computed, *RkNNQ* can be answered by locating the points lying inside it. In [15], the *SLICE* algorithm is proposed, which improves the filtering power of Six-Regions approach while utilizing its strength of being a cheaper filtering strategy. Recently, in [14] a comprehensive set of experiments to compare some of the most representative and efficient *RkNNQ* algorithms under various settings is presented and the authors propose an optimized version of TPL (called TPL++) for arbitrary dimensionality *RkNNQs*. SLICE is the state-of-the art *RkNNQ* algorithm, since it is the best for all considered performance parameters in terms of CPU cost.

There is not much work in developing efficient *RkNNQ* algorithms in parallel and distributed environments. The only contributions that have been implemented in MapReduce frameworks are [1,5–7]. In [1], the MRVoronoi algorithm is presented, which adopts the Voronoi diagram partitioning-based approach and applies MapReduce to answer *RNNQ* and other queries. In [6], the Basic MapReduce *RkNNQ* method based on the *inverted grid index* over large scale datasets is investigated. An optimization method, Lazy-MapReduce *RkNNQ* algorithm, that prunes the search space when all data points are discovered, is also proposed. In [7] several improvements of [6] have been presented. For instance, a novel decouple method is proposed to decomposes pruning-verification into independent steps and it can increase opportunities for parallelism. Moreover, new optimizations to minimize the network and disk input/output cost of distributed processing systems have been also investigated. Recently, in [5], parallel and distributed *RkNNQ* algorithms have been proposed for SpatialHadoop and LocationSpark. These parallel and distributed algorithms are based on the multistep *SFT* algorithm [9]. The experimental results demonstrated that LocationSpark is the overall winner for the execution time, due to the efficiency of in-memory processing provided by Spark. However, *MRSFT*, the SpatialHadoop version, shows interesting performance trends due to the nature of the proposed *RkNNQ* MapReduce algorithm, since it consists of a series of MapReduce jobs.

As we have seen above, the SLICE algorithm is the fastest algorithm for *RkNNQ* [14], and there is no MapReduce design and implementation of such an algorithm in parallel and distributed frameworks. Moreover, *RkNNQs* have received significant research attention in the past few years for centralized environments, but not for parallel and distributed data management systems. Motivated by these observations, the efficient design and implementation of *MRSLICE* (MapReduce SLICE version) in SpatialHadoop is the main objective of this research work.

3 Preliminaries and Background

In this section, we first present the basic definitions of the *kNNQ* and *RkNNQ*, followed by a brief introduction of preliminary concepts of SpatialHadoop and, next, we review the most relevant details of the SLICE algorithm for *RkNNQ*.

3.1 The Reverse *k*-Nearest Neighbor Query

We know the *RkNNQ* retrieves the data points which have the query point as one of their respective *kNNs*. We can deduce that the *RkNNQ* is based on the *kNNQ*, and we are going to define it.

Given a set of points $\mathbb{P}$, the *kNNQ* discovers the k points that are the nearest to a given query point q (i.e., it reports only the top-k points of $\mathbb{P}$ from q). It is one of the most important and studied spatial operations. The formal definition of the *kNNQ* for points is the following:

Definition 1. ***k*-Nearest Neighbor query, *kNN***
Let $\mathbb{P} = \{p_1, p_2, \cdots, p_n\}$ be a set of points in E^d (d-dimensional Euclidean space). Then, the result of the k-Nearest Neighbor query, with respect to a query point q in E^d and a number $k \in \mathbb{N}^+$, is an ordered set, $kNN(\mathbb{P}, q, k) \subseteq \mathbb{P}$, which contains the k ($1 \leq k \leq |\mathbb{P}|$) different points of $\mathbb{P}$, with the k smallest distances from q: $kNN(\mathbb{P}, q, k) = \{p_1, p_2, \cdots, p_k\} \subseteq \mathbb{P}$, such that $\forall p \in \mathbb{P} \setminus kNN(\mathbb{P}, q, k)$ we have $dist(p_i, q) \leq dist(p, q),\ 1 \leq i \leq k$.

For *RkNNQ*, given a set of points $\mathbb{P}$ and a query point q, a point p is called the *Reverse k Nearest Neighbor* of q, if q is one of the k closest points of p. A *RkNNQ* issued from point q returns all the points of $\mathbb{P}$ whose k nearest neighbors include q. Note that, this query is also called *Monochromatic RkNNQ* [8]. Formally:

Definition 2. Reverse *k*-Nearest Neighbor query, *RkNN* [13]
Let $\mathbb{P} = \{p_1, p_2, \cdots, p_n\}$ be a set of points in E^d. Then, the result of the Reverse k-Nearest Neighbor query, with respect to a query point q in E^d and a number $k \in \mathbb{N}^+$, is a set, $RkNN(\mathbb{P}, q, k) \subseteq \mathbb{P}$, which contains all the points of $\mathbb{P}$ whose k nearest neighbors include q:
$RkNN(\mathbb{P}, q, k) = \{p_i \in \mathbb{P}, \text{ such that } q \in kNN(\mathbb{P} \cup q, p_i, k)\}$

3.2 SpatialHadoop

SpatialHadoop [4] is a fully fledged MapReduce framework with native support for spatial data. It is an efficient disk-based distributed spatial query processing system. Note that MapReduce [3] is a scalable, flexible and fault-tolerant programming framework for distributed large-scale data analysis. A task to be performed using the MapReduce framework has to be defined as two phases: the *Map* phase, which is specified by a *Map function*, takes input (typically from Hadoop Distributed File System (HDFS) files), possibly performs some computations on this input, and distributes it to worker nodes; and the *Reduce* phase

which processes these results as specified by a *Reduce function*. An important aspect of MapReduce is that both the input and the output of the *Map* step are represented as *key-value pairs*, and that pairs with same key will be processed as one group by the *Reducer*. Additionally, a *Combiner function* can be used to run on the output of *Map* phase and perform some filtering or aggregation to reduce the number of keys passed to the *Reducer*.

SpatialHadoop is a comprehensive extension to Hadoop that injects spatial data awareness in each Hadoop layer, namely, the language, storage, MapReduce, and operations layers. *MapReduce* layer is the query processing layer that runs MapReduce programs, taking into account that SpatialHadoop supports spatially indexed input files. The *Operation* layer enables the efficient implementation of spatial operations, considering the combination of the spatial indexing in the storage layer with the new spatial functionality in the *MapReduce* layer. In general, a spatial query processing in SpatialHadoop consists of four steps [4]: (1) *Preprocessing*, where the dataset is partitioned according to a specific partitioning technique, generating a set of partitions. (2) *Pruning*, when the query is issued, where the master node examines all partitions and prunes (by a *Filter function*) those ones that are guaranteed not to include in any possible result of the spatial query. (3) *Local Spatial Query Processing*, where a local spatial query processing (*Map* function) is performed on each non-pruned partition in parallel on different nodes (machines). And finally, (4) *Global Processing*, where the results are collected from all nodes in the previous step and the final result of the concerned spatial query is computed. A *Combine* function can be applied in order to decrease the volume of data that is sent from the *Map* task. The *Reduce* function can be omitted when the results from the *Map* phase are final.

3.3 SLICE Algorithm

Like most of the *RkNNQ* algorithms, SLICE consists of two phases namely *filtering phase* and *verification phase*. SLICE's filtering phase dominates the total query processing cost [15].

Filtering Phase. SLICE divides the space of a set of points $\mathbb{P}$ around the query point q into multiple equally sized regions based on angle division. The experimental study in [15] demonstrated that the best performance is achieved when the space is divided into 12 equally sized regions. Given a region R and a point $p \in \mathbb{P}$, we can define the half-space that divides them as $H_{p:q}$. The intersection of this half-space with the limits of the region R allows us to obtain the *upper arc* of p w.r.t. R ($r^U_{p:R}$) and the *lower arc* of p w.r.t. R ($r^L_{p:R}$) whose radii meet the condition of $r^U > r^L$. In [15], it is shown that a point p' in the region R can be pruned by the point p if p' lies outside its upper arc, i.e., $dist(p', q) > r^U_{p:R}$. Note that a point $p' \in R$ cannot be pruned by p if p' lies inside its lower arc, i.e., $dist(p', q) < r^L_{p:R}$. The *bounding arc* of a region R, denoted as r^B_R, is the k-th smallest upper arc of that region and it is used to easily prune points or set of points. Note that any point p' that lies in R with $dist(p', q) > r^B_R$ can be

pruned by at least k points. A point p is called *significant* for the region R if it can prune points inside it, i.e., only if $r^L_{p:R} < r^B_R$. Therefore, SLICE maintains a list of significant points for each region that will be used in the *verification phase*. The following lemmas are used in this phase to reduce the search space by pruning non significant points.

Lemma 1. *A point $p \in R$ cannot be a significant point of R if $dist(p, q) > 2r^B_R$.*

Proof. Shown in [15] as Lemma 4.

Lemma 2. *A point $p \notin R$ cannot be a significant point of R if $dist(M, p) > r^B_R$ and $dist(N, p) > r^B_R$ where M and N are the points where the bounding arc of R intersects the boundaries of R.*

Proof. Shown in [15] as Lemma 5.

These lemmas can be easily extended to a complex entity e (i.e., e does not contain any significant point), by comparing $mindist(q, e)$ with the bounding arc of each region that overlaps with e.

Verification Phase. First, SLICE tries to reduce the search space by using the following lemma:

Lemma 3. *A point p prunes every point $p' \in R$ for which $dist(p', q) > r^U_{p:R}$ where $0° < maxAngle(p, R) < 90°$.*

Proof. Shown in [15] as Lemma 1.

To do this, each point $p \in \mathbb{P}$ is checked against several derived pruning rules: (1) if $dist(q, p) > r^B_R$, p is not part of the *RkNNQ* answer; (2) if $dist(q, p)$ is smaller than the k-th lower arc of R, p cannot be pruned; and (3) if once the maximum and minimum angles have been calculated of p w.r.t. q, there is at least one region R with $r^B_R > dist(q, p)$, p can be part of the *RkNNQ* answer. Once the search space has been reduced, each candidate point is verified as a result of *RkNNQ* if at most there are k-1 significant points closest to the query object in the region R in which it is located.

4 *MRSLICE* Algorithm in SpatialHadoop

In this section, we present how *RkNNQ* using *SLICE* can be implemented in SpatialHadoop. In general, our parallel and distributed *MRSLICE* algorithm is based on SLICE algorithm [15] and it consists of three of MapReduce jobs:

- **Phase 1**. The *Filtering phase* of SLICE is performed on the partition in which the query object is located.
- **Phase 1.B (optional)**. The filtering process is continued on those partitions that are still part of the search space.

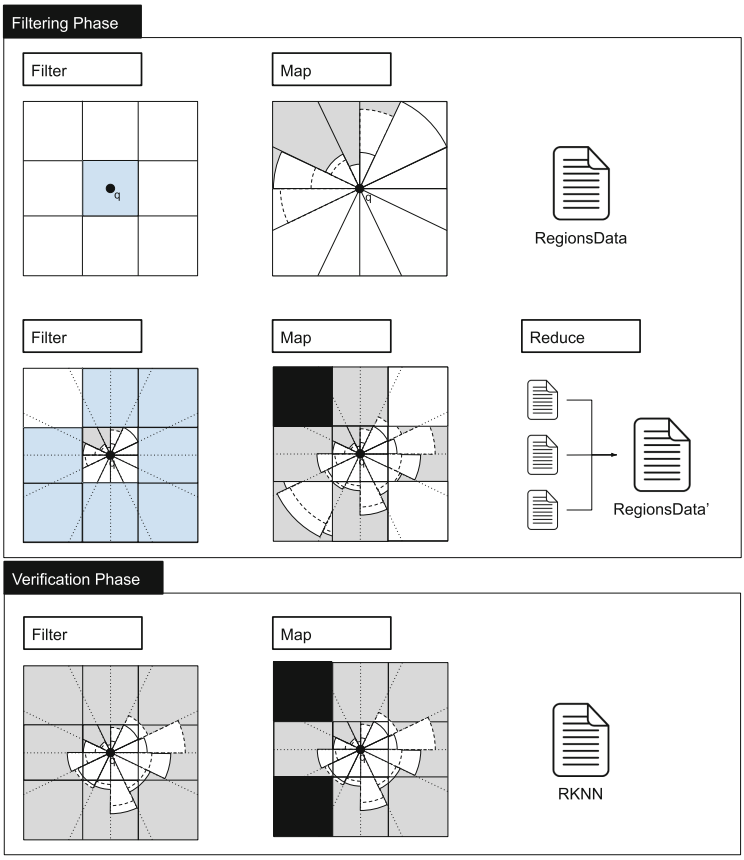

Fig. 1. Overview of *MRSLICE* in SpatialHadoop.

– **Phase 2**. The *Verification phase* is carried out with those partitions that have not been pruned as a result of applying Phases 1 and 1.B.

From Fig. 1, and assuming that $\mathbb{P}$ is the set of points to be processed and q is the query point, the basic idea is to have $\mathbb{P}$ partitioned by some method (e.g., grid) into n blocks or partitions of points ($\mathcal{P}^{\mathbb{P}}$ denotes the set of partitions from $\mathbb{P}$). The *Filtering phase* consists of two MapReduce jobs, being optional the second one, since in the case of all significant points were found by the first job, the execution of the second job is not necessary. Finally, the *Verification phase* is a MapReduce job that will check if the non pruned points are part of the *RkNNQ* answer.

4.1 *MRSLICE* Filtering Algorithm

In the first job (Algorithm 1), the *Filter* function selects the partition of $\mathbb{P}$ in which q is found. Then, in the Map phase, the *Filtering phase* is applied as

Algorithm 1. *MRSLICE* Filtering - Phase 1

```
1: function FILTER(P^P: set of partitions from P, q: query point)
2:     return FindPartition(P^P,q)
3: end function

4: function MAP(MBR: Minimum Bounding Rectangle of P, r: root of R-tree of actual partition,
   q: query point, k: number of points, t: number of equally sized regions)
5:     RegionsData.regions ← DivideSpace(MBR,q,t)
6:     RegionsData ← SliceFiltering(r,q,k,RegionsData)
7:     return RegionsData
8: end function

9: function SliceFiltering(r: root of R-tree, q: query point, k: number of points, RegionsData:
   SLICE Regions Data)
10:    Insert(Heap,null,r)
11:    while Heap is not empty do
12:        entry ← Pop(Heap)
13:        if !facilityPruned(entry,q,k,RegionsData) then
14:            if isLeaf(entry) then
15:                pruneSpace(entry,q,k,RegionsData)
16:            else
17:                for all child ∈ entry.children do
18:                    key ← mindist(q,child)
19:                    Insert(Heap,key,child)
20:                end for
21:            end if
22:        end if
23:    end while
24:    for all region ∈ RegionsData.regions do
25:        region.boundingArc ← FindkUpperArc(region)
26:    end for
27:    RegionsData.minLowerArc ← ComputeMinLowerArc(RegionsData.regions)
28:    return RegionsData
29: end function
```

Algorithm 2. *MRSLICE* Filtering - Phase 1.B

```
1: function FILTER(P^P: set of partitions from P, q: query point, RegionsData: SLICE Regions
   Data)
2:     for all p ∈ P^P do
3:         if !facilityPruned(p,q,RegionsData) then
4:             Insert(Result, p)
5:         end if
6:     end for
7:     return Result
8: end function

9: function MAP(r: root of R-tree of actual partition, q: query point, k: number of points,
   RegionsData: SLICE Partition Data)
10:    RegionsData′ ← SliceFiltering(r,q,k,RegionsData)
11:    return RegionsData′
12: end function

13: function REDUCE(RegionsDataArray: Array of SLICE Partition Data)
14:    RegionsData′ ← RegionsDataArray[0]
15:    for all RegionsData ∈ RegionsDataArray do
16:        RegionsData′.P ← Merge(RegionsData.regions, RegionsData′.P)
17:    end for
18:    for all partition ∈ RegionsData′.P do
19:        partition.kUpperArc ← FindkUpperArc(partition)
20:    end for
21:    RegionsData′.minLower ← ComputeMinLower(RegionsData′.P)
22:    return RegionsData′
23: end function
```

described in SLICE, that is, $\mathbb{P}$ is divided into t regions of equal space and the list of k smallest upper arcs is obtained for each R_i region along with its $r^B_{R_i}$ and its list of significant points, that will be returned as *RegionsData* for further use. To accelerate the *Filtering phase*, an R-tree index is used per partition and a *heap* is utilised to store the nodes based on their minimum distance to q. As the R-tree nodes are traversed, the *facilityPruned* function from [15] is used (Algorithm 1 line 13), which prunes the nodes which with the current *RegionsData* do not contain significant points. In the case of leaf nodes, the points are processed by the *pruneSpace* function from [15] (Algorithm 1 line 15), which is responsible for updating the *RegionsData* information. Finally the k-th lower arc is calculated for using in the next phase.

The second job (Algorithm 2) runs only if the function *Filter* returns some partition, that is, the *facilityPruned* [15] function is executed on each of the partitions by comparing its minimum distance to q with the bounding arc of each region R_i with which it overlaps. Note that the upper left partition of $\mathbb{P}$ in Fig. 1 is in the shaded area, and therefore can be pruned. However, the other partitions can contain significant points, and the *Filtering phase* must be applied to them during the *Map phase*. The result of each of the partitions will be merged on the *Reduce phase* to obtain the k-th upper arcs, bounding arcs and final significant points (*RegionsData'*).

Theorem 1 (Completeness). *MRSLICE Filtering Algorithm returns all the significant points.*

Proof. It suffices to show that MRSLICE Filtering does not discard significant points. A point p is discarded by MRSLICE Filtering only if it is pruned by the *facilityPruned* function by either applying Lemma 1 or 2. In any of these cases, it was shown in [15] that any point that is not inside the area defined by these lemmas is not a significant point. Points that are discarded can be split in different categories:

Phase 1. Points are pruned in this phase like in the non distributed SLICE version using Algorithm 1.

Phase 1.B - Partition granularity. Using the *FILTER* function in Algorithm 2, partitions that do not contain any significant point are pruned by applying both Lemmas 1 and 2 to the partition as a complex entity.

Phase 1.B - Point granularity. Points are discarded in the Map Phase in the same way that in *Phase 1* only on non pruned partitions.

Phase 1.B - Merging RegionsData. Finally when merging RegionsData in the Reduce Phase, both Lemmas 1 and 2 are again used to discard non significant points.

4.2 *MRSLICE* Verification Algorithm

Finally, in the *Verification phase*, a MapReduce job (Algorithm 3) is executed on the partitions that are not pruned by the *Filter* function when applying the pruning rules described in the Subsect. 3.3 in the *userPruned* function (Algorithm 3

Algorithm 3. *MRSLICE* Verification - Phase 2

```
1: function FILTER(P^P: set of partitions from P, q: query point, RegionsData: SLICE partition data)
2:     for all p ∈ P^P do
3:         if !USERPRUNED(p,q,RegionsData) then
4:             INSERT(Result, p)
5:         end if
6:     end for
7:     return Result
8: end function

9: function MAP(r: root of R-tree of actual partition, q: query point, k: number of points, RegionsData: SLICE Partition Data)
10:    INSERT(Stack,r)
11:    while Stack is not empty do
12:        entry ← POP(Stack)
13:        if !USERPRUNED(entry,q,RegionsData) then
14:            if ISLEAF(entry) then
15:                if ISRKNN(entry,q,k,RegionsData) then
16:                    OUTPUT(entry)
17:                end if
18:            else
19:                for all child ∈ entry.children do
20:                    INSERT(Stack,child)
21:                end for
22:            end if
23:        end if
24:    end while
25: end function

26: function ISRKNN(entry: candidate point, q: query point, k: number of points, RegionsData: SLICE Partition Data)
27:    region ← FINDREGION(entry,q,RegionsData)
28:    counter ← 0
29:    for all p ∈ region do
30:        if dist(entry,q) ≤ r^L_{p:R} then
31:            return true
32:        end if
33:        if dist(entry,p) < dist(entry,q) then
34:            counter ← counter + 1
35:            if counter ≥ k then
36:                return false
37:            end if
38:        end if
39:    end for
40:    return true
41: end function
```

line 3). That is, the algorithm is executed on those partitions that contain some white area. In the *Map phase*, the R-tree, that indexes each partition, is traversed with the help of a *stack* data structure and the search space is reduced by using the *userPruned* function again. Furthermore, the pruning rules are applied again to the points that are in the leaf nodes and, finally, they are verified if they are part of the final *RkNNQ* answer. The *isRkNN* function (Algorithm 3 line 15) verifies a candidate point p as part of the answer if there are at most k-1 significant points closer to p than q in the region R_i in which it is located.

Theorem 2 (Correctness). *MRSLICE Verification Algorithm returns the correct RkNNQ set.*

Proof. It suffices to show that *MRSLICE* Verification does not (a) discard *RkNNQ* points, and (b) return non *RkNNQ* points. First, the *MRSLICE* Verification Algorithm only prunes away those points or/and entries by using the pruning rules derived from Lemma 3, by using the information identified by the *MRSLICE* Filtering Algorithm, which guarantees no false negatives. Second, every non pruned point is verified by the *isRkNN* function, which ensures no false positives. We prove that these points are guaranteed to be *RkNNQ* points by contradiction. Assume a point p returned by *MRSLICE* Algorithm is not a *RkNNQ* point. Then, there exist k significant points closer to p than q, and p is also returned as part of the *RkNNQ* answer. But then p could not be in the *RkNNQ* answer, since it would have been evicted in line 35 of the *isRkNN* function in Algorithm 3.

5 Experimentation

In this section, we present the most representative results of our experimental evaluation. We have used real-world 2d point datasets to test our *RkNNQ* algorithms, that is, our previous *MRSFT* based algorithm [5] and the new *MRSLICE* algorithm in SpatialHadoop. We have used datasets from OpenStreetMap[1]: *LAKES* (L) which contains 8.4M records (8.6 GB) of boundaries of water areas (polygons), *PARKS* (P) which contains 10M records (9.3 GB) of boundaries of parks or green areas (polygons), *ROADS* (R) which contains 72M records (24 GB) of roads and streets around the world (line-strings), *BUILDINGS* (B) which contains 115M records (26 GB) of boundaries of all buildings (polygons), and *ROAD_NETWORKS* (RN) which contains 717M records (137 GB) of road network represented as individual road segments (line-strings) [4]. To create sets of points from these five spatial datasets, we have transformed the MBRs of line-strings into points by taking the center of each MBR. In particular, we have considered the centroid of each polygon to generate individual points for each kind of spatial object. Furthermore, all datasets have been previously partitioned by SpatialHadoop using the STR partitioning technique with a local R-tree index per partition. The main performance measure that we have used in our experiments has been the total execution time (i.e., total response time). In order to get a representative execution time, a random sample of 100 points from the smallest dataset (*LAKES*) has been obtained and the average of the execution time of the *RkNNQ* of these points has been calculated, since this query depends a lot on the location of the query point with respect to the dataset.

All experiments were conducted on a cluster of 12 nodes on an OpenStack environment. Each node has 4 vCPU with 8 GB of main memory running Linux operating systems and Hadoop 2.7.1.2.3. Each node has a capacity of 3 vCores for MapReduce2/YARN use. Finally, we used the latest code available in the repositories of SpatialHadoop[2].

[1] Available at http://spatialhadoop.cs.umn.edu/datasets.html.
[2] Available at https://github.com/aseldawy/spatialhadoop2.

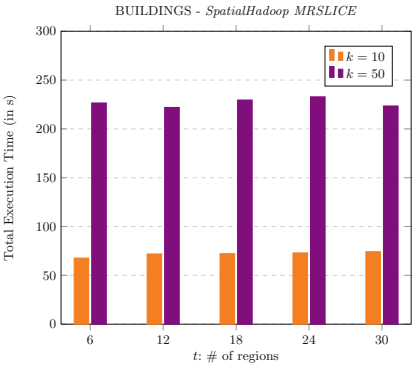

Fig. 2. *MRSLICE* execution times considering different t values.

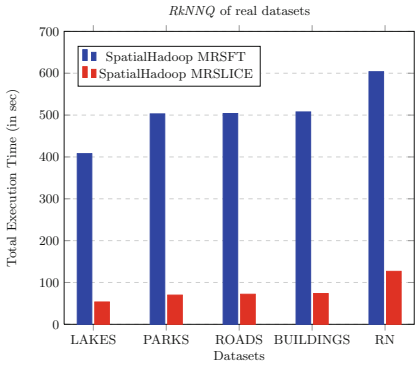

Fig. 3. *RkNNQ* execution times considering different datasets.

The first experiment aims to test the best t value (number of regions) for *MRSLICE*, using the *BUILDINGS* dataset and the k values of 10 and 50. In Fig. 2 we can see that there is a little difference in the results obtained when the t value is varied, especially for $k = 10$, being greater differences when a larger k value is used (i.e., $k = 50$). On one hand, for $k = 10$, smaller values of t get faster times (e.g., $t = 6$ has an execution time of 67 s which is 4 s faster than $t = 12$). On the other hand, for $k = 50$, $t = 12$ gets the smallest execution time (221 s) and for $t < 12$ and $t > 12$, the execution time increases. Although there are no large differences, the value of t that shows better performance for both k values is $t = 12$, reaching the same conclusion as in [15] but now in a distributed environment (from now on, we will use $t = 12$ in all our experiments).

Our second experiment studies the scalability of the *RkNNQ* MapReduce algorithms (*MRSLICE* and *MRSFT* [5]), varying the dataset sizes. As shown in Fig. 3 for the *RkNNQ* of real datasets (*LAKES*, *PARKS*, *ROADS*, *BUILDINGS* and *RN*) and a fixed $k = 10$. The execution times of *MRSLICE* are much lower than those from *MRSFT* (e.g., it is 477 s faster for the largest dataset *RN*) thanks to how the search space is reduced and the limited number of MapReduce jobs. Note that for *MRSFT* at least $k * 20 + 1$ jobs are executed while for the case of *MRSLICE*, 3 jobs are launched at most. In both algorithms, the execution times do not increase too much, showing quite stable performance, mainly for *MRSLICE*. This is due to the indexing mechanisms provided by SpatialHadoop that allow fast access to only the necessary partitions for the query processing. Furthermore, this behavior shows that the number of candidates for *MRSLICE* is almost constant (the expected number of candidates is less than $3.1 * k$ as stated in [15]), only showing a visible increment in the execution time for the *RN* dataset, due to the increase in the density of partitions and its distribution causes the need to execute the optional job (phase 1.b) of the *Filtering phase*.

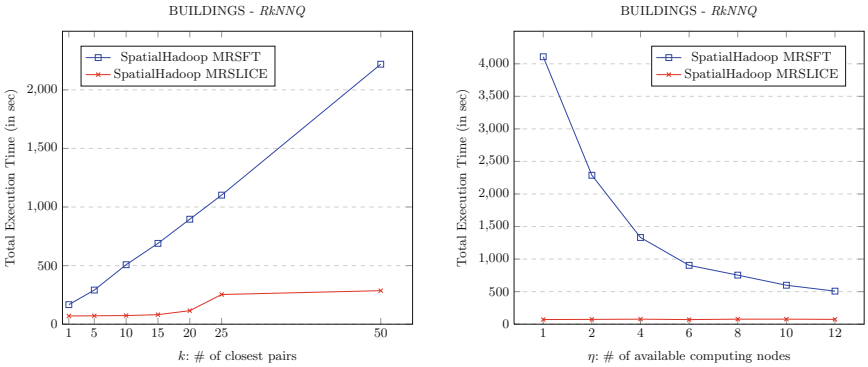

Fig. 4. *RkNNQ* cost (execution time) vs. K values (left). Query cost with respect to the number of computing nodes η (right).

The third experiment aims to measure the effect of increasing the k value for the dataset ($BUILDINGS$). The left chart of Fig. 4 shows that the total execution time grows as the value of k increases, especially for *MRSFT*. This is because as the value of k increases, the number of candidates $k * 20$ also grows and for each of them a MapReduce job is executed. On the other hand, *MRSLICE* limits the number of MapReduce jobs to 3, obtaining a much smaller increment and more stable results since the disk accesses are reduced significantly by traversing the index of the dataset a reduced number of times. Note that the small increment in the execution times when $k = 25$, mainly due to the fact that when reaching a certain k value, the result of the first job of the *Filtering phase* is not definitive and it has been necessary to execute the optional job (phase 1.b), in this case the number of involved partitions in the query increases as well. Finally, the execution time for $k = 50$ increases slightly.

The fourth experiment studies the speedup of the *RkNNQ* MapReduce algorithms, varying the number of computing nodes (η). The right chart of Fig. 4 shows the impact of different number of computing nodes on the performance of *RkNNQ* MapReduce algorithms, for $BUILDINGS$ with a fixed value of $k = 10$. From this chart, we can deduce that for *MRSFT*, better performance would be obtained if more computing nodes are added. *MRSLICE* is still outperforming *MRSFT* and it is not affected despite reducing the number of available computing nodes. This is because *MRSLICE* is an algorithm in which both the number of partitions involved in obtaining the result of the query and the number of MapReduce jobs are minimized. That is, depending on the location of the query point q and the k value, the number of partitions is usually one, and varying the number of computing nodes does not affect the execution time. However, the use of the computing resources of the cluster is quite small, which allows us the execution of several *RkNNQs* in parallel, taking advantage of the distribution of the dataset into the cluster nodes. On the other hand, *MRSFT* executes different *kNNQs* in parallel, using all computing nodes completely for large k values.

By analyzing the previous experimental results, we can extract several conclusions that are shown below:

- We have experimentally demonstrated the *efficiency* (in terms of total execution time) and the *scalability* (in terms of k values, sizes of datasets and number of computing nodes (η)) of the proposed parallel and distributed *MRSLICE* algorithm for *RkNNQ* and we have compared it with the *MRSFT* algorithm in SpatialHadoop.
- As stated in [15], the value of t (the number of equally sized regions in which the dataset is divided) that shows the best performance is 12.
- *MRSLICE* outperforms *MRSFT* several orders of magnitude (around five times faster), thanks to its pruning capabilities and the limited number of MapReduce jobs.
- The larger the k values, the greater the number of candidates to be verified, but for *MRSLICE* the number of jobs and partitions involved are quite restricted and the total execution time increases less than for *MRSFT*.
- The use of computing nodes by *MRSLICE* is small, allowing the execution of several queries in parallel, unlike *MRSFT* that can leave the cluster busy.

6 Conclusions and Future Work

In this paper, we have proposed a novel *RkNNQ* MapReduce algorithm, called *MRSLICE*, in SpatialHadoop, to perform efficient parallel and distributed *RkNNQ* on big spatial real-world datasets. We have also compared this algorithm with our previous *MRSFT* algorithm [5] in order to test its performance. The execution of a set of experiments has demonstrated that *MRSLICE* is the clear winner for the execution time, due to the efficiency of its pruning rules and the reduced number of MapReduce jobs. Furthermore, *MRSLICE* shows interesting performance trends due to the low requirements of computing nodes that allows the execution of multiple *RkNNQs* on large spatial datasets. Our current *MRSLICE* algorithm in SpatialHadoop is an example for the study of regions-based pruning on parallel and distributed environments. Therefore, future work might include the adaptation of half-space pruning algorithms [15] to this kind of environments so as to compare them. Other future work might cover studying other types of *RkNNQs* like the Bichromatic *RkNNQ* [15]. Finally, we are planning to improve the query cost of *MRSLICE* by using the *guardian set* of a rectangular region [10], that improves the original SLICE algorithm.

Acknowledgments. Research of all authors is supported by the MINECO research project [TIN2017-83964-R].

References

1. Akdogan, A., Demiryurek, U., Kashani, F.B., Shahabi, C.: Voronoi-based geospatial query processing with MapReduce. In: CloudCom Conference, pp. 9–16 (2010)

2. Cheema, M.A., Lin, X., Zhang, W., Zhang, Y.: Influence zone: efficiently processing reverse k nearest neighbors queries. In: ICDE Conference, pp. 577–588 (2011)
3. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI Conference, pp. 137–150 (2004)
4. Eldawy, A., Mokbel, M.F.: Spatialhadoop: a mapreduce framework for spatial data. In: ICDE Conference, pp. 1352–1363 (2015)
5. García-García, F., Corral, A., Iribarne, L., Vassilakopoulos, M.: RkNN query processing in distributed spatial infrastructures: a performance study. In: Ouhammou, Y., Ivanovic, M., Abelló, A., Bellatreche, L. (eds.) MEDI 2017. LNCS, vol. 10563, pp. 200–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66854-3_15
6. Ji, C., Hu, H., Xu, Y., Li, Y., Qu, W.: Efficient multi-dimensional spatial RkNN query processing with MapReduce. In: ChinaGrid Conference, pp. 63–68 (2013)
7. Ji, C., Qu, W., Li, Z., Xu, Y., Li, Y., Wu, J.: Scalable multi-dimensional RNN query processing. Concurrency Comput.: Pract. Experience **27**(16), 4156–4171 (2015)
8. Korn, F., Muthukrishnan, S.: Influence sets based on reverse nearest neighbor queries. In: SIGMOD Conference, pp. 201–212 (2000)
9. Singh, A., Ferhatosmanoglu, H., Tosun, A.S.: High dimensional reverse nearest neighbor queries. In: CIKM Conference, pp. 91–98 (2003)
10. Song, W., Qin, J., Wang, W., Cheema, M.A.: Pre-computed region guardian sets based reverse kNN queries. In: DASFAA Conference, pp. 98–112 (2016)
11. Stanoi, I., Agrawal, D., El Abbadi, A.: Reverse nearest neighbor queries for dynamic databases. In: ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, pp. 44–53 (2000)
12. Tao, Y., Papadias, D., Lian, X.: Reverse kNN search in arbitrary dimensionality. In: VLDB Conference, pp. 744–755 (2004)
13. Wu, W., Yang, F., Chan, C.Y., Tan, K.: FINCH: evaluating reverse k-nearest-neighbor queries on location data. PVLDB **1**(1), 1056–1067 (2008)
14. Yang, S., Cheema, M.A., Lin, X., Wang, W.: Reverse K nearest neighbors query processing: Experiments and analysis. PVLDB **8**(5), 605–616 (2015)
15. Yang, S., Cheema, M.A., Lin, X., Zhang, Y.: SLICE: reviving regions-based pruning for reverse k nearest neighbors queries. In: ICDE Conference, pp. 760–771 (2014)