F. García-García, A. Corral, L. Iribarne, M. Vassilakopoulos.(2020)Improving Distance-Join Query Processing with Voronoi-Diagram based Partitioning in SpatialHadoop. Future Generation Computer Systems, Volume 111, October 2020, Pages 723-740. ISSN: 0167-739X

Improving Distance-Join Query processing with Voronoi-Diagram based partitioning in SpatialHadoop

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

[a] Department on Informatics, University of Almeria, Almeria, Spain
[b] Department of Electrical & Computer Engineering, University of Thessaly, Volos, Greece

ARTICLE INFO

Article history:
Received 30 April 2019
Received in revised form 11 September 2019
Accepted 27 October 2019
Available online 31 October 2019

Keywords:
Data partitioning
K nearest neighbors join
K closest pairs
SpatialHadoop
MapReduce
Spatial query evaluation

ABSTRACT

SpatialHadoop is an extended MapReduce framework supporting global indexing techniques that partition spatial datasets across several machines and improve spatial query processing performance compared to traditional Hadoop systems. SpatialHadoop supports several spatial operations (e.g., *K* Nearest Neighbor search, range query, spatial intersection join, etc.) and seven spatial partitioning techniques (Grid, Quadtree, STR, STR+, *k*-d tree, Z-curve and Hilbert-curve). Distance-Join Queries (DJQs), like the *K* Nearest Neighbors Join Query (*K*NNJQ) and *K* Closest Pairs Query (*K*CPQ), are common operations used in numerous spatial applications. DJQs are costly operations, since they combine spatial joins with distance-based search. Data partitioning improves the management of large datasets and speeds up query performance. Therefore, performing DJQs efficiently with new partitioning methods in SpatialHadoop is a challenging task. In this paper, a new data partitioning technique based on Voronoi-Diagrams is designed and implemented in SpatialHadoop. Moreover, improved *K*NNJQ and *K*CPQ MapReduce algorithms, using the new partitioning mechanism, are also designed and developed for SpatialHadoop. Finally, the results of an extensive set of experiments with real-world datasets are presented, demonstrating that the new partitioning technique and the improved DJQ MapReduce algorithms are efficient, scalable and robust in SpatialHadoop.

1. Introduction

In the age of smart cities and mobile environments, the increase of the volume of available spatial data (e.g., location, routing, etc.) is huge all over the world. Recent developments of spatial big data systems have motivated the emergence of novel technologies for processing large-scale spatial data on shared-nothing clusters in a distributed environment. SpatialHadoop [1] is a disk-based Distributed Spatial Data Management System (DSDMS) based on Hadoop-MapReduce that allows users to work on distributed spatial data without worrying about computation distribution and fault-tolerance. SpatialHadoop is a full-fledged MapReduce [2] framework with native support for spatial data. SpatialHadoop injects spatial data awareness in each Hadoop layer (i.e., language, storage, MapReduce and operations layers).

Data partitioning is a powerful mechanism for improving efficiency of data management systems, and it is a standard feature in modern database systems. Aside from the fact that data partitioning improves the overall manageability of large datasets, it also speeds up query performance. By partitioning such datasets into smaller units, it enables processing of a query in parallel and reduces the I/O activity by only scanning a few partitions that contain data relevant to the query constraints. *Spatial data partitioning* is challenging, especially due to several important properties that are particular to spatial data and query processing, like spatial data skew and boundary object handling [3]. SpatialHadoop [1] supports seven spatial partitioning strategies to handle large-scale spatial data [4]. They are classified as *space-based* (Grid and Quadtree), *data-based* (STR, STR+ and *k*-d tree) and *space filling curve-based* (Z-curve and Hilbert-curve) partitioning strategies.

The *Voronoi-Diagram* is a partitioning of a geometric space that contains points data. Each partition of the Voronoi-Diagram, called *Voronoi-Cell*, is associated with a point *p* (pivot), such that any point inside *p's* cell has *p* as its nearest neighbor [5,6]. The resulting data structure from the Voronoi-Diagram is very efficient in exploring a local neighborhood in a geometric space. Voronoi-Diagrams are used in many algorithmic applications, like closest-site problems (nearest neighbor queries and closest pairs), clustering point sites (partitioning and hierarchical clustering methods), placement and motion planning, triangulating sites, connectivity graphs for sites, etc. [6]. In our case,

* 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).

https://doi.org/10.1016/j.future.2019.10.037

the points dataset is divided into partitions based on a Voronoi-Diagram with a careful method for selecting a set of suitable pivots. Then, these data partitions (Voronoi-Cells) are clustered into groups only if the distances between them are restricted by a specific distance bound. The *pivot selection strategy* is crucial for the creation of Voronoi-Diagrams and therefore for the DJQ processing. In [7], three pivot selection strategies were proposed: random selection, furthest selection and *k*-means selection. Random selection was faster than *k*-means, but during the *K*NN join phase, the performance of *k*-means selection was better. Hence, the use of a *clustering algorithm* improves the quality of the selected pivots, which separate the whole datasets more evenly, and also improves the power of the pruning rules for DJQs. This is because the clustering algorithms aim at grouping objects in such a way that similar ones belong to the same cluster and are different from the ones which belong to other clusters [8,9]. Finally, to optimize and speed up existing clustering algorithms, *sampling* is a very interesting technique when large datasets are managed [10]. It can be considered as a preprocessing step for clustering algorithms, and we will use a sampling method for providing a representative and relevant set of samples before executing the clustering algorithm.

Distance-Join Queries (DJQs) in spatial databases have received considerable attention from the database community, due to their importance in numerous applications, such as geographical information systems (GIS), location-based systems, continuous monitoring in streaming data settings and processing of road network constrained data, among others. DJQs are costly queries because they combine two datasets, taking into account a certain distance metric. Two of the most representative and known DJQs are the *K* Nearest Neighbor Join Query (*K*NNJQ) and the *K* Closest Pairs Query (*K*CPQ). Given two points datasets $\mathbb{P}$ and $\mathbb{Q}$, the *K*NNJQ finds, for each point of $\mathbb{P}$, its *K* nearest neighbors in $\mathbb{Q}$. The *K*CPQ finds the *K* pairs of points that have the K smallest distances between all possible pairs of points ($\mathbb{P} \times \mathbb{Q}$) that can be formed by choosing one point of $\mathbb{P}$ and one point of $\mathbb{Q}$. All these DJQ algorithms require spatial join query processing techniques [11], for example, the plane-sweep technique is used when neither datasets are indexed. Several research works have been devoted to improve the performance of these DJQs by proposing efficient algorithms in centralized environments, e.g., for *K*NNJQ [12–14] and for *K*CPQ [15–17]. However, with the fast increase in the scale of the big input datasets, processing large-scale data in a parallel and distributed way is becoming a popular practice. For this reason, a number of parallel and distributed DJQ algorithms in MapReduce have been designed and implemented for *K*NNJQ [7,18,19]. *K*CPQ MapReduce algorithms [20,21] have been also developed particularly in SpatialHadoop. In [22], a data partitioning technique using Voronoi-Diagrams is included in SpatialHadoop to improve the performance of the *K*NNJQ and *K*CPQ.

This paper substantially extends our previous work [22] with the following novel contributions:

1. We analyze existing parallel and distributed DJQs algorithms in MapReduce, using Voronoi-Diagram based partitioning techniques, and identify possible improvements and optimizations.
2. We improve the data partitioning technique based on Voronoi-Diagrams in SpatialHadoop by using new sampling methods and clustering algorithms to produce high quality partitions, making the DJQ MapReduce algorithms faster.
3. We improve the *K*NNJQ and *K*CPQ MapReduce algorithms by using the new partitioning method, new pruning rules and techniques for reducing the replication and shuffled data.
4. In experiments of DJQ MapReduce algorithms, we use the best spatial partitioning technique available in SpatialHadoop for joins [4,21] (i.e., Quadtree) to compare their performance against our improved Voronoi-Diagram based partitioning technique.
5. We present results of an extensive experimental study that compares the performance of the proposed DJQ MapReduce algorithms in SpatialHadoop and their improvements in terms of efficiency, extensibility and scalability, using big real-world spatial datasets [1].

The current research work differs from [7] in that we use the MapReduce algorithmic scheme presented in [18] for *K*NNJQ. Here, we have also improved the Voronoi-Diagram based partitioning technique of [7] by using new sampling methods and clustering algorithms for obtaining better quality of the selected pivots. Moreover, new pruning rules [23] and processing enhancements have been incorporated to speed up the response time of the *K*NNJQ in SpatialHadoop. Additionally, the treatment of the largest and densest partitions in [7] is handled by the geometric grouping technique (bottom-up way) and we use the repartitioning technique (top-down way) [11]. Finally, we have compared experimentally the proposed partitioning method based on Voronoi-Diagrams with other spatial partitioning techniques, as the Quadtree [4], proving its excellent performance.

This paper is organized as follows. Section 2 reviews related work. Section 3 gives the preliminary concepts related to DJQ, SpatialHadoop and data partitioning techniques based on Voronoi-Diagrams. Section 4 proposes a data partitioning technique based on Voronoi-Diagrams in SpatialHadoop. In Section 5, the parallel and distributed algorithms for processing *K*NNJQ and *K*CPQ in SpatialHadoop are presented. Section 6 describes several potential improvements of the proposed distributed algorithms. In Section 7, we present the most representative results of an extensive set of experiments that we have performed, using real-world datasets, for comparing the new data partitioning technique and the improved DJQ MapReduce algorithms in SpatialHadoop. Finally, Section 8 gives an overview of the conclusions and results from our paper and indicates research directions for future work.

2. Related work

Nowadays, researchers, developers and practitioners worldwide have started to take advantage of the Hadoop-MapReduce framework to support massive-scale geospatial data processing. The most representative research prototypes of Hadoop-based systems for scalable spatial data processing are the following:

- *Parallel-Secondo* [24] is a parallel spatial DBMS that uses Hadoop as a distributed task scheduler. It integrates Hadoop with SECONDO [25], a database that can handle non-standard data types, like spatial data, usually not supported by standard systems. It only supports *uniform* spatial data partitioning techniques, which cannot handle efficiently the spatial data skewness problem.
- *Hadoop-GIS* [26] extends Hive [27], a data warehouse infrastructure built on top of Hadoop with a uniform grid index for range queries, spatial joins and other spatial operations. It adopts Hadoop Streaming framework and integrates several open source software packages for spatial indexing and geometry computation. It utilizes SATO spatial partitioning [28] (similar to *k*-d tree) and local spatial indexing to achieve efficient query processing.
- *SpatialHadoop* [1] is a full-fledged MapReduce framework with native support for spatial data. It tightly integrates well-known spatial operations (including range queries,

*K*NN query, spatial join and CG_Hadoop [29]) into Hadoop. It supports various spatial data types (point, line string, polygon, multi-point, etc.), several spatial partitioning techniques [4] (uniform Grids, SRT, Quadtree, *k*-d tree, Hilbert-curve and Z-curve) and local spatial indexes (Grid file, R-tree and R^+-tree). In addition, SpatialHadoop has an excellent performance and it is one of the best maintained Hadoop-based Distributed Spatial Data Management System [30]. For all these reasons, we have focused on SpatialHadoop and not in other Hadoop-based DSDMSs.

In [3], an extension of SATO [28], that is a spatial data partitioning framework for scalable query processing, is presented. The main objective of [3] is to provide a comprehensive guidance for spatial data partitioning to support scalable and fast spatial data processing in distributed computing environments such as MapReduce. To accomplish this, the authors provide a systematic evaluation of six spatial partitioning methods with a set of different partitioning strategies and study their implications on the performance of spatial queries in MapReduce. In particular, the proposed spatial partitioning algorithms were Binary Split Partitioning (BSP), Fixed Grid Partitioning (FG), Strip Partitioning (SLC), Boundary Optimized Strip Partitioning (BOS), Sort-Tile-Recursive Partitioning (STR) and Hilbert Curve Partitioning (HC). The most important results are the runtime cost of the partitioning algorithms (there are three categories: fast -FG, BSP-, medium -HC, STR- and slow -SLC, BOS-) and spatial join query performance between two datasets, where BSP and STR have the best performance in terms of running time and, FG and HC are the worst.

In [4], seven different *spatial partitioning techniques* in SpatialHadoop are presented, and an extensive experimental study on the quality of the generated index and the performance of range and spatial join queries is reported. These seven partitioning techniques are also classified in two categories according to boundary object handling: *replication-based techniques* (Grid, Quadtree, STR+ and *k*-d tree) and *distribution-based techniques* (STR, Z-curve and Hilbert-curve) [4]. The *distribution-based techniques* assign an object to exactly one overlapping partition and the partition has to be expanded to enclose all contained points. The *replication-based techniques* avoid expanding partitions by replicating each point to all overlapping partitions, but the query processor has to employ a duplicate avoidance technique to account for replicated elements. The most important conclusions of [4] for distributed join processing, using the *overlap* spatial predicate, are the following: (1) the smallest running time is obtained when the same partitioning technique is used for the join processing, (2) Quadtree outperforms all other techniques with respect to running time, since it minimizes the number of overlapping partitions between the two files by employing a regular space partitioning, (3) Z-Curve reports the worst running times, and (4) *k*-d tree gets very similar results to STR.

The most representative papers that adopt the Voronoi-Diagram based partitioning technique within MapReduce are [7, 22,23,31,32]. In [31], a distributed Voronoi-Diagram index is proposed to answer geospatial (range and *K*NN) queries in 2d spaces. In [7], the problem of answering the *K*NNJ using MapReduce is studied. This is accomplished by exploiting the Voronoi-Diagram based partitioning method, that divides the input datasets into groups, such that *K*NNJ can answer by only checking object pairs within each group. Moreover, several pruning rules to reduce the shuffling cost as well as the computation cost are developed in the PGBJ (Partitioning and Grouping Block Join) algorithm, which works with two MapReduce phases. In [32], the vector projection pruning technique is proposed to process efficiently *K*NNJ, since it enables to prune non-*K*NN points and reduce the cost of distance computation. A new algorithm, *K*NN-MR, using this new pruning technique, that performs slightly better than PGBJ, is presented. [23] presents a new multi-round computation strategy for parallel *K*NNQ that exploits pivot-based data partitioning, by using data-driven bounds and a tiered support discovery technique which effectively limit data duplication. Finally, in [22], a new *K*NNJQ MapReduce algorithm and an improved *K*CPQ MapReduce algorithm, using Voronoi-Diagram based partitioning technique, are also developed for SpatialHadoop.

The main differences of these papers that use the Voronoi-Diagram based partitioning technique within MapReduce with respect to the current research work are the following: With respect to [7], [23] and [32], we have utilized a different algorithmic scheme for the KNNJQ MapReduce algorithm, better sampling methods for improved clustering algorithms, adapted pruning rules from [23], the less data technique for reducing the size of shuffled data and a repartitioning technique for the treatment of the densest areas. Moreover, we have implemented an improved algorithm for KCPQ, and we have performed many experiments, even comparing with other spatial partitioning techniques included in SpatialHadoop. Finally, in [31], MapReduce algorithms for range query, KNNQ and Reverse KNNQ have been proposed, but there is no design and implementation for DJQ MapReduce algorithms.

The research work of [7] and [22] have shown that the use of a suitable *pivot selection strategy* is very influential in the DJQ performance over distributed frameworks. The best performance was obtained by using *k*-means clustering algorithm [33], and for this reason the use of *clustering algorithms* is worth to be studied for the pivot selection strategy. Schemes to improve the pivot selection have been studied actively in the context of metric space indexes [34]. For instance, *iDistance* [35] selects pivots based on the clustering results of *k*-means. There are many existing clustering algorithms in the literature [8,9], and they can be classified into five categories: (1) partition-based clustering algorithms (e.g., *k*-means, *k*-medoids, etc.), (2) hierarchical clustering algorithms (e.g., BIRCH, CURE, etc.), (3) density-based clustering algorithms (e.g., DBSCAN, OPTICS, etc.); (4) grid-based clustering algorithms (e.g., STING, PROCLUS, etc.), and (5) model-based clustering algorithms (e.g., EM, COBWEB, etc.). Due to these, we expect that the use of an appropriate clustering algorithm will have an important influence on the pivot selection strategy and therefore on query performance, we will use such algorithms in this research work.

Apart of [7,22,23,31,32], other parallel and distributed *K*NNJQ algorithms that do not use Voronoi-Diagram based partitioning technique have been published in the literature. The most remarkable ones are [18,19,36–38]. In [36] the *RankReduce* approach for processing large amounts of data for *K*NNQ is proposed, using Locality Sensitive Hashing (LSH). The LSH algorithm implemented in MapReduce assigns similar objects to one fragment in the distributed file system enabling an effective selection of potential candidate neighbors reduced to a set of *K* nearest neighbors. In [19], novel (exact and approximate) algorithms in MapReduce to perform efficient parallel *K*NNJQ on large datasets are proposed, and they use the R-tree and Z-value-based partition joins to implement them. In [37], the existing solutions that perform the *K*NNJ operation in the context of MapReduce are reviewed and studied from the theoretical and experimental point of view. In [38], the *Spitfire* approach was presented; it improves over the duplication rate of [7] by utilizing grid-based partitioning to divide the data and bound the support set. Finally, the only DJQ algorithm already included in SpatialHadoop is the *K*CPQ MapReduce algorithm [20,21], that consists of a MapReduce job, adopting the plane-sweep technique [11] and improving the computation of an upper bound of the distance value of the *K*th closest pair from sampled data as a global or local preprocessing phase.

3. Preliminaries and background

In this section, we first present the basic definitions of the *K*NNJQ and *K*CPQ, followed by a brief introduction of SpatialHadoop, and finally the main concepts and properties of the Voronoi-Diagrams.

3.1. Distance-Join Queries

To introduce the details of the semantics of the DJQs studied in this paper, we define the *K*NNJ and *K*CP queries, assuming that the Euclidean distance (satisfying the properties of *non-negativity*, *identity*, *symmetry* and *triangular inequality*), *dist*, is the distance used along the article. Moreover, we also define the distance-based query that is the basis of *K*NNJQ, the *K* Nearest Neighbor (*K*NN) query, where just one dataset is processed.

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

Definition 1 (*K Nearest Neighbor Query, KNN Query*)**.** Let $\mathbb{P} = \{p_1, p_2, \ldots, p_n\}$ 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: $KNN(\mathbb{P}, q, K) = \{p_1, p_2, \ldots, p_K\} \subseteq \mathbb{P}$, such that $\forall p \in \mathbb{P} \setminus KNN(\mathbb{P}, q, K)$ we have $dist(p_i, q) \leq dist(p, q)$, $1 \leq i \leq K$.

When two datasets ($\mathbb{P}$ and $\mathbb{Q}$) are combined, two of the most studied DJQs are the K Nearest Neighbor Join (*K*NNJ) and the K Closest Pairs (*K*CP) queries.

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

Definition 2 (*K Nearest Neighbor Join Query, KNNJ Query*)**.** Let $\mathbb{P} = \{p_1, p_2, \ldots, p_n\}$ and $\mathbb{Q} = \{q_1, q_2, \ldots, q_m\}$ be two sets 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 $\mathbb{Q}$: $KNNJ(\mathbb{P}, \mathbb{Q}, K) = \{(p_i, q_j) : \forall\, p_i \in \mathbb{P}, q_j \in KNN(\mathbb{Q}, p_i, K)\}$

The most important properties of *K*NNJQ are the following: (1) *K*NNJQ is asymmetric, i.e., $KNNJ(\mathbb{P}, \mathbb{Q}, K) \neq KNNJ(\mathbb{Q}, \mathbb{P}, K)$, since *K*NNQ is asymmetric (i.e., if $KNN(\mathbb{P}, q, 1) = \{p\}$, there may exist another $q' \in \mathbb{Q}$ ($q' \neq q$) such that $dist(p, q') < dist(p, q)$ and $KNN(\mathbb{Q}, p, 1) = \{q'\}$). Moreover, given $\mathbb{P} \neq \mathbb{Q}$ ($|\mathbb{P}| \neq |\mathbb{Q}|$), the cardinality of $KNNJ(\mathbb{P}, \mathbb{Q}, K)$ is $K \times |\mathbb{P}|$ (similarly $|KNNJ(\mathbb{Q}, \mathbb{P}, K)| = K \times |\mathbb{Q}|$), and therefore the results are different. In the case of $|\mathbb{P}| = |\mathbb{Q}|$ ($\mathbb{P} \neq \mathbb{Q}$), although the cardinalities of the results are the same, the content is different, $KNNJ(\mathbb{P}, \mathbb{Q}, K) \neq KNNJ(\mathbb{Q}, \mathbb{P}, K)$, due to *K*NNQ is asymmetric. (2) Given $K \ll |\mathbb{Q}|$, the cardinality of $KNNJ(\mathbb{P}, \mathbb{Q}, K)$ is predictable ($|\mathbb{P}| \times K$), since it returns K nearest neighbors in $\mathbb{Q}$ for each point of $\mathbb{P}$. (3) The distance from each point of $\mathbb{P}$ to its K nearest neighbors is unknown a priori.

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 as follows.

Definition 3 (*K Closest Pairs Query, KCP Query*)**.** Let $\mathbb{P} = \{p_1, p_2, \ldots, p_n\}$ and $\mathbb{Q} = \{q_1, q_2, \ldots, q_m\}$ be two sets 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:

$KCP(\mathbb{P}, \mathbb{Q}, K) = \{(p_1, q_1), (p_2, q_2), \ldots, (p_K, q_K)\} \subseteq \mathbb{P} \times \mathbb{Q}$, such that for any $(p_i, q_j) \in \mathbb{P} \times \mathbb{Q} \setminus KCP(\mathbb{P}, \mathbb{Q}, K)$ we have $dist(p_l, q_l) \leq dist(p_i, q_j)$, $1 \leq l \leq K$.

*K*CPQ has the following properties: (1) *K*CPQ is symmetric, i.e., $KCP(\mathbb{P}, \mathbb{Q}, K) = KCP(\mathbb{Q}, \mathbb{P}, K)$, since it discovers the K pairs of points with the K smallest distances from all possible pairs that can be formed from the join of two datasets, and the Euclidean distance is symmetric $dist(p_i, q_j) = dist(q_j, p_i)$. (2) The cardinality of the result is known beforehand $|KCP(\mathbb{P}, \mathbb{Q}, K)| = K$. (3) The distance of the K closest pairs of points is unknown a priori.

3.2. SpatialHadoop

SpatialHadoop [1] is a full-fledged MapReduce framework with native support for spatial data. It is an efficient disk-based distributed spatial query processing system. Note that MapReduce [2] is a scalable, flexible and fault-tolerant programming framework for distributed large-scale data analysis (i.e., MapReduce is a shared-nothing platform for processing large-scale datasets). A task to be performed using the MapReduce framework consists of two phases: the *map* phase which is specified by a *map function* that takes input typically from Hadoop Distributed File System (HDFS) files, possibly performs some computations on this input, and distributes the result to worker nodes; and the *reduce* phase which processes these results as specified by a *reduce function*. An important aspect of MapReduce is that both the input and the output of the *map* step are represented as *key–value pairs*, and that pairs with the same key will be processed as one group by the *reducer*. Additionally, a *combiner function* can be used to run on the output of the *map* phase and perform some filtering or aggregation to reduce the number of keys passed to the *reducer*.

SpatialHadoop is a comprehensive extension to Hadoop that injects spatial data awareness in each Hadoop layer, namely, language, storage, MapReduce, and operations layers. *MapReduce* layer is the query processing layer that runs MapReduce programs, taking into account that SpatialHadoop supports spatially indexed input files. The *Operation* layer enables the efficient implementation of spatial operations, considering the combination of the spatial indexing in the *storage* layer with the new spatial functionality in the *MapReduce* layer. In general, a spatial query processing in SpatialHadoop consists of four steps [1,20,21], as we can observe in Fig. 1 for DJQs:

1. *Preprocessing*, where the dataset is partitioned according to a specific partitioning technique (e.g., Grid, Quadtree, STR, Hilbert-curve, etc.) [4], generating a set of partitions or cells. In this partitioning process, spatial data locality is obeyed, since spatially nearby objects are assigned to the same partition [1]. Each partition corresponds to a HDFS block, and the HDFS blocks in each file are globally indexed, generating a *spatially indexed file* (indexing).
2. *Pruning*, when the query is issued, this is the step where the master node examines all partitions and prunes (by a *filter function*) those ones that are guaranteed not to include in any possible result of the spatial query. SpatialHadoop enriches traditional Hadoop systems in this step with the *SpatialFileSplitter* component, that is, an extended splitter that exploits the global index(es) on input file(s) to prune easily file partitions not contributing to the answer [1].
3. *Local Spatial Query Processing*, where a local spatial query is performed on each non-pruned partition in parallel on different machines. SpatialHadoop also enriches traditional Hadoop systems in this step by the *SpatialRecordReader*,

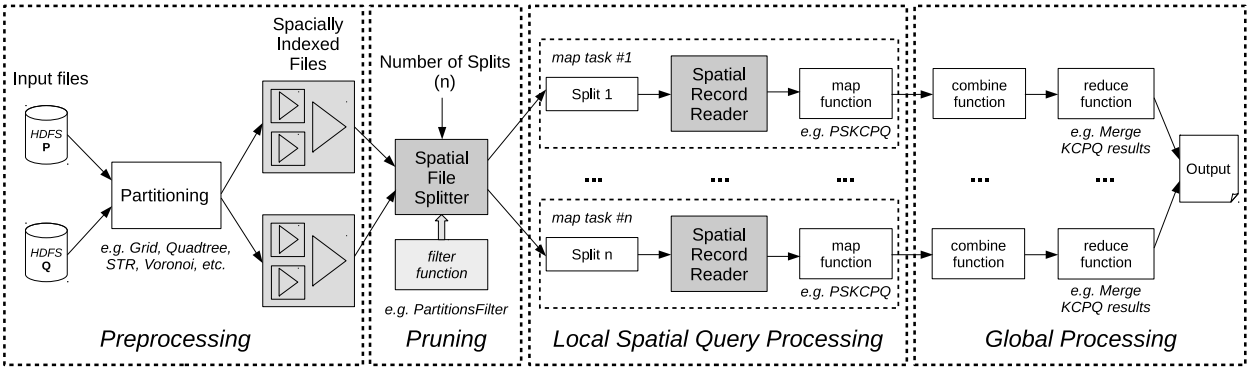


Fig. 1. Spatial query processing for DJQs in SpatialHadoop.

which reads a split originating from the spatially indexed input file(s) and exploits local index(es) to efficiently processes the spatial queries [1]. In this step, if we do not use the *SpatialRecordReader* component (i.e., we do not exploit the advantages of the local index(es)), we only use a *RecordReader* that extracts records as key–value pairs which are passed to the *map* function to perform, for example, a plane-sweep-based algorithm.

4. *Global Processing*, where the results are collected from all nodes (machines) in the previous step and the final result of the concerned spatial query is computed. A *combine* function can be applied in order to decrease the volume of data that is sent from the *map* task. The *reduce* function can be omitted when the results from the *map* phase are final.

3.3. Data partitioning technique based on Voronoi-Diagrams

Our proposed approach to address the new data partitioning technique in SpatialHadoop is based on *Voronoi-Diagrams*, and according to [37] it is a *distance based partitioning strategy*. A *Voronoi-Diagram* divides space into disjoint polygons where the nearest neighbor of any point inside a polygon is the generator or pivot of the polygon. Let $\mathcal{R} = \{r_1, r_2, \ldots, r_t\}$ be a set of t distinct points in the plane; these points can be called generators or *pivots*. We define the *Voronoi-Diagram* of $\mathcal{R}$ as the subdivision of the plane into r cells, one for each pivot r_i in $\mathcal{R}$, with the property that a point p lies in the cell corresponding to a pivot r_i if and only if $dist(p, r_i) < dist(p, r_j)$ for each $r_j \in \mathcal{R}$ with $j \neq i$. We can denote the Voronoi-Diagram generated by $\mathcal{R}$ as $VD(\mathcal{R})$. The cell of $VD(\mathcal{R})$ that corresponds to a pivot r_i is called the *Voronoi-Cell* of r_i and is denoted by $VC(r_i)$ or V_i for short. The Voronoi-Diagram of a set of point $\mathcal{R}$, $VD(\mathcal{R})$, is unique and it also satisfies the following property: $VD(\mathcal{R}) = \bigcup_{i1}^{t} V_i$ and $\bigcap_{i=1}^{t} V_i = \emptyset$, where $V_i = \{p : dist(p, r_i) < dist(p, r_j) \text{ for } j \neq i\}$.

According to [7], given a set of points $\mathbb{P}$, the main idea of *Voronoi-Diagram based partitioning* technique is to select a set $\mathcal{R}$ of points (which may not necessarily belong to $\mathbb{P}$) as *pivots*, and then split the points of $\mathbb{P}$ into $|\mathcal{R}|$ disjoint partitions, where each point is assigned to the partition of its closest pivot r_i in $\mathcal{R}$. In the case of multiple pivots that are closest to a particular point, then that point is assigned to the partition with the smallest number of points. In this way, the whole data space is split into $|\mathcal{R}|$ disjoint Voronoi-Cells. In summary, the set of points are divided into partitions based on a Voronoi-Diagram with carefully selected *pivots*. Then, data partitions (i.e., Voronoi-Cells) are clustered into groups only if the distances between them are restricted by a specific bound.

Moreover, two distance metrics are defined, $U(\mathcal{P}_i^{\mathbb{P}})$ and $L(\mathcal{P}_i^{\mathbb{P}})$, to be used in the MapReduce DJQ algorithms. Let $\mathcal{R}$ be the set of selected pivots, $\forall r_i \in \mathcal{R}$, $\mathcal{P}_i^{\mathbb{P}}$ denotes the set of points from $\mathbb{P}$ that has r_i as its closest pivot. We denote $U(\mathcal{P}_i^{\mathbb{P}})$ and $L(\mathcal{P}_i^{\mathbb{P}})$ as the maximum and minimum distance from the pivot r_i to the points of $\mathcal{P}_i^{\mathbb{P}}$, respectively. That is, $U(\mathcal{P}_i^{\mathbb{P}}) = \max\{dist(p, r_i) : \forall p \in \mathcal{P}_i^{\mathbb{P}}\}$ and $L(\mathcal{P}_i^{\mathbb{P}}) = \min\{dist(p, r_i) : \forall p \in \mathcal{P}_i^{\mathbb{P}}\}$. Table 1 shows the symbols and their meanings used throughout this paper.

Table 1
Symbols and their meanings.

Symbol	Definition
K	Number of the NNs or the CPs, $K \geq 1$
k	Number of clusters or partitions, $k \geq 1$
$\mathbb{P}$ (resp. $\mathbb{Q}$)	Set of points $\mathbb{P}$ (resp. $\mathbb{Q}$)
$dist(p_i, q_j)$	Distance from p_i to q_j
$\mathcal{S}^{\mathbb{P}}$	Sample set from $\mathbb{P}$
ρ	Sampling ratio, $0 \leq \rho \leq 1$
$\mathcal{R}^{\mathbb{P}}$	Set of selected pivots from $\mathbb{P}$
r_i	A pivot in $\mathcal{R}^{\mathbb{P}}$
V_i	A Voronoi-Cell of r_i
$HP(V_i, V_j)$	Hyperplane dividing two adjacent cells
$V_i.core$	Core points of a Voronoi-Cell V_i
$V_i.support$	Support set of Voronoi-Cell V_i
$corDist(V_i)$	Core-distance of a Voronoi-Cell V_i
$supDist(V_i)$	Support-distance of a Voronoi-Cell V_i
$\mathcal{P}^{\mathbb{P}}$	Set of partitions from $\mathbb{P}$
$\mathcal{P}_i^{\mathbb{P}}$	Subset from $\mathbb{P}$, having r_i as its closest pivot
$U(\mathcal{P}_i^{\mathbb{P}})$	Maximum dist. from r_i to the points of $\mathcal{P}_i^{\mathbb{P}}$
$L(\mathcal{P}_i^{\mathbb{P}})$	Minimum dist. from r_i to the points of $\mathcal{P}_i^{\mathbb{P}}$
$MBR(\mathcal{P}_i^{\mathbb{P}})$	MBR covering the points of $\mathcal{P}_i^{\mathbb{P}}$
$minDist()$	Minimum distance between two elements
$MmDist()$	max. min. dist. between two partitions

4. Voronoi-Diagram based partitioning technique in SpatialHadoop

In SpatialHadoop, the *Partitioning* phase of the indexing algorithm runs in three steps [1,4]. The first step computes the number of desired partitions x based on file size and HDFS block capacity, which are both fixed for all partitioning techniques. The second step reads a random sample (*Sampling*), with a sampling ratio ρ, from the input file and uses this sample to partition the space (*Space subdivision*) into x cells/partitions, such that the number of sample points in each partition is at most $\lfloor s/x \rfloor$, where s is the sample size. Finally, the third step partitions the file by assigning each point to one or more partitions (*Indexing*), i.e., every partition becomes a file that is duplicated to the number of nodes defined by the Hadoop cluster replication factor. Actually, SpatialHadoop supports seven spatial partitioning techniques: Grid, Quadtree, STR, STR+, k-d tree, Z-Curve and Hilbert-Curve.

Similarly, to include into SpatialHadoop the new data partitioning technique based on Voronoi-Diagram, we have implemented the following steps:

1. *Sampling*. A set $\mathcal{S}^{\mathbb{P}}$ of samples from an input dataset $\mathbb{P}$ is provided.
2. *Space subdivision*. A set $\mathcal{R}^{\mathbb{P}}$ of *pivots* is obtained from the sample set $\mathcal{S}^{\mathbb{P}}$, using some *pivot selection technique*.
3. *Indexing*. The points from the input dataset $\mathbb{P}$ are assigned to their closest *pivot* $r_i \in \mathcal{R}^{\mathbb{P}}$ and some properties of the pivot are calculated and stored in the global index.

4.1. Sampling large datasets

Sampling is an effective way to deal with large datasets, which attempts to find a small but representative profile of the dataset. The sample set is required to be small enough to satisfy the dataset size constraints and, and at the same time, the result of the sampling should be reliable and close enough to approximately represent the whole dataset. However, sampling methods cannot take into account the correlation among the data, hence it is hard to obtain the perfect sample. For example, if we have an input dataset that contains k clusters, ideally, the sample set should also contain k clusters. For this reason, ideal clustering result is difficult to obtain since the clusters cannot be determined easily.

In this research work, we will use three sampling methods: (1) *uniform random sampling* [1], it is the simplest and the most popular; (2) *partition-based sampling* [10], where the sampling is carried out according to a split of the dataset into a number of disjoint partitions that optimize a criterion function; and (3) *density-based sampling* [39], where distance concepts are managed for sampling to ensure space coverage and fit cluster shapes.

For uniform random sampling on large datasets, the size of the sample is usually set to a ratio between the sample dataset size and its original dataset size [1], that is $|\mathcal{S}^{\mathbb{P}}| = \rho \times |\mathbb{P}|$, where $0 \leq \rho \leq 1$ is the ratio of the sampled dataset. In [40], when $0.01 \leq \rho \leq 0.02$, the execution times are minimized for *K*NNJQ, since both small and large sample sizes tend to deteriorate the performance (i.e., small ratios are unable to accurately estimate dataset distribution and large ratios lead to high sampling overhead). In our experiments, we have chosen by default $\rho = 0.01$ (1%), since it was the best ratio value for real datasets when *K*NNJQ is executed [40], also for the *K*CPQ performance [21], and it produces high quality partitions in SpatialHadoop [4]. To generate the random sample efficiently when the input file is very large, SpatialHadoop provides a MapReduce job that scans all records and outputs each one with a probability of 1% ($\rho = 0.01$).

For *partition-based sampling*, k-means has also been successfully used as a preprocessing sampling step for sophisticated and expensive clustering techniques. It is executed with $k = |\mathcal{S}^{\mathbb{P}}|$, where $|\mathcal{S}^{\mathbb{P}}|$ is the desired sample size of $\mathbb{P}$, such as $|\mathcal{S}^{\mathbb{P}}| \ll |\mathbb{P}|$ [10]. For this reason, we can use k-means++ [41] for sampling purposes. To generate this kind of sample efficiently from a large dataset, we have implemented a MapReduce job, where the input dataset is split into a number of necessary parts to fit in the main memory of the mappers. Therefore, in the *map* phase, each $mapper_i$ performs the k-means++ algorithm from ELKI library [42] on its part with $k_i = s_i$, where s_i is the number of points resulting from applying the ratio ρ on the number of points that such $mapper_i$ receives. The final result of this MapReduce job is the combination of the partial results of applying k-means++ in each mappers.

The study of the theoretical analysis of error bounds of sampling to select the pivots for partitions in metric similarity join in MapReduce can be found in [43]. In addition, in [44], the study of the seeding methods for the k-means algorithm is presented, providing also the lower bound on the expected error of picking k initial centers for the k-means algorithm. According to [41], the k-means method does not perform well, since the random seeding will inevitably merge clusters together, and the algorithm will never be able to split them apart. The careful seeding method of k-means++ avoids this problem altogether, and it almost always attains the optimal results.

For *density-based sampling*, we will use the DENDIS algorithm [39], since it combines both DENsity and DIStance concepts to ensure space coverage and fit cluster shapes. In general, at each step of the algorithm a new point is added to the sample, choosing the furthest from the representative in the most important group. Like the previous sampling cases, we have implemented a MapReduce job, where the input dataset is also split into a number of necessary parts whose size can be processed by each mapper. Then, each $mapper_i$ executes the DENDIS algorithm on each part, using granularity $gr = 0.001$, as recommended in [39] to get a good accuracy. Finally, the individual results of each mapper are combined to obtain the final result.

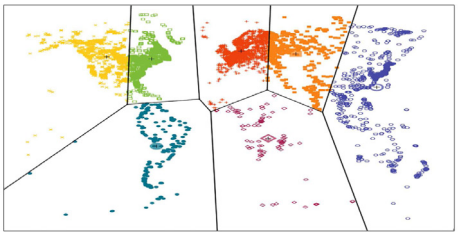
4.2. Pivot selection techniques for space subdivision

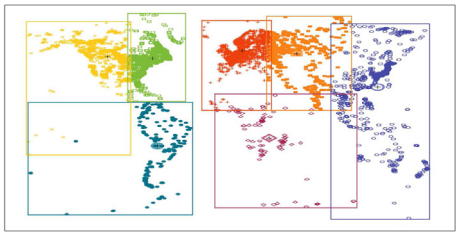
The Voronoi-Diagram based partitioning technique is well-known for maintaining data proximity, and it is especially appropriate for distance-based queries. For the creation of Voronoi-Diagrams, the method to select suitable pivots is very important and therefore, in the *Partitioning* process of the *Preprocessing* step (see Fig. 1) a module for selecting a set of pivots should be executed. In [7] three pivot selection strategies are proposed: random selection, furthest selection and k-means selection. Random selection was faster than k-means, but during the *K*NN join phase, the performance of k-means selection was better. For this reason, we have adapted *random selection* and *k-means selection* strategies to be included in SpatialHadoop. For the *random* selection technique, $\lfloor |\mathcal{S}^{\mathbb{P}}|/k \rfloor$ random sets of points are generated, then for each set, the total sum of the distances between every two points are computed and the points from the set with the largest total sum of distances are chosen as the pivots.

Taking into account the results of [7] and [22], the use of a clustering algorithm improves the quality of the selected pivots for splitting the whole dataset more evenly, and the partition-based and density-based clustering algorithms are the most appropriate for spatial big data [45]. *Partition-based clustering* attempts to directly decompose the dataset into a set of disjoint clusters. More specifically, this type of clustering algorithms attempts to determine an integer number of partitions that optimize a certain criterion function. On the other hand, the key idea of *density-based clustering* is to group neighboring objects of a dataset into clusters based on density conditions.

For the partition-based clustering category we have chosen the k-means algorithm [33], leading to the *k-means* selection technique. We have used the best recommendation for the k-means family in ELKI library [42], this is *Sort-Means* [46], which accelerates k-means, exploiting the triangle inequality and pairwise distances of means to prune candidate means (with sorting). Moreover, it uses k-means++ [41] to initialize means. When the k clusters have been generated, the center point of each cluster is chosen as a pivot for the Voronoi-Diagram based partitioning.

For the density-based clustering class, we have chosen the OPTICS algorithm [47], resulting the *OPTICS* selection technique. OPTICS is a density-based algorithm that attempts to overcome some of the drawbacks of its most famous counterpart DBSCAN [48]. The major weaknesses of DBSCAN are the inability to detect clusters in zones of varying density and, the choice of parameter values, for which it is very sensitive. The main difference between them is the ϵ value; in OPTICS it is an upper bound instead of an specific distance value. We have used the best recommendation for the density-based clustering family in ELKI library [42], this is OPTICSxi [49] with the implementation of





Partition	$MBR(\mathcal{P}_i^{\mathbb{P}})$	r_i	$\lvert\mathcal{P}_i^{\mathbb{P}}\rvert$	$L(\mathcal{P}_i^{\mathbb{P}})$	$U(\mathcal{P}_i^{\mathbb{P}})$
1	$MBR(\mathcal{P}_1^{\mathbb{P}})$	r_1	50000	0	30
2	$MBR(\mathcal{P}_2^{\mathbb{P}})$	r_2	60000	0	20
3	$MBR(\mathcal{P}_3^{\mathbb{P}})$	r_3	45000	0	60
4	$MBR(\mathcal{P}_4^{\mathbb{P}})$	r_4	35000	5	40
5	$MBR(\mathcal{P}_5^{\mathbb{P}})$	r_5	55000	5	35
6	$MBR(\mathcal{P}_6^{\mathbb{P}})$	r_6	50000	0	25
7	$MBR(\mathcal{P}_7^{\mathbb{P}})$	r_7	40000	5	60

Fig. 2. Overview of Voronoi-Index in SpatialHadoop.

FASTOptics [50]. In general terms, OPTICSxi generates a hierarchical classification of the clusters obtained when OPTICS is applied. The main parameters of OPTICSxi are ϵ (an upper bound of the distance to be considered), *minpts* (the minimum number of points required to form a cluster) and *xi* (contrast parameter that establishes the relative decrease in density). For our experiments, we have used $\epsilon = 2$, $minpts = 100$ and $xi = 0.025$. Since the output of the algorithm is a hierarchical structure, we have to find a level where at most k clusters are stored. When the k clusters have been selected, the center point of each cluster is chosen as a pivot.

4.3. Indexing data

The main idea of this step of Voronoi-Diagram based partitioning technique is to allocate each point of $\mathbb{P}$ to the partition with its closest pivot in $\mathcal{R}^{\mathbb{P}}$. That is, the points from the input dataset $\mathbb{P}$ are assigned to their closest pivot $r_i \in \mathcal{R}^{\mathbb{P}}$, leading to $|\mathcal{R}^{\mathbb{P}}|$ possible partitions. Moreover, some properties of the pivot r_i are calculated and stored for each partition, such as the number of points $|\mathcal{P}_i^{\mathbb{P}}|$, the $MBR(\mathcal{P}_i^{\mathbb{P}})$ which is the Minimum Bounding Rectangle (MBR) covering the points of $\mathcal{P}_i^{\mathbb{P}}$, $U(\mathcal{P}_i^{\mathbb{P}})$ and $L(\mathcal{P}_i^{\mathbb{P}})$. Fig. 2 illustrates the result of applying the Voronoi-Diagram based partitioning technique (Voronoi-Index) in SpatialHadoop. For more details, in the left chart, the data partitions, using the Voronoi-Diagram based partitioning technique from the selected pivots, are shown. The chart in the center shows the same data partitions, represented as pivots with their *MBR*s, in the same way that other spatial partitioning techniques are represented in SpatialHadoop. Finally, on the right, there is a table that summarizes the data available for each partition.

5. DJQ MapReduce algorithms in SpatialHadoop

In this section, we first present a MapReduce algorithm for *K*NNJQ in SpatialHadoop, adapted to the Voronoi-Diagram based partitioning technique. Next, an existing *K*CPQ MapReduce algorithm in SpatialHadoop is briefly reviewed and improved by Voronoi-Diagram based partitioning.

5.1. KNNJQ MapReduce Algorithm in SpatialHadoop

*K*NNJQ is an expensive operation, since it is a combination of the *K*NNQ and the join operation. The naive implementation of *K*NNJQ requires scanning $\mathbb{Q}$ once for each point $p_i \in \mathbb{P}$ (computing the distance between each pair of points from $\mathbb{P}$ and $\mathbb{P}$), easily leading to a complexity of $O(|\mathbb{P}| \times |\mathbb{Q}|)$ in the worst case. If points in datasets $\mathbb{P}$ and $\mathbb{Q}$ are sorted, the time complexity is reduced to $O(|\mathbb{P}| \times log_2|\mathbb{Q}|)$. Overall, for each $p_i \in \mathbb{P}$, to find the K nearest points in $\mathbb{Q}$ to p_i is reduced to $O(log_2|\mathbb{Q}|)$, since we use a plane-sweep algorithm to solve *K*NNQ, and $O(|\mathbb{P}| \times log_2|\mathbb{Q}|)$ in total for *K*NNJQ.

From the definition of *K*NNJQ, we can observe that it can be formulated on the basis of *K*NNQ. In [1], a *K*NNQ operation on SpatialHadoop was presented. The proposed *K*NNQ MapReduce algorithm is composed of the three steps: the *initial answer*, the *correctness check* and the *answer refinement*. Keeping this in mind, to develop a *K*NNJQ MapReduce algorithm in SpatialHadoop, we have followed the *K*NNJQ algorithm presented in [18]. The proposed *K*NNJQ algorithm in [18], on two datasets $\mathbb{P}$ and $\mathbb{Q}$, consists of four phases of MapReduce jobs: *information distribution* phase, *primitive computation* phase, *update lists* phase and *unify lists* phase. In the *information distribution* phase, a uniform partitioning of the dataset $\mathbb{Q}$ is made and the number of elements from $\mathbb{P}$ that are inside the partitions of $\mathbb{Q}$ are counted. Then, in the *primitive computation* phase, an initial response is provided by calculating the *K*NNQ for each point p_i of $\mathbb{P}$ with the points of $\mathbb{Q}$ of the partition in which p_i is located. Once this phase is completed, it is necessary to refine these initial *K*NN lists for each point of $\mathbb{P}$, if there have been found less than K neighbors, or if there are nearby partitions that overlap with the distance to each Kth nearest neighbor. All this refinement is done in the *update lists* phase where new non final *K*NN lists are obtained. Finally, in the *unify lists* phase, the merge of the all the *K*NN lists resulting from previous phases is performed, obtaining the final answer.

To adapt and implement the previous *K*NNJQ MapReduce algorithm in SpatialHadoop, we have to carry out several extensions and improvements that are the following: (1) The *information distribution* phase is implemented using the spatial partitioning techniques provided by SpatialHadoop, allowing us to use non-uniform partitions such as STR, Quadtree, Hilbert, etc. with the different improvements and particularities that they can offer. (2) The *information distribution* phase is performed only once for each dataset and is reused for further *K*NNJ queries. (3) SpatialHadoop indices are used in each of these phases to accelerate the processing of the partitions. (4) An implementation of new *K*NNQ based on a *plane-sweep* algorithm is carried out, which reduces the number of distance calculations obtaining a higher performance join operation. (5) Finally, a new *Repartitioning* phase is added as a first step to speed up the algorithm. This new phase uses Grid or Quadtree partitioning so as to split the largest partitions in smaller ones, dealing with skew data problems and getting smaller tasks.

Fig. 3 shows the phases of the proposed *K*NNJQ MapReduce algorithm in SpatialHadoop: *Repartitioning*, *Bin KNNJ*, *KNNJ on Overlapping Partitions* and *Merge Results*. First phase, called *Repartitioning*, uses an existing spatial partitioning technique, e.g., Grid or Quadtree, to subdivide the largest partitions from dataset $\mathbb{Q}$ and saves the information for further use in subsequent phases. Therefore, in the *map* function, the points of the largest partitions are sent to their corresponding reducer and in the *reduce* function, the partitioning technique is applied in order to have an index per repartitioned partition. Then in the *Bin KNNJ* phase (*information distribution* and *primitive computation* in [18]), a *Bin-Spatial Join* of the two input datasets, in which the join operator is the *K*NNQ, is accomplished. As described in Algorithm 1, in the *map* function of the *Bin KNNJ* phase, each point $p_i \in \mathbb{P}$ is combined with the partition in which it is located in the dataset $\mathbb{Q}$, so that in the *reduce* function, the *plane-sweep K*NNQ (PSKNNQ algorithm) of that point with the points of $\mathbb{Q}$ in the same partition is executed.

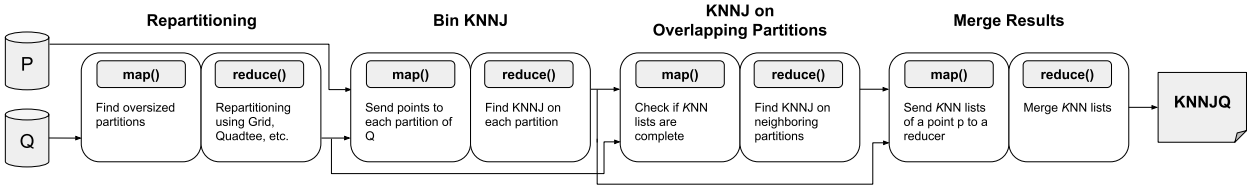


Fig. 3. Overview of the *K*NNJQ MapReduce algorithm in SpatialHadoop.

The result of this phase is a *KNN list* for each point $p_i \in \mathbb{P}$. Then a completeness check is made to find which of the previous *K*NN lists are not final and therefore it is necessary to continue with their processing. As shown in Algorithm 2, for the *KNNJ on Overlapping Partitions* phase (*update lists* in [18]), in the *map* function is checked, using the *GetOverlappedPartitions* function, if the previous *K*NN lists for each point $p_i \in \mathbb{P}$ contain less than *K* results (line 21) and also if there are neighboring partitions that overlap with the circular range, centered on $p_i \in \mathbb{P}$ and with radius the distance to the current *K*th nearest neighbor (line 24). These points are then sent together with the calculated neighboring partitions to the *reduce* phase where another *plane-sweep K*NNQ will be performed for each partition. Finally, the *Merge Results* phase (*unify lists* in [18]) consists of collecting the non final *K*NN lists of the two previous phases in the *map* function, obtaining the final *K*NNQ results for each point $p_i \in \mathbb{P}$ in the *reduce* function.

Algorithm 1 Bin *K*NNJ Algorithm

```
1: function MAP(p: point from ℙ or ℚ, 𝒫^ℚ: set of partitions from ℚ)
2:     partition ← FindPartition(𝒫^ℚ, p)
3:     output(partition.id, p)

4: function REDUCE(partitionId: current partition, PQ: set of points in partition, K: number of neighbors)
5:     P ← GetPointsFromP(PQ)
6:     Q ← GetPointsFromQ(PQ)
7:     for all p ∈ P do
8:         Initialize(KNNList, K)
9:         KNNList ← PSKNNQ(Q, p, K)
10:        output(p, KNNList)
```

Algorithm 2 *K*NNJ on Overlapping Partitions Algorithm

```
1: function MAP(p: point from ℙ or ℚ, 𝒫^ℚ: set of partitions from ℚ, K: number of neighbors)
2:     origin ← IsFromPorQ(p)
3:     if origin is from Q then
4:         partition ← FindPartition(𝒫^ℚ, p)
5:         output(partition.id, p)
6:     else
7:         overlappedParts ← GetOverlappedPartitions(𝒫^ℚ, p, K)
8:         for all partition ∈ overlappedParts do
9:             output(partition.id, p)

10: function REDUCE(partitionId: current partition, PQ: set of points in partition, K: number of neighbors)
11:    P ← GetPointsFromP(PQ)
12:    Q ← GetPointsFromQ(PQ)
13:    for all p ∈ P do
14:        Initialize(KNNList, K)
15:        KNNList ← PSKNNQ(Q, p, K)
16:        output(p, KNNList)

17: function GetOverlappedPartitions(𝒫^ℚ: set of partitions from ℚ, p: point from ℙ, K: number of neighbors)
18:    KNNList ← GetKnnList(p)
19:    nnNumber ← KNNList.size
20:    radius ← GetKthDistance(KNNList)
21:    while nnNumber < K do
22:        radius ← Increase(radius)
23:        nnNumber ← GetNumberOfNeighbors(𝒫^ℚ, p, radius)
24:    overlappedPartitions ← RangeQuery(𝒫^ℚ, p, radius)
25:    return overlappedPartitions
```

Voronoi-Diagram based partitioning can be incorporated, as shown in Fig. 4(a), into the proposed *K*NNJQ MapReduce algorithm in two ways: (1) performing the *initial Partitioning* process of the datasets in the *Preprocessing* step (see Fig. 1), and/or (2) subdividing the partitions from $\mathbb{Q}$ in the *Repartitioning phase* individually and then using its properties on the *KNNJ on Overlapping Partitions* phase. With the first one, we can take advantage of the characteristics of this technique globally, using the defaults parameters given by SpatialHadoop, in the same way that it is done for any built-in query. For the second way, we can accelerate the *K*NNJQ processing by decomposing the initial partitioning, by using the Voronoi-Diagram based partitioning technique, in smaller partitions given a maximum number of elements to solve skew data problems (*Repartitioning* phase) and reduce the number and size of the tasks of the *Bin KNNJ* and *KNNJ on Overlapping Partitions* phases. Furthermore, when calculating the overlapping partitions, the coordinates of each pivot r_i and the $U(\mathcal{P}_i^{\mathbb{P}})$ and $L(\mathcal{P}_i^{\mathbb{P}})$ values can be used to get better performance and accuracy than using only the *MBR* of each partition $\mathcal{P}_i^{\mathbb{P}}$, $MBR(\mathcal{P}_i^{\mathbb{P}})$. Fig. 4(b) shows that only the shaded part can contain points within the $MBR(\mathcal{P}_i^{\mathbb{P}})$, and therefore there is no overlap with the distance of the current *K*th nearest neighbor of p_i.

In our *K*NNJQ MapReduce algorithm, for the theoretical analysis, we divide the datasets $\mathbb{P}$ and $\mathbb{Q}$ into *w* partitions according to the Voronoi-Diagram based partitioning technique, where $\mathcal{P}_i^{\mathbb{P}} \in \mathbb{P}$; $\mathcal{P}_i^{\mathbb{Q}} \in \mathbb{Q}$; $1 \leq i \leq w$; $\mathcal{P}_i^{\mathbb{P}} \cap \mathcal{P}_j^{\mathbb{P}} = \emptyset$; $\mathcal{P}_i^{\mathbb{Q}} \cap \mathcal{P}_j^{\mathbb{Q}} = \emptyset$; $(i \neq j)$. When the *i*th partition ($\mathcal{P}_i^{\mathbb{P}}$ or $\mathcal{P}_i^{\mathbb{Q}}$) is sorted and each partition is concurrently processed by a computing node, the time complexity of the parallel *K*NNJQ is the maximum time of a slow computing node that handles more data than the other nodes. Thus, the time complexity is $max_{1 \leq i \leq w}\{O(|\mathcal{P}_i^{\mathbb{P}}| \times log_2|\mathcal{P}_i^{\mathbb{Q}}|)\}$. If the datasets $\mathbb{P}$ and $\mathbb{Q}$ can be well divided, the running time of the worst partition will be reduced. Theoretically speaking, shortening processing time of the worst partition tends to speed up the performance of *K*NNJQ MapReduce algorithms. Moreover, we have to take into account the communication overhead, according to [7] it is $O(|P| + |Q| + |RepQ| \times w)$, being $|RepQ|$ the total number of replications for the whole dataset $\mathbb{Q}$ that is needed for the computation of *K*NNJQ.

5.2. *K*CPQ MapReduce Algorithm in SpatialHadoop

The theoretical analysis for *K*CPQ is similar to the one of *K*NNJQ, since in the worst case, each point of one dataset ($\mathbb{P}$) is combined against each point of the other dataset ($\mathbb{Q}$), computing the distance between each pair of points and leading to a complexity of $O(|\mathbb{P}| \times |\mathbb{Q}|)$. If points in datasets $\mathbb{P}$ and $\mathbb{Q}$ are sorted, as is in our case to apply an efficient plane-sweep algorithm for *K*CPQ [17], the time complexity is reduced to $O(|\mathbb{P}| \times log_2|\mathbb{Q}|)$.

In general, the *K*CPQ MapReduce algorithm [20,21] in SpatialHadoop consists of a MapReduce job as is described in Algorithm 3. The *map* function aims to find the *K*CP between each local pair of partitions from $\mathbb{P}$ and $\mathbb{Q}$ with a *plane-sweep K*CPQ algorithm and the result is stored in a binary max heap (called *LocalKMaxHeap*). The *reduce* function aims to examine the candidate pairs of points from each *LocalKMaxHeap* and return the final set of

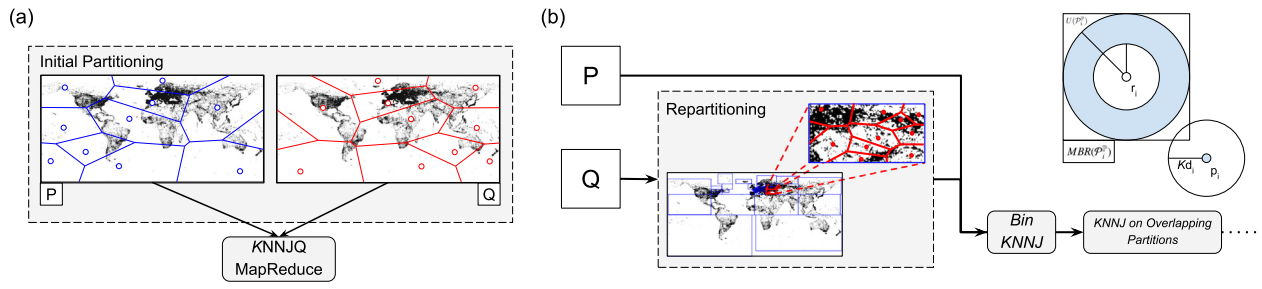


Fig. 4. Using Voronoi-Diagram based partitioning on the *initial partitioning* of the datasets (a) and in the *repartitioning* and *KNNJ on Overlapping Partitions* phases (b).

the K closest pairs in another binary max heap (called *GlobalK-MaxHeap*). To improve this approach, for reducing the number of possible combinations of pairs of partitions, we need to find in advance an upper bound of the distance value of the Kth closest pair of the joined datasets, called β. This β computation (lines 14–23) can be carried out by sampling globally from both datasets or by sampling locally for an appropriate pair of partitions and, then executing a *plane-sweep* KCPQ algorithm [17] (PSKCPQ, see Fig. 5(a)) over both samples. The *filter function* (PartitionsFilter) takes as input each combination of pairs of partitions in which the input set of points are partitioned and the distance value β, and it is used to prune pairs of partitions which have minimum distances ($minDist(MBR(\mathcal{P}_i^{\mathbb{P}}), MBR(\mathcal{P}_j^{\mathbb{Q}}))$ [16]) larger than β.

Using Voronoi-Diagram based partitioning, as shown in Fig. 5(a), the KCPQ MapReduce algorithm can be improved by modifying its local β computation and the *filter function*. For the computation of β, the most appropriate partitions, where an initial KCPQ is performed, are those whose pivots are closer to each other and have both higher density of points and area of intersection. Fig. 5(b) shows that for each partition $\mathcal{P}_i^{\mathbb{P}}$ of this partitioning technique, we have both its $MBR(\mathcal{P}_i^{\mathbb{P}})$ and its $U(\mathcal{P}_i^{\mathbb{P}})$ and $L(\mathcal{P}_i^{\mathbb{P}})$ values, allowing to detect areas of the former in which there are no points.

For the *filter function* (PartitionsFilter) two new distances metric can be used, the minimum distance between two pivots from two different partitions $minDist(r_i, r_j)$ and the maximum minimum distance between two partitions $MmDist(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$, they are exposed in Definitions 4 and 5, respectively. Therefore, as shown in Fig. 5(b), this function prunes pairs of partitions which have $MmDist(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$ larger than β, as we can see in the pruning Rule 1.

Definition 4. minimum distance between two pivots, $minDist(r_i, r_j)$

Given two pivots, $r_i \in \mathcal{R}^{\mathbb{P}}$ and $r_j \in \mathcal{R}^{\mathbb{Q}}$ $i \neq j$ that generate two partitions $\mathcal{P}_i^{\mathbb{P}}$ and $\mathcal{P}_j^{\mathbb{Q}}$, the minimum distance between two pivots, $minDist(r_i, r_j)$, is defined as

$$minDist(r_i, r_j) = dist(r_i, r_j) - U(\mathcal{P}_i^{\mathbb{P}}) - U(\mathcal{P}_j^{\mathbb{Q}})$$

Definition 5. maximum minimum distance between two partitions, $MmDist(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$

Given two partitions, $\mathcal{P}_i^{\mathbb{P}}$ and $\mathcal{P}_j^{\mathbb{Q}}$ $i \neq j$, the maximum minimum distance between two partitions, $MmDist(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$, is defined as

$$MmDist(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}}) = max\{minDist(MBR(\mathcal{P}_i^{\mathbb{P}}), MBR(\mathcal{P}_j^{\mathbb{Q}})), minDist(r_i, r_j)\}$$

Rule 1. *Pair of Partitions Pruning*

Given two partitions $\mathcal{P}_i^{\mathbb{P}}$ *and* $\mathcal{P}_j^{\mathbb{Q}}$ $i \neq j$, *if* $MmDist(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}}) > \beta$, *then the pair of partitions* $(\mathcal{P}_i^{\mathbb{P}}, \mathcal{P}_j^{\mathbb{Q}})$ *can be pruned, because they do not have any pair of points with distance* $< \beta$.

The Rule 1 allows us to prune combinations of partitions from $\mathbb{P}$ and $\mathbb{Q}$, reducing the number of *map* tasks that the KCPQ MapReduce algorithm needs to perform to get the final result.

Algorithm 3 KCPQ MapReduce Algorithm

```
1:  function MAP(ℙ: set of points, ℚ: set of points, K: number of pairs)
2:      SortX(ℙ)
3:      SortX(ℚ)
4:      LocalKMaxHeap ← PSKCPQ(ℙ, ℚ, K)
5:      if LocalKMaxHeap is not empty then
6:          for all DistanceAndPair ∈ LocalKMaxHeap do
7:              output(null, DistanceAndPair)

8:  function COMBINE, REDUCE(null, 𝔻: set of DistanceAndPair, K: number of pairs)
9:      Initialize(GlobalKMaxHeap, K)
10:     for all DistanceAndPair ∈ 𝔻 do
11:         Insert(GlobalKMaxHeap, DistanceAndPair)
12:     for all candidate ∈ GlobalKMaxHeap do
13:         output(null, candidate)

14:     function CALCULATEβ(ℙ: set of points, ℚ: set of points, ρ: sampling ratio,
        K: number of pairs)
15:         SampledP ← SamplingSortX(ℙ, ρ × |ℙ|)
16:         SampledQ ← SamplingSortX(ℚ, ρ × |ℚ|)
17:         LocalKMaxHeap ← PSKCPQ(SampledP, SampledQ, K)
18:         if LocalKMaxHeap is full then
19:             βDistanceAndPair ← pop(LocalKMaxHeap)
20:             β ← βDistanceAndPair.Distance
21:             output(β)
22:         else
23:             output(∞)

24: function PARTITIONSFILTER(𝒫ℙ: set of partitions from ℙ, 𝒫ℚ: set of partitions
    from ℚ, β: upper bound distance)
25:     for all c ∈ 𝒫ℙ do
26:         for all d ∈ 𝒫ℚ do
27:             minDistance ← MinDistance(c, d)          ▷ MmDist for Voronoi
28:             if minDistance ≤ β then                  ▷ Rule 1
29:                 output(c, d)
```

For the theoretical analysis, in our KCPQ MapReduce algorithm, we divide the datasets $\mathbb{P}$ and $\mathbb{Q}$ into w and v partitions respectively, according to the Voronoi-Diagram based partitioning technique, where $\mathcal{P}_i^{\mathbb{P}} \in \mathbb{P}$; $1 \leq i \leq w$; $\mathcal{P}_i^{\mathbb{P}} \cap \mathcal{P}_j^{\mathbb{P}} = \emptyset$; $(i \neq j)$ and $\mathcal{P}_i^{\mathbb{Q}} \in \mathbb{Q}$; $1 \leq i \leq v$; $\mathcal{P}_i^{\mathbb{Q}} \cap \mathcal{P}_j^{\mathbb{Q}} = \emptyset$; $(i \neq j)$. When the partitions ($\mathcal{P}_i^{\mathbb{P}}$ and $\mathcal{P}_j^{\mathbb{Q}}$) are sorted, for applying the KCPQ plane-sweep algorithm [17], and each pair of partitions is concurrently processed by a computing node, the time complexity of the parallel KCPQ is the maximum time of a slow computing node that handles more data than the other nodes. Thus, the time complexity is $max_{1 \leq i \leq w, 1 \leq j \leq v}\{O(|\mathcal{P}_i^{\mathbb{P}}| \times log_2|\mathcal{P}_j^{\mathbb{Q}}|)\}$. Moreover, in this case, the communication overhead is $O(|P| + |Q|)$.

6. Improvements for *K*NNJQ

In this section, we first adapt the distance metrics and pruning rules [23] for KNNJQ MapReduce algorithm in SpatialHadoop,

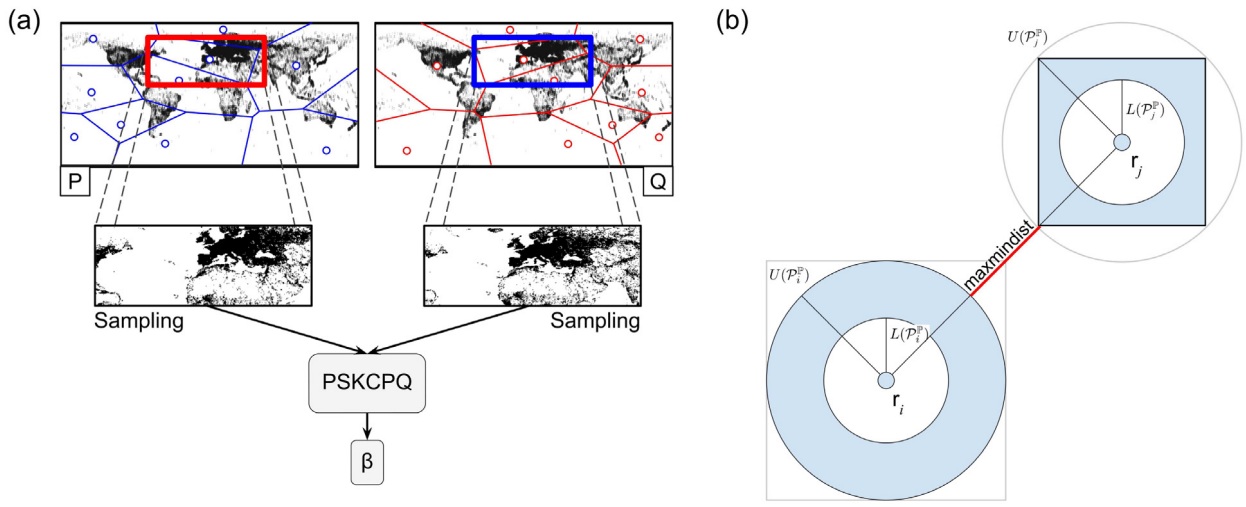


Fig. 5. β computation using Voronoi-Diagram based partitioning by sampling locally from both datasets (a), and partition refinement by its *MBR*, $U(\mathcal{P}_i^{\mathbb{P}})$ and $L(\mathcal{P}_i^{\mathbb{P}})$ properties and maximum minimum distance calculation (b).

and next, we incorporate into SpatialHadoop the *less data* technique [51] to try to move as less data as possible between computing nodes.

6.1. Pruning rules for KNNJQ

The points inside each Voronoi-Cell V_i are denoted as $V_i.core = \{p : p \in V_i\}$. The support set of a Voronoi-Cell V_i, called $V_i.support$, contains at least all data points that satisfy the following two conditions: (1) $\forall q \in V_i.support$, $q \notin V_i.core$, and (2) there exists at least one point $p \in V_i.core$ such that $q \in KNN(V_i, p, K)$. The $V_i.support$ must be sufficient to guarantee that the KNN of all core points in each cell V_i can be found among $V_i.core$ and $V_i.support$.

A large number of support points increases the computation costs per partition since many more points must be searched. To minimize the number of support points, in [23] two distance metrics and two pruning rules were defined.

The *core-distance* of a given Voronoi-Cell V_i represents the maximum distance from a core point p of V_i to its Kth nearest core neighbor q. It defines an upper bound on the distance between any core point of V_i and the possible support points. That is, given a point q outside V_i, it is guaranteed not to be a support point of V_i if its distance to any core point of V_i is larger than the $corDist(V_i)$.

Definition 6. core-distance of V_i, $corDist(V_i)$, [23]
$corDist(V_i) = max(dist(p, q)) \quad \forall p, q \in V_i.core$ *where* $q \in KNN(V_i, p, K) \in V_i.core$

The *support-distance* takes the pivot r_i of cell V_i into consideration, and it represents the maximum distance of a possible support point of V_i to the pivot r_i of V_i.

Definition 7. support-distance of V_i, $supDist(V_i)$, [23]
$supDist(V_i) = max(dist(p, r_i) + dist(p, q)) \; \forall p, q \in V_i.core$ *where* $q \in KNN(V_i, p, K) \in V_i.core$

Now, we will remind the two pruning rules proposed in [23] at Voronoi-Cell and point levels. The first pruning rule, Rule 2, which is applied in the *map* function of the *KNNJ on Overlapping Partitions* phase, avoids unnecessary data duplication (Algorithm 4, line 4), and also reduces the number of Voronoi-Cells each point must be checked against when mapping points to support sets (Algorithm 4, line 17).

Rule 2. *Support Cell Granularity Pruning, [23]*
Given two Voronoi-Cells V_i, V_j and their corresponding pivots r_i, r_j, $i \neq j$, if the $supDist(V_i) \leq dist(r_i, r_j)/2$, then V_j does not contain any support points of V_i.

The second one, Rule 3, allows us to prune, in the *map* phase of the *KNNJ on Overlapping Partitions* phase (Algorithm 4, line 7), the points of the support cells that are not part of any partial KNN list. This allows to reduce even more the shuffled data (fewer points are transferred to the *reduce* phase) and the complexity of the final KNN calculation for each point (the size of the set of support points is smaller).

Rule 3. *Support Point Granularity Pruning, [23]*
Given any point $p \in V_i$, $q \in V_j$, , $i \neq j$, if $dist(q, HP(V_i, V_j)) \geq corDist(V_i)$, then $q \notin KNN(V_j, p, K)$.

That is, Rule 3 allows us to prune points within the support cells that have not been already discarded by Rule 2. Furthermore, according to [52], the following lower bound can be used in place of the exact value of $dist(q, HP(V_i, V_j))$ in pruning Rule 3.

Definition 8. *Lower bound of $dist(q, HP(V_i, V_j))$*, [23]
Given two Voronoi-Cells V_i and V_j and their corresponding pivots r_i, r_j, $i \neq j$, and a point $q \in V_j$, $dist(q, HP(V_i, V_i)) \geq \frac{dist(q,r_i) - dist(q,r_j)}{2}$

Thanks to Definition 8 we can use a lower bound whose calculation is less complex than $dist(q, HP(V_i, V_j))$ leading to the pruning Rule 4, which reduces the calculation time, preventing it from penalizing the application of this pruning rule.

Rule 4. *Support Point Granularity Pruning by a Lower bound, [23]*
Given any point $p \in V_i$, $q \in V_j$, , $i \neq j$, if $\frac{dist(q,r_i) - dist(q,r_j)}{2} \geq corDist(V_i)$, then $q \notin KNN(V_j, p, K)$.

6.2. Less data technique

In [51], the *less data* technique is introduced for All-KNNQ in order to reduce the size of the shuffled data and the size of the output data of the *KNNJ on Overlapping Partitions* phase. Applying this technique in our KNNJQ MapReduce algorithm, each computing node will calculate and return a KNN list for every query

Algorithm 4 Improved *K*NNJ on Overlapping Partitions Algorithm

```
1: function MAP(p: point from ℙ or ℚ, 𝒫^ℚ: set of partitions from ℚ, K: number of neighbors)
2:     origin ← IsFromPorQ(p)
3:     if origin is from Q then
4:         filteredParts ← PrunePartitions(𝒫^ℚ, p, K)                ▷ Rule 2
5:         partition ← FindPartition(filteredParts, p)
6:         if partition is not NULL then
7:             if PrunePoint(filteredParts, p) == false then          ▷ Rule 4
8:                 output(partition.id, p)
9:     else
10:        overlappedParts ← GetOverlappedPartitions(𝒫^ℚ, p, K)
11:        for all partition ∈ overlappedParts do
12:            output(partition.id,p)

13: function GetOverlappedPartitions(𝒫^ℚ: set of partitions from ℚ, p: point from ℙ, K: number of neighbors)
14:    KNNList ← GetKnnList(p)
15:    nnNumber ← KNNList.size
16:    radius ← GetKthDistance(KNNList)
17:    supParts ← GetSupportPartitions(𝒫^ℚ, p, radius)               ▷ Rule 2
18:    nnNumber ← GetNumberOfNeighbors(supParts, p, radius)
19:    while nnNumber < K do
20:        supParts ← GetSupportPartitions(supParts, p, radius)
21:        nnNumber ← GetNumberOfNeighbors(supParts, p, radius)
22:    return supParts
```

point in the *Bin KNNJ* phase, based on its local data. Then some additional phases are needed to exchange data among nodes and find possible misses of nearer neighboring points, while trying to move as less data as possible between nodes. In the original algorithm, every point, that it is still not finished, is moved to its *reducer* of the *KNNJ on Overlapping Partitions* phase with its *K*NN list. Therefore, it adds a significant load to the network, especially for large *K* values. We decided to replace the *K*NN list with the distance to the *K*th neighbor as a bound, which is really the only info needed in the *reducer*. The partial *K*NN lists will be finally merged on the last *Merge Results* phase.

Continuing with the idea of reducing the size of the data that is handled in the different phases of the algorithm, the pruning Rule 5 allows determining which of the *K*NN lists have turned out to be final.

Rule 5. *Final KNN List Pruning*

Given any point $p \in V_i$, $q \in KNN(V_i, p, K)$, *if* $\frac{dist(r_i, r_j)}{2} \geq dist(r_i, p) + dist(p, q)\ \forall\ V_j \cap V_i.support \neq \varnothing$, *then* $KNN(V_i, p, K)$ *is final.*

Therefore, with this new pruning rule (Rule 5) we can split the output of the *reducers* of the *Bin KNNJ* phase into different group of files (final *K*NN lists and non-final *K*NN lists) thus reducing the input data size of the *KNNJ on Overlapping Partitions* and *Merge Results* phases. As a consequence of this reduction on the input data, the size of the shuffled data between the *map* and *reduce* tasks of these phases is also considerably smaller.

7. Experimental results

This section provides the results of an extensive experimental study aiming at measuring and evaluating the efficiency of the *DJQ* MapReduce algorithms proposed in Sections 5 and 6. In particular, Section 7.1 describes the experimental settings for this performance study in SpatialHadoop. Sections 7.2 and 7.3 show experimentally the advantages of the use of sampling and space subdivision in the building of the Voronoi index. Section 7.4 presents all experiments for *K*NNJQ using the Voronoi-Diagram based partitioning technique, paying special attention to the results in the different phases that are needed to perform this DJQ, the increment of *K* value and the use of some improvements to reduce the execution time and the shuffling cost. These results are compared with Quadtree-based partitioning. Section 7.5 exposes all experiments related to *K*CPQ, comparing Quadtree, which is the best spatial partitioning method in SpatialHadoop for DJQs, with the proposed data partitioning technique, and analyzing the increment of *K* value. Section 7.6 evaluates the extensibility of the *DJQ* MapReduce algorithms by increasing the dataset sizes. Section 7.7 shows the scalability of the proposed *DJQ* MapReduce algorithms, varying the number of computing nodes. Finally, in Section 7.8 a summary of the most important conclusions from the experimental results is reported.

Table 2
Configuration parameters used in our experiments.

Parameter	Values (default)
K for *K*NNJQ	(10), 25, 50, 75, 100
K for *K*CPQ	1, 10, (10^2), 10^3, 10^4, 10^5
Sampling	Random, *k*-means++, DENDIS
Pivot selection	Random, *k*-means, OPTICS
% ℙ area, γ	25, 50, 75, (100)
Number of nodes	1, 2, 4, 6, 8, 10, (12)

7.1. Experimental setup

For the experimental evaluation, we have used real-world 2d point datasets to test our DJQ MapReduce algorithms in SpatialHadoop. We have used datasets from OpenStreetMap[1]: *LAKES* (*L*) which contains 8.4M records (8.6 GB) of boundaries of water areas (polygons), *PARKS* (*P*) which contains 10M records (9.3 GB) of boundaries of parks or green areas (polygons), *ROADS* (*R*) which contains 72M records (24 GB) of roads and streets around the world (line-strings), *BUILDINGS* (*B*) which contains 115M records (26 GB) of boundaries of all buildings (polygons), and *ROAD_NETWORKS* (*RN*) which contains 717M records (137 GB) of road networks represented as individual road segments (line-strings) [1]. To create sets of points from these five spatial datasets, we have transformed the MBRs of line-strings into points by taking the center of each MBR. In particular, we have considered the centroid of each polygon to generate individual points for each kind of spatial object.

The main performance measure that we have used in our experiments has been the total execution time (i.e., total response time), that represents the time spent by the execution of each distributed DJQ algorithm. Another performance metric used in our experiments is the shuffled data, which refers to the amount of information produced in the *mapper* tasks and moved to the nodes where the *reducer* tasks will run. This measurement has been used to obtain more information about the behavior of the different phases of *K*NNJQ and it is shown in Gigabytes (GB). Table 2 summarizes the configuration parameters used in our experiments. Default parameters (in parentheses) are used unless otherwise mentioned.

All experiments were conducted on a 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. Finally, we used the latest code available in the repositories of SpatialHadoop.[2]

In [40], three data partitioning strategies for managing the data skewness problem in *K*NNJQ were proposed. They depend on whether (1) the first dataset of the join is partitioned and the second dataset is split according to the boundaries of such

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

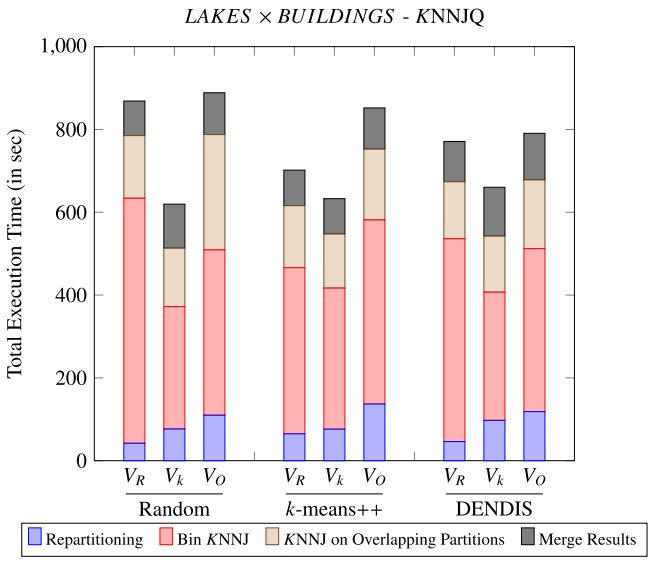


Fig. 6. *K*NNJQ cost (in sec) for the combination of the datasets, *LAKES* × *BUILDINGS*, considering different sampling methods and pivot selection techniques for $K = 10$.

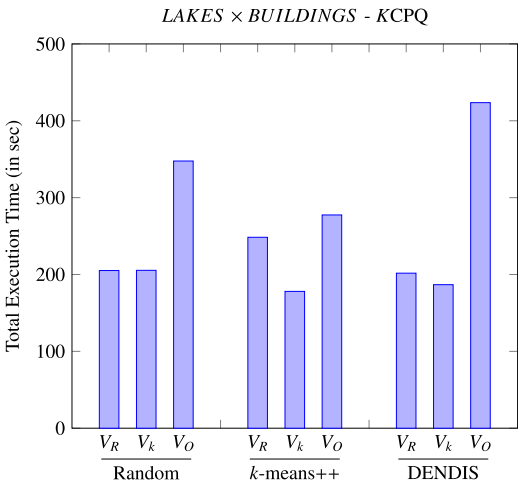


Fig. 7. *K*CPQ cost (in sec) for the combination of the datasets, *LAKES* × *BUILDINGS*, considering different sampling methods and pivot selection techniques for $K = 100$.

partition; (2) the second dataset is partitioned and the first is split based on that partition, or (3) the union of both datasets is partitioned. In our case, we have divided the largest dataset, trying to get the same number of points in each partition, and then the smallest dataset is partitioned according to the boundaries of each partition of the largest one. For instance, if we have *KNNJ*$(\mathbb{P}, \mathbb{Q}, K)$, such that $|\mathbb{P}| < |\mathbb{Q}|$, we divide $\mathbb{Q}$ into $|\mathcal{P}^{\mathbb{Q}}|$ partitions, trying to get the same number of points in each one of them, and then $\mathbb{P}$ dataset is partitioned according to the boundaries of each partition of $\mathbb{Q}$, generating $\mathcal{P}^{\mathbb{P}}$ with $|\mathcal{P}^{\mathbb{Q}}|$ partitions.

7.2. Effect of sampling methods

During the *Partitioning* phase, in the *Sampling* step, we collect a set of samples (e.g., $|\mathcal{S}^{\mathbb{P}}| = 0.01 \times |\mathbb{P}|$) from the input dataset to capture its distribution as best as possible, since this sample set will affect query performance. In this experiment, we evaluate three sampling techniques for the building of the Voronoi index (**R**andom, **k**-means++ and **D**ENDIS) for *K*NNJQ (Fig. 6) and *K*CPQ (Fig. 7) by considering the three pivot selection techniques: Random (V_R), *k*-means (V_k) and OPTICS (V_O). Fig. 6 shows that on average *k-means*++ sampling exhibits the best global performance (execution time) for *K*NNJQ, although *Random* and *DENDIS* report good results with V_k. *Random* sampling is the fastest, but it has a great component of randomness that exists between two different executions of the same query. *DENDIS* needs more time than *k-means*++ to be run, since it requires many distance computations and consumes many resources in its implementation. For *K*CPQ, Fig. 7 reveals again that *k-means*++ sampling shows the best global performance, mainly for V_k. *Random* and *DENDIS* with V_k get good results as well, but they have the previous drawbacks. The main conclusion of these results indicates that *k-means*++ is the best sampling technique (partition-based sampling) for the creation of Voronoi indexes in SpatialHadoop for DJQs.

7.3. Effect of space subdivision and indexing

In this experiment, we will compare our new proposed Voronoi-Diagram based partitioning algorithms with the *Quadtree* (Q) built-in partitioning technique which has shown to obtain the best performance results with the different spatial queries present in SpatialHadoop [1,4,20,21]. We will consider the *k*-means++ sampling (the best one of the previous experiment), and the three pivot selection techniques: *random selection* ($Voronoi_{kR}$, V_{kR}), *k-means selection* ($Voronoi_{kk}$, V_{kk}) and *OPTICS selection* ($Voronoi_{kO}$, V_{kO})) for the *Space subdivision* step, and the *Indexing data* step.

In Fig. 8, the partitioning cost of different datasets is shown with respect to the execution time, for both the *Space subdivision* and *Indexing* phases. The first conclusion we can draw is that the execution times for $Voronoi_{kR}$ and *Quadtree* grow similarly as the size of the datasets is increased. For $Voronoi_{kk}$ the increase in execution times is larger, since a *k-means* algorithm is used in the *Space subdivision* phase. This *k-means* algorithm takes longer times to converge towards a solution as the size of the datasets increases. The costliest pivot selection technique is $Voronoi_{kO}$, because the execution of *OPTICSxi* clustering algorithm is more expensive than *k-means*, being the number of partitions smaller. Finally, $Voronoi_{kR}$ presents the fastest execution times, mainly because it consumes the smallest time in the *Indexing* phase of the data, since in the *Space subdivision* phase the times are very similar to those of *Quadtree*. In Table 3, we can observe information of data distribution (points per partition) about the partitioning of *RN* dataset for each partitioning technique. On one hand, $Voronoi_{kO}$ presents a higher mean value due to having a lower number of partitions than the other techniques. On the other hand, $Voronoi_{kk}$ has a much lower standard deviation that allows better handling of data skew problems by having a more proportional distribution of the points in all partitions. This metric provides information about the gap between the different partitioning techniques and how it affects the performance of the DJQs, since the skewed data is a main factor for the increasing of the execution time. In addition, this result is aligned with the behavior obtained in Figs. 6 and 7, where the best execution times are obtained by applying *k*-means++ algorithm, either in sampling or partitioning phases, confirming that the results are close to the optimal values.

7.4. KNNJQ experiments

These experiments for *K*NNJQ MapReduce algorithm aim to measure the variation of different parameters like the dataset sizes to be joined, partitioning techniques (V_{kk}, V_{kR}, V_{kO} and *Quadtree*) and the increment of *K* value.

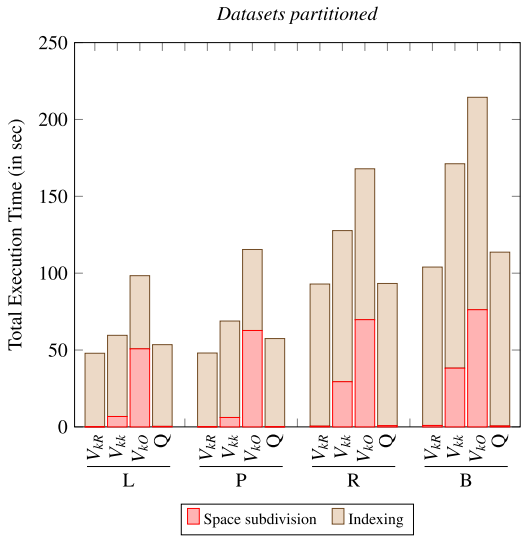


Fig. 8. Partitioning cost (total execution time) per phase, considering different partitioning techniques and datasets.

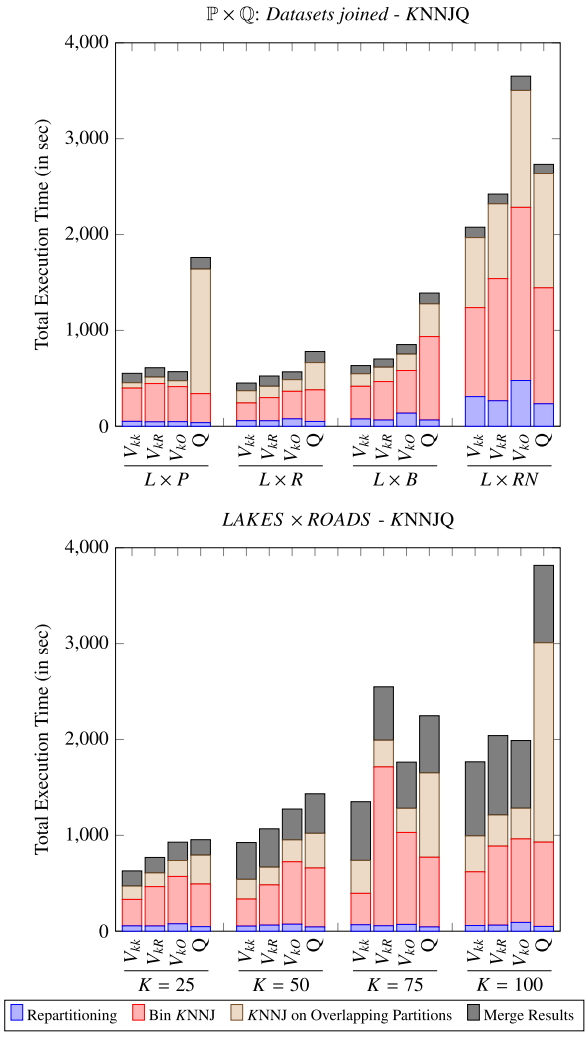


Fig. 9. Total execution time of *K*NNJQ, considering different partitioning techniques (upper) and varying the K values (lower).

Table 3
Information of data distribution (points per partition) of *RN* dataset per partitioning techniques.

	Num	Mean	Min	Max	Stdev
$Voronoi_{kk}$	512	1 400 486	19 914	3 684 694	623 909
$Voronoi_{kR}$	512	1 400 486	18 347	6 228 082	985 297
$Voronoi_{kO}$	72	9 959 011	1 149 113	40 703 435	8 512 796
Quadtree	430	1 667 555	218	4 275 451	1 130 277

7.4.1. Effect of pivot selection techniques

This experiment compares the three *pivot selection* techniques (**R**andom, **k**-means and **O**PTICS) with *k-means++* as the sampling method and *Quadtree* (Q) for the *K*NNJQ in SpatialHadoop, based on the execution time, in each of the phases. They are denoted as $Voronoi_{kk}$ (V_{kk}), $Voronoi_{kR}$ (V_{kR}) and $Voronoi_{kO}$ (V_{kO}). In Fig. 9, upper chart, the *K*NNJQ for the combination of different datasets ($L \times P$, $L \times R$, $L \times B$ and $L \times RN$) is shown for each *pivot selection* technique and for a fixed $K = 10$. We can observe that $Voronoi_{kk}$ exhibits the best performance in all cases. Moreover, *Quadtree* is much slower, especially in the *KNNJ on Overlapping Partitions* phase. This is due to the fact that with the three *Voronoi* techniques, every point of $\mathbb{P}$ is assigned to $\mathbb{Q}$ partition that contains at least K elements, so after the *Bin KNNJ* phase there are more final *KNN lists* and therefore the processing time of the next phase is reduced. Note that the *KNNJ on Overlapping Partitions* phase is usually more expensive if the number of final *KNN lists*, from the previous phase, is lower, because when the range query on the nearby partitions is executed, there is a large growth of the number of partitions to search for *K*NN candidates. Notice the high execution time needed for $L \times RN$ using V_{kO}, this is because the OPTICS algorithm does not generate a fixed number of clusters, but it depends strongly on the data distribution (and the number of clusters is less than k). In this figure we can also highlight that the differences in execution time between the four partitioning techniques are reduced with the combination of the larger dataset, $L \times RN$, mainly because the *Quadtree* technique returns more final *KNN lists*. As the volume and size of $\mathbb{Q}$ are much greater, the volume of points of $\mathbb{P}$ that fall into partitions of $\mathbb{Q}$ is also greater, obtaining final results that reduce the execution time of the *K*NNJQ algorithm. Another conclusion that can be obtained from the results is that *Quadtree* is the fastest while $Voronoi_{kk}$ is slower for the *Repartitioning* phase. This is due to the use of an algorithm based on *k-means* that makes the time increase slightly, in the same way to the *Indexing* time in previous experiment. However, thanks to this preprocessing, the best results are obtained, due to the good handling of the skewed data (e.g., the time spent in the *Bin KNNJ* phase is the smallest).

Moreover, similar behavior can be observed in Fig. 9, lower chart, where, as the K value is increased for the combination of the datasets, *LAKES* × *ROADS*. The execution time of the *KNNJ on Overlapping Partitions* phase is also higher. We have also to emphasize the high execution time needed for $K = 75$ using V_{kR}, this is mainly due to the random nature of the random selection technique. Note that the increase of the *Repartitioning* phase time for $Voronoi_{kk}$ is less than that shown in the *Indexing* process. This is due to the fact that the former is done within each partition using a MapReduce job, while the latter is carried out in the master node. Finally, in the *Merge Results* phase, we can observe how *Quadtree* exchanges more information than both *Voronoi* techniques, since in the previous phase more *KNN lists* have been generated for all the dataset combinations.

7.4.2. Effect of including the improvements

This experiment compares the best *K*NNJQ MapReduce algorithm in SpatialHadoop designed so far (V_{kk}), with the enhanced

version including all the improvements proposed in Section 6 (V_{kkI}), based on the execution time, in each of the phases. In Fig. 10, upper chart, the *K*NNJQ for the combination of different datasets ($L\times P$, $L\times R$, $L\times B$ and $L\times RN$) is shown for a fixed $K = 10$. We can observe that the $Voronoi_{kkI}$ exhibits the best performance in all cases. This is due to the fact that the execution times of phases 3 (*KNNJ on Overlapping Partitions*) and 4 (*Merge Results*) have been considerably reduced by the *pruning rules* (2 and 4) that eliminate points from the dataset $\mathbb{Q}$ that are not part of the final result and by the *less data* technique that decreases the size of the input set (only those points of $\mathbb{P}$ that have not finished) as well as the size of the shuffled data between the MapReduce phases.

Moreover, similar behavior can be observed in Fig. 10, lower chart, where, as the K value is increased for the combination of the datasets, *LAKES* × *ROADS*. The execution time of the V_{kkI} increases less than for V_{kk}. This time difference grows with the increment of the K value, due mainly to the increase in the size of the partial results (*K*NN lists). In the improved version, V_{kkI}, only non-final *K*NN lists are used in phases 3 and 4, causing that when K increases, the non-improved version works with more intermediate data.

These time differences are even larger when the size of the smallest dataset increases as can be seen in Fig. 11, upper chart. For the combination *ROADS* × *BUILDINGS* (72M points × 115M points), we observe how the times are higher for the unimproved version (V_{kkI} 2860 s vs. V_{kk} 3746 s), especially in the phases 3 (*KNNJ on Overlapping Partitions*) and 4 (*Merge Results*). This is shown in Fig. 11, lower chart, which shows that the size of the shuffled data of these phases is greater than double for the non-improved version. It should be noted that the calculation of the Rule 5 increases the time of the phase *Bin KNNJ*, although it is worth it for the best obtained results.

7.5. *K*CPQ experiments

These experiments aim to measure the behavior of the *K*CPQ MapReduce algorithm in SpatialHadoop, varying different parameters as the dataset sizes to be joined, the partitioning techniques and the values of K. In Fig. 12, upper chart, the *K*CPQ for a fixed $K = 100$ and for real spatial datasets ($L \times P$, $P \times R$, $R \times B$ and $B \times RN$) is shown with respect to the execution time for the different partitioning techniques ($Voronoi_{kk}$, $Voronoi_{kR}$, $Voronoi_{kO}$ and *Quadtree*). We can observe that the execution times in all partitioning techniques grow almost linearly as the size of the datasets is increased, except $Voronoi_{kO}$ that for $P \times R$ the time is very high due to mainly the high preprocessing cost. For *K*CPQ, the best partitioning technique is *Quadtree*, which is approximately 18% faster than $Voronoi_{kk}$. Moreover, for the combinations of $L \times P$ and $P \times R$, $Voronoi_{kk}$ is slightly faster than *Quadtree* (e.g., for $L \times P$ $Voronoi_{kk}$ is 14 s faster than Quadtree), but for the combinations of the biggest datasets ($R \times B$ and $B \times RN$) *Quadtree* is the fastest, e.g., for $B \times RN$ *Quadtree* is 18% (254 s) faster than $Voronoi_{kk}$. That is, $Voronoi_{kk}$ exhibits smaller runtime values for smaller dataset sizes, since it produces a slightly larger number of partition combinations (e.g., 24 vs. 23 partition pairs for $L\times P$) that are better distributed in tasks for this cluster of nodes. But for big dataset sizes, *Quadtree* is the fastest for *K*CPQ, since it minimizes the number of partitions for each dataset and the number of the ones that overlap between each other. For instance, for the combination of $B \times RN$, *Quadtree* obtains $78 \times 430 = 33540$ possible pairs of partitions, remaining 711 pairs of partitions (2%) after applying the Rule 1, with a total execution time of 1220 s In the case of $Voronoi_{kk}$, it generates $81 \times 512 = 41472$ pairs of partitions, remaining 1191 pairs of partitions (2.8%) after applying Rule 1, with a total execution time of 1474 s, that is slightly

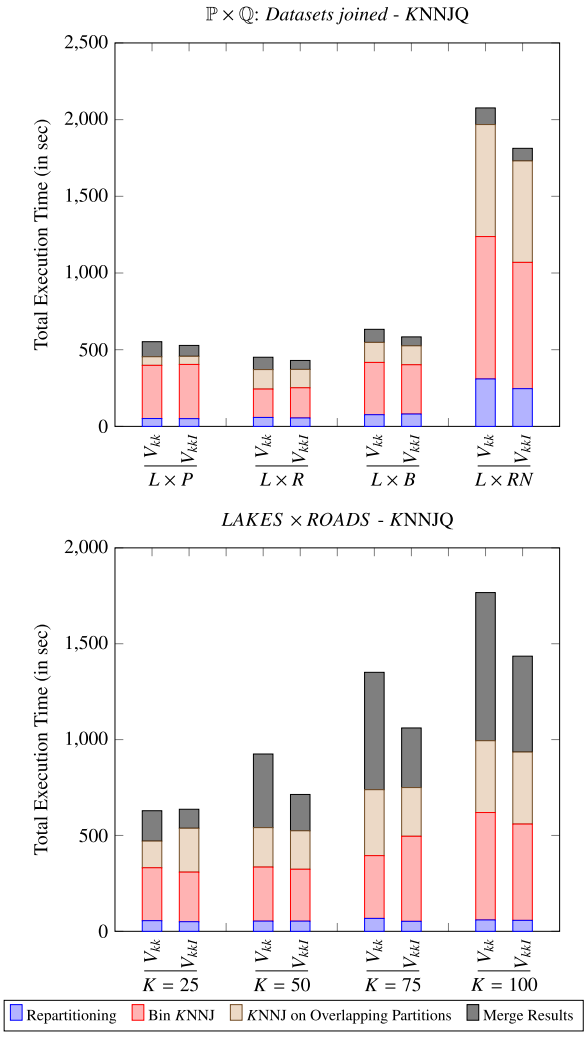


Fig. 10. Total execution time of *K*NNJQ, considering the improvements (upper) and varying the K values (lower).

higher than for Quadtree due to the increase on the number of *map* tasks. Finally, $Voronoi_{kO}$ shows the worst results, noting that the indexing time of $Voronoi_{kO}$ is much higher and the number of partitions is smaller. Fig. 12, lower chart, shows the effect of increasing the K value for the combination of the biggest datasets (*BUILDINGS* × *ROAD_NETWORKS*) for *K*CPQ. This experiment shows that the total execution time grows slowly as the number of results to be obtained (K) increases. All partitioning techniques report very stable execution times, even for large K values (e.g., $K = 10^5$), although, we can see that *Quadtree* still exhibits the lowest execution times.

7.6. Extensibility varying the ℙ dataset area

In this experiment, we evaluate the extensibility of the proposed DJQ MapReduce algorithms (*K*NNJQ and *K*CPQ), considering different percentages (γ) of the $\mathbb{P}$ dataset and keeping $\mathbb{Q}$ fixed. We aim to assess the performance of DJQs when the amount of data is massive, varying the smallest dataset ($\mathbb{P}$) by executing a Window Query centered on the MBR of $\mathbb{P}$ with a percentage (γ) of the original MBR. In the case of *ROADS* and the γ values of 25% 50% 75% and 100%, we have obtained a percentage of points of 2%, 27% 70% and 100% from the original dataset $\mathbb{P}$.

In Fig. 13 it is shown that, for *K*NNJQ, when the size of the data is small, *Quadtree* works better because the cost of the calculation

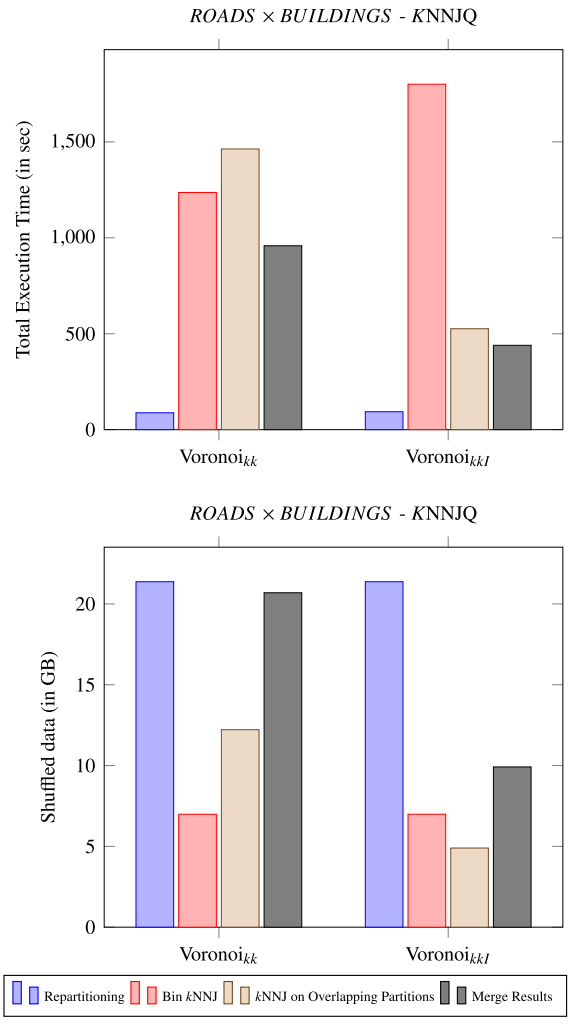


Fig. 11. *K*NNJQ cost per phase considering the improvements on the combination *ROADS* × *BUILDINGS*. Total execution time (upper) and shuffled data in GBytes (lower).

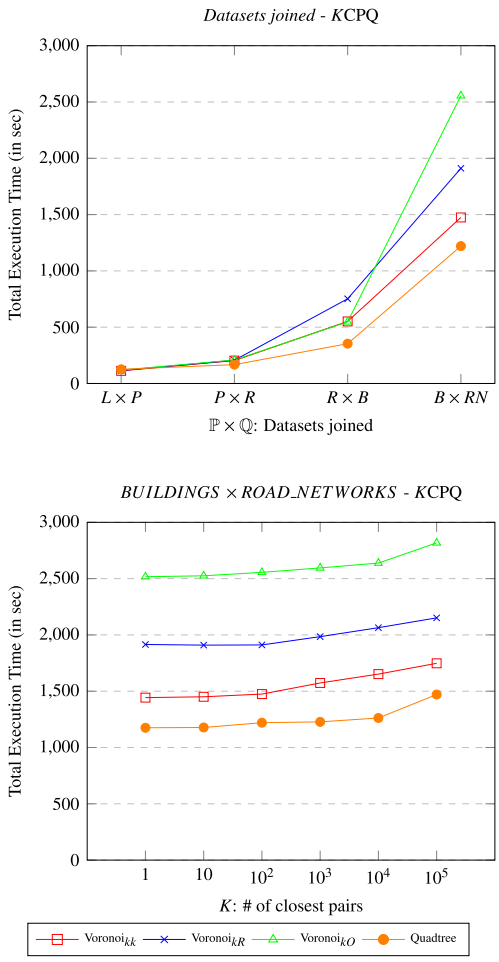


Fig. 12. Total execution time of *K*CPQ, considering different partitioning techniques (upper) and varying the K values (lower).

of rules is almost insignificant for the pruning data (very few points are pruned from the dataset). As the size of the query window increases, the time differences also increase for *Voronoi*$_{kkI}$, because although the running time of the *Bin KNNJ* phase is slightly higher for the calculation of the rules, the execution times of phases 3 and 4 decrease considerably thanks to the fact that the size of the input data (shuffled data) through the use of pruning rules is decreased.

For the case of *K*CPQ, Fig. 14 shows that *Voronoi*$_{kk}$ presents smaller execution times when the size of the datasets is smaller, since the pruning rule with *MmDist* works better and there is still a higher number of partitions. However, as shown in the experiments of Section 7.5, *Quadtree* minimizes the number of partitions and therefore obtains better results for high γ values.

7.7. Scalability varying the number of computing nodes

This new experiment aims to measure the speedup of the proposed DJQ MapReduce algorithms (*K*NNJQ and *K*CPQ), with respect to the number of computing nodes (η). To evaluate the scalability performance, we compare our best approach using the Voronoi-Diagram based partitioning technique to the same MapReduce algorithms using the *Quadtree* partitioning scheme.

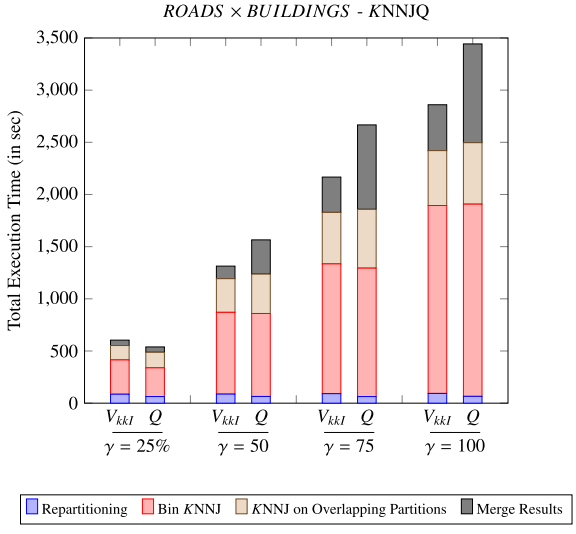


Fig. 13. *K*NNJQ cost (in sec) for the combination of the datasets, *ROADS* × *BUILDINGS*, considering different γ values for $K = 10$.

The upper chart of Fig. 15 shows the impact of different number of computing nodes on the performance of distributed *K*NNJQ algorithm, for *LAKES* × *PARKS* with the default configuration values. From this chart, we can conclude that the performance of our

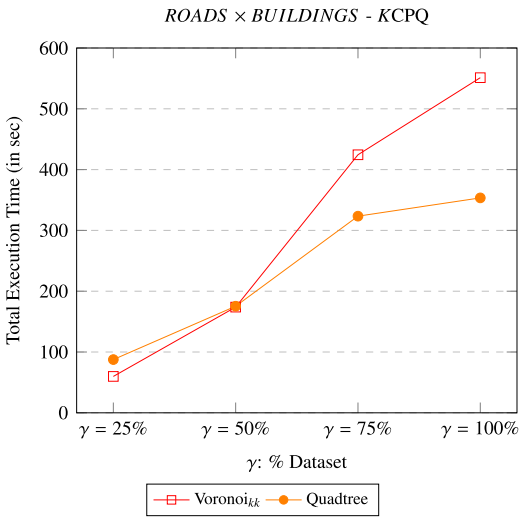


Fig. 14. Total execution time of *K*CPQ for the combination of the datasets, *ROADS* × *BUILDINGS*, considering different γ values for $K = 100$.

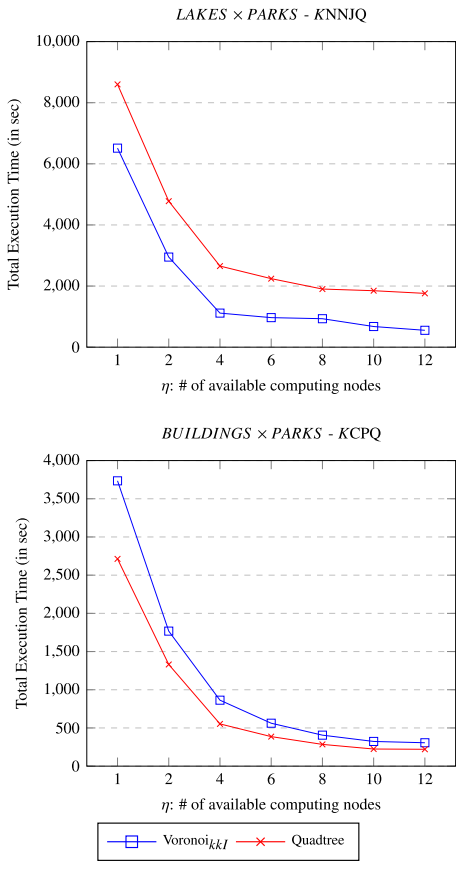


Fig. 15. Query cost with respect to the number of computing nodes η (Speed up).

approach has a direct relationship with the number of computing nodes. It could also be deduced that better performance would be obtained if more computing nodes are added, but when the number of computing nodes exceeds the number of *map* tasks no improvement is obtained. *Voronoi*$_{kk}$ is still showing a better behavior than *Quadtree*. In the lower chart of Fig. 15 shows smaller execution times for *K*CPQ than for *K*NNJQ, mainly since it is a less complex algorithm and it does not consist of several MapReduce phases. However, the trend of the behavior of both partitioning techniques for the combination *BUILDINGS* × *PARKS* is very similar to the one shown in *K*NNJQ, but exhibiting the lowest execution times for *Quadtree*.

7.8. Discussion of the results

The main conclusions extracted for this set of experiments on the proposed Voronoi-Diagram based partitioning techniques in SpatialHadoop for DJQ MapReduce algorithms are the following:

1. The best sampling technique to find a small but representative profile of the big spatial dataset for DJQ processing in SpatialHadoop is *k*-means++, which is a partition-based sampling method.
2. Using the *k*-means++ sampling, we have compared three clustering algorithms (Random, *k*-means and OPTICS) for the pivot selection. The partitioning execution times for V_{kR} are the smallest and grow almost linearly as the size of the datasets, while, for V_{kk}, this increment is larger due to the use of this clustering algorithm. The use of OPTICS, V_{kO}, is the slowest. But V_{kk} exhibits the best global performance in all cases for *K*NNJQ, because this combination of *k*-means algorithms indexes the data appropriately for the next phases in the *K*NNJQ MapReduce algorithm in SpatialHadoop, and the time consumed by *k*-means algorithm in the Repartitioning phase (it is a MapReduce job) is compensated by the gain in performance in subsequent phases of the query processing.
3. For *K*NNJQ (it follows a multiple nearest neighbor query processing schema), V_{kk} is faster than *Quadtree*, because it deals better with skewed data and it gets more final results in the *Bin KNNJ* phase.
4. The improved version of V_{kk}, V_{kkl}, has been designed to decrease considerably the execution time, especially in the *KNNJ on Overlapping Partitions* and *Merge Results* phases, by reducing the size of the input data, the shuffled data and the data that is handled in the *K*NN computation through the use of different pruning rules.
5. *Quadtree* outperforms all other techniques with respect to the execution time for the *K*CPQ (it follows a global query processing schema), although *Voronoi*$_{kk}$ or V_{kk} technique presents slightly better performance, for the combinations of the smallest datasets.
6. Each partitioning method is better for certain spatial queries since their performance depends strongly on the processing scheme. For our DJQs, *K*CPQ is favored by *Quadtree* partitioning and *K*NNJQ by *Voronoi*$_{kk}$ partitioning. For *K*NNJQ with *Voronoi*$_{kk}$ partitioning, the two datasets use the same partitions/pivots, so some properties and pruning rules can be applied on them. But for *K*CPQ with *Voronoi*$_{kk}$ partitioning, the two datasets use different partitions/pivots, so those same properties cannot be applied and the required time to repartition one of the datasets with the pivots of the other one will be very expensive. It means *K*CPQ with *Voronoi*$_{kk}$ partitioning is penalized, but with *Quadtree* partitioning *K*CPQ is favored, since *Quadtree* uses a regular decomposition of space, which reduces the number of partitions.
7. In the experiments of varying the γ values (extensibility), if the size of the MBR of $\mathbb{P}$ is very small compared to $\mathbb{Q}$, *Quadtree* presents a better behavior for *K*NNJQ than V_{kkl}

due to a lower efficiency of the pruning rules. But when the MBR is large enough, then V_{kkl} shows better performance than *Quadtree*. For the case of *K*CPQ, when γ is small, $Voronoi_{kkl}$ is slightly better than *Quadtree*, but for medium and large γ values, *Quadtree* gets the best performance.

8. Both $Voronoi_{kkl}$ and *Quadtree* show better performance when the number of computing nodes (η) is increased, but if there are not enough tasks available for a certain value of nodes, no performance improvements are obtained.

8. Conclusions and future work

Distance-Join Queries (DJQs) are important and common operations used in numerous spatial applications. DJQs are costly operations, since they combine spatial joins with distance-based search, and therefore, their efficient execution is a challenging task. For this reason, in this paper, a new data partitioning technique based on Voronoi-Diagrams in SpatialHadoop is designed and implemented. The best combination for the partitioning process is to use the *k*-means algorithm both in the sampling of datasets and in the space subdivision step (pivot selection), resulting in the V_{kk} variant. Improved *K*NNJQ and *K*CPQ MapReduce algorithms, using this new partitioning mechanism and the repartitioning technique, have also been proposed. KNNJQ MapReduce algorithm has been also improved by using adapted pruning rules and the less data technique to try to move as less data as possible between computing nodes, resulting in V_{kkl}. The execution of an extensive set of experiments on real-world datasets has demonstrated that distributed DJQ algorithms using Voronoi-Diagram based partitioning (V_{kk} is the best one) have shown excellent results in terms of running times and shuffled data, compared to other spatial partitioning techniques implemented already in SpatialHadoop (e.g., Quadtree). For *K*CPQ, *Quadtree* shows slightly better performance than V_{kk}, mainly for large dataset sizes. However, in the case of *K*NNJQ, the use of these new techniques to repartition the data leads to a great improvement in performance, especially through the use of *k*-means and the other improvements. Both V_{kkl} (*K*NNJQ) and Quadtree (*K*CPQ) show better performance when the number of computing nodes is increased. Finally, this experimental study on both real-world datasets demonstrates that our proposed data partitioning based on Voronoi-Diagrams for *K*NNJQ and *K*CPQ MapReduce algorithms are efficient, robust and scalable in SpatialHadoop.

Our proposal is a good foundation for the development of further improvements in this research line, and as part of our future work, we are planning to extend the current results in several contexts:

- include LSH (Locality Sensitive Hashing) partitioning technique [36,40] in SpatialHadoop and compare it with Voronoi-Diagram based partitioning technique.
- handle data skew [40] in these DJQs (mainly for *K*CPQ) in SpatialHadoop.
- implement other complex DJQs in SpatialHadoop, like multi-way distance-join queries [53].
- implement the data partitioning technique based on Voronoi-Diagrams and the improved DJQ MapReduce algorithms in Spark-based spatial analytics systems as GeoSpark [54] or LocationSpark [55].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

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

References

[1] A. Eldawy, M.F. Mokbel, Spatialhadoop: A mapreduce framework for spatial data, in: ICDE Conference, 2015, pp. 1352–1363.
[2] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, in: OSDI Conference, 2004, pp. 137–150.
[3] A. Aji, H. Vo, F. Wang, Effective spatial data partitioning for scalable query processing, 2015, pp. 1–12, CoRR abs/1509.00910.
[4] A. Eldawy, L. Alarabi, M.F. Mokbel, Spatial partitioning techniques in spatial hadoop, PVLDB 8 (12) (2015) 1602–1613.
[5] A. Okabe, B. Boots, K. Sugihara, S.N. Chiu, Spatial Tessellations: Concepts and Applications of Voronoi Diagrams, Wiley, 2000.
[6] F. Aurenhammer, Voronoi diagrams - A survey of a fundamental geometric data structure, ACM Comput. Surv. 23 (3) (1991) 345–405.
[7] W. Lu, Y. Shen, S. Chen, B.C. Ooi, Efficient processing of k nearest neighbor joins using MapReduce, PVLDB 5 (10) (2012) 1016–1027.
[8] A.K. Jain, M.N. Murty, P.J. Flynn, Data clustering: A review, ACM Comput. Surv. 31 (3) (1999) 264–323.
[9] D. Xu, Y. Tian, A comprehensive survey of clustering algorithms, Ann. Data Sci. 2 (2) (2015) 165–193.
[10] F. Ros, S. Guillaume, DIDES: a fast and effective sampling for clustering algorithm, Knowl. Inf. Syst. 50 (2) (2017) 543–568.
[11] E.H. Jacox, H. Samet, Spatial join techniques, ACM Trans. Database Syst. 32 (1) (2007) 7:1–44.
[12] C. Böhm, F. Krebs, The *k*-nearest neighbour join: Turbo charging the KDD process, Knowl. Inf. Syst. 6 (6) (2004) 728–749.
[13] C. Xia, H. Lu, B.C. Ooi, J. Hu, Gorder: An efficient method for KNN join processing, in: VLDB Conference, 2004, pp. 756–767.
[14] Y. Chen, J.M. Patel, Efficient evaluation of all-nearest-neighbor queries, in: ICDE Conference, 2007, pp. 1056–1065.
[15] A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Closest pair queries in spatial databases, in: SIGMOD Conference, 2000, pp. 189–200.
[16] 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.
[17] 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.
[18] 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.
[19] C. Zhang, F. Li, J. Jestes, Efficient parallel kNN joins for large data in MapReduce, in: EDBT Conference, 2012, pp. 38–49.
[20] F. García-García, A. Corral, L. Iribarne, M. Vassilakopoulos, Y. Manolopoulos, Enhancing spatialhadoop with closest pair queries, in: ADBIS Conference, 2016, pp. 212–225.
[21] 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.
[22] F. García-García, A. Corral, L. Iribarne, M. Vassilakopoulos, Voronoi-diagram based partitioning for distance join query processing in spatialhadoop, in: MEDI Conference, 2018, pp. 251–267.
[23] C. Kuhlman, Y. Yan, L. Cao, E.A. Rundensteiner, Pivot-based distributed k-nearest neighbor mining, in: ECML/PKDD Conference, 2017, pp. 843–860.
[24] J. Lu, R.H. Güting, Parallel secondo: Boosting database engines with Hadoop, in: ICPADS Conference, 2012, pp. 738–743.
[25] R.H. Güting, V.T. de Almeida, D. Ansorge, T. Behr, Z. Ding, T. Höse, F. Hoffmann, M. Spiekermann, U. Telle, SECONDO: an extensible DBMS platform for research prototyping and teaching, in: ICDE Conference, 2005, pp. 1115–1116.
[26] 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.
[27] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive - A warehousing solution over a MapReduce framework, PVLDB 2 (2) (2009) 1626–1629.
[28] H. Vo, A. Aji, F. Wang, SATO: a spatial data partitioning framework for scalable query processing, in: SIGSPATIAL Conference, 2014, pp. 545–548.

[29] A. Eldawy, Y. Li, M.F. Mokbel, R. Janardan, Cg_hadoop: computational geometry in mapreduce, in: SIGSPATIAL Conference, 2013, pp. 284–293.

[30] V. Pandey, A. Kipf, T. Neumann, A. Kemper, How good are modern spatial analytics systems? PVLDB 11 (11) (2018) 1661–1673.

[31] A. Akdogan, U. Demiryurek, F.B. Kashani, C. Shahabi, Voronoi-based geospatial query processing with MapReduce, in: CloudCom Conference, 2010, pp. 9–16.

[32] W. Kim, Y. Kim, K. Shim, Parallel computation of k-nearest neighbor joins using mapreduce, in: Big Data Conference, 2016, pp. 696–705.

[33] J. MacQueen, Some methods for classification and analysis of multivariate observations, in: 5-th Berkeley Symposium on Mathematical Statistics and Probability, 1967, pp. 281–297.

[34] H. Kurasawa, D. Fukagawa, A. Takasu, J. Adachi, Optimal pivot selection method based on the partition and the pruning effect for metric space indexes, IEICE Trans. 94-D (3) (2011) 504–514.

[35] H.V. Jagadish, B.C. Ooi, K. Tan, C. Yu, R. Zhang, Idistance: An adaptive b^+-tree based indexing method for nearest neighbor search, ACM Trans. Database Syst. 30 (2) (2005) 364–397.

[36] A. Stupar, S. Michel, R. Schenkel, Rankreduce - processing k-nearest neighbor queries on top of mapreduce, in: LSDS-IR@SIGIR Conference, 2010, pp. 13–18.

[37] 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.

[38] G. Chatzimilioudis, C. Costa, D. Zeinalipour-Yazti, W. Lee, E. Pitoura, Distributed in-memory processing of all k nearest neighbor queries, IEEE Trans. Knowl. Data Eng. 28 (4) (2016) 925–938.

[39] F. Ros, S. Guillaume, DENDIS: a new density-based sampling for clustering algorithm, Expert Syst. Appl. 56 (2016) 349–359.

[40] X. Zhao, J. Zhang, X. Qin, *K*nn-DP: Handling data skewness in *kNN* joins using mapreduce, IEEE Trans. Parallel Distrib. Syst. 29 (3) (2018) 600–613.

[41] D. Arthur, S. Vassilvitskii, K-means++: the advantages of careful seeding, in: SODA Conference, 2007, pp. 1027–1035.

[42] E. Schubert, A. Zimek, ELKI: a large open-source library for data analysis, 2019, pp. 1–134, CoRR abs/1902.03616.

[43] J. Wu, Y. Zhang, J. Wang, C. Lin, Y. Fu, C. Xing, Improving distributed similarity join in metric space with error-bounded sampling, 2019, pp. 1–17, CoRR abs/1905.05981.

[44] J. Blömer, C. Lammersen, M. Schmidt, C. Sohler, Theoretical analysis of the k-means algorithm - A survey, 2016, pp. 1–35, CoRR abs/1602.08254.

[45] G. Schoier, C. Gregorio, Clustering algorithms for spatial big data, in: ICCSA Conference, 2017, pp. 571–583.

[46] S.J. Phillips, Acceleration of k-means and related clustering algorithms, in: ALENEX Conference, 2002, pp. 166–177.

[47] M. Ankerst, M.M. Breunig, H. Kriegel, J. Sander, OPTICS: ordering points to identify the clustering structure, in: SIGMOD Conference, 1999, pp. 49–60.

[48] M. Ester, H. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, in: KDD Conference, 1996, pp. 226–231.

[49] E. Schubert, M. Gertz, Improving the cluster structure extracted from OPTICS plots, in: LWDA Conference, 2018, pp. 318–329.

[50] J. Schneider, M. Vlachos, Fast parameterless density-based clustering via random projections, in: CIKM Conference, 2013, pp. 861–866.

[51] 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.

[52] G.R. Hjaltason, H. Samet, Index-driven similarity search in metric spaces, ACM Trans. Database Syst. 28 (4) (2003) 517–580.

[53] A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Multi-way distance join queries in spatial databases, GeoInformatica 8 (4) (2004) 373–402.

[54] J. Yu, Z. Zhang, M. Sarwat, Spatial data management in apache spark: the geospark perspective and beyond, GeoInformatica 23 (1) (2019) 37–78.

[55] M. Tang, Y. Yu, W.G. Aref, A.R. Mahmood, Q.M. Malluhi, M. Ouzzani, Locationspark: In-memory distributed spatial query processing and optimization, 2019, pp. 1–15, CoRR abs/1907.03736.

Francisco José García García is a PhD Student at the Department of Informatics, University of Almeria (Spain). He received the Computer Science Engineering degree and the Master degree in Advanced Computer Techniques from the University of Almeria, (Spain). Since 2007 he works at the IT Service of the University of Almería. In 2015, he joined Applied Computing Research Group (TIC-211) of the University of Almeria. He has published in referred scientific international journals (GeoInformatica). He has published in referred conferences (ADBIS, MEDI, ICCSA, INTED, etc.) and book chapters. His research interests includes: Big Data, Cloud Computing, query processing, algorithms and spatial and spatio-temporal databases.

Antonio Corral is an Associate Professor at the Department of Informatics, University of Almeria (Spain). He received his PhD (2002) in Computer Science from the University of Almeria (Spain). He has participated actively in several research projects in Spain (INDALOG, vManager, ENIA, etc.) and Greece (CHOROCHRONOS, ARCHIMEDES, etc.). He has published in referred scientific international journals (Data & Knowledge Engineering, GeoInformatica, The Computer Journal, Information Sciences, Computer Standards & Interfaces, etc.), conferences (SIGMOD, SSD, ADBIS, SOFSEM, PADL, DEXA, OTM, MEDI, SAC, etc.) and book chapters. His main research interests include access methods, algorithms, query processing, databases and distributed query processing.

Luis Iribarne is an Associate Professor at the Department of Informatics, University of Almeria (Spain). He received the BS and MS degrees in Computer Science from the University of Granada, and the PhD degree in Computer Science from the University of Almeria, and conducted from the University of Malaga (Spain). From 1991 to 1993, he worked as a Lecturer at the University of Granada, and collaborated as IT Service Analyst at the University School of Almeria. Since 1993, he has served as a Lecturer in the Advanced College of Engineering at the University of Almeria. From 1993 to 1999, he worked in several national and international research projects on distributed simulation and geographic information systems. Since 2006, he has served as the main-coordinator of five R&D projects founded by the Spanish Ministry of Science and Technology, and the Andalusian Ministry ST. In 2007, he has founded the Applied Computing Group (ACG). He has also acted as evaluator for funding agencies in Spain and Argentina. He has published in referred JCR scientific international journals (ISO Abbrev): Comput. J., Comput. Stand. Interfaces, J. Log. Algebr. Methods Program, Softw.-Pract. Exp., Simul. Model. Pract. Theory, IEEE Trans. Geosci. Remote, J. Neurosci. Methods, Inf. Syst. Manage., ‘‘Behav. Brain Res., Comput. Ind. or J. Vis. Lang. Comput. (among others). He has also published in referred scientific international conferences (ICMT, ICSOC, ICSOFT, SOFSEM, ICAART, PAAMS, SEAA, EUROMICRO, among others) and book chapters. His main research interests include simulation & modeling, model-driven engineering, machine learning, and software technologies and engineering.

Michael Vassilakopoulos obtained a five-year Diploma in Computer Eng. and Informatics from the University of Patras (Greece) and a PhD in Computer Science from the Department of Electrical and Computer Eng. of the Aristotle University of Thessaloniki (Greece). He has been with the University of Macedonia, the Aristotle University of Thessaloniki, the Technological Educational Institute of Thessaloniki, the Hellenic Open University, the Open University of Cyprus, the University of Western Macedonia, the University of Central Greece and the University of Thessaly. For three years, he served the Greek Public Administration as an Informatics Engineer. Currently, he is an Associate Professor of Database Systems at the Department of Electrical and Computer Engineering of the University of Thessaly. He has participated in/coordinated several RTD projects related to Databases, GIS, WWW, Information Systems and Employment. His research interests include databases, data structures, algorithms, data mining, employment analysis, information systems, GIS and current trends of data management.