

Enhancing SpatialHadoop with Closest Pair Queries

Francisco García-García^{1,*}, Antonio Corral^{1,*}, Luis Iribarne^{1,*},
Michael Vassilakopoulos^{2,*}, and Yannis Manolopoulos^{3,*}

¹ Dept. of Informatics, University of Almeria, Almeria, Spain.

E-mail: {paco.garcia,acorral,liribarn}@ual.es

² Dept. of Electrical and Computer Engineering, University of Thessaly,

Volos, Greece. E-mail: mvasilako@uth.gr

³ Dept. of Informatics, Aristotle University,

Thessaloniki, Greece. E-mail: manolopo@cscd.auth.gr

Abstract. Given two datasets P and Q , the K Closest Pair Query ($KCPQ$) finds the K closest pairs of objects from $P \times Q$. It is an operation widely adopted by many spatial and GIS applications. As a combination of the K Nearest Neighbor (KNN) and the spatial join queries, $KCPQ$ is an expensive operation. Given the increasing volume of spatial data, it is difficult to perform a $KCPQ$ on a centralized machine efficiently. For this reason, this paper addresses the problem of computing the $KCPQ$ on big spatial datasets in SpatialHadoop, an extension of Hadoop that supports spatial operations efficiently, and proposes a novel algorithm in SpatialHadoop to perform efficient parallel $KCPQ$ on large-scale spatial datasets. We have evaluated the performance of the algorithm in several situations with big synthetic and real-world datasets. The experiments have demonstrated the efficiency and scalability of our proposal.

Keywords: Closest Pair Queries, Spatial Data Processing, SpatialHadoop, MapReduce.

1 Introduction

Given two point datasets P and Q , the K Closest Pair Query ($KCPQ$) finds the K closest pairs of points from $P \times Q$ according to a certain distance metric (e.g., Manhattan, Euclidean, Chebyshev, etc.). The $KCPQ$ has received considerable attention from the database community, due to its importance in numerous applications, such as spatial databases and GIS [1, 2], data mining [3], metric databases [4], etc. Since both the spatial join and the K Nearest Neighbor (KNN) queries are expensive, especially on large datasets, $KCPQ$, as a combination of both, is an even more costly query. Lots of researches have been devoted to improve the performance of the $KCPQ$ by proposing efficient algorithms [4, 5].

* Work funded by the MINECO research project [TIN2013-41576-R] and the Junta de Andalucía research project [P10-TIC-6114]

However, all these approaches focus on methods that are to be executed in a centralized environment.

With the fast increase in the scale of the big input datasets, processing large data in parallel and distributed fashions is becoming a popular practice. A number of parallel algorithms for spatial joins [6, 7], KNN joins [8, 9] and top- K similarity join [10] in MapReduce [11] have been designed and implemented. But, to the authors' knowledge, there is no research works on parallel and distributed $KCPQ$ in large spatial data, which is a challenging task and becoming increasingly essential as datasets continue growing.

Actually, extreme-scale data, parallel and distributed computing using shared-nothing clusters is becoming a dominating trend in the context of data processing and analysis. MapReduce [11] is a framework for processing and managing large-scale datasets in a distributed cluster, which has been used for applications such as generating search indexes, document clustering, access log analysis, and various other forms of data analysis [12]. MapReduce was introduced with the goal of providing a simple yet powerful parallel and distributed computing paradigm, providing good scalability and fault tolerance mechanisms.

However, as also indicated in [13], MapReduce has weaknesses related to efficiency when it needs to be applied to spatial data. A main shortcoming is the lack of any 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 [14], 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 are 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 [14], various algorithms are proposed for spatial queries, such as range and KNN queries, spatial joins and skyline query [15]. Efficient processing of $KCPQ$ over large-scale spatial datasets is a challenging task, and it is the main target of this paper.

Motivated by these observations, we first propose a general approach of $KCPQ$ for SpatialHadoop, using plane-sweep algorithms, and its improved version, using the computation of an upper bound of the distance of the K -th closest pair from sampled data points. The contributions of this paper are the following

- A novel algorithm in SpatialHadoop to perform efficient parallel $KCPQ$ on big spatial datasets,
- Improving the general algorithm with the computation of an upper bound of the distance value of the K -th closest pair from sampled data,
- The execution of an extensive set of experiments that demonstrate the efficiency and scalability of our proposal using big synthetic and real-world points datasets.

This paper is organized as follows. In Section 2 we review related work on Hadoop systems that support spatial operations, the specific spatial queries using MapReduce and provide the motivation for this paper. In Section 3, we present preliminary concepts related to $KCPQ$ and SpatialHadoop. In section

4 the parallel algorithm for processing *KCPQ* in SpatialHadoop is proposed, with an improvement to make the algorithm faster. In Section 5, 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 algorithm, taking into account different performance parameters. Finally, in Section 6 we provide the conclusions arising from our work and discuss related future work directions.

2 Related Work and Motivation

Researchers, developers and practitioners worldwide have started to take advantage of the MapReduce environment in supporting large-scale data processing. The most important contributions in the context of scalable spatial data processing are the following prototypes: (1) *Parallel-Secondo* [16] is a parallel spatial DBMS that uses Hadoop as a distributed task scheduler; (2) *Hadoop-GIS* [17] extends Hive [18], 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; (3) *SpatialHadoop* [14] 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; and (4) *SpatialSpark* [19] is a lightweight implementation of several spatial operations on top of the Apache Spark in-memory big data system. It targets at in-memory processing for higher performance. It is important to highlight that these four systems differ significantly in terms of distributed computing platforms, data access models, programming languages and the underlying computational geometry libraries.

Actually, there are several 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 works on specific spatial queries include: (1) Range query [20, 21], where the input file is scanned, and each record is compared against the query range. (2) *KNN* query [21, 22], where a brute force approach calculates the distance to each point and selects the nearest *K* points [21], while another approach partitions points using a Voronoi diagram and finds the answer in partitions close to the query point [22]. (3) Skyline queries [15, 25, 26]; in [25] the authors propose algorithms for processing skyline and reverse skyline queries in MapReduce; and in [15, 26] the problem of computing the skyline of a vast-sized spatial dataset in SpatialHadoop is studied. (4) Reverse Nearest Neighbor (RNN) query [22], where input data is partitioned by a Voronoi diagram to exploit its properties to answer RNN queries. (5) Spatial join [14, 21, 23]; in [21] the *partition-based spatial-merge* join [24] is ported to MapReduce, and in [14] the *map* function partitions the data using a grid while

the *reduce* function joins data in each grid cell. (6) *KNN* join [8,9,23], where the main target is to find for each point in a set P , its *KNN* points from set Q using MapReduce. Finally, (7) in [10], the problem of the top- K closest pair problem (where just one dataset is involved) using MapReduce is studied.

The *KCPQ* (where two spatial datasets are involved) has received considerable attention from the spatial database community, due to its importance in numerous applications. SpatialHadoop is equipped with a several spatial operations, including range query, *KNN* and spatial join [14], and other fundamental computational geometry algorithms as polygon union, skyline, convex hull, farthest pair, and closest pair [26]. And recently, two new algorithms for skyline query processing have been also proposed in [15]. And based on the previous observations, efficient processing of *KCPQ* over large-scale spatial datasets using SpatialHadoop is a challenging task, and it is the main motivation of this paper.

3 Preliminaries and Background

3.1 K Closest Pairs Query

The *KCPQ* discovers the K pairs of data elements formed from two datasets that have the K smallest distances between them (i.e. it reports only the top K pairs). It is one of the most important spatial operations when two spatial datasets are involved. It is considered a distance-based join query, because it involves two different spatial datasets and use distance functions to measure the degree of nearness between pairs of spatial objects. The formal definition of *KCPQ* for point datasets (the extension of this definition to other complex spatial objects is straightforward) is the following:

Definition 1. (*K Closest Pairs Query, KCPQ*) Let $P = \{p_0, p_1, \dots, p_{n-1}\}$ and $Q = \{q_0, q_1, \dots, q_{m-1}\}$ be two set of points in E^d , and a natural number K ($K \in \mathbb{N}, K > 0$). The *K Closest Pairs Query (KCPQ)* of P and Q ($KCPQ(P, Q, K) \subseteq P \times Q$) is a set of K different ordered pairs $KCPQ(P, Q, K) = \{(p_1, q_1), (p_2, q_2), \dots, (p_K, q_K)\}$, with $(p_i, q_i) \neq (p_j, q_j), i \neq j, 1 \leq i, j \leq K$, such that for any $(p, q) \subseteq P \times Q - \{(p_1, q_1), (p_2, q_2), \dots, (p_K, q_K)\}$ we have $dist(p_1, q_1) \leq dist(p_2, q_2) \leq \dots \leq dist(p_K, q_K) \leq dist(p, q)$.

This spatial query has been actively studied in centralized environments, regardless whether both spatial datasets are indexed or not [1,2,5,28]. In this context, recently, when the two datasets are not indexed and they are stored in main-memory, a new plane-sweep algorithm for *KCPQ*, called *Reverse Run*, was proposed in [5]. Additionally, two improvements to the *Classic* PS algorithm for this spatial query were also presented. Experimentally, the *Reverse Run* PS algorithm proved to be faster and it minimized the number of Euclidean distance computations. However, datasets that reside in a parallel and distributed framework have not attracted similar attention.

An example of this query using big data [14] could be to find the K closest pairs of buildings and water resources (since you may examine of other, more

ecological, sources of water supply for buildings). Moreover, due to the growing popularity of mobile and wearable location-aware devices that have access to the Web, *KCPQs* on big data are expected to appear in emerging new applications.

3.2 SpatialHadoop

SpatialHadoop [14] is a full-fledged MapReduce framework with native support for spatial data. Notice that MapReduce [11] 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 is specified by a *map function* takes input (typically from Hadoop Distributed File System (HDFS) files), possibly performs some computations on this input, and distributes it to worker nodes; and the *Reduce* phase which processes these results as specified by a *reduce function*. An important aspect of MapReduce is that both the input and the output of the *Map* step are represented as *Key/Value pairs*, and that pairs with 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*.

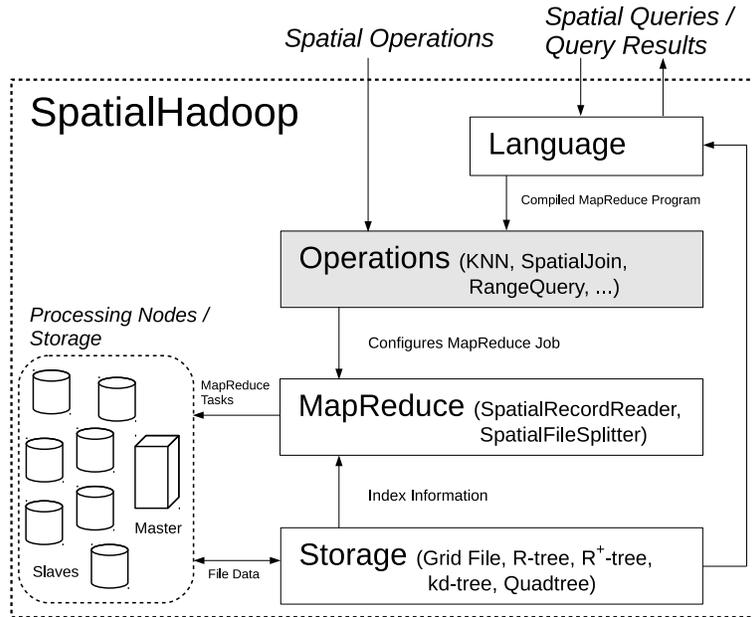


Fig. 1. SpatialHadoop system architecture [14].

SpatialHadoop, see in Figure 1 its architecture, 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 and R⁺-tree, to form a two-level spatial index [27]. SpatialHadoop enriches the *MapReduce* layer by new components to implement efficient and scalable spatial data processing. In the *Operations* layer, SpatialHadoop is also equipped with a several spatial operations, including range query, *K*NNQ 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 [26]. Moreover, in this context, [15] improved the processing of skyline query. Finally, we must emphasize that our contribution (*K*CPQ as a spatial operation) is located in the *Operations* layer, as we can observe in Figure 1 in the highlighted box.

Since our main objective is to include the *K*CPQ into SpatialHadoop, we are interested in the MapReduce and operations layers. MapReduce layer is the query processing layer that runs MapReduce programs, taking into account that SpatialHadoop supports spatially indexed input files. And the operation layer enables the efficient implementation of spatial operations, considering the combination of the spatial indexing in the storage layer with the new spatial functionality in the MapReduce layer. In general, a spatial query processing in SpatialHadoop consists of four steps: (1) *Partitioning*, where the data is partitioned according to a specific spatial index. (2) *Pruning*, when the query is issued, where the master node examines all partitions and prunes those ones that are guaranteed not to include any possible result of the spatial query. (3) *Local spatial query processing*, where a local spatial query processing is performed on each non-pruned partition in parallel on different machines. And finally, (4) *Global processing*, where a single machine collects all results from all machines in the previous step and compute the final result of the concerned query. And we are going to follow this query processing schema to include the *K*CPQ into SpatialHadoop.

4 *K*CPQ Algorithms in SpatialHadoop

In this section, we describe our approach to *K*CPQ algorithms on top of SpatialHadoop. This can be described as a generic top-*K* MapReduce job that can be parameterized by specific *K*CPQ algorithms. In general, our solution is similar to how *distributed join* algorithm [14] is performed in SpatialHadoop, where combinations of blocks from each dataset are the input for each *map* task, when the spatial query is performed. The *reducer* then emits the top-*K* results from all *mapper* outputs. In more detail, our approach make use of two plane-sweep *K*CPQ algorithms for main-memory resident datasets [5].

The *plane-sweep technique* [29] has been successfully used in spatial databases to report the result of *K*CPQ when the two datasets are indexed [1, 2], and recently it has been improved for main-memory resident-point sets [5]. In this

paper we will use the algorithms presented in [5] and their improvements to adapt them as MapReduce versions.

In [5], the *Classic Plane-Sweep* for *KCPQ* was reviewed and two new improvements were also presented, when the point datasets reside in main memory. In general, if we assume that the two point sets are P and 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) combine one point (*pivot*) of one set with all the points of the other set satisfying $point.x - pivot.x \leq \delta$, where δ is the distance of the K -th closest pair found so far. The algorithm chooses the *pivot* in P or Q , following the order on the sweeping axis. We must highlight that the search is only restricted to the closest points with respect to the *pivot*, according to the current distance threshold (δ). No duplicated pairs are obtained, since the points are always checked over sorted sets.

In [5], a new plane-sweep algorithm for *KCPQ* was proposed for minimizing the number of distance computations. It was called *Reverse Run Plane-Sweep* algorithm and it is based on two concepts. First, every point that is used as a *reference* point forms a *run* with other subsequent points of the same set. A *run* is a continuous sequence of points of the same set that doesn't contain any point from the other 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 second, 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 (*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, and (2) 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, and we avoid this distance computation.

The two improvements presented in [5], called *sliding window* and *sliding semi-circle*, can be applied both *Classic* and *Reverse Run* algorithms. For the *sliding window*, the general idea consists of 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 *pivot* or the *reference* point). And for the *sliding semi-circle* improvement, it consists of trying to reduce even more the search space, we can only select those points inside the semi-circle (or half-circle) centered in the *pivot* or in the *reference* point with radius δ .

The method for *KCPQ* in MapReduce is to adopt the top- K MapReduce methodology. The basic idea is to have P and Q partitioned by some method

(e.g., grid) into n and m blocks of points. Then, every possible pair of blocks (one from P and one from Q) is sent as the input for the *Map* phase. Each *mapper* reads its pair of blocks and performs a *KCPQ* PS algorithm (*Classic* or *Reverse Run*) between the local P and Q in that pair. That is, it finds *KCPs* between points in the local block of P and in the local block of Q using a *KCPQ* PS algorithm. To do so, each *mapper* sorts the local P and Q blocks in one axis (e.g., X axis in ascending order) and then applies a particular *KCPQ* algorithm. The K results from all *mappers* are sent to a single *reducer* that will in turn find the global top- K of all the *mappers*. Finally, the results are written into HDFS files, storing only the points coordinates and the distance between them.

Algorithm 1 *KCPQ* MapReduce General Algorithm

```

1: function MAP( $P$ : set of points,  $Q$ : set of points,  $K$ : # pairs)
2:   SORTX( $P$ )
3:   SORTX( $Q$ )
4:    $KMaxHeap \leftarrow KCPQ(P, Q, k)$ 
5:   if  $KMaxHeap$  is not empty then
6:     for all  $DistanceAndPair \in KMaxHeap$  do
7:       OUTPUT(null,  $DistanceAndPair$ )
8:     end for
9:   end if
10: end function

11: function COMBINE, REDUCE(null,  $P$ : set of  $DistanceAndPair$ ,  $K$ : # pairs)
12:   INITIALIZE( $CandidateKMaxHeap$ ,  $K$ )
13:   for all  $p \in P$  do
14:     INSERT( $CandidateKMaxHeap$ ,  $p$ )
15:   end for
16:   for all  $candidate \in CandidateKMaxHeap$  do
17:     OUTPUT(null,  $candidate$ )
18:   end for
19: end function

```

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 *KCPs* between the local pair of blocks from P and Q with a particular *KCPQ* algorithm (e.g. *Classic* or *Reverse Run*). *KMaxHeap* is a max binary heap 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, pairs of points from P and Q and their distance. 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 *KCP* set. It takes as input a set of *DistanceAndPair* elements from every mapper and the number of pairs. It also employs a binary max heap, called *CandidateKMaxHeap*, used to calculate

the final result. Each *DistanceAndPair* p is inserted into the heap if its distance value is less than the distance value of the top element stored in the heap. 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.

4.1 Improving the Algorithm

It can clearly be seen that the performance of the proposed solution will depend on the number of blocks in which the sets of points are partitioned. That is, the set of points P is partitioned into n blocks and the set of points Q is partitioned in m blocks, then we obtain $n \times m$ combinations or *map* tasks. Plane-Sweep-based algorithms use a δ value, which is the distance 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 [10], we need to find in advance an upper bound distance δ of the distance of the K -th closest pair of the datasets. As we can see in Algorithm 2, we take a small sample from both sets of points and calculate the K CPs using the same algorithm that is applied locally in every *mapper*. Then, we take the largest distance from the result and use it as input for *mappers*. That δ value assures us that there will be at least K closest pairs if we prune pairs of points with larger distances in every *mapper*.

Algorithm 2 Preprocessing δ Algorithm

```

1: function CALCULATE $\delta$ ( $P$ : set of points,  $Q$ : set of points,  $K$ : # pairs)
2:    $SampledP \leftarrow \text{SAMPLE}(P, 1\%)$ 
3:    $SampledQ \leftarrow \text{SAMPLE}(Q, 1\%)$ 
4:    $\text{SORTX}(SampledP)$ 
5:    $\text{SORTX}(SampledQ)$ 
6:    $KMaxHeap \leftarrow \text{KCPQ}(SampledP, SampledQ, K)$ 
7:   if  $KMaxHeap$  is not empty then
8:      $\deltaDistanceAndPair \leftarrow \text{POP}(KMaxHeap)$ 
9:      $\delta \leftarrow \deltaDistanceAndPair.Distance$ 
10:     $\text{OUTPUT}(\delta)$ 
11:   end if
12: end function

13: function CELLSFILTER( $C$ : set of cells,  $D$ : set of cells,  $\delta$ : upper bound distance)
14:   for all  $c \in C$  do
15:     for all  $d \in D$  do
16:        $minDistance \leftarrow \text{MINDISTANCE}(c, d)$ 
17:       if  $minDistance \leq \delta$  then
18:          $\text{OUTPUT}(c, d)$ 
19:       end if
20:     end for
21:   end for
22: end function

```

Furthermore, we can use this δ value in combination with the features of indexing that provides SpatialHadoop to further enhance the pruning phase. Before the *map* phase begins, we exploit the indexes to prune cells that cannot contribute to the final result. CELLSFILTER takes as input each combination of blocks / cells in which the input set of points are partitioned. Using SpatialHadoop built-in function *minDistance* we can calculate the minimum distance between two cells. That is, if we find a pair of blocks with points which cannot have a distance value smaller than δ , we can prune that combination. Performing the δ preprocessing filtering using 1% samples of the input data we have obtained results with a significant reduction of execution time.

5 Experimentation

In this section we present the results of our experimental evaluation. We have used synthetic (Uniform) and real 2d point datasets to test our *KCPQ* algorithms in SpatialHadoop. For synthetic datasets we have generated several files of different sizes using SpatialHadoop built-in uniform generator [14]. For real datasets we have used three datasets from OpenStreetMap¹: *BUILDINGS* which contains 115M records of buildings, *LAKES* which contains 8.4M points of water areas, and *PARKS* which contains 10M records of parks and green areas [14]. We have implemented and compared the *KCPQ* PS algorithms (Classic and Reverse Run [5]). We have used two performance metrics, the running time and the number of complete distance computations of each algorithm. All experiments are conducted on a cluster of 20 nodes on an OpenStack environment. Each node has 1 vCPU with 2GB of main memory running Linux operating systems and Hadoop 1.2.1.

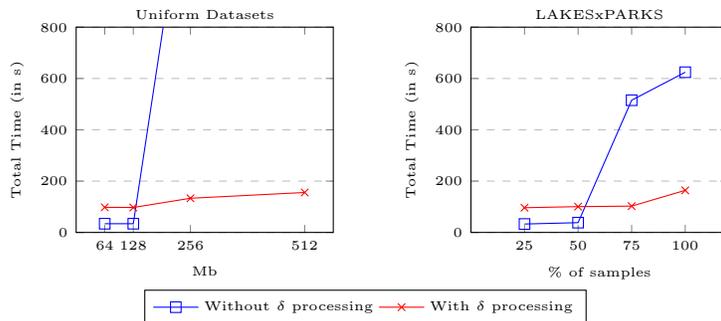


Fig. 2. Execution time vs. δ preprocessing phase.

Our first experiment is to examine the effect of the *preprocessing phase* to compute an upper bound of the distance value of the K -th closest pair (δ). As shown in Figure 2 the execution time for the algorithm without preprocessing

¹ Available at <http://spatialhadoop.cs.umn.edu/datasets.html>

is smaller when using uniform datasets with less than 256MB, see left graph. However, in the experiment with two grid partitioned datasets of 256MB the execution time increases considerably reaching several hours. As any combination of blocks is not removed, the calculation of $KCPQ$ is performed on pairs of blocks in which the value δ , that is being calculated, never exceeds the distance between these points. As a result pruning is never performed locally and, therefore, the calculation of all possible combinations of points is carried out. However, by adding δ preprocessing phase there are combinations of blocks which are first pruned obtaining times growing more or less linearly with the size of the datasets, see Figure 2 right graph. As an example, when using the complete dataset from *LAKES* and *PARKS* only 25 out of 64 possible combinations are considered for $K = 1$. In Table 1 all possible combinations of partitions are shown, considering different percentages of the datasets ($LAKES \times PARKS$) and, with or without the computation of the upper bound δ for $K = 1$ (for larger K values the percentage of reduction was similar).

% of Datasets	Without δ	With δ
25	4	3
50	12	6
75	56	20
100	64	25

Table 1. Number of combinations of partitions without or with using the δ preprocessing phase.

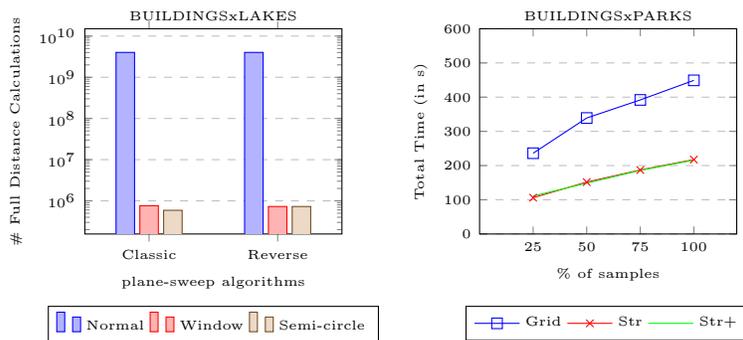


Fig. 3. Number of complete distance computation vs. $KCPQ$ algorithm (left) and execution time vs. partition technique (right).

The second experiment aims to find which of the different plane-sweep $KCPQ$ algorithms has the best performance. The times obtained do not show significant improvements between the different algorithms. This is due to various factors such as reading disk speed, network delays, the time for each individual task, etc. The metric shown in Figure 3 is based on the number of times the algorithm

performs a full calculation of the distance between two points. As shown in the left graph, any improvement (sliding window, semi-circle) on the *Classic* or *Reverse Run* algorithm obtains a much smaller number of calculations. The difference between these is not very noticeable being the *semi-circle reverse run* algorithm the one with better results in most of the cases.

The third experiment studies the effect of different spatial partitioning techniques included in SpatialHadoop. As shown in Figure 3 right graph, the choice of a partitioning technique greatly affects the execution time showing improvements of 200% when using *Str* or *Str+* instead of *Grid*. Using *Grid* partitioned files we get 211 combinations of blocks from input datasets while using *Str/Str+* partitioned files just 78 combinations are obtained. As expected, there is no real difference in using *Str* or *Str+*. This experiment is also useful to measure the scalability of the *KCPQ* algorithms, varying the dataset sizes. We can see that in our approach execution time increases linearly.

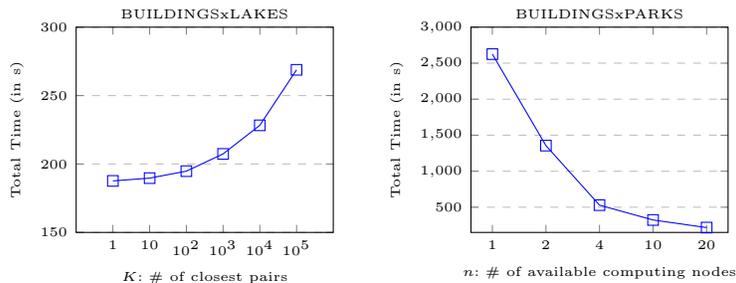


Fig. 4. Execution time vs. K value (left) and execution time vs. n (right).

The fourth experiment studies the effect of the increasing of the K value. As show on Figure 4 left graph, the total execution time grows very little as the number of results to be obtained increases. It could be concluded that there is no real impact on the execution time but it must be taken into account that a higher K , the greater the possibility that pairs of blocks are not pruned and more map tasks could be needed.

The fifth experiment aims to measure the speedup of the *KCPQ* algorithms, varying the number of computing nodes (n). Figure 4 right graph shows the impact of different node numbers on the performance of parallel *KCPQ* algorithm. From this figure, it could be concluded that the performance of our approach has direct relationship with the number of computing nodes. It could be deduced that better performance would be obtained if more computing nodes are added. When the number of computing nodes exceeds the number of *map* tasks no improvement for that individual job is obtained.

In summary, the experimental results showed that:

- We have demonstrated experimentally the efficiency (in terms of total execution time and number of distance computations) and the scalability (in terms of K values, sizes of datasets and number of computing nodes) of the proposed parallel *KCPQ* algorithm.

- We have improved this algorithm by using the computation of an upper bound δ of the distance of the K -th closest pair from sampled data.
- Both plane-sweep-based algorithms (*Classic* and *Reverse Run*) used in the MapReduce implementation have similar performance in terms of execution time, although the *Reverse Run* algorithm reduces slightly the number of complete distance computations.
- The use of an spatial partitioning technique included in SpatialHadoop as *Str* or *Str+* (instead of *Grid*) improves notably the efficiency of the parallel KCPQ algorithm. This is due to these variants index all partitions according to an R-tree structure (i.e. it can be viewed as a *global index of partitions*).

6 Conclusions and Future Work

The KCPQ is an operation widely adopted by many spatial and GIS applications. It returns the K closest pairs of spatial objects from the Cartesian Product of two spatial datasets P and Q . This spatial query has been actively studied in centralized environments, however, for parallel and distributed frameworks has not attracted similar attention. For this reason, in this paper, we studied the problem of answering the KCPQ in SpatialHadoop, an extension of Hadoop that supports spatial operations efficiently. To do this, we have proposed a new parallel KCPQ algorithm in MapReduce on big spatial datasets, adopting the plane-sweep methodology. We have also improved this MapReduce algorithm with the computation of an upper bound (δ) of the distance value of the K -th closest pair from sampled data as a preprocessing phase. The performance of the algorithm in various scenarios with big synthetic and real-world points datasets has been also evaluated. And, the execution of such experiments has demonstrated the efficiency (in terms of total execution time and number of distance computations) and scalability (in terms of K values, sizes of datasets and number of computing nodes) of our proposal. Future work might cover studying of KCPQ with other partition techniques not included in SpatialHadoop.

References

1. A. Corral, Y. Manolopoulos, Y. Theodoridis and M. Vassilakopoulos: “Closest pair queries in spatial databases”, *SIGMOD Conference*, pp. 189-200, 2000.
2. A. Corral, Y. Manolopoulos, Y. Theodoridis and M. Vassilakopoulos: “Algorithms for processing K -closest-pair queries in spatial databases”, *Data Knowl. Eng.* 49(1): 67-104, 2004.
3. A. Nanopoulos, Y. Theodoridis and Y. Manolopoulos: “C²P: Clustering based on closest pairs”, *VLDB Conference*, pp. 331-340, 2001.
4. Y. Gao, L. Chen, X. Li, B. Yao and G. Chen: “Efficient k -closest pair queries in general metric spaces”, *VLDB J.* 24(3): 415-439, 2015.
5. G. Roumelis, M. Vassilakopoulos, A. Corral and Y. Manolopoulos: “A new plane-sweep algorithm for the K -closest-pairs query”, *SOFSEM Conference*, pp. 478-490, 2014.

6. S. Zhang, J. Han, Z. Liu, K. Wang and Z. Xu: "SJMR: Parallelizing spatial join with MapReduce on clusters", *CLUSTER Conference*, pp. 1-8, 2009.
7. S. You, J. Zhang and L. Gruenwald: "Spatial join query processing in cloud: Analyzing design choices and performance comparisons", *ICPP Workshops*, pp. 90-97, 2015.
8. C. Zhang, F. Li and J. Jestes: "Efficient parallel k -NN joins for large data in MapReduce", *EDBT Conference*, pp. 38-49, 2012.
9. W. Lu, Y. Shen, S. Chen and B.C. Ooi: "Efficient processing of k nearest neighbor joins using MapReduce", *PVLDB* 5(10): 1016-1027, 2012.
10. Y. Kim and K. Shim: "Parallel top- K similarity join algorithms using MapReduce", *ICDE Conference*, pp. 510-521, 2012.
11. J. Dean and S. Ghemawat: "MapReduce: Simplified data processing on large clusters", *OSDI Conference*, pp. 137-150, 2004.
12. F. Li, B.C. Ooi, M.T. Ozsü and S. Wu: "Distributed data management using MapReduce", *ACM Comput. Surv.* 46(3): 31:1-31:42, 2014.
13. C. Doukeridis and K. Nørnvåg: "A survey of large-scale analytical query processing in MapReduce", *VLDB J.* 23(3): 355-380, 2014.
14. A. Eldawy and M.F. Mokbel: "SpatialHadoop: A MapReduce framework for spatial data", *ICDE Conference*, pp. 1352-1363, 2015.
15. D. Pertesis and C. Doukeridis: "Efficient skyline query processing in SpatialHadoop", *Inf. Syst.* 54: 325-335, 2015.
16. J. Lu and R.H. Güting: "Parallel Secondo: Boosting database engines with Hadoop", *ICPADS Conference*, pp. 738-743, 2012.
17. 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* 6(11): 1009-1020, 2013.
18. A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff and R. Murthy: "Hive - A warehousing solution over a MapReduce framework", *PVLDB* 2(2): 1626-1629, 2009.
19. S. You, J. Zhang and L. Gruenwald: "Large-scale spatial join query processing in cloud", *ICDE Workshops*, pp. 34-41, 2015.
20. Q. Ma, B. Yang, W. Qian and A. Zhou: "Query processing of massive trajectory data based on MapReduce", *CloudDB Conference*, pp. 9-16, 2009.
21. S. Zhang, J. Han, Z. Liu, K. Wang and S. Feng: "Spatial queries evaluation with MapReduce", *GCC Conference*, pp. 287-292, 2009.
22. A. Akdoğan, U. Demiryurek, F.B. Kashani and C. Shahabi: "Voronoi-based geospatial query processing with MapReduce", *CloudCom Conference*, pp. 9-16, 2010.
23. K. Wang, J. Han, B. Tu, J. Dai, W. Zhou and X. Song: "Accelerating spatial data processing with MapReduce", *ICPADS Conference*, pp. 229-236, 2010.
24. J.M. Patel and D.J. DeWitt: "Partition based spatial-merge join", *SIGMOD Conference*, pp. 259-270, 1996.
25. Y. Park, J.K. Min and K. Shim: "Parallel computation of skyline and reverse skyline queries using MapReduce", *PVLDB* 6(14): 2002-2013, 2013.
26. A. Eldawy, Y. Li, M.F. Mokbel and R. Janardan: "CG.Hadoop: computational geometry in MapReduce", *SIGSPATIAL Conference*, pp. 284-293, 2013.
27. A. Eldawy, L. Alarabi and M.F. Mokbel: "Spatial partitioning techniques in SpatialHadoop", *PVLDB* 8(12): 1602-1613, 2015.
28. G. Gutierrez and P. Sáez: "The k closest pairs in spatial databases - When only one set is indexed", *GeoInformatica* 17(4): 543-565, 2013.
29. F.P. Preparata and M.I. Shamos: "*Computational geometry: An introduction*", Springer-Verlag, 1985.