RkNN Query Processing in Distributed Spatial Infrastructures: A Performance Study

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. 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 received increasing attention in the past years. *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. With the current increasing volume of spatial data, it is difficult to perform *RkNN* queries efficiently in spatial data-intensive applications, because of the limited computational capability and storage resources. In this paper, we investigate how to design and implement distributed *RkNN* query algorithms using shared-nothing spatial cloud infrastructures as SpatialHadoop and LocationSpark. SpatialHadoop is a framework that inherently supports spatial indexing on top of Hadoop to perform efficiently spatial queries. LocationSpark is a recent spatial data processing system built on top of Spark. We have evaluated the performance of the distributed *RkNN* query algorithms on both SpatialHadoop and LocationSpark with big real-world datasets. The experiments have demonstrated the efficiency and scalability of our proposal in both distributed spatial data management systems, showing the performance advantages of LocationSpark.

Keywords: Spatial data processing · RNNQ · SpatialHadoop · LocationSpark

1 Introduction

In the age of smart cities and mobile environments, there is a huge increase of the volume of available spatial data (e.g. location, routing, navigation, etc.) world-wide. Recent developments of spatial big data systems have motivated the emergence of novel technologies for processing large-scale spatial data on clusters

of computers in a distributed environment. These Distributed Data Management Systems (DDMSs) can be classified in disk-based [9] and in-memory-based [18]. The disk-based Distributed Spatial Data Management Systems (*DSDMSs*) are characterized by being Hadoop-based systems and the most representative ones are SpatialHadoop [4] and Hadoop-GIS [1]. On the other hand, the in-memory (*DSDMSs*) are characterized by being Spark-based systems and the most remarkable ones are Simba [15] and LocationSpark [12]. These systems allows users to work on distributed in-disk or in-memory spatial data without worrying about computation distribution and fault-tolerance.

A Reverse *k*-Nearest Neighbor (*RkNN*) query [8,11] 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 *RkNN* query (*RkNNQ*) is to identify the *influence* of a query object on the whole dataset, and several real examples are mentioned in [8]. Although the *RkNN* problem is the complement of the *k*-Nearest Neighbor problem, the relationship between *kNN* and *RkNN* is not symmetric and the number of the reverse *k*-nearest neighbors of a query object is not known in advance. A naive solution of 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 years for centralized environments [16]. But, with the fast increase in the scale of the big input datasets, parallel and distributed algorithms for *RkNNQ* in MapReduce [2] have been designed and implemented [6,7], and there are no *RkNNQ* implementations in Spark [17].

The most important contributions of this paper are the following:

– The design and implementation of novel algorithms in SpatialHadoop and LocationSpark to perform efficient parallel and distributed *RkNNQ* on big real-world spatial datasets.
– The execution of a set of experiments for examining the efficiency and the scalability of the new parallel and distributed *RkNNQ* algorithms. And the comparison of the performance of the two DSDMSs (SpatialHadoop and LocationSpark).

This paper is organized as follows. In Sect. 2, we present preliminary concepts related to *RkNNQ*. In Sect. 3, the parallel and distributed algorithms for processing *RkNNQ* in SpatialHadoop and LocationSpark are proposed. In Sect. 4, we present the most representative results of the experiments that we have performed, using real-world datasets, for comparing these two cloud computing frameworks. Finally, in Sect. 5, we provide the conclusions arising from our work and discuss related future work directions.

2   The Reverse *k*-Nearest Neighbor Query

Given a set of points, the *kNN* query (*kNNQ*) discovers the *k* points that are the nearest to a given query point (i.e. it reports only the top-*k* points from a

given query point). It is one of the most important and studied spatial operations, where one spatial dataset and a distance function are involved. The formal definition of the *kNNQ* for points (the extension of this definition to other, more complex spatial objects, as line-segments, is straightforward) is the following:

Definition 1 (k-Nearest Neighbor query, *kNN*) [14]. *Let $\mathbb{P} = \{p_0, p_1, \cdots, p_{n-1}\}$ a set of points in E^d (d-dimensional Euclidean space), a query point q in E^d, and a number $k \in \mathbb{N}^+$. Then, the result of the k-Nearest Neighbor query with respect to the query point q is a set, $kNN(\mathbb{P}, q, k) \subseteq \mathbb{P}$, which contains k ($1 \le k \le |\mathbb{P}|$) different points of $\mathbb{P}$, with the k smallest distances from q:*
$kNN(\mathbb{P}, q, k) = \{p \in \mathbb{P} \colon |p' \in \mathbb{P} : dist(p', q) < dist(p, q)| < k\}$.

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 Reverse k-Nearest Neighbors (*RkNN*) query issued from point q returns all the points of $\mathbb{P}$ whose k nearest neighbors include q. Formally:

Definition 2 (Reverse k-Nearest Neighbor query, *RkNN*) [14]. *Let $\mathbb{P} = \{p_0, p_1, \cdots, p_{n-1}\}$ a set of points in E^d, a query point q in E^d, and a number $k \in \mathbb{N}^+$. Then, the result of the Reverse k-Nearest Neighbor query with respect to the query point q 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:*
$RkNN(\mathbb{P}, q, k) = \{p \in \mathbb{P} : q \in kNN(\mathbb{P}, p, k)\}$.

3   *RkNNQ* Algorithms in SpatialHadoop and LocationSpark

In this section, we present how *RkNNQ* can be implemented in SpatialHadoop and in LocationSpark. But in general, our parallel and distributed *RkNNQ* algorithm is based on the SFT algorithm [10] and it consists of a series of MapReduce jobs. As we can observe in Algorithm 1, the *FILTER* function aims to find a candidate set of points which are the initial results from a MapReduce-based *kNNQ* that uses the partitions from $\mathbb{P}$ that are around q [7]. The *VERIFY* function aims to examine the candidate points from the *FILTER* function using another MapReduce job and return the final set of points that are the reverse k nearest neighbours of q.

3.1   *RkNNQ* Algorithm in SpatialHadoop

From Algorithm 1, we can obtain our proposed solution for *RkNNQ* in Spatial-Hadoop which follows its general processing steps described in [5] and consists of a combination of already implemented Spatial MapReduce operations [4]. Assuming that $\mathbb{P}$ is the dataset 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 b blocks or cells of points. Then, a MapReduce-based *kNNQ* is executed in order to find every possible candidate point from $\mathbb{P}$. To carry out that, we find the partition

Algorithm 1. General Distributed *RkNNQ* Algorithm

1: **function** FILTER($\mathbb{P}$: set of points, q: query point, k: number of points, d: dimensionality)
2:     $K \leftarrow k * d * 10$ ▷ $K = 10 \times 2 \times k$, where $d = 2$
3:     $CandidateSet \leftarrow$ KNN$(\mathbb{P}, q, K)$
4:     **return** $CandidateSet$
5: **end function**

6: **function** VERIFY($\mathbb{P}$: set of points, q: query point, k: number of points, $CandidateSet$: set of PointAndDistance)
7:     INITIALIZE(Result)
8:     **for all** $candidate \in CandidateSet$ **do**
9:         $NumberOfPoints \leftarrow$ RANGE$(\mathbb{P}, candidate.point, candidate.distance)$
10:         **if** $NumberOfPoints < k$ **then**
11:             INSERT($Result$, $candidate$)
12:         **end if**
13:     **end for**
14:   **return** $Result$
15: **end function**

---

from $\mathbb{P}$ where q is located. A first answer for the $kNN(\mathbb{P}, q, K)$ is obtained and we use the distance from the *K-th* point to q in order to find if there are possible candidates in other partitions close to q. To ensure an exact result, the value of K must be greater than k ($K \gg k$) as proposed in [13], at a magnitude of at least $K = 10 \times d \times k$, where d is the dimensionality of the dataset being examined (e.g. for 2d points, $K = 20 \times k$). Next, a range query with a circle centered in q with that distance as radius is run to finally answer the *kNNQ*. The candidates with their distance to the query point q are written into Hadoop Distributed File System (HDFS) files in order to be the input for the next jobs. At this moment, each candidate is checked to see if it is part of the final answer. That is, it finds the number of points that are part of the range query centered on the candidate point and with radius the distance to q. If this *number* is less than k, the point is verified to be a *RkNN* of q. Finally, the results are written into HDFS files, storing only the points coordinates and the distance with q.

3.2   *RkNNQ* Algorithm in LocationSpark

The implementation in LocationSpark uses the provided *knnfilter* and *rangefilter* functions [12] and is very similar to the one implemented for SpatialHadoop. It should be noted that the most important difference when implementing *RkNNQ* in LocationSpark and SpatialHadoop is the fact that the former does not need to store intermediate results on disk, since it is an in-memory *DSDMS*.

Table 1. Configuration parameters used in the experiments.

Parameter	Values (default)
k	1, 5, (10), 15, 20, 25, 50
Number of nodes	1, 2, 4, 6, 8, 10, (12)
Type of partition	Quadtree
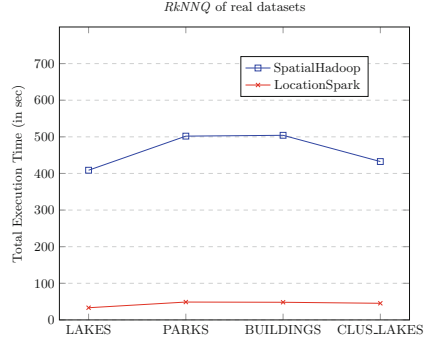
Fig. 1. *RkNNQ* execution times considering different datasets.

4 Experimentation

In this section we present the results of our experimental evaluation. We have used real 2d point datasets to test our *RkNNQ* algorithms in SpatialHadoop and LocationSpark. We have used three datasets from OpenStreetMap[1]: *BUILD-INGS* which contains 115M records of buildings, *LAKES* which contains 8.4M points of water areas, and *PARKS* which contains 10M records of parks and green areas [4]. Moreover, to experiment with the biggest real dataset (*BUILD-INGS*), we have created a new big quasi-real dataset from *LAKES* (8.4M), with a similar quantity of points. The creation process is as follows: taking one point of *LAKES*, *p*, we generate 15 new points gathered around *p* (i.e. the center of the cluster), according to a Gaussian distribution with mean = 0.0 and standard deviation = 0.2, resulting in a new quasi-real dataset, called *CLUS_LAKES*, with around 126M of points. The main performance measure that we have used in our experiments has been the total execution time (i.e. total response time). These values are the average of the execution times of the query on 10 previously obtained random points. All experiments are 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. The version of Spark used is 1.6.2. Finally, we used the latest code available in the repositories of SpatialHadoop[2] and LocationSpark[3].

Table 1 summarizes the configuration parameters used in our experiments. Default values (in parentheses) are used unless otherwise mentioned. Spatial-Hadoop needs the datasets to be partitioned and indexed before invoking the spatial operations. The times needed for that pre-processing phase are

---

[1] Available at http://spatialhadoop.cs.umn.edu/datasets.html.

[2] Available at https://github.com/aseldawy/spatialhadoop2.

[3] Available at https://github.com/merlintang/SpatialSpark.

94 s for $LAKES$, 103 s for $PARKS$, 175 s for $BUILDINGS$ and 200 s for $CLUS\_LAKES$. We decided to exclude indexing time of SpatialHadoop (disk-based DSDMS) for the comparison, since this is an *independent operation*. Data are indexed and the index is stored on HDFS and for subsequent spatial queries, data and index are already available (this can be considered as an advantage of SpatialHadoop). On the other hand, LocationSpark (in-memory-based DSDMS) always partitions and indexes the data for every operation. The partitions/indexes are not stored on any persistent file system and cannot be reused in subsequent operations.

Our first experiment aims to measure the scalability of the distributed *RkNNQ* algorithms, varying the dataset sizes. As shown in Fig. 1, the execution times in both DSDMSs do not vary too much, showing quite stable performance. This is due to the indexing mechanisms provided by both DSDMSs that allow fast access to only the necessary partitions for the spatial query processing. The smaller execution times shown by $LAKES$ and $CLUS\_LAKES$ datasets is due to how the points are distributed into the space and because one dataset is built based on the other, and they show a similar behavior. From the results with real data, we can conclude that LocationSpark is faster for all the datasets (e.g. it is 2131 s faster for the biggest dataset, $CLUS\_LAKES$) thanks to its memory-based processing that allows to reduce execution times considerably.
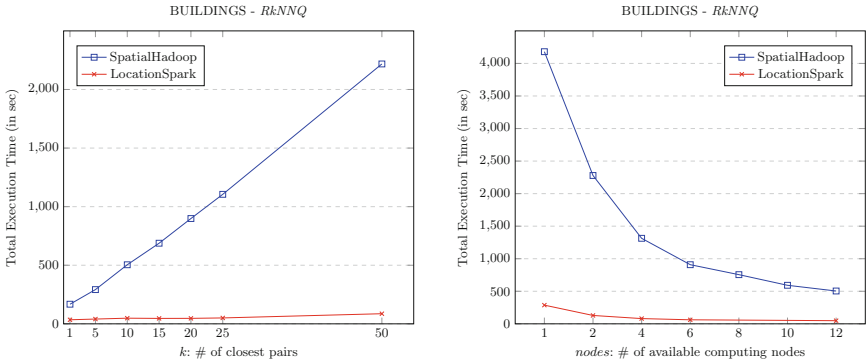
Fig. 2. *RkNNQ* cost (execution time) vs. k values (left). Query cost with respect to the number of computing nodes (*nodes*) (right).

The second experiment studies the effect of the increasing k value for the largest full-real dataset ($BUILDINGS$). The left chart of Fig. 2 shows that the total execution time grows as the value of k increases. As we can see from the results, the execution time for SpatialHadoop grows much faster than for LocationSpark. This is because as the value of k increases, so does the number of candidates K and for each of them a MapReduce job is done. Due to the fact that SpatialHadoop is a disk-based DSDMS, the cost of multiple MapReduce jobs increases the execution time by having to perform different input and output operations for each of the candidate points (for instance, the dataset is read from

disk for each candidate). On the other hand, LocationSpark is a in-memory DSDMS, which allows to reduce the number of disk accesses since the data is already available for each candidate point and thus achieving faster and more stable results even for large k values.

The third experiment aims to measure the speedup of the *RkNNQ* MapReduce algorithms, varying the number of computing nodes (*nodes*). The right chart of Fig. 2 shows the impact of different number of computing nodes on the performance of parallel *RkNNQ* algorithm, for $BUILDINGS$ with the default configuration values. From this chart, it could be concluded that the performance of our approach has a direct relationship with the number of computing nodes. It could also be deduced that better performance would be obtained if more computing nodes are added. LocationSpark is still outperforming Spatial-Hadoop and it is affected to a lesser degree despite reducing the number of available computing nodes.

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*) of the proposed parallel and distributed algorithms for *RkNNQ* in SpatialHadoop and LocationSpark.
– The larger the k values, the greater the number of candidates to be verified, more tasks will be needed and more total execution time is consumed for reporting the final result.
– The larger the number of computing *nodes*, the faster the *RkNNQ* algorithms are.
– Both DSDMSs have similar behavior trends, in terms of execution time, although LocationSpark shows better values in all cases (if an adequate number of processing nodes with adequate memory resources are provided), thanks to its in-memory processing performance and capabilities.

5   Conclusions and Future Work

The *RkNNQ* has received increasing attention in the past years. This spatial query has been actively studied in centralized environments, however, it has not attracted similar attention for parallel and distributed frameworks. For this reason, in this paper, we compare two of the most modern and leading DSDMSs, namely SpatialHadoop and LocationSpark. To do this, we have proposed novel algorithms in SpatialHadoop and LocationSpark, the first ones in the literature, to perform efficient parallel and distributed *RkNNQ* algorithms on big spatial real-world datasets. The execution of a set of experiments has demonstrated that LocationSpark is the overall winner for the execution time, due to the efficiency of in-memory processing provided by Spark. However, SpatialHadoop shows interesting performance trends due to the nature of the proposed algorithm, since the use of multiple MapReduce jobs in a disk-based DSDMS needs multiple disk accesses to datasets. Our current proposal is a good foundation

for the development of further improvements in which the number of candidate points could be reduced by adapting recent *RkNNQ* algorithms [16] to MapReduce methodology. Other future work might cover studying other Spark-based DSDMSs like *Simba* [15] and implement other spatial partitioning techniques [3].

References

1. Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., Saltz, J.H.: Hadoop-GIS: a high performance spatial data warehousing system over MapReduce. PVLDB **6**(11), 1009–1020 (2013)
2. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI Conference, pp. 137–150 (2004)
3. Eldawy, A., Alarabi, L., Mokbel, M.F.: Spatial partitioning techniques in Spatial-Hadoop. PVLDB **8**(12), 1602–1613 (2015)
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., Manolopoulos, Y.: Enhancing SpatialHadoop with closest pair queries. In: Pokorný, J., Ivanović, M., Thalheim, B., Šaloun, P. (eds.) ADBIS 2016. LNCS, vol. 9809, pp. 212–225. Springer, Cham (2016). doi:10.1007/978-3-319-44039-2_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. Concurr. Comput.: Pract. Exp. **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. Li, F., Ooi, B.C., Özsu, M.T., Wu, S.: Distributed data management using MapReduce. ACM Comput. Surv. **46**(3), 1–42 (2014)
10. Singh, A., Ferhatosmanoglu, H., Tosun, H.S.: High dimensional reverse nearest neighbor queries. In: CIKM Conference, pp. 91–98 (2003)
11. Stanoi, I., Agrawal, D., El Abbadi, A.: Reverse nearest neighbor queries for dynamic databases, pp. 44–53. In: SIGMOD Workshop on Research Issues, Data Mining and Knowledge Discovery (2000)
12. Tang, M., Yu, Y., Malluhi, Q.M., Ouzzani, M., Aref, W.G.: LocationSpark: a distributed in-memory data management system for big spatial data. PVLDB **9**(13), 1565–1568 (2016)
13. Tao, Y., Papadias, D., Lian, X.: Reverse kNN search in arbitrary dimensionality. In: VLBD Conference, pp. 744–755 (2004)
14. Wu, W., Yang, F., Chan, C.Y., Tan, K.L.: FINCH: evaluating reverse k-Nearest-Neighbor queries on location data. PVLDB **1**(1), 1056–1067 (2008)
15. Xie, D., Li, F., Yao, B., Li, G., Zhou, L., Guo, M.: Simba: efficient in-memory spatial analytics. In: SIGMOD Conference, pp. 1071–1085 (2016)
16. Yang, S., Cheema, M.A., Lin, X., Wang, W.: Reverse k nearest neighbors query processing: experiments and analysis. PVLDB **8**(5), 605–616 (2015)
17. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI Conference, pp. 15–28 (2012)
18. Zhang, H., Chen, G., Ooi, B.C., Tan, K.-L., Zhang, M.: In-memory big data management and processing: a survey. TKDE **27**(7), 1920–1948 (2015)