Closest-Pairs Query Processing in Apache Spark

George Mavrommatis, Panagiotis Moutafis, and Michael Vassilakopoulos
Data Structuring & Engineering Lab
Dept. of Electrical & Computer Eng.
University of Thessaly
Volos, Greece
e-mail: {gmav, pmoutafis, mvasilako}@uth.gr

***Abstract*— Processing of spatial queries when the datasets involved are big can be accomplished efficiently in a parallel and distributed environment. The (K) Closest-Pair(s) Query, KCPQ, is a common query in many real-life applications involving geographical, or, in general, spatial data. It consists in finding the (K) closest pair(s) of objects between two spatial datasets. Although, processing of this query has been studied extensively for centralized environments, few solutions have appeared for parallel and distributed frameworks. Apache Spark is such a framework that has several advantages compared to other popular ones, like Hadoop MapReduce. In this work, we present an algorithm for processing the KCPQ in Apache Spark and experimentally study its efficiency and scalability, using big real-world datasets.**

Keywords-Closest-Pairs Query; Spatial Query Processing; Apache Spark.

I. INTRODUCTION

Geographic information systems (GIS) [1] 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.).

The term *Big Data* refers to unprecedented volumes of data. Such data appear in numerous modern applications, like applications based on sensor networks, commercial transactions, social media, web searches, etc.

Spatial databases [2] 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 big, or if the volume of the combinations of data objects that need to be examined for answering such a query are big.

Some typical spatial queries are: the point query, range query, spatial join, and nearest neighbor query [3]. 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 (K) Closest-Pair(s) Query, 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) [4][5]. The KCPQ can be very demanding if the datasets involved are big, 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 traveler's interest for history / civilization and leisure / enjoyment.

Parallel and distributed computing using shared-nothing clusters on big data has been very popular during last years. Hadoop MapReduce [6] 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 [7] 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 [8]. 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 [9][10]. In this paper, considering ideas and methods presented in [9][10] we present a Spark based algorithm for computing KCPQs. Moreover, we present an experimental analysis of the performance of this algorithm, based on big real-world datasets.

More specifically, in Section II, we review related frameworks and work; in Section III, we present Spark basics, we define the query that we study and present our algorithm; in Section IV, we present experimentation

 ISBN: 978-1-61208-529-6

settings and the results of experiments we performed for studying the efficiency of the proposed method. Finally in the last section, we present our conclusions and our plans for future work.

II. RELATED WORK

Extensions of Hadoop MapReduce supporting large-scale spatial data processing include Parallel-Secondo [11], Hadoop-GIS [12] and SpatialHadoop [13]. In [14], 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

- SpatialSpark [15], that has been used for spatial join algorithms based on point-in-polygon test and on point-to-polyline distance,
- GeoSpark [16], that supports spatial range, join query and K nearest neighbors queries,
- LocationSpark [17], that offers several spatial query operators, including range search, K nearest neighbors, spatio-textual operations, spatial join and K nearest- neighbors join, and
- Spatial In-Memory Big data Analytics (SIMBA) [18] that supports box and circle range queries, K nearest neighbors, distance joins and K nearest-neighbors joins.

The KCPQ has been actively studied in centralized environments, when both [19][20][21][22][23], one [24], or none [9][10] 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 [9] for processing KCPQs when the two datasets are not indexed and reside in main-memory. In [10], 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 [9].

To the best of our knowledge, the only work about KCPQs in a parallel and distributed framework is [14]. In this paper, we utilize ideas presented in [9][10] to develop an algorithm for processing KCPQs in Spark, by separating data in strips and utilizing a plane-sweep approach within each strip.

III. CLOSEST-PAIR QUERIES IN SPARK

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. It's not intended to replace Hadoop MapReduce, but to extend it and allow the development of solutions for different big data problems and requirements.

Spark was written in the Scala Programming Language. Programmers usually write Spark applications in Java, Scala, or Python, with Scala being the most popular choice. In addition to Map and Reduce operations, it supports SQL queries, streaming data, machine learning and graph data processing. These capabilities can be combined in a data pipeline. With Apache Spark, programmers can combine data pipelines in a directed acyclic graph (DAG). The DAG execution model can be seen as a generalization of the MapReduce model. Moreover, Apache Spark supports in-memory data sharing across DAGs. Spark can run on top of an existing Hadoop Distributed File System (HDFS) infrastructure. Spark also supports lazy evaluation and holds intermediate results in memory. When data cannot fit in memory, disk storage is utilized. In fact, part of a data set can reside in memory and another part on secondary storage. The RDD is the fundamental data structure of Spark. An RDD can be resembled to a database table. It is a read-only collection of objects, partitioned in the cluster of machines.

In the following, we present our algorithm 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.

Our algorithm, for 2-dimensional space (for the ease of exposition), consists of the following steps:

- Samples $P' \subset P$ and $Q' \subset Q$ are taken from both datasets P and Q. Spark function sample() was used for sampling the two datasets. sample() takes a parameter, *fraction*, denoting the expected size of the sample as a fraction of the dataset in question.
- Proper keys are set, a join between P' and Q' is performed and the K closest pairs (CP) among all joined pairs are computed. Function join() is also provided by the Spark API.
- Let *Bound* be the K-th smaller distance as computed previously. This is our pruning factor.
- Both datasets are divided into n strips [25] corresponding to ascending intervals along one of the dimensions (x axis dimension is assumed in the following, w.l.o.g) (Fig. 1). Partitioning of each of the two datasets into strips of unequal width was done by sampling, calculating the border points from samples and applying the partition to the whole dataset.

 ISBN: 978-1-61208-529-6

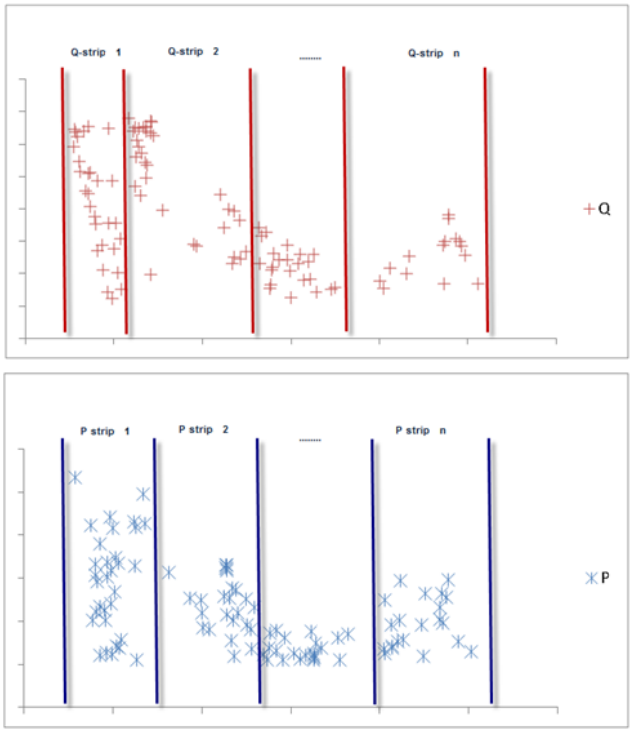

Figure 1. Strips partitioning.

- Using the distance of the K-th CP (*Bound*), combinations of strips are examined. If two strips reside in a distance smaller than the distance of the K-th CP, the pairs between data objects of these two strips are examined as candidates for the result. To achieve this, all (vertical) pairs of strips from P and Q are being evaluated with respect to their x-axis distance combined to the *Bound*.
- Pairs of strips are classified into two categories, namely eligible and not eligible for further processing. The first category consists of two major subcategories: overlapping pairs, and pairs that do not overlap but have their x-distance smaller than *Bound*. For example, (Fig. 2) strip Ps1 from P overlaps with strips Qs1 and Qs2 from Q. Furthermore, the x-distance between Ps1 and Qs3 is d1 < *Bound*, while the x-distance between Ps1 and Qs4 is d2 > *Bound* (this holds for every consecutive Q-strip). Therefore, the eligible pairs that we derive for Ps1 are (Ps1,Qs1), (Ps1,Qs2) and (Ps1,Qs3). These pairs, and all other pairs identified by this procedure, are the pairs that will be subject to computation by the cluster. Note, that in the case of pairs like (Ps1, Qs3), not all points from both strips need to be considered. For example, since we know a bound for the K CPs, we can use it as a pruning condition with the filter() function of Spark to reduce Qs3 to these points that their x-axis distance from Ps1 is smaller than *Bound*.

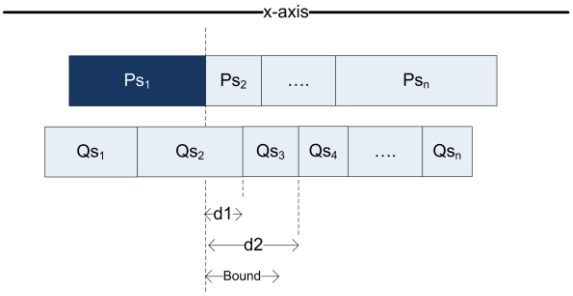

Figure 2. Eligible pairs of strips.

- Within each eligible pair of strips from P and Q Plane-sweep is applied for calculating K CPs storing the result in a maximum binary heap (maxHeap) [9][10]. A separate maxHeap is utilized for each partition. *Bound* is sent -we used Spark's broadcast() function- to all workers and they use it as stop condition for the plane sweep algorithm.
- All binary heaps are used to form a RDD consisting of tuples (*distance*, *Ppoint*, *Qpoint*). Since all eligible pairs of strips contain all pairs of points from P and Q that may contribute to the final solution and there are no duplicate pairs, taking the first (sorted on distance) K tuples with the smaller distances, yields the final (and exact) solution.

IV. EXPERIMENTAL EVALUATION

To evaluate the performance of our algorithm, we used the following three big real 2d datasets from OpenStreetMap [13]: 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 5 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 2.0.2 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 5 computing nodes, one was running the NameNodes for Hadoop and Master for Spark, while the remaining four (4 nodes x 4 vCPUs = 16 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.

All datasets are text files stored in HDFS. Each line contains an index and a pair of coordinates. We used the textFile() function of Spark to import the data, and set the numPartitions parameter to 4. Typically, Spark creates one partition for each block. We can increase the number of partitions by passing a larger value but it is not possible to have fewer partitions than the blocks of each file.

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.

 ISBN: 978-1-61208-529-6

We varied sample fraction (values used: 0.01, 0.001, 0.0001), the number of closest pairs K (values used: 1, 10, 100, 1000, 10000) and the number of strips per dataset (values used: 16, 32, 64, 80). We tested all possible combinations between the three datasets (PARKSxWATER, BUILDINGSxWATER, BUILDINGSxPARKS). In the following, we present a representative portion of the results.

In Fig. 3, we present the results for the PARKSxWATER combination, for K=10, using different combinations of n (number of strips) and f (sample fraction). As one can see, there is a tradeoff between total execution time and the time taken in order to sample the datasets and compute the value of *Bound*. If we take a small fraction of the datasets as sample, the bound we compute is not tight enough, therefore leading to increased KCPQ computation time. The larger the fraction of dataset we sample, the better (lower) is the upper bound we obtain. But if we surpass a certain fraction, then the computation of Bound in the sample dominates the total computation time.

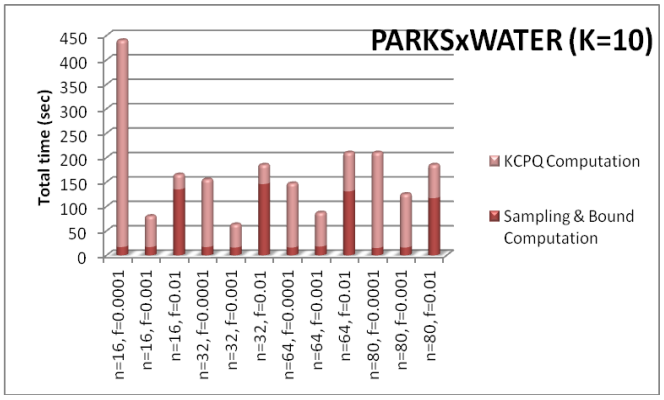

Figure 3. Effect of sample fraction.

Studying the results of the above experiment leads us to the observation that a fraction of 0.001 is a good selection for the rest of our experiments.

In Fig. 4, we present the results for the PARKSxWATER combination, for all K values, using 16, 32, 64 and 80 strips per each dataset. Initially, we ran each experiment independently from the others. We faced a problem, though. Phase two (the KCPQ computation) relies on the value of *Bound* that is computed in phase one. Since phase one uses a randomly selected sample, *Bound* is likely to be different in each experiment. In order to be able to extract better and comparable results, we used the following procedure for our second experiment: having taken into consideration that phase one is independent from phase two, we conducted the first phase of the experiment (K=1, n=16, fraction=0.001) and saved the calculated value of *Bound*. In all consecutive phases of the experiment, the bound was computed as usual, but we used the value we found in the first phase of the experiment instead.

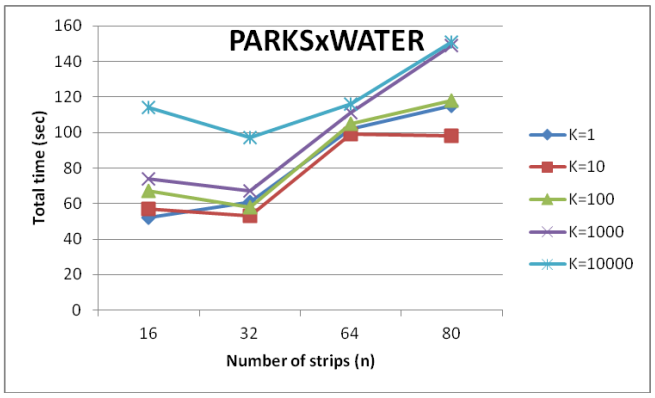

Figure 4. KCPQ (PARKS x WATER).

As we observe, n = 32 strips seems to be the optimal partitioning size for PARKS and WATER datasets, although n = 16 gives similar results. As K increases from 1 to 10,000, execution time is hardly affected, in some cases showing a tendency to increase slightly, as expected.

We conducted our third experiment in order to see to what extent the value of *Bound* affects the running time of the algorithm. We used a value for *Bound* with an order of magnitude 10 times greater than the one previously used. Time for sampling and bound computation was taken into account when counting total running time. Fig. 5 presents the running times compared to the ones that were measured in the previous experiment.

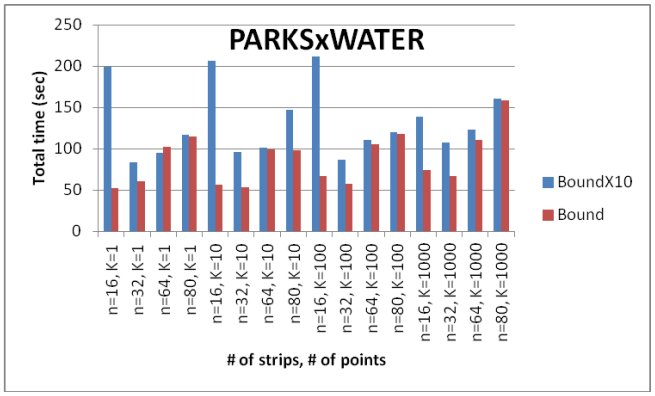

Figure 5. Effect of lower Bound

From the above comparison, we conclude that the value of *Bound* is more significant than the number of strips and the number of partitions provided to Spark as well.

In Fig. 6, we present the results for the BUILDINGSxWATER combination, for several K values using 8, 16, 32 and 64 strips per each dataset (once again *Bound* was set to a constant value for all cases, to an order of e-05). We observe than in the case of BULDINGS the algorithm gives better results for a lower number of strips than in the case of PARKS.

 ISBN: 978-1-61208-529-6

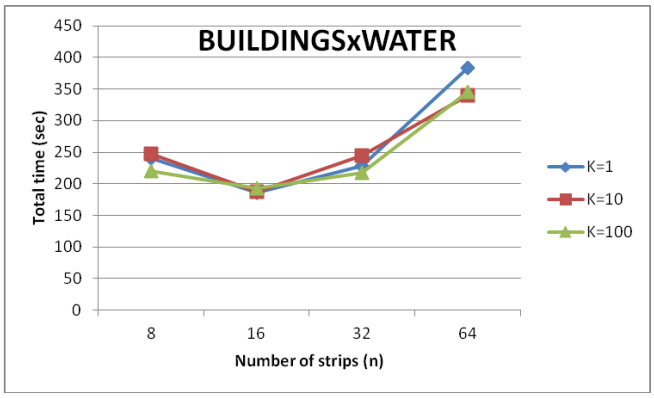

Figure 6. KCPQ (BUILDINGS x WATER).

We believe that this has to do with a combination of the characteristics of the multi-parametric system we study (hardware, HDFS, Spark, our algorithm). The combination of available cores, starting partitions, Spark partitioning procedures, number of strips that lead to a number of eligible pairs, results to an increased number of partitions that in the cases of larger n (strips) overwhelms the computing cluster.

In all previously described experiments, both datasets are being sliced into strips along x-axis (y-axis can also be used). Then, within each partition created by the eligible pairs of points from P and Q, plane sweep is applied along the other axis, in our case the y-axis. It is possible to slice the strips and sweep along the same axis (Fig. 7).

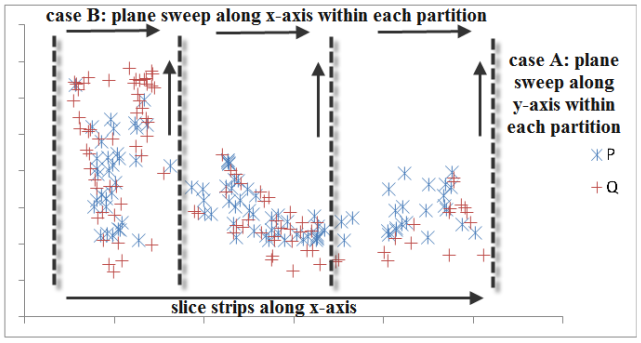

Figure 7. Strips Slice & Plane Sweep cases.

In order to check which choice is better (slicing and sweeping along the same or different axes), we conducted our next experiment. We used the BUILDINGSxPARKS combination with n = 8, 16, 32, K =10 and fraction f = 0.001.

We ran each combination three times, used the average time and the results are being presented in Fig. 8.

The results seem to lead us to the conclusion that "crossing" the axes for slicing and sweeping is more efficient than working on the same axis. This observation is clearer in the cases of smaller strips number, when the algorithm gives the best results.

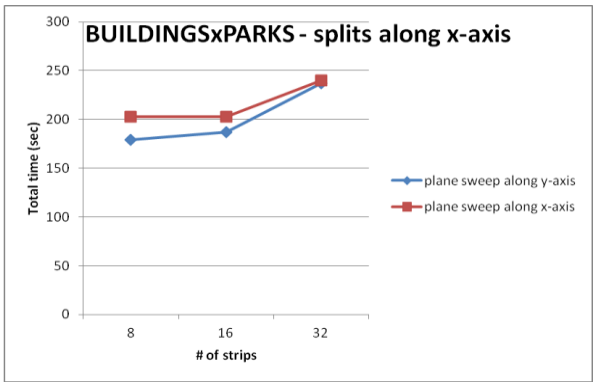

Figure 8. Split axis vs plane sweep axis.

Although this is consistent with other observations we have made during our experiments, we believe that it needs further investigation, an action we plan to take in the near future.

V. CONCLUSIONS AND FUTURE PLANS

In [9][10], plane-sweep algorithms and separation of data in strips were utilized for computing KCPQs in a centralized environment, taking advantage of main memory. In this paper, we present an algorithm for Spark, a parallel and distributed framework that supports in-memory processing, separating data in strips and processing by plane sweep within each strip. To the best of our knowledge, this is the first KCPQ algorithm in Spark. By conducting experiments on big real datasets we have explored the performance of our algorithm.

In the future, we plan to further elaborate this algorithm by exploring different ways to create strips of variable size 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. Another important research direction is finding a better, fast and stable technique that will yield a good upper bound for the KCPQ problem in a parallel system. We also plan to compare the performance of our algorithm against other solutions working in parallel and distributed environments. Finally, we plan to study the scalability of our algorithm.

REFERENCES

[1] S. Shekhar and H. Xiong, Encyclopedia of GIS. Springer, 2008.

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

[3] A. Corral and M. Vassilakopoulos, "Query processing in spatial databases," Encyclopedia of Database Technologies and Applications, pp. 511-516, 2005.

[4] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Closest Pair Queries in Spatial Databases," SIGMOD Conference, pp. 189-200, 2000.

[5] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Algorithms for processing K-closest-pair queries in spatial databases," Data Knowl. Eng., vol. 49, no. 1, pp. 67-104, 2004.

 ISBN: 978-1-61208-529-6

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," OSDI 2004, pp. 137-150, 2004.

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, pp. 10–10, 2010.

[8] M. Zaharia, et al., "Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing," NSDI 2012, pp. 15-28, 2012.

[9] G. Roumelis, M. Vassilakopoulos, A. Corral, and Y. Manolopoulos, "A new plane-sweep algorithm for the K-closest-pairs query," SOFSEM 2014, pp. 478-490, 2014.

[10] 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.

[11] J. Lu and R. H. Güting, "Parallel Secondo, Boosting Database Engines with Hadoop," ICPADS 2012, pp. 738-743, 2012.

[12] A. Aji, et al., "Hadoop-GIS: A high performance spatial data warehousing system over MapReduce," PVLDB, vol. 6, no. 11, pp. 1009-1020, 2013.

[13] A. Eldawy and M. F. Mokbel, "SpatialHadoop: A MapReduce framework for spatial data," ICDE 2015, pp. 1352-1363, 2015.

[14] F. García-García, A. Corral, L. Iribarne, M. Vassilakopoulos, and Y. Manolopoulos, "Enhancing SpatialHadoop with closest pair queries," ADBIS 2016, pp. 212-225, 2016.

[15] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in Cloud," ICDE Workshops 2015, pp. 34-41, 2015.

[16] J. Yu, J. Wu, and M. Sarwat, "GeoSpark: a cluster computing framework for processing large-scale spatial data," 23rd ACM SIGSPATIAL/GIS, pp. 70:1-70:4, 2015.

[17] 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.

[18] D. Xie, et al., "Simba: Efficient in-memory spatial analytics," SIGMOD Conference 2016, pp. 1071-1085, 2016.

[19] A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, "Algorithms for processing K-closest-pair queries in spatial databases," Data Knowl. Eng., vol. 49, no. 1, pp. 67-104, 2004.

[20] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Closest pair queries in spatial databases," SIGMOD Conference 2000, pp. 189-200, 2000.

[21] G. R. Hjaltason and H. Samet, "Incremental distance join algorithms for spatial databases," SIGMOD Conference 1998, pp. 237-248, 1998.

[22] 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.

[23] C. Yang and K-I. Lin, "An index structure for improving closest pairs and related join queries in spatial databases," IDEAS 2002, pp. 140-149, 2002.

[24] 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.

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

 ISBN: 978-1-61208-529-6