New Plane-Sweep Algorithms for Distance-Based Join Queries in Spatial Databases

George Roumelis · Antonio Corral · Michael Vassilakopoulos · Yannis Manolopoulos

Abstract Efficient and effective processing of the *distance-based join query* (DJQ) is of great importance in spatial databases due to the wide area of applications that may address such queries (mapping, urban planning, transportation planning, resource management, etc.). The most representative and studied DJQs are the K Closest Pairs Query (KCPQ) and εDistance Join Query (εDJQ). These spatial queries involve two spatial data sets and a distance function to measure the degree of closeness, along with a given number of pairs in the final result (K) or a distance threshold (ε). In this paper, we propose four new plane-sweep-based algorithms for KCPQs and their extensions for εDJQs in the context of spatial databases, without the use of an index for any of the two disk-resident data sets (since, building and using indexes is not always in favor of processing performance). They employ a combination of *plane-sweep* algorithms and space partitioning techniques to join the data sets. Finally, we present results of an extensive experimental study, that compares the efficiency and effectiveness of the proposed algorithms for KCPQs and εDJQs. This performance study, conducted on medium and big spatial data sets (real and synthetic) validates that the proposed plane-sweep-based algorithms are very promising in terms of both efficient and effective measures, when neither inputs are indexed. Moreover, the best of the new algorithms is experimentally compared to the best algorithm that is based on the R-tree (a widely accepted access method), for KCPQs and εDJQs, using the same data sets. This comparison shows that the new algorithms outperform R-tree based algorithms, in most cases.

Keywords Spatial Databases · Query Processing · Plane-Sweep Technique · Distance-based Join Queries · Spatial Query Evaluation

1 Introduction

A *Spatial Database* is a database system that offers spatial data types in its data model and query language, and it supports spatial data types in its implementation, providing at least spatial indexing and efficient spatial query processing [2]. In a computer system, these spatial data are represented by points, line-segments, regions, polygons, volumes and other kinds of 2-d/3-d geometric entities and are usually referred to as *spatial objects*. For example, a spatial database may contain polygons that represent building footprints from a satellite image, or points that represent the positions of cities, or line segments that represent roads. Spatial databases include specialized systems like Geographical databases, CAD databases, Multimedia databases, Image databases, etc. Recently, the role of spatial databases is continuously increasing in many modern applications; e.g. mapping, urban planning, transportation planning, resource management, geomarketing, environmental modeling are just some of these applications.

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

George Roumelis · Yannis Manolopoulos
Dept. of Informatics, Aristotle University, GR-54124 Thessaloniki, Greece.
E-mail: groumeli, manolopo@csd.auth.gr

Antonio Corral
Dept. on Informatics, University of Almeria, 04120 Almeria, Spain.
E-mail: acorral@ual.es

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

The most basic use of such a system is for answering spatial queries related to the spatial properties of the data. Some typical spatial queries are: *point query*, *range query*, *spatial join*, and *nearest neighbor query* [3]. One of the most frequent spatial queries in spatial database systems is the *spatial join*, which finds all pairs of spatial objects from two spatial data sets that satisfy a spatial predicate, θ. Some examples of the spatial predicate θ are: *intersects, contains, is_enclosed_by, distance, adjacent, meets*, etc. [4]; and when θ is a *distance*, we have *distance-based join queries* (DJQ). The most representative and studied DJQ in the spatial database field are the K Closest Pairs Query (KCPQ) and εDistance Join Query (εDJQ). The *KCPQ* combines join and nearest neighbor queries: like a join query, all pairs of objects are candidates for the final result, and like a nearest neighbor query, the K Nearest Neighbor property is the basis for the final ordering [5,6]. The *εDJQ*, also known as *Range Distance Join*, also involves two spatial data sets and a distance threshold ε, and it reports a set pairs of objects, one from each input set, that are within distance ε of each other. DJQ are very useful in many applications that use spatial data for decision making and other demanding data handling operations. For example, we can use two spatial data sets that represent the cultural landmarks and the most populated places of the United States of America. A KCPQ ($K = 10$) can discover the 10 closest pairs of cities and cultural landmarks providing an increasing order based on their distances. On the other hand, a εDJQ ($\varepsilon = 10$) will return all possible pairs (populated place, cultural landmark) that are within 10 kilometers of each other.

The distance functions are typically based on a distance metric (satisfying the non-negative, identity, symmetry and Δ-inequality properties) defined on points in the data space. A general distance metric is called L_t-distance or Minkowski distance between two points, in the d-dimensional data space, D^d. For $t = 2$ we have the *Euclidean distance*, for $t = 1$ the *Manhattan distance* and for $t = \infty$ the *Maximum distance*. They are the most known L_t-distances. Often, the Euclidean distance is used as the distance function but, depending on the application, other distance functions may be more appropriate. The *d-dimensional Euclidean space*, E^d, is the pair (D^d, L_2). That is, E^d is D^d with the Euclidean distance L_2. In the following we will use *dist* instead of L_2 as the Euclidean distance between two points in E^d and this will be the basis for DJQs studied on this paper.

One of the most important techniques in the computational geometry field is the *plane-sweep* algorithm which is a type of algorithm that uses a conceptual *sweepline* to solve various problems in the Euclidean plane, E^2, [7]. The name of *plane-sweep* is derived from the idea of sweeping the plane from left to right with a vertical line (front) stopping at every transaction point of a geometric configuration to update the front. All processing is carried out with respect to this moving front, without any backtracking, with a look-ahead on only one point each time [8]. The *plane-sweep* technique has been successfully applied in spatial query processing, mainly for intersection joins, regardless whether both spatial data sets are indexed or not [9]. In the context of DJQ the *plane-sweep* technique has been used to restrict all possible combinations of pairs of points from the two data sets. That is, using this technique instead of the brute-force nested loop algorithm, the reduction of the number of Euclidean distances computations has been proven [10,6], and thus the reduction of execution time of the query processing.

It is generally accepted that indexing is crucial for efficient processing of spatial queries. Even more, it is well-known that a spatial join is generally fastest if both data sets are indexed. However, there are many situations where indexing does not necessarily pay off. In particular, the time needed to build the index before the execution of the spatial query plays an important role in the global performance of the spatial database systems. For instance, if the output of a spatial query serves as input to another spatial query, and such an output is not reused several times for subsequent spatial queries, then it may not be worthwhile to spend the time for building a new index. This is especially emphasized for spatial intersection joins that make use of indexes which need a long time to be built (e.g. R*-tree [11]) [12]. For the previous reasons, the time necessary to build the indexes is an important constraint, especially if the input data sets are not used often for spatial query processing. Thus the main motivation of this article is to propose new algorithms for DJQs (the KCPQ and εDJQ) on disk resident data, when none inputs are indexed, and to study their behavior in the context of spatial databases. Our proposal is also motivated by the work of [13,14] for spatial intersection joins.

Nowadays, the unnecessity of indexes for query processing is not infrequent in practical applications, when the data sets change at a very rapid rate, or the data sets are not reusable for subsequent queries and the use of indexes can be omitted. Moreover, disk-based solutions are necessary, since main memory of a computing system is, in many cases, shared among applications, and it is usually not enough to hold big data (although, main memory increases in size and decreases in cost, acquired data increase at higher rates than main memory, for example, scientific data). As a possible application scenario, consider cadastre, or urban planning very big data sets with spatial and non-spatial characteristics. Big subsets of the data sets may be formed by considering certain (mainly non-spatial) characteristics of the stored properties, or buildings (like, properties owned by the state, buildings higher than 50 meters, constructions older than 50 years, or built under an obsolete anti-seismic construction standard, non build-up large areas, etc). These (big and non-storable in main memory) subsets are dynamic, or non-reusable, in the sense that an engineer, or an official may create them by setting conditions for certain characteristics, use them to answer a query, modify these conditions (and the created subsets), answer again this query, and so on. In the process of conducting a study, like an emergency planning study, the DJQ of interest might be to find pairs of

buildings vulnerable by an earthquake and earthquake-safe public buildings that could temporarily host people, at a limited distance.

This paper substantially extends our previous work [1] and its contributions are summarized as follows:

1. We present theorems (the proofs of these theorems are included in [15]) regarding the correctness of both algorithms for KCPQ, that is, *Classic Circle Plane-Sweep* (CCPS) and *Reverse Run Circle Plane-Sweep* (RCPS) algorithms. They are the basis of the following algorithms for DJQ, when neither inputs are indexed and the data are stored on disk.
2. There are many contributions in the context of spatial intersection joins when both, one, or neither inputs are indexed. For DJQs most of the contributions have been proposed when both inputs are indexed (mainly using R-trees for KCPQ). For this reason, in this article we propose four algorithms (FCCPS, SCCPS, FRCPS and SRCPS) for KCPQs and their extensions for εDJQs for performing DJQs, without the use of an index on any of the two disk-resident data sets. These algorithms employ a combination of the *plane-sweep* algorithms (CCPS) and (RCPS) and space partitioning techniques (uniform splitting and uniform filling) to join the disk-resident data sets.
3. We present results of an extensive experimental study, that compares the performance (in terms of efficiency and effectiveness) of the proposed algorithms.
4. We also compare the performance (efficiency) of the best of the new algorithms to the best algorithm that is based on the R-tree (a widely accepted access method).

The rest of this paper is organized as follows. Section 2 defines the KCPQ and εDJQ, which are the queries studied on this paper, in the context of spatial databases. Moreover a classification of spatial join and distance-based join queries taking into account whether both, one, or neither inputs are indexed is presented. The *Classic Plane-Sweep* algorithm for DJQs is described in Section 3, as well as two improvements to reduce the number of distance computations. In Section 4, the new *plane-sweep* algorithm (*Reverse Run Plane-Sweep*, *RRPS*) for KCPQ is presented. In Section 5, we present and analyse the new plane-sweep-based algorithms for the KCPQ and εDJQ. Section 6 exposes the results of an extensive experimental study, taking into account different parameters for comparison. Moreover, Secton 6 exposes the results of an extensive experimental comparison between the best of the new algorithms and the best R-tree based algorithm. Section 7 contains some concluding remarks and makes suggestions for future research.

2 Preliminaries and Related Work

Given two spatial data sets and a distance function to measure the degree of closeness, DJQs between pairs of spatial objects are important joins queries that have been studied actively in the last years. Section 2.1 defines the KCPQ and εDJQ, which are the kernel of this paper. Section 2.2 describes a classification of spatial join and distance-based join queries taking into account whether both, one, or neither inputs are indexed, along with the review of other recent contributions related to these DJQs.

2.1 K Closest Pairs Query and εDistance Join Query

In spatial database applications, the nearness or farness of spatial objects is examined by performing distance-based queries (DBQs). The most known DBQs in the spatial database framework when just a spatial data set is involved are the range query (RQ) and the K Nearest Neighbors query (KNNQ). When we have two spatial data sets the most representative DBQ are the K Closest Pairs Query (KCPQ) and the εDistance Join Query (εDJQ). They are considered DJQs, because they involve two different spatial data sets and use distance functions to measure the degree of nearness between spatial objects. The former reports only the top K pairs, and the latter, also known as *Range Distance Join*, finds all the possible pairs of spatial objects, having a distance between ε_1 and ε_2 of each other ($\varepsilon_1 \leq \varepsilon_2$). Their formal definitions for point data sets (the extension of these definitions to other complex spatial objects is straightforward) are the following:

Definition 1 *(K **Closest Pairs Query, KCPQ**)* Let $P = \{p_0, p_1, \cdots, p_{n-1}\}$ and $Q = \{q_0, q_1, \cdots, q_{m-1}\}$ 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 $KCPQ(P,Q,K) = \{(p_{Z1}, q_{L1}), (p_{Z2}, q_{L2}), \cdots, (p_{ZK}, q_{LK})\}$, with $(p_{Zi}, q_{Li}) \neq (p_{Zj}, q_{Lj})$, $Zi \neq Zj \wedge Li \neq Lj$, such that for any $(p,q) \in P \times Q - \{(p_{Z1}, q_{L1}), (p_{Z2}, q_{L2}), \cdots, (p_{ZK}, q_{LK})\}$ we have $dist(p_{Z1}, q_{L1}) \leq dist(p_{Z2}, q_{L2}) \leq \cdots \leq dist(p_{ZK}, q_{LK}) \leq dist(p,q)$.

Definition 2 *(**εDistance Join Query, εDJQ**)* Let $P = \{p_0, p_1, \cdots, p_{n-1}\}$ and $Q = \{q_0, q_1, \cdots, q_{m-1}\}$ be two set of points in E^d, and a range of distances defined by $[\varepsilon_1, \varepsilon_2]$ such that $\varepsilon_1, \varepsilon_2 \in \mathbb{R}^+$ and $\varepsilon_1 \leq \varepsilon_2$. The εDistance Join Query (ε**DJQ**) of P and Q ($\varepsilon\text{DJQ}(P, Q, \varepsilon_1, \varepsilon_2) \subseteq P \times Q$) is a set which contains all the possible pairs of points (p_i, q_j) that can be formed by choosing one point $p_i \in P$ and one point of $q_j \in Q$, having a distance between ε_1 and ε_2 for each other: $\varepsilon\text{DJQ}(P, Q, \varepsilon_1, \varepsilon_2) = \{(p_i, q_j) \in P \times Q : \varepsilon_1 \leq dist(p_i, q_j) \leq \varepsilon_2\}$.

These two DJQs have been actively studied in the context of R-trees [16,5,10,6]. However, when the data sets are not indexed they have attracted similar attention.

2.2 Related Work

This section presents a classification of the spatial join and distance-based join queries depending on one, both or neither inputs are indexed. Moreover, other related DJQ are also revised in the recent literature, in order to show the importance of this type of query in the context of spatial databases.

2.2.1 Spatial Join

Spatial data processing is well-known to be both data and computing intensive. The *spatial join* is one of the most studied spatial query, where given two datasets of spatial objects in Euclidean space, it finds all pairs of spatial objects satisfying a given spatial predicate, such as intersects, contains, etc [4]. Various techniques, such as minimizing disk I/O overheads in spatial indexing and the two phase filter-refinement strategy in spatial joins have been proposed in [9]. During the past decades many algorithms for spatial joins where the datasets reside on disk have been proposed in the literature [17,18,9] and recently, several contributions in the context of in-memory spatial join have been proposed. In [19], the authors have developed *TOUCH*, a novel in-memory spatial join algorithm, inspired with previous works on disk-based approaches and the requirements of the computational neuroscientists. It combines hierarchical data-oriented partitioning, batch processing and filtering concepts, with the target to decrease the number of comparisons, execution time and memory footprint of a spatial join process. In [20], a thorough experimental performance study of several (ten) spatial join techniques in main memory is reported. The techniques are first optimized for in-memory performance and then studied in the same framework. This study suggests that specialized join strategies over simple index structures, such as Synchronous Traversal over R-trees, should be the methods of choice for the considered cases. In [21], the authors re-implement the worst performing technique presented in [20] without changing the underlying high-level algorithm and the conclusion is that the resulting re-implementation is capable of outperforming all the other techniques. It means substantial performance gains can be achieved by means of careful implementation. Finally, in [22] a thorough review of a wide range of in-memory data management and processing proposals and systems is presented, including both data storage systems and data processing frameworks. The authors give a comprehensive presentation of important technology in memory management, and some key factors that need to be considered in order to achieve efficient in-memory data management and processing. In this paper, we are going to focus on disk-resident data, new algorithms for in-memory DJQs is a task for further research.

The *spatial join* is one of the most related and influential spatial queries with respect to DJQs in spatial databases and GIS. Depending on the existence of indexes or not, different spatial join algorithms have been proposed [23]. If both inputs are indexed, several contributions have been proposed, but the most influential one is the R-tree join algorithm (*RJ*) [24], due to its efficiency and the popularity of R-trees [25,11]. *RJ* synchronously traverses both trees in a Depth-First order. Two optimization techniques were also proposed, *search space restriction* and *plane-sweep*, to improve the CPU speed and to reduce the cost of computing overlapping pairs between the nodes to be joined, respectively.

Most research after *RJ*, focused on spatial join processing when one or both inputs are non-indexed. In this category, the paper that is most closely related to our work is [14], where several spatial joins strategies when only one input data set is indexed are investigated. The main contribution is a method that modifies the *plane-sweep* algorithm. This approach reads the data pages from the index in a one-dimensional sorted order and inserts entire data pages into the sweep structure (i.e. in this case, one sweep structure will contain objects, while another sweep structure will contain data pages).

Directly related to this paper, when both data sets are non-indexed, are methods that involve sorting and external memory plane-sweep [13,12], or spatial hash join algorithms [26], like partition based spatial merge join [27]. In [13] the *Scalable Sweeping-Based Spatial Join, SSSJ*, was proposed, that employs a combination of plane-sweep and space partitioning to join the data sets, and it works under the assumption that in most cases the limit of the *sweepline* will fit in main memory. In [27] a hash-join algorithm was presented, so called *Partition Based Spatial Merge Join*, that regularly partitions the space, using a rectangular grid, and hashes both inputs data sets

into the partitions. It then joins groups of partitions that cover the same area using plane-sweep to produce the join results. Some objects from both sets may be assigned in more than one partitions, so the algorithm needs to sort the results in order to remove the duplicate pairs. Finally, [12] extends the *SSSJ* of [13] to process data sets of any size by using external memory, proposing a new join algorithm referred as *iterative spatial join*.

2.2.2 KCPQ and εDJQ

The problem of closest pairs has received significant research attention by the computational geometry community (see [28] for an exhaustive survey), when all data are stored into the main memory. However, when the amount of data is too large (e.g. when we are working with spatial databases) it is not possible to maintain these data structures in main memory, and it is necessary to store the data on disk. Here, we are going to review the KCPQ and εDJQ, focusing on whether the input data sets are indexed or not. We must emphasize that most of the contributions that have been published until now are focused on the case when both data sets are indexed on R-trees.

Remind that given two spatial data sets P and Q, the KCPQ asks for the K closest pairs of spatial objects in $P \times Q$. If both P and Q are indexed by R-trees, the concept of synchronous tree traversal and Depth-First (DF) or Best-First (BF) traversal order can be combined for the query processing [16,5,6]. For a more detailed explanation of the processing of KCPQ-DF and KCPQ-BF algorithms on two R*-trees from the non-incremental point of view, see [6,15]. In [16], incremental and non-recursive algorithms based on Best-First traversal using R-trees and additional priority queues for DJQs were presented. In [10], additional techniques as sorting and application of plane-sweep during the expansion of node pairs, and the use of the estimation of the distance of the K-th closest pair to suspend unnecessary computations of MBR distances are included to improve [16]. A Recursive Best-First Search (RBF) algorithm for DBQ between spatial objects indexed in R-trees was presented in [29], with an exhaustive experimental study that compares DF, BF and RBF for several distance-based queries (Range Distance, K-Nearest Neighbors, K-Closest Pairs and Range Distance Join). Recently, in [30], an extensive experimental study comparing the R*-tree and Quadtree-like index structures for K-Nearest Neighbors and K-Distance Join queries together with index construction methods (dynamic insertion and bulk-loading algorithm) is presented. It was shown that when data are static the R*-tree shows the best performance. However, when data are dynamic, a bucket Quadtree begins to outperform the R*-tree. This is due to, once the dynamic R*-tree algorithm is used, the overlap among MBRs increases with increasing data set sizes, and the R*-tree performance degrades.

In the case where just only one data set is indexed, recently in [31] a new algorithm has been proposed for KCPQs. The main idea is to partition the space occupied by the data set without an index into several cells or subspaces (according to the VA-File structure [32]) and to make use of the properties of a set of distance functions defined between two MBRs [6].

To the best og the authors knowledge, there are no papers in the literature of spatial databases that have addressed the problem of DJQs if both data sets are non-indexed, and for this reason this is the main motivation of this research work.

εDJQ, also known as Range Distance Join, is a generalization of the *Buffer Query*, which is characterized by two spatial data sets and a distance threshold ε, which permits search pairs of spatial objects from the two input data sets that are within distance ε from each other. In our case, the distance threshold is a range of distances defined by an interval of distance values $[\varepsilon_1, \varepsilon_2]$ (e.g. if $\varepsilon_1 = 0$ and $\varepsilon_2 > 0$, then we have the definition of *Buffer Query* and if $\varepsilon_1 = \varepsilon_2 = 0$, then we have the *spatial intersection join*, which retrieves all different intersecting spatial object pairs from two distinct spatial data sets [9]). This query is also related to the *similarity join* in multidimensional databases [33], where the problem of deciding if two objects are similar is reduced to the problem of determining if two multidimensional points are within a certain distance of each other. In [34], the *Buffer Query* is solved for non-point (lines and regions) spatial data sets using R-trees, where efficient algorithms for computing the minimum distance for lines and regions, pruning techniques for filtering in a Depth-First search algorithm (performance comparisons with other search algorithms are not included), and extensive experimental results are presented. We must emphasize that there are no contributions in the literature of spatial databases for εDJQ when one or both inputs are non-indexed.

2.2.3 Other related Distance-Based Join Queries

Several DJQs have been studied in the literature which are related to KCPQ and εDJQ. In [35] a new index structure, called $bRdnn-Tree$, to solve different distance-based join queries is proposed. Other variants of KCPQ have also been studied in the context of spatial databases. More specifically, approximate K closest pairs in high dimensional data [36,37] and constrained K closest pairs [38] have been presented. In [39] the *exclusive closest pairs* problem is introduced (which is a spatial assignment problem) and several solutions that solve it in main

Strips	Points {index, (x, y)}			
PS_0	{0,(0,4)}	{1,(4,15)}	{2,(10,21)}	{3,(17,2)}
PS_1	{4,(19,8)}	{5,(20,21)}	{6,(22,1)}	{7,(23,17)}
PS_2	{8,(23,20)}	{9,(25,28)}	{10,(26,23)}	{11,(27,2)}
PS_3	{12,(29,9)}	{13,(30,10)}	{14,(33,28)}	{15,(37,18)}

Table 1 The data set $\mathcal{P}$ with 16 points in X-sorted order.

memory are proposed, exploiting the space partitioning. In [40] a unified approach that supports a broad class of *top-K pairs queries* (i.e. K-closest pairs queries, K-furthest pairs queries, etc.) is presented. And recently, in In [41] an external-memory algorithm, called *ExactMaxRS*, for the maximizing range sum (MaxRS) problem is proposed. The basic processing scheme of *ExactMaxRS* follows the distribution sweep paradigm, which was introduced as an external version of the plane-sweep algorithm. Moreover, other related problem, the maximizing circular range sum (MaxCRS), is also studied and an approximation algorithm is presented, which uses the *ExactMaxRS* algorithm.

Other complex DJQs using R-trees have been studied in the literature of spatial databases, as *Iceberg Distance Join* [42], *K Nearest Neighbors Join* [43] queries, and closely related to DJQ processing is the All-Nearest-Neighbor (ANN) query [44]. For a more detailed review of this classification, see [15].

3 Plane-Sweep in Distance-Based Join Queries

An important improvement for join queries is the use of the *plane-sweep* technique, which is a common technique for computing intersections [7]. The *plane-sweep* technique is applied in [8] to find the closest pair in a set of points which resides in main memory. The basic idea, in the context of spatial databases, is to move a line, the so-called *sweepline*, perpendicular to one of the axes, e.g. X-axis, from left to right, and processing objects (points or MBRs) as they are reached by such *sweepline*. We can apply this technique for restricting all possible combinations of pairs of objects from the two data sets. If we do not use this technique, then we must check all possible combinations of pairs of objects from the two data sets and process them. That is, using the *plane-sweep* technique instead of the brute-force nested loop algorithm, the reduction of CPU cost is proven (e.g. for intersection joins [24, 13, 12] and KCPQ [10, 6]).

3.1 Classic Plane-Sweep Algorithm

In general, let's assume that the spatial objects are points. The data sets are $\mathcal{P}$ and $\mathcal{Q}$ and they can be organized as arrays. Let's also consider a distance threshold δ, which is the distance of the K-th pair found so far for the KCPQ (the initial value of δ is ∞), or the constant given maximum distance for the εDJQ. The *Classic Plane-Sweep* (CPS) algorithm consists of the following steps [1, 15]:

1. It sorts the entries of the two arrays of points, based on the coordinates of one of the axes in (e.g. X-axis) in increasing order.
2. After that, two pointers p and q are maintained initially pointing to the first entry for processing of each sorted array of points. Let the *reference* point be the point with the smallest X-value pointed by one of these two pointers, e.g. $\mathcal{P}$, then as *reference* point will be defined the p.
3. Afterwards, the *reference* point must be paired up with the points stored in the other sorted array of points (called *comparison* points, $q \in \mathcal{Q}$) from left to right, satisfying $dx \equiv q.x - p.x < \delta$, processing all *comparison* points as candidate pairs where the *reference* point is fixed. After all possible pairs of entries that contain the *reference* point have been paired up (i.e. the forward lookup stops when $dx \equiv q.x - p.x \geq \delta$ is verified), the pointer of the *reference* array is increased to the next entry, the *reference* point is updated with the point of the next smallest X-value pointed by one of the two pointers, and the process is repeated until one of the sorted array of points is completely processed.

Highlight that *Classic Plane-Sweep* algorithm applies the distance function over the sweeping axis (in this case, the X-axis, dx) because in the *plane-sweep* technique, the sweep is only over one axis. Moreover, the search is only restricted to the closest points with respect to the *reference* point according to the current *distance threshold* (δ). No duplicated pairs are obtained, since the points are always checked over sorted arrays.

Clearly, the application of this technique can be viewed as a *sliding vertical area* on the sweeping axis with a width equal to the δ value starting from the *reference* point (i.e. $[0, \delta]$ in the X-axis), where we only choose all possible pairs of points that can be formed using the *reference* point and the *comparison* points that fall into the current *vertical area* (see Figure 1). This figure shows the points of the data set $\mathcal{P}$ marked with filled circles and the points of the data set $\mathcal{Q}$ marked with empty circles. Their coordinates are shown in, Tables 1 and 2. Note that the ticks on axes are put every two units of length for both dimensions. In the particular instance on Figure 1, a reference point is shown, $p = \{1, (4, 15)\}$, and it is marked by the horizontal arrow with solid line. All points of

Strips	Points {index, (x, y)}			
QS_0	{0,(2,20)}	{1,(7,16)}	{2,(11,4)}	{3,(15,27)}
QS_1	{4,(18.5,30)}	{5,(20,12)}	{6,(21,24)}	{7,(24,6)}
QS_2	{8,(30,9)}	{9,(32,10)}	{10,(36,25)}	{11,(40,6)}

Table 2 The data set $\mathcal{Q}$ with 12 points in X-sorted order.

both sets on the left of p are already processed as $reference$ points. The points Q_1, Q_2 on the right of the reference point according to the CPS (step 2) satisfy the requirement $dx \equiv q.x - p.x < \delta$ (step 3) and they are combined with p to create candidate pairs: the two empty circles located within the gray area which has a width equal to threshold δ. The first point of $\mathcal{Q}$ to the right of p which has dx-distance from p larger than δ, $q = \{3, (15, 27)\}$, is marked by the arrow with dashed line. Once the algorithm reaches this point and calculates the dx-distance it will stop creating pairs with p and continues with the next iteration, setting as reference point $q = \{1, (7, 16)\}$.

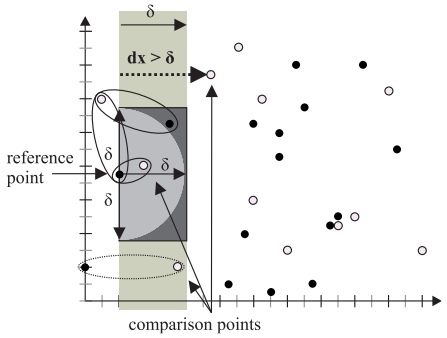

Fig. 1 Classic Plane-Sweep Algorithm using sliding vertical area, window and semi-circle.

3.2 Improving the Classic Plane-Sweep Algorithm

The basic idea to reduce even more the CPU cost is to restrict as much as possible the search space near the *reference* point in order to avoid unnecessary distance computations (that involve *square roots*) which are the most expensive operations for DJQs. The proposed approach makes use of the *plane-sweep* technique and *restricting of the search space*.

The *Classic Plane-Sweep* algorithm applies the distance function only over the sweeping axis (X-axis) and for this reason some distances have to be computed even when the points of the other data set are faraway from the *reference*, since those points are included in the *sliding vertical area* with width δ. Here we will propose two improvements of the *Classic Plane-Sweep* algorithm over two data sets to reduce the number of Euclidean distance computations on KCPQ algorithms.

1. An intuitive way to save distance computations is to bound the other axis (not only the sweeping axis) by δ as is illustrated in Figure 1. In this case, the search space is now restricted to the closest points inside the *window* with width δ and a height $2 * \delta$ (i.e. $[0, \delta]$ in the X-axis and $[-\delta, \delta]$ in the Y-axis, from the *reference* point). Clearly, the application of this technique can be viewed as a *sliding window* on the sweeping axis with a width equal to δ (starting from the *reference* point) and height equal to $2 * \delta$. And we only choose all possible pairs of points that can be formed using the *reference* point and the *comparison* points that fall into the current *window*. For example in Figure 1 it is shown the point $q = \{2, (11, 4)\}$ is outside this window and will not be paired with the $reference$ point p.
2. If we try to reduce even more the search space, we can only select those points inside the *semi-circle* centered at the *reference* point with radius δ (remember that the equation of all points $t = (t.x, t.y) \in E^2$ that fall completely inside the circle, centered at the *reference* point $reference = (reference.x, reference.y) \in \mathcal{P}$ with radius δ is $circle(reference, t, \delta) \equiv (reference.x - t.x)^2 + (reference.y - t.y)^2 < \delta^2$). See Algorithm 1 at lines 11 and 23. For this reason we call this variant *Classic Circle Plane-Sweep* algorithm, $CCPS$ for short. And the application of this new improvement can be viewed as a *sliding semi-circle* with radius δ along the sweeping axis and centered on the *reference* point, choosing only the *comparison* points that fall inside that *semi-circle*. See in Algorithm 1 how this improvement works on two X-sorted arrays of points $PS.P \in \mathcal{P}$ and $QS.P \in \mathcal{Q}$, considering the sweeping axis the X-axis. When a new pair of points (p, q) is chosen, we have to determine whether it will be inserted in the $MaxKHeap$ or not. If the $MaxKHeap$ is not full, we calculate the distance between (p, q) and insert $(dist, p, q)$ unconditionally (lines 6,7 or 18,19). If the $MaxKHeap$ is full, we check

the following condition $dx \equiv q.x - p.x \geq \delta$. If it is true, the process will stop and the new reference point must be defined next (lines 9,10 or 21,22). If not, we check the placement of q. If q is inside the circle (p, δ) the pair is inserted into the heap. The insertion process (lines 11-13 or 23-25) consists of (1) removing the pair with the maximum distance ($keydistofMaxKHeaproot \equiv \delta$), (2) adding the *newPair* and reorganizing the data structure to restore the (binary) max-heap property based on *dist* and (3) updating the value of δ with the new $keydistofMaxKHeaproot$. See in Figure 1, the *semi-circle*, in light grey color, centered at the *reference* point p. This point p will be paired only with the point $q = \{1, (7, 16)\}$. As a conclusion of this improvement is that the smaller the δ value the greater the power of discarding unnecessary *comparison* points to pair up with the *reference* point for computing the DJQ.

The PS and QS structures contain the information needed for processing the $\mathcal{P}$ and $\mathcal{Q}$ data sets in strips, respectively. $PS = \{first, start, end, P[0..n-1]\}$ and $QS = \{first, start, end, P[0..m-1]\}$, where $P[\cdots]$ is a sorted (according to the sweeping axis) array of the maximum number of points per strip, that is, of n and m points of the $\mathcal{P}$ and $\mathcal{Q}$ sets, respectively. We note that n and m values depend on the size of page which may be different for the two sets. The array $P[\cdots]$ may hold one or more pages of points read from the secondary memory; $first$ is the absolute (in relation to the respective data set) index of the first point of this array (used in the algorithms of Section 5); *start* and *end* specify the part of this array that forms the current strip of the respective data set on which a plane sweep algorithm is applied.

In [15], we provide a proof of the correctness of the *Classic Circle Plane-Sweep* algorithm for KCPQ ($CCPS$) algorithm (Algorithm 1) through the Theorem 1.

Theorem 1 (**Correctness**) *Let $PS.P[PS.start \cdots PS.end]$ and $QS.P[QS.start \cdots QS.end]$ be two arrays of points in E^2, sorted in ascending order of X-coordinate values (i.e. X-axis is the sweeping axis), the sweeping direction is from left to right, and $MaxKHeap$ is an initially empty binary max-heap storing K pairs of points, where K is a natural number ($K \in \mathbb{N}, 0 < K \leq |PS.P| \times |QS.P|$). The $CCPS$ Algorithm outputs K closest pairs of points from $PS.P$ and $QS.P$ correctly and without any repetition.*

Moreover, as we know from [24], the *plane-sweep* algorithm for intersection of MBRs from two sets R and S of MBRs can be performed in $O(|R| + |S| + k_X)$, where $|R|$ and $|S|$ are the numbers of MBRs of both sets, and k_X denotes the numbers of pairs of intersecting intervals creating by projecting the MBRs of R and S onto the X-axis. Following the same idea, $CCPS$ can be performed in $O(|PS.P| + |QS.P| + k_{SA})$, where k_{SA} denotes the number of candidate closest pairs generated by the *reference* points from $PS.P$ and $QS.P$ on the sweeping axis (e.g. X-axis).

Algorithm 1 CCPS

Input: PS, QS: structures representing current strips of the X-sorted arrays of points. $MaxKHeap$: Max-Heap storing $K > 0$ pairs
Output: $MaxKHeap$: Max-Heap storing the K closest pairs between $PS.P$ and $QS.P$

1: Set pointers p, q at to the starting points of $PS.P$, $QS.P$
2: **while** last point of $PS.P$ **and** $QS.P$ not reached **do**
3: **if** p is on the left of q **then** ▷ p : reference point
4: **for** $t = q$ **to** the last point of $QS.P$ **do** ▷ get comparison points from QS
5: **if** $MaxKHeap$ is not full **then**
6: $dist = \sqrt{(p.x - t.x)^2 + (p.y - t.y)^2}$
7: Insert pair (p, t) with key $dist$ into $MaxKHeap$
8: **else**
9: **if** $t.x - p.x \geq$ key $dist$ of $MaxKHeap$ root **then**
10: **break**
11: **if** $(p.x - t.x)^2 + (p.y - t.y)^2 < \delta^2$ **then** ▷ key $dist$ of $MaxKHeap$ root $\equiv \delta$
12: $dist = \sqrt{(p.x - t.x)^2 + (p.y - t.y)^2}$
13: Remove root of $MaxKHeap$ insert pair (p, t) with key $dist$ into $MaxKHeap$ and update δ
14: Move p at the next point of $PS.P$
15: **else** ▷ $p \geq q$ and q: reference point
16: **for** $t = p$ **to** the last point of $PS.P$ **do** ▷ get comparison points from PS
17: **if** $MaxKHeap$ is not full **then**
18: $dist = \sqrt{(t.x - q.x)^2 + (t.y - q.y)^2}$
19: Insert pair (t, q) with key $dist$ into $MaxKHeap$
20: **else**
21: **if** $t.x - q.x \geq$ key $dist$ of $MaxKHeap$ root **then**
22: **break**
23: **if** $(t.x - q.x)^2 + (t.y - q.y)^2 < \delta^2$ **then** ▷ key $dist$ of $MaxKHeap$ root $\equiv \delta$
24: $dist = \sqrt{(t.x - q.x)^2 + (t.y - q.y)^2}$
25: Remove root of $MaxKHeap$ insert pair (t, q) with key $dist$ into $MaxKHeap$ and update δ
26: Move q at the next point of $QS.P$

3.3 Extension to εDistance Join Query

The adaptation of the $CCPS$ algorithm from KCPQs to εDJQs is not so difficult, and we get the *Classic Circle Plane-Sweep* algorithm for εDJQ ($\varepsilon CCPS$). If we have two sorted sets of points, we only select the pairs of points in the range of distances $[\varepsilon_1, \varepsilon_2]$ for the final result (lines 11 and 23: **if** ($dist \geq \varepsilon_1$ **and** $dist \leq \varepsilon_2$)). This means the result of this query must not be ordered and the $MaxKHeap$ is unnecessary (lines 5, 6, 7 and 8; and lines 17, 18, 19, and 20), since in the case of εDJQ we do not know beforehand the exact number of pairs of points that belong to the result. And now, the distance threshold will be ε_2 instead of key $dist$ of $MaxKHeap$ root (*line 9*: **if** $(t.x - p.x \geq \varepsilon_2)$, *line 21*: **if** $(t.x - q.x \geq \varepsilon_2)$, *line 11*: **if** $((p.x - t.x)^2 + (p.y - t.y)^2 < (\varepsilon_2)^2)$ and *line 23*: **if** $((t.x - q.x)^2 + (t.y - q.y)^2 < (\varepsilon_2)^2)$). Therefore, the data structure that holds the result set will be a file of records (*resultFile*), with three fields $(dist, p, q)$. The modifications of this storing are in lines 13 and 25, where we have to replace them by $resultFile.write(newPair)$. To accelerate storing on the *resultFile* we maintain a buffer on main memory ($B_{resultFile}$), and when it is full, its content is flushed to disk. If the distance threshold for the query (ε_2) is large enough, the compact representation of the join result can be applied [45]. It consists of reporting groups of nearby pairs of points instead of every join link separately. This phenomenon is known as *output explosion* [45] and it can appear when data density of the sets of points is locally very large compared to the range of distances (distance threshold, ε_2), and the output of the distance-based joins becomes unwieldy. In fact, the output can become quadratic rather than linear in the total number of data points. Finally, the proof of the correctness of $\varepsilon CCPS$ algorithm is similar to the proof of Theorem 1 for the $CCPS$ algorithm (KCPQ).

4 Reverse Run Plane-Sweep Algorithm for Distance Join Queries

An interesting improvement of the *Classic Plane-Sweep* algorithm is the *Reverse Run Plane-Sweep* algorithm, $RRPS$ for short [1]. The main characteristics of this new algorithm are the use of the concept of *run* and, as long as the *reference* points are considered in an order (e.g. ascending order), processing of the *comparison* points in reverse order (e.g. descending order) until a left limit is reached, in order to generate candidate pairs for the required result.

4.1 Reverse Run Plane-Sweep Algorithm for KCPQs

The *Reverse Run Plane-Sweep* ($RRPS$) algorithm [1] is based on two concepts, illustrated in Figure 2. 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. 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 (current *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. Second, the *reference points* (and their *runs*) are processed in ascending X-order (the sets are X-sorted before the application of the $RRPS$ algorithm). Each point of the *active run* is compared with the points of the other set (current *comparison* points) in the opposite or reverse order (descending X-order). Figure 2 depicts a particular instance of the algorithm. We see the data sets $\mathcal{P}, \mathcal{Q}$ with the points of Tables 1 and 2. The current reference point is $q = \{6, (21, 24)\}$, and it is marked by an arrow with solid line. All points of both sets on the left of q have already been processed as $reference$ points. The points P_5, P_4 and P_3 on the left of the reference point according to the $RRPS$ satisfy the requirement $dx \equiv q.x - p.x < \delta$ and they are combined with q to create candidate pairs: the three full circles located within the gray area which has a width equal to threshold δ. The first point of $\mathcal{P}$ to the left of q which has dx-distance from q larger than δ, $p = \{2, (10, 21)\}$, is marked by the arrow with dashed line. Once the algorithm reaches this point and calculates the dx-distance it will stop creating pairs with q also it will update the $leftlimit$ of the data set $\mathcal{P}$ with the point $p = \{2, (10, 21)\}$. The algorithm will continue with the next iteration, setting as reference point $p = \{6, (22, 21)\}$.

The *Reverse Run Circle Plane-Sweep* algorithm for the KCPQ ($RCPS$) is depicted in Algorithm 2: this is the $RRPS$ algorithm with the *sliding semi-circle* improvement. Again, a *binary max-heap* (keyed by pair distances, $dist$), $MaxKHeap$, that keeps the K closest point pairs found so far is used. For each point of the *active run* (*reference* point) being compared with a point of the other set (current *comparison* point) there are 2 cases.

Case 1: If the pair of points *(reference point, comparison point)* is inside the circle centered at the *reference* point with radius δ, then this pair with its distance $dist$ is inserted in the $MaxKHeap$ *(rule 1)*. The insertion process (lines 23-25 or 39-41) consists of (1) removing the pair with the maximum distance

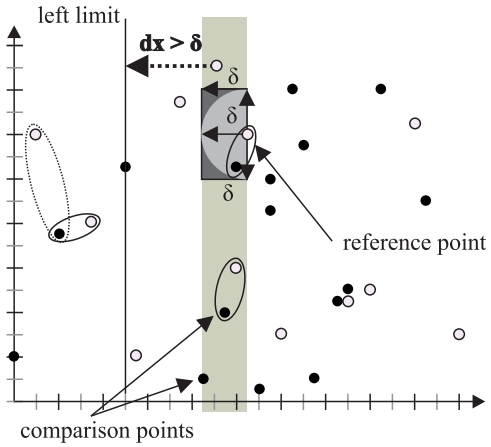

Fig. 2 Reverse Run Plane-Sweep algorithm using sliding strip, window and semi-circle.

($keydistofMaxKHeaproot \equiv \delta$), (2) adding the *newPair* and reorganizing the data structure to restore the (binary) max-heap property based on *dist* and (3) updating the value of δ with the new $keydistofMaxKHeaproot$. In case the heap is not full (it contains less than K pairs), the pair will be inserted in the heap, regardless of the pair distance, *dist*.

Case 2: If the distance between this pair of points in the sweeping axis (e.g. X-axis) dx is larger than or equal to δ, then there is no need to calculate the distance *dist* of the pair *(rule 2)*. The *left limit* of the comparison set must be updated at the point being compared (a comparison with a previous point of the the updated *left limit* will have X-distance larger than dx and is unnecessary).

Moreover, if the rightmost current *comparison* point is equal to the *left limit* of its set, then all the points of the *active run* will have larger dx from all the current *comparison* points of the other set and the relevant pairs need not participate in calculations, i.e. the algorithm advances to the start of the next run *(rule 3)*.

The $RCPS$ algorithm (Algorithm 2) is an enhanced version of Algorithm 1 of [1]. Since the present paper focuses on disk resident data that are gradually transferred and processed in RAM, the $RCPS$ algorithm is applied on strips (sorted subarrays) of data and not on the whole arrays of data, like Algorithm 1 of [1]. In Algorithm 2, p, q are pointers to the current points, and $leftp$ and $leftq$ hold the left limits of the two strips, respectively (in the algorithms of Section 5, $gleftp$ and $gleftq$ are analogous variables that hold the left limits of the whole two data sets, respectively); *stop_run* stores the end-limit of the X-coordinates of the current run of the PS, or QS strip. *run_setP* is set to *false* when $p.x < q.x$ (then the current *active* run will get $reference$ points from the $QS.P$, starting from q, and the *comparison* points will come from the $PS.P$, starting from the previous point of p). Analogously, *run_setP* is set to *true* when $q.x \leq p.x$ (then the current *active* run will get $reference$ points from the $PS.P$, starting from p, and the *comparison* points will come from the $QS.P$, starting from the previous point of q). Note that, since *active runs* always alternate between the data sets, in Algorithm 2, there is no need for an *Else* block to follow the *If* block of lines 11-26 (the execution of code in lines 11-26 should be followed by execution of code in lines 27-42).

In [15], we provide a proof of the correctness of the *Reverse Run Circle Plane-Sweep* algorithm for KCPQ ($RCPS$) algorithm (Algorithm 2) through the Theorem 2.

Theorem 2 (**Correctness**) *Let $PS.P[PS.start \cdots PS.end]$ and $QS.P[QS.start \cdots QS.end]$ be two arrays of points in E^2, sorted in ascending order of X-coordinate values (i.e. X-axis is the sweeping axis), the sweeping direction is from left to right, and $MaxKHeap$ is an initially empty binary max-heap storing K pairs of points, where K is a natural number ($K \in \mathbb{N}, 0 < K \leq |PS.P| \times |QS.P|$). The RCPS Algorithm outputs K closest pairs of points from $PS.P$ and $QS.P$ correctly and without any repetition.*

In [15], an example illustrating the operation of the $RCPS$ algorithm is included (not included here, to limit the size of the present article). Note that, the $CCPS$ algorithm always processes pairs from left to right, even when the distance of the *reference* point to its closest point of the other array is large (this is likely, since, *runs* of the two arrays can be in general interleaved). On the contrary, $RCPS$ processes pairs of points in opposite X-orders, starting from pairs consisting of points that are the closest possible, avoiding further processing of pairs that is guaranteed not to be part of the result and substituting distance calculations by simpler dx calculations, when possible. This way, δ is expected to be updated more fastly and the processing cost of $RCPS$ to be lower. This is verified in the specific example appearing in [15].

Algorithm 2 RCPS

Input: PS, QS: structures representing current strips of the X-sorted arrays of points. $MaxKHeap$: Max-Heap storing $K > 0$ pairs
Output: $MaxKHeap$: Max-Heap storing the K closest pairs between $PS.P$ and $QS.P$

1: Set pointers p, q and local $left\_limits$ at the staring points of $PS.P$ and $QS.P$
2: Define as sentinels the first points of the next strips of the X-sorted arrays of points $QS.P$, $PS.P$
3: Initialize the sentinels to ∞
4: **if** $p.x < q.x$ **then** ▷ find the most left point of two data Sets
5: Initialize p at the first point of $QS.P$ that satisfies $p.x \geq q.x$
6: $stop\_run = p.x$ $run\_SetP =$ FALSE ▷ stop the run of $QS.P$ Set at the start of the 2nd run of the $PS.P$
7: **else**
8: Initialize q at the first point of $PS.P$ that satisfies $q.x > p.x$
9: $stop\_run = q.x$ $run\_SetP =$ TRUE ▷ stop the run of $PS.P$ Set at the start of the 2nd run of the $QS.P$
10: **while** last point of $PS.P$ **or** $QS.P$ not reached **do**
11: **if** $run\_SetP =$ TRUE **then** ▷ the active run is from the PS Set
12: **while** $p.x < stop\_run$ **do** ▷ while active run unfinished. p: reference point
13: **if** previous point of q is equal to $leftq$ **then** ▷ q: last current comparison point - rule 3
14: Move p up to the next $PS.P$-run and **break** ▷ **while**
15: **for** $t = q$ **to** the next point of $leftq$ **do** ▷ t: current comparison point
16: **if** $MaxKHeap$ is not full **then**
17: $dist = \sqrt{(p.x - t.x)^2 + (p.y - t.y)^2}$
18: Insert pair (p, t) with key $dist$ into $MaxKHeap$
19: **else**
20: **if** $p.x - t.x \geq$ key $dist$ of $MaxKHeap$ root **then** ▷ $dx \geq \delta$ - rule 2
21: Update the local and global left $limitq$ up to t
22: **break** ▷ **for**
23: **if** $(p.x - t.x)^2 + (p.y - t.y)^2 < \delta^2$ **then** ▷ key $dist$ of $MaxKHeap$ root $\equiv \delta$ - rule 1
24: $dist = \sqrt{(p.x - t.x)^2 + (p.y - t.y)^2}$
25: Remove root of $MaxKHeap$ insert pair (p, t) with key $dist$ into $MaxKHeap$ and update δ
26: Move p on the next point of $PS.P$
27: Set $sentinel_P.x$ to a value larger than the x-value the last point of $QS.P$ $stop\_run = p.x$ ▷ now the active run is from the QS
28: **while** $q.x \leq stop\_run$ **do** ▷ while active run unfinished. q: reference point
29: **if** previous point of p is equal to $leftp$ **then** ▷ p: last current comparison point - rule 3
30: Move p up to the next $QS.P$-run and **break** ▷ **while**
31: **for** $t = p$ **to** to the next point of $leftp$ **do** ▷ t: current comparison point
32: **if** $MaxKHeap$ is not full **then**
33: $dist = \sqrt{(t.x - q.x)^2 + (t.y - q.y)^2}$
34: Insert pair (t, q) with key $dist$ into $MaxKHeap$
35: **else**
36: **if** $q.x - t.x \geq$ key $dist$ of $MaxKHeap$ root **then** ▷ $dx \geq \delta$ - rule 2
37: Update the local and global left $limitp$ up to t
38: **break** ▷ **for**
39: **if** $(t.x - q.x)^2 + (t.y - q.y)^2 < \delta^2$ **then** ▷ key $dist$ of $MaxKHeap$ root $\equiv \delta$ - rule 1
40: $dist = \sqrt{(t.x - q.x)^2 + (t.y - q.y)^2}$
41: Remove root of $MaxKHeap$ insert pair (t, q) with key $dist$ into $MaxKHeap$ and update δ
42: Move q on the next point of $QS.P$
43: $sentinel_P.x = \infty$
44: $stop\_run = q.x$ $run\_SetP =$ TRUE

4.2 Extension to εDistance Join Query

Like adapting $CCPS$ to εDJQs ($\varepsilon CCPS$), the adaptation of the $RCPS$ algorithm from KCPQs to εDJQs ($\varepsilon RCPS$) is quite straightforward. If we have two sorted arrays of points, we only select the pairs of points in the range of distances $[\varepsilon_1, \varepsilon_2]$ for the final result (lines 23 and 39: **if** ($dist \geq \varepsilon_1$ **and** $dist \leq \varepsilon_2$)). Since, the result of this query need not be ordered, $MaxKHeap$ is unnecessary (lines 16, 17, 18 and 19; and lines 32, 33, 34, and 35 can be omitted). Now the distance threshold will be ε_2 instead of *key dist of $MaxKHeap$ root* (lines 20, 36, 23 and 39). Like $\varepsilon CCPS$, the data structure that holds the result set will be a file of records (*resultFile*), with three fields ($dist, PS.P[i], QS.P[j]$) and lines 25 and 41 should be replaced by $resultFile.write(newPair)$. Finally, the proof of the correctness of $\varepsilon RCPS$ algorithm is similar to the proof of Theorem 2 for the $RCPS$ algorithm.

5 External Sweeping-Based Distance Join Algorithms

Firstly, we present in this section four new algorithms to solve the problem of finding the KCPQ when neither of the inputs are indexed, following similar ideas proposed in [13,14] for spatial intersection join. We combine *plane-sweep* and *space partitioning* to join the data sets and report the required result. These new algorithms extend the $CCPS$ and $RRPS$ algorithms to solve the KCPQ where the two set of points are stored on separate data files on disk. Moreover, we will also extend them to solve the εDistance Join Query (εDJQ).

5.1 The External Sweeping-Based KCPQ Algorithms

In general, the External Sweeping-Based KCPQ algorithms sort the data files containing the sets of points, then perform the Plane-Sweep-Based KCPQ algorithm on the two sorted disk-resident data files and, finally, return the K closest pairs of points in $maxKHeap$ data structure.

Sorting each data file by the values of the sweeping axis can be done with the classical external sort/merge algorithm [46]. For instance, to sort $\mathcal{P}$ on the X-axis, first $\mathcal{P}$ is partitioned in $\lceil \mathcal{P}/B \rceil$ runs (where B is the size of a buffer in main memory); each run is sorted in main memory; and finally the runs are recursively merged in larger runs, obtaining the sorted file $\mathcal{P}$.

The External Sweeping-Based KCPQ algorithms start with the two sorted data files ($\mathcal{P}$ and $\mathcal{Q}$) and then, as in the *Scalable Sweeping-Based Spatial Join* [13,14], divide the sweeping axis on a set of *strips*. As is defined Section 3.2, we maintain two strips, PS and QS, one for each file, in main memory, for applying the Plane-Sweep-Based KCPQ algorithm ($CCPS$ or $RRPS$) and return the K closest pairs of points from $\mathcal{P}$ and $\mathcal{Q}$ on the $maxKHeap$ data structure. While strips are filled with data reading the pre-defined number of pages of points from secondary memory into $PS.P$ and $QS.P$ arrays, the External Sweeping-Based KCPQ algorithms call repetitively *CCPS / RCPS* with a possibly non empty heap and with, in general, different *PS.start / PS.end* and *QS.start / QS.end* limits and different $PS.P$ and $QS.P$ arrays. At the end of all such calls, the heap will host K closest pairs formed from the two data sets.

Once the data sets are sorted, one can think about: (i) partitioning policies on the sweeping axis and (ii) the appropriate number of strips ($numOfStrips$). We could consider two basic strategies for partitioning the sweeping axis:

1. **Uniform Filling**. A strip hosts a number of points that fit in one or more disk-pages. Using the disk-page size, we calculate the number of points that fit in each strip and divide the data of each set into equally populated *numOfStrips (= data file size / strip size)* strips (with a possibly underfilled last strip). Thus, *numOfStrips* is different for each set.
2. **Uniform Splitting**. We partition the sweeping axis to a number of strips (or intervals) covering, every time, the same interval on the sweeping axis for both data sets. To accomplish this, we use a part of main memory as a buffer for $PS.P$ and $QS.P$ arrays (equal to a pre-defined number of disk pages for each set) and load it with points. Next, a synchronization process takes place. We compare the X coordinates (w.l.o.g. we consider that X is the sweeping axis) of the last points of the two arrays of the sets. The smallest coordinate is set as the right *border* of the current two strips and the points of the other set (not the one where the point with the smallest X coordinate belongs) that are located after the right border (have greater value of X coordinate) are left to be examined and processed in the future. Thus, the strip for each set contains the points of this set up to the right border. In this way, after the first iteration, the data examined are located in an X interval with specified limits. Subsequently, we process the points residing in the two strips. Next, we load from secondary to main memory data points from any of the sets which does not have any points left unprocessed and we repeat the synchronization between the points of the two sets that are located in main memory. Of course, null strips could be created in some cases, but only for one of the two data sets at every iteration. This situation is not problematic, however. It helps prune pairs that will not be part of the result.

As we can see in Figures 3 and 7, for each set, the search space is partitioned to non-overlapping vertical *strips*, whatever the partition policy. We assign each point of $\mathcal{P}$ and $\mathcal{Q}$ to one (and only one) strip. This is a very important condition for the correctness of the algorithms, because, in this way, the same pair cannot be generated twice.

5.2 Algorithms using Uniform Filling

*5.2.1 The **FCCPS** Algorithm*

Following the *Uniform Filling* partitioning policy, the two sorted data sets $\mathcal{P}$ and $\mathcal{Q}$ are partitioned in strips equally full. W.l.o.g let's consider strips and pages that have equal sizes, as we can see in Figure 3. The first sorted set ($\mathcal{P}$) is partitioned in four strips (PS_0, PS_1, PS_2, and PS_3). The second sorted set ($\mathcal{Q}$) is partitioned in three strips (QS_0, QS_1, and QS_2).

The $FCCPS$ algorithm, see Algorithm 3, requires every time two strips, one from each data set, to be present in the main memory. Starting the first iteration of the algorithm we load one page from each set, $\mathcal{P}$ and $\mathcal{Q}$. Since every strip corresponds to a page, we have the two strips PS_0 and QS_0 in main memory. These two strips are the current strips. One of the current strips will be set as the $reference$ strip, that is, the strip with the leftmost $first$ point; and the other one as a *comparison* strip.

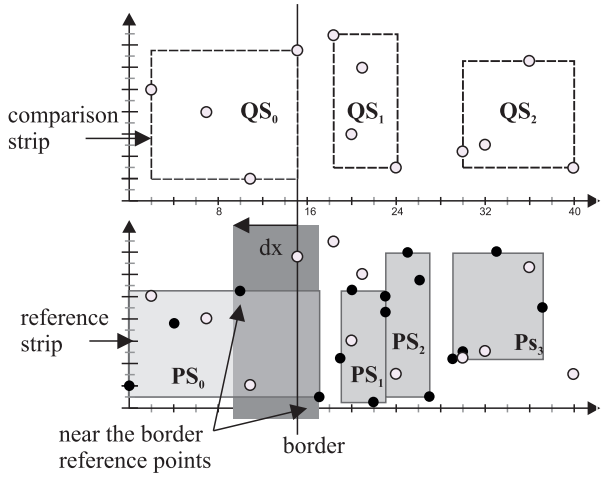

Fig. 3 Applying the $FCCPS$ algorithm on two data sets partitioned in strips equally full (4 points/strip).

The process is starting by loading the first two strips PS_0 and QS_0. In the first step we set the leftmost strip (PS_0) as the $reference$ strip, the other strip (QS_0) as a *comparison* strip (as it is shown in Figure 3; lines 6 and 20 of Algorithm 3). Next we examine the K closest pairs in these strips by using the $ClassicPlaneSweep$ ($CCPS$) algorithm at lines 8 and 22, during the first iteration of while-loop at lines 7 and 21, respectively, of Algorithm 3.

In the second step we must examine the points near the border (i.e. the coordinate on the sweeping axis of the last point of the current *comparison* strip) with the next *comparison* strip. If $maxKHeap$ is not full, all the points of the $reference$ strip (PS_0) must be joined with the next *comparison* strip (QS_1). If $maxKHeap$ is full, we must check the points of the $reference$ strip which have dx distance from the *border* smaller than the key *dist* of $maxKHeap$ root. In Figure 3 we can see the *border* after the join between PS_0 and QS_0, and the points of the $reference$ strip (the two *last* points) which are near the *border* in the dark gray area. Then we load in main memory the next *comparison* strip (QS_1) to continue searching the K closest pairs between the PS_0 and QS_1. After the join between the $reference$ strip (PS_0) and the *comparison* strip (QS_1) we update the *border* with a new value, because of a new last point of the current *comparison* strip. The process will continue by loading a new *comparison* strip (QS_2) as long as we have strips in the *comparison* set ($\mathcal{Q}$) and the $maxKHeap$ is not full or there is at least one point of the $reference$ strip near the *border*. This step is implemented by lines 7-17 and 21-31 in the Algorithm 3.

In the third step, we will load in main memory the next page which corresponds to the next strip PS_1 of the $reference$ set $\mathcal{P}$ as one of the current strips. The pair of current strips in the new iteration will consist of PS_1 and QS_0 and the process will be restarted (from the first step) by examining which of the two current strips of the sets is the left most one. This step is implemented at lines 18 and 32 in the Algorithm 3.

We must also highlight that in Algorithm 3, TS is a temporary strip which sometimes is loaded with points of the $\mathcal{P}$ set and other times of the $\mathcal{Q}$ set. We use this strip to read the sequence of the next (for the $CCPS$ algorithm) or the previous (for the $RRPS$ algorithm) points of the *current* strip which must give us *comparison* points. Moreover, the function $check\_near\_border(border, reference\_strip)$ discovers the first point of the $reference\_strip$ which has dx smaller than δ from the (right) *border*, for a more detailed algorithmic presentation see [15].

*5.2.2 The **FRCPS** Algorithm*

For the $FRCPS$ algorithm, see Algorithm 4, we scan the strips in a different order to the previous algorithm ($FCCPS$). The $reference$ strips are scanned in the same order in which the points of the data sets are sorted (i.e. in ascending order in X-axis), but the *comparison* strips are scanned in the opposite order (i.e. in descending order in X-axis). In this way, we continue to apply the basic concept of the $RRPS$ algorithm. If A is a $reference$ point from the one data set and B, C (with $B.x > C.x$) are *comparison* points from the other data set and moreover: (i) $A.x > B.x$, that is the reference points are always on the right of the *comparison* points (ii) The points B and C are adjacent to the X-axis (no other item of the same set lies between them), then we first calculate the distance of the pair (A, B) and next the distance of the pair (A, C). Unlike the previous algorithm ($FCCSP$), now we have every time in main memory four strips, two from each data set. The leftmost strip of each data set will be defined as *current* and the other as *next* (of the *current* strip). So we have two pairs of strips, the *current* pair and the *next* pair.

Algorithm 3 FCCPS

Input: Two X-sorted files of points $\mathcal{P}$ and $\mathcal{Q}$, $|\mathcal{P}| = N$, $|\mathcal{Q}| = M$. $MaxKHeap$: Max-Heap storing $K > 0$ pairs
Output: $MaxKHeap$: Max-Heap storing the K closest pairs between $\mathcal{P}$ and $\mathcal{Q}$

```
1: Allocate memory for strips PS, QS, TS
2: border is a local variable to hold the right border of the calculated strip so far
3: Read from LRU Buffer pages for the first strips of P, Q into PS.P, QS.P          ▷ a strip corresponds to one or more pages
4: while both sets have points not processed do
5:     if first point of PS.P is on the left of first point of QS.P then
6:         TS ← QS                                          ▷ temporary strip TS is loaded with current strip of Q
7:         while TRUE do
8:             CCPS(PS, TS)
9:             if all the points of Q are not processed then
10:                Update border with x-value of the last point of TS.P
11:                check_near_border(border, PS)
12:                if points of PS.P reside near the border then
13:                    Read from LRU Buffer pages for the next strip of Q into TS.P
14:                else
15:                    break                                ▷ all the rest points are too far from the border
16:            else
17:                break                                    ▷ end of the set Q
18:        Read from LRU Buffer pages for the next strip of P into PS.P
19:    else                ▷ first point of QS.P has x-coordinate equal or smaller than the first point of PS.P
20:        TS ← PS                                          ▷ temporary strip TS is loaded with current strip of P
21:        while TRUE do
22:            CCPS(TS, QS)
23:            if all the points of set P are not processed then
24:                Update border with x-value of the last point of TS
25:                check_near_border(border, QS)
26:                if points of QS.P reside near the border then
27:                    Read from LRU Buffer pages for the next strip of P into TS.P
28:                else
29:                    break                                ▷ all the rest points are too far from the border
30:            else
31:                break                                    ▷ end of the set P
32:        Read from LRU Buffer pages for the next strip of set Q into QS.P
```

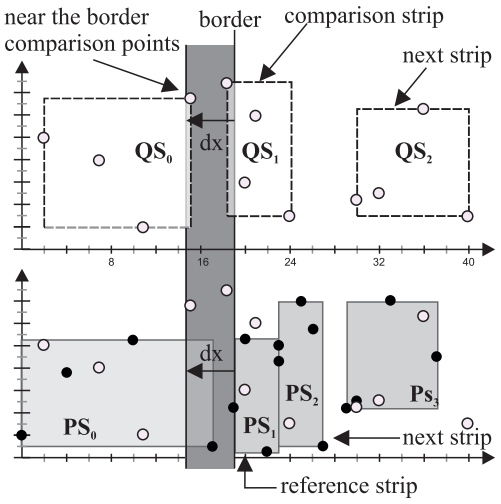

Fig. 4 Applying the $FRCPS$ algorithm on two data sets partitioned in strips equally full (4 points/strip).

As it shown in Figure 4, during the execution of the algorithm, we can have as *current* pair the strips PS_1 and QS_1 and *next* pair the strips PS_2 and QS_2 (denoted by nPS and nQS, respectively, in Algorithm 4).

In the first step of this iteration, we join the strips of the *current* pair (PS_1 and QS_1). From the *current* pair, we will set as $reference$ strip, the strip which has the rightmost $first$ point (PS_1) and the other strip will be set as *comparison* strip (QS_1). This step is implemented by lines 13-18 and 25-30 of Algorithm 4.

In the second step, while the left limit is outside of the *comparison* strip, we will load the previous strip (QS_0) of the *current comparison* strip and we make the join between the strips PS_1 and QS_0. This loop will continue until the left limit will be reached inside the *comparison* strip. This step is implemented by lines 19-23 and 31-35 of Algorithm 4.

The third step is to prepare the new pair of the current strips. One of the strips of the *current* pair will be replaced by one strip of the *next* pair. The leftmost of the strips of the *next* pair will moved from the *next* pair

to the *current* pair, and this strip will be replaced by a new strip which will be loaded from secondary memory. This step is implemented by lines 36-47 of the Algorithm 4.

We must also highlight that in Algorithm 4, $gleftp$ and $gleftq$ are variables that hold global left limits for the sorted sets $\mathcal{P}$ and $\mathcal{Q}$. $leftlim$ is local variable that saves the old values of $gleftp$ and $gleftq$ (previous strips). nPS and nQS are the next strips of (the current strips) PS and QS. For a more detailed presentation of Algorithm 4, see [15].

Algorithm 4 FRCPS

Input: Two X-sorted files of points $\mathcal{P}$ and $\mathcal{Q}$, $|\mathcal{P}| = N$, $|\mathcal{Q}| = M$. $MaxKHeap$: Max-Heap storing $K > 0$ pairs
Output: $MaxKHeap$: Max-Heap storing the K closest pairs between $\mathcal{P}$ and $\mathcal{Q}$

1: Allocate memory for strips PS, QS, TS, nPS, nQS
2: Initialize the left limits at a non existing point on the left of two sets
3: Read from LRU Buffer pages for the first strips of sets $\mathcal{P}, \mathcal{Q}$ into $PS.P, QS.P$
4: **if** the first point of $PS.P$ is on the left of the first point of $QS.P$ **then**
5: **while** all the points of set $\mathcal{P}$ are not processed **and** the last point of $PS.P$ is on the left of the first point of $QS.P$ **do**
6: Read from LRU Buffer pages for the next strip of set $\mathcal{P}$ into $PS.P$
7: **else**
8: **while** all the points of set $\mathcal{Q}$ are not processed **and** the last point of $QS.P$ is on the left of the first point of $PS.P$ **do**
9: Read from LRU Buffer pages for the next strip of set $\mathcal{Q}$ into $QS.P$
10: Read from LRU Buffer pages for the next strips of sets $\mathcal{P}, \mathcal{Q}$ into $nPS.P, nQS.P$
11: **while** $gleftp$ differs to the last point of $\mathcal{P}$ **and** $gleftq$ differs to the last point of $\mathcal{Q}$ **do**
12: **if** first point of $PS.P$ is on the left of first point of $QS.P$ **then**
13: **if** $gleftp$ is on the right of the first point of $PS.P$ **then**
14: Update $PS.start$ with the index of the next point of $gleftp$
15: **if** $PS.start \leq PS.end$ **then**
16: **RCPS**(PS, QS)
17: **else**
18: **RCPS**(PS, QS)
19: $leftlim = gleftp$ Update the first point of $TS.P$ with the first point of $PS.P$
20: **while** ($MaxKHeap$ is not full **or** (dx-distance b/t first points of $QS.P, TS.P <$ key $dist$ of $MaxKHeap$ root)) **do**
21: Read from LRU Buffer pages for the previous strip of set $\mathcal{P}$ into $TS.P$
22: **RCPS**(TS, QS)
23: $gleftp = leftlim$
24: **else** ▷ first point of $PS.P$ is not the left of first point of $QS.P$
25: **if** $gleftq$ is on the right of the first point of QS **then**
26: Update $QS.start$ with the index of the next point of $gleftq$
27: **if** $QS.start \leq QS.end$ **then**
28: **RCPS**(PS, QS)
29: **else**
30: **RCPS**(PS, QS)
31: $leftlim = gleftq$ Update the first point of $TS.P$ with the first point of $QS.P$
32: **while** ($MaxKHeap$ is not full **or** (dx-distance b/t first points of $PS.P, TS.P <$ key $dist$ of $MaxKHeap$ root)) **do**
33: Read from LRU Buffer pages for the previous strip of set $\mathcal{Q}$ into $TS.P$
34: **RCPS**(PS, TS)
35: $gleftq = leftlim$
36: **if** all the points of set $\mathcal{P}$ are processed **then**
37: **if** all the points of set $\mathcal{Q}$ are processed **then**
38: **break** ▷ end of sets, terminate the process
39: $QS \leftarrow nQS$ ▷ the next strip of $\mathcal{Q}$ becomes current
40: Read from LRU Buffer pages for the next strip of set $\mathcal{Q}$ into $nQS.P$
41: **else** ▷ all the points of set $\mathcal{P}$ are not processed
42: **if** $nQS.first \neq M$ **and** first point of $QS.P$ is on the left of first point of nPS **then**
43: $QS \leftarrow nQS$ ▷ the next strip of $\mathcal{Q}$ becomes current
44: Read from LRU Buffer pages for the next strip of set $\mathcal{Q}$ into $nQS.P$
45: **else**
46: $PS \leftarrow nPS$ ▷ the next strip of $\mathcal{Q}$ becomes current
47: Read from LRU Buffer pages for the next strip of set $\mathcal{P}$ into $nPS.P$

Now, we are going to show a step-by-step example of the application of the $FRCPS$ algorithm to find the $K(=3)$ closest pair of the data sets $\mathcal{P}$ and $\mathcal{Q}$ having 16 and 12 points, respectively. We also consider that the maximum number of points per strip is 4 and every page from disk can host the same number of points (4). The data sets and the separation into strips are shown in Tables 1 and 2 and in Figure 3.

The $FRCPS$ algorithm firstly reads the strips: $PS_0\{first = 0, start = 0, end = 3, P[\mathbf{0,1,2,3}]\}$, $QS_0\{first = 0, start = 0, end = 3, P[\mathbf{0,1,2,3}]\}$ as current strips and $PS_1\{first = 4, start = 0, end = 3, P[\mathbf{4,5,6,7}]\}$, $QS_1\{first = 4, start = 0, end = 3, P[\mathbf{4,5,6,7}]\}$ as *next* strips (see Figure 5). Both left limits ($leftp$ and $leftq$) are initialized to non existing point on the left of two sets: $leftp = lefpq = \{-1, (-1, 0)\}$.

The function using the algorithm $RCPS$ executes the $K(=3)$CPQ for the strips PS_0 and QS_0. Finishing this join the $maxKHeap$ has the pairs $\{(dist(P_2, Q_1) = 5.831), (dist(P_1, Q_0) = 5.385), (dist(P_1, Q_1) = 3.162)\}$, where $dist(P_i, Q_j)$ is the distance $dist$ between the points ($P[i]$ and $Q[j]$) from sets $\mathcal{P}$ and $\mathcal{Q}$, having absolute indexes in their sets i and j respectively (regardless of the strip in which they are located), and values for left limits $leftp = P_1$, $leftq = Q_2$.

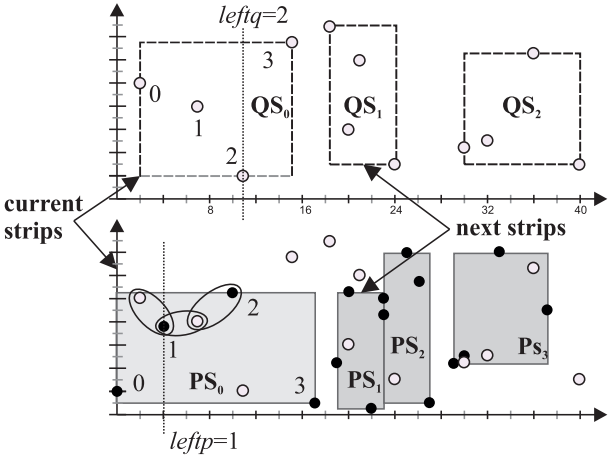

Fig. 5 Join of strips PS_0 and QS_0 using the $FRCPS$ algorithm.

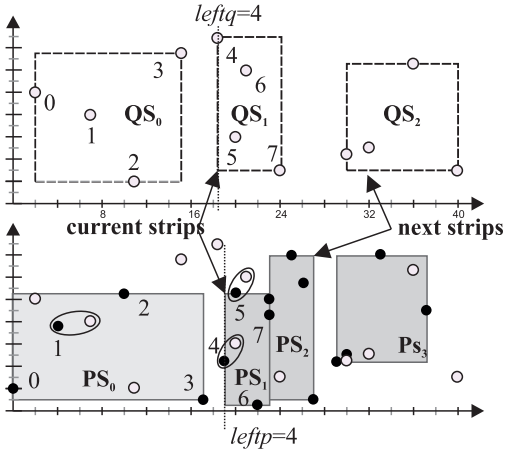

Fig. 6 Join of strips PS_1 and QS_1 using the $FRCPS$ algorithm.

In this first iteration there are no strips on the left of the current strips, so we skip the second step and we are going to execute the third step of the algorithm. In order to prepare the next cycle, the algorithm compares the X-coordinates of the $first$ points of the $next$ strips $PS_1.P[4].x = 19$ and $QS_1.P[4].x = 18.5$. Since the point $QS_1.P[4]$ is on the left, the strip QS_2 is read.

For the second iteration, we have that $PS_0\{first = 0, start = 2, end = 3, P[0, 1, \mathbf{2,3}]\}$, $QS_1\{first = 4, start = 0, end = 3, P[\mathbf{4,5,6,7}]\}$ are the current strips, and $PS_1\{first = 4, start = 0, end = 3, P[\mathbf{4,5,6,7}]\}$, $QS_2\{first = 8, start = 0, end = 3, P[\mathbf{8,9,10,11}]\}$ are the $next$ strips.

The $RCPS$ executes the K(=3)CPQ for the current strips. Note that the $current$ strip PS_0 is starting from the point $PS_0.P[2]$ because of the $leftp = P_1$ value from the previous iteration. Exiting from $RCPS$ function, no new pair is inserted into $maxKHeap$, but the left limits are updated to $leftp = P_3$, $leftq = Q_2$. In order to prepare the next cycle, the algorithm compares the $first$ points of the $next$ strips, $PS_1.P[4].x = 19$ and $QS_2.P[8].x = 30$. Since the point $PS_1.P[4]$ is on the left, the strip PS_2 is read.

For the third iteration, we have that $PS_1\{first = 4, start = 0, end = 3, P[\mathbf{4,5,6,7}]\}$, $QS_1\{first = 4, start = 0, end = 3, P[\mathbf{4,5,6,7}]\}$ are the current strips, and $PS_2\{first = 8, start = 0, end = 3, P[\mathbf{8,9,10,11}]\}$, $QS_2\{first = 8, start = 0, end = 3, P[\mathbf{8,9,10,11}]\}$ are the $next$ strips (see Figure 6).

The $RCPS$ executes the K(=3)CPQ for the current strips. Exiting from $RCPS$ function, the $maxKHeap$ has now the pairs $\{(dist(P_4, Q_5) = 4.123), (dist(P_5, Q_6) = 3.162), (dist(P_1, Q_1) = 3.162)\}$ and $leftp = P_4$, $leftq = Q_4$. Since the dx distance between points $PS_1.P[4]$ and $QS_1.P[4]$ is $dx(P_4, Q_4) = 19 - 18.5 = 0.5 < 4.123$, the algorithm continues checking the points near the left border. The $RCPS$ is called to join the strips PS_1 and QS_0. No new pair is inserted into $maxKHeap$. Since the point $PS_2.P[8]$ is on the left of the point $QS_2.P[8]$, the strip PS_3 is read.

For the forth iteration, we have that $PS_2\{first = 8, start = 0, end = 3, P[\mathbf{8,9,10,11}]\}$, $QS_1\{first = 4, start = 0, end = 3, P[\mathbf{4,5,6,7}]\}$ are the current strips, and $PS_3\{first = 12, start = 0, end = 3, P[\mathbf{12,13,14,15}]\}$, $QS_2\{first = 8, start = 0, end = 3, P[\mathbf{8,9,10,11}]\}$ are the *next* strips.

The $RCPS$ executes the $K(=3)$CPQ for the current strips. Exiting from $RCPS$ function, the $maxKHeap$ has no changes, but the left limit of the $\mathcal{Q}$ set is updated to $leftq = Q_6$. Since the dx distance between the ($first$) points $PS_2.P_8$ and $QS_1.P_4$ is $dx(P_8, Q_4) = 23 - 18.5 = 4.5 > 4.123$, the algorithm has no need to continues checking the points near the left border.

Now, the data set $\mathcal{P}$ has no *next* strip (it is finished) and, then the status for the fifth cycle is as follow: $PS_2\{first = 12, start = 0, end = 3, P[\mathbf{12,13,14,15}]\}$, $QS_1\{first = 4, start = 3, end = 3, P[4, 5, 6, \mathbf{7}]\}$ are the current strips and only $QS_2\{first = 8, start = 0, end = 3, P[\mathbf{8,9,10,11}]\}$ is the *next* strip.

The $RCPS$ executes the $K(=3)$CPQ for the current strips. Exiting from $RCPS$ function, the $maxKHeap$ has no changes, but the left limit of the $\mathcal{Q}$ set is updated to $leftq = Q_7$. The data set $\mathcal{P}$ has no *next* strip (it is finished), then the status for the sixth cycle is as follows: $PS_2\{first = 12, start = 0, end = 3, P[\mathbf{12,13,14,15}]\}$, $QS_2\{first = 8, start = 0, end = 3, P[\mathbf{8,9,10,11}]\}$ are the current strips and there is not any *next* strip.

Finally, the $RCPS$ executes the $K(=3)$CPQ for the current strips. Exiting from $RCPS$ function, the $maxKHeap$ has new pairs $\{(dist(P_{12}, Q_8) = 1.000), (dist(P_{13}, Q_8) = 1.000), (dist(P_{13}, Q_9) = 2.000)\}$ and the left limits are updated to $leftp = P_{15}$ and $leftq = Q_9$. Since the dx distance between points $PS_3.P_{12}$ and $QS_2.P_8$ is $dx(P_{12}, Q_8) = |29 - 30| = 1 < 2.0$, the algorithm will continue by checking the points near the left border between the strips PS_2 and QS_2. But, no new pair is found and the algorithm is finished.

As a summary, the pages which are read from disk were 9, the pairs involved in calculations were 57, the dx calculations were 89 and the complete $dist$-calculations were 10.

5.3 Algorithms using Uniform Splitting

*5.3.1 The **SCCPS** Algorithm*

Following the Uniform Splitting partitioning policy, the first sorted set ($\mathcal{P}$) is partitioned in five strips (PS_0, PS_1, PS_2, PS_3 and PS_4). The second sorted set ($\mathcal{Q}$) is partitioned in seven strips (QS_0, QS_1, QS_2, QS_3, QS_4 and QS_5).

The $SCCPS$ algorithm, see Algorithm 5, requires two strips, one of each data set, to be present in main memory. We define the *width* of a strip as the distance between the leftmost ($first$) and rightmost ($last$) points of the strip on the sweeping axis. After loading a buffer of disk pages from secondary memory with points from the two data sets into strip arrays $\mathcal{P}$, we have to execute a *synchronization* process (through *sync_queues* function). This process determines the points in the two arrays that form the respective two strips. Note that every point between the leftmost and rightmost points of both strips has been read from secondary memory. The coordinate of the rightmost point of the strips is defined as *border*.

We examine the coordinates on the sweeping axis (i.e. X-axis) of the *last* points of the current arrays $PS.P$ and $QS.P$. As it is shown in Figure 7 the strip PS_0 has the array of points with indexes $P = [0, 1, 2, 3]$ which are depicted with filled circles. The strip QS_0 has the array of points with indexes $P = [0, 1, 2, 3]$ which are depicted with empty circles. The *last* point $QS_0.P[3]$ is on the left of the *last* point $PS_0.P[3]$ ($QS_0.P[QS_0.end].x < PS_0.P[PS_0.end]$). Since, it is not known if the $first$ point of the $\mathcal{Q}$ set next to the *last* point of the QS_0 strip (the point $QS_1.P[4]$) is on the left or on the right of the *last* point of the *current* PS_0 page, we set as right *border* the coordinate on the sweeping axis of the *last* point of the QS page ($QS_0.P[3].x$). In this way, at least one strip (QS_0) will have the maximum number of points per strip while the other strip (PS_0) will have points from zero to the maximum number of points per strip (as we can see in Figure 7 the PS_0 strip has three points).

The process starts loading pages of points into $PS_0.P$ and $QS_0.P$. After the *synchronization* process we have two strips and the value of the *border = border1*. If both strips have some points (are not empty) we examine the K closest pairs of points inside these strips by using the Classic Plane Sweep ($CCPS$). This first step is implemented by lines 5-6 of Algorithm 5.

The second step is to examine in any not empty strip, first PS and next QS the points near the *border*. If the $maxKHeap$ is full only the points that reside near the border, having dx-distance from the border smaller than the key $dist$ of $maxKHeap$ root, will be selected for joins. If the $maxKHeap$ is not full all the points of the current strips will be eligible for joins. First, we must join the points of the PS strip near the *border* with the points of the QS strip that have not been joined with the points of the PS strip in the previous first step. Then we must update the value of the *border* with the coordinate of the *last* point of the QS_0 strip, find the eligible points of the PS_0 strip taking into account the new value of the border. If there are some points left, we must continue by loading the next page of the $\mathcal{Q}$ set (QS_1). The process will continue as long as we have a strip in the *comparison* set ($\mathcal{Q}$) and there is at least one point of the $reference$ strip (PS_0) near the current value of the *border*. This

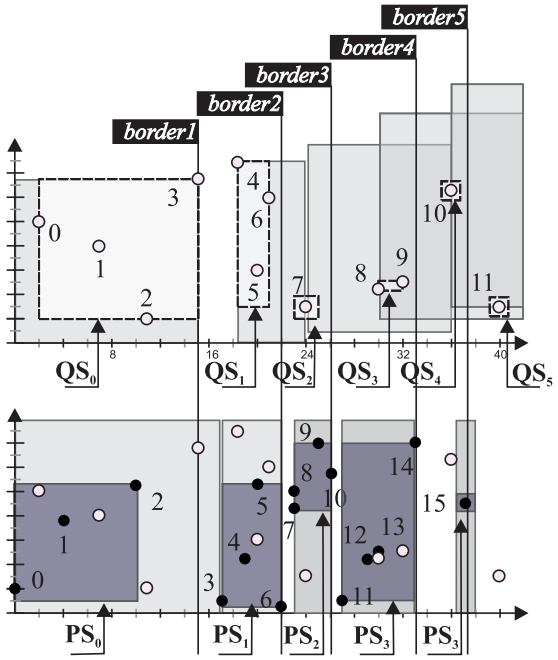

Fig. 7 Applying the *SCCPS* algorithm on two data sets partitioned in strips of variable *width*.

second step will be executed setting as $reference$ strip the QS_0 and *comparison* strips the rest of the points of PS_0, PS_1, $\cdots$. This step is implemented by lines 7-39 of Algorithm 5.

The third step is to prepare the next pair of strips (PS_1 and QS_1) by loading pages of points from secondary memory into arrays $\mathcal{P}$, *synchronizing* them and continuing from the first step as long as we have points for both strips. This step is implemented by lines 40-44 of Algorithm 5.

We must also highlight that in Algorithm 5, the function $sync\_queues(PS, QS)$ finds which of the last points of the two strips is the leftmost one. Then it sets the value of the right border equal to the X-coordinate of this point. Finally, it returns the value of the right border. *border* is a variable that holds the right border of the current strips. *cur_border* is a local variable of the if-structure in lines 7-23 that holds the updated value of the current border of the current comparison strip. For the other if-structure in lines 24-39 the variable *border* holds the updated value of the current border. For a more detailed presentation of Algorithm 5, see [15].

*5.3.2 The **SRCPS** Algorithm*

The $SRCPS$ algorithm, see Algorithm 6, requires two strips, one of each data set, to be present in main memory. Before the main process of this algorithm and for the leftmost set we reach either the first strip which has *overlap* with the first strip of the other set, or the last strip (which has no *overlap* with the first strip of the other set); lines 5-12 of Algorithm 6.

The first step is to synchronize the current strips (if both are not empty) and afterwards the $RRPS$ algorithm is called to join the points between them. This step is implemented by lines 14-15 and 16-19 of Algorithm 6.

The second step consists of two parts. In the first part, we examine three conditions: (1) if the strip of the first set $\mathcal{P}$ has at least one point in the area on left of the right border (see section 5.3.1), (2) if the current strip of the other set $\mathcal{Q}$ has points on the left of its starting point (in the same strip or in previous strips), and (3) if the $maxKHeap$ is not full or if the first point of the PS_i strip has a distance on the sweeping axis (dx) from the left border (the coordinate of the last point of the previous strip of QS_j) less than the *key dist of maxKHeap root* (line 16 of Algorithm 6). If all conditions are true then we call the subroutine *srcps_on_border* (Algorithm is presented in [15]). In this subroutine we join the points of the strip PS and all points of the set $\mathcal{Q}$ which are on the left of the starting point of the current QS strip. This process continues while the $maxKHeap$ is not full or the points have dx distance from the left border smaller than the *key dist of maxKHeap root*. For each set, we keep a left limit ($leftp$, $leftq$), 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. In Figure 8 we can see the dx distance of the first point of the PS_1 strip from the *lborderq* which is smaller than the key *dist* of $maxKHeap$ root. In the second part, we swap the roles between PS and QS and we execute the same process as in the first part. This step is implemented by lines 24-27 of Algorithm 6.

Algorithm 5 SCCPS

Input: Two X-sorted files of points $\mathcal{P}$ and $\mathcal{Q}$, $|\mathcal{P}| = N$, $|\mathcal{Q}| = M$. $MaxKHeap$: Max-Heap storing $K > 0$ pairs
Output: $MaxKHeap$: Max-Heap storing the K closest pairs between $\mathcal{P}$ and $\mathcal{Q}$

```
1: Allocate memory for strips PS, QS, TS
2: Read from LRU Buffer pages for the first strips of sets P, Q into PS.P, QS.P
3: border = sync_queues(PS, QS)                    ▷ determine the points in the arrays that form the respective strips
4: while TRUE do
5:     if both strips PS, QS have points up or on the left of the border then
6:         CCPS(PS, QS)
7:     if points of strip PS reside near the border then
8:         cur_border = border
9:         check_near_border(cur_border, PS)
10:        if points of strip PS reside near the border then
11:            TS ← QS                                 ▷ the next strip of Q becomes current
12:            Update the strip TS to join with the rest points of QS
13:            while TRUE do
14:                CCPS(PS, TS)
15:                if all the points of set Q are not processed then
16:                    Update cur_border with the x-coordinate of the last point of TS.P
17:                    check_near_border(cur_border, PS)
18:                    if points of strip PS near the border then
19:                        Read from LRU Buffer pages for the next strip of set Q into TS.P
20:                    else
21:                        break                       ▷ all points are too far from the border
22:                else
23:                    break                           ▷ end of the set P
24:    if points of strip QS reside near the border then
25:        check_near_border(border, QS)               ▷ cur_border instead of border
26:        if points of strip QS reside near the border then
27:            TS ← PS                                 ▷ the next strip of P becomes current
28:            Update the strip TS to join with the rest points of PS
29:            while TRUE do
30:                CCPS(TS, QS)
31:                if all the points of set P are not processed then
32:                    Update border with the x-coordinate of the last point of TS.P
33:                    check_near_border(border, QS)
34:                    if points of strip QS reside near the border then
35:                        Read from LRU Buffer pages for the next strip of set P into TS.P
36:                    else
37:                        break                       ▷ all points are too far from the border
38:                else
39:                    break                           ▷ end of the set Q
40:    Read from LRU Buffer pages for the next strips of sets P, Q into PS.P, QS.P
41:    if both strips are not empty then
42:        border = sync_queues(PS, QS)                ▷ determine the points in the arrays that form the respective strips
43:    else
44:        break                                       ▷ terminate the process
```

The third step is to prepare the next iteration from the beginning by updating the values of the borders and loading the next of the current strips of both sets. This step is implemented by lines 30-47 of Algorithm 6. We must also highlight that in Algorithm 6, *lborderp* and *lborderq* are variables that store the current left borders of the sorted sets $\mathcal{P}$ and $\mathcal{Q}$. For a more detailed presentation of Algorithm 6, see [15].

Next, we are going to show a step-by-step example for the $SRCPS$ algorithm, using the same input data sets as in the previous example (for $FRCPS$). The query is also the same, that is, we are looking for the $K(=3)$ closest pairs in the data sets $\mathcal{P}$ and $\mathcal{Q}$. As in the previous example,we define that disk-page and array $\mathcal{P}$ in the strip have the same size, enough to fit four points. The data sets and the separation into strips, having variable *width*, are shown in the Figure 8.

The algorithm $SRCPS$ firstly reads the pages with the points [0,1,2,3] of the $\mathcal{P}$ set and $\mathcal{P}$[0,1,2,3] of the $\mathcal{Q}$ set. After the *synchronization* process the current strips are $PS_0\{first = 0, start = 0, end = 2, P[\mathbf{0,1,2}, 3]\}$ and $QS_0\{first = 0, start = 0, end = 3, P[\mathbf{0,1,2,3}]\}$ (see Figure 9). Both left limits ($leftp$ and $leftq$) are initialized to non existing point on the left of two sets: $leftp = lefpq = \{-1, (-1, 0)\}$. In the first step, the algorithm $RCPS$ executes the $K(=3)$CPQ for the strips PS_0 and QS_0. Finishing this task the $maxKHeap$ has the pairs $\{(dist(P_2, Q_1) = 5.831), (dist(P_1, Q_0) = 5.385), (dist(P_1, Q_1) = 3.162)\}$ and the values for left limits are $leftp = P_1$, $leftq = Q_0$. Since there are no strips on the left of the current strips, we must skip the second step and continue with the third one, in which the algorithm must prepare the next iteration. Therefore, the array $PS_0.P$ will remain in main memory. Setting the values of the indexes *start* and *end* to the value 3, $PS_1\{first = 0, start = 3, end = 3, P[0, 1, 2, \mathbf{3}]\}$ will be created and the next page, containing points [4,5,6,7], will be read from disk into array $QS_1.P$.

For the second iteration and after the *synchronization* process, the current strips are $PS_1\{first = 0, start = 3, end = 3, P[0, 1, 2, \mathbf{3}]\}$ and $QS_1\{first = 4, start = 0, end = -1, P[4, 5, 6, 7]\}$ (see Figure 10). The value of $QS_1.end$ is smaller than $QS_1.start$ and the first step (join between current strips PS_1 and QS_1) will be omitted (line 16 of the

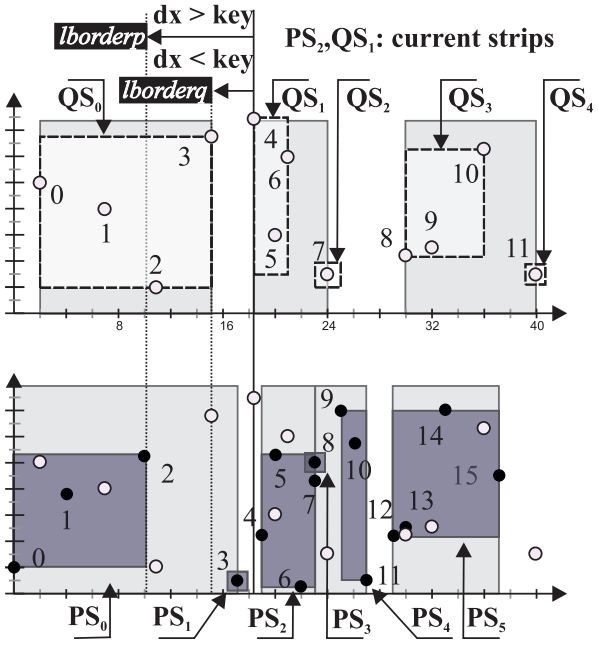

Fig. 8 Applying the *SRCPS* algorithm on two data sets partitioned in strips of variable *width*.

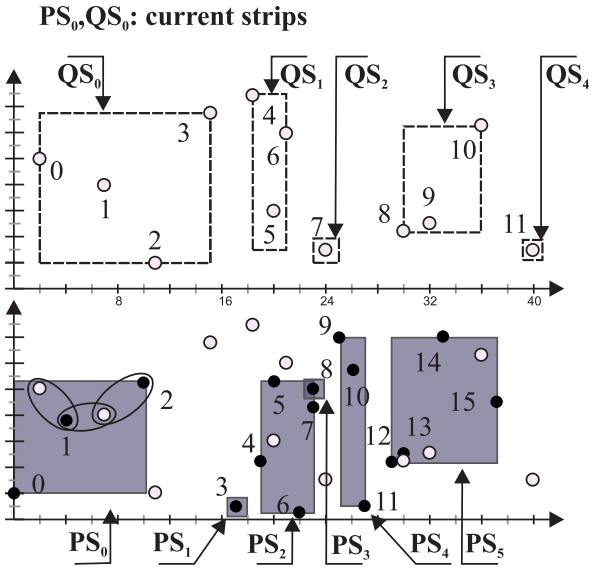

Fig. 9 Join of strips PS_0 and QS_0 using the *SRCPS* algorithm.

Algorithm 6). The *current* strip PS_1 has the point $PS_1.P[3]$ which is at the right border ($PS_1.start = PS_1.end$), the starting point of the *current* strip QS_1 is not the first point of the set $\mathcal{Q}$. The task will continue with the second step by comparing the dx distance between the starting point of the *current* PS_1 strip ($PS_1.P[\mathbf{3}].x = 17$) and the value of *lborderq* which is equal to the value of the last point of the previous strip QS_0 ($QS_0.P[3].x = 15$). Thus it is possible to find closest pairs comparing the point $PS_1.P[\mathbf{3}]$ with the points of the strip QS_0. The second part of the second step will not be executed since the *current* strip QS_1 is empty ($QS_1.start > QS_1.end$). Finishing this step, the $maxKHeap$ has not been updated with new pairs, but the left limit $leftq = Q_2$. In the third step, the algorithm must prepare the current strips for the next iteration. Therefore, the page of $\mathcal{P}$ points [4,5,6,7] is read and the array $QS_1.P$ is kept in main memory for the next iteration.

For the third iteration and after the *synchronization* process, the current strips are $PS_2\{first = 4, start = 0, end = 3, P[\mathbf{4,5,6,7}]\}$ and $QS_1\{first = 4, start = 0, end = 2, P[\mathbf{4,5,6},7]\}$ (see Figure 10). In the first step, the $RCPS$ executes the K(=3)CPQ for the current strips. Exiting from the $RCPS$ function, $maxKHeap$ has new

Algorithm 6 SRCPS

Input: Two X-sorted files of points $\mathcal{P}$ and $\mathcal{Q}$, $|\mathcal{P}| = N$, $|\mathcal{Q}| = M$. $MaxKHeap$: Max-Heap storing $K > 0$ pairs
Output: $MaxKHeap$: Max-Heap storing the K closest pairs between $\mathcal{P}$ and $\mathcal{Q}$
1: Allocate memory for strips PS, QS
2: Initialize the left limits at a non existing point on the left of two sets
3: Read from LRU Buffer pages for the first strips of sets $\mathcal{P}, \mathcal{Q}$ into $PS.P, QS.P$
4: Initialize $lborderp, lborderq$ with the x-coordinates of the first points of $PS.P, QS.P$
5: **if** the first point of the strip PS is on the left of left point of the strip QS **then**
6: **while** all the points of set $\mathcal{P}$ are not processed **and** the last point of $PS.P$ is on the left of the first point of $QS.P$ **do**
7: Initialize $lborderp$ with the x-coordinate of the last point of $PS.P$
8: Read from LRU Buffer pages for the next strip of set $\mathcal{P}$ into $PS.P$
9: **else** ▷ if first point of the $PS.P$ is not on the left of first of the $QS.P$
10: **while** all the points of set $\mathcal{Q}$ are not processed **and** the last point of $QS.P$ is on the left of the first point of $PS.P$ **do**
11: Initialize $lborderq$ with the x-coordinate of the last point of $QS.P$
12: Read from LRU Buffer pages for the next strip of set $\mathcal{Q}$ into $QS.P$
13: **while TRUE do**
14: **if** both strips PS, QS are not empty **then**
15: **sync_queues**(PS, QS) ▷ determine the points in the arrays that form the respective strips
16: **if** both strips PS, QS have points up or on the left of the *border* **then**
17: $gleftp = oleftp \quad gleftq = oleftq$
18: **RCPS**(PS, QS)
19: **swap**$(gleftp, oleftp)$, **swap**$(gleftq, oleftq)$
20: **if** the strip PS is not empty **and** the strip QS is not the first one **and** ($MaxKHeap$ is not full **or** (dx-distance b/t first point of PS and $lborderq <$ key $dist$ of $MaxKHeap$ root) **then**
21: **srcps_on_border**$(PS, QS, lborderq, \mathcal{Q}, gleftq, MaxKHeap)$ ▷ CurS, ComS, lborder, X, left, MaxKHeap
22: **if** $gleftq > oleftq$ **then**
23: $oleftq = gleftq$
24: **if** the strip QS is not empty **and** the strip PS is not the first one **and** ($MaxKHeap$ is not full **or** (dx-distance b/t first point of QS and $lborderp <$ key $dist$ of $MaxKHeap$ root) **then**
25: **srcps_on_border**$(QS, PS, lborderp, \mathcal{P}, gleftp, MaxKHeap)$ ▷ CurS, ComS, lborder, X, left, MaxKHeap
26: **if** $gleftp > oleftp$ **then**
27: $oleftp = gleftp$
28: **if** $oleftp$ differs to the last point of $\mathcal{P}$ **or** $oleftq$ differs to the last point of $\mathcal{Q}$ **then**
29: **break**
30: **if** the strip PS is not empty **then**
31: Update $lborderp$ with the x-coordinate of the last point of $PS.P$
32: **if** all points of the strip PS are not processed **then**
33: Update the end of strip PS with the $|PS.P|$
34: **else**
35: Read from LRU Buffer pages for the next strip of set $\mathcal{P}$ into $PS.P$
36: **if** the strip PS is empty **then**
37: **if** all the points of $\mathcal{Q}$ are processed **or** ($MaxKHeap$ is full **and** dx-distance b/t first point of QS and $lborderp \geq$ key $dist$ of $MaxKHeap$ root) **then**
38: **break** ▷ terminate the process
39: **if** the strip QS is not empty **then**
40: Update $lborderq$ with the x-coordinate of the last point of $QS.P$
41: **if** all points of the strip QS are not processed **then**
42: Update the end of strip QS with the $|QS.P|$
43: **else**
44: Read from LRU Buffer pages for the next strip of set $\mathcal{Q}$ into $QS.P$
45: **if** the strip QS is empty **then**
46: **if** all the points of $\mathcal{P}$ are processed **or** ($MaxKHeap$ is full **and** dx-distance b/t first point of PS and $lborderq \geq$ key $dist$ of $MaxKHeap$ root) **then**
47: **break** ▷ terminate the process

values $\{(dist(P_4, Q_5) = 4.123), (dist(P_5, Q_6) = 3.162), (dist(P_1, Q_1) = 3.162)\}$, and the left limits have values $leftp = P_1, leftq = Q_4$. The *current* strip PS_2 has points (all points) at, or on the left of, the right border, the starting point of the *current* strip QS_1 is not the first point of the set $\mathcal{Q}$ and the difference $PS_2.P[4].x - lborderq = 19 - 15 = 4 < 4.123$. Therefore, the second step will continue by checking the strips PS_2 and QS_0 (previous strip of the *current* strip QS_1). The *current* strip QS_1 has (three) points at, or on the left of, the right border, the starting point of the *current* strip PS_2 is not the first point of the set $\mathcal{P}$ and the difference $QS_1.P[4].x - lborderp = 18.5 - 17 = 1.5 < 4.123$. Therefore, the second step will continue by checking the strips QS_1 and PS_1 (previous strip of the *current* strip PS_2). The $maxKHeap$ is not updated with new pairs, but the left limits of the sets are updated to the new values $leftp = P_2$ and $leftq = Q_4$. In the third step, the algorithm must prepare the current strips for the next iteration. Therefore, the page of $\mathcal{P}$ points [8,9,10,11] is read and the array $QS_1.P$ remains in main memory for next iteration.

For the forth iteration and after the *synchronization* process, the current strips are $PS_3\{first = 8, start = 0, end = 0, P[\mathbf{8}, 9, 10, 11]\}$ and $QS_2\{first = 4, start = 3, end = 3, P[4, 5, 6, \mathbf{7}]\}$. In the first step, the $RCPS$ executes the K(=3)CPQ for the current strips. Exiting from the $RCPS$ function, the $maxKHeap$ has not been updated with new values, and the left limits keep the same values $leftp = P_2$, $leftq = Q_4$. The *current* strip PS_3 has (one) point at or on the left of the right border, the starting point of the *current* strip QS_2 is not the first point of the set $\mathcal{Q}$ and the difference $PS_3.P[8].x - lborderq = 23 - 21 = 2 < 4.123$. Therefore, the second step will continue by checking the strips PS_3 and QS_1 (previous points of the starting point of the *current* strip QS_2). The

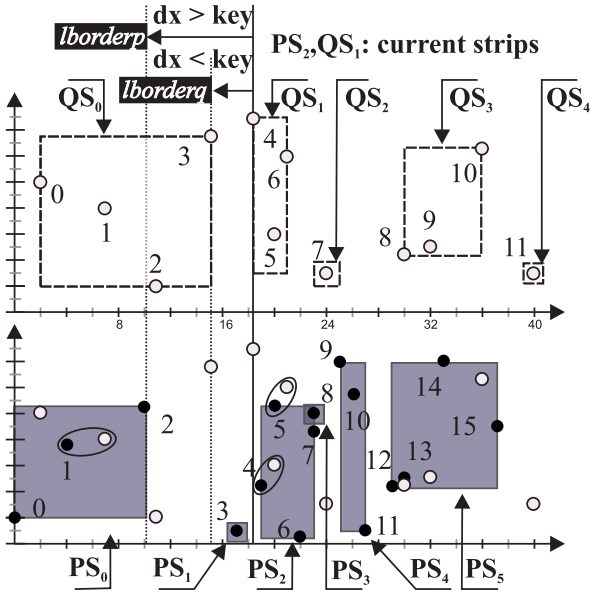

Fig. 10 Join of strips PS_1 and QS_1 using the $SRCPS$ algorithm.

current strip QS_2 has (one) point at, or on the left of, the right border, the starting point of the *current* strip PS_3 is not the first point of the set $\mathcal{P}$ and the difference $QS_2.P[7].x - lborderp = 24 - 23 = 2 < 4.123$. Therefore, the second step will continue by checking the strips QS_2 and PS_2 (previous strip of the *current* strip PS_3). The $maxKHeap$ is not updated with new pairs, but the left limits of the sets are updated to the new value $leftp = P_4$. In the third step, the algorithm must prepare the current strips for the next iteration. Therefore, the array $PS_2.P$ is kept and the page of $\mathcal{Q}$ points [8,9,10,11] is read from the disk for next iteration.

For the fifth iteration and after the *synchronization* process, the current strips are $PS_4\{first = 8, start = 1, end = 3, P[8, \mathbf{9,10,11}]\}$ and $QS_3\{first = 8, start = 0, end = -1, P[8, 9, 10, 11]\}$. The value of index $QS_3.end$ is smaller than the index $QS_3.start$ and the first step (join between current strips PS_4 and QS_3) will be omitted (lines 16-19 of the Algorithm 6). The *current* strip PS_4 has three points at, or on the left of, the right border, the starting point of the *current* strip QS_3 is not the first point of the set $\mathcal{Q}$ and the difference $PS_4.P[9].x - lborderq = 25 - 24 = 1 < 4.123$. Therefore, the second step will continue by checking the strips PS_4 and QS_2 (previous points of the starting point of the *current* strip QS_3). The second part of the second step will not be executed since the *current* strip QS_3 has no points at, or on the left of, the right border ($QS_3.end < QS_3.start$). The $maxKHeap$ is not updated with new pairs, but the left limit of the set $\mathcal{Q}$ updated to the new value $leftq = Q_6$. In the third step, the algorithm must prepare the current strips for the next iteration. Therefore, the page of $\mathcal{P}$ points [12,13,14,15] is read and the array $QS_2.P$ is kept in main memory for the next iteration.

For the sixth iteration and after the *synchronization* process, the current strips are $PS_5\{first = 12, start = 0, end = 3, P[\mathbf{12,13,14,15}]\}$ and $QS_3\{first = 8, start = 0, end = 2, P[\mathbf{8,9,10}, 11]\}$. In the first step, the $RCPS$ executes the $K(=3)$CPQ for the current strips. Exiting from $RCPS$ function, the $maxKHeap$ has new values $\{(dist(P_{13}, Q_9) = 2), (dist(P_{13}, Q_8) = 1), (dist(P_{12}, Q_8) = 1)\}$, and the left limits have values $leftp = P_{14}$, $leftq = Q_9$. The *current* strip PS_5 has all its four points at, or on the left of, the right border, the starting point of the *current* strip QS_3 is not the first point of the set $\mathcal{Q}$ but the difference $PS_5.P[12].x - lborderq = 29 - 24 = 5 > 2$. Therefore, the first part of the second step will be skipped. The *current* strip QS_3 has three points at, or on the left of, the right border, the starting point of the *current* strip PS_5 is not the first point of the set $\mathcal{P}$ but the difference $QS_3.P[8].x - lborderp = 30 - 27 = 3 > 2$. Therefore, the second part of the second step will be skipped. In the third step, the algorithm must prepare the current strips for the next iteration. Therefore, the strip PS_5 is finished and will be updated to the following values $PS_5\{first = 12, start = 4, end = -2, P[12, 13, 14, 15]\}$ and the array $QS_2.P$ is kept in main memory for the next iteration.

In the last iteration (seventh), the first step and the first part of the second step are skipped because the set $\mathcal{P}$ is finished ($PS_5.end = -2 < 0$). The *current* strip QS_4 has only one point that resides at the right border, the starting point of the *current* strip PS_5 is not the first point of the set $\mathcal{P}$ but the difference $QS_4.P[11].x - lborderp = 40 - 37 = 3 > 2$. Therefore, the second part of the second step will be skipped. In the third part, the algorithm must prepare the current strips for the next iteration. For this, the strips PS and QS do not need any update because they have finished their points from the two sets and the algorithm is terminated.

As a summary, the pages which are read from the disk were 12, the pairs involved for calculations were 52, the dx calculations were 84 and the complete $dist$-calculations were 10.

5.4 Analysis

The proofs of the correctness of the External Sweeping-Based KCPQ algorithms ($FCCPS$, $FRCPS$, $SCCPS$ and $SRCPS$) are similar to the proofs of $CCPS$ and $RRPS$ given by the Theorems 1 and 2, respectively. Since the latter are the kernel for the query processing of the former. To extend that proof we must take into account the split of the sweeping axis into strips and the processing strategy of those strips. To see that External Sweeping-Based KCPQ algorithms report the K closest pairs correctly and without any repetition, one key property is that each point (from $\mathcal{P}$ or $\mathcal{Q}$) is assigned to one and only one strip, hence a same pair of points cannot be generated twice. And taking into account the treatment on the borders of the strips, the External Sweeping-Based KCPQ algorithms guarantee that all possible candidate pairs of points are considered and no duplicates are generated.

The I/O cost of the External Sweeping-Based KCPQ algorithms can be estimated, following a similar reasoning as in [14]:

1. The cost of sorting each data set can be expressed as $2m \times \mathcal{P}$, where m represents the number of merge levels and is logarithmic in $|\mathcal{P}|$ [47], and the constant factor 2 accounts for reading and writing $\mathcal{P}$ at each merge level.
2. The cost of the External Sweeping-Based KCPQ algorithms depends of the number of strips that must be read from disk (sr). Let MR_{max} the maximum value of MR (memory requirements) during the execution of a plane-sweep-based algorithm, the sr can be estimated by: $sr \backsimeq numOfStrips \times \lceil \max\{(MR_{max}/M), 0\} \rceil$, where M is the available main memory size. Each point belonging to one of the strips must be read just once. Therefore, the I/O cost of the External Sweeping-Based KCPQ algorithms can be estimated as $(|\mathcal{P}|+|\mathcal{Q}|) \times sr/numOfStrips$.

In summary, the I/O cost of the External Sweeping-Based KCPQ algorithms can be estimated as:

$$2m \times (\mathcal{P} + \mathcal{Q}) + (\mathcal{P} + \mathcal{Q}) \times sr/numOfStrips$$

In the best case ($M > MR_{max}$), $sr = numOfStrips$ and the cost is $2m \times (\mathcal{P} + \mathcal{Q}) + (\mathcal{P} + \mathcal{Q})$. In the worst case ($M \leq MR_{max}$), additional readings are necessary to complete the processing for each strip as we have mentioned above.

5.5 Extension to εDistance Join Query

The adaptation of the External Sweeping-Based KCPQ algorithms from KCPQ to εDJQ is not difficult. As we know, for εDJQ, we have two sets of points $\mathcal{P}$ and $\mathcal{Q}$ as input, and the pairs of points in the range of distances $[\varepsilon_1, \varepsilon_2]$ are selected for the final result and stored in a file of records (*resultFile*) with three fields $(dist, P[i], Q[j])$, where $0 \leq i \leq N-1$ and $0 \leq j \leq M-1$. The $MaxHKeap$ data structure is not needed. The modifications are related to the file operations on *resultFile* and instead of calling to $CCPS$ or $RCPS$, the algorithms should call to $\varepsilon CCPS$ or $\varepsilon RCPS$, respectively. Moreover, instead of calling $check\_near\_border(border, reference\_strip)$, the algorithm will call the function $\varepsilon check\_near\_border(border, reference\_strip)$, which will do the same functionality, discovering the first point of the $reference\_strip$ which has dx smaller than ε_2 from the (right) $border$. More specifically, from $FCCPS$ to get $\varepsilon FCCPS$ we should call $\varepsilon CCPS$ instead of $CCPS$ at lines 8 and 22, and $\varepsilon check\_near\_border(border, reference\_strip)$ should be called at lines 11 and 25.
From $SCCPS$ to get $\varepsilon SCCPS$ we should call $\varepsilon CCPS$ instead of $CCPS$ at lines 6, 14 and 30, and $\varepsilon check\_near\_border(border, reference\_strip)$ should be called at lines 9, 17, 25 and 33.
From $FRCPS$ to get $\varepsilon FRCPS$ we should call $\varepsilon RCPS$ instead of $RCPS$ at lines 16, 22, 28 and 34. Line 20 should be replaced by **while**(dx-distance b/t first points of $QS.P, TS.P \leq \varepsilon_2$) and line 32 by**while**(dx-distance b/t first points of $PS.P, TS.P \leq \varepsilon_2$).
And from $SRCPS$ to get $\varepsilon SRCPS$ we should call $\varepsilon RCPS$ instead of $RCPS$ at line 18. Line 20 should be replaced by **if** (the strip PS is not empty **and** the strip QS is not the first one **and** (dx-distance b/t first point of PS and $lborderq \leq \varepsilon_2$)) and line 24 by **if** (the strip QS is not empty **and** the strip PS is not the first one **and** (dx-distance b/t first point of QS and $lborderp \leq \varepsilon_2$)). Finally, we have to replace $RCPS$ by $\varepsilon RCPS$ in line 18, $maxKHeap$ is not used at all, $\varepsilon srcps\_on\_border$ is called in lines 21 and 25.

6 Performance Evaluation

This section provides the results of an extensive experimental study a) aiming at measuring and evaluating the efficiency of the new algorithms proposed in Section 5 (Sections 6.2-6.6) and effectiveness of these algorithms

(Section 6.7), and b) the comparison of the new algorithms proposed in Section 5 and four algorithms that process the same queries on R-trees (Section 6.8). Section 6.1 presents the experimental setup that is common for parts (a) and (b).

6.1 Experimental Setup

In order to evaluate the behavior of the proposed algorithms, we have used four real spatial data sets of North America, representing cultural landmarks (NAcl) consisting of 9203 points and populated places (NApp) consisting of 24491 points, roads (NArd) consisting of 569082 line-segments, and railroads (NArr) consisting of 191558 line-segments. To create sets of points, we have transformed the MBRs of line-segments from NArd and NArr into points by taking the center of each MBR. Moreover, in order to get the double amount of points from NArr and NArd we choose the two points (min, max) of the MBR of each line-segment. The data of these 6 files were normalized in the range $[0, 1]^2$. We have also created 6 combinations of input sets ($NAppN \times NArrN$, $NAppN \times NArdN$, $NArrN \times NArdN$, $NArrN \times NArdND$, $NArrND \times NArdN$ and $NArrND \times NArdND$) for query processing. We have also used big real spatial data (retrieved from http://spatialhadoop.cs.umn.edu/datasets.html) to justify the use of spatial query algorithms on disk-resident data instead of using them in-memory. They represent water resources (Water) consisting of 5836360 line-segments, parks or green areas (Park) consisting of 11504035 polygons and world buildings (Build) consisting of 114736611 polygons. To create sets of points, we have transformed the MBRs of line-segments from Water into points by taking the center of each MBR and we have considered the centroid of polygons from Park and Build. We have also created 3 combinations of input sets ($Water \times Park$, $Water \times Build$, $Park \times Build$) for query processing.

We have also created synthetic clustered data sets of 125000, 250000, 500000 and 1000000 points, with 125 clusters in each data set (uniformly distributed in the range $[0, 1]^2$), where for a set having N points, $N/125$ points were gathered around the center of each cluster, according to Gaussian distribution. We made 4 combinations of synthetic data sets by combining two separate instances of data sets, for each of the above 4 cardinalities (i.e. $125KC1N \times 125KC2N$, $250KC1N \times 250KC2N$, $500KC1N \times 500KC2N$, and $1000KC1N \times 1000KC2N$) and 1 combination of synthetic data sets by combining two data sets of different cardinalities ($500KC2N \times 1000KC1N$).

All experiments were performed on a PC with Intel Core 2 Duo, 2.2 GHz CPU with 4 GB of RAM and 2TBs of secondary storage, with Ubuntu Linux v. 14.04 LTS (Linux OS), using the GNU C/C++ compiler (gcc).

In our previous paper [1], it is shown that the semi-circle variant of both *Classic Plane-Sweep* and *Reverse Run Plane-Sweep* algorithms has the highest execution-time efficiency, for the KCPQ. Therefore, all experiments were executed using $CCPS$ and $RCPS$. For the KCPQ and for all (4) algorithms we study how the value of K, disk page size, size of the strips and size of the LRU buffer affects efficiency, by executing experiments for the previous 14 combinations of data sets. As efficiency measures we used:

1. The overall execution time (i.e. response time); this measurement is reported in milliseconds (ms) and represents the overall CPU time consumed, as well as the I/O time needed by each algorithm.
2. The number of X-axis distance calculations (dx).
3. The number of disk accesses (disk-pages read).

To measure the effectiveness of the new algorithms, we can use the *selection ratio*, which is defined as the fraction of pairs considered by the algorithms for processing over the total number of possible pairs. This is just the opposite to the pruning ratio, and a pair selection occurs when a candidate pair from two strips is considered for processing according to its dx distance.

6.2 The effect of the number of pairs (K)

In order to examine the effect of the number of pairs (K) on the new algorithms, K is set equal to 1, 10, 100, 1000 and 10000; the size of disk page equals to 4 KBytes; the size of strip is 16 KBytes; and there is no LRU buffer (its size is 0).

6.2.1 The execution time

The results for execution time are similar for all input data sets. Table 3 shows the execution time in ms when KCPQ is processed by the FCCPS, SCCPS, FRCPS and SRCPS algorithms on the $NArrN \times NArdND$ data sets. As the value of K increases, the execution time increases, but the rate of the increment gets higher as K increases. For example, using the FCCPS algorithm, from $K = 1$ to $K = 10$ the time increased by 0%, from $K = 10$ to $K = 10^2$ by 3%, from $K = 10^2$ to $K = 10^3$ by 16% and from $K = 10^3$ to $K = 10^4$ by 29%.

K	FCCPS	SCCPS	FRCPS	SRCPS	Total
1	41.48	34.84	24.03	**22.14**	122.49
10	41.31	33.70	23.38	**21.30**	119.69
100	42.44	35.22	24.17	**22.67**	124.50
1000	49.07	45.89	**29.97**	32.68	157.61
10000	63.12	62.33	**42.51**	50.59	218.55

Table 3 Execution time in ms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $NArrN \times NArdND$, in relation to K.

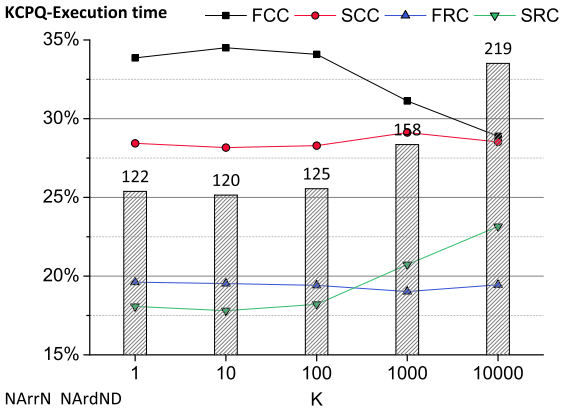

Fig. 11 Fractions of execution time for KCPQ, using FCCPS, SCCPS, FRCPS and SRCPS on $NArrN \times NArdND$, in relation to K.

K	FCCPS	SCCPS	FRCPS	SRCPS	Total
1	3.17	2.06	0.99	**0.99**	7.11
10	5.80	4.69	3.61	**3.61**	17.72
100	12.45	11.34	10.23	**10.23**	44.26
1000	33.82	32.73	31.48	**31.46**	129.48
10000	101.46	100.40	98.46	**98.31**	398.63

Table 4 Number of dx distance calculations is millions ($\times 10^6$) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $NArrN \times NArdND$, in relation to K.

Considering all experiments and all data sets, we find that SCCPS overcomes FCCPS 58-12 times and FRCPS overcomes SRCPS 36-34 times. Comparing the best result among FCCPS and SCCPS (variants of Classic Plane-Sweep algorithm) and the best result among FRCPS and SRCPS (variants of Reverse Run Plane-Sweep algorithm) for every combination of data sets, we conclude that Reverse Run algorithms are faster in all cases (70-0).

Figure 11 shows the execution time of each algorithm for KCPQ as a fraction of the total time consumed by all algorithms(represented by the respective bar). It is shown that the SRCPS (line with down facing triangles as markers) was the fastest for $K = 1, 10, 100$, while FRCPS (line with up facing triangles as markers) was the fastest for $K = 1000, 10000$. This situation is dominating in most data set combinations.

6.2.2 The number of the dx distance calculations

The results with respect to the number of dx distance calculations are similar for all input data sets. Table 4 shows the values of this metric when KCPQ is processed by the FCCPS, SCCPS, FRCPS and SRCPS algorithms on the $NArrN \times NArdND$ data sets. As the value of K increases, the number of dx distance calculations also increases. However, while the number of K increases geometrically with a ratio of 10, the number of dx distance calculations increases with a ratio ranging between 1.83 and 2.66. For example, using the SRCPS algorithm from $K = 1$ to $K = 10$ the number of dx distance calculations increased by 266%, from $K = 10$ to $K = 10^2$ by 183%, from $K = 10^2$ to $K = 10^3$ by 207% and from $K = 10^3$ to $K = 10^4$ 213%.

Considering all experiments and all data sets, we find that SCCPS overcomes FCCPS 49-21 times and SRCPS overcomes FRCPS 61-9 times. Comparing the best result among FCCPS and SCCPS and the best result among (the almost identical results of) FRCPS and SRCPS for every combination of data sets, we conclude that Reverse Run algorithms need fewer dx distance calculations in all cases (70-0).

Figure 12 shows the number of dx distance calculations of each algorithm for KCPQ as a fraction of the total number of dx distance calculations performed by all algorithms (represented by the respective bar). It is shown that the SRCPS (line with down-facing triangles as markers) took from 13.7% up to 24.7% of the total number of dx distance calculations needed to execute the queries. The FRCPS algorithm has almost equal number of dx distance calculations so its line (with up-facing triangles as markers) is overwritten from the line of the SRCPS (note that overlapping down-facing and up-facing triangles appear as stars). RR algorithms need fewer dx distance calculations in all cases.

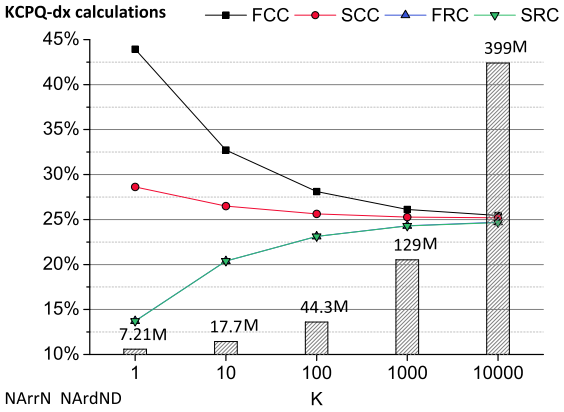

Fig. 12 Fractions of the number of dx distance calculations for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $NArrN \times NArdND$, in relation to K.

K	FCCPS	SCCPS	FRCPS	SRCPS	Total
1	13340	13991	**7824**	9136	44291
10	13340	16455	**7828**	11928	49551
100	13348	18495	**7856**	14252	53951
1000	13388	19387	**7940**	15364	56079
10000	13540	19583	**8232**	15772	57127

Table 5 Number of disk accesses for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $NArrN \times NArdND$, in relation to K.

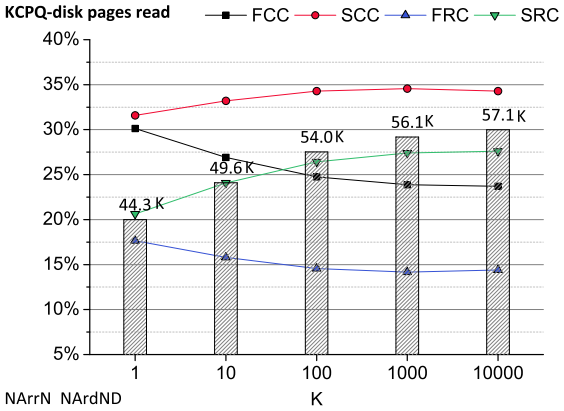

Fig. 13 Fractions of number of disk accesses for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $NArrN \times NArdND$, in relation to K.

6.2.3 The number of the disk accesses (pages read)

The results for number of disk accesses are similar for all input data sets and this performance measure proved to be the most important factor that shaped the results. Table 5 shows the values of this metric when KCPQ is processed by the FCCPS, SCCPS, FRCPS and SRCPS algorithms on the $NArrN \times NArdND$ data sets. As the value of K increases, the number of disk accesses increases slightly, or marginally. While K increases geometrically with a ratio of 10, the number of pages read increases, for example, in the FRCPS algorithm, by 0.051%, 0.358%, 1.069%, and 3.678%.

Considering all experiments and all data sets, we find that FCCPS overcomes SCCPS 68-2 times and FRCPS overcomes SRCPS 70-0 times. Comparing the best result among FCCPS and SCCPS and the best result among FRCPS and SRCPS for every combination of data sets, we conclude that Reverse Run algorithms need fewer disk accesses in all cases (70-0).

Figure 13 shows the number of disk accesses of each algorithm for KCPQ as a fraction of the total number of disk accesses needed by all algorithms (represented by the respective bar).

Summarizing the results of experiments on the effect of K, we note that: (1) The exponential growth of K causes (non geometrical) increase in the execution time. (2) The exponential growth of K causes increase in the number of dx-distance calculations with a lower ratio (up to 3 for most datasets and up to 7 for the biggest data set combination). (3) The number of disk accesses required by FCCPS and FRCPS algorithms increases marginally

pg	FCCPS	SCCPS	FRCPS	SRCPS	Total
1	136.85	141.50	**113.74**	144.13	536.22
2	119.73	123.98	**105.02**	126.90	475.63
4	114.44	116.96	**100.49**	117.62	449.51
8	112.13	111.93	**98.86**	112.87	435.79
16	111.52	112.03	**100.84**	112.70	437.09

Table 6 Execution times in ms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $1000KC1N \times 1000KC2N$, in relation to pg.

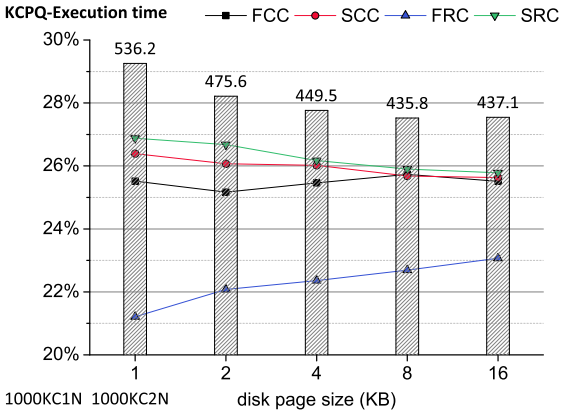

Fig. 14 Fractions of execution time for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $1000KC1N \times 1000KC2N$, in relation to pg.

with the growth of K, unlike SCCPS and SRCPS where the increment is more pronounced. Moreover, (4) the fastest algorithm proved to be the SRCPS for small values of K, while FRCPS is the fastest for large values of K. Finally, (5) SRCPS was slightly the most economical algorithm in terms of dx distance calculations.

6.3 The effect of the disk page size (pg)

In order to examine the effect of the disk page size (pg) on the new algorithms, the size of disk pages (pg) is set equal to 1, 2, 4, 8 and 16 KBytes; $K = 1000$; the size of strips is 16 KBytes; and there is no LRU buffer (its size is 0).

6.3.1 The execution time

The results for execution time are similar for all input data sets. Table 6 shows the execution time in ms when KCPQ is executed by the algorithms FCCPS, SCCPS, FRCPS and SRCPS on the $1000KC1N \times 1000KC2N$ data sets. As the page size increases the execution time is reduced, but the rate of decrement continuously decreases. For example, using SRCPS algorithm from $pg = 1KB$ to $pg = 2KB$ the time decreased by 12%, from $pg = 2KB$ to $pg = 4KB$ by 7.3%, from $pg = 4KB$ to $pg = 8KB$ by 4% and from $pg = 8KB$ to $pg = 16KB$ by 0.15%.

Figure 14 shows the execution time of each algorithm values as a fraction of the total execution time consumed by all algorithms (represented by the respective bar). Considering all experiments and all data sets, we find that SCCPS overcomes FCCPS 54-16 times and FRCPS overcomes SRCPS 51-19 times. Comparing the best result among FCCPS and SCCPS and the best result among FRCPS and SRCPS, for every combination of data sets, we conclude that Reverse Run algorithms are the fastest in all cases. In Figure 14, it is shown that the increment of the disk page size for sizes larger than 8 KB, does not give any advantage in query execution for any algorithm. Experiments with page sizes larger than 32 KB show that the execution becomes slightly slower.

6.3.2 The number of dx distance calculations

The results of the number of dx distance calculations are similar for all input data sets. In Table 7, we can see the values of this metric when KCPQ is executed by the algorithms FCCPS, SCCPS, FRCPS and SRCPS on the $1000KC1N \times 1000KC2N$ data sets. As the value of disk page size increases, the number of dx distance calculations stays almost constant.

pg	FCCPS	SCCPS	FRCPS	SRCPS	Total
1	528.0	554.6	381.0	**380.4**	1844.0
2	529.3	554.8	381.8	**381.0**	1846.9
4	529.3	554.8	381.8	**381.0**	1846.9
8	529.5	554.9	382.0	**381.4**	1847.8
16	529.5	554.9	382.0	**381.4**	1847.8

Table 7 Number of dx distance calculations in millions ($\times 10^6$) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $1000KC1N \times 1000KC2N$, in relation to pg.

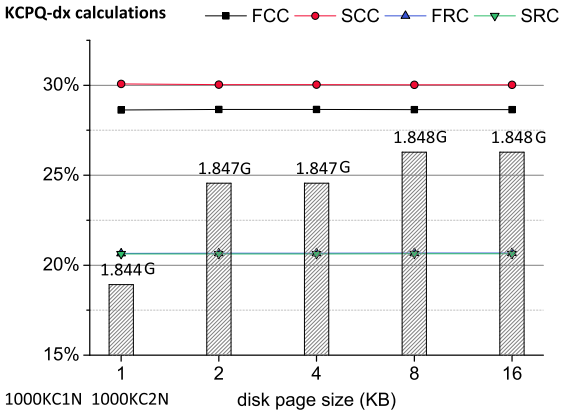

Fig. 15 Fractions of number of dx distance calculations for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $1000KC1N \times 1000KC2N$, in relation to pg.

pg	FCCPS	SCCPS	FRCPS	SRCPS	Total
1	63044	96126	**53988**	105668	318826
2	31178	47453	**26594**	52082	157307
4	15590	23729	**13298**	26042	78659
8	7802	11806	**6678**	13032	39318
16	3902	5905	**3340**	6517	19664

Table 8 Number of disk accesses (pages read) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $1000KC1N \times 1000KC2N$, in relation to pg.

Considering all experiments and all data sets, we find that SCCPS overcomes FCCPS 45-25 times and SRCPS overcomes FRCPS 58-12 times. Comparing the best result among FCCPS and SCCPS and the best result among FRCPS and SRCPS, for every combination of data sets, we conclude that Reverse Run algorithms need fewer dx calculations in all cases (70-0).

Figure 15 shows the number of dx distance calculations of each algorithm as a fraction of the total number of dx distance calculations needed by all algorithms (represented by the respective bar). SRCPS (line with down-facing triangles as markers) needed 20.63% up to 20.64% of the total number of dx distance calculations needed to execute the queries. FRCPS has almost equal numbers of dx distance calculations, so its line (with up-facing triangles as markers) is overwritten by the line of the SRCPS. The Reverse Run algorithms need fewer dx distance calculations in all cases.

6.3.3 The number of the disk accesses (pages read)

The results for the number of disk accesses (pages read) are similar for all input data sets and this performance measure proved to be the most important factor that shaped the results. Table 8 shows the values of this metric when KCPQ is executed by the FCCPS, SCCPS, FRCPS and SRCPS algorithms on the $1000KC1N \times 1000KC2N$ data sets. As the disk page size (pg) increases, the number of disk accesses decreases. The rate of this decrement is quite stable. While the disk page size increases geometrically with a ratio of 2, the number of pages read decreases smoothly, for example, in the FRCPS algorithm steps by 50.74%, 50.00%, 49.78%, 49.99%.

Considering all experiments and all data sets, we find that FCCPS overcomes SCCPS 64-6 times and FRCPS overcomes SRCPS 70-0 times. Comparing the best result among FCCPS and SCCPS and the best result among FRCPS and SRCPS for every combination of data sets, we conclude that FRCPS needs fewer disk accesses in all cases (70-0). Figure 16 shows the values of the number of disk accesses of each algorithm as a fraction of the total number of disk accesses needed by all algorithms (represented by the respective bar).

Summarizing the results of experiments on the effect of disk page size, pg, we note that: (1) Doubling the size of pg causes decrease in execution time not larger than 20% on real and synthetic data sets and not larger than

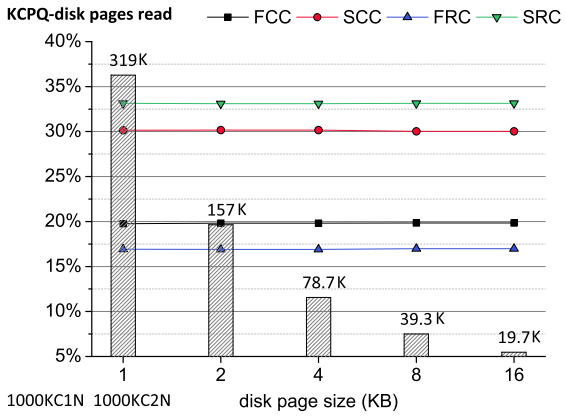

Fig. 16 Fractions of number of disk accesses (pages read) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $1000KC1N \times 1000KC2N$, in relation to pg.

ss	FCCPS	SCCPS	FRCPS	SRCPS	Total
2	963.56	753.53	**582.91**	666.26	2966.26
4	841.25	636.73	**511.40**	570.36	2559.74
8	801.21	592.29	**484.13**	534.10	2411.73
16	802.37	586.02	**484.95**	527.10	2400.44
32	652.37	579.34	**487.89**	526.44	2246.04

Table 9 Execution time in ms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $Water \times Park$, in relation to ss.

30% on the big real data sets. (2) As pg increases, the number of disk accesses required by the FCCPS and FRCPS algorithms decreases significantly, but for the SCCPS and SRCPS algorithms this decrease is limited. (3) The number of dx distance calculations remains quite stable (not affected by pg). Moreover, (4) the fastest algorithm proves to be FRCPS, while SRCPS proves to be quite economical in terms of dx distance calculations.

6.4 The effect of the size of strips (ss)

In order to examine the effect of the size of the strips (ss) in terms of performance of the new algorithms, we set the value of $K = 1000$; $pg = ss$ (size of disk page = size of strip), the size of strip (ss) = 2, 4, 8, 16 and 32 KBytes; and there is no LRU buffer (its size is 0). In the previous section 6.3.1 it was proved that the page size, having constant the size of strip (but larger than the disk page size), affects the execution time up to 20% in some cases. In order to neutralize this effect of page size with respect to the execution time, we set equal size for pg and ss.

6.4.1 The execution time

The results for execution time are similar for all input data sets. Table 9 shows the execution time in ms when KCPQ is executed by the FCCPS, SCCPS, FRCPS and SRCPS algorithms on the $Water \times Park$ data sets. As the strip size increases, the execution time is reduced, with a decreasing rate. For example, using SCCPS, from $ss = 2KB$ to $ss = 4KB$ the time decreased by 15.5%, from $ss = 4KB$ to $ss = 8KB$ by 7%, from $ss = 8KB$ to $ss = 16KB$ by 1% and from $ss = 16KB$ to $ss = 32KB$ by 1%. The Reverse Run algorithms are shown to be faster than the Classic ones.

Figure 17 shows the execution time of each algorithm as a fraction of the total execution time consumed by all algorithms (represented by the respective bar). Considering all experiments and all data sets, we find that SCCPS overcomes FCCPS 54-16 times and FRCPS overcomes SRCPS 52-18 times. Comparing the best result among FCCPS and SCCPS and the best result among FRCPS and SRCPS for every combination of data sets, we conclude that Reverse Run algorithms are the fastest in most cases (68-2). In Figure 17, it is shown that the increment of the strip size, for sizes larger than 32 KB does not give advantage in query execution time for any algorithm. Experiments with strip sizes larger than 32 KB show that execution becomes slower. The Reverse Run algorithms are faster and the best strip size is 8 or 16 KB for all types of data sets, which, in all cases, is larger than the physical I/O unit.

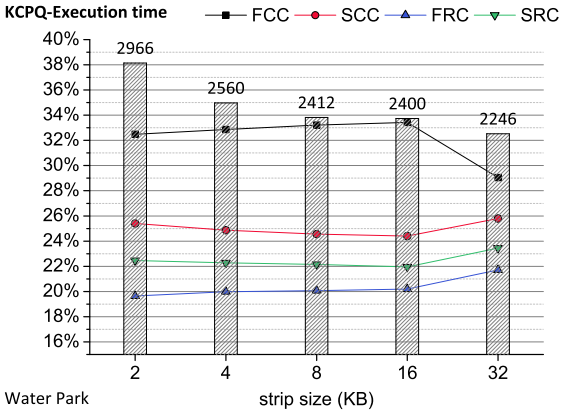

Fig. 17 Fractions of execution time for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $Water \times Park$, in relation to ss.

ss	FCCPS	SCCPS	FRCPS	SRCPS	Total
2	497.47	481.46	469.63	**469.52**	1918.07
4	497.54	479.69	469.55	**469.49**	1916.27
8	498.09	478.17	469.53	**469.47**	1915.25
16	498.46	476.74	469.49	**469.46**	1914.15
32	490.56	476.24	469.48	**469.45**	1905.73

Table 10 Number of dx distance calculations in millions ($\times 10^6$) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $Water \times Park$, in relation to ss.

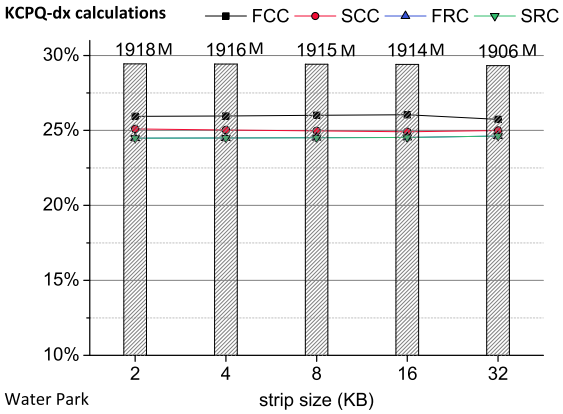

Fig. 18 Fractions of number of dx distance calculations for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $Water \times Park$, in relation to ss.

6.4.2 The number of dx distance calculations

The results for the number of dx distance calculations are similar for all input data sets. Table 10 shows the values of this metric when KCPQ is executed by the FCCPS, SCCPS, FRCPS and SRCPS algorithms on the $Water \times Park$ data sets. As the value of strip size increases the number of dx distance calculations remains almost constant.

Considering all experiments and all data sets, we find that SCCPS overcomes FCCPS 45-25 times and SRCPS overcomes FRCPS 49-21 times. Comparing the best result among FCCPS and SCCPS and the best result among FRCPS and SRCPS for every combination of data sets, we conclude that the Reverse Run algorithms need fewer dx calculations in all cases (70-0).

Figure 18 shows the number of dx distance calculations of each algorithm as a fraction of the total number of dx distance calculations of all algorithms (represented by the respective bar). It is shown that FRCPS (line with up-facing triangles as markers) needed from 24.48% up to 24.63% of the total number of dx distance calculations. The SRCPS algorithm has almost equal number of dx distance calculations so its line (with down-facing triangles as markers) is overwritten from the line of the FRCPS. The Reverse Run algorithms need fewer dx distance calculations in all cases.

ss	FCCPS	SCCPS	FRCPS	SRCPS	Total
2	373,962	345,686	**236,166**	346,931	1,302,745
4	182,414	164,051	**110,000**	159,128	615,593
8	89,871	81,247	**52,843**	77,217	301,178
16	44,678	40,583	**25,902**	38,079	149,242
32	16,199	20,263	**12,822**	18,902	68,186

Table 11 Number of disk accesses (pages read) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $Water \times Park$, in relation to ss.

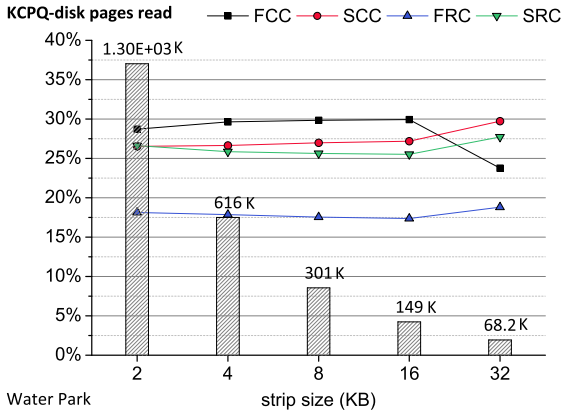

Fig. 19 Fractions of number of disk accesses (pages read) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $Water \times Park$, in relation to ss.

6.4.3 The number of the disk accesses (pages read)

The results for the number of disk accesses (pages read) are similar for all input data sets and this performance measure proved to be the most important factor that shaped the results. Table 11 shows the values of this metric when KCPQ is executed by the FCCPS, SCCPS, FRCPS and SRCPS algorithms on the $Water \times Park$ data sets. As the strip size (ss) increases the number of disk accesses decreases. The rate of this decrement is quite stable. While the strip size increases geometrically with a ratio of 2, the number of pages read decreases, for example, in the SRCPS algorithm, by 54.13%, 51.47%, 50.69% and 50.36%.

Considering all experiments and all data sets, we find that FCCPS overcomes SCCPS 62-8 times and FRCPS overcomes SRCPS 70-0 times. Comparing the best result among FCCPS and SCCPS and the best result among FRCPS and SRCPS for every combination of data sets, we conclude that FRCPS needs fewer disk accesses in all cases (70-0). Figure 19 shows the number of disk accesses of each algorithm as a fraction of the total number of disk accesses needed by all algorithms (represented by the respective bar).

Summarizing the results of experiments on the effect of strip size, ss, we note that: (1) The exponential growth of ss causes decrease in the execution time not larger than 15% for all, real and synthetic data sets. (2) As ss increases, the number of disk accesses needed by each of the algorithms decreases notably, but the best behaviour for this performance measure is for FRCPS. (3) The number of dx distance calculations remains quite stable (not affected by ss). Moreover, (4) the fastest algorithm proves to be FRCPS, while SRCPS proves to be quite economical in terms of dx distance calculations.

6.5 The effect of the LRU buffer

In order to examine the effect of the size of the LRU buffer on the performance of the new algorithms, we examined several LRU buffer sizes. Although, one might expect that, as a result of finding in RAM (and not reading from disk) some of the strips needed for processing, the execution time would be possibly reduced, in fact, the cost for the management of the LRU-buffer proved to overcome any such reduction and the execution time increased when the LRU buffer size increased. For all LRU buffer sizes, FRCPS proved to be faster than the SRCPS algorithm in double the cases (47-23), and the one with the smallest number of strips found in the buffer (the fastest execution of FRCPS was the one without any buffering). Moreover, the LRU-buffer does not have any effect on the number of dx distance calculations, since this performance measure is not affected whether the data are in RAM or in disk.

$\varepsilon \times 10^{-3}$	εFCCPS	εSCCPS	εFRCPS	εSRCPS	Total
0.00	2.85	2.54	1.77	**1.51**	8.67
1.25	6.33	6.85	**5.13**	5.83	24.14
2.50	9.40	10.04	**8.16**	8.97	36.57
5.00	15.50	16.34	**14.25**	15.12	61.21
10.00	27.62	28.98	**26.35**	27.69	110.65

Table 12 Execution time in s for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS on $Park \times Build$, in relation to ε.

$\varepsilon \times 10^{-3}$	εFCCPS	εSCCPS	εFRCPS	εSRCPS
0.00	32.89%	29.31%	20.43%	**17.37** %
1.25	26.23%	28.36%	**21.25**%	24.16%
2.50	25.71%	27.47%	**22.31**%	24.52%
5.00	25.33%	26.69%	**23.28**%	24.70%
10.00	24.96%	26.19%	**23.82**%	25.03%

Table 13 Fractions of execution time of each algorithm over the total execution time for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS on $Park \times Build$, in relation to ε.

$\varepsilon \times 10^{-3}$	εFCCPS	εSCCPS	εFRCPS	εSRCPS	Total
0.00	0.237	0.120	0.018	**0.018**	0.393
1.25	2.884	2.767	2.666	**2.666**	10.982
2.50	5.530	5.415	**5.313**	**5.313**	21.572
5.00	10.823	10.711	**10.608**	10.608	42.750
10.00	21.409	21.303	**21.198**	21.198	85.108

Table 14 Number of dx distance calculations in billions ($\times 10^9$) for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS on $Park \times Build$, in relation to ε.

6.6 Experimental results for εDJQ

In this section, we study the effect of the increment of the distance threshold (ε) on the εDJQ. In order to examine the effect of ε on the εDJQ algorithms, ε_1 is set equal to 0 and $\varepsilon_2 = \varepsilon$. $\varepsilon = 0, 1.25 \times 10^{-5}, 2.5 \times 10^{-5}, 5 \times 10^{-5}$ and 10×10^{-5} for medium real and synthetic data, and $\varepsilon = 0, 1.25 \times 10^{-3}, 2.5 \times 10^{-3}, 5 \times 10^{-3}$ and 10×10^{-3} for big real data. $pg = 4$ KBytes, $ss = 16$ KBytes and there is no LRU buffer (its size is 0).

6.6.1 The execution time

The results for execution time are similar for all input data sets. Table 12 shows the execution time in s when the εDJQ is processed by the εFCCPS, εSCCPS, εFRCPS and εSRCPS algorithms on $Park \times Build$ data sets. As the value of ε increases the execution time grows, and the rate of the increment continuously grows (after the first non zero value of the maximum distance ε). The εFRCPS algorithm is shown to be faster in the most cases.

Considering all experiments and all data sets, we find that εFCCPS overcomes εSCCPS 52-18 times and εFRCPS overcomes εSRCPS 50-20 times. Comparing the best result among εFCCPS and εSCCPS and the best result among εFRCPS and εSRCPS for every combination of data sets, we conclude that ε Reverse Run algorithms are faster in the most cases (67-3).

Table 13 shows the values of the execution time of each algorithm as a fraction of the total time consumed by all algorithms on $Park \times Build$ data sets. It is shown that the εFRCPS needed from 20.43% up to 23.82% of the total time to execute the queries and it is the fastest algorithm for all values of $\varepsilon > 0$. For, $\varepsilon = 0$ the εSRCPS algorithm was faster, since its fraction of time was 17.37%.

6.6.2 The number of dx distance calculations

The results for the number of dx distance calculations are similar for all input data sets. Table 14 shows the values of this metric when εDJQ is processed by the εFCCPS, εSCCPS, εFRCPS and εSRCPS algorithms on $Park \times Build$ data sets. As the value of ε increases the number of dx distance calculations also increases. However, while the value of ε increases geometrically with a ratio of 2, the number of dx distance calculations increases to a same ratio near to 2.

Considering all experiments and all data sets, we find that εSCCPS overcomes εFCCPS 70-0 times and εFRCPS overcomes εSRCPS 35-29 times. Comparing the best result among εFCCPS and εSCCPS and the best result among εFRCPS and εSRCPS for every combination of data sets, we conclude that Reverse Run algorithms need fewer dx distance calculations in all cases (70-0).

Table 15 shows the number of dx distance calculations of each algorithm as a fraction of the total number of dx distance calculations needed by all algorithms on $Park \times Build$ data sets. It is shown that the εFRCPS needed 4.63% for the case of $\varepsilon = 0$ and for the other cases from 24.21% up to 24.91% of the total number of dx distance calculations. The εSRCPS algorithm has a little fewer dx distance calculations than εFRCPS only in the case $\varepsilon = 0$ and in all other cases the number of dx distance calculations is almost equal. The Reverse Run algorithms need fewer dx calculations in all cases.

$\varepsilon \times 10^{-3}$	εFCCPS	εSCCPS	εFRCPS	εSRCPS
0.00	60.31%	30.49%	4.63%	**4.58** %
1.25	26.26%	25.20%	**24.27**%	**24.27**%
2.50	25.64%	25.10%	**24.63**%	**24.63**%
5.00	25.32%	25.05%	**24.81**%	**24.81**%
10.00	25.15%	25.03%	**24.91**%	**24.91**%

Table 15 Fraction of number of dx distance calculations of each algorithm over the total number of dx distance calculations for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS on $Park \times Build$, in relation to ε.

$\varepsilon \times 10^{-3}$	εFCCPS	εSCCPS	εFRCPS	εSRCPS	Total
0.00	1354.9	1348.9	**742.6**	**742.6**	4189.0
1.25	1360.7	1905.9	**754.2**	1370.2	5391.0
2.50	1366.4	1956.1	**765.7**	1434.5	5522.7
5.00	1377.8	2015.0	**788.7**	1517.5	5699.0
10.00	1400.9	2390.0	**834.4**	1938.4	6563.7

Table 16 Number of disk accesses (pages read) in thousands ($\times 10^3$) for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS on $Park \times Build$, in relation to ε.

$\varepsilon \times 10^{-5}$	εFCCPS	εSCCPS	εFRCPS	εSRCPS
0.00	32.34%	32.20%	**17.73**%	**17.73** %
1.25	25.24%	35.35%	**13.99**%	25.42%
2.50	24.74%	35.42%	**13.86**%	25.97%
5.00	24.18%	35.36%	**13.84**%	26.63%
10.00	21.34%	36.41%	**12.71**%	29.53%

Table 17 Fraction of number of disk accesses of each algorithm on the total number of disk accesses for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS on $Park \times Build$, in relation to ε.

6.6.3 The number of the disk accesses (pages read)

The results for the number of disk accesses (pages read) are similar for all input data sets and this performance measure proved to be the most important factor that shaped the results. Table 16 shows the values of this metric when εDJQ is executed by the εFCCPS, εSCCPS, εFRCPS and εSRCPS algorithms on $Park \times Build$ data sets. As the value of ε increases, the number of disk accesses increases. But the rate of this increment is too small for the Reverse Run algorithms and bigger for the Classic ones. While ε increases geometrically with a ratio of 2, the number of pages read increases with a lower ratio, for example, the number of pages read for the εFRCPS algorithm increases by 1.57%, 1.51%, 3.00% and 5.80%.

Considering all experiments and all data sets, we find that εFCCPS overcomes 63-7 times εSCCPS and εFRCPS overcomes 57-0 times εSRCPS (there are 13 cases of tie). Comparing the best result between εFCCPS and εSCCPS and the best result between εFRCPS and εSRCPS for every combination of data sets, we can conclude that ε Reverse Run algorithms need fewer disk accesses in all cases (70-0). Table 17 shows the values of the number of disk accesses of each algorithm as a fraction of the total number of disk accesses needed by all algorithms.

Summarizing the results of experiments on the effect of ε, note that: (1) the geometrical growth of ε causes a non-geometrical increase of execution time. (2) the exponential growth of ε causes an increase in the number of dx distance calculations with a lower ratio (ranging very close to 2). (3) The number of disk accesses required by εFCCPS and εFRCPS algorithms increases marginally with the growth of ε, unlike εSCCPS and εSRCPS, where the increment is more pronounced. Moreover, (4) faster algorithm proves to be the εFRCPS than εSRCPS (50-20), while εFRCPS proves to be more economical in terms of dx distance calculations than εFRCPS (35-29).

The experiments were continued in the same manner as in sections 6.3, 6.4 and 6.5 for KCPQ, in order to study the effect of the disk page size, the strip size and the size of the LRU-buffer. In general, the results are similar between KCPQ and εDJQ. The fastest in execution time and reading fewer pages from the disk proved to be the εFRCPS algorithm. We note only that, for the case of $\varepsilon = 0$, SRCPS is slightly better than FRCPS. For the cases where $\varepsilon > 0$, the results for εFRCPS are not as good as the ones of FRCPS. This is explained, since the most important factors that improve the performance of an algorithm for the KCPQ are (1) how quickly pairs with very small distances will enter the $maxKHeap$, and (2) how efficiently the algorithm will manage the largest distance of these pairs. In contrast to KCPQ, εDJQ does not need the fast finding of pairs with small distance, since the maximum acceptable distance is consistently defined by the user beforehand (ε). Only the smart and economical management of the given distance affects the final performance of an algorithm.

6.7 Effectiveness study

To study the effectiveness of the proposed algorithms we will use the *selection ratio*, that is, the fraction of pairs considered by the new algorithms for processing over the total number of possible pairs (a pair is selected for processing if its dx distance is smaller than the distance of the K-th closest pair found so far). This effectiveness

K	FCCPS	SCCPS	FRCPS	SRCPS
1	12.72%	7.67%	3.00%	**2.99**%
10	18.75%	13.70%	9.01%	**9.01**%
100	33.98%	28.94%	24.19%	**24.18**%
1000	82.96%	77.96%	72.89%	**72.84**%
10000	237.97%	233.17%	226.47%	**226.12**%

Table 18 Fraction of pairs ($\times 10^{-6}$) processed over the total number of possible pairs (*selection ratio*) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $NArrN \times NArdND$.

K	FCCPS	SCCPS	FRCPS	SRCPS
1	1.85%	1.61%	**1.13**%	1.21%
10	32.66%	29.66%	**23.65**%	23.68%
100	85.20%	88.43%	67.62%	**66.83**%
1000	266.13%	278.41%	191.37%	**190.96**%
10000	691.96%	699.59%	**509.33**%	510.08%

Table 19 Fraction of pairs ($\times 10^{-6}$) processed over the total number of possible pairs (*selection ratio*) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $1000KC1N \times 1000KC2N$.

K	FCCPS	SCCPS	FRCPS	SRCPS
1	5.51%	3.16%	0.98%	**0.97**%
10	43.33%	27.40%	12.60%	**12.58**%
100	64.64%	48.70%	33.78%	**33.76**%
1000	137.71%	121.45%	105.65%	**105.51**%
10000	459.94%	442.55%	421.18%	**418.22**%

Table 20 Average fraction of pairs ($\times 10^{-6}$) processed over the total number of possible pairs (*selection ratio*) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for all real data sets.

K	FCCPS	SCCPS	FRCPS	SRCPS
1	3.76%	3.80%	2.74%	**2.73**%
10	75.68%	77.67%	62.26%	**59.76**%
100	189.65%	203.28%	171.06%	**165.91**%
1000	573.11%	587.07%	462.33%	**462.16**%
10000	1456.99%	1485.84%	**1252.13**%	1253.75%

Table 21 Average fraction of pairs ($\times 10^{-6}$) processed over the total number of possible pairs (*selection ratio*) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for all synthetic data sets.

measure is the opposite to the pruning ratio, and therefore the smaller the selection ratio, the higher the power of pruning of the algorithm.

We are going to focus on the increment of K. Tables 18 and 19 report the effect of K on the selection ratio for real and synthetic data, respectively. In order to extract conclusions from the tables, we have to take into account that for the specific combination of real data there are $191,558 \times 1,134,164 = 217{,}258{,}187{,}512$ possible pairs (2.17×10^{11}) and for the specific synthetic data there are 10^{12} possible pairs. We can observe that an increasing K makes the selection ratio of the proposed algorithms to increase continuously. Therefore, the effectiveness of our algorithms degrades as K turns to be too large, due to the increase of the distance of the K-th closest pair. And, the larger the K value, the smaller the difference between Reverse Run algorithms and Classic plane-sweep algorithms (we mainly observe this fact on real data) in terms of selection ratio. From these tables, we observe that SRCPS is the winner in most of the cases, but FRCPS is very close to it (being the winner in the remaining cases). This means that FRCPS sacrifices slightly effectiveness for efficiency, in the use of the partitioning technique. An interesting conclusion from this effectiveness measure is that the best algorithms in pruning are the Reverse Run ones and this conclusion is in accordance to efficiency. Moreover, since the *selection ratio* depends on the dx distance, it is the most representative measure for pruning and for effectiveness.

We note that the average performance in pruning, for all combinations of real and synthetic data, follows the same trend. This behaviour is shown in Tables 20 and 21, where SRCPS is the winner in most of the cases, but FRCPS is very close to it.

6.8 Performance Comparison to R-trees

In order to examine the performance of the new algorithms in comparison to a widely accepted access method, like R-trees, we have performed experiments for measuring:

- The creation time of the sorted files needed for the new algorithms and the creation time of the R-tree structure. In the case of the new algorithms, we used external merge sort with 16 buffers of 16KB and in the case of the R-tree, we used advanced bulk loading [48] (using code available form http://libspatialindex.org), to reduce the time needed for tree construction.

data set name	number of objects ($\times 10^3$)	R-tree tree file length (MB)	R-tree creation time (s)	new algorithms bin file length (MB)	new algorithms creation time (s)
NarrN	191.6	16.7	0.6	4.4	0.2
NarrND	383.2	33.2	1.2	8.8	0.3
500KC1N	500.0	43.4	1.7	11.4	0.4
500KC2N	500.0	43.4	1.7	11.4	0.4
NardN	569.1	49.3	1.9	13.0	0.5
1000KC1N	1000.0	86.6	4.8	22.9	0.8
1000KC2N	1000.0	86.6	4.8	22.9	0.8
NardND	1138.2	98.7	7.0	26.1	0.9
Water	5836.4	505.9	39.8	133.6	5.4
Park	11504.0	997.0	81.7	263.3	10.6
Build	114736.6	9943.8	1104.0	2626.1	128.5

Table 22 Creation time of R-tree structures and sorted data files used by the new algorithms, for several data sets.

K	query execution time (ms) CCPS-BF	query execution time (ms) FRCPS	creation + query execution time (s) CCPS-BF	creation + query execution time (s) FRCPS
1	467.81	31.99	8.698	1.421
10	473.99	32.44	8.704	1.422
100	481.24	34.27	8.711	1.423
1000	496.50	44.53	8.727	1.434
10000	586.10	66.24	8.816	1.455

Table 23 Query execution time(ms) and total time(s) (creation+execution) of CCPS-BF and FRCPS for $NArdN \times NArdND$ data sets.

– The execution time and number of disk accesses for processing the KCPQ and εDJQ.

The experiments were run for the following data set combinations: $NArrN \times NArdN$, $NArrND \times NArdND$, $500KC1N \times 500KC2N$, $500KC2N \times 1000KC1N$, $1000KC1N \times 1000KC2N$, $Water \times Park$, $Water \times Build$ and $Park \times Build$.

6.8.1 Experimental results for KCPQ

Table 22 shows the name and the number of objects for each data set in ascending order of size. It also shows the sizes and creation times of the R-tree structure and sorted data files used by the new algorithms. It is obvious that the size of the files used by the new algorithms is approximately 3.7 times smaller than the size of the R-tree structure. The time for the creation of files for the new algorithms ranges from 3.9 up to 8.6 times smaller than the time for the R-tree structure.

The 8 combinations of data sets were chosen in order to take measurements between small and big data sets and also between real and synthetic data sets. We have chosen the fastest algorithm executing the KCPQ with R-trees, the CCPS-BF. It is the algorithm which scans the nodes of the R-tree in Best First manner (using one global minimum heap to sort the pairs of the nodes reached so far with *minmin* distance) and when a pair of leafs is reached, the pairs of points are processed using the classic plain sweep algorithm. On the other hand we have chosen to compare to the FRCPS algorithm executing the same KCPQs because it is faster than SRCPS algorithm in more cases (36-34) of all combinations and all K values. The smaller number of total pairs was for the combination $NArrN \times NArdN$ having a total number of pairs equal to $191,558 \times 569,082$, while the biggest number of total pairs was for the combination $Park \times Build$ data sets having a total number of pairs equal to $11,504,035 \times 114,736,611$. Observing the values of the metrics of experiments on real and synthetic data for the first 5 combinations we see that the FRCPS was the absolute winner for all values of K and for all metrics. Table 23 shows both the values of query time and total time (creation + query) needed by the CCPS-BF and FRCPS algorithms to execute the KCPQ on $NArdN \times NArdND$ data sets. As the value of K increases, the relative difference of execution time between the two algorithms is slightly reduced (gain by 93.16%, 93.16%, 92.88%, 91.03% and 88.70%). The total time, creation and query execution time showcased similar behavior with an even smaller reduction. FRCPS needs less total time by gain by 83.61%.

In Table 24 (second and third columns) we can see the number of disk accesses when the KCPQ is executed by the CCPS-BF and FRCPS algorithms on $NArdN \times NArdND$ data sets. The increment of K didn't significantly affect the number of needed disk accesses for both algorithms. FRCPS again proved to be more efficient than CCPS-BF by an average gain of 88.73%.

Table 25 shows the values of query time and total time (creation + query) when the KCPQ is executed by the CCPS-BF and FRCPS algorithms on the combination of big data sets $Water \times Build$. As the value of K increases, the relative difference of execution time between the two algorithms didn't showcase a clear pattern. The FRCPS

	number of disk accesses			
	$NArdN \times NArdND$		$Water \times Build$	
K	CCPS-BF	FRCPS	CCPS-BF	FRCPS
1	80,562	8,955	813,256	709,320
10	80,582	8,967	813,386	709,592
100	80,616	8,987	813,786	710,464
1000	80,784	9,095	815,094	712,796
10000	81,290	9,519	819,130	720,012

Table 24 Number of disk accesses of CCPS-BF and FRCPS algorithms for $NArdN \times NArdND$ and $Water \times Build$ data sets.

	query execution time (s)		creation + query execution time (s)	
K	CCPS-BF	FRCPS	CCPS-BF	FRCPS
1	3.577	3.256	1,147.371	137.094
10	3.607	1.646	1,147.401	135.484
100	3.596	1.829	1,147.391	135.667
1000	3.619	2.319	1,147.413	136.157
10000	3.810	5.302	1,147.605	139.140

Table 25 Query execution time(s) and total time(s) (creation+execution) of CCPS-BF and FRCPS for $Water \times Build$ data sets.

	query execution time (s)		creation + query execution time (s)	
$\varepsilon \times 10^{-3}$	εCCPS-BF	εFRCPS	εCCPS-BF	εFRCPS
0.00	127.9	1.770	1,314	140.8
1.25	158.3	5.129	1,344	144.2
2.50	159.4	8.157	1,345	147.2
5.00	161.1	14.249	1,347	153.3
10.0	164.2	26.354	1,350	165.4

Table 26 Query execution time(s) and total time(s) (creation+execution) of εCCPS-BF and εFRCPS for $Park \times Build$ data sets.

remained faster for all K values smaller than 10.000 while CCPS-BF was faster for the last value of K. The relative differences (gain) are as follows: 8.99%, 54.35%, 49.14%, 35.93% and −39.13%. For positive (negative) values the FRCPS is faster (slower). FRCPS was also faster for all K values of the total time, creation and query execution time. FRCPS needs less total time by 88.09%. In Table 24 (forth and fifth columns) we can see the number of disk accesses in relation to K values for the same data sets combination. The increment of K didn't significantly affect the number of needed disk accesses for both algorithms. FRCPS again proved to be more efficient than CCPS-BF by an average gain of 12.58%.

Considering all experiments and all data sets, we find that FRCPS overcomes CCPS-BF 38-2 times for query execution time, 40-0 for total time and 38-2 times for disk accesses.

6.8.2 Experimental results for εDJQ

The same 8 combinations of data sets were used in order to take measurements while executing the εDJQs with $\varepsilon = 0, 1.25 \times 10^{-5}, 2.5 \times 10^{-5}, 5 \times 10^{-5}$ and 10×10^{-5} for the first 5 combinations (medium real and synthetic data), and with $\varepsilon = 0, 1.25 \times 10^{-3}, 2.5 \times 10^{-3}, 5 \times 10^{-3}$ and 10×10^{-3} between the last 3 combinations (big real data). We have chosen the fastest algorithm for executing εDJQ with R-trees: the εCCPS-BF. On the other hand we have chosen to compare to the εFRCPS algorithm executing the same εDJQs because it is faster than εSRCPS algorithm in more cases (50-20) of all combinations and all ε values. Observing the values of the metrics of experiments on real and synthetic data, small or big data sets for all the 8 combinations we see that the εFRCPS was the absolute winner for all values of ε and for all metrics. Table 26 shows both the values of query time and total time (creation + query) needed by the εCCPS-BF and εFRCPS algorithms to execute the εDJQ on the biggest combination, $Park \times Build$ data sets. As the value of ε increases, the relative difference of execution time between the two algorithms was slightly reduced (gain by 98.62%, 96.76%, 94.88%, 91.16% and 83.95%). The total time, creation and query execution time showcased similar behavior with an even smaller reduction. εFRCPS needs less total time gain of 88.80%.

In Table 27 we can see the number of disk accesses when εDJQ is executed by the εCCPS-BF and εFRCPS algorithms on $Park \times Build$ data sets. The increment of ε didn't significantly affect the number of needed disk accesses for both algorithms. εFRCPS again proved to be more efficient than εCCPS-BF by an average gain of 89.46%.

Considering all experiments and all data sets, we find that εFRCPS overcomes εCCPS-BF in all cases of ε values and for all data sets in all performance metrics.

	number of disk accesses	
$\varepsilon \times 10^{-3}$	εCCPS-BF	εFRCPS
0.00	7,347,596	742,593
1.25	7,356,476	754,233
2.50	7,365,108	765,657
5.00	7,382,220	788,657
10.0	7,416,790	834,369

Table 27 Number of disk accesses of εCCPS-BF and εFRCPS algorithms for $Park \times Build$ data sets.

6.9 Conclusions from the experiments

In our previous work [1], it was shown that the Reverse Run PS algorithms are faster than the Classic ones for the KCPQ, when the data is stored and processed in main memory. Classic PS algorithms always process data sets from left to right and the runs of the two sets are generally interleaved. On the other hand, RR PS algorithms process pairs of points in opposite sweeping order, starting from pairs of points that are the closest possible to each other, avoiding further processing of pairs that is guaranteed not to be part of the final result and restricting the search space by using dx distance values on the sweeping axis. Due to these, the pruning distance (*key dist of MaxKHeap root*) is expected to be updated more quickly and the query processing cost of RR PS algorithms is expected to be smaller.

From the experiments presented previously, when the data are stored on disk, we conclude that the main factors that determine the *execution time* are: (1) The number of operations and comparisons; (2) The number of pages that are transferred from disk to main memory; (3) The volume of memory required and its management; and (4) How quickly $maxKHeap$ is filled up with pairs having small distances and how fast the pruning distance is reduced (it is important for the KCPQ, unlike the εDJQ), because the lower its value is, the greater the power of pruning. Each of these factors affects differently the final result. FRCPS is faster in more cases, considering different values of K, disk page size (pg), size of strips (ss) and size of LRU buffer (bs), although SRCPS requires less memory in comparison to FRCPS.

With respect to the *number of dx distance calculations*, the SRCPS algorithm seems to be better (lower number of calculations) in most cases, although FRCPS is quite close (i.e. the difference compared to SRCPS in total calculations is rather small). This is due to the fact that for the RR PS algorithms, if we ignore the non-sweeping dimension, the number of calculations can be proved to be optimal, since we always start with the closest pair of points.

With respect to the number of disk accesses, FRCPS needs the least disk accesses in all experiments (considering different values of K, pg and ss). This is due to the combination of RR PS processing and the *uniform filling* technique, since, for *uniform filling*, the number of strips is predefined beforehand and it is smaller than for *uniform splitting* (higher non-uniformity of data leads to larger difference between the two techniques). This means that the number of strips read from disk, or the number of disk accesses, is smaller. In addition, the number of disk accesses seems to be the most influential factor governing an algorithm's efficiency in execution time, and the difference between SRCPS and FRCPS becomes significant for this performance measure: FRCPS is totally dominating, and thus, faster.

In conclusion, FRCPS is the best algorithm for all performance or efficiency measures for the following reasons:

1. a smaller number of strips partition the space,
2. a smaller number of strips are read from disk,
3. a more consistent application of RR PS processing is applied in the management of strips.

Moreover, this work emphasizes on the effective use of dx distance for pruning, considering the selection ratio as the effectiveness measure. The main conclusion in this context is that RR PS algorithms are the most effective ones for pruning, highlighting that SRCPS is slightly better than FRCPS.

Finally, from this extensive experimental study of the new algorithms, we conclude that RR PS algorithms are the most efficient and effective ones for the KCPQ and εDJQ, and the FRCPS variant is the best one.

Regarding the comparison of the new algorithms to the widely accepted R-tree based methods, the file needed by the new algorithms is created in extremely smaller time than the R-tree structure. The best new algorithm (FRCPS) answers the KCPQ in significantly smaller time than the best R-tree based algorithm (CCPS-BF), in most cases. It is slower in only 2 cases out of the 40 cases studied. An analogous situation (in 2 out of the 40 cases the new algorithm looses) arises for the number of disk pages read by the algorithms. This can be attributed to the data distribution of these cases that favors the algorithms that work on a tree structure. Nevertheless, even in these 2 cases, the total (creation + query) time of the new algorithm for answering the KCPQ is significantly smaller. The best new algorithm (εFRCPS) answers the εDJQ in significantly smaller time and with significantly less pages read from disk than the best R-tree based algorithm (εCCPS-BF), in all cases. The fact that the εDJQ requires finding of all (not only K) pairs within a specified distance forces the R-tree algorithms to search within the whole tree, and thus the new algorithm is faster even in the above 2 cases.

Overall, even when the data sets do not change at a very rapid rate, or are reusable for subsequent queries, the best new algorithm is a better choice than the best R-tree based algorithm for the KCPQ and the εDJQ.

7 Conclusions and Future Work

This paper has presented several efficient and effective algorithms (FCCPS, SCCPS, FRCPS and SRCPS) for the KCPQ and εDJQ, when neither inputs are indexed. First of all, we have enhanced the classic plane-sweep algorithm for DJQs with two improvements: *sliding window* and *sliding semi-circle*. Next, we proposed a new algorithm called *Reverse Run Plane-Sweep*, that improves the processing of the classic plane-sweep algorithm for DJQs, minimizing the Euclidean and sweeping axis distance calculations. Then, as the main contribution of this work, four algorithms (FCCPS, SCCPS, FRCPS and SRCPS) for KCPQ and εDJQ are proposed, without the use of indexes on both disk-resident data sets. These four algorithms employ a combination of *plane-sweep* and space partitioning techniques to join the data sets. We also presented results of an extensive experimental study, where efficiency and effectiveness measures are explored for the proposed algorithms. From this performance study, that was conducted on medium and big spatial (real and synthetic) data sets, when neither input is indexed, we conclude that RR PS algorithms are the most efficient and effective for the KCPQ and εDJQ, and that FRCPS is the best variant, which combines RR PS processing with *uniform filling* partitioning technique. Finally, the best of the new algorithms was experimentally compared to the best algorithm that is based on the R-tree (a widely accepted access method), for KCPQs and εDJQs and it was shown that the new algorithms outperform R-tree based algorithms, in most cases. For future work, we plan to further investigate the adaptation of the new plane-sweep-based algorithms, when neither input is indexed, to other DJQs (as Iceberg Distance Join Query [42] and K Nearest Neighbour Join query [43]). Moreover, it would be interesting to study approximate implementations of the proposed algorithms by using the distance-based approximate techniques presented in [37] and to implement new in-memory DJQ algorithms inspired in the disk-based approaches.

Acknowledgements Work of all authors funded by the Development of a GeoENvironmental information system for the region of CENtral Greece (GENCENG) project (SYNERGASIA 2011 action, supported by the European Regional Development Fund and Greek National Funds); project number 11SYN 8 1213. Work of Antonio Corral also supported by the MINECO research project [TIN2013-41576-R] and the Junta de Andalucia research project [P10-TIC-6114].

References

1. G. Roumelis, M. Vassilakopoulos, A. Corral, Y. Manolopoulos, A new plane-sweep algorithm for the k-closest-pairs query, in: SOFSEM Conference, 2014, pp. 478–490.
2. R. H. Güting, An introduction to spatial database systems, VLDB J. 3 (4) (1994) 357–399.
3. S. Shekhar, S. Chawla, Spatial databases - a tour, Prentice Hall, 2003.
4. V. Gaede, O. Günther, Multidimensional access methods, ACM Computing Surveys 30 (2) (1998) 170–231.
5. A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Closest pair queries in spatial databases, in: SIGMOD Conference, 2000, pp. 189–200.
6. A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Algorithms for processing k-closest-pair queries in spatial databases, Data Knowl. Eng. 49 (1) (2004) 67–104.
7. F. P. Preparata, M. I. Shamos, Computational Geometry - An Introduction, Springer, 1985.
8. K. Hinrichs, J. Nievergelt, P. Schorn, Plane-sweep solves the closest pair problem elegantly, Information Processing Letters 26 (5) (1988) 255–261.
9. E. H. Jacox, H. Samet, Spatial join techniques, ACM Trans. Database Syst. 32 (1) (2007) 7.
10. H. Shin, B. Moon, S. Lee, Adaptive and incremental processing for distance join queries, IEEE Trans. Knowl. Data Eng. 15 (6) (2003) 1561–1578.
11. N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The r*-tree: An efficient and robust access method for points and rectangles, in: SIGMOD Conference, 1990, pp. 322–331.
12. E. H. Jacox, H. Samet, Iterative spatial join, ACM Trans. Database Syst. 28 (3) (2003) 230–256.
13. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. S. Vitter, Scalable sweeping-based spatial join, in: VLDB Conference, 1998, pp. 570–581.
14. C. Gurret, P. Rigaux, The sort/sweep algorithm: A new method for r-tree based spatial joins, in: SSDBM Conference, 2000, pp. 153–165.
15. G. Roumelis, A. Corral, M. Vassilakopoulos, Y. Manolopoulos, New plane-sweep algorithms for distance-based join queries in spatial databases, Tech. Rep. TR-01-2014, Data Eng. Lab, AUTH, Greece, http://delab.csd.auth.gr/m̃ichalis/TR-01-2014.pdf (2014).
16. G. R. Hjaltason, H. Samet, Incremental distance join algorithms for spatial databases, in: SIGMOD Conference, 1998, pp. 237–248.
17. P. Rigaux, M. Scholl, A. Voisard, Spatial databases - with applications to GIS, Elsevier, San Francisco, CA, 2002.
18. H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann, San Francisco, CA, 2007.
19. S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, A. Ailamaki, TOUCH: in-memory spatial join by hierarchical data-oriented partitioning, in: SIGMOD Conference, 2013, pp. 701–712.
20. B. Sowell, M. A. V. Salles, T. Cao, A. J. Demers, J. Gehrke, An experimental analysis of iterated spatial joins in main memory, PVLDB 6 (14) (2013) 1882–1893.

21. D. Sidlauskas, C. S. Jensen, Spatial joins in main memory: Implementation matters!, PVLDB 8 (1) (2014) 97–100.
22. H. Zhang, G. Chen, B. C. Ooi, K. Tan, M. Zhang, In-memory big data management and processing: A survey, IEEE Trans. Knowl. Data Eng. 27 (7) (2015) 1920–1948.
23. N. Mamoulis, D. Papadias, Multiway spatial joins, ACM Trans. Database Syst. 26 (4) (2001) 424–475.
24. T. Brinkhoff, H.-P. Kriegel, B. Seeger, Efficient processing of spatial joins using r-trees, in: SIGMOD Conference, 1993, pp. 237–246.
25. A. Guttman, R-trees: A dynamic index structure for spatial searching, in: SIGMOD Conference, 1984, pp. 47–57.
26. M.-L. Lo, C. V. Ravishankar, Spatial hash-joins, in: SIGMOD Conference, 1996, pp. 247–258.
27. J. M. Patel, D. J. DeWitt, Partition based spatial-merge join, in: SIGMOD Conference, 1996, pp. 259–270.
28. M. Smid, Closest-point problems in computational geometry, in: J.-R. Sack, J. Urrutia (Eds.), Handbook of Computational Geometry, Elsevier, 2000, Ch. 20, pp. 877–935.
29. A. Corral, J. M. Almendros-Jiménez, A performance comparison of distance-based query algorithms using r-trees in spatial databases, Inf. Sci. 177 (11) (2007) 2207–2237.
30. Y. J. Kim, J. M. Patel, Performance comparison of the r*-tree and the quadtree for knn and distance join queries, IEEE Trans. Knowl. Data Eng. 22 (7) (2010) 1014–1027.
31. G. Gutiérrez, P. Sáez, The k closest pairs in spatial databases when only one set is indexed, GeoInformatica 17 (4) (2013) 543–565.
32. R. Weber, H.-J. Schek, S. Blott, A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces, in: VLDB Conference, 1998, pp. 194–205.
33. N. Koudas, K. C. Sevcik, High dimensional similarity joins: Algorithms and performance evaluation, IEEE Trans. Knowl. Data Eng. 12 (1) (2000) 3–18.
34. E. P. F. Chan, Buffer queries, IEEE Trans. Knowl. Data Eng. 15 (4) (2003) 895–910.
35. C. Yang, K.-I. Lin, An index structure for improving nearest closest pairs and related join queries in spatial databases, in: IDEAS Conference, 2002, pp. 140–149.
36. F. Angiulli, C. Pizzuti, An approximate algorithm for top-k closest pairs join query in large high dimensional data, Data Knowl. Eng. 53 (3) (2005) 263–281.
37. A. Corral, M. Vassilakopoulos, On approximate algorithms for distance-based queries using r-trees, The Computer Journal 48 (2) (2005) 220–238.
38. J. Shan, D. Zhang, B. Salzberg, On spatial-range closest-pair query, in: SSTD Conference, 2003, pp. 252–269.
39. L. H. U, N. Mamoulis, M. L. Yiu, Computation and monitoring of exclusive closest pairs, IEEE Trans. Knowl. Data Eng. 20 (12) (2008) 1641–1654.
40. M. A. Cheema, X. Lin, H. Wang, J. Wang, W. Zhang, A unified approach for computing top-k pairs in multidimensional space, in: ICDE Conference, 2011, pp. 1031–1042.
41. D. Choi, C. Chung, Y. Tao, Maximizing range sum in external memory, ACM Trans. Database Syst. 39 (3) (2014) 21:1–21:44.
42. Y. Shou, N. Mamoulis, H. Cao, D. Papadias, D. W. Cheung, Evaluation of iceberg distance joins, in: SSTD Conference, 2003, pp. 270–288.
43. C. Böhm, F. Krebs, The *k*-nearest neighbour join: Turbo charging the kdd process, Knowl. Inf. Syst. 6 (6) (2004) 728–749.
44. J. Zhang, N. Mamoulis, D. Papadias, Y. Tao, All-nearest-neighbors queries in spatial databases, in: SSDBM Conference, 2004, pp. 297–306.
45. B. Bryan, F. Eberhardt, C. Faloutsos, Compact similarity joins, in: ICDE Conference, 2008, pp. 346–355.
46. G. Graefe, Query evaluation techniques for large databases, ACM Comput. Surv. 25 (2) (1993) 73–170.
47. A. Aggarwal, J. S. Vitter, The input/output complexity of sorting and related problems, Commun. ACM 31 (9) (1988) 1116–1127.
48. S. T. Leutenegger, J. M. Edgington, M. A. Lopez, Str: A simple and efficient algorithm for r-tree packing, in: ICDE Conference, 1997, pp. 497–506.