Efficient distance join query processing in distributed spatial data management systems

Francisco García-García[a], Antonio Corral[a,*], Luis Iribarne[a], Michael Vassilakopoulos[b], Yannis Manolopoulos[c]

[a] *Department on Informatics, University of Almeria, Almeria, Spain*
[b] *Department of Electrical & Computer Engineering, University of Thessaly, Volos, Greece*
[c] *School of Natural & Applied Sciences, Open University of Cyprus, Nicosia, Cyprus*

ARTICLE INFO

Article history:
Received 4 September 2018
Revised 2 October 2019
Accepted 14 October 2019
Available online 14 October 2019

Keywords:
Distance join
Spatial data processing
Space partitioning
SpatialHadoop
LocationSpark
Spatial query evaluation

ABSTRACT

Due to the ubiquitous use of spatial data applications and the large amounts of such data these applications use, the processing of large-scale distance joins in distributed systems is becoming increasingly popular. *Distance Join Queries* (DJQs) are important and frequently used operations in numerous applications, including data mining, multimedia and spatial databases. DJQs (e.g., k Nearest Neighbor Join Query, k Closest Pair Query, ε Distance Join Query, etc.) are costly operations, since they involve both the join and distance-based search, and performing DJQs efficiently is a challenging task. Recent Big Data developments have motivated the emergence of novel technologies for distributed processing of large-scale spatial data in clusters of computers, leading to *Distributed Spatial Data Management Systems* (DSDMSs). Distributed cluster-based computing systems can be classified as Hadoop-based or Spark-based systems. Based on this classification, in this paper, we compare two of the most recent and leading DSDMSs, SpatialHadoop and LocationSpark, by evaluating the performance of several existing and newly proposed parallel and distributed DJQ algorithms under various settings with large spatial real-world datasets. A general conclusion arising from the execution of the distributed DJQ algorithms studied is that, while SpatialHadoop is a robust and efficient system when large spatial datasets are joined (since it is built on top of the mature Hadoop platform), LocationSpark is the clear winner in total execution time efficiency when medium spatial datasets are combined (due to in-memory processing provided by Spark). However, LocationSpark requires higher memory allocation when large spatial datasets are involved in DJQs (even more so when k and ε are large). Finally, this detailed performance study has demonstrated that the new distributed DJQ algorithms we have proposed are efficient, robust and scalable with respect to different parameters, such as dataset sizes, k, ε and number of computing nodes.

* Corresponding author.
E-mail addresses: paco.garcia@ual.es (F. García-García), acorral@ual.es (A. Corral), liribarn@ual.es (L. Iribarne), mvasilako@inf.uth.gr (M. Vassilakopoulos), yannis.manolopoulos@ouc.ac.cy (Y. Manolopoulos).

https://doi.org/10.1016/j.ins.2019.10.030

1. Introduction

Nowadays, the volume of available spatial data, generated by different sources, like smart phones, satellites, etc., is rapidly increasing all over the world. Recent research on Big Data has motivated the emergence of novel technologies for distributed processing of large-scale spatial data on clusters of computers [7]. These Distributed Spatial Data Management Systems (DSDMSs) can be classified as disk-based [26] or in-memory [49]. The disk-based DSDMSs are characterized as Hadoop-based systems, and the most representative are Hadoop-GIS [2] and SpatialHadoop [15]. The Hadoop-based systems enable the execution of spatial queries using predefined high-level spatial operators, without having to worry about fault tolerance and computation distribution. The in-memory DSDMSs are characterized as Spark-based systems, and the most representative are SpatialSpark [43], GeoSpark [45], Simba [39], STARK [21] and LocationSpark [36,37]. Spark-based systems allow users to work on distributed in-memory data, without (as in Hadoop-based systems) worrying about the data distribution mechanism and fault-tolerance.

Distance join queries (DJQs) have received considerable attention from the database community, due to their importance in numerous applications, such as spatial databases and GIS, data mining, multimedia databases, etc. DJQs are costly queries because they combine two datasets, taking into account a distance metric. The most representative are the k Nearest Neighbor Join Query (kNNJQ), k Closest Pair Query (kCPQ), ε Distance Range Join Query (εDRJQ) and ε Distance Join Query (εDJQ). Given two points datasets $\mathbb{P}$ and $\mathbb{Q}$, the kNNJQ finds the k nearest neighbors in $\mathbb{Q}$ for each point in $\mathbb{P}$. The kCPQ finds the k closest pairs of points from $\mathbb{P} \times \mathbb{Q}$ for a given distance function (e.g., Manhattan, Euclidean, Chebyshev, etc.). The εDRJQ finds for each point in $\mathbb{P}$, all points in $\mathbb{Q}$ that fall within the circular shape with radius equal to ε centered at that point in $\mathbb{P}$. The εDJQ finds all the possible pairs of points from $\mathbb{P} \times \mathbb{Q}$ within a distance threshold ε of each other. These DJQs have received much attention in the literature, because they have important roles in many real-life applications. Three representative application cases are given here: *Application case 1* (Resource Management in Agriculture), authorities planning the sustainable exploitation of water resources are considering two spatial datasets, locations of water wells and areas of cultivable lands. εDRJQ could “find all the land areas within 3 km from every water well”, (the borders, or the centroid of each land area could be used for processing this query). *Application case 2* (Mobile Location Services), with two spatial datasets, locations of shopping centers and positions of possible customers using a smart phone with mobile data and GPS enabled. kNNJQ could “find the 100 nearest possible customers to each shopping center” for sending an advertising SMS about a fashion brand available there. *Application case 3* (Transportation Monitoring and Moving Objects), considering two spatial datasets, locations of users of a taxi app and positions of free taxis. kCPQ could “find the 10 pairs of app users and taxis with the shortest distances between them”, to be able to offer these users fast service at a reduced price (as a promotion strategy), or for analysis by the taxi service. Several studies have attempted to improve the performance of these DJQs by proposing efficient algorithms in centralized environments [6,9,10,33]. However, with the rapid increase in the scale of large input datasets, parallel and distrituted processing of large-scale data is becoming a popular practice. Therefore, a number of parallel algorithms for DJQs have been designed and implemented [16,18,27,31,35,36,42,48] in MapReduce [13] and Spark [47].

Apache Hadoop[1] is a reliable, scalable, and efficient cloud computing framework enabling distributed processing of large datasets using the MapReduce programming model [5]. However, it is a type of disk-based computing framework, which writes all intermediate data between *map* and *reduce* tasks to the disk. MapReduce [13] is a framework for processing and managing large-scale datasets in a distributed cluster. It was introduced to provide a simple yet powerful parallel and distributed computing paradigm, providing good scalability and fault tolerance mechanisms. Apache Spark[2] is a fast, reliable and distributed in-memory large-scale data processing framework. It takes advantage of Resilient Distributed Datasets (RDDs), that allow data to be stored transparently in memory and persisted to disk only if necessary [47]. Hence, it can avoid a huge number of disk writes and reads, and outperform the Hadoop platform. Since Spark maintains the status of assigned resources until a job is completed, it reduces time consumed in resource preparation and collection [23].

Both Hadoop and Spark are less efficient when applied to spatial data [43,45]. One main shortcoming is that there is no indexing mechanism for selective access to specific regions of spatial data, which would make query processing algorithms more efficient. This problem could be solved by a Hadoop extension called *SpatialHadoop* [15], which is a framework supporting spatial indexing on top of Hadoop, i.e., spatial data is organized on a two-level index (global and local). Another possible solution is *LocationSpark* [36,37], which is a spatial data processing system built on top of Spark that employs various spatial indexes for in-memory data.

Thus, there are several distributed systems based on Hadoop or Spark for managing spatial data, but there are not many studies comparing their use for spatial query processing of large real-world datasets. The only publications in this regard are [20,25,44,45]. In [44,45], SpatialHadoop was compared to SpatialSpark and GeoSpark, but only for spatial join query processing. SpatialHadoop and GeoSpark architectures were compared in [25]. In [20], the existing solutions for spatial data processing on Hadoop-based (Hadoop-GIS and SpatialHadoop) and Spark-based (GeoSpark, SpatialSpark and STARK) systems were analyzed and compared. Their features and performance in a micro-benchmark for spatial filters and join queries were also studied. kCPQ and εDJQ have been implemented in SpatialHadoop [18,19] and LocationSpark [16]. However, kNNJQ has

[1] Available at https://hadoop.apache.org/ .
[2] Available at https://spark.apache.org/ .

Table 1
DJQs in SpatialHadoop and LocationSpark.

DSDMS	kNNJQ	εDRJQ	kCPQ	εDJQ
SpatialHadoop	**New**	**New**	[18,19]	[17,19]
LocationSpark	[36]	**New**	[16]	[16]

not been implemented in SpatialHadoop, and εDRJQ has not been included in either of the two DSDMSs, so far. Motivated by these observations, we designed and implemented **New** distributed DJQ algorithms in DSDMSs, as shown in Table 1. We also compared the performance of SpatialHadoop and LocationSpark for all the DJQs in the table, using big real-world datasets. SpatialHadoop was chosen for this comparative study from among other Hadoop-based DSDMSs because it is a very efficient MapReduce framework for spatial data processing, and already includes several spatial operations. Other disk-based DSDMSs, like Hadoop-GIS, are not directly implemented on top of Hadoop, but on other components and layers in its ecosystem like Hive, making it more difficult to implement efficient new spatial queries using the MapReduce methodology. Moreover, these DSDMSs lack the richness of functionalities and spatial operations, provided by SpatialHadoop. LocationSpark [36] was selected from among other Spark-based DSDMSs for this comparative study, since several spatial queries have already been implemented (e.g., kNNJQ) in this framework, and creating new ones is very easy, because it is directly implemented over the RDD API, and it provides promising features for efficient query processing (e.g., new components like *sFilter* and *Query Scheduler* for managing skewed data and reducing network communication cost) [32]. Other Spark-based DSDSMs, such as Simba [39], are implemented on other layers or APIs such as the Dataframe or SQL APIs, making the implementation of new spatial queries, as well as the translation of MapReduce algorithms to Spark, more complicated.

The present research work is based on a completely new setting with respect to [16,18,19], as we have enhanced SpatialHadoop with the implementation of kNNJQ and εDRJQ and LocationSpark with εDRJQ. In [18,19], kCPQ and εDJQ were implemented in SpatialHadoop with enhancements based on sampling and approximation techniques. In this paper, these two DJQs were implemented in LocationSpark and compared to the respective distributed algorithms in SpatialHadoop. Moreover, in this paper, new methodologies and improvements are proposed to speed up the response time of the DJQs studied in cloud computing environments. Finally, new experiments with new and larger real-world datasets were performed to deduce new conclusions from the new distributed algorithms and improvements.

This paper substantially extends our previous works [18] and [16] with the following novel contributions:

1. Novel MapReduce algorithms for kNNJQ and εDRJQ in SpatialHadoop and εDRJQ in LocationSpark.
2. Improved MapReduce algorithms proposed for kNNJQ and εDRJQ in SpatialHadoop using *repartitioning* techniques in dense spatial areas.
3. Extended distributed algorithms for managing spatial objects more complex than points, like polygons or line-segments.
4. Results of an extensive experimental study comparing the performance, using execution time and shuffled data, of the proposed MapReduce algorithms and their improvements of efficiency and scalability, using big real-world spatial datasets on both DSDMSs (SpatialHadoop and LocationSpark).

This paper is organized as follows. In Section 2, we review related work on Hadoop-based and Spark-based systems that support spatial operations and provide the motivation for this paper. In Section 3, we present preliminary concepts related to DJQs, SpatialHadoop and LocationSpark. The distributed algorithms for processing DJQs in SpatialHadoop and LocationSpark are discussed in Section 4. Section 5 presents several potential improvements in the proposed distributed algorithms. Section 6 presents representative results of the extensive experimental study performed for comparing these two cloud computing frameworks, using real-world datasets. We also discuss the most important conclusions drawn from these experiments. Finally, Section 7 presents the conclusions arising from our work and discuss related future directions of study.

2. Related work

Researchers, developers and practitioners worldwide are now benefiting from cluster-based systems supporting large-scale data processing. There are several cluster-based systems supporting spatial queries in distributed spatial datasets (briefly reviewed in the following subsections) which can be classified as Hadoop-based or Spark-based DSDMSs. We also give an overview of current research on distributed algorithms for computing DJQs.

2.1. Distributed spatial data management systems

The most important contributions in the context of Hadoop-based DSDMSs are the following research prototypes: *Hadoop-GIS* [2] extends Hive, adopts the Hadoop Streaming framework and integrates several open source software packages for spatial indexing and geometry computation. Hadoop-GIS only supports data up to two dimensions and two query types, rectangle range query and spatial joins. *SpatialHadoop* [15] is an extension of the MapReduce framework [13], based on Hadoop, with native support for spatial 2d data. It is an efficient disk-based distributed spatial query processing system. SpatialHadoop can support spatial index structures including R-tree and Grid file, which is a built-in Hadoop Distributed

Table 2
The most representative existing DSDMSs based on Hadoop or Spark.

DSDMS	Architecture	Spatial index	Spatial query
Hadoop-GIS [2]	Hadoop	Grid	range query, spatial join
ST-Hadoop [4]	Hadoop	ST-index	ST-range query, ST-join, ST-aggregates, *k*NNQ
SpatialHadoop [15]	Hadoop	R-tree, Grid	range query, *k*NNQ, spatial join
SpatialSpark [43]	Spark	Grid, *k*d-tree	range query, spatial join
GeoSpark [46]	Spark	R-tree, Quadtree	range query, *k*NNQ, spatial join, distance join
Simba [39]	Spark	R-tree	range query, *k*NNQ, distance join, *k*NNJQ
STARK [21]	Spark	R-tree, Grid	range query, *k*NNQ, spatial join
LocationSpark [37]	Spark	R-tree, Grid, Quadtree, IR-tree	range query, *k*NNQ, spatial join, distance join, *k*NNJQ

File System (HDFS). SpatialHadoop is equipped with several spatial operations, including range query, *k*NN and spatial join [15], and other fundamental computational geometry algorithms, such as polygon unions, skylines, convex hulls, farthest pairs, and closest pairs. Other spatial queries implemented on SpatialHadoop are skyline and *k*CP [18,19] queries. Finally, *ST-Hadoop* [4] is a full-fledged open-source MapReduce framework with native support for spatio-temporal data. ST-Hadoop is a comprehensive Hadoop and SpatialHadoop extension that injects spatio-temporal data awareness inside each layer.

The most outstanding Spark-based DSDMS research prototypes are: *SpatialSpark* [43], a lightweight implementation of several spatial operations on top of Spark in-memory big data system. It has in-memory processing for higher performance. SpatialSpark includes data partitioning strategies, like fixed grid or *k*d-tree on data files in HDFS and builds an index to accelerate spatial operations. It supports range queries and spatial joins over geometric objects using spatial conditions, like *intersect* and *within*. *GeoSpark* [45,46] extends Spark for processing spatial data. It provides a new abstraction, called Spatial Resilient Distributed Datasets (SRDDs), and a few spatial operations. An index (e.g., Quadtree and R-tree) can be the object in each local RDD partition. From the viewpoint of query processing, GeoSpark supports range query, *k*NNQ and spatial joins over SRDDs. *Simba* (Spatial In-Memory Big data Analytics) [39] offers scalable and efficient in-memory spatial query processing and analytics for big spatial data. Simba is based on Spark and runs over a cluster of commodity machines. In particular, Simba extends the Spark SQL engine to support rich spatial queries and analytics through both SQL and the DataFrame API. It introduces partitioning techniques (e.g., STR), indexes (global and local) based on R-trees over RDDs to work with big spatial data and complex spatial operations (e.g., range query, *k*NNQ, distance join and *k*NNJQ). The *STARK* [21] framework adds spatio-temporal support to Spark, includes spatial partitioners, several modes for indexing, as well as filter, join, and clustering operators. *LocationSpark* [36,37] is an efficient in-memory distributed spatial query processing system characterized as a Spark-based system. It provides promising features for the query processing efficiency, like data and query skew components to improve load balancing while executing spatial operators (e.g., spatial range, *k*NN search, spatial range join and *k*NN join), by generating cost-optimized query execution plans over in-memory distributed spatial data. In addition, each data partition has a local spatial index (e.g., a Grid local index, an R-tree, a variant of the Quadtree, or an IR-tree). Finally, according to [40], Table 2 lists the most representative DSDMSs based on Hadoop or Spark, which are compared for architecture, spatial index, and spatial query.

[32] explored the availability of spatial analytics systems (based on Spark), comparing their features and queries by running experiments that evaluated their performance and other metrics using real-world datasets. Only LocationSpark and Simba support *k*NNJQ, and LocationSpark also had the best performance, scalability and shuffle cost results. As a conclusion, the authors emphasized that LocationSpark is a very interesting option, since it has a very good query scheduler and optimizer. It also has a spatial bloom filter (sFilter) which lowers query costs. Moreover, they also suggested that these features could be incorporated in the other Spark-based systems studied.

2.2. Distance join queries in distributed environments

The *k*NNJQ MapReduce algorithm has been extensively studied in the literature. Yokoyama et al. [42] presented a *k*NNJQ MapReduce algorithm for 2D spatial data. It decomposes the data space into small equal cells (Grid) and afterwards merges some neighboring cells, always in 2×2 sets, if they do not contain a total k points or more. The algorithm thus creates larger cells so that the *k*NN list of a query point is always complete. In [31], the algorithm in [42] was improved by replacing the merging step with a circle of increasing radius around the query point, so that it checks for candidate neighbors in nearby cells, making the merging step unnecessary, and the number of distance calculations may be significantly reduced.

The Voronoi-Diagram based partitioning technique is used in *k*NNJQ MapReduce algorithms [3,24,27] for exploiting pruning rules for distance filtering, and hence reduces both shuffling and computational costs. In [3], the Voronoi-Diagram based partitioning approach using MapReduce is used to answer range search and *k*NN search queries in 2d spaces. In this algorithm, each object in the dataset is taken as a pivot for partitioning the space. Lu et al. [27] used MapReduce to solve the problem of answering the *k*NNJ by exploiting the Voronoi-Diagram based partitioning method, which divides the input datasets into groups, and can answer *k*NNJ by checking object pairs within each group. Moreover, several pruning rules developed reduce the shuffling and computation costs in the PGBJ (Partitioning and Grouping Block Join) algorithm, which works with two MapReduce phases. Kim et al. [24] proposed the vector projection pruning technique to process *k*NNJ efficiently by enabling non-*k*NN points to be pruned and reduce the cost of distance computations. They presented a new algorithm, *k*NN-MR, using this new pruning technique that performs better than PGBJ.

Zhang et al. [48] proposed novel (exact and approximate) MapReduce algorithms to perform efficient parallel *k*NNJQ on large datasets, and used the R-tree and Z-value-based partition joins to implement them. Current solutions performing the *k*NNJ operation in the MapReduce context were reviewed and studied from the theoretical and experimental point of view in [35].

Finally, the *k*CPQ and εDJQ MapReduce algorithms [18,19] are included in SpatialHadoop. These algorithms consist of MapReduce jobs, using the plane-sweep technique and improving computation of an upper bound of the distance value of the *k*-th closest pair from data sampled as a local and global preprocessing phase. Mavrommatis et al. [29] proposed a new distributed algorithm (SliceNBound) for solving closest pairs and distance join queries in Apache Spark based on a simple, and not very computationally demanding, partitioning scheme (slicing the plane into strips and sampling to bound the solution space) that enables the two datasets to share the same partitioning.

3. Preliminaries and background

Semantic details of the respective DJQs and the corresponding notation and processing paradigms are presented below. The most important characteristics of both DSDMSs (SpatialHadoop and LocationSpark) for DJQ processing are also reviewed.

3.1. Distance-based join queries

To introduce the details of the DJQ semantics studied here, we define the *k*CP, *k*NNJ, εDJ and εDRJ queries, assuming that the Euclidean distance, *dist*, is the distance used throughout the article. Moreover, we also define two distance-based queries which form their basis, the *k* Nearest Neighbor (*k*NN) and ε Distance Range (εDR) queries, where only one dataset is processed.

Given a point dataset, the *k*NNQ discovers the *k* closest points to a given query point (i.e., it reports only the top *k* points). It involves one spatial dataset and a distance function, and is one of the most important and studied spatial operations. The formal definition of the *k*NNQ for points is:

Definition 1. ***k*Nearest Neighbor query, kNN query**

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 an ordered collection, $kNN(\mathbb{P}, q, k) \subseteq \mathbb{P}$, which contains the k ($1 \leq k \leq |\mathbb{P}|$) different points of $\mathbb{P}$, with the k smallest distances from q:

$kNN(\mathbb{P}, q, k) = (p_1, p_2, \cdots, p_k) \in \mathbb{P}$, such that for any $p_i \in \mathbb{P} \setminus kNN(\mathbb{P}, q, k)$ we have $dist(p_1, q) \leq dist(p_2, q) \leq \cdots \leq dist(p_k, q) \leq dist(p_i, q)$.

The εDRQ query, given a point dataset, finds all the points of the dataset that fall within the circular shape, centered on a query point (q) with distance threshold radius ε. This query is also called a *circle range query* or *circular query*. Its formal definition is:

Definition 2. ***εDistance Range query, εDR query***

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 distance threshold $\varepsilon \in \mathbb{R}^+$. Then, the result of the εDistance Range query with respect to the query point q is a set, $\varepsilon DR(\mathbb{P}, q, \varepsilon) \subseteq \mathbb{P}$, which contains all points $p_i \in \mathbb{P}$ that fall on the circular shape, centered in q with radius ε:

$\varepsilon DR(\mathbb{P}, q, \varepsilon) = \{p_i \in \mathbb{P} \ : \ dist(p_i, q) \ \leq \ \varepsilon\}$

When two datasets ($\mathbb{P}$ and $\mathbb{Q}$) are combined, four of the most studied DJQs are the *k* Nearest Neighbor Join (*k*NNJ) query, the εDistance Range Join (εDRJ) query, the *k* Closest Pairs (*k*CP) query and the εDistance Join (εDJ) query.

The *k*NNJQ, given two point datasets ($\mathbb{P}$ and $\mathbb{Q}$) and a positive number *k*, finds for each point in $\mathbb{P}$, its *k* nearest neighbors in $\mathbb{Q}$. The formal definition of this kind of DJQ is given below:

Definition 3. ***k*Nearest Neighbor Join query, kNNJ query**

Let $\mathbb{P} = \{p_0, p_1, \cdots, p_{n-1}\}$ and $\mathbb{Q} = \{q_0, q_1, \cdots, q_{m-1}\}$ be two set of points in E^d, and a natural number $k \in \mathbb{N}^+$. Then, the result of the *k* Nearest Neighbor Join query is a set $kNNJ(\mathbb{P}, \mathbb{Q}, k) \subseteq \mathbb{P} \times \mathbb{Q}$, which contains for each point of $\mathbb{P}$ ($p_i \in \mathbb{P}$) its *k* nearest neighbors in $\mathbb{Q}$:

$kNNJ(\mathbb{P}, \mathbb{Q}, k) = \{(p_i, q_j) : \ \forall \ p_i \in \mathbb{P}, \ q_j \in kNN(\mathbb{Q}, p_i, k)\}$

Since, $kNNJ(\mathbb{P}, \mathbb{Q}, k)$ returns for each point in $\mathbb{P}$, its *k*NNs in $\mathbb{Q}$, it is equivalent to the query called *All-k-Nearest Neighbor* (A*k*NN) query [8,50] in the literature. Moreover, a variant of the A*k*NN query for εDR query is the All-ε-Distance Range ($A\varepsilon DR$) or εDistance Range Join (εDRJ) query.

The εDRJ query, given two point datasets ($\mathbb{P}$ and $\mathbb{Q}$) and a distance threshold ε, finds, for each point $p_i \in \mathbb{P}$, all the points in $\mathbb{Q}$ that fall within the circular shape, centered on p_i with radius ε. This query is also called *spatial range join query*. The formal definition is:

Definition 4. ***εDistance Range Join query, εDRJ query***
Let $\mathbb{P} = \{p_0, p_1, \cdots, p_{n-1}\}$ and $\mathbb{Q} = \{q_0, q_1, \cdots, q_{m-1}\}$ be two set of points in E^d, and a distance threshold $\varepsilon \in \mathbb{R}^+$. Then, the result of the εDistance Range Join query is a set, $\varepsilon DRJ(\mathbb{P}, \mathbb{Q}, \varepsilon) \subseteq \mathbb{P} \times \mathbb{Q}$, which contains for each point of $\mathbb{P}$ ($p_i \in \mathbb{P}$) all points from $\mathbb{Q}$ ($q_j \in \mathbb{Q}$) that fall on the circular shape, centered in p_i with radius ε:
$\varepsilon DRJ(\mathbb{P}, \mathbb{Q}, \varepsilon) = \{(p_i, q_j) : \ \forall\ p_i \in \mathbb{P},\ \forall q_j \in \varepsilon DR(\mathbb{Q}, p_i, \varepsilon)\}$

The *k*CPQ discovers the *k* pairs of points formed from two datasets ($\mathbb{P}$ and $\mathbb{Q}$) having the *k* smallest distances between them (i.e., it reports only the top *k* pairs). The formal definition of this DJQ is:

Definition 5. ***k Closest Pairs query, kCP query***
Let $\mathbb{P} = \{p_0, p_1, \cdots, p_{n-1}\}$ and $\mathbb{Q} = \{q_0, q_1, \cdots, q_{m-1}\}$ be two set of points in E^d, and a natural number $k \in \mathbb{N}^+$. Then, the result of the *k* Closest Pairs query is an ordered collection, $kCP(\mathbb{P}, \mathbb{Q}, k)$, containing *k* different pairs of points from $\mathbb{P} \times \mathbb{Q}$, ordered by distance, with the *k* smallest distances between all possible pairs:
$kCP(\mathbb{P}, \mathbb{Q}, k) = ((p_1, q_1), (p_2, q_2), \cdots, (p_k, q_k)),\ \ (p_i, q_i) \in \mathbb{P} \times \mathbb{Q},\ \ 1 \le i \le k,$ such that for any $(p, q) \in \mathbb{P} \times \mathbb{Q} \setminus kCP(\mathbb{P}, \mathbb{Q}, k)$ we have $dist(p_1, q_1) \le dist(p_2, q_2) \le \cdots \le dist(p_k, q_k) \le dist(p, q)$.

The εDJQ finds all the possible pairs of points from two datasets, that are within a threshold distance ε of each other. The formal definition of this query is given bellow:

Definition 6. ***ε Distance Join query, εDJ query***
Let $\mathbb{P} = \{p_0, p_1, \cdots, p_{n-1}\}$ and $\mathbb{Q} = \{q_0, q_1, \cdots, q_{m-1}\}$ be two set of points in E^d, and a distance threshold $\varepsilon \in \mathbb{R}^+$. Then, the result of the εDistance Join query is the set, $\varepsilon DJ(\mathbb{P}, \mathbb{Q}, \varepsilon) \subseteq \mathbb{P} \times \mathbb{Q}$, containing all the possible different pairs of points from $\mathbb{P} \times \mathbb{Q}$ that have a distance of each other smaller than, or equal to ε:
$\varepsilon DJ(\mathbb{P}, \mathbb{Q}, \varepsilon) = \{(p_i, q_j) \in P \times Q\ :\ dist(p_i, q_j)\ \le\ \varepsilon\}$

Note that εDJQ may be considered an extension of the *k*CPQ, where the threshold distance of the pairs is known beforehand. This query is also related to the *similarity join* in multidimensional databases, where the problem of deciding if two objects are similar is reduced to the problem of determining if two multidimensional points are within a certain distance of each other.

After studying the results of εDRJQ and εDJQ, they are found be equivalent, i.e., both DJQs report the same result set. The main difference resides in the order of the pairs returned in the final result. While $\varepsilon DRJ(\mathbb{P}, \mathbb{Q}, \varepsilon)$ reports pairs clustered around every point in $\mathbb{P}$ (i.e., for each point $p_i \in \mathbb{P}$, it returns all points in $\mathbb{Q}$ overlapping with a circular shape, centered on p_i with radius ε), while $\varepsilon DJ(\mathbb{P}, \mathbb{Q}, \varepsilon)$ reports unrelated pairs of points (i.e., it returns a sequence of pairs within a distance threshold (ε) from each other). Another difference between the two DJQs is the algorithmic technique used to solve them. While εDRJQ is processed based on multiple executions of εDRQ on $\mathbb{Q}$ for every point in $\mathbb{P}$, the algorithm for solving εDJQ is based on a sort-merge join approach (i.e., it is a plane-sweep algorithm between $\mathbb{P}$ and $\mathbb{Q}$).

Finally, other related DJQs can be deduced and formulated from the above. For example, *εDistance Range k* query, which returns the *k* points in $\mathbb{Q}$ with the smallest distances within the specified distance threshold ε around each query point $p_i \in \mathbb{P}$. Or the *εDistance Join k* query, which returns only the *k* pairs with the smallest distances from all possible different pairs of points, having a distance less than or equal to ε of each other. Or the *Iceberg Distance Join* query [34], which returns object pairs (p_i, q_j) such that $p_i \in \mathbb{P}$ and $q_j \in \mathbb{Q}$, within distance ε from each other, provided that p_i appears at least *k* times in the join result. These are straightforward extensions of those above and may be considered a target for further research in the context of DSDMSs.

3.2. SpatialHadoop and LocationSpark for DJQ processing

SpatialHadoop [15] is an efficient disk-based distributed spatial query processing system. SpatialHadoop enables the efficient implementation of spatial operations by considering the combination of spatial indexing with new spatial functionality in *MapReduce*. In general, spatial query processing in SpatialHadoop [15] (in particular, for DJQ processing [18,19]) consists of four steps : (1) *Preprocessing*, where the data is partitioned according to a specific spatial index, generating a set of partitions or cells. (2) *Pruning*, where the master node examines all partitions and by means of a *filter* function, prunes those it is sure will not include any possible result of the spatial query. (3) *Local Spatial Query Processing*, where local spatial query processing is performed on each unpruned partition in parallel on different machines (*map* tasks). Finally, (4) *Global Processing*, where the results are collected from all machines in the previous step and the final result of the spatial query concerned is computed. A *combine* function can be applied 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.

LocationSpark [36,37] is a library in Spark that provides an API for spatial query processing and optimization based on Spark's standard dataflow operators. It is an efficient in-memory distributed spatial query processing system. LocationSpark is optimized to enhance Spark for managing spatial data, and is organized in layers: Memory Management, Spatial Index, Query Executor, Query Scheduler, Spatial Operators and Spatial Analytical. LocationSpark builds two levels of spatial indexes (global and local). For the global index, it samples the underlying data to find out the data distribution in a space and provides a grid and a region Quadtree. Each data partition has a local index (e.g., a grid local index, an R-tree, a variant of the

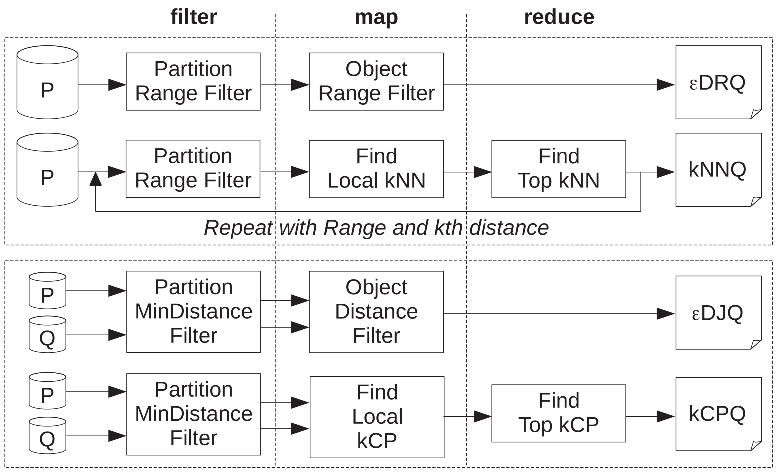

Fig. 1. Overview of several MapReduce distance-based query algorithms in SpatialHadoop.

Quadtree, or an IR-tree). For DJQs, given two datasets $\mathbb{P}$ and $\mathbb{Q}$, $\mathbb{P}$ is partitioned into N partitions based on spatial index criteria (e.g., N leaves of an R-tree) by the *Partitioner* [37] leading to the *PRDD* (global index). The *sFilter* [37] determines whether a point is contained inside a spatial range or not. Next, each *worker* has a local data partition $\mathbb{P}_i$ $(1 \leq i \leq N)$ and builds up a local index [37]. *QRDD* is generated from $\mathbb{Q}$ by a member function of RDD natively supported by Spark, *partitionBy* [37], that forwards such point to the partitions that spatially overlap it. Now, each point of $\mathbb{Q}$ is replicated to the partitions that are identified using the *PRDD* (global index), leading to the *Q'RDD*. Then a post-processing step (using the *Skew Analyzer* and the *Plan Optimizer*) is performed to combine the local results to generate the final output [37].

4. DJQ Algorithms in SpatialHadoop and LocationSpark

In this section, we first review the most representative DJQ MapReduce algorithms already implemented in SpatialHadoop, and then we present new MapReduce algorithms for kNNJQ and εDRJQ in SpatialHadoop. In the second subsection, we describe the distributed DJQ algorithms implemented in LocationSpark and the new algorithms for kCPQ, εDJQ and εDRJQ.

4.1. DJQ MapReduce algorithms in SpatialHadoop

From the definitions of kNNJQ and εDRJQ, it may be observed that they can be formulated on the basis of kNNQ and εDRQ, respectively. Eldawy and Mokbel [15] proposed a generic range query operation in SpatialHadoop. But, a εDRQ MapReduce algorithm on top of SpatialHadoop was efficiently implemented in [17]. In general, the solution for εDRQ is similar to how the *range query* algorithm [15] is performed in SpatialHadoop, except instead of having a generic query area, there is a circular region defined by the query point q and a distance threshold ε. Fig. 1 shows operation of the εDRQ algorithm in SpatialHadoop, which consists of two MapReduce jobs: initial response and refinement. The initial response is obtained by a *filtering* function in which the partitions from $\mathbb{P}$ that intersect with the circular region centered at the query point q with a radius equal to the distance threshold ε are selected. Next, a *map*-type task is performed in which, for each selected cell, a plane-sweep algorithm is used to select only those points within a distance smaller than ε. Finally, these points are written in files as the final result.

[15] presented a kNNQ operation in SpatialHadoop. The proposed kNNQ MapReduce algorithm has three steps: the *initial answer*, the *correctness check* and *answer refinement*. In Fig. 1, the previous steps are shown as a pair of MapReduce jobs that calculate the initial result and are run again iteratively if they do not pass the correctness check until the final answer is obtained. Similar to the εDRQ algorithm, a *filtering* function selects the cell in which the query point q is found. Then, the *map* task is responsible for obtaining the initial answer and in the *reduce* task, the k nearest neighbors from q in that cell are returned. The correction phase checks whether the result obtained is less than k, whether cells are within the circular range query, and centered on q with a radius equal to the k greatest distance obtained so far. If yes, the answer refinement starts by rerunning the previous MapReduce job from the first step, but this time, the cells within the range query are selected by the *filtering* function. Otherwise, the final result has already been found.

In general, the kCPQ MapReduce algorithm in SpatialHadoop [18,19] consists of a MapReduce job, as shown in Fig. 1. The aim of the *map* function is to find the kCP between each local pair of partitions from $\mathbb{P}$ and $\mathbb{Q}$ with a particular plane-sweep kCPQ algorithm [33] and the result is stored in a binary max heap (called *LocalKMaxHeap*). The *reduce* function examines the candidate pairs of points from each *LocalKMaxHeap* and returns the final set of the k closest pairs in another binary max heap (called *GlobalKMaxHeap*). This approach can be improved by finding in advance, an upper bound of the distance of the

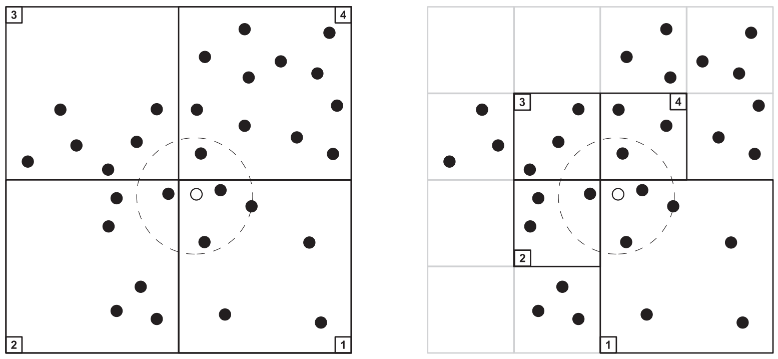

Fig. 2. Uniform-based partitioning (Grid) vs. Non-uniform-based partitioning (Quadtree) in SpatialHadoop.

k-th closest pair of the joined datasets, called β, and thus reduce the number of possible combinations of pairs of partitions. This β can be computed by global sampling both datasets or by local sampling for an appropriate pair of partitions, and then executing a plane-sweep kCPQ algorithm over both samples. The *filter* function input is each combination of pairs of cells in which the input set of points are partitioned and distance β, and it prunes pairs of cells which have minimum distances larger than β.

The εDJQ method in MapReduce, adapted from kCPQ in SpatialHadoop [18,19], adopts the *map* phase of the join MapReduce methodology (i.e., it is a Map-based Join algorithm), as shown in Fig. 1. The idea is to partition $\mathbb{P}$ and $\mathbb{Q}$ by any method (e.g., Grid) into two sets of cells, with n and m point cells, respectively. Then, every possible pair of cells is sent as input to the *filter* function. This function's input is combinations of cell pairs with partitioned point sets and a distance threshold ε, and it prunes pairs of cells which have minimum distances larger than ε. By using SpatialHadoop's built-in function *MinDistance*, the minimum distance between two cells can be calculated (i.e., this function computes the minimum distance between the two MBRs, Minimum Bounding Rectangles, of the two cells). In the *map* phase, each *mapper* reads the points of a pair of filtered cells and performs a plane-sweep εDJQ algorithm [33] (variation of the plane-sweep-based kCPQ algorithm) between the points inside that pair of cells. The results of all *mappers* are simply combined in the *reduce* phase and written into HDFS files, storing only the pairs of points with distances up to ε. In this case, the *filter* function input, as for kCPQ, is combinations of cell pairs with partitioned points sets and a distance threshold ε, and it prunes pairs of cells which have minimum distances larger than ε.

With respect to kNNJQ MapReduce algorithms, a method for classifying multidimensional data using a kNNJ algorithm in the MapReduce framework is presented in [31], using space decomposition techniques for processing the classification procedure in a parallel and distributed manner. The proposed kNNJ algorithm for two datasets $\mathbb{P}$ and $\mathbb{Q}$ consists of a series of phases of MapReduce jobs: *information distribution* phase, *primitive computation* phase, *update lists* phase and *unify lists* phase. One last phase consists of classification of the multidimensional data, which is outside the scope of the *kNNJ* algorithm definition. In the *information distribution* phase, the $\mathbb{Q}$ dataset is uniformly partitioned and the number of elements from $\mathbb{P}$ that are within the partitions of $\mathbb{Q}$ are counted. Then, in the *primitive computation* phase, an initial response is provided by calculating the kNNQ for each point p_i in $\mathbb{P}$ with the points of $\mathbb{Q}$ in the partition where p_i is located. Once this phase is completed, these initial kNN lists must be refined for each point in $\mathbb{P}$ if fewer than k neighbors have been found or if there are nearby cells overlapping the distance to each k-th nearest neighbor. This refinement is done in the *update lists* phase where new non-final kNN lists are obtained. Finally, in the *unify lists* phase, all kNN lists from the previous phases are merged to provide the final answer.

Moutafis et al. [30] extended the work presented in [31]. The *information distribution* phase was implemented by Quadtrees with dataset sampling to capture the data distribution skewness, balance the load and free the end user from having to refine data partitioning parameters. The *primitive computation* phase employes plane-sweep to reduce distance calculations, and the *update lists* and *unify lists* phases are restructured to reduce network traffic. The modified algorithms are extended to also handle 3d data, implemented in Hadoop and compared to the performance of the algorithm in [31] for real datasets. Note that the aim of [30] was to efficiently compute kNNJQ on 2d and 3d data in plain Hadoop, without the 2d spatial data capabilities provided by systems like SpatialHadoop and LocationSpark (as in our study).

To adapt and implement the previous kNNJQ MapReduce algorithms in SpatialHadoop, we have made several extensions and improvements as described below:

1. The *information distribution* phase is implemented with the indexing methods provided by SpatialHadoop, non-uniform partitions such as STR, Quadtree, Hilbert, etc. can be used, with the improvements and particularities they offer. Fig. 2 illustrates how the same dataset is partitioned using a uniform-based partitioning technique like Grid (on the left) and a non-uniform-based partitioning technique like Quadtree (on the right), where the selected partitions are highlighted.
2. The *information distribution* phase is performed only once for each dataset and reused for further kNNJ queries.
3. SpatialHadoop indices are used in each of these phases to accelerate the partition processing.

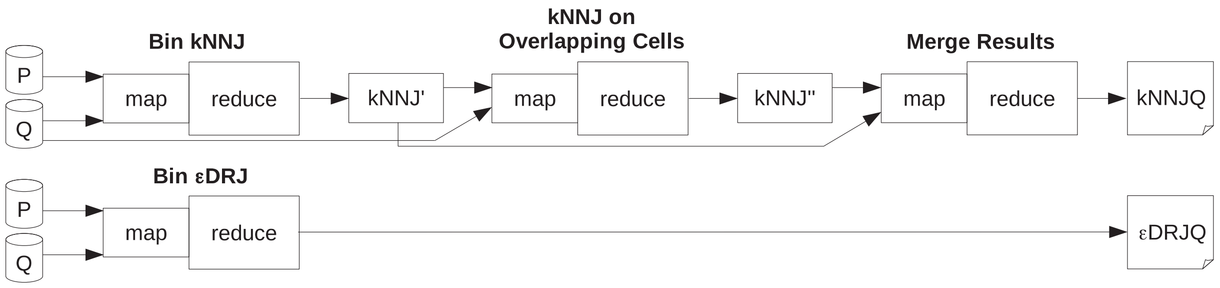

Fig. 3. Overview of the *k*NNJQ and *ε*DRJQ MapReduce algorithms in SpatialHadoop.

4. Finally, a *k*NNQ based on a plane-sweep algorithm is implemented, which reduces the number of operations and calculations, resulting in a join operation with higher performance.

Fig. 3 shows the phases in the *k*NNJQ MapReduce algorithm proposed: *Bin kNNJ, kNNJ on Overlapping Cells* and *Merge Results*. The first phase, called *Bin kNNJ*, consists of a Bin-Spatial Join of the input datasets in which the join operand is *k*NNQ. As described in Algorithm 1 for the *map* function in the *Bin kNNJ* phase, each point in $\mathbb{P}$ is combined with the partition in

Algorithm 1 Bin *k*NNJ algorithm.

```
1: function MAP($p$: point from $\mathbb{P}$ or $\mathbb{Q}$, $Cells$: set of partitions of $\mathbb{Q}$)
2:     $partitionId \leftarrow$ FINDPARTITION($Cells, p$)
3:     OUTPUT($partitionId, p$)

4: function REDUCE($partitionId$: current partition, $PQ$: set of points in partition, $k$: number of neighbors)
5:     $P \leftarrow$ GETPOINTSFROMP($PQ$)
6:     $Q \leftarrow$ GETPOINTSFROMQ($PQ$)
7:     for all $p \in P$ do
8:         INITIALIZE($kNNList, k$)
9:         $kNNList \leftarrow$ PS_KNN($p, Q, k$)
10:        OUTPUT($p, kNNList$)
```

dataset $\mathbb{Q}$ it is located in, so that, the plane-sweep *k*NNQ of that point with the points in $\mathbb{Q}$ in the same partition is executed in the *reduce* function. The result of this phase is a *k*NN list for each point in $\mathbb{P}$.

After that, completeness is checked to find previous *k*NN lists that are not final, and which therefore, must continue to be processed. As shown in Algorithm 2, in the *kNNJ on Overlapping Cells* phase, the *map* function checks whether the previous *k*NN lists contain less than *k* results for each point in $\mathbb{P}$ and also whether there are neighboring cells overlapping with the circular range centered on *p* and with radius the distance to the current *k*-th nearest neighbor. These points are then sent together with the neighboring cells calculated to the *reduce* phase where another plane-sweep *k*NNQ is performed on each cell.

Finally, the *Merge Results* phase consists of collecting the non-final *k*NN lists from the two previous phases, for the final *k*NNQ results for each point.

Note that just as the *ε*DJQ can be formulated and implemented as a derivative of *k*CPQ, in which the pruning distance *ε* is known, we can also define the *ε*DRJQ based on the *k*NNJQ algorithm by means of a Reduce-based Join algorithm, as shown in Fig. 3. Of the three phases discussed above, as the *ε* distance is known, the *Bin kNNJ* and *kNNJ on Overlapping Cells* phases are combined into just one, and since *k*NN lists do not have to be unified, the last phase need not be performed.

4.2. DJQ algorithms in LocationSpark

LocationSpark supports four types of spatial query predicates [36,37]: Spatial range search, *k*NN search, spatial range join and *k*NN join. The spatial range search is a generic spatial query involving one dataset and a spatial range area (e.g., rectangle or circle). The *k*NN search consists of three steps similar to the approach implemented in SpatialHadoop. First, the partition to which the query point belongs is located, and *k*NN in that partition are calculated. Next, a range search is carried out on the overlapping partitions by the circle region centered at the query point with radius the distance to the *k*-th nearest neighbor. Finally, the points in the query range are combined with the initial *k*NN result for the final result.

There are two algorithms in LocationSpark [36] for the spatial range join. The first is an indexed nested-loop join, where the spatial index from the largest dataset (points) is repeatedly traversed by a range query for each item from the smallest dataset (query points). Note that it is the naive version of *ε*DRJQ algorithm in LocationSpark. The second is a block-based

Algorithm 2 *k*NNJ on overlapping cells algorithm.

```
1:  function MAP($p$: point from $\mathbb{P}$ or $\mathbb{Q}$, $Cells$: set of partitions of $\mathbb{Q}$, $k$: number of neighbors)
2:      $origin \leftarrow$ IsFromPorQ($p$)
3:      if $origin$ is from $Q$ then
4:          $partitionId \leftarrow$ FindPartition($Cells$, $p$)
5:          output($partitionId$, $p$)
6:      else
7:          $kNNList \leftarrow$ GetKnnList($p$)
8:          $nnNumber \leftarrow kNNList.size$
9:          $radius \leftarrow$ GetKthDistance($kNNList$)
10:         while $nnNumber < k$ do
11:             $radius \leftarrow$ Increase($radius$)
12:             $nnNumber \leftarrow$ GetNumberOfNeighbors($Cells$, $p$, $radius$)
13:         $overlappedCells \leftarrow$ RangeQuery($Cells$, $p$, $radius$)
14:         for all $cell \in overlappedCells$ do
15:             output($cell.id$, $p$)

16: function REDUCE($partitionId$: current partition, $PQ$: set of points in partition, $k$: number of neighbors)
17:     $P \leftarrow$ GetPointsFromP($PQ$)
18:     $Q \leftarrow$ GetPointsFromQ($PQ$)
19:     for all $p \in P$ do
20:         Initialize($kNNList$, $k$)
21:         $kNNList \leftarrow$ PS_KNN($p$, $Q$, $k$)
22:         output($p$, $kNNList$)
```

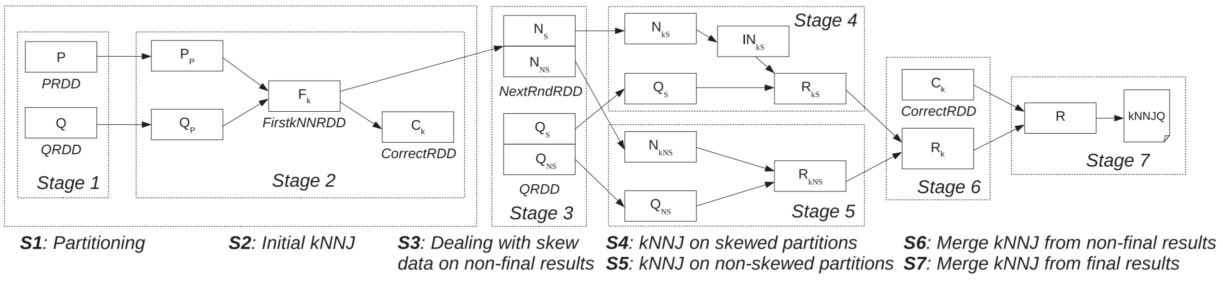

Fig. 4. Execution Plan for *k*NNJQ in LocationSpark.

algorithm using a parallel tree traversal, i.e., it builds two spatial indexes over both the input datasets and performs a depth-first search over both trees simultaneously. This algorithm is the first approach of εDRJQ algorithm in LocationSpark. The query execution plan for the spatial range join is shown in [36]. It should be noted that the execution plan of the spatial range join treats the partitions that can cause skew separately by repartitioning the latter to obtain better performance and because of that, a merge step is necessary to unify the results.

Like the spatial range join, indexed nested-loops and block-based algorithms can be applied to *k*NNJQ. For the second case, the algorithm partitions both datasets (query points and points) in two different set of blocks and finds the *k*NN candidates for query points in the same block. Then a post-processing refine step computes *k*NN points for each query point in the same block. As shown in Fig. 4, the *Execution Plan* of *k*NNJQ follows a scheme in LocationSpark similar to the one presented in [31] and the spatial range join algorithm discussed above. In Stage 1, the two datasets $\mathbb{P}$ and $\mathbb{Q}$ are partitioned according to a given spatial index method. In Stage 2, the initial *k*NN lists (*FirstkNNRDD*) are calculated, using a nested loop-based algorithm on Quadtrees, for each point in the partitions where it is located. Stage 3 collects those points where a final answer has not been obtained (*NextRndRDD*) and then a spatial range join is performed in Stages 4 and 5 using the distance to the *k*-th nearest neighbor. The above last two stages are necessary because the spatial range join treats the skewed partitions in parallel, and are therefore repartitioned from those that are not. Finally, in Stage 7, the results of the correct initial *k*NNs lists (*CorrectRDD*) are merged with the distances to the points from the range join obtained in Stage 6.

Assuming that $\mathbb{P}$ is the largest dataset to be combined and $\mathbb{Q}$ is the smallest, and according to [36], the *Execution Plan* for *k*CPQ in LocationSpark, (see Fig. 5) can be described as follows: In Stage 1, the two datasets are partitioned according to a given spatial index schema. In Stage 2, statistical data are added to each partition, $S_{\mathbb{P}}$ and $S_{\mathbb{Q}}$, which are combined by pairs, $S_{\mathbb{PQ}}$. In Stage 3, the partitions from $\mathbb{P}$ and $\mathbb{Q}$ with the largest point density, $\mathbb{P}_\beta$ and $\mathbb{Q}_\beta$, are selected for combination by a plane-sweep *k*CPQ algorithm [33] to compute an upper bound of the distance from the *k*-th closest pair (β). In Stage 4, the

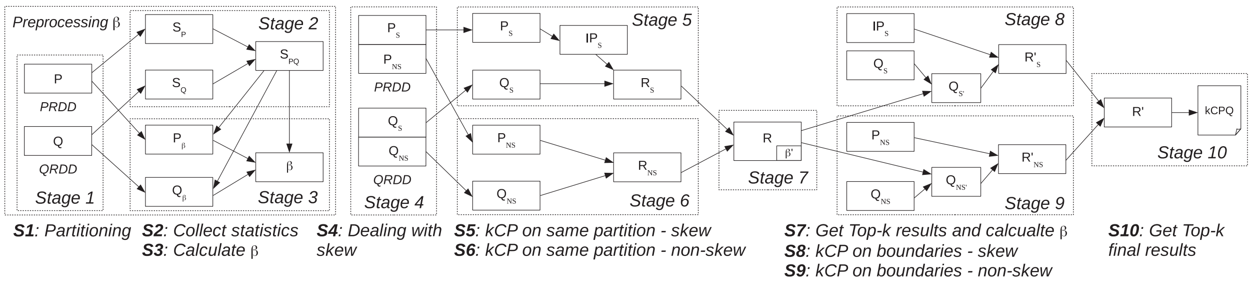

Fig. 5. Execution Plan for *k*CPQ in LocationSpark.

combination of all possible pairs of partitions from $\mathbb{P}$ and $\mathbb{Q}$, $S_{\mathbb{PQ}}$, is filtered by its β value (i.e., only the pairs of partitions with minimum distance between the MBRs of the partitions is smaller than or equal to β are selected), producing $FS_{\mathbb{PQ}}$, and all pairs of filtered partitions are processed by using a plane-sweep *k*CPQ algorithm. Finally, the results are merged for the final output. With the above *Execution Plan* and by increasing the size of the datasets, the execution time increases considerably due to skew and shuffle problems. We have therefore modified Stage 4 with the query plan that is used for the algorithms shown in [36], which leaves the plan shown in Fig. 5.

Stages 1, 2 and 3 are still used to calculate the β value which will accelerate local pruning of each partition. In Stage 4, using the *Query Plan Scheduler*, $\mathbb{P}$ is partitioned into $\mathbb{P}_{\mathbb{S}}$ and $\mathbb{P}_{\mathbb{NS}}$ which are the partitions that are or are not skewed, respectively. The same partitioning is used for $\mathbb{Q}$. In Stage 5, a *k*CPQ algorithm [33] is applied between points of $\mathbb{P}_{\mathbb{S}}$ and $\mathbb{Q}_{\mathbb{S}}$ within the same partition and also $\mathbb{P}_{\mathbb{NS}}$ and $\mathbb{Q}_{\mathbb{NS}}$ in Stage 6. These two stages are executed independently, and the results are combined in Stage 7. However, any candidate for each partition present on the boundaries of that same partition in the other dataset must still be calculate. To do this, β', which is the maximum distance from the current set of candidates as a radius of a range filter with center in each partition is used to find any possible new candidates on those boundaries. The calculation of *k*CPQ in each partition with its candidates is executed in Stages 8 and 9 and these results are combined in Stage 10 for the final answer. With these changes, the execution plan of *k*CPQ is very similar to the one in *k*NNJQ, where the only difference is that instead of maintaining different *k*NN lists for each point, only a single global *k*NN list is maintained.

The *Execution Plan* for εDJQ in LocationSpark is a variation of the one for *k*CPQ, where the filtering stages (Stages 1, 2, 3 and 7) are removed, since $S_{\mathbb{PQ}}$ is filtered by ε (i.e., $\beta = \beta' = \varepsilon$), which is the threshold distance known beforehand. The εDRJQ execution plan is a simplification of the one for *k*NNJQ, in which again a previously known limit ε is used. However, as mentioned above, the *k*CPQ and *k*NNJQ algorithms follow a similar scheme, so it can be assumed that the execution plans of εDJQ and εDRJQ will not be any different in LocationSpark, since they are simplified versions of the previous algorithms.

5. Extensions and improvements in the DJQ MapReduce algorithms

When extending and improving the DJQ MapReduce algorithms, several factors taking into account the different characteristics of real-world spatial objects and the DSDMSs execution environment must be considered. Therefore, the following factors [41] must be analyzed for better algorithms with optimal performance:

F.1 *Spatial Objects*. They are the smallest unit of non-divisible / non-splittable information (e.g., points, line-segments, polygons, regions, etc.).

F.2 *Spatial Location*. Normally, instead of using a complex geometry, exactly describing the spatial object, an approximation is used (e.g., center, centroid, MBR, etc.).

F.3 *Spatial Distribution*. By the nature of spatial objects, they usually show localization patterns that tend to show skew. In addition, adjacent objects must be partitioned in the same blocks as much as possible, while seeking a balance that reduces skew problems.

F.4 *Object Volume*. Size of the object in the physical storage layer (bytes).

F.5 *Block Size*. It determines when a block of data in HDFS is subdivided or merged (e.g., the default value for Hadoop 2 is 128 MB).

In this section, first we explain the extensions of DJQ MapReduce algorithms for managing other geometric objects different from points, and then, we present improvements to the distributed algorithms to deal with the problems that arise when there are too many objects inside a particular partition, i.e., the treatment of skewed data.

5.1. Extensions of the DJQ MapReduce algorithms for processing complex spatial objects

Real-world datasets are usually not limited only to points, but include other geometric objects, such as line-segments, polygons, regions, etc. For instance, a dataset containing the buildings in a city may use polygons, while line-segments may be used to represent roads. Because of this, the distributed algorithms presented above must be extended to be able to process datasets consisting of more complex spatial objects (**F.1**).

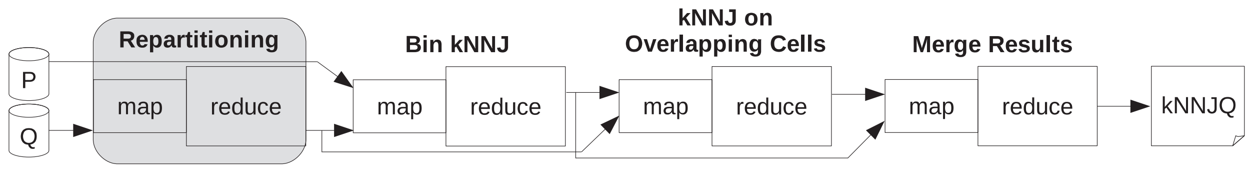

Fig. 6. Repartitioning phase in the *k*NNJQ MapReduce algorithm in SpatialHadoop.

When extending the algorithms, each of the steps that comprise them must be modified. First, recall that the replication method (Grid and STR+) in SpatialHadoop avoids expanding partitions by replicating each record to all overlapping partitions. Therefore, the query processor has to employ a duplicate avoidance technique to account for replicated records. In our approach, we used the *reference-point duplicate avoidance technique* [15], which consists of selecting a single point in the geometry and discarding the partitions in which the point is not found to avoid duplicates. Furthermore, to simplify algoirthm distance operations and calculations, the MBR (Minimum Bounding Rectangle) covering the different spatial objects is used. Using the MBR instead of the exact geometrical representation of the spatial object, its representational complexity is reduced to two points (i.e., *min* and *max*), where the most important object features (position and extension) are maintained (**F.2**). The MBR is therefore a widely employed approximation. This way, the plane-sweep algorithms only have to calculate the minimum distance between MBRs without computing complex calculations based on their shapes (e.g., calculate the distance between a convex polygon and a line-segment). This is commonly known as the *filtering step*, since it finds all MBRs of spatial objects that verify the query condition. The processing of the exact geometry of the spatial objects is only required in the final phase for obtaining the exact distance values. This is commonly known as the *refinement step*, and an efficient computational geometry algorithm [12] is needed to produce the final result (e.g., algorithm to compute the distance between two convex polygons).

5.2. Improvements for processing skewed data

A problem that usually appears in MapReduce tasks is so-called *skewed data*. In general, the problem is that some partitions have more elements than the rest, and therefore, some tasks take a long time to be executed and the final result may be delayed (**F.3**). Furthermore, the partitioning techniques in SpatialHadoop and other systems are usually based on making partitions close to the underlying file-system block size (**F.5**) established in the corresponding big data cluster. However, DJQ MapReduce algorithms, like *k*NNJ, may produce combinations of partitions with a very large number of elements that would delay results and increase main memory used (**F.4**).

The purpose of the proposed improvements is to repartition each local partition from a set of data already partitioned by SpatialHadoop (e.g., Grid, Quadtree) as necessary to solve the abovementioned problem. A kind of double index is created for this from the original global index plus a subindex for each of the partitions when a certain number of elements is exceeded, and they need to be repartitioned. For instance, a dataset may be partitioned by Quadtree in 12 partitions and then each partition is split into a Grid of 4×4 partitions. To create this index, only Factors **F.1, F.2, F.3** and **F.4** must be taken into account, since SpatialHadoop also considers Factor **F.5** for the initial partitions, and because the resulting partitions are not saved as new HDFS files. This *repartitioning technique* is used mainly for *k*NNJQ and εDRJQ in SpatialHadoop, although it can be applied to other distributed DJQ algorithms and DSDMSs. Fig. 6 shows the new phase (*Repartitioning*) of the proposed *k*NNJQ MapReduce algorithm in SpatialHadoop. The *Repartitioning* phase uses an existing partitioning technique to subdivide the largest partitions from dataset $\mathbb{Q}$ and saves the information for further use in subsequent phases. Note that there is no repartitioning in [30], since the Quadtree-based partitioning in Hadoop is completely under the control of the [30] *k*NNJQ algorithm and is not limited by the file-system block size (unlike the partitioning techniques provided by SpatialHadoop).

In our proposal, we implemented two types of repartitioning techniques. One is a *Grid repartitioning* method based on a maximum number of elements L, which divides the original partition into as many *rows* and *columns* as necessary so that each cell has at most L elements. Our experiments used $num\_rows = num\_columns = \sqrt{(\frac{num\_elements}{L})}$, where *num_elements* is the number of elements in the original partition and $L = 50000$. For *k*NNJQ, this repartitioning is done in the *Bin Join* phase described in Algorithm 2 and illustrated in Fig. 3, where, during the *map* phase, the elements of both sets are distributed based on a formula that determines the new partition they belong to. This is the great advantage of Grid partitioning since no previous preprocessing is necessary to divide a partition into a certain number of rows and columns (i.e., sub-partitions). Then, in the *reduce* phase, the elements in the largest set in each sub-partition created are counted. This way, the next phase can use the index recently created to obtain the partitions that overlap with the partial results. These sub-partitions are smaller than the original partition and therefore candidates from calculations of *k*NNJQ will be pruned. However, even if a limit has been set on elements, it is impossible to know if any of the sub-partitions will exceed it, since it is unknown a priori how the elements are distributed.

The second type designed and implemented is a *Quadtree-based repartitioning* technique. As this repartitioning method is based on how the data is distributed, the simple formula that splits the partition into *rows* $\times$ *columns* used for Grid repartitioning is unnecessary. However, a new task must be performed. As shown in Algorithm 3, this is a MapReduce job,

Algorithm 3 Quadtree based repartitioning algorithm.

```
1: function MAP($p$: point from $\mathbb{Q}$, $Cells$: set of partitions of $\mathbb{Q}$, $r$: sample ratio)
2:     $partitionId \leftarrow$ FINDPARTITION($Cells$, $p$)
3:     if RANDOM $\leq r$ then
4:         OUTPUT($partitionId$, $p$)

5: function REDUCE($partitionId$: current partition, $Q$: set of points in partition, $L$: max number of elements, $r$: sample ratio)
6:     INITIALIZE($Quadtree$, $L \times r$)
7:     for all $q \in Q$ do
8:         INSERTINTO($Quadtree$, $q$)
9:     OUTPUT($partitionId$, $Quadtree$)
```

which repartitions each of the partitions in the initial index. It uses a maximum number of elements L and a data sampling process in the *map* phase to find a representative element distribution set (r is a sample ratio), which reduces the creation time of the following Quadtree. As mentioned below in the experimental section, $L = 100000$ and $r = 2\%$ in our experiments. Finally, in the *reduce* phase, it inserts the sampled elements in a Quadtree per partition that will form part of the new sub-index. Once the repartitioning is done, the algorithm behaves in the same way as for the repartitioning based on Grid. In Algorithm 4, the new range query method *RangeQueryWithRePartitioning* for selecting cells/partitions overlapping with the

Algorithm 4 RangeQuery with repartitioning algorithm.

```
1: function RANGEQUERYWITHREPARTITIONING($circle$: circular region, $Cells$: set of partitions of $\mathbb{Q}$, $Quadtrees$: set of quadtrees
   for each $Cells$)
2:     INITIALIZE($SelectedCells$)
3:     for all $c \in Cells$ do
4:         if INTERSECTS($circle$, $c$) then
5:             INSERTINTO($SelectedCells$, $c$)
6:     INITIALIZE($SelectedSubCells$)
7:     for all $c \in SelectedCells$ do
8:         $Quadtree \leftarrow$ FINDQUADTREE($Quadtrees$, $c$)
9:         $SubCells \leftarrow$ INTERSECTS($Quadtree$, $circle$)
10:        INSERTINTO($SelectedSubCells$, $SubCells$)
11:    return SelectedSubCells
```

circular region centered on query point q and with a radius equal to the distance threshold δ is shown. At first, the global index is used to select those cells that overlap with the region, and then the corresponding Quadtree is used to obtain the sub-cells/sub-partitions that really overlap with it. Note that in this way, more candidates are pruned and therefore the search spatial dataset is also reduced.

6. Experimental results

This section presents the results of an extensive experimental study measuring and evaluating the efficiency of the algorithms and improvements proposed in Sections 4 and 5. Section 6.1 describes the experimental settings, Section 6.2 reports on all the experiments related to kCPQ and εDJQ, taking into account parameters, such as the spatial partitioning techniques included in SpatialHadoop. Section 6.3 shows all experiments for kNNJQ, with special attention to the results in SpatialHadoop, where several phases are necessary to perform this DJQ, and repartitioning techniques are used to reduce the execution time. Section 6.4 compares εDRJQ in both DSDMSs, regarding scalability of the datasets to be combined and ε. Section 6.5 shows the speedup of the proposed DJQ MapReduce algorithms, varying the number of computing nodes in the cluster. Finally, in Section 6.6, the experimental results are summarized.

6.1. Experimental setup

For the experimental evaluation, we used the SpatialHadoop[3] and LocationSpark[4] implementations with the addition of our open-source DJQ MapReduce algorithms, which can be downloaded from https://github.com/acgtic211/spatialhadoop2/tree/DJQ and https://github.com/acgtic211/SpatialSpark/tree/DJQ. We used real-world 2d points and geometric datasets to test our DJQ MapReduce algorithms in SpatialHadoop and LocationSpark. We did not use synthetic data, because the most

[3] Available at https://github.com/aseldawy/spatialhadoop2 .
[4] Available at https://github.com/merlintang/SpatialSpark .

Table 3
Configuration parameters used in our experiments.

Parameter	Values (default)
k for kCPQ	1, 10, (10^2), 10^3, 10^4, 10^5
k for kNNJQ	1, (10), 25, 50, 75, 100
ε ($\times 10^{-5}$)	7.5, 10, 25, 50, 75, (100)
Number of nodes	1, 2, 4, 6, 8, 10, (12)
Partitioning tech.	Grid, Str, (Quadtree), Hilbert
Repartitioning tech.	None, Grid, (Quadtree)
DSDMS	SpatialHadoop, LocationSpark

representative results and conclusions for this kind of experiment have been found from real data, as seen in [19]. The real-world datasets we used were five medium/large/big spatial datasets from OpenStreetMap[5]:

- *LAKES* (*L*) which contains 8.4M records (8.6GB) of boundaries of water areas (represented as polygons),
- *PARKS* (*P*) which contains 10M records (9.3GB) of boundaries of parks or green areas (represented as polygons),
- *ROADS* (*R*) which contains 72M records (24GB) of roads and streets around the world (represented as line-strings),
- *BUILDINGS* (*B*) which contains 115M records (26GB) of boundaries of all buildings (represented as polygons), and
- *ROAD_NETWORKS* (*RN*) which contains 717M records (137GB) of road network represented as individual road segments (represented as line-strings) [15].

To create sets of points from these five spatial datasets, we transformed the MBRs of line-strings into points by taking the center of each MBR. In particular, we considered the centroid of each polygon to generate individual points for each kind of spatial object.

To study the performance of the DJQ MapReduce algorithms with two datasets, we experimented using the spatial datasets above and the most representative spatial partitioning techniques (*Grid, STR, Quadtree* and *Hilbert*) provided by SpatialHadoop, according to [14,19]. Note that *STR* is equivalent to *STR+* for points. To test the improvements related to the use of repartitioning techniques (*Grid* and *Quadtree*), the experiments used datasets previously partitioned by Quadtree and where these techniques were applied later.

All experiments were conducted on our in-house cluster of 12 nodes on an OpenStack environment. Each node has 4 vCPU with 8GB 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 main measure of performance in our experiments was the total *execution time* (i.e., running time or response time) in seconds (s), and represents the time spent for the execution of each distributed DJQ algorithm in both DSDMSs (SpatialHadoop and LocationSpark). *Shuffled data*, the amount of information produced in the *mapper* tasks and moved to the nodes where the *reducer* tasks will run, shown in Gigabytes (GB), was also used as a performance metric in our experiments to acquire more information on the behavior of the different phases of kNNJQ in SpatialHadoop.

Table 3 summarizes the configuration parameters used in our experiments. Default parameters (in parentheses) are used unless otherwise mentioned. SpatialHadoop requires the datasets to be partitioned and indexed before invoking any spatial operations. For instance, the times needed for the *Preprocessing* phase using a *Quadtree* partitioning technique are 94 s for *LAKES*, 103 s for *PARKS*, 150 s for *ROADS*, 175 s for *BUILDINGS* and 1053 s for *ROAD_NETWORKS*. Data are indexed and stored on HDFS and for the subsequent execution of spatial queries, data and index are already available (this can be considered an advantage of SpatialHadoop). On the other hand, LocationSpark (in-memory-based DSDMS) partitions and indexes the data using a *Quadtree* partitioning technique for every spatial query and only caches the result in memory for that current operation. Therefore, the partitions/indexes are not stored in any persistent file system and cannot be reused in subsequent spatial operations.

6.2. k$CPQ and εDJQ experiments

Our first set of experiments measured the behavior of the kCPQ and εDJQ algorithms in both DSDMSs, varying different parameters, such as dataset size, type of spatial object, partitioning technique in SpatialHadoop and the k and ε values. In Fig. 7, the chart on the left shows the $kCP(\mathbb{P}, \mathbb{Q}, k)$ for point datasets (where $\mathbb{P} \times \mathbb{Q} \equiv$ *LAKES* $\times$ *PARKS* ($L \times P$), *PARKS* $\times$ *ROADS* ($P \times R$), *ROADS* $\times$ *BUILDINGS* ($R \times B$) and *BUILDINGS* $\times$ *ROAD_NETWORKS* ($B \times RN$)) respect to the execution time for a fixed $k = 100$. The first conclusion is that the execution times in both DSDMSs (SpatialHadoop and LocationSpark) grow as dataset size increases. For SpatialHadoop, the best partitioning technique was *Quadtree*, which was approximately 15% faster than *STR*. Moreover, for the combinations of *LxP* and *PxR*, LocationSpark was faster than SpatialHadoop (e.g., for $P \times R$ LocationSpark was 48% (74 s) faster than SpatialHadoop-Quadtree), but for the combinations of the biggest datasets ($R \times B$ and $B \times RN$) SpatialHadoop-Quadtree was the fastest, e.g., for $B \times RN$ SpatialHadoop-Quadtree was 38% (740 s) faster than LocationSpark and 12% (174 s) faster than SpatialHadoop-STR. That is, LocationSpark runtime values were smaller for medium-

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

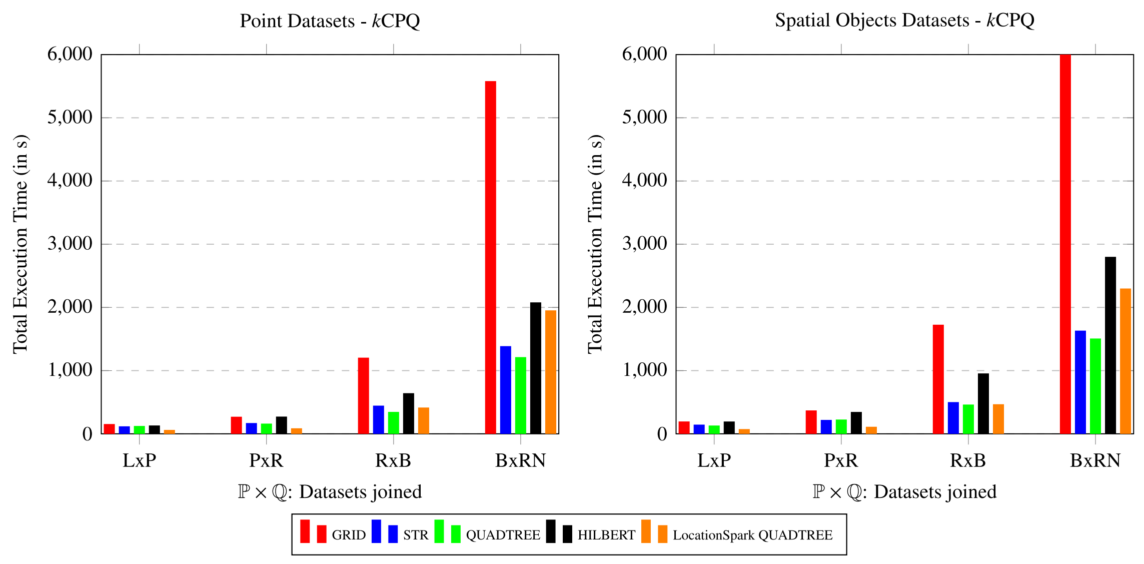

Fig. 7. *k*CPQ cost considering different partitioning techniques.

to-large dataset sizes, despite the fact that they were neither pre-partitioned nor pre-indexed. But for big dataset sizes, SpatialHadoop-Quadtree was the fastest, even though it required a pre-indexing time for each dataset, and that difference may have been because of memory constraints in the cluster. By increasing the size of the joined datasets, there is more data in each partition, and the *memory pressure* of the tasks increases. It may therefore be concluded that LocationSpark is more affected by memory constraints than SpatialHadoop for the same cluster.

In Fig. 7, chart on the right shows the *k*CPQ for real spatial object datasets (*LxP: polygons* $\times$ *polygons, P* $\times$ *R: polygons* $\times$ *line-strings, RxB: line-strings* $\times$ *polygons* and $B \times RN$*: polygons* $\times$ *line-strings*) with respect to the execution time. A trend similar to the chart on the left (for points) is observed for the combination of medium-to-large dataset sizes. LocationSpark was the fastest, but *Quadtree* was the fastest partitioning technique for the biggest dataset combinations (it was slightly better than *STR*), and the *Grid* was the slowest. In SpatialHadoop, *Quadtree* outperformed all other partitioning techniques with respect to the running time, since it minimizes the number of overlapping partitions between the two files for a DJQ by employing regular space partitioning. Moreover, comparing both charts in Fig. 7, it may be seen that when a *k*CPQ is executed between two datasets of spatial objects, it is more costly than when the two datasets are points, although the trend is very similar. This is because computation of the distance between two spatial objects (e.g., between two polygons or between a polygon and a line-string) is more costly than to calculate the distance between a pair of points. It should also be borne in mind that the size of the datasets of spatial objects is larger than the size of point datasets, and are therefore more costly to retrieve and process. In addition, the distances between spatial objects are much smaller because some objects occupy a certain area with respect to the centroids that do not have any. This reduction in the distance values between spatial objects produces a smaller pruning bound by the plane-sweep *k*CPQ algorithm which discards fewer elements in each step of the algorithm.

Fig. 8 shows the $\varepsilon DJ(\mathbb{P}, \mathbb{Q}, \varepsilon)$ execution times with the same configuration as in Fig. 7 for a fixed $\varepsilon = 0.001$ (100×10^{-5}). As for *k*CPQ, the choice of partitioning technique in SpatialHadoop clearly affected the εDJQ execution time, and again *Quadtree* performance was the best for point datasets (slightly better than *STR*), even with respect to LocationSpark for medium-to-large dataset sizes, as seen in the chart on the left. For the combinations of the biggest datasets ($R \times B$ and $B \times RN$) SpatialHadoop-Quadtree was the fastest, e.g., for $B \times RN$ SpatialHadoop-Quadtree was 2.8 times (1938 s) faster than LocationSpark and 8% (91 s) faster than SpatialHadoop-STR. *Memory pressure* problems in LocationSpark appeared again due to the dataset sizes, the number of elements computed in each partition and the shuffling data and garbage collection processes performed on them. On the other hand, SpatialHadoop performed better for this problem due to the use of *CombineFileSplits* [22], which enables joins by partitioning at disk reading level, and therefore, eliminating the *reduce* shuffling cost.

In the chart on the right in Fig. 8, the results of εDJQ for real spatial object datasets are shown with respect to execution time ($\varepsilon = 0.001$). The trend is similar to the left-hand chart, where the *STR* partitioning technique was the fastest in all cases (slightly better than *Quadtree*, except for $B \times RN$), and again the Grid was the slowest. For example, *Quadtree* was 9% (100 s) faster than STR in the combination of the biggest datasets. It may therefore be concluded that the bigger the datasets, the better the performance of *Quadtree* for εDJQ. A comparison of the two charts in Fig. 8 shows that when a εDJQ is executed between two datasets of spatial objects, it is more costly than when the two datasets are points (the same as for *k*CPQ), although the trend is very similar. This is because calculation of the distance between spatial objects is more costly than the distance between points, and because, just as above for *k*CPQ, the distances between spatial objects are smaller, returning

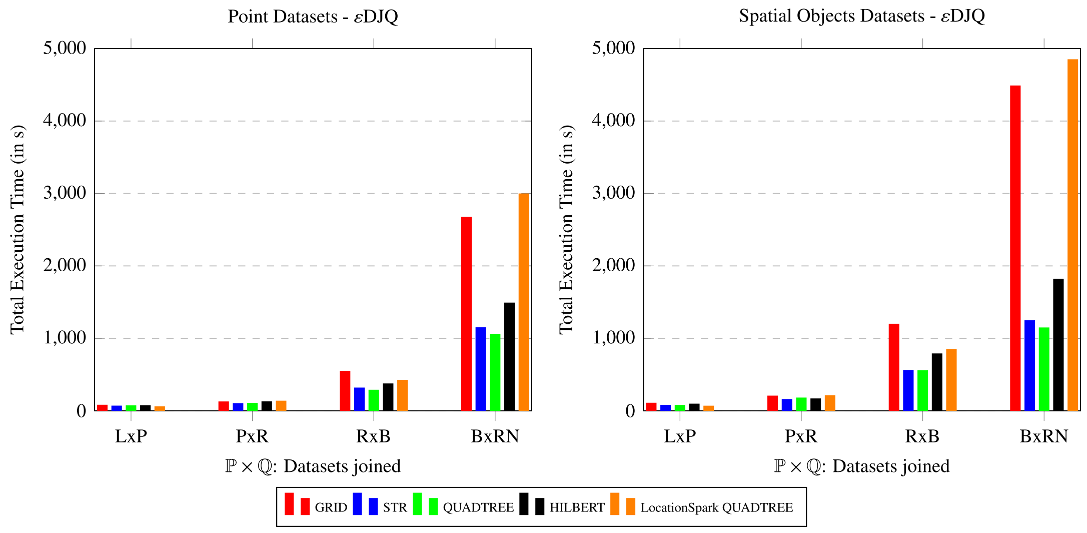

Fig. 8. εDJQ cost considering different partitioning techniques.

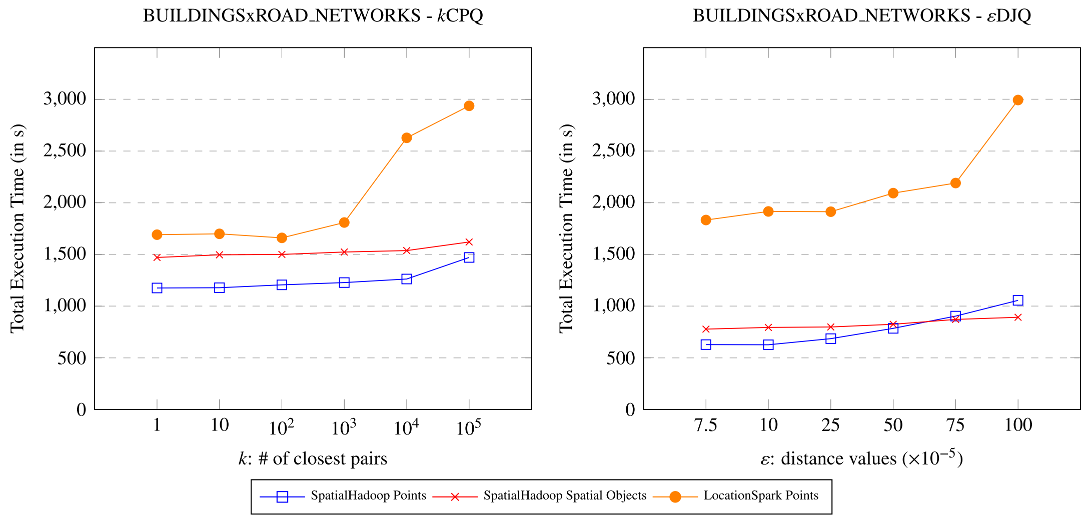

Fig. 9. kCPQ cost (execution time) vs. k values (left). εDJQ cost (execution time) vs. ε values (right).

more results for the same ε. For the combination of the largest datasets ($B \times RN$), LocationSpark again showed *memory pressure* problems, and the execution time was also higher because the size of the datasets for spatial objects is larger than for points.

Fig. 9 shows the effect of increasing both k and ε values in the combination of the biggest datasets (*BUILDINGS* $\times$ *ROAD_NETWORKS*) for kCPQ and εDJQ. The chart on the left in Fig. 9 shows that the total execution time grew slowly as the number of results to be found (k) increased. SpatialHadoop, employing *Quadtree*, had very stable execution times, even for large k values (e.g., $k = 10^5$) and when the sets of spatial objects (*polygons* $\times$ *line-strings*) were joined. This means that the *Quadtree* is less affected by the increase of k, because *Quadtree* employs a regular space partitioning technique depending on the concentration/density of the points. LocationSpark is also stable when k is small or medium ($k \le 10^3$); however, when k is high ($k = 10^4$ and $k = 10^5$), the execution time is very long due to memory constraints in the cluster. As k increases, the possibility of selecting more cells is also greater, since the distance of the k-th nearest pair increases as well. Therefore, the number of resources needed by the kCPQ algorithm is increased. In Fig. 9 all algorithms show a deviation for the highest values of k, more evident in LocationSpark.

As shown in the chart on the right in Fig. 9, the total execution time grew as ε increased. Relative performance of both DSDMSs (SpatialHadoop and LocationSpark) was similar for all ε values, but SpatialHadoop was faster in all cases, even when the datasets of spatial objects were combined. This difference is due to the way εDJQ is calculated in SpatialHadoop and its pre-indexing phase, which reduces the time considerably even for very big datasets. At larger values, execution times started to increase in both systems, due to the increase in the number of elements in the results. A special case

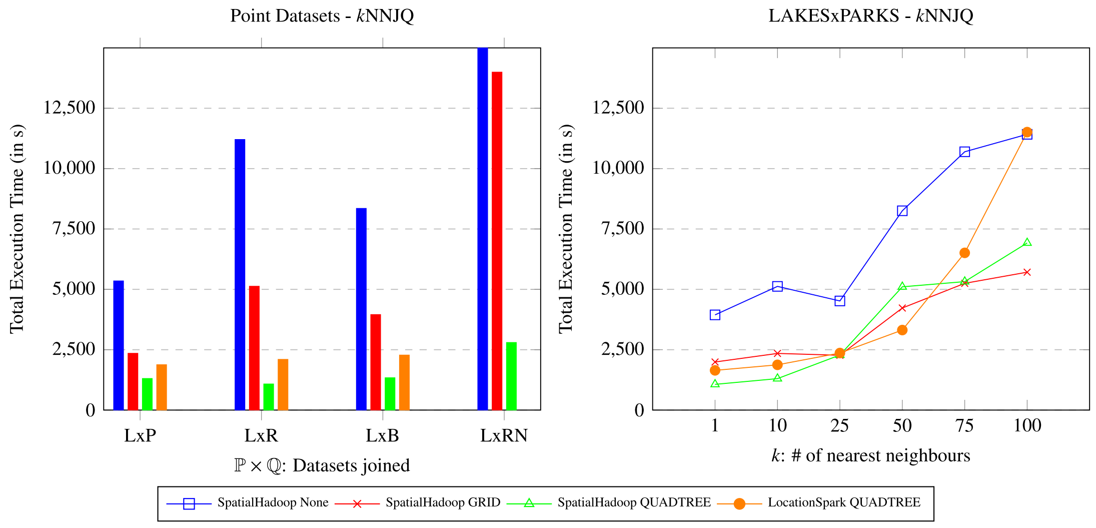

Fig. 10. *k*NNJQ cost (execution time) considering different datasets (left) and varying the *k* values (right).

is found in LocationSpark, as when $\varepsilon = 10^{-3}$, the execution time is very high, because it once again has more has more aggressive memory pressure problems than SpatialHadoop. If εDJQ behavior in joining points and spatial objects is compared in SpatialHadoop, it may be seen that when a εDJQ is executed between two point datasets, the execution time is smaller than when the two datasets are spatial objects for small and medium ε values. However, for higher ε values ($\varepsilon >= 75 \times 10^{-5}$) its performance is worse, even though calculation of the distance between spatial objects is more costly. That is because the result from *Quadtree* partitioning is different for each type, and for spatial objects it tends to create more partitions with fewer elements. Therefore, in this particular case, the workload is more balanced and there are fewer skewed data when dealing with spatial objects than with points for higher ε values.

The main conclusions that can be arrived at for this set of experiments are:

1. The higher the k or ε values, the higher the possibility that pairs of candidates are not pruned, more tasks are needed, and more total execution time is consumed.
2. SpatialHadoop performance is better, especially for higher k and ε values, due to *Quadtree* partitioning technique and the reduction in the number of candidates by the Preprocessing step.
3. The trend of SpatialHadoop is quite stable for the execution time, even if the biggest sets of spatial objects are combined, which is more costly than for points.
4. The SpatialHadoop εDJQ had the shortest and most stable execution times, demonstrating the benefits of its Map-based Join implementation and the use of *CombineFileSplits* [22].
5. LocationSpark is faster than SpatialHadoop with medium and large dataset sizes, but for big datasets it needs a longer time to execute the *k*CPQ and εDJQ, even for small k and ε values; this is due to *memory pressure* problems resulting from the increase in the number of elements processed and the size of memory consumed, as well as the increase in the processing time needed for the data shuffling and garbage collection.

6.3. *kNNJQ experiments*

Like *k*CPQ, the first set of experiments for *k*NNJQ algorithms measured the variation of different parameters, such as the dataset sizes to be joined (scalability), repartitioning techniques on SpatialHadoop (the global partitioning technique is always set to *Quadtree* due to the excellent results reported in all join operations [14,19]) and k values. Also, note that $L = 100000$ and $r = 2\%$ for Quadtree repartitioning and $L = 50000$ for the *Grid* repartitioning technique. In the chart on the left in Fig. 10, the $kNNJ(\mathbb{P}, \mathbb{Q}, k)$ query, where $\mathbb{P} = LAKES$ has been fixed as the smallest dataset and the others as $\mathbb{Q}$ (*PARKS, ROADS, BUILDINGS* and *ROAD_NETWORKS*, resulting in the following combinations: $L \times P$, $L \times R$, $L \times B$ and $L \times RN$) is shown with respect to the execution time, for a fixed $k = 10$. The most important conclusion that can be arrived at from this chart is that SpatialHadoop using *Quadtree* repartitioning technique is the fastest, next is *Grid*, whereas the worst alternative is not to use any repartitioning technique (mainly when the biggest datasets are joined). For example, for $L \times P$, *Quadtree* was 1.8 times faster than *Grid* and for $L \times RN$, *Quadtree* was 4.8 times faster. Another important result is that *Quadtree* repartitioning technique is quite stable with increase in size of $\mathbb{Q}$ dataset for fixed $k = 10$. For instance, from $L \times P$ to $L \times RN$ the increment was 53.4% (1491 s), when the increment of *ROAD_NETWORKS* (717M) with respect to *PARKS* (10M) was huge in terms of the number of points. Another conclusion is that both DSDMSs (SpatialHadoop-Quadtree and LocationSpark) are quite stable and the increment is sub-linear as the size of the datasets ($\mathbb{Q}$) grows, with around a 40% difference (note that due to the memory

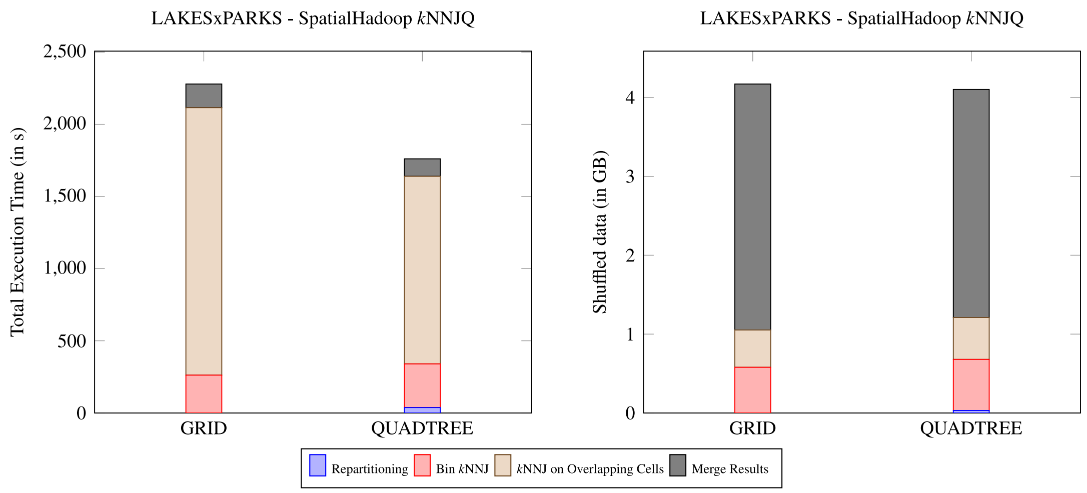

Fig. 11. *k*NNJQ cost per phase considering different repartitioning techniques on the combination of the smallest datasets. Execution time (left) and shuffled data in GBytes (right).

restrictions of the cluster, the *k*NNJQ could not be performed for the combination of $L \times RN$ in LocationSpark). This excellent behavior of SpatialHadoop with *Quadtree* repartitioning technique compared to LocationSpark is because its repartitioning technique deals with skewed data so well. While SpatialHadoop, with the new *Quadtree* repartitioning technique processes cells that exceed a certain number of elements, LocationSpark, in the current implementation, treats only the number of cells with the highest number of elements based on the input datasets. Note that for the same execution conditions in our cluster, LocationSpark could not execute the *k*NNJQ for the combination of $L \times RN$ because it consumed all available resources on some workers, and the execution had to be aborted.

The chart on the right in Fig. 10 shows the effect of increasing the value of k for the combined datasets ($LAKES \times PARKS$). We can observe that SpatialHadoop without any repartitioning technique had the worst performance, which means that the use of repartitioning techniques improves highly the performance for this DSDMS. Moreover, we can see that for small k values ($k \leq 25$), *Quadtree* is faster than *Grid*, but when k is large ($k > 25$) *Grid* takes a similar or shorter time to report the query result. This behavior is because the repartitioning techniques produce different types of partition subsets. In the case of *Grid* it is a uniform distribution where all cells are the same size, whereas for *Quadtree* it is a regular space partitioning technique based on the concentration of elements, and therefore, generates different sized cells. In an algorithm like *k*NNJQ, increase in k augments the possibility of selecting more partitions overlapping with the ranges of distances found in the *Bin kNN Join* phase. Therefore, the same point must be compared in more than one partition, increasing the size of shuffled data and having a partial list for each partition that must be combined in the last phase of the algorithm. These results suggest that as k increases, the number of overlapping partitions also increases to a greater extent and more suddenly for Quadtree repartitioning than for *Grid*. Finally, comparing the two DSDMSs, SpatialHadoop is observed to be faster than LocationSpark, except for $k = 50$, where LocationSpark is faster. At first sight, LocationSpark seems to scale better when increasing the value of k, but from $k = 75$ problems start to appear due to memory limitations in the cluster, because of the increase in the number of elements and partitions. It is again demonstrated that LocationSpark is more sensitive to this type of problems than SpatialHadoop, since it is a memory-based DSDMS.

The following experiment with the *k*NNJQ MapReduce algorithms compares the *Grid* and *Quadtree* repartitioning techniques in SpatialHadoop by evaluating the cost, in execution time and shuffled data, of each of the phases in these algorithms. In Fig. 11, in the chart on the left, the $kNNJ(\mathbb{P}, \mathbb{Q}, k)$ query for the combination of the $LAKES \times PARKS$ datasets is shown for each repartitioning technique and for a fixed $k = 10$. We can observe that SpatialHadoop with the *Quadtree* repartitioning technique had the best performance. *Grid* is much slower, especially in the *kNNJ on Overlapping Cells* phase. This is because the *Quadtree* technique partitions the data better, since it takes into account its distribution, so after the *Bin kNNJ* phase there are more final *k*NN lists, and therefore, the processing time for the next phase is shorter. The *kNNJ on Overlapping Cells* phase is usually more costly if the number of final *k*NN lists from the previous phase is smaller, because during the range query on the nearby cells, the number of partitions to be searched for *k*NN candidates grows. Finally, the execution time required for *Quadtree* repartitioning technique in the *Repartitioning* phase is very short (2% over the total time) compared to the saved time (28% faster than *Grid*).

The chart on the right in Fig. 11 shows the results of the same query and parameters as in the previous experiment, but with the amount of shuffling data exchanged in the different MapReduce phases of the *Grid* and *Quadtree* repartitioning techniques for *k*NNJQ in SpatialHadoop. On one hand, we can observe that the difference in the *Bin kNNJ* and *kNNJ on Overlapping Cells* phases was almost negligible. On the other hand, there were more shuffled data in the *Merge Results* phase for *Grid* than for *Quadtree*. This confirms that the *Grid* repartitioning technique generated more partial *k*NN lists, and

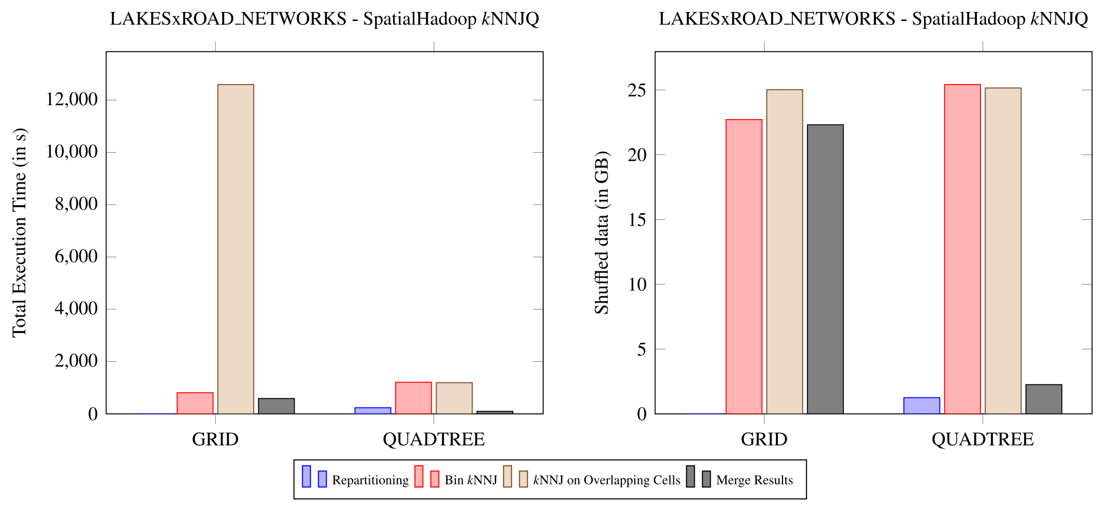

Fig. 12. *k*NNJQ cost per phase considering different repartitioning techniques on the combination with the biggest dataset. Execution time (left) and shuffled data in GBytes (right).

therefore, the *Merge Results* phase must process all of them for the final result. In addition, *Grid* has to process more data, and as a consequence, more time is spent in the *Merge Results* phase, as seen in Fig. 11, left-hand chart.

Continuing with the above experiment, Fig. 12 shows the $kNNJ(\mathbb{P}, \mathbb{Q}, k)$ query for the combination of the datasets ($LAKES \times ROAD\_NETWORKS$) for each repartitioning technique and for a fixed $k = 10$. The time used by the different phases of the algorithms is shown in the chart on the left in Fig. 12. The first conclusion would be that differences are greater than for the join with smaller datasets. The widest time difference between *Grid* and *Quadtree* appears in the *kNNJ on Overlapping Cells* phase (10 times slower). This is because the *Grid-based* repartitioning technique leaves fewer final *k*NN lists after the *Bin kNNJ* phase, and so the algorithm generates more cells than *Quadtree*, and therefore, requires more tasks. Moreover, *Grid* may have problems with skewed data because its uniform partitioning does not take into account the spatial distribution of the data, which could also generate cells that have many more elements. As a consequence of this increment in the number of partial results, the *Merge Results* phase also requires more time to find the final result of the query. Finally, in the *Repartitioning* phase, the execution time required by *Quadtree* repartitioning technique is barely 8.5% (234 s) over the total time, in comparison with the saved time (5 times faster than *Grid*). The right-hand chart in Fig. 12 shows the cost in shuffled data corresponding to the previous execution times. With the *Quadtree* repartitioning technique, there is a bit more shuffled data in the *Bin kNNJ* phase than with *Grid*, since there are more partitions. The following *kNNJ on Overlapping Cells* phase presents practically the same values, because despite having more final *k*NN lists, the data must be sent for the largest dataset, since it is unknown in advance whether they will be used in the *reduce* part of that phase. Finally, in the *Merge Results* phase, 9.8 times more information is exchanged with *Grid* than with *Quadtree*, since more *k*NN lists were generated in the previous phase.

The chart on the left in Fig. 13 shows the $kNNJ(\mathbb{P}, \mathbb{Q}, k)$ query executed for the combinations of $\mathbb{P} \times \mathbb{Q}$: $L \times P$, $L \times R$, $L \times B$ and $L \times RN$, and the shuffled data cost in Gigabytes for a fixed $k = 10$. The first conclusion is that the shuffled data for both techniques (*Grid* and *Quadtree*) grow as the size of the datasets increases. *Grid* values are a little higher than *Quadtree* for all the combinations of datasets, because it usually produces fewer final *k*NN lists for that fixed *k*. That is, with *Quadtree* shuffled data values are lower for all dataset sizes, despite pre-indexing in the *Repartitioning* phase. The right-hand chart in Fig. 13 shows the effect of the increment of *k* value for the combination of the $LAKES \times PARKS$ datasets. For small / medium *k* values ($k \leq 50$), the shuffled data cost is lower for *Quadtree* than *Grid*, but when *k* is large ($k > 50$) *Grid* exchanges fewer data to report the result of the query. As mentioned above, in an algorithm such as *k*NNJQ, as the value of *k* increases, the possibility that the number of overlapping partitions also increases, and thereby, the shuffled data size of the algorithm. This increment depends on the morphology of the underlying partitioning technique. In *Grid*, partitioning is uniform, and all the partitions are the same size and shape, as shown in the right-hand chart in Fig. 13, since its values are more stable. *Quadtree* presents sharper changes because this repartitioning technique is not uniform, partitions have different sizes and shapes, and therefore, when the distances increase in the range queries, the number of selected partitions does not increase uniformly.

The main conclusions extracted from this set of experiments on the proposed *k*NNJQ MapReduce algorithm based on phases (see Fig. 3) are the following:

1. SpatialHadoop is the fastest, especially for lower values of *k*, because *Quadtree* repartitioning technique and the reduction in the number of candidates in the *Repartitioning* phase, although SpatialHadoop with *Grid* repartitioning technique is faster when *k* is higher ($k > 50$).

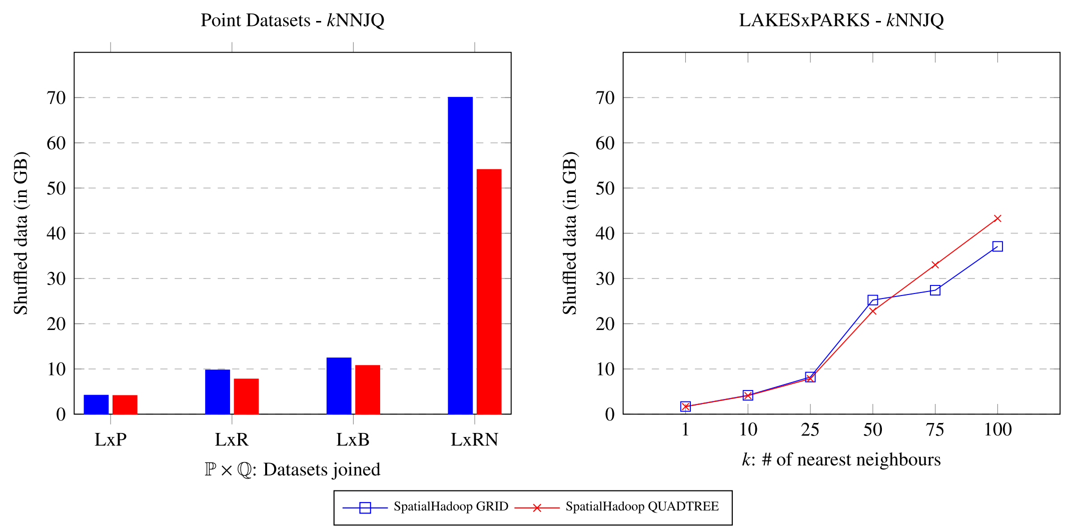

Fig. 13. kNNJQ cost (shuffled bytes) considering different datasets (left) and varying the k values (right).

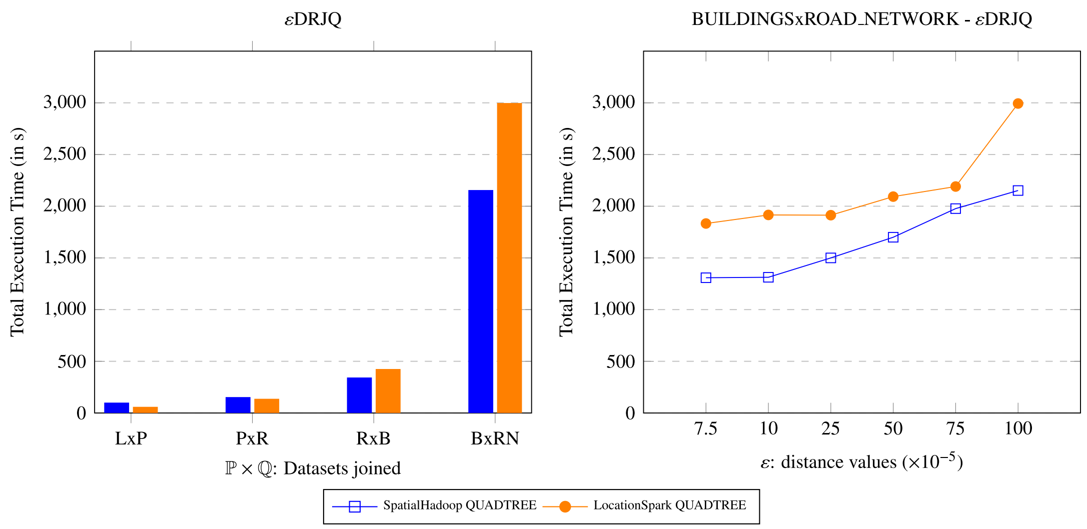

Fig. 14. εDRJQ cost considering different datasets (left) and varying the k values (right).

2. LocationSpark's execution times are short and stable with medium and large dataset sizes and for small k values, but its results are poor for larger datasets and higher k values, since it is more sensitive to memory constraints.
3. It is important to perform a good repartitioning technique in SpatialHadoop that allows to obtain, in a quick and efficient manner, the largest number of final kNN lists in the *Bin kNNJ* phase to reduce the size of the search space and the execution time of the following phases, mainly for the *kNNJ on Overlapping Cells* phase.
4. Similarly, for kCPQ, the higher the k value is, the higher the possibility that pairs of candidates will not be pruned, more tasks will be needed, and more total execution time will be consumed.
5. In SpatialHadoop, using both repartitioning techniques (*Grid* and *Quadtree*), the shuffled data grows with datasets of increasing size. However, with big datasets, the performance of the *Quadtree* repartitioning technique is reduced in the *Merge Results* phase.

6.4. εDRJQ experiments

The following experiment compared εDRJQ execution times in SpatialHadoop and LocationSpark. Fig. 14 shows εDRJQ execution times with the joined datasets ($L \times P$, $P \times R$, $R \times B$ and $B \times RN$) and fixed $\varepsilon = 0.001$. First, for combined medium-to-large datasets ($L \times P$ and $P \times R$), SpatialHadoop execution times were slightly longer. This is because it is a Reduce-based Join algorithm, and time is consumed by having to perform data shuffling and sorting between the *map* and *reduce* phases. However, for the combinations of the biggest datasets ($R \times B$ and $B \times RN$) SpatialHadoop was faster than LocationSpark (e.g.,

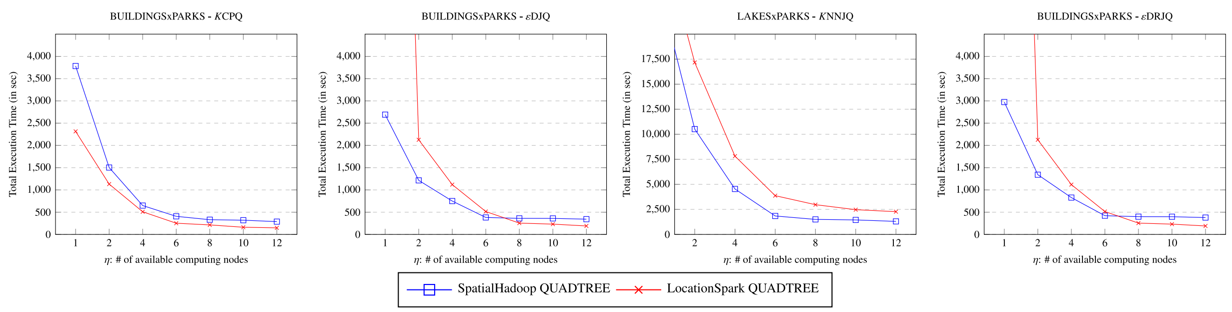

Fig. 15. Query cost with respect to the number of computing nodes η (Speedup).

for $B \times RN$ is 40% faster). Therefore, *memory pressure* problems in LocationSpark seem to influence the execution time more than those caused by the shuffled data in SpatialHadoop. As shown in the right-hand chart in Fig. 14, total execution time grows as ε increases. Performance is relatively similar with both DSDMSs for all ε values, but SpatialHadoop is faster in all cases. As shown above for εDJQ, the execution time of the SpatialHadoop εDRJQ increases for larger ε values, since more elements participate in the final results. In addition, execution times grew faster with εDRJQ, because the size of the data exchanged between the *map* and *reduce* phases also increases.

The main conclusions extracted from this comparison of performance are:

1. εDRJQ in SpatialHadoop is faster than LocationSpark for large datasets and for any ε value.
2. LocationSpark is the fastest for medium-to-large datasets (*LxP* and *PxR*), but when the dataset sizes grow, its performance in terms of running time is diminished, because of *memory pressure* problems.
3. The execution times of SpatialHadoop εDRJQ grow faster than εDJQ as the ε value increases, due to the increase in data shuffling, since the first is a Reduce-based Join algorithm and the second uses a Map-based Join algorithm.

6.5. *Speedup varying the number of computing nodes*

Finally, our last experiment measured speedup of all the proposed DJQ MapReduce algorithms (kCPQ, εDJQ, kNNJQ and εDRJQ), with respect to the number of computing nodes (η). The first chart in Fig. 15 shows the impact of different numbers of computing nodes on the performance of the distributed kCPQ algorithm for *BUILDINGS* $\times$ *PARKS* with the default configuration values. From this chart, it may be concluded that the performance of our approach has a direct relationship with the number of computing nodes. It may also be deduced that performance would improve if more computing nodes were added, but when the number of computing nodes exceeded the number of *map* tasks, there was no improvement. LocationSpark still performed better than SpatialHadoop. In the second chart in Fig. 15, the trend for εDJQ MapReduce algorithm was similar with a shorter execution time. However, in this case, LocationSpark shows worse performance for a smaller number of nodes. This is because LocationSpark and εDJQ depend more on available memory. Thus, when the number of nodes decreases, this memory also decreases considerably. The third chart in Fig. 15 shows much higher execution times for kNNJQ than for previous DJQ algorithms, mainly because it is a much more complex algorithm consisting of several phases. Nevertheless, the trend in the performance of both systems is very similar to kCPQ, exhibiting the lowest execution times for LocationSpark. Finally, the last chart in Fig. 15 shows the execution times for εDRJQ, and as for kCPQ and εDJQ, this algorithm has shorter values than kNNJQ, which is based on it. Furthermore, as seen for εDRJQ, SpatialHadoop shows better performance than LocationSpark when there are fewer computing nodes in use due to the sensitivity of the latter to memory constraints, a resource which is reduced by this decrease in the number of nodes.

The main conclusions extracted from these experiments varying the number of computing nodes (η), are:

1. All algorithms behave better when the number of computing nodes is increased, but if there are not enough tasks available for a certain number of nodes, there is no improvement in performance.
2. For LocationSpark, the number of nodes is not as determinant a parameter for speedup of the algorithms as the availability of enough memory resources.

6.6. *Discussion of the results*

By analyzing all the above experimental results, several important conclusions may be arrived at as summarized below:

1. We have experimentally demonstrated the *efficiency* (in terms of total execution time) and *scalability* (in terms of k and ε values, sizes of datasets and number of computing nodes, η) of the distributed algorithms proposed for DJQs in SpatialHadoop and LocationSpark.
2. With kCPQ and εDJQ, the larger the k or ε, the higher the possibility that pairs of candidates will not be pruned, more tasks will be needed, and longer total execution time will be consumed for reporting the final result. SpatialHadoop

performance for large k and ε were excellent due to use of *Quadtree* partitioning technique, which was the best partitioning technique of all those included in current SpatialHadoop framework. When more complex spatial objects were combined, the running time for SpatialHadoop was a bit more costly than for points, following a similar trend. Moreover, LocationSpark was faster than SpatialHadoop for medium and large dataset sizes, but for big datasets, it required more time to execute the queries, even for large k and ε values due to *memory pressure* problems.

3. For kNNJQ, we have proposed a MapReduce algorithm based on phases in SpatialHadoop, which we improved by using an initial phase for repartitioning the dense partitions. This makes SpatialHadoop the fastest, especially at lower k values, because of *Quadtree* repartitioning technique and the reduction in the number of candidates in the *Repartitioning* phase, although SpatialHadoop with *Grid* repartitioning technique has shorter execution times at higher k values. Moreover, it is important to mention that the use of repartitioning techniques in the *Repartitioning* phase generates more kNN lists in the *Bin kNNJ* phase to reduce the size of the search space and the execution time in the *kNNJ on Overlapping Cells* phase. For shuffled data, again, *Quadtree* repartitioning technique for combining the biggest datasets lowers this performance measure considerably in the *Merge Results* phase. LocationSpark, using the currently available implementation, has short and stable execution times with medium and large dataset sizes and for small k values, but shows poor results for larger datasets and k due to its greater sensitivity to memory constraints problems.
4. With εDRJQ, SpatialHadoop is the fastest for big datasets and at any ε value, except for medium-to-large dataset sizes where LocationSpark performance is the best.
5. The larger the number of computing nodes (η), the faster the DJQ MapReduce algorithms.
6. The use of *CombineFileSplits* [22] in SpatialHadoop [15] reduces the execution times considerably by avoiding the cost of data shuffling and sorting in the *reduce* phase. Thus, it would be of interest to study its use for the improvement of other algorithms such as KNNJQ, in which the size of shuffling data is an important factor.
7. The use of repartitioning techniques in SpatialHadoop considerably reduces execution times and shuffled data cost, mainly when big datasets are joined in kNNJQ. This indicates that this repartitioning is a good policy for MapReduce algorithms based on phases.
8. LocationSpark is very sensitive to memory restrictions problems, making its performance worse than SpatialHadoop for the same cluster when dataset sizes or k and ε are very large.
9. Finally, as a general conclusion, performance trends of both DSDMSs are similar in terms of execution time, although LocationSpark shows better performance when medium datasets are combined (if a suitable number of computing nodes with adequate memory resources are provided), even without pre-partitioning or pre-indexing is done. This suggests that further improvements are needed in LocationSpark (which is a very recent DSDMS), such as the treatment of skewed data. On the other hand, SpatialHadoop is a more robust and mature DSDMS, since several improvements have been included over the years (this research paper further included the use of a repartitioning technique for kNNJQ) and its performance is better for the DJQ MapReduce algorithms studied when the sizes of the datasets are large.

7. Conclusions and future work

DJQs are spatial operations widely adopted by many spatial and GIS applications. These spatial operations are very costly, especially when big spatial datasets are combined. These spatial queries have been actively studied in centralized environments, however, they have not attracted similar attention for parallel and distributed frameworks. Therefore, in this paper, we proposed new distributed DJQ algorithms (kNNJQ and εDRJQ) and compared them in two of the most recent and leading DSDMSs, SpatialHadoop and LocationSpark.

We designed novel distributed DJQ algorithms in SpatialHadoop and LocationSpark to do this, in particular, the first MapReduce algorithms in the literature for kNNJQ and εDRJQ in SpatialHadoop and εDRJQ in LocationSpark. Moreover, we have improved the MapReduce algorithms proposed for kNNJQ and εDRJQ in SpatialHadoop by using repartitioning (*Grid* and *Quadtree*) techniques in dense spatial areas. In addition, we have extended the distributed algorithms for more complex spatial objects, such as polygons and line-segments. An extensive set of exhaustive experiments has demonstrated that LocationSpark is the clear winner for execution time when medium datasets are combined, due to the efficiency of in-memory processing provided by Spark and additional improvements, like the *Query Plan Scheduler*. However, SpatialHadoop is faster when big real-world datasets are joined, since it is a more mature and robust DSDMS, due to the time invested in research and development (e.g., it provides more spatial partitioning techniques, computational geometry algorithms, repartitioning techniques for skewed data, etc.). Moreover, this detailed performance study also demonstrated that our distributed DJQ algorithms are efficient, robust and scalable with respect to such parameters as dataset sizes, k, ε, number of computing nodes (η), etc.

As part of our future work, we are planning to extend our current research in several directions:

– Improve the performance of kNNJQ through the use of SpatialHadoop or LocationSpark features, such as *CombineFileSplits* [22], which would enable the amount of shuffled data to be reduced and simplification of the different phases to achieve lower execution times,
– Implement other complex DJQs in SpatialHadoop and LocationSpark, like iceberg distance joins [34], multi-way spatial joins [28] and multi-way distance join [11],

– Implement other partitioning techniques [1,38] in SpatialHadoop, because this is an important factor for processing distance-based join queries, as we have demonstrated,
– Study other Spark-based DSDMSs like GeoSpark [45], since it being very actively developed and does not include any distance join queries [32].

Declaration of Competing Interest

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Acknowledgment

Work of all authors funded by the MINECO research project [TIN2017-83964-R].

References

[1] A. Aji, H. Vo, F. Wang, Effective spatial data partitioning for scalable query processing, CoRR abs/1509.00910 (2015) 1–12.
[2] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, J.H. Saltz, Hadoop-GIS: a high performance spatial data warehousing system over mapreduce, PVLDB 6 (11) (2013) 1009–1020.
[3] A. Akdogan, U. Demiryurek, F.B. Kashani, C. Shahabi, Voronoi-based geospatial query processing with MapReduce, in: CloudCom Conference, 2010, pp. 9–16.
[4] L. Alarabi, M.F. Mokbel, M. Musleh, St-hadoop: a mapreduce framework for spatio-temporal data, Geoinformatica 22 (4) (2018) 785–813.
[5] A. Bechini, F. Marcelloni, A. Segatori, A mapreduce solution for associative classification of big data, Inf. Sci. 332 (2016) 33–55.
[6] C. Böhm, F. Krebs, The *k*-nearest neighbour join: turbo charging the KDD process, Knowl. Inf. Syst. 6 (6) (2004) 728–749.
[7] C.L.P. Chen, C. Zhang, Data-intensive applications, challenges, techniques and technologies: a survey on big data, Inf. Sci. 275 (2014) 314–347.
[8] Y. Chen, J.M. Patel, Efficient evaluation of all-nearest-neighbor queries, in: ICDE Conference, 2007, pp. 1056–1065.
[9] A. Corral, J.M. Almendros-Jimenez, A performance comparison of distance-based query algorithms using r-trees in spatial databases, Inf. Sci. 177 (11) (2007) 2207–2237.
[10] A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Algorithms for processing k-closest-pair queries in spatial databases, Data Knowl. Eng. 49 (1) (2004) 67–104.
[11] A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Multi-way distance join queries in spatial databases, Geoinformatica 8 (4) (2004) 373–402.
[12] M. de Berg, O. Cheong, M.J. van Kreveld, M.H. Overmars, Computational geometry: Algorithms and applications, Springer, 2008.
[13] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, in: OSDI Conference, 2004, pp. 137–150.
[14] A. Eldawy, L. Alarabi, M.F. Mokbel, Spatial partitioning techniques in spatialhadoop, PVLDB 8 (12) (2015) 1602–1613.
[15] A. Eldawy, M.F. Mokbel, Spatialhadoop: A mapreduce framework for spatial data, in: ICDE Conference, 2015, pp. 1352–1363.
[16] F. García-García, A. Corral, L. Iribarne, G. Mavrommatis, M. Vassilakopoulos, A comparison of distributed spatial data management systems for processing distance join queries, in: ADBIS Conference, 2017, pp. 214–228.
[17] F. García-García, A. Corral, L. Iribarne, M. Vassilakopoulos, Distance range queries in spatialhadoop, in: JISBD Conference, 2016, pp. 1–14.
[18] F. García-García, A. Corral, L. Iribarne, M. Vassilakopoulos, Y. Manolopoulos, Enhancing spatialhadoop with closest pair queries, in: ADBIS Conference, 2016, pp. 212–225.
[19] F. García-García, A. Corral, L. Iribarne, M. Vassilakopoulos, Y. Manolopoulos, Efficient large-scale distance-based join queries in spatialhadoop, Geoinformatica 22 (2) (2018) 171–209.
[20] S. Hagedorn, P. Götze, K. Sattler, Big spatial data processing frameworks: Feature and performance evaluation, in: EDBT Conference, 2017, pp. 490–493.
[21] S. Hagedorn, T. Räth, Efficient spatio-temporal event processing with STARK, in: EDBT Conference, 2017, pp. 570–573.
[22] S. Karanth, Mastering Hadoop, Packt Publishing, 2014.
[23] M.R. Karim, M. Cochez, O.D. Beyan, C.F. Ahmed, S. Decker, Mining maximal frequent patterns in transactional databases and dynamic data streams: a spark-based approach, Inf. Sci. 432 (2018) 278–300.
[24] W. Kim, Y. Kim, K. Shim, Parallel computation of k-nearest neighbor joins using mapreduce, in: Big Data Conference, 2016, pp. 696–705.
[25] R.K. Lenka, R.K. Barik, N. Gupta, S.M. Ali, A. Rath, H. Dubey, Comparative analysis of spatialhadoop and geospark for geospatial big data analytics, CoRR abs/1612.07433 (2016) 1–6.
[26] F. Li, B.C. Ooi, M.T. Özsu, S. Wu, Distributed data management using mapreduce, ACM Comput. Surv. 46 (3) (2014) 31:1–31:42.
[27] W. Lu, Y. Shen, S. Chen, B.C. Ooi, Efficient processing of k nearest neighbor joins using MapReduce, PVLDB 5 (10) (2012) 1016–1027.
[28] N. Mamoulis, D. Papadias, Multiway spatial joins, ACM Trans. Database Syst. 26 (4) (2001) 424–475.
[29] G. Mavrommatis, P. Moutafis, M. Vassilakopoulos, F. García-García, A. Corral, Slicenbound: solving closest pairs and distance join queries in apache spark, in: ADBIS Conference, 2017, pp. 199–213.
[30] P. Moutafis, G. Mavrommatis, M. Vassilakopoulos, S. Sioutas, Efficient processing of all-k-nearest-neighbor queries in the mapreduce programming framework, Data Knowl. Eng. 121 (2019) 42–70.
[31] N. Nodarakis, E. Pitoura, S. Sioutas, A.K. Tsakalidis, D. Tsoumakos, G. Tzimas, Kdann+: a rapid aknn classifier for big data, Trans. Large-Scale Data-Knowl.-Centered Syst. 24 (2016) 139–168.
[32] V. Pandey, A. Kipf, T. Neumann, A. Kemper, How good are modern spatial analytics systems? PVLDB 11 (11) (2018) 1661–1673.
[33] G. Roumelis, M. Vassilakopoulos, A. Corral, Y. Manolopoulos, New plane-sweep algorithms for distance-based join queries in spatial databases, Geoinformatica 20 (4) (2016) 571–628.
[34] Y. Shou, N. Mamoulis, H. Cao, D. Papadias, D.W. Cheung, Evaluation of iceberg distance joins, in: SSTD Conference, 2003, pp. 270–288.
[35] G. Song, J. Rochas, L.E. Beze, F. Huet, F. Magoulès, K nearest neighbour joins for big data on mapreduce: a theoretical and experimental analysis, IEEE Trans. Knowl. Data Eng. 28 (9) (2016) 2376–2392.
[36] M. Tang, Y. Yu, W.G. Aref, A.R. Mahmood, Q.M. Malluhi, M. Ouzzani, Locationspark: in-memory distributed spatial query processing and optimization, CoRR abs/1907.03736 (2019) 1–15.
[37] M. Tang, Y. Yu, Q.M. Malluhi, M. Ouzzani, W.G. Aref, Locationspark: a distributed in-memory data management system for big spatial data, PVLDB 9 (13) (2016) 1565–1568.
[38] H. Vo, A. Aji, F. Wang, SATO: a spatial data partitioning framework for scalable query processing, in: SIGSPATIAL Conference, 2014, pp. 545–548.
[39] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, M. Guo, Simba: efficient in-memory spatial analytics, in: SIGMOD Conference, 2016, pp. 1071–1085.
[40] X. Yao, G. Li, Big spatial vector data management: a review, Big Earth Data 2 (1) (2018) 108–129.
[41] X. Yao, M.F. Mokbel, L. Alarabi, A. Eldawy, J. Yang, W. Yun, L. Li, S. Ye, D. Zhu, Spatial coding-based approach for partitioning big spatial data in hadoop, Computers & Geosciences 106 (2017) 60–67.

[42] T. Yokoyama, Y. Ishikawa, Y. Suzuki, Processing all k-nearest neighbor queries in hadoop, in: WAIM Conference, 2012, pp. 346–351.
[43] S. You, J. Zhang, L. Gruenwald, Large-scale spatial join query processing in cloud, in: ICDE Workshops, 2015, pp. 34–41.
[44] S. You, J. Zhang, L. Gruenwald, Spatial join query processing in cloud: analyzing design choices and performance comparisons, in: ICPPW Conference, 2015, pp. 90–97.
[45] J. Yu, J. Wu, M. Sarwat, Geospark: a cluster computing framework for processing large-scale spatial data, in: SIGSPATIAL Conference, 2015, pp. 70:1–70:4.
[46] J. Yu, Z. Zhang, M. Sarwat, Spatial data management in apache spark: the geospark perspective and beyond, Geoinformatica 23 (1) (2019) 37–78.
[47] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, in: NSDI Conference, 2012, pp. 15–28.
[48] C. Zhang, F. Li, J. Jestes, Efficient parallel kNN joins for large data in MapReduce, in: EDBT Conference, 2012, pp. 38–49.
[49] H. Zhang, G. Chen, B.C. Ooi, K. Tan, M. Zhang, In-memory big data management and processing: a survey, IEEE Trans. Knowl. Data Eng. 27 (7) (2015) 1920–1948.
[50] J. Zhang, N. Mamoulis, D. Papadias, Y. Tao, All-nearest-neighbors queries in spatial databases, in: SSDBM Conference, 2004, pp. 297–306.