Geoinformatica manuscript No.
(will be inserted by the editor)

Efficient Large-scale Distance-Based Join Queries in SpatialHadoop

Francisco García-García · Antonio Corral · Luis Iribarne · Michael Vassilakopoulos · Yannis Manolopoulos

Received: date / Accepted: date

Abstract Efficient processing of Distance-Based Join Queries (*DBJQs*) in spatial databases is of paramount importance in many application domains. The most representative and known *DBJQs* are the K Closest Pairs Query ($KCPQ$) and the ε Distance Join Query (εDJQ). These types of join queries are characterized by a number of desired pairs (K) or a distance threshold (ε) between the components of the pairs in the final result, over two spatial datasets. Both are expensive operations, since two spatial datasets are combined with additional constraints. Given the increasing volume of spatial data originating from multiple sources and stored in distributed servers, it is not always efficient to perform *DBJQ*s on a centralized server. For this reason, this paper addresses the problem of computing *DBJQs* on big spatial datasets in SpatialHadoop, an extension of Hadoop that supports efficient processing of spatial queries in a cloud-based setting. We propose novel algorithms, based on plane-sweep, to perform efficient parallel *DBJQs* on large-scale spatial datasets in SpatialHadoop. We evaluate the performance of the proposed algorithms in several situations with large real-world as well as synthetic datasets. The experiments demonstrate the efficiency and scalability of our proposed methodologies.

Keywords Distance-Based Join Queries · Spatial Data Processing · SpatialHadoop · MapReduce · Spatial Query Evaluation

1 Introduction

Distance-Based Join Queries (*DBJQs*) in spatial databases [2] have received considerable attention from the database community, due to its importance in numerous applications, such as image processing [3], location-based systems [4], geographical information systems (GIS) [5], continuous monitoring in streaming data settings [6] and road network constrained data [7].

The most representative and known *DBJQs* are the K Closest Pairs Query ($KCPQ$), that discovers the K closest pairs of objects between two spatial datasets, and the ε Distance Join Query (εDJQ), that discovers the pairs of objects with distance smaller than ε between two spatial datasets (detailed definitions appear in Subsections 3.1.1 and 3.1.2, respectively).

Both join queries are expensive operations since two spatial datasets are combined with additional constraints, and they become even more costly operations for large-scale data. Several different approaches have been proposed, aiming to improve the performance of *DBJQs* by proposing efficient algorithms [8–11]. However, all these approaches focus on methods that are executed in a centralized environment.

With the fast increase in the scale of big input datasets, processing large data in parallel and distributed fashions is becoming a common practice. A number of parallel algorithms for *DBJQs*, like the K Closest Pair Query (KCPQ) [1], K Nearest Neighbor Join (KNNJ)

A preliminary partial version of this work appeared in [1].

F. García-García · Antonio Corral · Luis Iribarne
Department on Informatics, University of Almeria, 04120 Almeria, Spain.
E-mail: paco.garcia, acorral, liribarn@ual.es

Michael Vassilakopoulos
Department of Electrical and Computer Engineering, University of Thessaly, 38221 Volos, Greece.
E-mail: mvasilako@uth.gr

Yannis Manolopoulos
Department of Informatics, Aristotle University, 54124 Thessaloniki, Greece.
E-mail: manolopo@csd.auth.gr

[12–15] and similarity join [16] in MapReduce [17] have been designed and implemented recently. However, as real-world spatial datasets continue to grow, novel approaches and paradigms are needed.

Parallel and distributed computing using shared-nothing clusters on extreme-scale data is becoming a dominating trend in the context of data processing and analysis. MapReduce [17] is a framework for processing and managing large-scale datasets in a distributed cluster, which has been used for applications such as generating search indices, document clustering, access log analysis, and various other forms of data analysis [18]. MapReduce was introduced with the goal of supplying a simple yet powerful parallel and distributed computing paradigm, providing good scalability and fault tolerance mechanisms. The success of MapReduce stems from hiding the details of parallelization, fault tolerance, and load balancing in a simple and powerful programming framework [18–21].

However, as indicated in [22], MapReduce has weaknesses related to efficiency when it needs to be applied to spatial data. A main shortcoming is the lack of an indexing mechanism that would allow selective access to specific regions of spatial data, which would in turn yield more efficient query processing algorithms. A recent solution to this problem is an extension of Hadoop, called *SpatialHadoop* [23], which is a framework that inherently supports spatial indexing on top of Hadoop. In SpatialHadoop, spatial data is deliberately partitioned and distributed to nodes, so that data with spatial proximity is placed in the same partition. Moreover, the generated partitions can be indexed, thereby enabling the design of efficient query processing algorithms that access only part of the data and still return the correct result query. As demonstrated in [23], various algorithms have been proposed for spatial queries, such as range, nearest neighbor, spatial joins and skyline queries. Efficient processing of the most representative and studied *DBJQs* over large-scale spatial datasets is a challenging task, and is the main target of this paper.

SpatialHadoop is an efficient MapReduce disk-based distributed spatial query-processing system. Actually, SpatialHadoop is a mature and robust spatial extension of Hadoop (the most well-known shared-nothing parallel and distributed system). SpatialHadoop has been developed for a longer time than related Spark-based spatial extensions, although Spark-based systems are, in general, faster than Hadoop-based systems, especially for iterative problems [24]. SpatialHadoop utilizes pure MapReduce based processing and not DAG (Directed Acyclic Graph) based processing (a generalization of MapReduce), as Spark-based systems. The problem we study, processing *DBJQs*, is well suited to pure MapReduce based processing, since it has limited iterativeness and works on the whole datasets, in batch mode. In this paper, we develop MapReduce algorithms for these queries and study them in SpatialHadoop (a popular system with a wide installation base), as a first step of a series of studies of spatial processing in shared-nothing parallel and distributed systems that will also include Spark-based spatial extensions during further research steps.

Motivated by these observations, we first propose new parallel algorithms, based on plane-sweep technique, for *DBJQs* in SpatialHadoop on big spatial datasets. In addition to the plane-sweep base technique, we present a methodology for improving the performance of the KCPQ algorithms by the computation of an upper bound of the distance of the K-th closest pair. To demonstrate the benefits of our proposed methodologies, we present the results of the execution of an extensive set of experiments that demonstrate the efficiency and scalability of our proposals using big synthetic and real-world points datasets.

This paper substantially extends our previous work [1], which was the foundation of the present research results, with the following novel contributions:

1. We improve the plane-sweep-based KCPQ MapReduce algorithm in SpatialHadoop [1] by using new sampling and approximate techniques, that take advantage of SpatialHadoop partitioning techniques, to compute an upper bound of the distance of the K-th closest pair and make the KCQP MapReduce algorithm much more efficient.
2. We have implemented a new distributed KCPQ algorithm using the local index(es) (R-trees) provided by SpatialHadoop, similarly to the distributed join algorithm [23], and we compare this approach to our plane-sweep-based KCPQ MapReduce algorithm, proving experimentally that our algorithm outperforms the one that uses the local index(es).
3. We propose a new MapReduce algorithm for $\varepsilon DJQs$ in SpatialHadoop, based on the plane-sweep technique, similar to our KCPQ MapReduce algorithm.
4. In experiments of *DBJQ* MapReduce algorithms, we utilize additional partitioning techniques available in SpatialHadoop to check if performance improvements are obtained with respect to the partitioning used in [1].
5. We present results of an extensive experimental study that compares the performance of the proposed MapReduce algorithms and their improvements in terms of efficiency and scalability. For synthetic datasets' experiments, we have used clustered (more realistic) datasets, instead of uniform ones [1]. More-

over, for real datasets' experiments, we have created a new big quasi-real dataset that is combined with the biggest real dataset used in [1].

The current research work is based on a completely new setting with respect to the one of [11], since we have used a scalable and distributed MapReduce framework supporting spatial data, SpatialHadoop, while in [11] processing in a centralized system is followed. Here, we have only used the new plane-sweep KCPQ algorithm published in [11] and executed it in each parallel task. Moreover, new methodologies and improvements have been proposed to speedup the response time of the studied *DBJQs* under cloud computing.

The rest of this article is organized as follows. In Section 2, we review related work about different research prototype systems that have been proposed for large-scale spatial query processing, the MapReduce implementations of the most representative spatial queries and the recent SpatialHadoop framework for spatial query processing. Section 3 defines the *KCPQ* and *εDJQ*, which are the *DBJQs* studied in this work. Moreover, a detailed presentation of SpatialHadoop in the context of spatial query processing is exposed, which is the core framework of this paper. In Section 4, we present the parallel (MapReduce) algorithms for the processing of *DBJQs* (the KCPQ and εDJQ) in SpatialHadoop, using plane-sweep techniques and local spatial indices. Section 5 presents several improvements of the KCPQ MapReduce algorithm with main objective to make the algorithm faster. In Section 6, we present representative results of the extensive experimentation that we have performed, using real-world and synthetic datasets, for comparing the efficiency of the proposed algorithms. Finally, in Section 7, we provide the conclusions arising from our work and discuss potential directions for future work.

2 Related Work

In this section we review related literature to highlight the most representative prototype systems that have been developed for large-scale spatial query processing. Next, we look over specific spatial operations using MapReduce and finally, we review the proposed spatial queries that have been implemented in SpatialHadoop.

2.1 Research prototype systems for large-scale spatial query processing

Researchers, developers and practitioners worldwide have started to take advantage of the MapReduce environment in supporting large-scale spatial data processing. Until now, the most representative contributions in the context of scalable spatial data processing are the following prototypes:

- *Parallel-Secondo* [25] is a parallel spatial DBMS that uses Hadoop as a distributed task scheduler.
- *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.
- *SpatialHadoop* [23] is a full-fledged MapReduce framework with native support for spatial data. It tightly integrates well-known spatial operations (including indexing and joins) into Hadoop.
- *SpatialSpark* [28] is a lightweight implementation of several spatial operations on top of the *Apache Spark*[1] in-memory big data system. It targets at in-memory processing for higher performance.
- *GeoSpark* [29] is an in-memory cluster computing system for processing large-scale spatial data, and it extends the core of *Apache Spark* to support spatial data types, indices and operations.
- *Simba* (Spatial In-Memory Big data Analytics) [30] offers scalable and efficient in-memory spatial query processing and analytics for spatial big data. Simba extends the Spark SQL engine to support rich spatial queries and analytics through both SQL and the DataFrame API.
- *LocationSpark* [31] has been recently presented as a spatial data processing system built on top of *Apache Spark*. It offers a rich set of spatial query operators, e.g., range search, KNN, spatio-textual operation, spatial join and KNN join. Moreover, it offers an efficient spatial Bloom filter into LocationSpark's indices to avoid unnecessary network communication overhead when processing overlapped spatial data.

All the previous prototypes have been designed for processing and analysis of massive spatial vectorial data (e.g. points, line-segments, etc.), but there are other prototypes for managing spatial raster data derived from imaging and spatial applications (e.g. climate data [32], satellite data, etc.). The most remarkable scientific prototype systems for handling raster data are: *SciHadoop* [33], *Shahed* [34] and *SciSpark* [35]. *SciHadoop* [33] supports array-based query processing of climate data in Hadoop and defined a query language to express common data analysis tasks. *Shahed* [34] is a MapReduce-

[1] `http://spark.apache.org/`

based system for querying, visualizing, and mining large scale satellite data. It considers both the spatial and temporal aspects of remote sensing data to build a multi-resolution Quadtree-based spatio-temporal index to conduct selection and aggregate queries in real-time using MapReduce. *SciSpark* [35] extends Apache Spark to achieve parallel ingesting and partitioning of multi-dimensional scientific data.

It is important to highlight that the previous prototype systems differ significantly in terms of distributed computing platforms, data access models, programming languages and the underlying computational geometry libraries. Moreover, all these prototypes support query processing for the most representative spatial operators and use the MapReduce software framework to carry them out. In the next subsection we review the most remarkable contributions of the literature for spatial query processing using MapReduce.

2.2 Spatial query processing using MapReduce

Actually, there are a lot of works on specific spatial queries using MapReduce. This programming framework adopts a flexible computation model with a simple interface consisting of *map* and *reduce* functions whose implementations can be customized by application developers. Therefore, the main idea is to develop *map* and *reduce* functions for the required spatial operation, which will be executed on-top of an existing Hadoop cluster. Examples of such research works on specific spatial queries using MapReduce include:

- *Region query* [36,37], where, in general, the input file is scanned, and each record is compared against the query region.
- *K Nearest Neighbors (KNN) query* [36,38,39]. In [36], a brute force approach calculates the distance to each point and selects the nearest K points. Another approach partitions points using a Voronoi diagram and finds the answer in partitions close to the query point [38]. Lastly, in [39] a MapReduce-based approach for KNN classification is proposed.
- *Skyline query* [40,41]. In [40], new algorithms for processing skyline and reverse skyline queries in MapReduce are proposed. In [41], an advanced two-phase MapReduce solution that efficiently addresses skyline queries on large datasets is presented.
- *Reverse Nearest Neighbor (RNN) query* [38,42]. In [38], the input data is partitioned by a Voronoi diagram to exploit its properties to answer *RNN* queries. In [42], the problem of *RNN* query is investigated on the inverted grid index over large scale spatial datasets in a distributed environment using MapReduce.
- *Spatial Join query* [36,43]. In [36,43] the *partition-based spatial-merge* join [44] is ported to MapReduce, giving rise to a new algorithm for Spatial Join with MapReduce (*SJMR*).
- *K Nearest Neighbor (KNN) Join query* [12–15]. In [12], novel (exact and approximate) algorithms in MapReduce to perform efficient parallel KNN join on large data are proposed, and they use the R-tree and a Z-value-based partition join to implement them. In [13], the authors use Voronoi diagram-based partitioning method that exploits pruning rules for distance filtering, and hence reduces both the shuffling and computational costs. In [14], methods for accelerating the KNN join processing have been also proposed. Recently, in [15], a novel method for classifying multidimensional data using a KNN join algorithm in the MapReduce framework is proposed.
- *Top-K closest pair* problem (where just one dataset is involved) [45], this problem is studied with Euclidean distance using MapReduce.
- *Similarity Join query* for high-dimensional data using MapReduce has been also studied. One of the most representative work is [16], where a partition-based similarity join for MapReduce is proposed (called *MRSimJoin*), based on the *QuickJoin* algorithm [46].
- *Multi-Way Spatial Join query.* In [47], the problem of processing multi-way spatial joins on MapReduce platform is investigated. The authors study two common spatial predicates, *overlap* and *range*, and discuss how the join queries involving both predicates are processed. The most important contribution of this paper is a *Controlled-Replicate* framework, and how it is carefully engineered to minimize the communication among cluster nodes.
- *Trajectory query.* In [48], a new distributed R-tree index, called the Distributed Trajectory R-tree (DTR-tree), in Apache Spark has been developed, and it is used to solve the problem of a distributed trajectory query search with activities.

As apparent from the discussion, multiple efforts addressing various aspects of spatial query processing using MapReduce have appeared during last years. However, our work is complementary to these, in the sense that we have implemented new approaches and improvements to solve *DBJQs* (i.e. KCPQ and εDJQ) for spatial big data.

2.3 Spatial queries in SpatialHadoop

SpatialHadoop is equipped with a several spatial operations, including range query, KNN and spatial join [23], and other fundamental computational geometry algorithms as polygon union, skyline, convex hull, farthest pair, and closest pair [49]. In [50] a scalable and efficient framework for skyline query processing that operates on top of SpatialHadoop is presented, and it can be parameterized by individual techniques related to filtering of candidate points as well as merging of local skyline sets. Then, the authors introduce two novel algorithms that follow the pattern of the framework and boost the performance of skyline query processing. Recently, a first parallel KCPQ algorithm in MapReduce on big spatial datasets, adopting the plane-sweep technique, was proposed in [1]. The MapReduce algorithm was also improved with the computation of an upper bound of the distance value of the K-th closest pair from sampled data as a global preprocessing phase.

The efficient processing of *DBJQs* over large-scale spatial datasets using SpatialHadoop is a challenging task. The improvements of the KCPQ MapReduce algorithm [1] and a new MapReduce algorithm for εDJQ are the main targets of this work and, as we will demonstrate, our approaches accelerate the response time by using plane-sweep, specific spatial partitioning, and determining the needed number of computing nodes depending on the parallel tasks.

3 Preliminaries and Background

We now introduce the details of the semantics of the studied queries, along with the corresponding notation and processing paradigms. We start with the definitions and characteristics of both *DBJQs* and then, we review SpatialHadoop, the scalable and distributed framework for managing spatial data and the steps for spatial query processing.

3.1 Distance-Based Join Queries

A *DBJQ* is characterized as a join between two datasets based on a distance function, reporting a set of pairs according to a given constraint (e.g. a number of desired pairs, a distance threshold, etc.) over two datasets. The most representative and known *DBJQs* are the K Closest Pairs Query ($KCPQ$) and the ε Distance Join Query (εDJQ).

3.1.1 K Closest Pairs Query

The KCPQ discovers the K pairs of data formed from the elements of two datasets having the K smallest respective distances between them (i.e. it reports only the top K pairs). It is one of the most important spatial operations, where two spatial datasets and a distance function are involved. It is considered a distance-based join query because it involves two different spatial datasets and uses distance functions to measure the degree of nearness between pairs of spatial objects. The formal definition of the KCPQ for point datasets (the extension of this definition to other, more complex spatial objects – e.g. line-segments, objects with extents, etc. – is straightforward) is the following:

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

Note that if multiple pairs of points have the same K-th distance value, more than one set of K different pairs of points are suitable as a result of the query. It is straightforward to extend the presented algorithms so as to discover all such sets of pairs.

This spatial query has been actively studied in centralized environments, regardless whether both spatial datasets are indexed or not [8,9,11,51–55]. In this context, recently, when the two datasets are not indexed and stored in main-memory, a new plane-sweep algorithm for KCPQ, called *Reverse Run*, was proposed in [9]. Two improvements on the *Classic plane-sweep* algorithm for this spatial query were presented as well. Experimentally, the *Reverse Run plane-sweep* algorithm proved to be faster since it minimized the number of Euclidean distance computations. However, datasets that reside in a parallel and distributed framework have not attracted similar attention and this is the main objective of this work.

3.1.2 εDistance Join Query

The ε Distance Join Query (εDJQ) reports all the possible pairs of spatial objects from two different spatial objects datasets, having a distance smaller than a distance threshold ε [11]. Note that, if $\varepsilon = 0$, then we have

the condition of *spatial overlap join*, which retrieves all different intersecting spatial object pairs from two distinct spatial datasets [2]. This query is also related to the similarity join [16], where the problem of deciding if two objects are similar is reduced to the problem of determining if two high-dimensional points are within a certain distance threshold ε of each other. The formal definition of εDJQ for point datasets is the following:

Definition 2 *(ε **Distance Join Query, εDJQ**)* Let $\mathbb{P} = \{p_0, p_1, \cdots, p_{n-1}\}$ and $\mathbb{Q} = \{q_0, q_1, \cdots, q_{m-1}\}$ be two set of points in E^d, and a distance threshold $\varepsilon \in \mathbb{R}_{\geq 0}$. Then, the result of the ε Distance Join Query (ε**DJQ**) is the set $\varepsilon DJQ(\mathbb{P}, \mathbb{Q}, \varepsilon) \subseteq \mathbb{P} \times \mathbb{Q}$ containing all the possible different pairs of points from $\mathbb{P} \times \mathbb{Q}$ that have a distance of each other smaller than, or equal to ε:

$$\varepsilon DJQ(\mathbb{P}, \mathbb{Q}, \varepsilon) = \{(p_i, q_j) \in P \times Q \ : \ dist(p_i, q_j) \ \leq \ \varepsilon\}$$

The εDJQ can be considered as an extension of the KCPQ, where the distance threshold of the pairs is known beforehand and the processing strategy (e.g. plane-sweep technique) is the same as in the KCPQ for generating the candidate pairs of the final result. On the other hand, in the case of the KCPQ the distances of the K closest pairs are not known beforehand and they are updated during the processing of the algorithm.

3.2 SpatialHadoop

SpatialHadoop [23] is a full-fledged MapReduce framework with native support for spatial data. Note that, MapReduce [17] is a scalable, flexible and fault-tolerant programming framework for distributed large-scale data analysis. A task to be performed using the MapReduce framework has to be specified as 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, and distributes it to worker nodes; and the *reduce* phase that processes these results as specified by a *reduce function*. An important aspect of MapReduce is that both the input and the output of the *map* step are represented as *key-value pairs*, and that pairs with same key will be processed as one group by the *reducer*: $map : (k_1, v_1) \rightarrow list(k_2, v_2)$ and $reduce : k_2, list(v_2) \rightarrow list(v_3)$. Additionally, a *combiner function* can be used to run on the output of *map* phase and perform some filtering or aggregation to reduce the number of keys passed to the *reducer*.

SpatialHadoop [23] is a comprehensive extension to Hadoop that injects spatial data awareness in each Hadoop layer, namely, the language, storage, MapReduce, and operations layers. In the *Language* layer, SpatialHadoop adds a simple and expressive high level language for spatial data types and operations. In the *Storage* layer, SpatialHadoop adapts traditional spatial index structures as Grid, R-tree, R^+-tree, Quadtree, etc. to form a two-level spatial index [56]. SpatialHadoop enriches the *MapReduce* layer by two new components, *SpatialFileSplitter* and *SpatialRecordReader* for efficient and scalable spatial data processing. At the *Operations* layer, SpatialHadoop is also equipped with a several spatial operations, including range query, kNN query and spatial join. Other computational geometry algorithms (e.g. polygon union, skyline, convex hull, farthest pair and closest pair) are also implemented following a similar approach [49]. Finally, we must emphasize that our contribution for *DBJQs* is located in the *Operations* and *MapReduce* layers.

In general, a spatial query processing in SpatialHadoop consists of four steps [23,1]:

1. *Preprocessing*, where data are partitioned according to a specific spatial partitioning technique (e.g. Grid, STR, Quadtree, Hilbert, etc.) [56], generating a set of partitions, called *cells*. Each HDFS block corresponds to a cell, and the HDFS blocks in each file are globally indexed, generating a *spatially indexed file*. In the partitioning phase, *spatial data locality* is obeyed, since spatially nearby objects are assigned to the same cell [23].
2. *Pruning*, when the query is issued, the master node examines all partitions and prunes by a *filter* function those ones that are guaranteed not to include any possible result of the spatial query. Note that, 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 cells/partitions not contributing to the answer. The two steps (Preprocessing and Pruning) can be seen in [1] and in the Figure 2.
3. *Local Spatial Query Processing*, where local spatial query processing is performed on each non-pruned partition in parallel on different machines (*map* tasks). Note that SpatialHadoop also enriches traditional Hadoop systems in this step by the *SpatialRecordReader*, which reads a split originating from the spatially indexed input file(s) and exploits local index(es) to efficiently processes the spatial queries. In this step, if we do not want to use the *SpatialRecordReader* component (for example, to use the *plane-sweep technique*) and exploit the advantages of the local index(es), we just use a *RecordReader* that extracts records as key-value pairs which are

passed to the *map* function. We can see this option in Figure 1, between *SSR* and *map* function.

4. *Global Processing*, where the results are collected from all machines in the previous step and the final result of the concerned spatial query is computed (*reduce* tasks). A *combine* function may be applied in order to decrease the volume of data that is sent from the *map* task. The *reduce* function is omitted when the results from the *map* phase are final. See Figure 1 to observe these last two steps, MapReduce query processing in SpatialHadoop.

Next we are going to follow this query processing scheme to include *DBJQs* into SpatialHadoop.

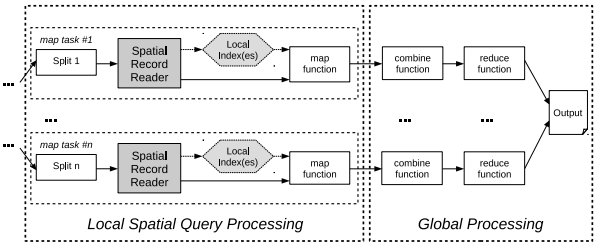

Fig. 1 MapReduce query processing in SpatialHadoop.

4 DBJQs Algorithms in SpatialHadoop

In this section, we will state our algorithmic approaches for DBJQs algorithms on top of SpatialHadoop. First, we present the KCPQ MapReduce algorithm using the plane-sweep technique for each *map* task and next, we will extend such MapReduce algorithm to design the distributed algorithm for εDJQ in SpatialHadoop.

4.1 KCPQ Algorithms in SpatialHadoop

In this subsection, we describe our approach to KCPQ algorithms on top of SpatialHadoop. This can be described as a generic top-K MapReduce job that takes one of the specific KCPQ algorithms as a parameter. In general, our solution scheme is similar to how the distributed join algorithm [23] is performed in SpatialHadoop, where combinations of cells from each dataset are the input for each *map* task, when the spatial query is performed. Then the *reducer* emits the top-K results from all *mapper* outputs. In particular, our approach makes use of plane-sweep KCPQ algorithms for main-memory resident datasets [9].

The *plane-sweep technique* [57] has been successfully used in spatial databases to report the result of KCPQ for two indexed datasets [8,51,53,58], whereas it has been improved recently for non-indexed sets of points [9,11]. In this paper we will use the algorithms presented in [9,11] and their improvements to adapt them to MapReduce versions in SpatialHadoop. When the partitions are locally indexed by R-trees, we will adapt algorithms proposed in [8] to KCPQ to the distributed join algorithm [23] to compare them with our KCPQ MapReduce algorithms based on plane-sweep technique.

In [9,11], the *Classic Plane-Sweep* for KCPQ [8, 53] was reviewed and two new improvements were also presented to reduce the search space, when the point datasets reside in main memory. In general, if we assume that the two point sets are $\mathbb{P}$ and $\mathbb{Q}$, the *Classic PS* algorithm consists of the two following steps: (1) sorting the entries of the two point sets, based on the coordinates of one of the axes (e.g. X) in increasing order, and (2) combining one point (*reference*) of one set with all the points of the other set satisfying $point.x - reference.x \leq \delta$ ($point.x - reference.x$ is called dx distance function on the X-axis), where δ is the distance of the K-th closest pair found so far, and choosing those pairs with point distance (*dist*) smaller than δ. The algorithm chooses the *reference* point from $\mathbb{P}$ or $\mathbb{Q}$, following the order on the sweeping axis. We notice that the search space is only restricted to the closest points with respect to the *reference* point, according to the current distance threshold (δ) on the sweeping axis, and this is called *sliding strip*. No duplicated pairs are obtained, since the points are always checked over sorted sets.

In [9,11], a new plane-sweep algorithm for KCPQ was proposed for minimizing the number of distance computations. It is called *Reverse Run Plane-Sweep* algorithm and is based on the concept of *run*, which is a continuous sequence of points of the same set that doesn't contain any point from the other set. Each point used as a *reference* forms a *run* with other subsequent points of the same set. During the algorithm processing, for each set, we keep a *left limit*, which is updated (moved to the right) every time that the algorithm concludes that it is only necessary to compare with points of this set that reside on the right of this limit. Each point of the *active run* (*reference* point) is compared with each point of the other set (*comparison* point) that is on the left of the first point of the *active run*, until the *left limit* of the other set is reached. And the *reference* points (and their *runs*) are processed in ascending X-order (the sets are X-sorted before the application of the algorithm). Each point of the *active run* is compared with the points of the other set (*comparison* points) in the opposite or reverse order (descending X-order).

Moreover, for each point of the *active run* being compared with a current *comparison* point, there are two cases: (1) if the distance between this pair of points in the sweeping axis *(dx)* is larger than or equal to δ, then there is no need to calculate the distance *(dist)* of the pair; thus, we avoid this distance computation, and (2) if the distance *(dist)* between this pair of points *(reference, comparison)* is smaller than the δ distance value, then the pair will be considered as a candidate for the result. For more details of the algorithm see [9,11].

The two improvements of the plane-sweep technique for KCPQs presented in [9,11] for reducing the search space, called *Sliding Window* and *Sliding Semi-Circle*, can be applied both in *Classic* and *Reverse Run* algorithms. The general idea of *Sliding Window* consists in restricting the search space to the closest points inside the window with width δ and a height $2 * \delta$ (i.e. $[0, \delta]$ in the X-axis and $[-\delta, \delta]$ in the Y-axis, from the *reference* point). The core idea of the *Sliding Semi-Circle* improvement consists in reducing the search space even more; by select only those points inside the semi-circle (or half-circle) centered in the *reference* point with radius δ.

Processing the KCPQ in MapReduce [1] adopts the top-K MapReduce methodology. The basic idea is to partition $\mathbb{P}$ and $\mathbb{Q}$ by some method (e.g., Grid) into n and m cells of points and generate $n \times m$ possible pairs of cells to possibly combine. Then, every suitable pair of cells (one from $\mathbb{P}$ and one from $\mathbb{Q}$) is sent as the input for the *map* phase. Each *mapper* reads the points from the pair of cells and performs a plane-sweep (*Classic* or *Reverse Run*) KCPQ algorithm (*PSKCPQ*) between the points inside that pair of cells. That is, it finds the K closest pairs between points in the local cell from $\mathbb{P}$ and in the local cell from $\mathbb{Q}$ using a plane-sweep KCPQ algorithm (*PSKCPQ*). To this end, each *mapper* sorts the points inside the pair of cells from $\mathbb{P}$ and $\mathbb{Q}$ in one axis (e.g., X axis in ascending order) and then applies a plane-sweep KCPQ algorithm. The results from all *mappers* are sent to a single *reducer* that will in turn find the global top-K results of all the *mappers*. Finally, the results are written into HDFS files, storing only the point coordinates and the distance between them.

In Algorithm 1 we can see our proposed solution for KCPQ in SpatialHadoop which consists of a single MapReduce job. The *map* function aims to find the K closest pairs between the local pair of cells from $\mathbb{P}$ and $\mathbb{Q}$ with a particular plane-sweep (*Classic* or *Reverse Run*) KCPQ algorithm (*PSKCPQ*). *KMaxHeap* is a max binary heap [59] used to keep record of local selected top-K closest pairs that will be processed by the *reduce* function. The output of the *map* function is in the form of a set of *DistanceAndPair* elements (called $\mathbb{D}$ in Algorithm 1), i.e. pairs of points from $\mathbb{P}$ and $\mathbb{Q}$ and their distances. As in every other top-K pattern, the *reduce* function can be used in the *combiner* to minimize the shuffle phase. The *reduce* function aims to examine the candidate *DistanceAndPair* elements and return the final set of the K closest pairs. It takes as input a set of *DistanceAndPair* elements from every mapper and the number of pairs. It also employs a max binary heap, called *CandidateKMaxHeap*, to calculate the final result. Each *DistanceAndPair* element is inserted into the heap if its distance value is less than the distance value of the heap root. Otherwise, that pair of points is discarded. Finally, candidate pairs which have been stored in the heap are returned as the final result and stored in the output file.

Algorithm 1 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:     KMaxHeap ← PSKCPQ(ℙ, ℚ, K)
5:     if KMaxHeap is not empty then
6:         for all DistanceAndPair ∈ KMaxHeap do
7:             output(null, DistanceAndPair)
8:         end for
9:     end if
10: end function

11: function COMBINE, REDUCE(null, 𝔻: set of DistanceAndPair, K: number of pairs)
12:     Initialize(CandidateKMaxHeap, K)
13:     for all DistanceAndPair ∈ 𝔻 do
14:         Insert(CandidateKMaxHeap, DistanceAndPair)
15:     end for
16:     for all candidate ∈ CandidateKMaxHeap do
17:         output(null, candidate)
18:     end for
19: end function
```

To compare the plane-sweep-based KCPQ MapReduce algorithms, an implementation using the local indices provided by SpatialHadoop similarly to distributed join algorithm [23] has been made. When a spatial dataset is partitioned using a partitioning technique (e.g. Grid, Str, etc.), SpatialHadoop generates only a global index of the data. However, if a file is partitioned using Str or Str+ there is the option to generate a local index in the form of one R-tree for each of the partitions/cells that are part of the previous global index. The new distributed KCPQ algorithm follows the same scheme presented in Algorithm 1, consisting of a single MapReduce job whose only difference is the processing performed in the *map* function, keeping the *reduce* function unmodified. In this case, the *map* function applies a plane-sweep algorithm over the nodes of the R-trees as described in [8]. This algorithm

consists in traversing both R-trees in a best-first order, keeping a global min binary heap [59] prioritized by the minimum distance between the considered pairs of MBRs. When dealing with leaf nodes, a plane-sweep algorithm is applied to the elements that are contained on them, whereas the δ value is updated appropriately. In the case of internal nodes, plane-sweep is also applied for processing two internal nodes; the MBR pairs with minimum distance greater than δ are pruned. We have chosen the best-first traversal order for the combination of the two R-trees, since it is the fastest algorithm for processing of KCPQs according to [8].

4.2 εDJQ in SpatialHadoop

Processing the εDJQ in MapReduce adopts the *map* phase of the join MapReduce methodology. The basic idea is to have $\mathbb{P}$ and $\mathbb{Q}$ partitioned by some method (e.g., Grid) into two set of cells, $\mathbb{C}_\mathbb{P}$ and $\mathbb{C}_\mathbb{Q}$, with n and m cells of points, respectively. Then, every possible pair of cells (one from $\mathbb{C}_\mathbb{P}$ and one from $\mathbb{C}_\mathbb{Q}$) is sent as input for the *filter* phase. The CELLSFILTER function takes as input, combinations of cells in which the input set of points are partitioned and a distance threshold ε, and it prunes pairs of cells which have minimum distances larger than ε. Using SpatialHadoop built-in function *MinDistance* we can calculate the minimum distance between two cells, i.e. this function computes the minimum distance between the two MBRs, Minimum Bounding Rectangles, of the two cells (each of the two MBRs covers the points of a different cell). That is, if we find a pair of cells with points which cannot have a distance value smaller than ε, we can prune this pair.

On the *map* phase each *mapper* reads the points of a pair of cells and performs a plane-sweep (*Classic* or *Reverse Run*) εDJQ algorithm ($PS\varepsilon DJQ$) between the points inside that pair of cells from $\mathbb{C}_\mathbb{P}$ and $\mathbb{C}_\mathbb{Q}$. That is, it computes the εDJQ between points in the local cell of $\mathbb{C}_\mathbb{P}$ and in the local cell of $\mathbb{C}_\mathbb{Q}$ using a plane-sweep εDJQ algorithm (variation of the plane-sweep-based KCPQ algorithm [11]). To this end, each *mapper* sorts the points inside the pair of cells from $\mathbb{C}_\mathbb{P}$ and $\mathbb{C}_\mathbb{Q}$ in one axis (e.g., X axis in ascending order) and then applies a particular plane-sweep (*Classic* or *Reverse Run*) εDJQ algorithm ($PS\varepsilon DJQ$). The results from all *mappers* are just combined in the *reduce* phase and written into HDFS files, storing only the pairs of points with distance up to ε, as we can see in Algorithm 2.

In addition, we can use the local indices provided by SpatialHadoop to obtain improvements in the performance of the previous εDJQ MapReduce algorithm. This new algorithm follows the same scheme of a single MapReduce job whose only difference is the processing that is realized in the *map* function, maintaining the function CELLSFILTER without any modification. In this case, we have locally indexed the data in each partition by R-tree structures that we can use to process the query. The algorithm consists of performing a iterative depth-first search over the R-trees (this is used for the implementation of the distributed join algorithm [23]). That is, for each pair of internal nodes, one from each index, the minimum distance between their MBRs is calculated; if it is larger than ε, then this pair is pruned. Otherwise, the children of the nodes will be checked in the next step following a depth-first order. When the leaf nodes are reached, the same plane-sweep algorithm as the one without local indices is applied. We have chosen the iterative depth-first traversal order for the combination of two R-trees and not the best-first one, because, if ε is large enough, the global min binary heap can grow very quickly and exceed the available main memory and, thus management of secondary memory is needed and the response time of the algorithm execution will be notably extended.

Algorithm 2 εDJQ MapReduce Algorithm

```
1: function MAP(ℙ: set of points, ℚ: set of points, ε: threshold distance)
2:     SortX(ℙ)
3:     SortX(ℚ)
4:     Results ← PSεDJQ(ℙ, ℚ, ε)
5:     for all DistanceAndPair ∈ Results do
6:         output(null, DistanceAndPair)
7:     end for
8: end function

9: function CELLSFILTER(ℂ_ℙ: set of cells, ℂ_ℚ: set of cells, ε: threshold distance)
10:    for all c ∈ ℂ_ℙ do
11:        for all d ∈ ℂ_ℚ do
12:            minDistance ← MinDistance(c, d)
13:            if minDistance ≤ ε then
14:                output(c, d)
15:            end if
16:        end for
17:    end for
18: end function
```

5 Improvements for KCPQ in SpatialHadoop

It can be clearly seen that the performance of the proposed solution of the KCPQ MapReduce algorithm (Algorithm 1) will depend on the number of cells in which the two sets of points are partitioned. That is, if the set of points $\mathbb{P}$ is partitioned into n cells (the set $\mathbb{C}_\mathbb{P}$) and the set of points $\mathbb{Q}$ is partitioned in m cells (the set $\mathbb{C}_\mathbb{Q}$), then we obtain $n \times m$ combinations of cells or *map* tasks. On the other hand, we know that plane-

sweep-based KCPQ algorithms use a pruning distance value, which is the distance value of the K-th closest pair found so far, to discard those combinations of pairs of points that are not necessary to consider as a candidate of the final result. As suggested in [1], we need to find in advance an upper bound distance of the distance value of the K-th closest pair of the joined datasets, called β. The computation of β can be carried out (a) by sampling globally both big datasets and executing a *PSKCPQ* algorithm over the two samples, or (b) by appropriately selecting a specific pair of cells to which the two big datasets are partitioned and either (b1) by sampling locally the cells of this pair and executing a *PSKCPQ* algorithm over the two samples, or (b2) by applying an approximate variation of a plane-sweep KCPQ algorithm over the entries of the cells of this pair. In the following subsections we will see all these methods.

5.1 Computing β by Global Sampling

The first method of computing β can be seen in Algorithm 3 (computing β by global sampling algorithm), where we take a small sample from both sets of points ($\mathbb{P}$ and $\mathbb{Q}$) and calculate the K closest pairs using a plane-sweep-based KCPQ algorithm (*PSKCPQ* [11]) that is applied locally. Then, we set β equal to the distance of the K-th closest pair of the result and use this distance value as input for *mappers*. This β value guarantees that there will be at least K closest pairs if we prune pairs of points with larger distances in every *mapper*. Figure 2 shows the general schema of computing β (upper bound of the distance of the K-th closest pair) using global sampling, which is used to filter only pairs of cells/partitions with minimum distance of their MBRs smaller than or equal to β.

Furthermore, we can use this β value in combination with the features of the global indexing that SpatialHadoop provides to further enhance the pruning phase. Before the *map* phase begins, we exploit the global indices to prune cells that cannot contribute to the final result. CELLSFILTER takes as input each combination of pairs of cells in which the input set of points are partitioned. Using SpatialHadoop built-in function *MinDistance*, we can calculate the minimum distance between two MBRs of the cells. That is, if we find a pair of cells with points which cannot have a distance value smaller than or equal to β, we can prune this combination of pairs of cells. Using different percentages of samples of the input datasets in Algorithm 3, we have obtained results with a significant reduction of execution time as explained later in the section of experimentation. Note that to obtain a sample from each dataset,

Algorithm 3 Computing β by global sampling Alg.

```
1: function CALCULATEβ(ℙ: set of points, ℚ: set of points, ρ: global sampling ratio, K: number of pairs)
2:     SampledP ← SAMPLEMR(ℙ, ρ)
3:     SampledQ ← SAMPLEMR(ℚ, ρ)
4:     SORTX(SampledP)
5:     SORTX(SampledQ)
6:     KMaxHeap ← PSKCPQ(SampledP, SampledQ, K)
7:     if K MaxHeap is full then
8:         βDistanceAndPair ← POP(KMaxHeap)
9:         β ← βDistanceAndPair.Distance
10:        OUTPUT(β)
11:    else
12:        OUTPUT(∞)
13:    end if
14: end function

15: function CELLSFILTER(ℂℙ: set of cells, ℂℚ: set of cells, β: upper bound distance)
16:    for all c ∈ ℂℙ do
17:        for all d ∈ ℂℚ do
18:            minDistance ← MINDISTANCE(c, d)
19:            if minDistance ≤ β then
20:                OUTPUT(c, d)
21:            end if
22:        end for
23:    end for
24: end function
```

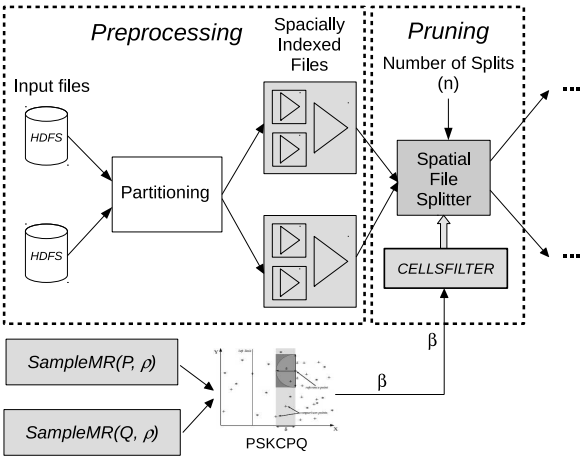

Fig. 2 Schema for computing β by global sampling.

we use a SpatialHadoop built-in MapReduce function, called *SampleMR*, which extracts a percentage of samples (sampling ratio ρ in %, $0.0 < \rho \leq 100.0$) following a sampling Without Replacement (WoR) pattern [60].

5.2 Computing β by Local Processing

Analyzing the above method for the β calculation, it is clearly observed that the greatest time overhead occurs in the execution of the two calls to the *SampleMR* function, since they are complete MapReduce jobs. Therefore, to try to improve the previous algo-

rithm and avoid to call the *SampleMR* function, we are looking to take advantage of the information provided by the indices and other features of SpatialHadoop, and, thus, to make faster the β computation.

Global indices in SpatialHadoop provide the MBR of index cells, as well as the number of elements contained in them, so that we can get an idea of the distribution of data into each cell. To simplify the sampling process we will find a suitable pair of cells, that by their characteristics, may contain K closest pairs with a β value as small as possible. Then we can sample locally those cells without having to execute a MapReduce job (as *SampleMR*).

Since we are looking for the K closest pairs, the search for the most suitable pair of cells can be reduced to look for the pair of cells with an MBR containing them that has the highest density of points and whose intersection is the largest. The larger the area of intersection of two cells, the larger the probability that points in one set are near points in the other set. If the density is also higher, the distances between points are smaller and therefore we will be able to obtain better candidate pairs of cells. Let $c \in \mathbb{C}_\mathbb{P}$ and $d \in \mathbb{C}_\mathbb{Q}$ be a pair of cells from two global indices generated in SpatialHadoop from $\mathbb{P}$ and $\mathbb{Q}$, $|c|$ is the number of elements inside cell c (cardinality of c), $Area(c \cup d)$ is the area of the MBR that covers both MBRs of cells c and d (union MBR), and $Area(c \cap d)$ is the area of the intersection MBR of both MBRs of cells c and d. Then, by *PDDAI(c,d)* we denote a metric that expresses the *suitability*, based on data density and area intersection, of these two cells to allocate K closest pairs with as small distances as possible (*PDDAI* is the acronym of Pair Data Density Area Intersection).

$$PDDAI(c,d) = \frac{|c| + |d|}{Area(c \cup d)} \times (1 + Area(c \cap d))$$

We will select the pair of cells with the maximum value of this metric, so that we will have the pair with the larger combination of density of points and area of intersection. In the case of pairs of cells that do not intersect only the data density is taken into count.

5.2.1 Computing β by Local Sampling

The new method of computing β can be seen in Algorithm 4 (computing β by *local sampling* algorithm), which follows a scheme similar to that of global sampling. There is a new step, the SELECTCELLS function, in which the pair of cells (c and d) having the highest value for the *PDDAI(c,d)* metric is obtained. To do this, the cells of the two global indices are joined by calculating the *PDDAI* metric for each combination. Then the candidate pair of cells is sampled by recalculating the sampling ratio ρ, since we are dealing with a subset of elements and we want to obtain the same number of elements as for the case of global sampling. Once the samples are obtained locally and verified that they reside in memory, a local plane-sweep-based KCPQ algorithm (*PSKCPQ*) is applied to obtain β. Finally, this value is used in the CELLSFILTER function just as in Algorithm 3.

Algorithm 4 Computing β by local sampling Alg.

```
1: function SELECTCELLS(ℂ_ℙ: set of cells, ℂ_ℚ: set of cells)
2:     maxDensity ← 0
3:     bestPair ← ∅
4:     for all c ∈ ℂ_ℙ do
5:         for all d ∈ ℂ_ℚ do
6:             pairDensity ← PDDAI(c, d)
7:             if pairDensity > maxDensity then
8:                 maxDensity ← pairDensity
9:                 bestPair ← (c, d)
10:            end if
11:        end for
12:    end for
13:    OUTPUT(bestPair)
14: end function

15: function CALCULATEβ(ℙ: set of points, ℚ: set of points, ρ: global sampling ratio, K: number of pairs)
16:    localPρ ← CALCULATELOCALRATIO(|ℙ|, ρ)
17:    SampledP ← SAMPLING(ℙ, localPρ)
18:    localQρ ← CALCULATELOCALRATIO(|ℚ|, ρ)
19:    SampledQ ← SAMPLING(ℚ, localQρ)
20:    SORTX(SampledP)
21:    SORTX(SampledQ)
22:    KMaxHeap ← PSKCPQ(SampledP, SampledQ, K)
23:    if KMaxHeap is full then
24:        βDistanceAndPair ← POP(KMaxHeap)
25:        β ← βDistanceAndPair.Distance
26:        OUTPUT(β)
27:    else
28:        OUTPUT(∞)
29:    end if
30: end function

31: function CELLSFILTER(ℂ_ℙ: set of cells, ℂ_ℚ: set of cells, β: upper bound distance)
32:    for all c ∈ ℂ_ℙ do
33:        for all d ∈ ℂ_ℚ do
34:            minDistance ← MINDISTANCE(c, d)
35:            if minDistance ≤ β then
36:                OUTPUT(c, d)
37:            end if
38:        end for
39:    end for
40: end function
```

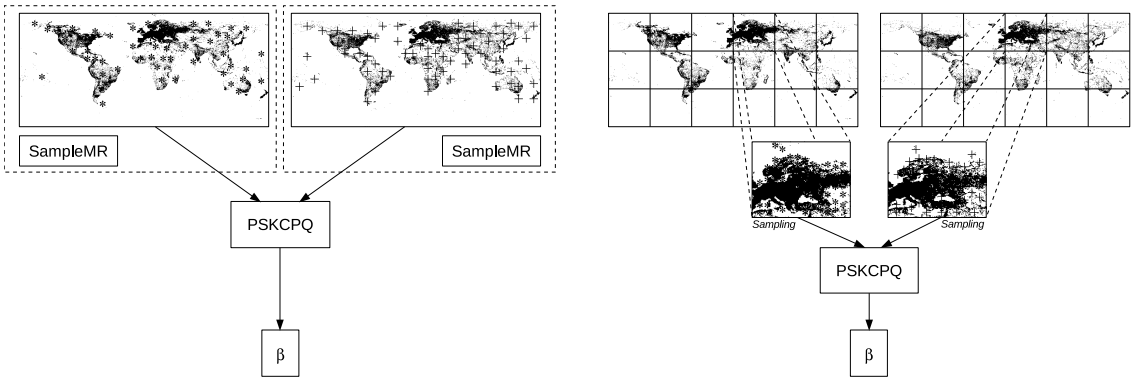

Fig. 3 Schema for computing β. Global sampling vs. local sampling (with Grid partitioning technique).

5.2.2 Computing β by Local Approximate Methods

Several approximation techniques (ε-approximate, α-allowance, N-consider and *Time*-consider) have been proposed for distance-based queries using R-trees in [61]. These techniques can be also used to obtain approximate solutions with a faster execution time, trying to find a balance between computational cost and accuracy of the result. N-consider is an approximate technique that depends on the quantity of points to be combined and *Time*-consider depends only on the time for query processing. On the other hand, ε-approximate and α-allowance are distance-based approximate techniques, and can be used for adjustment of quality of the result (KCPQ). For this reason, we will consider them as candidates for application in our problem. Since $\varepsilon \geq 0$ values are unlimited, according to the conclusions of [61,10], it is not easy to adjust the β value (upper bound of the distance value of K-th closest pair). For this reason, here we will choose the α-allowance technique, where α is a bounded positive real number ($0 \leq \alpha \leq 1$). With this approximate method we can easily adjust the balance between execution time of the KCPQ algorithm and the accuracy of the final result. Notice that this α-allowance technique can be easily transformed to the ε-approximate technique with $\alpha = 1/(1+\varepsilon)$ [10].

According to [61], we can apply the α-allowance approximate technique in plane-sweep-based KCPQ algorithms ($PSKCPQ$) [9,11] and the three sliding variants (Strip, Window and Semi-Circle) to adjust the final result. It can be carried out by multiplying δ by $(1-\alpha)$, giving rise to $\alpha PSKCPQ$, since it is a distance-based approximate technique. In this case, when $\alpha = 0$ we will get the normal execution of the plane-sweep $PSKCPQ$ algorithm, when $\alpha = 1$ we will invalidate the δ value (it will be always 0) and no pair of points will be selected for the result. Finally, when $0 < \alpha < 1$, we can adjust the sizes of the strip, the window and the semi-circle over the sweeping axis, since all of them depend on the δ value. Therefore, the smaller α value, the larger the upper bound of the δ value (i.e. more points will be considered and fewer points will be discarded); on the other hand, the larger α value, the smaller the upper bound of the δ value (i.e. fewer points will be considered and more points will be discarded).

The schema to compute β by using the α-allowance approximate technique with a plane-sweep-based KCPQ algorithm (αPSKCPQ) is very similar to the schema of computing β by local sampling illustrated in the right diagram of Figure 3. The difference is essentially that sampling is not used in the selected pair of cells and all points from the two cells are combined by the $\alpha PSKCPQ$ algorithm, obtaining a β value in a faster way if the α value is large enough.

The adaptation of the previous Algorithm 4 to local approximate is straightforward. The CALCULATEβ function no longer accepts ρ as parameter, since we do not perform a sampling of the input datasets, but for each set we get a number of elements that allow us to work with in main memory. Furthermore we have a new α parameter and the function $PSKCPQ$ is replaced by the new $\alpha PSKCPQ$ function that takes this new parameter for the adjustment of the approximate technique. The next steps of the algorithm remain unmodified.

6 Performance Evaluation

This section provides the results of an extensive experimental study aiming at measuring and evaluating the efficiency of the algorithms proposed in Section 5. In particular, Subsection 6.1 describes the experimental settings. Subsection 6.2 shows experimentally the advantages of using the proposed techniques to compute β and use this upper bound distance for KCPQ in SpatialHadoop. Subsection 6.3 compares different

plane-sweep techniques and the use of local indices. Subsection 6.4 shows the effect of using different spatial partitioning techniques included in SpatialHadoop. Subsections 6.5 and 6.6 examine the effect of incrementing the K values for KCPQ and the ε values for εDJQ, respectively. Subsection 6.7 shows the scalability of the proposed *DBJQ* MapReduce algorithms, varying the number of computing nodes. Finally, in Subsection 6.8 a summary from the experimental results is reported.

6.1 Experimental Setup

For the experimental evaluation, we have used real 2d point and synthetic (clustered) datasets to test our *DBJQ* MapReduce algorithms in SpatialHadoop. For real-world datasets we have used three datasets from OpenStreetMap[1]: *BUILDINGS* which contains 115M points (25 GB) of buildings (see Figure 5), *LAKES* which contains 8.4M points (8.6 GB) of water areas, and *PARKS* which contains 10M points (9.3 GB) of parks and green areas [23].

For synthetic datasets, we have created *clustered data*, since data in real-world are often clustered or correlated; in particular, real spatial data may follow a distribution similar to the clustered one. We have generated several files of different sizes using our own generator of clustered distributions, implemented in SpatialHadoop and with a similar format to the real data. The sizes of the datasets are 25M (5.4 GB), 50M (10.8 GB), 75M (16.2 GB), 100M (21.6 GB) and 125M points (27 GB), with 2500 clusters in each dataset (uniformly distributed in the range [(-179.7582155, -89.96783429999999) - (179.84404100000003, 82.51129005000003)]), which is the MBR of *BUILDINGS*), where for a set having N points, *N/2500* points were gathered around the center of each cluster, according to Gaussian (normal) distribution with mean 0.0 and standard deviation 0.2 as in [49]. For example, for an artificial dataset of 100M of points, we have 2500 clusters uniformly distributed, and for each cluster we have generated 40000 points according to Gaussian distribution with (mean = 0.0, standard deviation = 0.2). In Figure 4, we can observe a small area of a clustered dataset. We made 5 combinations of synthetic datasets by combining two separate instances of datasets, for each of the above 5 cardinalities (i.e. $25MC1 \times 25MC2$, $50MC1 \times 50MC2$, $75MC1 \times 75MC2$, $100MC1 \times 100MC2$ and $125MC1 \times 125MC2$).

Moreover, to experiment with the biggest real dataset (*BUILDINGS*, which contains 115M points) for DBJQ MapReduce algorithms, we have created a new big quasi-real dataset from *LAKES* (8.4M), with a similar quantity of points. The creation process is as follows: taking one point of *LAKES*, p, we generate 15 new points gathered around p (i.e. the center of the cluster), according to the Gaussian distribution described above, resulting in a new quasi-real dataset, *CLUS_LAKES*, with around 126M of points (27.5 GB). This dataset has the same shape as *LAKES*, but with more dense areas along the world.

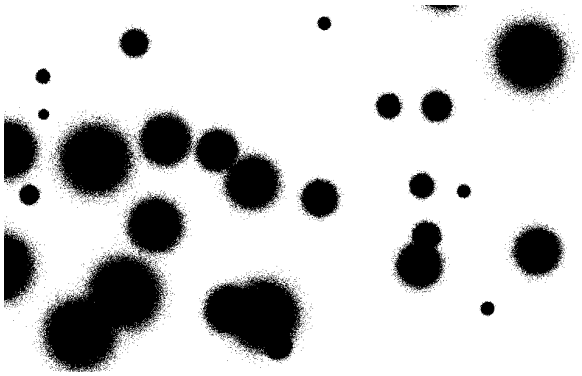

Fig. 4 Synthetic dataset. Small area from a clustered dataset.

To study the performance *DBJQ* MapReduce algorithms where two datasets are involved, we experimented using the above datasets and the most representative spatial partitioning techniques (Grid, Str, Quadtree and Hilbert) provided by SpatialHadoop, according to [56]. In our case, STR is equivalent to STR+ because we are working with points.

In Figures 5 and 6 (as an example) we show the effect of the partitioning phase using the STR technique [56] for *PARKS* and *BUILDINGS*, respectively. It is evident that each *cell* contains points which are close in space. If fact, all the partitioning methods respect spatial locality and distribute the points of a dataset to cells, considering (each method in a different way) spatial locality of these points. Since, processing of a pair of cells in a computing node during the *map* phase is only done if the spatial distance between these cells is below a threshold (avoiding unnecessary computations), the MapReduce algorithms we study take advantage of spatial locality.

To further study the spatial locality characteristics of the different spatial partitioning techniques, in Table 1, for each such technique, we show the number of cells generated by SpatialHadoop, the average of the number of points per cell and the standard deviation, for all real datasets. From this table, we can deduce that:

- The number of cells created by Quadtree partitioning is larger than the other methods [56] and this

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

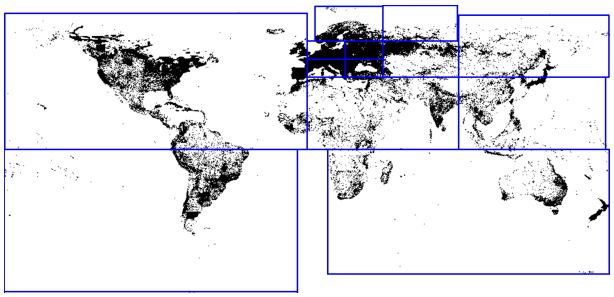

Fig. 5 Real-world dataset. *PARKS* (10M records of parks) with STR partitioning.

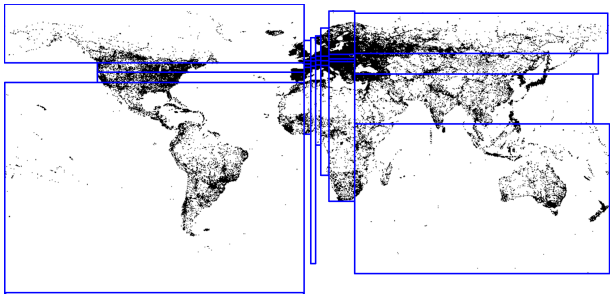

Fig. 6 Real-world dataset. *BUILDINGS* (115M records of buildings) with STR partitioning.

has as a result a smaller average number of points per cell.
- The standard deviation of the number of points per cell of Quadtree partitioning is larger than STR and Hilbert. This is explained by the fact that Quadtree partitioning divides space regularly, along fixed lines (the middle axes of the current subspace): an overflown area (quadrant) that is divided to four subquadrants may result to non-overflown cells (subquadrants) with uneven numbers of points. This area would probably be divided by STR or Hilbert to cells with borders not falling on the middle axes of the current subspace, but with almost equal numbers of points.
- The standard deviation of the number of points per cell of Quadtree partitioning is smaller than Grid, since Grid partitioning is not guided by data distribution.

These observations, along with the principles guiding the different partitioning techniques, lead to following conclusions regarding trends of query processing performance:

- The larger number of cells of Quadtree partitioning permits finer pruning of pairs of cells based on the distance between them (i.e. the pruning is more selective).

# of Cells	Grid	Str	Quadtree	Hilbert
LAKES	6	3	7	3
PARKS	6	3	13	3
BUILDINGS	24	28	78	27
CLUS_LAKES	36	45	115	42
Average	Grid	Str	Quadtree	Hilbert
LAKES	1403216	2806432	1202756	2806432
PARKS	2846245	3320619	766296	3320619
BUILDINGS	4783185	4099873	1471749	4251720
CLUS_LAKES	3508040	2806432	1098169	3006891
Stdev	Grid	Str	Quadtree	Hilbert
LAKES	1774152	6917	1192916	596
PARKS	2974069	6345	1109663	30441
BUILDINGS	12393021	20098	1190434	21064
CLUS_LAKES	8095904	14628	805211	16175

Table 1 Number of cells generated by SpatialHadoop, average and standard deviation of the spatial partitioning techniques for all real datasets.

- Quadtree, STR and Hilbert partitioning produce cells that adapt to the data distribution, contrary to Grid. This improves distance-based pruning of pairs of cells.
- Note that, when processing a pair of cells, the possible pairs of points that can be formed from these cells affects the necessary number of calculations during plane-sweep for this pair, but this is not the only such factor. The current distance threshold and the distribution of each dataset within the cell also affect the number of calculations. Depending on the distributions of the specific datasets involved, having larger collections of cells with varying numbers of points in Quadtree partitioning, or having smaller collections of cells with similar numbers of points in STR or Hilbert partitioning may favor load balancing between nodes.

To find out the actual effect of these trends on query processing performance, we performed extensive experimentation.

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.

The main performance measure that we have used in our experiments has been the total execution time (i.e. response time); this measurement is reported in seconds (*sec*) and represents the overall CPU time spent, as well as the I/O time needed by the execution of each *DBJQ* MapReduce algorithm in SpatialHadoop.

Table 2 summarizes the configuration parameters used in our experiments (sampling ratio values express % of the whole datasets). Default parameters (in parentheses) are used unless otherwise mentioned.

Parameter	Values (default)
K	1, 10, (10^2), 10^3, 10^4, 10^5
ε ($\times 10^{-4}$)	2.5, 5, 7.5, 12.5, (25), 50
α	0.0, 0.25, 0.50, (0.75), 0.85, 0.95
Sampling ratio, ρ	0.005, 0.01, 0.05, (0.1), 0.5, 1, 5, 10
% Dataset, γ	25, 50, 75, (100)
Number of nodes	1, 2, 4, 6, 8, 10, (12)
Type of partition	Grid, (Str), Quadtree, Hilbert
PS algorithms	Classic, (Reverse Run)
PS improvements	Strip, Window, (Semi-Circle)

Table 2 Configuration parameters used in our experiments.

6.2 The effect of applying the β computation

Our first experiment is to examine the use of β distance value for KCPQ MapReduce algorithms in SpatialHadoop (computed by global sampling (Algorithm 3), by local sampling (Algorithm 4) or by using the α-allowance approximate technique) as the upper bound of the distance value of the K-th closest pair.

As shown in Figure 7, upper chart, for large real datasets $LAKES \times PARKS$ (Grid) and different sampling ratios (ρ), the execution time is almost constant for the three methods. This trend in the results is mainly due to the fact that there is a trade-off between the time of sampling and β calculation with the one of the individual MapReduce tasks. With a larger sampling ratio ρ, a better β is obtained, which in turn improves the final *PSKCPQ* execution time. However, increasing the value of ρ also increases the time to calculate β. The use of β values accelerates the answer of the KCPQ and using the method of local sampling reduces the response time by around 22 times; whereas for the global sampling, the reduction is around 4 times faster than without β computation. This means that the use of local sampling shortens notably the execution time because by selecting suitably two cells for each dataset and applying sampling over this pair of cells reduces the computed β values and increases the power of pruning when it is passed to the *mappers*. For instance, for a sampling ratio (ρ) equal to 0.1%, the β values obtained by global sampling is 0.0144191, whereas by local sampling it is 0.0054841.

In the lower chart of Figure 7, we see a different behavior if we apply the STR partitioning technique for the same large datasets. We observe that the use of global sampling for the computation of β is more expensive than without β values in the preprocessing phase; this is due to the fact that with the dataset sizes and the used partitioning technique (STR), the time spent to perform the MapReduce sampling jobs (SampleMR) produces an overhead much higher than the improvement in response time that can be obtained. On the other hand, the use of local sampling to get the KCPQ is faster than the other two alternatives, because the time required to perform the local sampling is very small and the use of β improves the time of the individual *map* tasks. In addition, a similar trend is observed between global and local sampling that confirms that the improvement comes actually from reducing as much as possible the time required to obtain β. Finally, when comparing both charts, STR outperforms Grid due to the fact that STR is a partitioning technique based on how the data is distributed; therefore, partitions/cells with more uniform numbers of elements are produced, improving distance-based pruning of pairs of cells and load balancing between nodes. However, the Grid partitioning is based on a uniform division of space without taking into account the data; therefore, it produces some partitions with much more elements than others. so that certain *map* tasks can delay the total response time of the query. Note that, we have chosen for this first experiment the GRID and STR partitioning techniques, because they are used in [23] for performance comparison of the spatial queries and, GRID is the simplest (uniform grid of $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ grid cells, where n is the desired number of partitions) and STR corresponds to R-trees which are widely used (this technique bulk loads a random sample of the dataset into an R-tree using the STR algorithm [62] and the capacity of each node is set to $\lfloor k/n \rfloor$, where k is the sample size).

Figure 8 illustrates the same type of experiment (reporting the total execution times), but now for the biggest real datasets $BUILDINGS \times PARKS$. In the upper chart we can see the same trend for Grid partitioning as in Figure 7, where the preprocessing phase for computing β with local sampling is 2.7 times faster than using global sampling (whereas without the preprocessing phase needed around 21900 seconds and it is not depicted in the figure). In the lower chart, STR is faster than Grid (e.g. for $\rho = 0.1\%$ and global sampling, STR is 2.7 times faster than Grid), and the use of local sampling is also 80 seconds faster than global sampling for computing β for the same reasons explained previously. Notice that without the computation of β, around 2900 seconds to carry out the KCPQ were needed (not depicted in the figure). Again, comparing both charts, STR outperforms Grid according to the same reasons exposed above.

From these experiments we can conclude that the use of local sampling for computing β (Algorithm 4) generates smaller β values (e.g. $BUILDINGS \times PARKS$ (STR) and $\rho = 0.1\%$, the β value obtained by global sampling is 0.00211, whereas for local sampling it is 0.00050) and then this is more effective than global

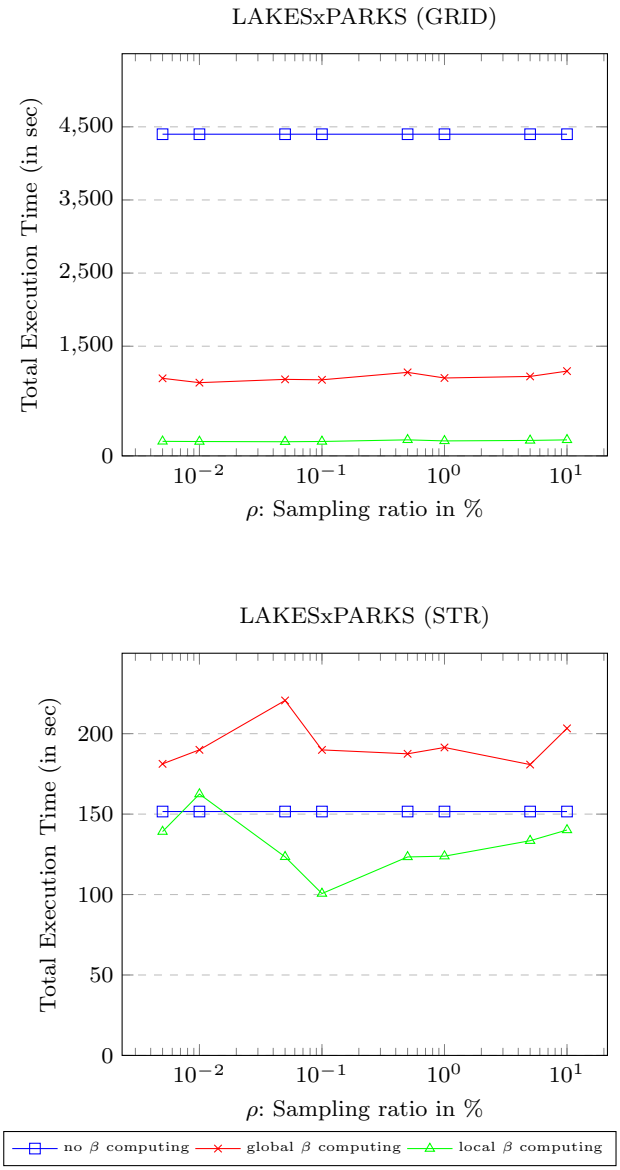

Fig. 7 KCPQ cost without and with β computation (large datasets).

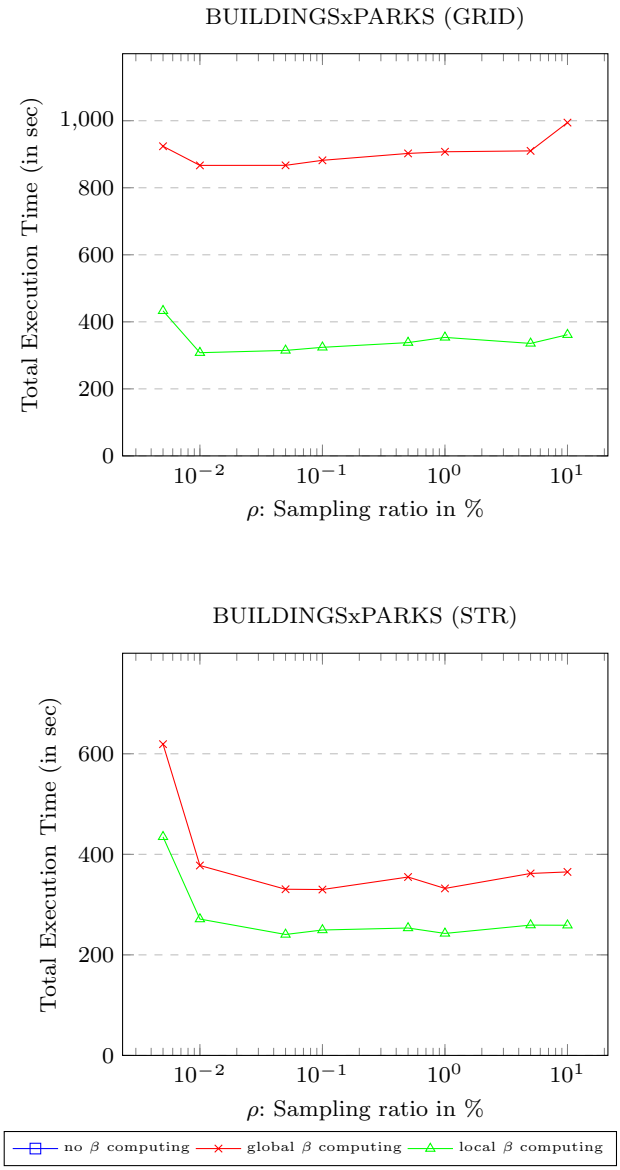

Fig. 8 KCPQ cost without and with β computation (big datasets).

sampling when it is passed to the *mappers*. Moreover, the partitioning technique is an important factor to take into account for this kind of distance-based join; in particular STR outperforms Grid in all cases. Finally, the value of ρ (sampling ratio) is an important parameter to be considered, and we have to find a trade-off between the time of sampling and the value of β computation (the smaller β value, the larger the time of sampling). Therefore, we have chosen $\rho = 0.1\%$ as the value for the remaining experiments, due to its excellent results.

Interesting results are also shown in Table 3, where all possible pairs of cells/partitions are shown, considering different percentages (γ) of the datasets ($BUILDINGS \times CLUS\_LAKES$ (STR)) and, with (GS $\equiv$ using global sampling and LS $\equiv$ using local sampling) or without using the computation of β for $K = 100$ (for other K values the percentage of reduction was similar). We can extract three interesting conclusions from this table: (1) with the use of β, we reduce significantly the number possible pairs of cells to be joined (e.g. using the complete datasets, only 120 out of 1260 possible pairs of cells are considered), (2) the β value returned by global or local sampling is not that determinant for the reduction of the number of pairs of cells to be combined (i.e. a smaller β value does not imply the reduction of the number of considered pairs of cells) as one can see in the two right columns; (3) the percentage of datasets to be joined is related with the number of considered pairs of cells when a β value is applied for the STR partitioning technique (e.g. the

75%, 50% and 25% of 120 are very close to 85, 55 and 32).

γ (%)	Without β	β GS	β LS
25	120	32	**32**
50	315	55	**55**
75	672	85	**84**
100	1260	120	**120**

Table 3 Number of considered pairs of cells without or with (global sampling (GS) or local sampling (LS)) β computation.

In Figure 9 we study the behavior of the KCPQ MapReduce algorithm in SpatialHadoop with respect to the total execution time, when β is computed locally from a suitable pair of cells by local sampling or by using the α-allowance approximate technique for the combination of the biggest datasets (real and artificial) and by using two partitioning techniques (Grid and STR). In the upper chart, one can see the trends for different sampling ratios (ρ). Again the STR partitioning reduces significantly the response time for real datasets (2.6 times faster when $\rho = 0.1\%$) with respect to Grid, but for the combination of synthetic data the reduction is smaller (1.3 times faster when $\rho = 0.1\%$); even for $\rho = 1.0\%$, $\rho = 5.0\%$ and $\rho = 10.0\%$ the execution times are almost the same. Moreover, notice that when ρ is larger than 0.5% the execution time with local sampling is increased slightly, since the time needed to compute β increases with the increment of the sampling ratio. In the lower chart, one can see the effect of applying the $\alpha PSKCPQ$ algorithm to the two selected cells for computing β by using different α values (0.0, 0.25, 0.50, 0.75, 0.85 and 0.95) to report the results of KCPQ. The response time is stable for all α values when the partitioning technique is Grid (real and synthetic) and STR (synthetic), but for $BUILDINGS \times CLUS\_LAKES$ (STR) the reduction from $\alpha = 0.95$ to $\alpha = 0.0$ is around 580 sec. Taking into account this result, we can deduce that the use of this approximate technique is useful for computing β, using high values of α. Moreover, for this case, the difference between $\alpha = 0.75$, $\alpha = 0.85$ and $\alpha = 0.95$ is very small. This behavior could be due to the fact that at the beginning of the $\alpha PSKCPQ$ processing, this algorithm gets quickly a small β value and then it is executed very fast. Finally, if we compare both charts of Figure 9, we can conclude that both techniques are very suitable to compute β and get the result of KCPQ in SpatialHadoop very fast, in particular when $\rho \in [0.1\%, 1.0\%]$ and $\alpha \in [0.75, 0.95]$.

Figure 10 shows the time spent in each phase that processing of the KCPQ in SpatialHadoop is split, when the three approaches to compute β are applied in the *pruning step* according to Figure 1. The configuration for this experiment is $BUILDINGS \times CLUS\_LAKES$, STR, $\rho = 0.1$, $K = 100$. The three phases are: *preprocessing*, *filtering* and *MapReduce*. The time spent in the *preprocessing* phase (STR) is the same for the three bars (498 sec), whereas the times spent for the *filtering* phase are different depending on the technique (global sampling, local sampling or approximate) applied for computing β. By using the local sampling, we get the smallest time spent (7 sec), next the approximate (40 sec) and the largest execution time is for global sampling (106 sec). When the filtering phase is ended, a β value is passed to the next phase; the smaller the β value, the faster the next phase (*MapReduce*). With this in mind, the time spent in the last

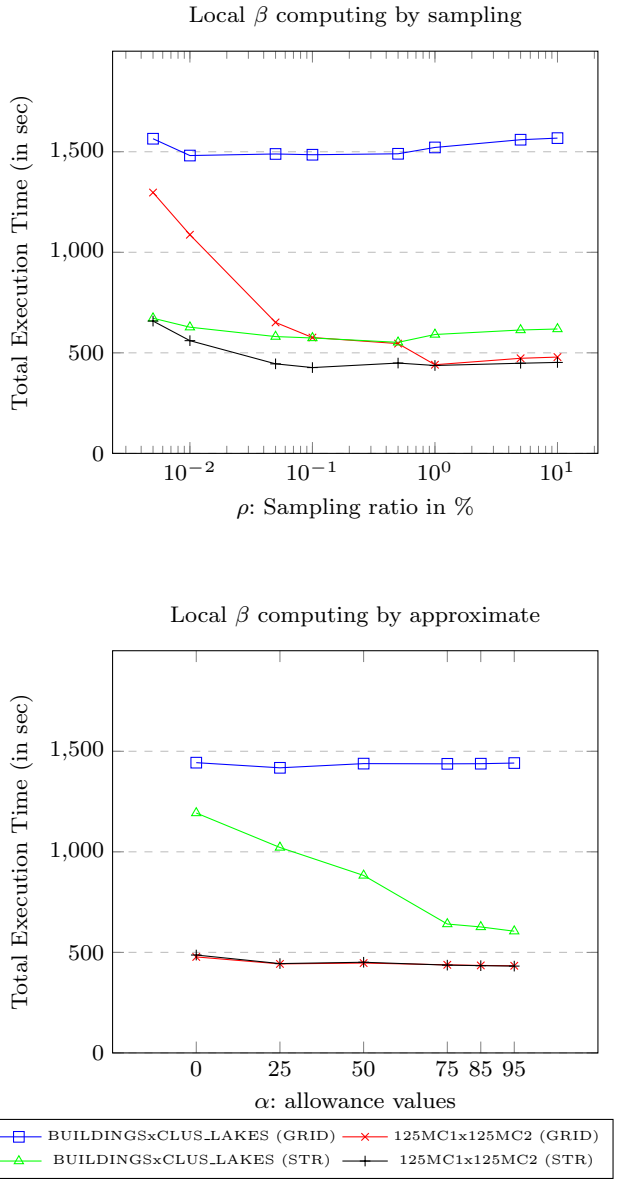

Fig. 9 KCPQ cost using local sampling and α-allowance approximate technique for β computation.

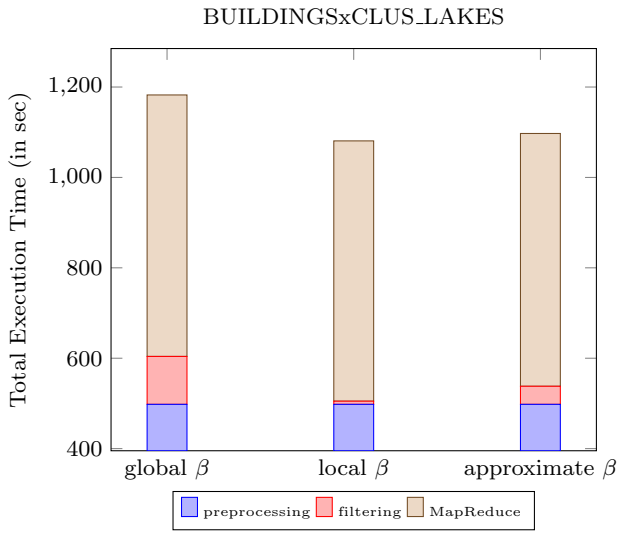

Fig. 10 KCPQ cost of different phases in the execution of KCPQ MapReduce algorithm in SpatialHadoop.

phase for the three techniques are: *global* $\beta = 578.498$ sec ($\beta = 0.00157$), *local* $\beta = 575.854$ sec ($\beta = 0.00062$) and *approximate* $\beta = 559.254$ sec ($\beta = 0.00013$).

6.3 Comparison of different plane-sweep algorithms and the use of local indices

This experiment aims to find the combination of one of the two different plane-sweep-based KCPQ algorithms (*Classic* and *Reverse Run*) and an improvement (Sliding Strip, Windows, or Semi-Circle) that has the best performance. As we can see in Table 4, the total execution times obtained do not show significant improvements between the different plane-sweep algorithms and variants. This is due to various factors such as reading disk speed, network delays, the time for each individual task, etc. As shown in this table, the difference between them is not quite significant (mainly for large datasets $LAKES \times PARKS$ (LxP)), the *Semi-Circle Reverse Run* algorithm being the fastest in all cases, and the *Classic Strip* the slowest variant (with the largest execution time). This is due to the fact that the *Reverse Run* algorithm has been specifically designed to reduce the number of distance computations [9,11]. For this reason we have chosen the *Semi-Circle Reverse Run* as the plane-sweep algorithm for all our experiments.

Finally, since our framework to perform *DBJQs* in SpatialHadoop can utilize local indices (R-trees), we have used this possibility to execute the KCPQ to compare it with the plane-sweep algorithms (without indices). To achieve this, we have adapted the distributed join algorithm [23] to perform the distributed KCPQ using the *Reverse Run* plane-sweep technique in each combination of pairs of nodes, in a similar way that the *Classic* one is used in [8]. The running time is shown in the last row of Table 4, and it is slower than the execution times of the plane-sweep-based algorithms without using the local indices (R-trees). The reason why the use of local indices is slower is the fragmentation of the data produced by the R-tree's own structure. When no local indices are used, all elements present in the corresponding cells are loaded into main memory, and then the appropriate plane-sweep-based KCPQ algorithm is performed. However, when using R-tree structures the data are finally stored in the leaves and the number of leaves is determined by the degree of the tree. This degree, for the node size and configuration used for the experiments, is 26 (suggested by [23]). When finally it is necessary to compare leaf nodes, multiple *PSKCPQ* algorithms with small quantities of data are performed. The sum of execution times of these tasks becomes greater than working with all the data in the cells directly in main memory. We can see this behavior when two big datasets are combined, $BUILDINGS \times PARKS$ (BxP), where *Reverse Run Semi-Circle* is around 30% faster than using the local indices (R-trees).

KCPQ Algorithm	LxP	BxP
Classic Strip	126.871	293.852
Classic Window	124.661	283.441
Classic Semi-Circle	121.263	267.171
Reverse Strip	123.013	276.398
Reverse Window	121.768	230.390
Reverse Semi-Circle	120.648	229.226
Local indices (R-tree)	147.023	318.450

Table 4 Total execution time (in seconds) spent by each KCPQ algorithm, plane-sweep without indices and with local indices (R-tree).

For the εDJQ we have designed and executed the same type of experiment as the one for the KCPQ, to detect which is the best variant of plane-sweep algorithm. Table 5 shows these results, and we can observe that the Strip variant of *Classic* and *Reverse Run* is the slowest, but Window and Semi-Circle have very close execution times, the *Classic Semi-Circle* being slightly the fastest. Moreover, as for the KCPQ, we have adapted the distributed join algorithm [23] to implement a distributed εDJQ algorithm using the *Classic* plane-sweep technique in each combination of pairs of nodes of local R-trees. The total execution time is shown in the last row of Table 5, which it is much slower than the execution times of the plane-sweep-based algorithms without using the local indices (R-trees). The justification of this behavior is very similar to the one exposed above for the KCPQ. We can highlight that when two large datasets, $LAKES \times PARKS$ (LxP), are combined, the

Classic Semi-Circle is around 23 times faster than using the local indices, while for the join of two big datasets, $BUILDINGS \times PARKS$ (BxP), *Classic Semi-Circle* is around 25 times faster.

εDJQ Algorithm	LxP	BxP
Classic Strip	275.701	2798.069
Classic Window	98.024	418.473
Classic Semi-Circle	91.923	391.612
Reverse Strip	268.777	2506.165
Reverse Window	99.150	437.814
Reverse Semi-Circle	98.981	434.038
Local indices (R-tree)	2129.338	9748.563

Table 5 Total execution time (in seconds) spent by each εDJQ algorithm, plane-sweep without indices and with local indeces (R-tree).

6.4 The effect of using different spatial partitioning techniques

In [56], seven different partitioning techniques 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 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) [56]. The *distribution-based techniques* assign an object to exactly one overlapping cell and the cell has to be expanded to enclose all contained records. The *replication-based techniques* avoid expanding cells by replicating each record to all overlapping cells) but the query processor has to employ a duplicate avoidance technique to account for replicated elements (in accordance to the literature, we follow this naming of techniques, although, in the case of points no replication takes place). The most important conclusions in [56] 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 (except for Z-Curve, that reports the worst running times), and (2) the Quadtree outperforms all other techniques with respect to the running time, since it minimizes the number of overlapping partitions between the two files by employing a regular space partitioning. According to the first conclusion, we are going to experiment with the *DBJQ* MapReduce algorithms, where both datasets are partitioned with the same technique. Finally, the partitioning techniques that we have chosen are: Grid, STR, Quadtree and Hilbert-Curve, because they showed the best performance for distributed overlap join in [56].

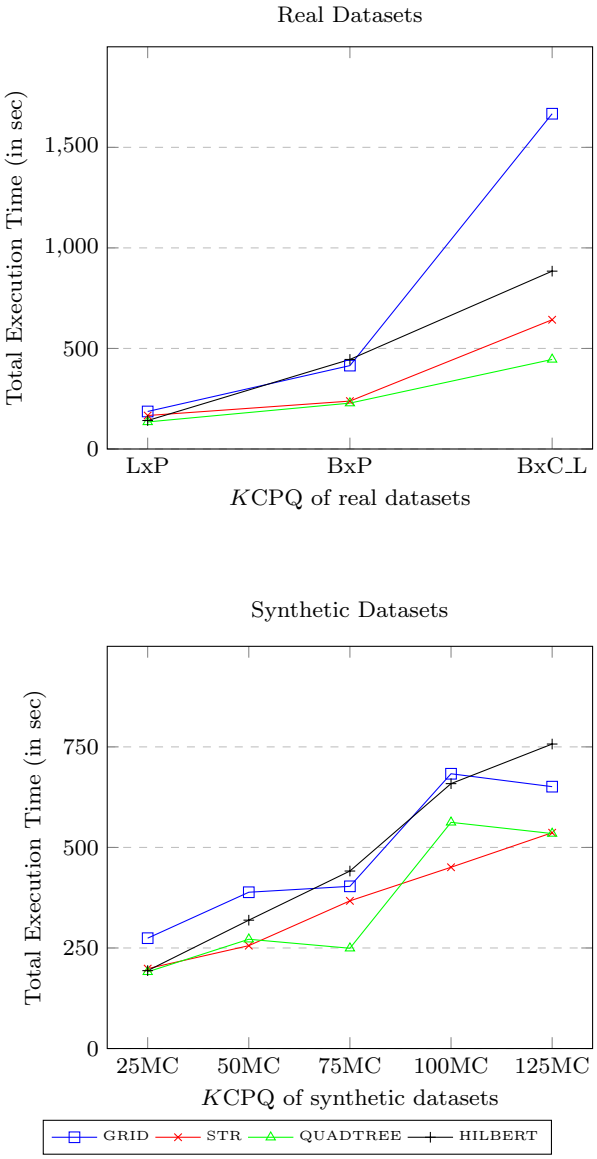

Fig. 11 KCPQ cost considering different partition techniques in SpatialHadoop.

As shown in the upper part of Figure 11 for the KCPQ of real datasets ($LAKES \times PARKS$, $BUILDINGS \times PARKS$ and $BUILDINGS \times CLUS\_LAKES$), the choice of a partitioning technique clearly affects the execution time. For instance, Quadtree is the fastest (445 sec), the *STR* is the second (642 sec), the third is Hilbert (884 sec) and the slowest is the Grid (1667 sec), for the combination of the biggest real datasets, $BUILDINGS \times CLUS\_LAKES$ (BxC_L). Moreover, we can see that the influence of the partitioning technique is less for the combination of the smallest datasets, $LAKES \times PARKS$ (LxP), where the execution times are almost the same (e.g. Quadtree is only 32 sec faster than STR). The be-

havior for synthetic datasets is different (see the lower chart of Figure 11), due to the nature of the data distribution (uniform distribution of the centers of the clusters) and the type of partitioning technique (replication-based and distribution-based). The trends of replication-based techniques (Quadtree and Grid) are very similar, as is the case for distribution-based (STR and Hilbert). Moreover, for the combination of the biggest synthetic datasets, $125MC1 \times 125MC2$ (125M), the fastest partitioning technique is Quadtree (534 sec), and STR has a very close running time (only 2 sec slower), Grid takes 651 sec and Hilbert is the slowest with 757 sec. Note that, a label like *25MC* on x-axis of the chart for synthetic datasets signifies the combination $25MC1 \times 25MC2$.

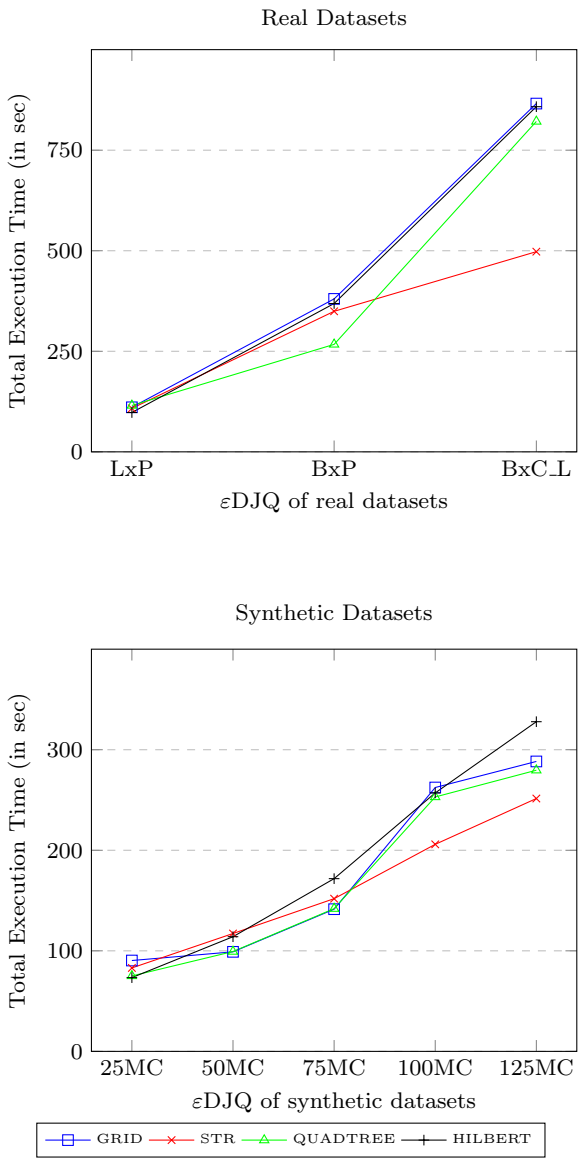

Fig. 12 εDJQ cost considering different partition techniques in SpatialHadoop.

As we have just seen for KCPQ, the choice of a partitioning technique clearly affects the execution time of εDJQ, regardless of whether the datasets are real or synthetic. For instance, for real datasets (see the upper chart of Figure 12), for the combination of large datasets, $LAKES \times PARKS$ (LxP), Hilbert partitioning is slightly faster than the other techniques (e.g. it is 11 sec faster than STR, which is the second), but for $BUILDINGS \times PARKS$ (BxP), Quadtree is the fastest (82 sec faster than the second, STR), and for the big datasets, $BUILDINGS \times CLUS\_LAKES$ (BxC_L), STR is the fastest (324 sec faster than Quadtree). From these results with real data, we can conclude that the bigger the datasets, the better the performance of STR for εDJQ. The behavior for synthetic dataset is also different (see the lower chart of Figure 12), mainly due to the nature of the data distribution and the type of partitioning technique. In the same way as for KCPQ, the trends of replication-based techniques (Quadtree and Grid) are very similar, as the case for distribution-based (STR and Hilbert), with small gaps between them. Moreover, for the combination of large synthetic datasets, $25MC1 \times 25MC2$ (25M), again Hilbert is slightly the fastest (only 2 sec faster than Quadtree). The Quadtree is the fastest for the combination of $50MC1 \times 50MC2$ (50M) and $75MC1 \times 75MC2$ (75M), while STR is the fastest for the biggest synthetic datasets (e.g. for $125MC1 \times 125MC2$ (125M), STR is 28 sec faster than Quadtree, which is the second). In the same way as for real datasets, we can conclude for synthetic data that the bigger the datasets, the better the performance of STR for εDJQ. Note as well that, when we write on the x-axis of the chart for synthetic datasets $25MC$, we really mean $25MC1 \times 25MC2$.

Last, it is very important to highlight the behavior of *Quadtree* partitioning technique, that reports the smallest execution times in most of the cases (mainly for real datasets and KCPQ), as in [56] for distributed overlap join. This will be the partitioning technique to apply in the remainder experiments, together with STR, which shows an excellent performance for εDJQ using big datasets.

6.5 The effect of the increment of K values

This experiment studies the effect of increasing of the K value for the combination of the biggest datasets (real and artificial). The upper chart of Figure 13 shows the total execution time for real datasets ($BUILDINGS \times CLUS\_LAKES$) grows slowly as the number of results to be obtained (K) increases, until $K = 10^4$, but for $K = 10^5$ the increment is larger mainly for STR (around 850 sec). The *Quadtree* reports

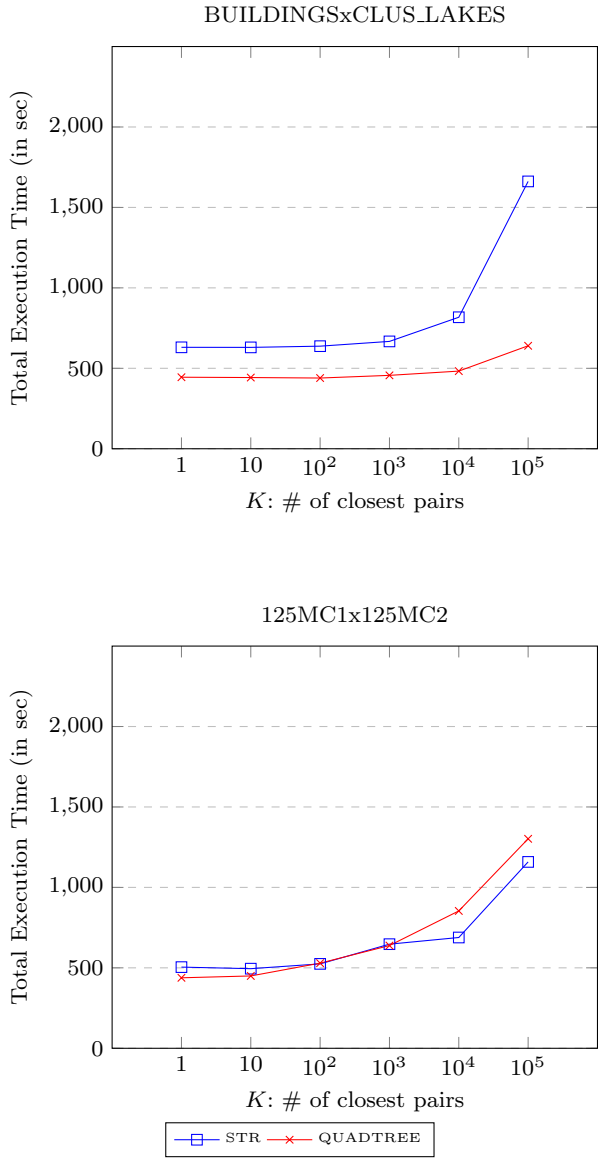

Fig. 13 KCPQ cost (execution time) vs. K values.

the best execution times, even for large K values (e.g. $K = 10^5$). This means that the Quadtree is less affected by the increment of K, because Quadtree employs regular space partitioning depending on the concentration of the points. For the combination of synthetic datasets ($125MC1 \times 125MC2$) in the lower chart, for small K values the Quadtree is slightly faster than STR, but for larger K values the roles are swapped and STR is faster than Quadtree.

The main conclusions that we can extract for this experiment are: (1) the Quadtree again satisfies KCPQ in the fastest way, mainly for real datasets, and (2) the higher the K values, the greater the possibility that pairs of cells are not pruned, more *map* tasks could be needed and more total execution time is needed.

6.6 The effect of the increment of ε for εDJQ

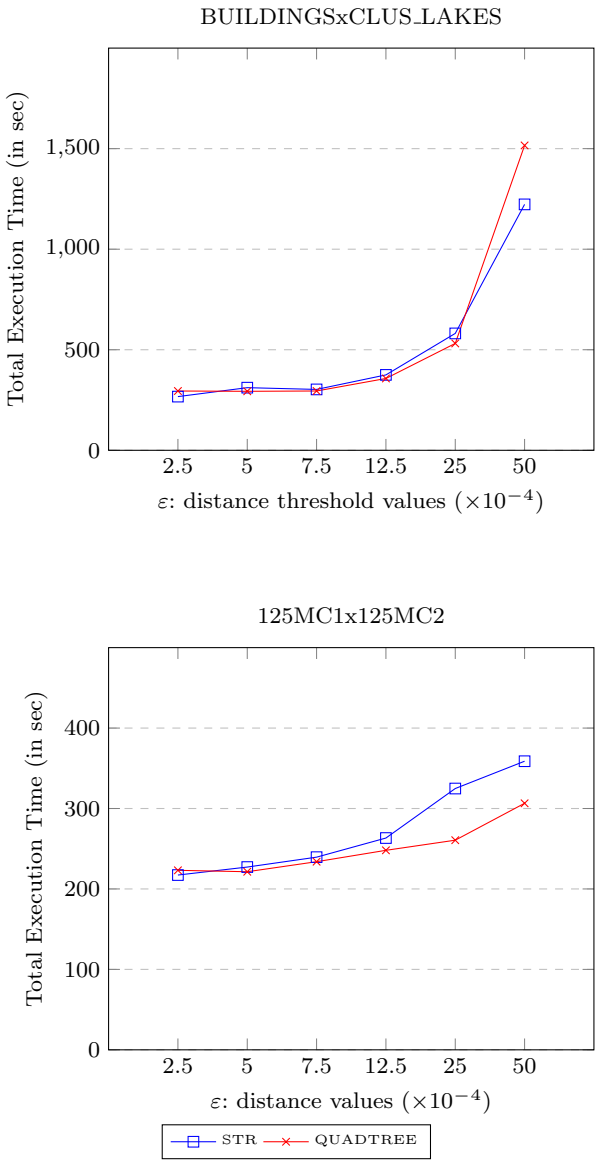

Fig. 14 εDJQ cost (execution time) vs. ε values.

In this experiment we study the effect of increasing of the ε value in εDJQ MapReduce algorithm in SpatialHadoop for the combination of the biggest datasets (real and synthetic). As shown in the upper chart of Figure 14, the total execution time for real datasets ($BUILDINGS \times CLUS\_LAKES$) grows as the ε value increases. Both partitioning techniques (Quadtree and STR) have similar performance for all ε values, except for $\varepsilon = 50 \times 10^{-4}$, where STR outperforms Quadtree (i.e. STR is 295 sec faster). For the combination of synthetic datasets ($125MC1 \times 125MC2$) in the lower chart, for small ε values both techniques (Quadtree and STR) have the same performance, but for larger ε va-

lues Quadtree is faster than STR (e.g. Quadtree is 65 sec faster for $\varepsilon = 25 \times 10^{-4}$).

Similar conclusions to the KCPQ can be extracted for the εDJQ: (1) the Quadtree outperforms STR for the εDJQ mainly for synthetic datasets (for real datasets, except for large ε values) and (2) the higher the ε values, the greater the possibility that pairs of cells are not pruned, more *map* tasks are needed and more total execution time is needed.

6.7 The speedup of the algorithms

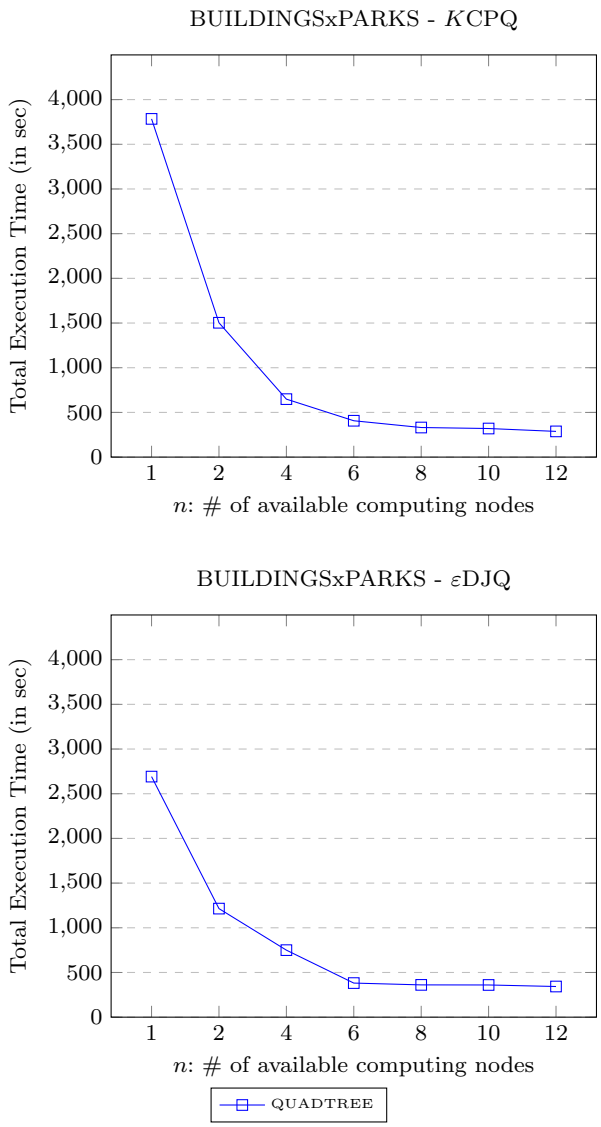

Fig. 15 Query cost with respect to the number of computing nodes n.

This experiment aims to measure the speedup of the *DBJQ* MapReduce algorithms (KCPQ and εDJQ), varying the number of computing nodes (n). We have used the Quadtree as the partitioning technique, but STR follows the same trend. The upper chart of Figure 15 shows the impact of different number of computing nodes on the performance of parallel KCPQ algorithm, for $BUILDINGS \times PARKS$ with the default configuration values. From this chart, it could be concluded that the performance of our approach has 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. However, when the number of computing nodes exceeds the number of *map* tasks, no improvement for the whole job is obtained. In the lower chart of Figure 15, we can observe a similar trend for εDJQ MapReduce algorithm with less execution time, and we can extract the same conclusions.

6.8 Conclusions from the experiments

We have experimentally demonstrated the *efficiency* (in terms of total execution time) and the *scalability* (in terms of K and ε values, sizes of datasets and number of computing nodes) of the proposed parallel algorithms for *DBJQ*s (the KCPQ and εDJQ) in SpatialHadoop. By studying the experimental results, we can extract several conclusions that are shown below:

- The algorithm proposed in [1] for the KCPQ is significantly improved by utilizing alternative methods for the computation of an upper bound β of the distance of the K-th closest pair. More specifically, we proposed new such methods that use a local preprocessing phase and are based either on sampling, or on the α-allowance approximate technique, and, through an extensive set of experiments, we have shown the improved efficiency of the new methods.
- Alternative plane-sweep-based algorithms (*Classic* and *Reverse Run*) in the MapReduce implementation have similar performances, in terms of execution time, although they are faster than using local indices (R-trees) in each *map* task.
- The *Quadtree*, or the *STR* spatial partitioning technique included in SpatialHadoop (instead of the *Grid* or *Hilbert* ones) improves notably the efficiency of the parallel *DBJQs* algorithms. This is due to the partition of space according to the data distribution (the concentration of the cells depends on the concentration of points) [56].
- The larger the K or ε values, the larger the probability that pairs of cells are not pruned, more *map* tasks will be needed and more total execution time is spent for reporting the final result.
- The larger the number of computing nodes (n), the faster the DBJQ MapReduce algorithms are, but

when n exceeds the number of *map* tasks, no improvement for the whole job is obtained.

7 Concluding Remarks and Future Work

DBJQs (the KCPQ and εDJQ) are operations widely adopted by many spatial and GIS applications. Both operations are costly, especially in large-scale datasets, since the combination (Cartesian Product) of two spatial datasets is coupled with additional constraints. These *DBJQs* have been actively studied in centralized environments. However, for parallel and distributed frameworks they have not attracted similar attention. For this reason, here we studied the problem of processing the most representative *DBJQs* (the KCPQ and εDJQ) in SpatialHadoop, an extension of Hadoop supporting spatial operations efficiently.

To achieve this, we have proposed new MapReduce algorithms in SpatialHadoop on big spatial datasets, adopting the plane-sweep technique. For the KCPQ, we have improved the MapReduce algorithm presented in [1], regarding the computation of an upper bound (β) of the distance value of the K-th closest pair, by using a local preprocessing phase based either on sampling, or on approximate techniques. We have shown experimentally the efficiency of such improvements, taking into account different comparison parameters and performance measures. We have also proposed the first MapReduce algorithm in SpatialHadoop for the εDJQ. More specifically, we have implemented the *Reverse Run* plane-sweep algorithm [9,11] for the εDJQ, following a similar scheme to that for the KCPQ. The result is achieved in competitive response times to the response times obtained with an alternative method, the distributed εDJQ computation using local R-trees indices.

We performed a detailed performance comparison of the proposed algorithms in various scenarios with big synthetic and real-world points datasets. The execution of such experiments has demonstrated the *efficiency* (in terms of total execution time) and *scalability* (in terms of K and ε values, sizes of datasets, number of computing nodes, etc.) of our proposals.

As part of our future work, we are planning to extend the current results in several contexts:

- implement other *DBJQs* in SpatialHadoop, like the KNN join query framework [15] and distance join queries with spatial constraints [63],
- implement other complex spatial queries in SpatialHadoop, like multi-way spatial joins [64] and multi-way distance joins queries [65],
- implement other partitioning techniques [66,67] in SpatialHadoop, because this is an important factor for processing distance-based join queries, as we have demonstrated.
- implement KCPQs and εDJQs in Spark-based distributed spatial data management systems, like LocationSpark [31].

Acknowledgements Work of all authors funded by the MINECO research project [TIN2013-41576-R].

References

1. F. García-García, A. Corral, L. Iribarne, M. Vassilakopoulos, Y. Manolopoulos, Enhancing spatialhadoop with closest pair queries, in: ADBIS Conference, 2016, pp. 212–225.
2. S. Shekhar, S. Chawla, Spatial databases - a tour, Prentice Hall, 2003.
3. H. Samet, Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS, Addison-Wesley, Boston, MA, 1990.
4. J. H. Schiller, A. Voisard (Eds.), Location-Based Services, Morgan Kaufmann, 2004.
5. P. Rigaux, M. Scholl, A. Voisard, Spatial databases - with applications to GIS, Elsevier, San Francisco, CA, 2002.
6. L. H. U, N. Mamoulis, M. L. Yiu, Computation and monitoring of exclusive closest pairs, Trans. Knowl. Data Eng. 20 (12) (2008) 1641–1654.
7. E. Ahmadi, M. A. Nascimento, K-closest pairs queries in road networks, in: MDM Conference, 2016, pp. 232–241.
8. 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.
9. G. Roumelis, A. Corral, M. Vassilakopoulos, Y. Manolopoulos, A new plane-sweep algorithm for the k-closest-pairs query, in: SOFSEM Conference, 2014, pp. 478–490.
10. Y. Gao, L. Chen, X. Li, B. Yao, G. Chen, Efficient k-closest pair queries in general metric spaces, VLDB J. 24 (3) (2015) 415–439.
11. 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.
12. C. Zhang, F. Li, J. Jestes, Efficient parallel kNN joins for large data in MapReduce, in: EDBT Conference, 2012, pp. 38–49.
13. W. Lu, Y. Shen, S. Chen, B. C. Ooi, Efficient processing of k nearest neighbor joins using MapReduce, PVLDB 5 (10) (2012) 1016–1027.
14. K. Wang, J. Han, B. Tu, J. Dai, W. Zhou, X. Song, Accelerating spatial data processing with MapReduce, in: ICPADS Conference, 2010, pp. 229–236.
15. 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- and Knowledge-Centered Systems 24 (2016) 139–168.
16. Y. N. Silva, J. M. Reed, Exploiting mapreduce-based similarity joins, in: SIGMOD Conference, 2012, pp. 693–696.
17. J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, in: OSDI Conference, 2004, pp. 137–150.

18. F. Li, B. C. Ooi, M. T. Özsu, S. Wu, Distributed data management using mapreduce, ACM Comput. Surv. 46 (3) (2014) 31:1–31:42.
19. C. L. P. Chen, C. Zhang, Data-intensive applications, challenges, techniques and technologies: A survey on big data, Information Sciences 275 (2014) 314–347.
20. R. Giachetta, A framework for processing large scale geospatial and remote sensing data in mapreduce environment, Computers & Graphics 49 (2015) 37–46.
21. A. Gani, A. Siddiqa, S. Shamshirband, F. Hanum, A survey on indexing techniques for big data: taxonomy and performance evaluation, Knowl. Inf. Syst. 46 (2) (2016) 241–284.
22. C. Doulkeridis, K. Nørvåg, A survey of large-scale analytical query processing in mapreduce, VLDB Journal 23 (3) (2014) 355–380.
23. A. Eldawy, M. F. Mokbel, Spatialhadoop: A mapreduce framework for spatial data, in: ICDE Conference, 2015, pp. 1352–1363.
24. J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, F. Özcan, Clash of the titans: Mapreduce vs. spark for large scale data analytics, PVLDB 8 (13) (2015) 2110–2121.
25. J. Lu, R. H. Güting, Parallel secondo: Boosting database engines with Hadoop, in: ICPADS Conference, 2012, pp. 738–743.
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. S. You, J. Zhang, L. Gruenwald, Large-scale spatial join query processing in cloud, in: ICDE Workshops, 2015, pp. 34–41.
29. J. Yu, J. Wu, M. Sarwat, Geospark: a cluster computing framework for processing large-scale spatial data, in: SIGSPATIAL Conference, 2015, pp. 70:1–70:4.
30. D. Xie, F. Li, B. Yao, G. Li, L. Zhou, M. Guo, Simba: Efficient in-memory spatial analytics, in: SIGMOD Conference, 2016, pp. 1071–1085.
31. M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, W. G. Aref, Locationspark: A distributed in-memory data management system for big spatial data, PVLDB 9 (13) (2016) 1565–1568.
32. Z. Li, Q. Huang, G. J. Carbone, F. Hu, A high performance query analytical framework for supporting data-intensive climate studies, Computers, Environment and Urban Systems 62 (2017) 210–221.
33. J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, S. A. Brandt, Scihadoop: array-based query processing in hadoop, in: SC Conference, 2011, pp. 66:1–66:11.
34. A. Eldawy, M. F. Mokbel, S. Al-Harthi, A. Alzaidy, K. Tarek, S. Ghani, SHAHED: A mapreduce-based system for querying and visualizing spatio-temporal satellite data, in: ICDE Conference, 2015, pp. 1585–1596.
35. R. Palamuttam, R. M. Mogrovejo, C. Mattmann, B. Wilson, K. Whitehall, R. Verma, L. J. McGibbney, P. M. Ramirez, Scispark: Applying in-memory distributed computing to weather event detection and tracking, in: Conference on Big Data, 2015, pp. 2020–2026.
36. S. Zhang, J. Han, Z. Liu, K. Wang, S. Feng, Spatial queries evaluation with MapReduce, in: GCC Conference, 2009, pp. 287–292.
37. Q. Ma, B. Yang, W. Qian, A. Zhou, Query processing of massive trajectory data based on MapReduce, in: CloudDb Conference, 2009, pp. 9–16.
38. A. Akdogan, U. Demiryurek, F. B. Kashani, C. Shahabi, Voronoi-based geospatial query processing with MapReduce, in: CloudCom Conference, 2010, pp. 9–16.
39. J. Maillo, I. Triguero, F. Herrera, A mapreduce-based k-nearest neighbor approach for big data classification, in: TrustCom/BigDataSE/ISPA Conference, 2015, pp. 167–172.
40. Y. Park, J. Min, K. Shim, Parallel computation of skyline and reverse skyline queries using mapreduce, PVLDB 6 (14) (2013) 2002–2013.
41. J. Zhang, X. Jiang, W. Ku, X. Qin, Efficient parallel skyline evaluation using mapreduce, IEEE Trans. Parallel Distrib. Syst. 27 (7) (2016) 1996–2009.
42. C. Ji, Z. Li, W. Qu, Y. Xu, Y. Li, Scalable nearest neighbor query processing based on inverted grid index, J. Network and Computer Applications 44 (2014) 172–182.
43. S. Zhang, J. Han, Z. Liu, K. Wang, Z. Xu, SJMR: parallelizing spatial join with MapReduce on clusters, in: CLUSTER Conference, 2009, pp. 1–8.
44. J. M. Patel, D. J. DeWitt, Partition based spatial-merge join, in: SIGMOD Conference, 1996, pp. 259–270.
45. Y. Kim, K. Shim, Parallel top-k similarity join algorithms using MapReduce, in: ICDE Conference, 2012, pp. 510–521.
46. E. H. Jacox, H. Samet, Metric space similarity joins, ACM Trans. Database Syst. 33 (2) (2008) 1–38.
47. H. Gupta, B. Chawda, S. Negi, T. A. Faruquie, L. V. Subramaniam, M. K. Mohania, Processing multi-way spatial joins on map-reduce, in: EDBT Conference, 2013, pp. 113–124.
48. H. Wang, A. Belhassena, Parallel trajectory search based on distributed index, Information Sciences 388-399 (2017) 62–83.
49. A. Eldawy, Y. Li, M. F. Mokbel, R. Janardan, Cg_hadoop: computational geometry in mapreduce, in: SIGSPATIAL Conference, 2013, pp. 284–293.
50. D. Pertesis, C. Doulkeridis, Efficient skyline query processing in spatialhadoop, Information Systems 54 (2015) 325–335.
51. A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Closest pair queries in spatial databases, in: SIGMOD Conference, 2000, pp. 189–200.
52. G. R. Hjaltason, H. Samet, Incremental distance join algorithms for spatial databases, in: SIGMOD Conference, 1998, pp. 237–248.
53. H. Shin, B. Moon, S. Lee, Adaptive and incremental processing for distance join queries, IEEE Trans. Knowl. Data Eng. 15 (6) (2003) 1561–1578.
54. C. Yang, K. Lin, An index structure for improving closest pairs and related join queries in spatial databases, in: IDEAS Conference, 2002, pp. 140–149.
55. G. Gutierrez, P. Sáez, The k closest pairs in spatial databases - when only one set is indexed, GeoInformatica 17 (4) (2013) 543–565.
56. A. Eldawy, L. Alarabi, M. F. Mokbel, Spatial partitioning techniques in spatial hadoop, PVLDB 8 (12) (2015) 1602–1613.
57. F. P. Preparata, M. I. Shamos, Computational Geometry - An Introduction, Springer, 1985.
58. A. Corral, J. M. Almendros-Jiménez, A performance comparison of distance-based query algorithms using r-trees in spatial databases, Information Sciences 177 (11) (2007) 2207–2237.

59. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms (3. ed.), MIT Press, 2009.
60. S. Chaudhuri, R. Motwani, V. R. Narasayya, On random sampling over joins, in: SIGMOD Conference, 1999, pp. 263–274.
61. A. Corral, M. Vassilakopoulos, On approximate algorithms for distance-based queries using r-trees, Computer Journal 48 (2) (2005) 220–238.
62. S. T. Leutenegger, J. M. Edgington, M. A. Lopez, Str: A simple and efficient algorithm for r-tree packing, in: ICDE Conference, 1997, pp. 497–506.
63. A. N. Papadopoulos, A. Nanopoulos, Y. Manolopoulos, Processing distance join queries with constraints, Computer Journal 49 (3) (2006) 281–296.
64. N. Mamoulis, D. Papadias, Multiway spatial joins, ACM Trans. Database Syst. 26 (4) (2001) 424–475.
65. A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Multi-way distance join queries in spatial databases, GeoInformatica 8 (4) (2004) 373–402.
66. H. Vo, A. Aji, F. Wang, SATO: a spatial data partitioning framework for scalable query processing, in: SIGSPATIAL Conference, 2014, pp. 545–548.
67. A. Aji, H. Vo, F. Wang, Effective spatial data partitioning for scalable query processing, CoRR abs/1509.00910.