Binary Space Partitioning for Parallel and Distributed Closest-Pairs Query Processing

George Mavrommatis, Panagiotis Moutafis, and Michael Vassilakopoulos

Data Structuring & Engineering Lab
Dept. of Electrical & Computer Eng.
University of Thessaly
Volos, Greece
Email: {gmav, pmoutafis, mvasilako}@uth.gr

***Abstract*—The (k) Closest-Pair(s) Query, kCPQ, consists in finding the (k) closest pair(s) of objects between two spatial datasets. Up to date, only few solutions have appeared that process the kCPQ in parallel and distributed frameworks. Currently, Apache Spark is the state of the art of parallel and distributed frameworks, having several advantages compared to other popular ones, like Hadoop MapReduce. A major step towards answering a query is proper partitioning, a task that is of even greater importance in distributed environments. In this work, we present algorithms for processing the kCPQ in Apache Spark that split the datasets into strips across an axis. Two variations of the Binary Space Partitioning (BSP) technique are used to partition the data, based on two different criteria: equal size and equal width of the child strips. These schemes are compared to a third strategy (previously developed by us), namely splitting into a predefined number of strips. We have performed an extensive set of experiments to evaluate the efficiency and scalability of the algorithm and the performance of the different partitioning schemes by using large real-world datasets. Results show that splitting into strips by means of BSP achieves better performance. This is mainly due to the fact that selecting the number of points within each strip as the preset criterion, instead of the number of strips, provides more flexibility in fine tuning the system.**

***Index Terms*—Closest-Pairs Query; Spatial Query Processing; Apache Spark; Binary Space Partitioning.**

I. Introduction

The (k) Closest-Pair(s) Query (kCPQ) consists in finding the (k) closest pair(s) of objects between two spatial datasets. Up to date, only few solutions have appeared that process the kCPQ in parallel and distributed frameworks. Currently, Apache Spark is the leader of such frameworks, having several advantages compared to other popular ones, like Hadoop MapReduce. In [1], for the first time in the literature, we presented an algorithm for answering the kCPQ in this framework. In this paper, by extending the method of [1], we present new algorithms for answering the kCPQ in Apache Spark, along with an extended experimental comparison of their performance.

Geographic information systems (GIS) [2] have been around for several decades. They provide the means for storing, querying, analyzing and sharing geographic information and have proven valuable in many modern application domains (e.g., disaster management, mapping, urban planning, transportation planning, environmental impact analysis, etc.).

Spatial databases [3] are specialized databases that support storage and querying of multidimensional data (usually, points, line-segments, regions, polygons, volumes). They are core elements of GIS. Processing of spatial queries can become very demanding if the volume of data on which such a query is applied is large, or if the number of the combinations of data objects that need to be examined for answering such a query is large.

Some typical spatial queries are: the point query, range query, spatial join, and nearest neighbor query [4]. Spatial Join queries find all pairs of spatial objects from two spatial data sets that satisfy a spatial predicate, like intersects, contains, is enclosed by, etc. Nearest neighbor queries locate the spatial object(s) that is (are) nearest to a query object. The kCPQ discovers the (K) closest pair(s) of object(s) (usually ordered by distance), between two spatial datasets. It combines join and nearest neighbor queries: like a join query, all pairs (combinations) of objects from the two datasets are candidates for the result, and like a nearest neighbor query, the (K) smallest distance(s) is (are) the basis for inclusion in the result (and the final ordering) [5], [6]. The kCPQ can be very demanding if the datasets involved are large, since all the combinations of pairs of objects from the two datasets are candidates for the result.

For example, we can use two spatial datasets that represent the archaeological sites and popular beaches of Greece. A kCPQ (K=10) can discover the 10 closest pairs of archaeological sites and beaches (in increasing order of their distances). The result of this query can be used for planning tourist trips in Greece that combine travelers interest for history / civilization and leisure / enjoyment.

Parallel and distributed computing using shared-nothing clusters on large volumes of data has been very popular during last years. Hadoop MapReduce [7] is an open-source software framework for storing data and running applications on such clusters. MapReduce is file-intensive and computing nodes intercommunicate only through sorts and shuffles. Therefore, MapReduce is suitable mostly for non-iterative batch processing jobs.

Apache Spark [8] is another, more recent, open-source cluster-computing framework with an application programming interface based on Resilient Distributed Datasets (RDDs), read-only multisets of data items distributed over the cluster of machines [9]. It was developed to overcome limitations of the MapReduce paradigm. Through RDDs a form of distributed shared memory is provided and the implementation of iterative algorithms is facilitated.

Recently, the utilization of main memory in processing kCPQs on big datasets in centralized systems has been explored [10], [11]. In [1], considering ideas and methods presented in [10], [11] we presented a Spark based algorithm for computing kCPQs. Moreover, we presented an experimental analysis of the performance of this algorithm, based on large real-world datasets. In this paper, we extend [1] by developing three alternative algorithms. The first algorithm is a simple modification of the method of [1] that is based on single sampling for partitioning data. The other two algorithms are based on more elaborate partitioning techniques. Moreover, through an extensive experimental evaluation, we compare the performance of the three algorithms. Contrary to [1], where we performed experiments using 4 data nodes only, in this paper, we perform experiments using 4, as well as, 8 data nodes, to study the scalability of the presented techniques.

The rest of the paper is organized as follows. In Section II, we review related frameworks and work (extending the material presented in [1]); in Section III, we present Spark basics, we define the query that we study and present the algorithm of [1]; in Section IV we present two different data portioning schemes that lead to alternative algorithms for kCPQ; in Section V, we present experimentation set-up and the results of extensive experiments we performed for studying the efficiency of the presented methods. Finally, in the last section, we present our conclusions and our plans for future work.

II. Related Work

In [12] Spark is used to compute top-k similarity join in large multidimensional data. Data are being partitioned into buckets so that points that are close to each other are grouped into the same bucket, with high probability. Partitioning is made by means of locality-sensitive hashing and hamming distance computation between every two elements. The method uses Cartesian product, as provided by Spark, to create all possible buckets couples and computes local top-k over each node, then collects the results and combines them to the final solution. Divide & Conquer strategy and pruning is performed at local level.

In [13] Spark is used to perform several computational geometry operations such as Geometry Union, Convex Hull, Closest and Farthest pair, Spatial Range, Join and Aggregation on both small, medium and large data sets. Computation of Farthest Pair is performed by brute force and Closest Pair is reported difficult and time costly to be solved the same way, being efficient solely for small data sets. In order to overcome the problem, computation is being performed in two steps. The first step works per partition and computes the closest point in each subset, plus the points that may still be candidates (found by sorting the x-axis of the points per partition). Local computation is performed by a Divide and Conquer method that splits the local dataset recursively. The second step creates one single partition containing the closest points that were found in step one and all the candidates and once again performs the same Divide and Conquer approach. As authors report "there is a huge jump in execution time for the "large" dataset suggesting algorithm's effectiveness probably decreases as size increase" and "the increase in computational resources is offset by the communication cost in the latter case". The latter case refers to the "large" dataset of the experiments, which is about 100MB.

Extensions of Hadoop MapReduce supporting large-scale spatial data processing include Parallel-Secondo [14], Hadoop-GIS [15] and SpatialHadoop [16]. In [17], a general plane-sweep approach for processing kCPQs in SpatialHadoop and a more sophisticated version that first computes an upper bound of the distance of the K-th closest pair from sampled data points have been presented.

Extensions of Apache Spark supporting large-scale spatial data processing include the following:

- GeoSpark [18], an in-memory cluster computing framework for processing large-scale spatial data. The project is still under development. It uses Spark as its base layer and adds two more layers, the Spatial RDD (SRDD) Layer and Spatial Query Processing Layer, thus providing Spark with in-house spatial capabilities. The SRDD layer consists of three newly defined RDDs, PointRDD, RectangleRDD and PolygonRDD. SRDDs support geometrical operations, like Overlap and Minimum Bounding Rectangle. SRDDs are automatically partitioned by using the uniform grid technique, where the global grid file is splitted into a number of equal geographical size grid cells. Elements that intersect with two or more grid cells are being duplicated. GeoSpark provides spatial indexes like Quad-Tree and R-Tree on a per partition base. The Spatial Query Processing Layer includes spatial range query, spatial join query, spatial KNN query. GeoSpark relies heavily on the JTS topology suite and therefore conforms to the specifications published by the Open Geospatial Consortium. Experiments, reported by the paper, show that GeoSpark outperforms its Hadoop-based counterparts (e.g., SpatialHadoop). Mainly because caches the datasets in memory, a functionality that is natively built in the underlying Spark platform.
- SpatialSpark [19], that supports indexed spatial joins and range queries. Same as with GeoSpark it utilizes the JTS suite (written in Java). As reported by the authors, JTS seems to be faster than GEOS, a C/C++ port of a subset of JTS and selected functions. Authors report that in some cases of data intensive applications SpatialSpark performs worse on multiple computing nodes than on a single node, thus showing low scalability. This fact is attributed to pos-

sible bottlenecks due to communication overheads among computing nodes a factor that is related to the number of partitions, thus rising an interesting research question: "optimizing the number of partitions which represents the tradeoffs between the degrees of parallelisms (the higher the better) and the communication overheads (the lower the better)".

- LocationSpark [20], an ambitious project, built as a library on top of Spark. It requires no modifications to Spark and provides spatial query APIs on top of the standard operators. It provides Dynamic Spatial Query Execution and operations (Range, kNN, Insert, Delete, Update, Spatial-Join, kNN-Join, Spatio-Textual). The system builds two indexes, a global (grid, quadtree and a Spatial-Bloom Filter) and a local per-worker, user-decided index (grid, rtree, etc). Global index is constructed by sampling the data. Spatial indexes are aiming to tackle unbalanced data partitioning. Additionally, the system contains a query scheduler, aiming to tackle query skew. As reported in the paper, LocationSpark can outperform GeoSpark by one order of magnitude.
- Spatial In-Memory Big data Analytics (SIMBA) [21] that is perhaps the most mature framework. It extends the Spark SQL engine to support spatial queries and analytics through SQL and the DataFrame API. Simba partitions data in a manner that they are of proper and balanced size and gathers records that locate close to the same partition. It builds a local index per partition and a global index by aggregating information from local indexes. It supports range and kNN queries, kNN and distance joins. As being reported in the paper, Simba outperforms SpatialHadoop, HadoopGIS, SpatialSpark and GeoSpark by a few or more orders of magnitude. In the case of distance join queries, Simba runs about 1.2-1.5 times faster than its closest counterparts GeoSpark and SpatialSpark.

The kCPQ has been actively studied in centralized environments, when both [5], [6], [22]–[24], one [25], or none [10], [11] of the two spatial datasets are indexed. Two improvements of the classic plane-sweep algorithm and a new plane-sweep algorithm, called Reverse Run Plane Sweep, were proposed in [10] for processing kCPQs when the two datasets are not indexed and reside in main-memory. In [11], it is assumed that the (big) spatial datasets reside on secondary storage and are progressively transferred in main memory, by dividing them in strips, for processing utilizing the methods of [10].

In this paper, we utilize ideas presented in [10], [11] to develop an algorithm for processing kCPQs in Spark, by separating data in strips and utilizing a plane-sweep approach within each strip.

III. kCPQ in a Parallel and Distributed Context

Hadoop MapReduce processing is based on pairs of Map and Reduce phases. It is an excellent solution for one-step computations on massive datasets, but it not very efficient for problems that require multi-step computations. The output of each step is stored in the distributed file system, so that it can be used as input for the next, or one of the following steps. Replication and disk storage contribute to slowing down the overall computation. Apache Spark (or more simply, Spark) is an alternative to Hadoop MapReduce. Its not intended to replace Hadoop MapReduce, but to extend it and allow the development of solutions for different big data problems and requirements.

Spark, the distributed in-memory computation framework has reached a significant level of maturity, being already at version 2.2.0, which has been recently released. Spark is written in Scala, a relatively new functional programming language but it supports multiple programming languages, with special focus on Scala, Java, Python, and R. It allows a user application to cache data in memory, in a flexible manner that lets the application to decide what data should be cached and at what point in the processing flow. This is a major step forward from the classic Hadoop MapReduce procedure that uses disk I/O extensively. Spark uses an advanced job execution scheme based on creation of a directed acyclic graph (DAG) of stages. In contrast to MapReduce that in many cases constrains the programmer to split a complex algorithm into jobs executed sequentially, Spark uses a lazy evaluation scheme that allows previous knowledge of the full processing path, thus making it easier to optimize the execution. This functionality makes Spark ideal for iterative algorithms implementation. The Spark API relies on two important abstractions, namely SparkContext and Resilient Distributed Datasets (RDDs). An application interacts with Spark by means of these two abstractions. Data are being represented as RDDs in the Spark context, and are distributed among Workers of the cluster. Spark provides several methods defined in the RDD class or other subclasses of RDD. These methods operate on the RDD and finally on the underlying data. They are classified in two categories: transformations that create a new RDD and actions that return values to the Driver program. Transformations are lazy, which means that Spark does not perform any computation when they are called in an application. Actual computation is triggered by action methods. As already mentioned, this scheme provides Spark with the power to optimize RDD operations. Although Spark uses a shared-nothing architecture, it also supports the concept of shared variables that are being materialized as broadcasts and accumulators. By using broadcast variables, Spark sends data to each node, thus enabling all Workers to share a piece of information. Broadcast variables may be useful in cases of problems in the field of combinatorial optimization. For example, in NP-hard graph problems such as the maximum clique number, knowing a good lower bound of the maximum clique helps pruning the search space and speeding up the computation. A similar notion holds for the kCPQ. If we know a good upper bound for the k-th smaller distance and transmit it to all Workers, this will lead to discarding a large volume of computation among pairs of points that their distance is greater than the upper bound.

In the following, we present the basics for kCPQ processing in Spark. Let two datasets P and Q of spatial objects, a positive natural number K and a distance function between pairs of data objects formed from P and Q (members of the Cartesian Product of P and Q). The kCPQ discovers k pairs of data objects formed from P and Q that have the k smallest distances between them among all pairs of data objects that can be formed from P and Q.

Since distances between objects may not be unique, note that if multiple pairs of objects have the same k-th distance value between them, more than one sets of k different pairs of objects can form the result of this query. The presented algorithm can be easily tailored to report all such sets of pairs.

An important step, towards answering a query in a parallel and distributed environment, is proper partitioning of the datasets. Data partitioning improves the query performance in two ways [26]:

1) partitioning the data into smaller units enables processing of a query in parallel and
2) I/O can be significantly reduced by only scanning a few partitions that contain relevant data to answer the query.

In the case of the kCPQ, (b) is not applicable; in order to answer the query, we have to search pairs of points from the whole dataset. Therefore, in the case of kCPQ, the most severe obstacle, one has to face, in datasets partitioning, is data skewness. In most real-world cases, data is not uniformly distributed in a dataset. Using partitioning techniques such as uniform grid [27] very often leads to partitions that contain much more objects than others, a fact that in a parallel system may prevent proper load balancing and therefore delay the computation of the final result. There are many alternative strategies that can be found in the literature aiming to deal with the skewness problem. Most of them require the construction of a spatial data structure, which allows the queries about spatial relationships of objects to be answered. The simplest spatial data structure is the uniform grid, but, as already said, it very often leads to bad performance in the case of non-uniformly distributed data. This observation has led to more elaborate partition schemes, many of them being generalizations of binary search trees.

A quad-tree [28] is a non-uniform subdivision of area where a region is split into four quadrants by two axis-aligned dividing lines. The decomposition proceeds until a certain property is met, i.e., each quadrant contains less than a predefined number of points.

An R-tree [29] is a hierarchical data structure derived from the B-tree [30]. Data objects are represented by their enclosing MBRs, which are grouped into larger nodes hierarchically until the root node of the tree. Each leaf node contains the actual objects, and can store a certain, predefined number of objects.

In most cases within the context parallel and distributed frameworks, the creation of these hierarchical data structures is based on reading a random sample from the input file and using this sample to partition the whole space.

In order to efficiently compute the k closest pairs query in the Spark context, there are three main tasks, our algorithm has to deal with:

1) Find a good bound for the kCPQ and broadcast it to Workers. This will lead to good pruning criteria, on both Driver and Workers contexts.
2) Partition the data, by setting a proper indexing, and check all pairs of partitions so that all eligible pairs of points from P and Q will be considered, thus preventing any loss of the optimal solution.
3) Use a fast algorithm to compute kCPQ in the Workers context, collect the results and select the top k, having the smallest distance.

The method for answering the kCPQ, as presented in [1] consists of four steps that cover all the above mentioned tasks.

A. Lower Bound Computation

We initially compute an upper $bound$ for the k-th closest pair. We use the Spark-provided function $sample$ to create two RDDs containing samples from each one of the two datasets. We use a sample ratio of f = 0.001 on each dataset.

In order to obtain a good upper bound, we partition the sampled RDDs in a manner so that points with close x-axis values fall into the same partition.

Partitioning each of the two sampled datasets into n strips of unequal width is done by calculating the border (separation) x-axis points, separately for each sampled dataset. Both samples are collected to the Driver and their x-values are extracted and stored in two sorted arrays sP and sQ. The predefined number n and the sizes of the two arrays obtain the indices of each array that contain the separation x-points $PSep$ and $QSep$. Value $stepP$ is $sP.size/n$ and value $stepQ$ is $sQ.size/n$. The two arrays $PSep$ and $QSep$ are merged into a sorted array $PQSep$ = $PSep$ ++ $QSep$ that contains the separation x-axis points applicable on both sampled RDDs. This array is passed to Spark and all points in both the sampled RDDs are being assigned the proper keys.

A *join* is performed between the two keyed RDDs, creating an RDD of type $(Int, (Point, Point))$ that is mapped to an RDD$[Double, (Point, Point)]$ where the Double presents the distance (Euclidean in our case) of every pair of points in the joined RDD. By using the $takeOrdered$ function of Spark, we select the k pairs with smaller distance. The k-th distance is our upper $bound$. This means that in the following steps we do not need to seek for pairs that have their distance greater than this $bound$ and consequently we do not need to examine pairs that have their x-axis (y-axis can also be used) distance greater than $bound$.

B. Datasets Partitioning

After the bound computation from samples, both datasets are separately divided into a, user defined, number of n strips. Partitioning each of the two datasets into strips of unequal width is done by calculating the border (separation)

points from samples, separately for each dataset by the same procedure shown in the previous step.

Sampling with a ratio m is used here. In [1] the sample size was set to $dataset.size/n$ where n is the number of desired partitions. Each point from the sample is mapped to its x-axis coordinate and all 1D points are collected to the Master node as an Array $A[m]$. Sorting is performed on the array and the number of predefined strips n determines the $step = m/n$ on the indices of the array that contain the separation x-points. The actual x-values, depicted in Fig. 1, are $X1 = A[step], X2 = A[2 * step], \ldots, Xn = A[(n-1) * step]$.

Each subinterval contains approximately m/n points and therefore the projection of this upon the whole dataset creates strips with approx. equal size of points. The separation points, shown in gray in Fig. 1, are being used on the whole dataset, to split it into n strips. The partitioning is being done a function $xPartitionSpace$, which assembles the envelopes of the strips

$$Env[minX : X1, minY : maxY],$$
$$Env[X1, X2, minY : maxY], \ldots,$$
$$Env[Xn - 1 : maxX, minY : maxY]$$

The function returns an array of type (Int, Env), by assigning consecutive integers to each partition presented as Envelope. Actual partitioning is done by passing a function to Spark with parameter the array of Envelopes and each Worker scans the dataset and assigns the proper key to each point.

C. Classification of Strips

The third step of the algorithm presented in [1] uses $bound$, the distance of the k-th closest pair, which was computed from the sample as described in step 1, to classify all pairs of strips from the two datasets into two categories, *Eligible* and *non-Eligible*.

This is accomplished by first finding the relative position between each pair of strips. The criterion used to derive the relative position is based on the relation of the minimum and maximum value of the x-coordinates of the strips. In Fig. 2 all possible cases are being depicted.

In the case of overlapping pairs, as it happens with W and B, the expression $(W.x1 < B.x2 \;\&\&\; W.x2 > B.x1)$ evaluates to *true*.

If it evaluates to *false*, then there are two cases, either strip is on the left of W (strip A), and $W.x1 > A.x2$, or strip is on the right of W (strip C) and $W.x2 < C.x1$.

Eligible and *non-Eligible* categories are defined as follows:

Fig. 1. Selection of splitting points in [1]

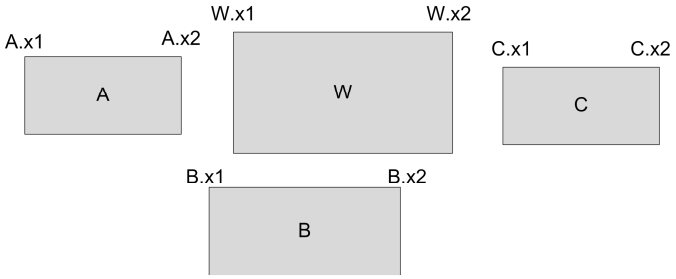

Fig. 2. Relative position of strips

1) *Eligible* pairs consist of all pairs of strips containing points that may contribute to the final query answer. The eligible pairs may be:
   a) Pairs that overlap. As seen in Fig. 3, strip P1 from P overlaps with strips Q1 and Q2 from Q.
   b) Pairs that do not overlap, but have their x-axis distance smaller than $bound$. In Fig. 3 the x-distance between P1 and Q3 is $d1 < bound$.
2) *non-Eligible* pairs consist of all pairs of strips that do not overlap and have their x-axis distance greater than $bound$. The contained points cannot contribute to the final query answer. Such a pair is P1 and Q4, two strips that have their x-distance $d2 > bound$, as shown in Fig. 3. The same holds for every consecutive Q-strip after Q4.

In the case of non-overlapping but yet eligible pairs (case 1-a, above), not all points from both strips need to be considered in the forthcoming step, since pruning can be performed to reduce both strips to these points that their x-axis distance from each other is smaller than $bound$.

For example, in the case of pair P1, Q3, we use the $filter$ function of Spark to reduce Q3 to these points (Q3.x, Q3.y) such that $Q3.x - P1.max < bound$ and also reduce P1 to these points (P1.x, P1.y) such that $Q3.xmin - P1.x < bound$.

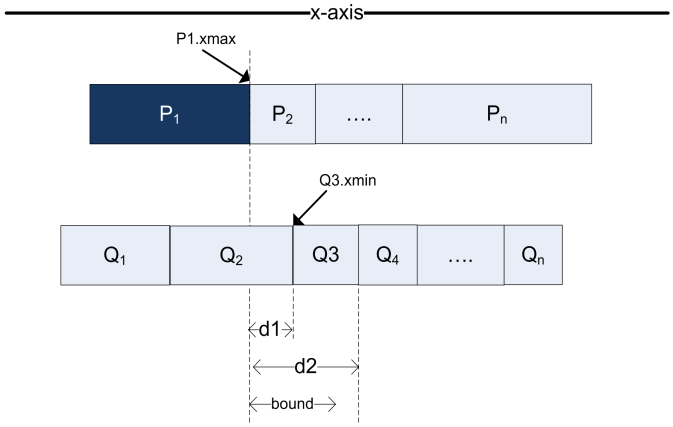

Fig. 3. Eligible strips and filtering of non overlapping strips

D. kCPQ computation

Having located all the eligible pairs, we create two new RDDs, with all possible pairs of strips containing points that may contribute to the answer of the query. This is done by duplicating, as needed, the points from strips of each dataset that have to be accounted with points from strips from the other dataset (a union) and assigning proper keys.

In the final step, a Plane-sweep algorithm is applied within each eligible (and filtered) pair of strips from P and Q for calculating k Closest Pairs and storing the result in a, separately for each partition, maximum binary heap (max-Heap). Previously, the $bound$ (computed in step 1) has been broadcasted to all workers to use it as stop condition for the plane sweep algorithm. Taking the first (sorted on distance) k tuples with the smaller distances, yields the final (and exact) solution.

IV. Binary Space Partitioning for the kCPQ

As already described, the method of [1] uses a lightweight partitioning scheme that splits the datasets into strips. In the current work we are maintaining the fundamental concept of partitioning into strips, but implement two additional partitioning schemes, influenced by the Quad-tree and R-tree principles. Since the partitioning derives from Quad-tree and R-tree, we call them Q-split and R-split partitionings, respectively.

A. Outline of the partitioning procedure

As in the original method, partitioning of datasets is performed along an axis, which in our case is the x-axis. Both Q-split and R-split partitions use a Binary Search Tree (BST) to store the splitting points. A *class BST* extends Scala collection *mutable.Map* and is used to implement the Binary Search Tree. Each Node of the BST is defined as a Scala class of type *class Node(key, value, left: Node, right: Node)*.

Parameter key is of type Double and is the actual splitting x-coordinate. Parameter $value$, in our case is of type (Int, Double) where the Int in $value$ is showing the level of decomposition and the Double is the splitting x-point. Parameter $value$ can store any kind of information and we intend to use it in further experimentation, for example, one can store sizes of every child area and use it to make decisions regarding the computation. The BST class is equipped with a function $compare$ that is used to compare the keys of the points to be added.

Every new node is being added by a function + that recursively scans the BST, locates its correct position and adds it to the tree.

By sampling the dataset with ratio f, we map the points to their x-coordinates and collect the results to the Driver program, in an array DS. The "middle" point selection depends on the criterion used to partition the array and afterwards the dataset.

In the case of R-split, $middle$ is selected as the point that leaves equal number of points on the two sub-intervals, while in the case of Q-split $middle$ is the point that splits the interval into two sub-intervals with the same x-axis width. This means that in the first case $middle$ is an actual x-point from the dataset, while in the latter it may be not since it is computed as the median of each interval.

Each dataset is partitioned separately, as happens in [1]. In both partitioning schemes, we set two parameters $PCapacity$ and $QCapacity$, which represent the maximum number of points that a node can store. In contrast to [1], no additional sampling is performed, as partitioning is done by using the samples taken for upper bound computation.

B. R-split

We perform Quicksort on array DS (the full sample) and locate the first splitting point of the array, named $middle$. In the case of R-split, initial value of $middle$ is set as the quotient $DS.length/2$. The value of $capacity$, idx (an integer counting the level of decomposition), TR (an empty BST), DS (the array with x-points from sample) and the splitting point $middle$ are passed to a function $partitionEqualSize$ that uses recursion to create and return the BST with the splitting points. The pseudo code snippet in Fig. 4 outlines the procedure.

C. Q-split

The initial value of $middle$ is computed as the median of the maximum and minimum x-values of each dataset. For example, in the case of dataset P, middle is the quotient $(P.maxX + P.minX)/2$. Function $partitionEqualwidth$ is similar to $partitionEqualSize$, with two differences. Line 12 is replaced with $ml = (FPLeft.maxX + FPLeft.minX)/2$ and Line 16 is $mr = (FPRight.maxX + FPRight.minX)/2$

```
 1: function PARTITIONEQUALSIZE(capacity, idx, TR, DS, middle): BST
 2:     function PS(capacity,idx, TR, DS, middle): Int
 3:         id = idx + 1
 4:         TR += middle -> (id, middle)
 5:         FPLeft = DS.filter(x => x < middle)
 6:         FPRight = DS.filter(x => x >= middle)
 7:         FPLeftSize = FPLeft.length
 8:         FPRightSize = FPRight.length
 9:         FL = FPLeftSize / f                    ▷ f is the sampling ratio
10:         FR = FPRightSize / f
11:         if (FL > capacity) then
12:             ml = FPLeft(FPLeftSize / 2)
13:             id = ps(capacity,id, TR, FPLeft, ml)
14:         end if
15:         if (FR > capacity) then
16:             mr = FPRight(FPRightSize / 2)
17:             id = ps(capacity, id, TR, FPRight, mr)
18:         end if
19:         return id
20:     end function
21:     ps(capacity, idx, TR, DS, middle)
22:     return TR
23: end function
```

Fig. 4. R-SPLIT outline.

In both R-split and Q-split, the splitting points are retrieved from the BST by using an inorder traversal, being utilized by means of a properly designed iterator.

V. Experimental Evaluation

To evaluate the performance of our methods, we used three large real 2d datasets from OpenStreetMap [16]: WATER resources consisting of 5,836,360 line segments, PARKS (or green areas) consisting of 11,504,035 polygons and BUILDINGS of the world consisting of 114,736,611 polygons. To create sets of points, we used the centers of the Minimum Bounding Rectangles (MBRs) of the line-segments from WATER and the centroids of polygons from PARK and BUILDINGS.

All experiments were conducted on a cluster of 9 nodes. Each node has 4 vCPUs running at 2.1GHz, with a total of 16GB of main memory per node, running Ubuntu Linux 16.04 operating system. Spark 2.1.1 running on Hadoop 2.7.2 Distributed File System (HDFS) was used as our parallel computing system. The block size of HDFS was 128 MB. Of the 9 computing nodes, one was running the NameNodes for Hadoop and Master for Spark, while the remaining eight (8 nodes x 4 vCPUs = 32 vCPUs) were used as HDFS DataNodes and Spark Worker nodes. Java openjdk ver. 1.8.0 and Scala code runner ver. 2.11 were used.

In all experiments, the data sets P and Q are in the form of text files formatted in columns with a separator (in our case a tab), one point per line (x, y coordinates). We also make the assumption that the two data files have already been stored in the HDFS, at an earlier phase without being subject to any kind of processing (e.g., sorting, indexing, and so forth). The datasets are read by using the $textFile$ function of Spark. Each dataset is presented as an RDD[$Point$], where $Point$ is of type Tuple2[Double, Double]. In all experiments the number of closest pairs is set to k = 10.

A. Speedup of method in [1]

First we measure the total computing time for 4 and 8 computing nodes in the case of BUILDINGS x PARKS (Fig. 5) and PARKS x WATER (Fig. 6) of the method presented in [1], varying the number of the preset partitions.

We measured total execution time (i.e., response time) in seconds (sec) that expresses the overall CPU, I/O and communication time needed for the execution of each query.

Let $T4$ be the execution time for four nodes and $T8$ be the execution time for eight nodes. The speedup is defined by $T4/T8$. We observe that execution time decreases more rapidly in the cases of larger number of partitions. This is expected since a larger number of partitions leads to more Spark jobs being more efficiently managed when a larger number of computing nodes is available. In the case of BUILDINGS x PARKS, best speedup (1.7) is measured at 32 partitions. Best response time is measured with 16 partitions, where the speedup is about 1.6. In the case of PARKS x WATER, best speedup (1.84) is measured at 64 partitions. Best response time comes when number of partitions is set to 32, and the speedup is 1.3

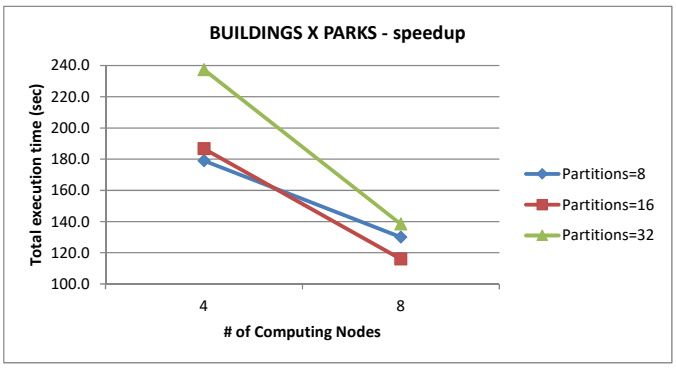

Fig. 5. kCPQ(BUILDINGS x PARKS), Nodes =4, 8

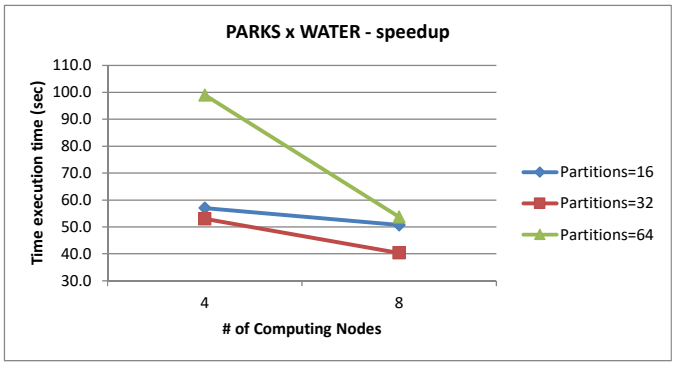

Fig. 6. kCPQ(BUILDINGS x PARKS), Nodes =4, 8

B. Original method in [1] vs improved (single sampling)

The second experiment deals with the improvement we have made to the algorithm in [1] and mentioned beforehand. By refactoring the code, we have removed the second sampling that is used by the original method in order to partition the datasets. Instead, we use the sample already taken for the upper bound computation. The results are being presented in Fig. 7 and Fig. 8.

It was observed that a sample ratio of f = 0.001 is adequate to efficiently partition the datasets. In general, the system runs faster, mostly in the case of larger datasets where a relatively small fixed number of partitions are used. This is reasonable, since in the original method the sample size is computed as the quotient of dataset size and number of partitions; therefore as partitions number decreases it results in an increase of the sample size and therefore an increase in upper bound computation time.

We use this strategy (a single sampling per dataset) in all following experiments, and use only one very small sample, with ratio 0.001 for computing both upper bound and partitioning x-axis points.

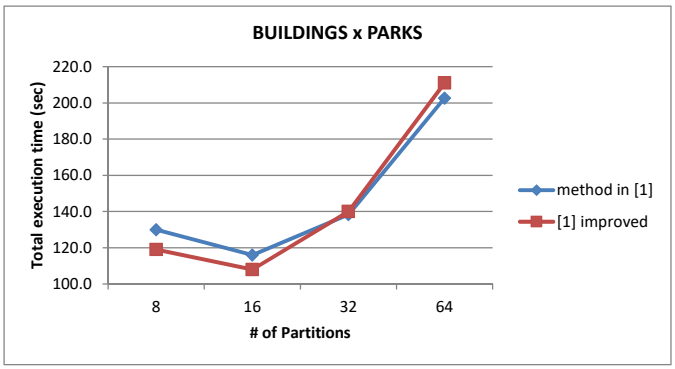

Fig. 7. BUILDINGS x PARKS. Method [1] vs method [1] improved

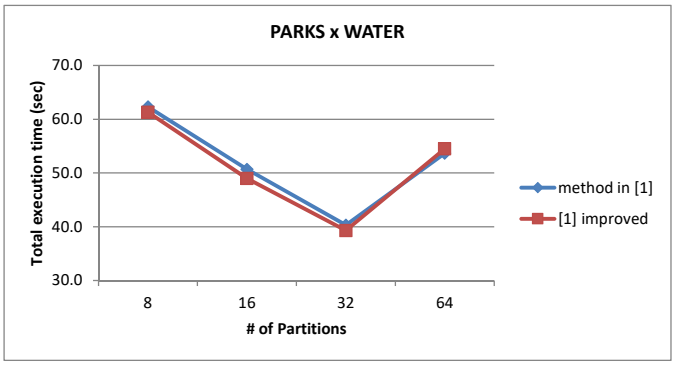

Fig. 8. PARKS x WATER. Method [1] vs method [1] improved

C. R-split and Q-split performance

In the third experiment we test the performance of the method in the case of both R-split and Q-split. We have run several experiments with various combinations of capacity for each dataset. Table I presents the results of the experiment regarding R-split for BUILDINGS x PARKS and PARKS x WATER datasets. As it can be observed, the new partitioning scheme provides much more freedom in fine tuning the system, since we now don't need to preset the number of partitions, but rather set capacities and have the system compute the number of partitions that meet the settings.

In Fig. 9 and Fig. 10 we compare the results measured for the improved version of [1], as described above, with the best obtained by using the R-split, subject to the same number of partitions between the two methods. It can be deduced that an R-split achieves better running times compared to both the original and the improved version of method presented in [1] especially when dealing with larger datasets.

Table II presents the results of the experiment regarding Q-split for BUILDINGS x PARKS and PARKS x WATER datasets. In contrast to R-split, using Q-split leads to a, more or less, worse performance in almost every case when compared to both R-split and the improved method in [1]. Another observation is that Q-split leads to a number of derived partitions that varies more widely than in the case of R-split, where the number of partitions is mostly constant.

TABLE I. R-SPLIT RESULTS FOR THE KCPQ

Capacity (millions of points)		Derived partitions #		Eligible	Time
BUILDINGS	PARKS	BUILDINGS	PARKS	pairs #	(sec)
23	2.3	8	8	15	111.3
14.5	1.5	8	8	15	104
14.5	1.4	8	16	23	126
14	1.4	16	16	31	104.3
12	1.4	16	16	31	101.3
10	1.4	16	16	31	96
10	0.75	16	16	31	98
10	0.7	16	32	47	138
8	1.4	16	16	31	99
8	0.75	16	16	31	95.7
8	0.7	16	32	47	138.7
6	1.4	32	16	47	104.3
6	0.7	32	32	63-64	125
4	1.4	32	16	47	109
2	1.4	64	16	79	131.3
2	0.35	64	64	127-128	199.3
Capacity (millions of points)		**Derived partitions #**		**Eligible**	**Time**
BUILDINGS	PARKS	BUILDINGS	PARKS	pairs #	(sec)
1.4	0.7	16	16	31	49.7
1.4	0.35	16	32	47	39
2.8	1.4	8	8	15	61
0.7	0.35	32	32	63	40.3
0.7	0.7	32	16	47	40.7
0.35	0.175	64	64	127-128	53

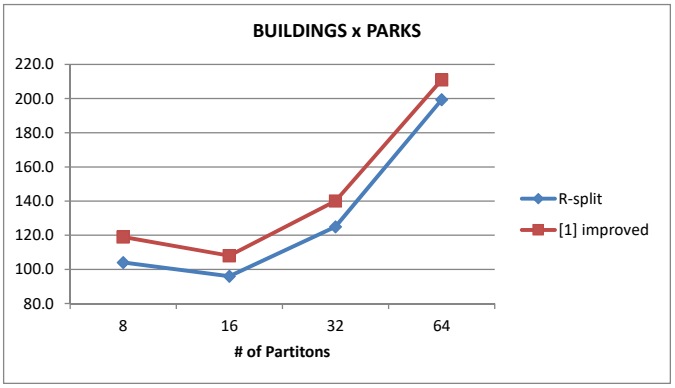

Fig. 9. BUILDINGS x PARKS. R-split vs method [1] improved

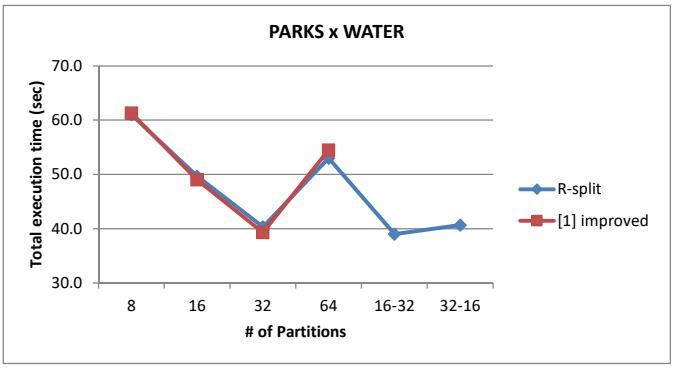

Fig. 10. PARKS x Water. R-split vs method [1] improved

TABLE II. Q-SPLIT RESULTS FOR THE KCPQ

Capacity (millions of points)		Derived partitions #		Eligible	Time
BUILDINGS	PARKS	BUILDINGS	PARKS	pairs #	(sec)
23	2.3	8	8	17	141.7
14.5	1.5	15	11-14	25-28	122.3
14.5	1.4	14-15	12-15	25-29	121.3
14	1.4	15-16	14-15	29-30	118.3
12	1.4	16	15-16	30-31	120
10	1.4	18-19	14-15	32-33	116.7
10	0.75	18-19	22-24	41-42	132.3
10	0.7	19	24	42	135.7
8	1.4	22	15	36	118.3
8	0.75	22-23	22-24	44-45	127.3
8	0.7	22-23	24-25	45-47	136
6	1.4	28	14	41	116
6	0.7	28	23-27	50-54	127.7
4	1.4	46	14-15	59-60	130.7
2	1.4	84-86	14-15	98-99	205.3
2	0.35	86-87	47-48	134	230
1.4	0.7	14-15	15-16	28-30	35
1.4	0.35	14-15	26-27	40-41	39.3
2.8	1.4	8	8-9	16-15-15	59.7
0.7	0.35	24-25	27-28	50-52	41
0.7	0.7	24-25	16-17	39-41	33.3
0.35	0.175	48-49	55-56	102-104	49

D. R-split and Q-split performance

The fourth experiment is aiming to check the quality of R-split and Q-split partitions. As it can be seen, R-split partitioning results in strips with uniformly allocated number of points (Fig. 11 and Fig. 12). Furthermore, the selection of a small sample with ratio 0.001 is proved to be sufficient for this purpose.

On the other hand, a Q-split partition results in strips with unequal number of points (Fig. 13 and Fig. 14).

E. Dataset points replication

The fifth experiment is aiming to enlighten the differences measured in execution times regarding different $PCapacity$ and $QCapacity$ settings, for both R-split and Q-split partitions.

As already mentioned, after locating all the eligible pairs of strips, two RDDs are derived containing all possible pairs of strips with points that may contribute to the answer of the query. These RDDs are created as a union of properly keyed points from strips of each dataset that have to be accounted with points from strips from the other dataset.

In the (usual) case a strip from P has to be combined to more than one strip from Q, then P is being duplicated (and filtered in the case of non overlapping strips) as needed and proper keys are being assigned. This means that the derived RDDs upon which the actual computation is being performed contain replicated points from both datasets.

In Fig. 15 and Fig. 16 we present the derived datasets in two different cases of partitioning by using R-split and Q-split

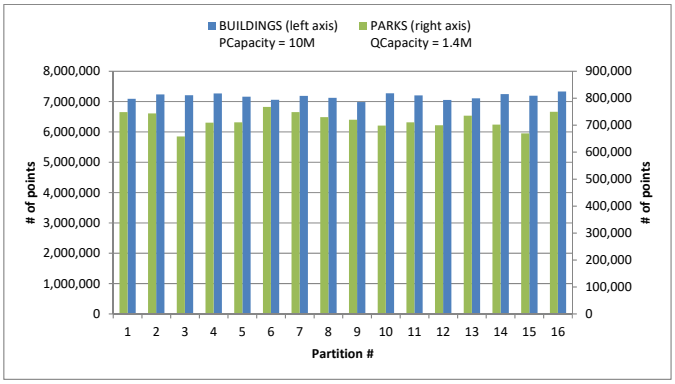

Fig. 11. R-split: # of points per partition. Capacity = (10M, 1.4M)

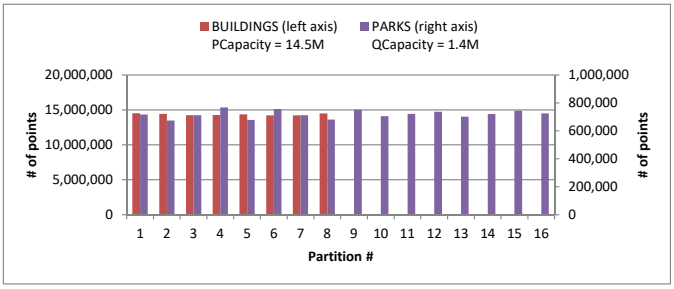

Fig. 12. R-split: # of points per partition. Capacity = (14.5M, 1.4M)

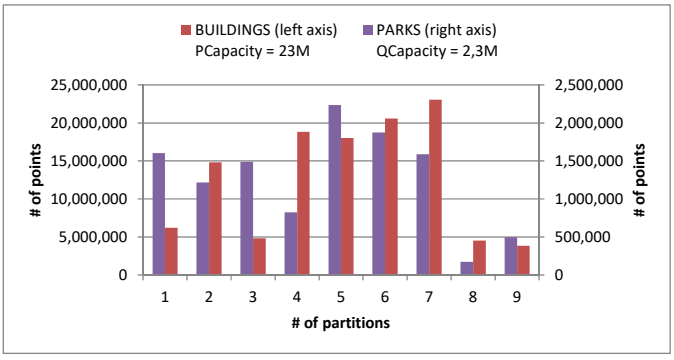

Fig. 13. Q-split: # of points per partition. Capacity = (23M, 2.3M)

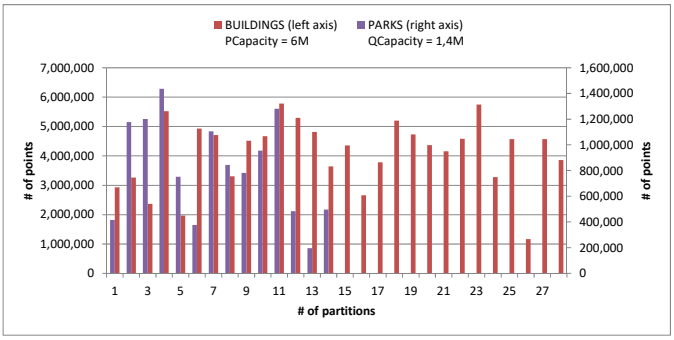

Fig. 14. Q-split: # of points per partition. Capacity = (6M, 1.4M)

respectively. In both figures, the first pair of columns presents the sizes of the original datasets and the rest two pairs present the sizes of the derived datasets (that are actually used for kCPQ computation) for two pairs of capacity settings. Using larger capacities leads to a smaller number of partitions (as shown in Table I and Table II) but this leads to an increased number of eligible points, a fact that influences the total execution time (also shown in figures).

F. Testing with larger datasets

In order to test our method on even larger pairs of datasets, we used CLUS_LAKES[1], a new big quasi-real dataset derived from a real one. To create this dataset, for each point of LAKES, p, 15 new points gathered around p (i.e., the center of the cluster) are generated according to a Gaussian distribution with mean = 0.0 and standard deviation = 0.2. The dataset CLUS_LAKES contains around 126M of points. We computed the kCPQ on the pair P = BUILDINGS and Q = CLUS_LAKES for several combinations of PCapacity and QCapacity and the total execution time (averaged) is shown in Table III (R-split has been used for the partitioning of the whole datasets).

VI. Conclusion and Future Plans

In this paper, extending [1], we have presented a method for the kCPQ computation in Spark. This method splits data into strips and computes closest pairs by plane sweep within each strip. Since proper partitioning is of great essence, especially in the context of parallel and distributed environments, we have particularized this method by presenting and implementing three different partitioning schemes that split the datasets into strips.

By conducting experiments on large real datasets we have explored the performance of our method and the performance of the partitioning schemes. Splitting into strips by means of Binary Space Partitioning techniques is proven to provide flexibility in tuning the system, thus resulting to faster execution time. R-split (divide datasets into parts with equal sizes) was shown to work better than Q-split (divide datasets into parts of equal width).

In a very recently published paper [31] we have presented SliceNBound, an algorithm for the kCPQ and Distance Join Query (DJQ), influenced by the ideas presented in the current paper. Simba [21] does not support KCPQs, but does support DJQs, so a comparison had been performed between the methods [31] and [21] on DJQ. In the future, we plan to compare the methods for kCPQ presented in current paper with kCPQ implemented in Simba and other spatial oriented, Spark based, platforms. We also plan to further elaborate this method and investigate partitioning schemes for Spark to reduce the need for examining combinations of data that reside in different strips and also reduce the network communication traffic. Furthermore, we plan to research for a faster and stricter upper bound computation, since we have observed that this bound strongly influences the total running time of the query.

[1] Kindly provided by Antonio Corral and Francisco García-García, University of Almeria, Spain.

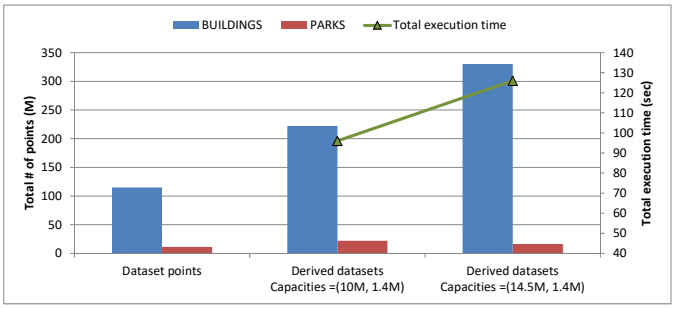

Fig. 15. R-split: Replicated points and execution time

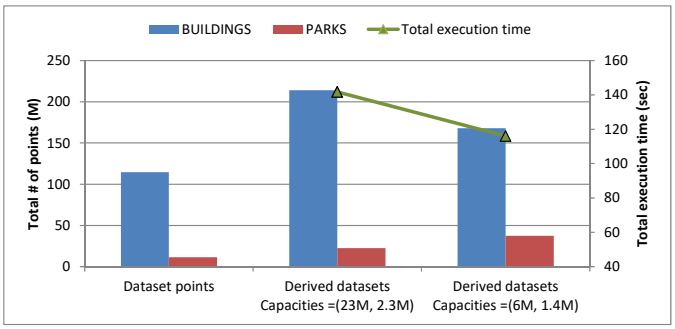

Fig. 16. Q-split: Replicated points and execution time

TABLE III. R-SPLIT RESULTS FOR THE KCPQ(BUILDINGS X CLUS_LAKES)

R-split					
Capacity (millions of points)		**Derived partitions #**		**Eligible pairs #**	**Time (sec)**
BUILD-INGS	CLUS_ LAKES	BUILD-INGS	CLUS_ LAKES		
2	2	64	64	127-129	433.7
6	6	32	32	63	316.0
6	8	32	16	47	287.7
8	6	16	32	48	548.7
8	8	16	16	31	385.0

References

[1] G. Mavrommatis, P. Moutafis, and M. Vassilakopoulos, "Closest-Pairs Query Processing in Apache Spark," in *Proceedings of the 8th International Conference on Cloud Computing, GRIDs and Virtualization (CLOUD COMPUTING 2017), Athens, Greece, February 19-23*, 2017, pp. 26–31, ISBN: 978-1-61208-529-6, ISSN: 2308-4294.

[2] S. Shekhar and H. Xiong, Eds., *Encyclopedia of GIS*. Springer, 2008.

[3] P. Rigaux, M. Scholl, and A. Voisard, *Spatial databases - with applications to GIS*. Elsevier, 2002.

[4] A. Corral and M. Vassilakopoulos, "Query processing in spatial databases," in *Encyclopedia of Database Technologies and Applications*. Idea Group, 2005, pp. 511–516.

[5] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Closest pair queries in spatial databases," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, USA, May 16-18*, 2000, pp. 189–200.

[6] ——, "Algorithms for processing k-closest-pair queries in spatial databases," *Data Knowl. Eng.*, vol. 49, no. 1, pp. 67–104, 2004.

[7] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, Boston, MA, USA, June 22-25*, 2010, pp. 10–10.

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, San Jose, CA, USA, April 25-27*, 2012, pp. 2–2.

[10] G. Roumelis, M. Vassilakopoulos, A. Corral, and Y. Manolopoulos, "A new plane-sweep algorithm for the k-closest-pairs query," in *SOFSEM 2014: Theory and Practice of Computer Science - Proceedings of the 40th International Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 26-29*, 2014, pp. 478–490.

[11] G. Roumelis, A. Corral, M. Vassilakopoulos, and Y. Manolopoulos, "New plane-sweep algorithms for distance-based join queries in spatial databases," *GeoInformatica*, vol. 20, no. 4, pp. 571–628, 2016.

[12] D. Chen, C. Shen, J. Feng, and J. Le, "An efficient parallel top-k similarity join for massive multidimensional data using spark," *International Journal of Database Theory and Application*, vol. 8, no. 3, pp. 57–68, 2015.

[13] D. N. Rao and D. S. Rao, "Computational geometry leveraged by apache spark," *Journal of Innovation in Electronics and Communication Engineering*, vol. 5, no. 2, pp. 15–31, 2015, ISSN: 2249-9946, Online ISSN: 2455-3514.

[14] J. Lu and R. H. Güting, "Parallel secondo: Boosting database engines with hadoop," in *Proceedings of the 18th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2012, Singapore, December 17-19*, 2012, pp. 738–743.

[15] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz, "Hadoop-gis: A high performance spatial data warehousing system over mapreduce," *PVLDB*, vol. 6, no. 11, pp. 1009–1020, 2013.

[16] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *Proceedings of the 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17*, 2015, pp. 1352–1363.

[17] F. García-García, A. Corral, L. Iribarne, M. Vassilakopoulos, and Y. Manolopoulos, "Enhancing spatialhadoop with closest pair queries," in *Advances in Databases and Information Systems - Proceedings of the 20th East European Conference, ADBIS 2016, Prague, Czech Republic, August 28-31*, 2016, pp. 212–225.

[18] J. Yu, J. Wu, and M. Sarwat, "Geospark: a cluster computing framework for processing large-scale spatial data," in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6*, 2015, pp. 70:1–70:4.

[19] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *Proceedings of the 31st IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2015, Seoul, South Korea, April 13-17*, 2015, pp. 34–41.

[20] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: A distributed in-memory data management system for big spatial data," *PVLDB*, vol. 9, no. 13, pp. 1565–1568, 2016.

[21] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26-July 01*, 2016, pp. 1071–1085.

[22] G. R. Hjaltason and H. Samet, "Incremental distance join algorithms for spatial databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA, June 2-4*, 1998, pp. 237–248.

[23] H. Shin, B. Moon, and S. Lee, "Adaptive and incremental processing for distance join queries," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 6, pp. 1561–1578, 2003.

[24] C. Yang and K. Lin, "An index structure for improving closest pairs and related join queries in spatial databases," in *Proceedings of the International Database Engineering & Applications Symposium, IDEAS'02, Edmonton, Canada, July 17-19*, 2002, pp. 140–149.

[25] G. Gutierrez and P. Sáez, "The k closest pairs in spatial databases - when only one set is indexed," *GeoInformatica*, vol. 17, no. 4, pp. 543–565, 2013.

[26] A. Aji, H. Vo, and F. Wang, "Effective spatial data partitioning for scalable query processing," *CoRR*, vol. abs/1509.00910, 2015.

[27] A. Eldawy, L. Alarabi, and M. F. Mokbel, "Spatial partitioning techniques in spatial hadoop," *PVLDB*, vol. 8, no. 12, pp. 1602–1605, 2015.

[28] H. Samet, C. A. Shatter, R. C. Nelson, Y. Huang, K. Fujimura, and A. Rosenteld, "Recent developments in linear quadtree-based geographic information systems," *Image Vision Comput.*, vol. 5, no. 3, pp. 187–197, 1987.

[29] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21*, 1984, pp. 47–57.

[30] S. T. Leutenegger, J. M. Edgington, and M. A. López, "STR: A simple and efficient algorithm for r-tree packing," in *Proceedings of the Thirteenth International Conference on Data Engineering, Birmingham, U.K., April 7-11*, 1997, pp. 497–506.

[31] G. Mavrommatis, P. Moutafis, M. Vassilakopoulos, F. García-García, and A. Corral, "Slicenbound: Solving closest pairs and distance join queries in apache spark," in *Advances in Databases and Information Systems - Proceedings of the 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27*, 2017, pp. 199–213, ISBN: 978-3-319-66916-8, ISSN: 0302-9743.