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

George Roumelis[1], Antonio Corral[2], Michael Vassilakopoulos[3] and Yannis Manolopoulos[1]

[1]*Dept. of Informatics, Aristotle University, GR-54124 Thessaloniki, Greece.*
[2]*Dept. on Informatics, University of Almeria, 04120 Almeria, Spain.*
[3]*Dept. of Electrical and Computer Engineering, University of Thessaly, 37 Glavani−28th October Str, GR-38221 Volos, Greece.*
Email: acorral@ual.es

Efficient and effective processing of *distance-based join queries* (DJQs) 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). The output of these queries are a set of pairs (one from each input set) of spatial objects, with the K lowest distances, and within distance threshold ε, respectively. In this paper, we enhance the classic plane-sweep algorithm for DJQs with two improvements, so called *sliding window* and *sliding semi-circle*. Moreover, we propose 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. But the most important contribution is the proposal of four new algorithms (FCCPS, SCCPS, FRCPS and SRCPS) for KCPQ and their extensions for εDJQ in the context of spatial databases, without the use of an index on each data set saving on disk (neither inputs are indexed). They employ a combination of the *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 KCPQ and εDJQ. That performance study conducted on long spatial data sets (real and synthetic) validates that our proposed plane-sweep-based algorithms are very promising in terms of both efficient and effective measures, when neither inputs are indexed.

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 [1]. 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.

The most basic form of such a system is 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* [2, 3]. One of the most frequent spatial query in spatial database systems is *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* (DJQs). The most representative and studied DJQs in the spatial database field are the K Closest Pairs Query (KCPQ) and εDistance Join Query (εDJQ). *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]. *εDJQs*, 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. DJQs 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 increase order based on its 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 ($L_t : Points \times Points \rightarrow \mathbb{R}^+$) or Minkowski distance between two points, $P = (P[0], P[1], \cdots, P[d-1])$ and $Q = (Q[0], Q[1], \cdots, Q[d-1])$, in the d-dimensional data space, D^d, and it is defined as follows:

$$L_t(p,q) = \begin{cases} \left(\sum_{i=0}^{d-1} |p_i - q_i|^t\right)^{1/t} & \text{if} \quad 1 \leq t < \infty \\ \max_{0 \leq i \leq d-1}\{|p_i - q_i|\} & \text{if} \quad t = \infty \end{cases}$$

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 it will be the basis for DJQs studied on this paper.

$$dist(p,q) = \sqrt{\sum_{i=0}^{d-1} |p_i - q_i|^2}$$

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 DJQs 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 is proven [10, 6], and thus the reduction of execution time of the query processing.

In the context of computational geometry, in [8], the *plane-sweep* algorithm is applied to find the closest pair in a set of points, in an elegant way. Two improvements when a new pair can be formed are proposed. The first one is when all candidates which may form a new closest pair with the fixed point p on the *sweepline* lie in a half-circle centered at this point, with radius δ (it is called *half-circle query*). The second one, since the half-circle query is complex and in particular when it is embedded in a *plane-sweep* algorithm, a boundary rectangle query (a rectangle with width δ in X-axis and, height $2 * \delta$ in Y-axis ($p+\delta$ and $p-\delta$ from p)) is proposed and adopted as the final improvement. A critical observation made in [8] is that as the *sweepline* passes through a fixed point, there are at most constant number of points need to be checked. But this property does not exist in our case which is essentially a bichromatic closest pair problem, because the number of points in such a problem cannot be bounded. Moreover, the algorithm proposed in [8] uses an array and a balanced binary tree (e.g. AVL-tree) to sort both axes of the data set, while we will use one array for each data set, sorting on one axis (e.g. X axis). Finally, and as we will explain along the paper, our proposed *plane-sweep* algorithm can be easy adapted to distance-based join query processing on disk resident data.

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 necessary pay off. In particular, the time needed to build the index before the execution of the spatial query takes an important relevance 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 as output is not reused several times for subsequent spatial queries, then it may not be worthwhile to expend the time of building a new index. This is especially emphasized for spatial intersection joins that make use of indexes, which needs 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 a new algorithm for DJQs (KCPQ and εDJQ) when none inputs are indexed, and to study its behavior in the context of spatial databases. Moreover, our proposal is also motivated by the work of [13, 14] for spatial intersection joins. And the contributions of this paper are summarized as follows:

1. We enhance the *Classic Plane-Sweep* algorithm for DJQs with two improvements (*sliding window* and *sliding semi-circle*). Which were proposed in [8] for the closest pair problem over one data set, and here have been adapted to DJQs for spatial query processing, where two data sets are involved.
2. We improve processing of the *Classic Plane-Sweep* algorithm for DJQs, with a new algorithm called *Reverse Run Plane-Sweep*, *RRPS*, that minimizes Euclidean and sweeping axis distance calculations.
3. We have provided the proofs of 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 DJQs, when neither inputs are indexed and the data are stored on disk.
4. There are many contributions in the context of spatial intersection joins when one, both 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 KCPQ and their extensions for εDJQ for performing such as DJQs, without the use of an index on each data set saving on disk. They employ a combination of the *plane-sweep* algorithms (Classic Circle (CCPS) and Reverse Run Circle (RCPS)) and space partitioning techniques (uniform splitting and uniform filling) to join the data sets.
5. We present results of an extensive experimentation, that compares the performance (in terms of efficiency and effectiveness) of the proposed algorithms.

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 we show a complete classification of spatial join and distance-based join queries taking into account whether one, both or neither inputs are indexed. The *Classic Plane-Sweep* algorithm in DJQs is described in Section 3, as well as two improvements to reduce the number of distance computations, and the correctness of the *Classic Plane-Sweep* algorithm is also proven. In Section 4, the new *plane-sweep* algorithm (*Reverse Run Plane-Sweep*, *RRPS*) for KCPQ is presented, proving its correctness and that it makes less or at most equal number of sweeping axis distance calculations in comparison to *Classic Plane-Sweep* algorithm. In Section 5, we present and analyse the Sweeping-Based Distance Join Algorithm for KCPQ ($SBKCPQ$) and εDJQ ($SB\varepsilon DJQ$). Section 6 contains the results of the experimental study, taking into account different parameters for comparison. 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 complete classification of spatial join and distance-based join queries taking into account whether one, both or neither inputs are indexed, along with the review of other recent contribution 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 range query (RQ) and K Nearest Neighbors query (KNNQ). When we have two spatial data sets the most representative DBQs 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 DJQ reports only the top K pairs, and the latter, also know 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 2.1. (**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 (K**CPQ**)) 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 2.2. (**εDistance Join Query**) *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 ($\varepsilon DJQ(P,Q,\varepsilon_1,\varepsilon_2) \subseteq P \times Q$) 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: $\varepsilon DJQ(P,Q,\varepsilon_1,\varepsilon_2) = \{(p_i,q_j) \subseteq P \times Q : \varepsilon_1 \leq dist(p_i,q_j) \leq \varepsilon_2\}$.*

These two DJQs have been actively studied in the context of R-trees [15, 5, 10, 6], but when the data sets are not indexed they have not been paid similar attention.

2.2. Related Work

This section presents a complete classification of the spatial join and distance-based join queries depending on one, both or neither inputs are indexed. Moreover, other related DJQs 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

The *spatial join* is one of the most related and influential spatial query with respect to DJQs. Therefore, we are going to revise a complete classification of the spatial joins depending on whether one, both or neither input data sets are indexed.

As we know, given two spatial data sets P and Q, the *spatial join* finds pairs of spatial object in the Cartesian product $P \times Q$ which satisfy a spatial predicate, most commonly *intersect*. The *spatial join* is one of the most important and studied query in spatial databases and GIS, in [9] a variety of techniques for performing a spatial join are reviewed. Depending on the existence of indexes or not, different spatial join algorithms have been proposed [16]. If both inputs are indexed, several contributions have been proposed [17, 18, 19, 20, 21]. The most influential one in this category is the R-tree join algorithm (*RJ*) [19], due to its efficiency and the popularity of R-trees [22, 11]. *RJ* synchronously traverses both trees in a Depth-First order, and 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. On the other hand, [21] proposes a Breadth-First traversal order for the R-tree join with global optimizations that sorts the output at each level in order to reduce the number of page accesses. Recently, in [23] a new interactive spatial query processing technique for GIS is proposed when two R-trees are processed for spatial join queries.

Most research after *RJ*, focused on spatial join processing when one or both inputs are non-indexed. If just only one data set (let P) is indexed, a common method [24, 25] is to build an R-tree for Q (i.e. to use the existing R-tree R_P as a skeleton to build a *seeded tree* for the non-indexed input) and then apply *RJ*. Another spatial join consist of spatially sorting the non-indexed objects but, instead of building the packed tree, it matches each in-memory created leaf node with the leaf nodes of the existing tree that intersect it [26]. In [14] several spatial joins strategies when only one input data set is indexed are investigated. The main contribution is a method that modifies the *Classic Plane-Sweep* algorithm. This approach reads the data pages from the index in a one-dimensional sorted order and insert entire data pages into the sweep structure (i.e. in this case, one sweep structure will contain objects, while the other sweep structure will contain data pages). Finally, the *slot index spatial join* [27] applies hash-join, using the structure of the existing R-tree to determine the extents of the spatial partitions.

Directly related to this paper, if both data sets are non-indexed, the most representative methods include sorting and external memory plane-sweep [13, 12], or spatial hash join algorithms [24], like partition based spatial merge join [28]. 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 [28] 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. Another algorithm based on regular space decomposition is the *Size Separation Spatial Join* [29]. It avoids replication of objects during the partitioning phase by introducing more than one partition layers (introducing the concept of *filter tree*). Each object is assigned in a single partition, but one partition may be joined with many upper layers. The number of layers is usually small enough for one partition from each layer to fit in memory, thus multiple scans during the join phase are not needed and therefore speeds up the join. In [30] several improvements of two previous join algorithms were proposed. In particular, it deals with

the impact of data redundancy and duplicate detection on the performance of theses methods. *Spatial hash-join* [31] avoids duplicate results by performing an irregular decomposition of space, based on the data distribution of the built input. Finally, in [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 queries has received significant research attention by the computational geometry community (see [32] 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 [15, 5, 6]. A KCPQ-DF algorithm visits the roots of the two R-trees (R_P and R_Q) and recursively follow the pair of MBRs $\langle M_P, M_Q \rangle$, $M_P \in R_P$ and $M_Q \in R_Q$, whose $MINMINDIST$ [5] is the minimum among all possible pairs. The process is repeated recursively until the leaf levels are reached, where potential closest pairs are found. During backtracking to the upper levels, the algorithm only visits MBRs whose $MINMINDIST(M_P, M_Q)$ is smaller than or equal to the distance of the K-th closest pair found so far. A KCPQ-BF algorithm keeps a *binary min-heap* with tuples of the following structure: $\langle M_P, M_Q, MINMINDIST(M_P, M_Q) \rangle$ and the pair of MBRs with the minimum $MINMINDIST$ is visited first. The corresponding tuple is replaced with tuples of the form $\langle M_{P_i}, M_{Q_j}, MINMINDIST(M_{P_i}, M_{Q_j}) \rangle$ for each MBR M_{P_i} in the node pointed by M_P and each MBR M_{Q_j} pointed by M_Q. This algorithm is I/O optimal because it only visits the pairs nodes necessary for obtaining the closest pairs. In [15], 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 [15]. A Recursive Best-First Search (RBF) algorithm for DBQ between spatial objects indexed in R-trees was presented in [33], 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 [34], 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 that just only one data set is indexed, recently in [35] a new algorithm has been proposed for KCPQ, which 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 [36]) and to make use of the properties of a set of distance functions defined between two MBRs [6].

If both data sets are non-indexed, the only approach published until now for KCPQ is [37], which addresses the case where neither data set has a spatial index. They proposed an algorithm that considers two stages: in a first stage, the algorithm partitions both sets of points into buckets, assigning to each bucket a memory buffer and an MBR that includes all the points in the bucket and, a pointer to a list of disk blocks, or a file, where the objects from the bucket are stored. In a subsequent stage, the algorithm processes the lists of objects by means of the metrics defined between MBRs [6].

εDJQ, also know 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 [38], 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 [39], 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 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 [40] a new index structure, called $bRdnn-Tree$, to solve different distance-based join queries is proposed. Moreover, as conclusion is shown that their approach outperforms previous R-tree-based algorithms for KCPQ. Other variants of KCPQ have also been studied in the literature. More specifically, approximate K closest pairs in high dimensional data [41, 42] and constrained K closest pairs [43, 44, 45] have been presented. In [46] the *exclusive closest pairs* problem is introduced (which is a spatial assignment problem) and several solutions that solve it in main memory are proposed, exploiting the space partitioning. Moreover, a dynamic version of the problem is also presented, where the objective is to continuously monitor the exclusive closest pair join solution, in an environment where the joined data sets change positions and content. And recently, in [47] 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. Efficient internal and external memory algorithms are proposed with a theoretical analysis which shows that the expected performance of the algorithms is optimal when two or less attributes are involved. Moreover, such approach does not require any pre-built indexes, is easy to implement and has low memory requirement. In [48] top-K similarity join queries over multi-valued objects is studied. *Quantile based distance* is applied to explore the relative instance distribution among the multiple instances of objects. And efficient and effective techniques to process top-K similarity joins over multi-valued objects are developed following a filtering-refinement framework.

Other complex DJQs using R-trees have been studied in the literature of the spatial databases, as *Iceberg Distance Join* and *K Nearest Neighbors Join* queries. In [49], the Iceberg Distance Join Query is studied for hash-based algorithms and index-based methods (R-trees). It involves two spatial data sets, a distance threshold ε and a cardinality threshold K ($K > 0$). The answer is a set of pairs of spatial objects from the two input data sets that are within distance ε from each other, provided that the first spatial object appears at least K times in the join result. On the other hand, in [50], the *K Nearest Neighbors Join Query* (KNNJ) is studied for R-tree-based data structures. This DJQ involves two spatial data sets and a cardinality threshold K ($K > 0$). The answer is a set of pairs of spatial objects from the two input data sets that includes, for each of the spatial objects of the first data set, the pairs formed with each of its K nearest neighbors in the second data set.

Closely related to Distance-based Join processing is the All-Nearest-Neighbor (ANN) query. The first work on ANN was [51], which suggests two approaches to address the ANN problem when the inner data set is indexed: *Multiple nearest neighbor search* (MNN), and *Batched nearest neighbor search* (BNN). For the case where neither data set has an index, they also propose a hash-based method using spatial hashing introduced in [28]. In [52], the R*-tree and a Quadtree index enhanced with MBR keys for the internal nodes (MBRQuadtree) have been compared with respect to the ANN query. For this operation, a new distance metric between two MBRs was proposed, called NXNDIST (the minimum MINMAXDIST [6]). As conclusion, they showed that for ANN queries, using the MBRQuadtree is a much more efficient indexing structure than the R*-tree index.

Recently, in [53] the *plane-sweep* technique is used to obtain the α-Distance for spatial query processing for fuzzy objects. Essentially, the computation of the α-Distance is to find the closest pair of qualified points of two fuzzy objects. The main property of this variant of the *plane-sweep* method is the use of two *sweeplines* to facility the search for the particular types of spatial queries with fuzzy objects, that has been presented in such research work.

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 [19, 13, 12] and KCPQ [10, 6]).

3.1. Classic Plane-Sweep Algorithm

In general, if we assume the spatial object are points (the data sets are P and Q and they can be organized as arrays) and a distance threshold δ, the *Classic Plane-Sweep* algorithm, applying the *plane-sweep* technique

to both sets of points stored in two arrays having the distance δ (upper bound) as the kernel of the processing, consists of the following steps:

1. Sorting the entries of the two arrays of points, based on the coordinates of one of the axes in increasing or decreasing order. The axis for the *sweepline* can be established based on sweeping axis criteria (e.g. X-axis) and the order can be fixed by sweeping direction criteria (e.g. *forward sweep* (increasing order) or *backward sweep* (decreasing order)), both criteria are presented in [10].
2. After that, two pointers are maintained initially pointing to the first entry for processing of each sorted array of points. Assuming that X-axis is the sweeping axis and the order is increasing (from left to right, i.e *forward sweep*), let the *reference* point be the point with the smallest X-value pointed by one of these two pointers, e.g. P, then the *reference* point is initialized to this point, $P[i]$.
3. Afterwards, the *reference* point must be paired up with the points stored in the other sorted array of points (called *comparison* points, $Q[j] \in Q$) from left to right, satisfying $dx \equiv Q[j].x - P[i].x < \delta$, 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 $dx \equiv Q[j].x - P[i].x \geq \delta$ 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 (e.g. the best axis according to the criteria suggested in [10]). 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. Note that, this kind of processing is called *forward sweep*, since it scans from left to right the *sorted* sets in order to obtain pairs of points that will be a distance smaller than or equal to δ.

Clearly, the application of this technique can be viewed as a *sliding strip* 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 *strip*, see Figure 1. If in the Algorithm 1 we remove lines 12 and 25, then we will get the *Classic Plane-Sweep* algorithm with *sliding strip*.

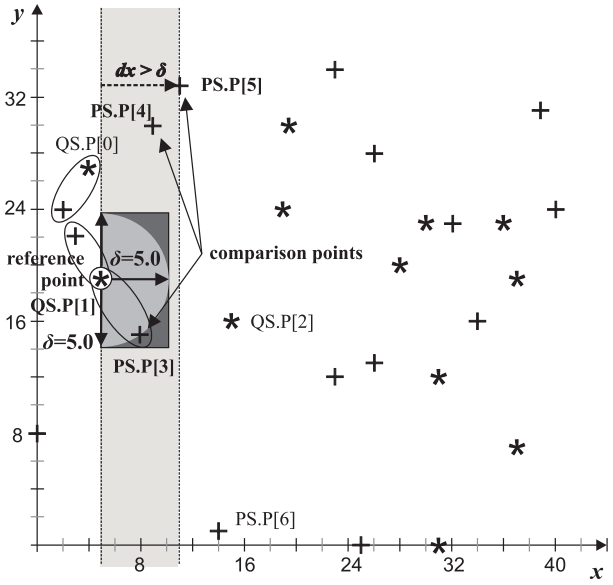

FIGURE 1. Classic Plane-Sweep Algorithm using sliding strip, window and semi-circle.

The adaptation of the algorithm from KCPQ to εDJQ is not so difficult. 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. That means the size of the result of this query is not known a priori and the $MaxKHeap$ is not needed, and hence the distance threshold will be ε_2 instead of δ. Therefore, the data structure that holds the result set will be a file of records (*resultFile*), each of one with three fields $(dist, P[i], Q[j])$.

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 to the *reference* point in order to avoid unnecessary distance computations (that involve *square roots*) which are the most expensive operations for DJQs [54]. The proposed approach makes use the *plane-sweep* technique and the *restricting the search space*.

Notice that in Figure 1, 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 strip* 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). If in the Algorithm 1 we replace line 12 by **if** ($|PS.P[i].y - QS.P[k].y| <$ *key dist of MaxKHeap root*) and line 25 by **if** ($|PS.P[k].y - QS.P[j].y| <$ *key dist of MaxKHeap root*), then we will get a new variant of *Classic Plane-Sweep* algorithm. 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*.

2. If we try to reduce even more the search space, we can only select those points inside the *semi-circle* centered in 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 in the *reference* point $reference = (reference.x, reference.y) \in P$ and radius δ is $circle(reference, t, \delta) \equiv (reference.x - t.x)^2 + (reference.y - t.y)^2 < \delta^2$). See the Algorithm 1 at the lines 12 and 25, and for this reason we call to 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 the Algorithm 1 to know how this improvement works on two X-sorted arrays of points $PS.P$ and $QS.P$, considering the sweeping axis the X-axis, which it returns a (binary) max-heap [55] with the K closest pairs ($MaxKHeap$). Notice $MaxKHeap$ should be initialize to K pairs of points, having all of them a distance ∞. And when a new pair of points $(dist, PS.P[i], QS.P[j])$ is chosen to be inserted in $MaxKHeap$, the insertion process consists of removing the pair with the maximum distance (root of the $MaxKHeap$) and adding the *newPair*, reorganizing the data structure to restore the (binary) max-heap property based on *dist*. See in Figure 1, the *semi-circle*, in light grey color, centered in the *reference* point. 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 distance-based join query.

We must highlight that PS and QS are two strips (sorted data sets on a sweeping axis), such as $PS = \{first, start, end, P[0..n-1]\}, QS = \{first, start, end, P[0..m-1]\}$, where *first* is the absolute index of the first point of the array in the set; *start* and *end* are local relative indexes of calculating part of array of points; and P is a sorted array of n or m ($\geq numOfPoints$ per strip) points, which is a subset of set P or Q.

Below, we provide a proof of the correctness of the *Classic Circle Plane-Sweep* algorithm for KCPQ ($CCPS$) algorithm (Algorithm 1) through the Theorem 3.1.

Theorem 3.1. (**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.*

Proof. Let the *reference* point be the leftmost point of both arrays that has not been processed yet.It can belong to $PS.P$ or $QS.P$ depending of their X-coordinate value (line 3). That is, if $PS.P[i] < QS.P[j]$ the reference point is $PS.P[i]$. It is $QS.P[j]$ in case $QS.P[j] \leq PS.P[i]$. The selection of the *reference* point without any repetition is guided by the order determined by the sweeping direction (from left to right) over the sweeping axis, since the index i of $PS.P$ and the index j of $QS.P$ are always incremented by one (lines 15 and 28) when they are processed.

Let the *comparison* points be all points of the array that does not contain the *reference* point. That is, if $PS.P[i]$ is the *reference* point, the *comparison* points will be the subset of all points $QS.P[k]$ such that $j \leq k \leq QS.end$ (line 4); and if $QS.P[j]$ is the *reference* point, the *comparison* points will be the subset of all points $PS.P[k]$ such that $i \leq k \leq PS.end$ (line 17). The index of the *comparison* points, k, is always incremented by one.

From the previous way of creating pair of points from $PS.P$ and $QS.P$ in the form $(reference, comparison)$, we can conclude that the $CCPS$ algorithm generates at most $n \times m$ possible pairs of points correctly and without any repetition.

Now, in order to prove that $MaxKHeap$ contains, at the end of the execution of the algorithm, at least K closest pairs of points from all possible pairs generated from $PS.P$ and $QS.P$ (since the algorithm starts with an empty $MaxKHeap$, and the first K pairs created will be inserted in the $MaxKHeap$, $K \leq |PS.P| \times |QS.P|$), we are going to study the different types of pairs generated by the algorithm:

Category 1. Pairs of points that are never generated by the algorithm due to their dx distance value. That is, when one pair *(reference, comparison)* with $dx(reference, comparison) \geq$ *key dist of $MaxKHeap$ root* is discovered (lines 9, 10 or 22, 23), then all other pairs of the form *(reference, comparison')*, where *comparison'* is on the right of *com-*

parison, are not generated (lines 11 or 24), since they have dx distance $\geq$ *key dist of MaxKHeap root*.

Category 2. Pairs of points generated by the algorithm with $dx(reference, comparison) <$ *key dist of MaxKHeap root*, but not inserted into $MaxKHeap$, due to having an actual distance from the *reference* point larger than *key dist of MaxKHeap root*. That is, those pairs that are outside the circle centered at the reference point with radius *key dist of MaxKHeap root* are discarded. They are rejected at the **else** statement of lines 12 and 25.

Category 3. Pairs of points generated by the algorithm, inserted into $MaxKHeap$ and later removed from it. That is, those pairs that are inside the circle centered at the reference point with radius *key dist of MaxKHeap root* are inserted into $MaxKHeap$, but later they will be deleted from it because the $MaxKHeap$ needs to host another pair with a smaller *dist* value than them (lines 13, 14, 26 and 27).

Category 4. Pairs of points generated by the algorithm, inserted into $MaxKHeap$ and not removed from it until the end of the execution of the algorithm. That is, the pairs that are inside the circle centered at the *reference point* with radius *key dist of MaxKHeap root* that are inserted and remain into $MaxKHeap$, since their *dist* values are always smaller than *key dist of MaxKHeap root* (lines 13, 14, 26 and 27).

Moreover, we must highlight that the pairs of points generated by the algorithm and inserted in $MaxKHeap$ while it is not full belong to categories 3 or 4. That is, the first K pairs of points generated by the algorithm are always inserted in the $MaxKHeap$, since *key dist of MaxKHeap root* $= \infty$, and they can be removed from or remain into the heap until the end of the execution of the algorithm (lines 5, 6, 7, 17, 18 and 19).

Thanks to the (binary) max-heap property, every pair stored at the $MaxKHeap$ as part of the final result has a *dist* smaller than or equal to the *dist* of the pair at the root of the $MaxKHeap$. Moreover, every pair generated by the algorithm and not stored at the $MaxKHeap$ as part of the final result has a *dist* larger than or equal to the *dist* of the pair in the root of the $MaxKHeap$.

So, it proves that each of the K pairs of the final result has a *dist* smaller than or equal to the *dist* of every pair not generated by the algorithm (category 1), generated by the algorithm and not inserted into the $MaxKHeap$ (category 2), or temporarily inserted into $MaxKHeap$, but removed later (category 3). □

Moreover, as we know from [19], that 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. In the same line, $CCPS$ 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).

Here, we mush highlight that PS and QS are two strips (sorted data sets over a sweeping axis), which are subsets of sets P and Q. They are defined as $PS = \{first, start, end, P[0..n-1]\}$, and $QS = \{first, start, end, P[0..m-1]\}$, where $first$ is the absolute index of the first point of the array $P[\cdots]$; *start* and *end* are local relative indexes of calculating part of array of points; and $P[\cdots]$ is a sorted array of n and m points, respectively.

3.3. Extension to εDistance Join Query

The adaptation of the $CCPS$ algorithm from KCPQ to εDJQ is not so difficult, and we will get *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 12 and 25: **if** ($dist \geq \varepsilon_1$ **and** $dist \leq \varepsilon_2$)). That means the result of this query must not be ordered and the $MaxKHeap$ is unnecessary (lines 5, 6, 7 and 8; and lines 18, 19, 20, and 21), because for 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 10: **if** ($QS.P[k].x - PS.P[i].x \geq \varepsilon_2$), line 23: **if** ($PS.P[k].x - QS.P[j].x \geq \varepsilon_2$), line 12: **if** ($(QS.P[k].x - PS.P[i].x)^2 + (QS.P[k].y - PS.P[i].y)^2 < (\varepsilon_2)^2$) and line 25: **if** ($(PS.P[k].x - QS.P[j].x)^2 + (PS.P[k].y - QS.P[j].y)^2 < (\varepsilon_2)^2$)). Therefore, the data structure that holds the result set will be a file of records (*resultFile*), each one with three fields ($dist, PS.P[i], QS.P[j]$). The modifications of this storage are in the lines 14 and 27, where we have to replace them by $resultFile.write(newPair)$. To accelerate the storing on the *resultFile* we will 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 [56]. It consists of reporting groups of nearby pairs of points instead of every join link separately. This phenomenon is known as *output explosion* [56] and it can appear when data density of the sets of points is locally very large compared to the range of distances (distance threshold), 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 3.1 for the $CCPS$ algorithm for KCPQ.

Algorithm 1 CCPS

Input: $PS.P[0..n-1], QS.P[0..m-1]$: X-sorted arrays of points. $MaxKHeap$: Max-Heap storing $K > 0$ pairs
Output: $MaxKHeap$: Max-Heap storing the K closest pairs between PS and QS

```
1: i = PS.start   j = QS.start
2: while i ≤ PS.end and j ≤ QS.end do
3:     if PS.P[i].x < QS.P[j].x then                          ▷ PS.P[i]: reference point
4:         for k = j to QS.end increment k do                 ▷ QS.P[k]: current comparison point
5:             if MaxKHeap is not full then
6:                 calculate distance dist between PS.P[i] and QS.P[k]
7:                 insert (PS.P[i], QS.P[k]) with key dist into MaxKHeap
8:             else
9:                 calculate x-distance dx between PS.P[i] and QS.P[k]
10:                if dx ≥ key dist of MaxKHeap root then
11:                    break
12:                if (PS.P[i].x − QS.P[k].x)^2 + (PS.P[i].y − QS.P[k].y)^2 < (key dist of MaxKHeap root)^2 then
13:                    calculate distance dist between PS.P[i] and QS.P[k]
14:                    insert (PS.P[i], QS.P[k]) with key dist into MaxKHeap
15:        increment i
16:    else                                                   ▷ PS.P[i] ≥ QS.P[j] and QS.P[j]: reference point
17:        for k = i to PS.end increment k do                 ▷ PS.P[k]: current comparison point
18:            if MaxKHeap is not full then
19:                calculate distance dist between PS.P[k] and QS.P[j]
20:                insert (PS.P[k], QS.P[j]) with key dist into MaxKHeap
21:            else
22:                calculate x-distance dx between PS.P[k] and QS.P[j]
23:                if dx ≥ key dist of MaxKHeap root then
24:                    break
25:                if (PS.P[k].x − QS.P[j].x)^2 + (PS.P[k].y − QS.P[j].y)^2 < (key dist of MaxKHeap root)^2 then
26:                    calculate distance dist between PS.P[k] and QS.P[j]
27:                    insert (PS.P[k], QS.P[j]) with key dist into MaxKHeap
28:        increment j
```

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 [57]. 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) the *comparison* points are processed 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 KCPQ

The *Reverse Run Plane-Sweep* (*RRPS*) algorithm [57] is based on two concepts. First, every point that is used as a *reference* point forms a *run* with other subsequent points of the same set. A *run* is a continuous sequence of points of the same set that doesn't contain any point from the other set. 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).

The *Reverse Run Circle Plane-Sweep* algorithm for KCPQ (*RCPS*) algorithm is depicted in Algorithm 2, which it is *RRPS* 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 in the *reference* point with radius δ, then this pair with its distance *dist* is

inserted in the $MaxKHeap$ *(rule 1)*. 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 other set must be updated at the index value of the point being compared (a comparison with a point of the other set having an index value smaller than or equal to the updated *left limit* will have X-distance larger than dx and is unnecessary).

Moreover, if the rightmost current *comparison* point has an index value 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)*.

For the $RCPS$ algorithm (Algorithm 2), remind that PS and QS are two strips (sorted data sets over a sweeping axis), such as $PS = \{first, start, end, P[0..n-1]\}, QS = \{first, start, end, P[0..m-1]\}$, where $first$ is the absolute index of the first point of the array in the strip; *start* and *end* are local relative indexes of calculating part of array of points; and $P[\cdots]$ is a sorted array of n and m points, respectively. $leftp$ and $leftq$ are variables that hold the local left limits of the given strips. $gleftp$ and $gleftq$ are variables that hold the global left limits of the given set. *sav_sentp* and *sav_sentq* are variables that save the current X-coordinate of the points that follow the last point of each set. They are necessary because these points must be the sentinels of the strips. *stop_run* is a variable that stores the end of the X-coordinates of the current run of set $PS.P$ or $QS.P$. *run_setP* is a variable whose value is set to *false* when $PS.P[i] < QS.P[j]$, then the current *active* run will get $reference$ points from the set $QS.P$ (starting from the index j) and the *comparison* points will be from set $PS.P$ (starting from the index $i-1$). Analogously, *run_setP* is set to *true* when $QS.P[j] < PS.P[i]$, then the current *active* run will get $reference$ points from the set $PS.P$ (starting from the index i) and the *comparison* points will be from set $QS.P$ (starting from the index $j-1$).

Theorem 4.1. (**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 RCPS Algorithm outputs K closest pairs of points from $PS.P$ and $QS.P$ correctly and without any repetition.*

Proof. The proof of correctness is similar to the proof for the *CCPS* algorithm given for Theorem 3.1. To extend that proof, we must keep in mind the concepts of *active run*, *reference point*, *left limits* and *comparison points*.

As in Theorem 3.1, the $RCPS$ algorithm generates candidates pairs correctly and without any repetition, which is based on *sorting* of both arrays by the sweeping axis, and the configuration of the *active runs* with respect to the *reference points*, which are guided by this order, taking into account that a *run* is a continuous sequence of points of the same array that does not contain any point from the other array.

We will prove this, since in the $RCPS$ algorithm, the *reference point* at each given moment is either $PS.P[i]$ (lines 21-33) or $QS.P[j]$ (lines 39-51), depending on which of the two arrays gives the *active run*. When all the pairs with *reference point* $PS.P[i]$ and *comparison points* $QS.P[k]$ where $leftq < k < j$ are created, then index i is incremented by one; and if the (new) point $PS.P[i+1]$ belongs to the *active run*, then it will be the next *reference point*. If it is not part of the *active run*, then $PS.P[i+1]$ belongs to the next run of the same array. In this case, the *active run* will be determined by $QS.P[j]$ as a starting point. Thus, eventually $PS.P[i+1]$ will be set as a *reference point*, unless it is guaranteed that it is not necessary to do this (lines 19-20).

Moreover, since index i is never decremented, there is no way that $PS.P[i]$ will ever be redefined as a *reference point*. So there is no way to have duplications or repetitions (this is also valid for every point $QS.P[j]$ of the other array). Thus, each point of each array is defined only once as a *reference point* and all points of the other array must be defined as *comparison points* (subject to Rule 3 (line 19), or Rule 3 (line 37)). Finally, from the previous way of creating pair of points from $PS.P$ and $QS.P$ in the form $(reference, comparison)$, we can conclude that the $RCPS$ algorithm generates at most $n \times m$ possible pairs of points correctly and without any repetition.

As in Theorem 3.1, to prove that $MaxKHeap$ contains, at the end of the execution of the algorithm, at least K closest pairs of points from all possible pairs generated from $PS.P$ and $QS.P$ (since the algorithm starts with an empty $MaxKHeap$, and the first K pairs will be inserted in the $MaxKHeap$, $K \leq |PS.P| \times |QS.P|$), we are going to study the different types of pairs generated by the algorithm:

Category 1. Pairs of points that are never generated by the algorithm due to the left limits (*leftq* and *leftp*) and the dx distance function. That is, when one pair *(reference, comparison)* with $dx(reference, comparison) \geq$ *key dist of* $MaxKHeap$ *root* is discovered (rule 2, lines 27 or 45), then all pairs of the form *(refer-*

Algorithm 2 RCPS

Input: $PS.P[0..n-1], QS.P[0..m-1]$: 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: $i = PS.start \quad leftp = i - 1 \quad j = QS.start \quad leftq = j - 1$
2: $sav\_sentp = PS.P[PS.end + 1] \quad sav\_sentq = QS.P[QS.end + 1]$
3: $PS.P[PS.end + 1].x = QS.P[QS.end + 1] = DBL\_MAX$ ▷ initialize the sentinels to DBL_MAX (∞)
4: **if** $PS.P[i].x < QS.P[j].x$ **then** ▷ find the most left point of two data sets
5: **if** $PS.P[PS.end].x \leq QS.P[j].x$ **then** ▷ the two sets do not overlap
6: $i = PS.end + 1 \quad PS.P[PS.end + 1].x = QS.P[QS.end].x + 1$
7: **else** ▷ the two sets are overlapped, skip the first run of the set PS
8: find in the strip PS, the first point $PS.P[i]$ that satisfies $PS.P[i].x \geq QS.P[j].x$ and update i
9: $stop\_run = PS.P[i].x \quad run\_setP =$ FALSE ▷ stop the run of QS set at the start of the second run of the PS set
10: **else**
11: **if** $QS.P[QS.end].x \leq PS.P[i].x$ **then** ▷ the two sets do not overlap
12: $j = QS.end + 1$
13: **else** ▷ the two sets are overlapped, skip the first run of the set QS
14: find in the strip QS, the first point $QS.P[j]$ that satisfies $QS.P[j].x > PS.P[i].x$ and update j
15: $stop\_run = QS.P[j].x \quad run\_setP =$ TRUE ▷ stop the run of PS set at the start of the second run of the QS set
16: **while** $i \leq PS.end$ **or** $j \leq QS.end$ **do** ▷ while even one data set is not finished
17: **if** $run\_setP =$ TRUE **then** ▷ the active run is from the PS set
18: **while** $PS.P[i].x < stop\_run$ **do** ▷ while active run unfinished. $PS.P[i]$: reference point
19: **if** $j - 1 = leftq$ **then** ▷ $QS.P[j-1]$: last current comparison point - rule 3
20: advance i to next PS-run and **break** ▷ **while**
21: **for** $k = j - 1$ **downto** $leftq + 1$ **decrement** k **do** ▷ $QS.P[k]$: current comparison point
22: **if** $MaxKHeap$ is not full **then**
23: calculate distance $dist$ between $PS.P[i]$ and $QS.P[k]$
24: insert ($PS.P[i]$, $QS.P[k]$) with key $dist$ into $MaxKHeap$
25: **else**
26: calculate x-distance dx between $PS.P[i]$ and $QS.P[k]$
27: **if** $dx \geq$ key $dist$ of $MaxKHeap$ root **then** ▷ $dx \geq \delta$ - rule 2
28: $leftq = k$ ▷ update the local value of the left limit
29: $gleftq = QS.first + k$ ▷ update the global value of the left limit
30: **break** ▷ **for**
31: **if** $(PS.P[i].x - QS.P[k].x)^2 + (PS.P[i].y - QS.P[k].y)^2 <$ (key $dist$ of $MaxKHeap$ root)^2 **then** ▷ rule 1
32: calculate distance $dist$ between $PS.P[i]$ and $QS.P[k]$
33: insert ($PS.P[i], QS.P[k]$) with key $dist$ into $MaxKHeap$
34: **increment** i ▷ update the reference point $PS.P[i]$
35: $PS.P[PS.end + 1].x = QS.P[QS.end].x + 1 \quad stop\_run = PS.P[i].x$ ▷ now the active run is from the QS set
36: **while** $QS.P[j].x \leq stop\_run$ **do** ▷ while active run unfinished. $QS.P[j]$: reference point
37: **if** $i - 1 = leftp$ **then** ▷ $PS.P[i-1]$: last current point - rule 3
38: advance j to the next QS-run and **break** ▷ **while**
39: **for** $k = i - 1$ **downto** $leftp + 1$ **decrement** k **do** ▷ $PS.P[k]$: current comparison point
40: **if** $MaxKHeap$ is not full **then**
41: calculate distance $dist$ between $PS.P[k]$ and $QS.P[j]$
42: insert ($PS.P[k]$, $QS.P[j]$) with key $dist$ into $MaxKHeap$
43: **else**
44: calculate x-distance dx between $PS.P[k]$ and $QS.P[j]$
45: **if** $dx \geq$ key $dist$ of $MaxKHeap$ root **then** ▷ $dx \geq \delta$ - rule 2
46: $leftp = k$ ▷ update the local value of the left limit
47: $gleftp = PS.first + k$ ▷ update the global value of the left limit
48: **break** ▷ **for**
49: **if** $(PS.P[k].x - QS.P[j].x)^2 + (PS.P[k].y - QS.P[j].y)^2 <$ (key $dist$ of $MaxKHeap$ root)^2 **then** ▷ rule 1
50: calculate distance $dist$ between $PS.P[k]$ and $QS.P[j]$
51: insert ($PS.P[k]$, $QS.P[j]$) with key $dist$ into $MaxKHeap$
52: **increment** j ▷ update the reference point $QS.P[j]$
53: $PS.P[PS.end + 1].x = QS.P[QS.end + 1].x$ ▷ revert the PS sentinel at the maximum real X-value (DBL_MAX)
54: $stop\_run = QS.P[j].x \quad run\_setP =$ TRUE
55: $PS.P[PS.end].x = sav\_sentp \quad QS.P[QS.end].x = sav\_senq$ ▷ revert the original values

ence, comparison'), where *comparison'* is to the left of *comparison* and on the right of the left limit, are not generated (lines 30 or 48), since they have dx distance $\geq$ *key dist of MaxKHeap root*. In addition, the left limit is updated with the index of *comparison* (lines 28 or 46), preventing the generation of pairs of the form *(reference', comparison)*, where *reference'* is on the right of *reference*, in future iterations of the algorithm. Especially, if the left limit is updated with the index of the first *comparison* point (rule 3, lines 19 or 37), then all the rest *reference* points of the current run are skipped (lines 20 or 38).

Category 2. Pairs of points generated by the algorithm with $dx(reference, comparison) <$ *key dist of MaxKHeap root*, but not inserted into $MaxKHeap$, due to having an actual distance from the *reference* point larger than *key dist of MaxKHeap root*. That is, those pairs that are outside the circle centered at the reference point with radius *key dist of MaxKHeap root* are discarded. They are rejected due to Rule 1, lines 31 or 49.

Category 3. Pairs of points generated by the algorithm, inserted into $MaxKHeap$ and later removed from it. That is, those pairs inside the circle centered at the reference point with radius *key dist of MaxKHeap root* that are inserted into $MaxKHeap$, but later are deleted from it because the $MaxKHeap$ needs to host another pair with a smaller *dist* value than them (lines 32, 33, 50 and 51).

Category 4. Pairs of points generated by the al-

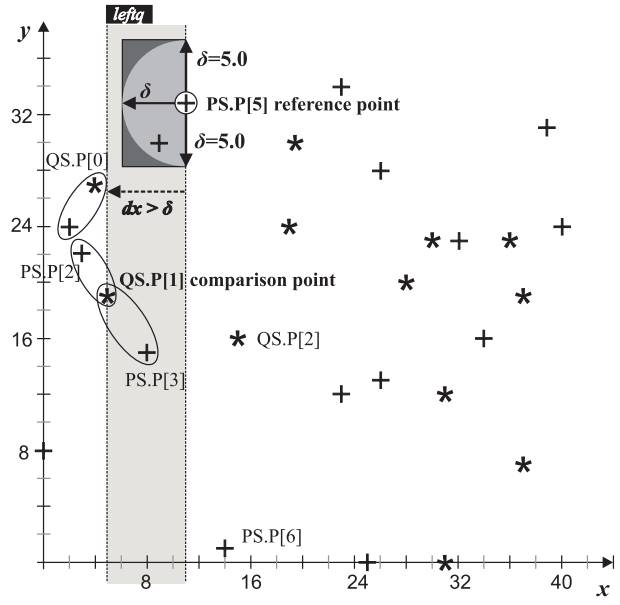

FIGURE 2. Reverse Run Plane-Sweep algorithm using sliding strip, window and semi-circle.

gorithm, inserted into $MaxKHeap$ and not removed from it until the end of the execution of the algorithm. That is, the pairs that are inside the circle centered at the *reference point* with radius *key dist of MaxKHeap root* that are inserted and remain into $MaxKHeap$, since their *dist* values are always smaller than *key dist of MaxKHeap root* (lines 32, 33, 50 and 51).
Moreover, we must highlight that the pairs of points generated by the algorithm and inserted in $MaxKHeap$ while it is not full belong to categories 3 or 4. That is, the first K pairs of points generated by the algorithm are always inserted in the $MaxKHeap$, since *key dist of* $MaxKHeap$ *root* $= \infty$, and they can be removed from or remain into the heap until the end of the execution of the algorithm (lines 22, 23, 24, 40, 41 and 42).
An finally, thanks to the (binary) max-heap property, every pair stored at the $MaxKHeap$ as part of the final result has a *dist* smaller than or equal to the *dist* of the pair in the root of the $MaxKHeap$. Moreover, every pair generated by the algorithm and not stored at the $MaxKHeap$ as part of the final result has a *dist* larger than or equal to the *dist* of the pair in the root of the $MaxKHeap$.
So, it is proved that each of the K pairs of the final result has a *dist* smaller than or equal to the *dist* of every pair not generated by the algorithm (category 1), generated by the algorithm and not inserted into the $MaxKHeap$ (category 2), or temporarily inserted into $MaxKHeap$, but removed later (category 3). □

The following example illustrates the operation of the algorithm. Let's consider the points of Figure 2, presented, in commonly sorted X-order, in Table 1. The algorithm starts initializing the local variables $i = 0$ (equal to $PS.start$ since the first run of $PS.P$ array starts at $PS.P[PS.start]$), $j = 0$ (equal to $QS.start$ since the first run of $QS.P$ array starts at $QS.P[QS.start]$), and the local left limits ($leftp = -1$ for the array $PS.P$ and $leftq = -1$ for the array $QS.P$) since the last $PS.P$ ($QS.P$) point to be used in the comparison is $PS.P[leftp + 1]$ ($QS.P[leftq + 1]$) (line 1). Saves the values of the points after the last of each array in two local variables *sav_setp* and *sav_sentq*, because it is possible to get as arguments parts of the arrays (from $PS.P[PS.start]$ to $PS.P[PS.end]$ while $PS.end < PS.P.n$) (line 2). To simplify the algorithm operation (the stopping conditions), a sentinel point with X-coordinate equal to ∞ is added to each array in the point after the last (line 3).

Since $PS.P[0].x < QS.P[0].x$ is true (line 4), and $PS.P[15].x \leq QS.P[0].x$ is false (line 5), the value of i is advanced to 3 ($i = 3$), since $PS.P[3].x$ is the first point of $PS.P$ which has X-coordinate greater than the $QS.P[0].x$ (line 8), the *stop_run* value is set to 8 ($PS.P[3].x$) and flag *run_setP* is set to $FALSE$ (i.e. the first *active run* is set from the $QS.P$ array having all points with X-coordinates ($QS.P[0], QS.P[1]$) smaller than or equal to the *stop_run* value) (line 9). From the lines 16 and 17 the processing of the algorithm jumps to the line 35, just at the end of the part of the algorithm in which the *active run* is from the $PS.P$ array (lines 17 - 34).

The sentinel of $PS.P$ is set to $PS.P[PS.end+1].x = 38$ (38 is the value larger than the X-coordinate of the last point of the $QS.P$ array and smaller than the sentinel of the $QS.P$ array, i.e. $QS.P[QS.end].x + 1$). Moreover, the *stop_run* value is set to equal to 8 (the X-coordinate of the first point of the next run of the $PS.P$ array, i.e. $PS.P[3].x$) (line 35). The *active run* consists of $QS.P[0]$ and $QS.P[1]$ ($QS.P[1]$ is the last point of $QS.P$ before $PS.P[4]$) and these points are compared with each of the current *comparison* points of $PS.P$ (in reverse X-order) which form the sequence $PS.P[i-1], \ldots, PS.P[leftp + 1]$ ($PS.P[2], PS.P[1], PS.P[0]$). The condition $QS.P[0].x \leq stop\_run$ ($4 \leq 8$) is true (line 36) and the processing of the first run of the $QS.P$ array is starting from this point. The condition $i - 1 = leftp$ (2 = -1) is not true (line 37) so the process of the *rule 3* (line 38) will not be executed. The *for* loop starts with $k = 2$ to $leftp + 1$ (0) (line 39), creating the pairs ($QS.P[0], PS.P[2]$), ($QS.P[0], PS.P[1]$), ($QS.P[0], PS.P[0]$) in the format ($reference, comparison$) pair of points and because the $maxKHeap$ is not full (line 40) the distances of these pairs (*dist*) are calculated at $dist_1 = 5.099$, $dist_2 = 3.606$, $dist_3 = 19.416$ (line 41), and they are inserted in the not full $maxKHeap$ with their keys *dist* values (line 42). The *for* loop (line 39) terminates because of the value of k ($k = -1$) and the local index value of j is incremented by 1 (line 52).

The condition $QS.P[1].x \leq stop\_run$ ($5 \leq 8$) is

still true (line 36) and the *while* loop will continue by setting as the *reference* point to $QS.P[1]$. The condition $i-1 = leftp$ (2 = -1) is not true (line 37) so the process of the *rule 3* (line 38) will not be executed. The *for* loop starts with $k = 2$ to $leftp + 1$ (0) (line 39), creating the pairs $(QS.P[1], PS.P[2])$, $(QS.P[1], PS.P[1])$, $(QS.P[1], PS.P[0])$ and because the $maxKHeap$ is full (line 43), the process of each pair starts by calculating the dx distance of these pairs (i.e. the *rule 2* is checked). For the first one $(QS.P[1], PS.P[2])$ the dx distance ($dx = 2$) is calculated (line 44) and this value is compared with the key $dist$ of $maxKHeap.root$ (line 45). Since this is smaller ($2 < 19.416$), then the algorithm calculates the sum of the squares of dx and dy distances of the current pair and compares this sum with the square of the key $dist$ of $maxKHeap.root$ (line 49). That is $PS.P[k].x - QS.P[j].x)^2 + (PS.P[k].y - QS.P[j].y)^2 < (key\ dist\ of\ MaxKHeap.root)^2$ (*rule 2*) and the distance $dist$ of the pair is calculated ($dist = 3.606$) (line 50). This pair with its key value is inserted in full $maxKHeap$, decreasing the key $dist$ value of the $maxKHeap$ at the value of 5.099 by deleting the previous root (line 51). The *for* loop (line 39) will be continued with $k = 1$ and the lines 44 and 45 are executed like the previous pair (*rule 2* is checked). But the condition for $dist$ of the (line 50) is false and the pair $(QS.P[1], PS.P[1])$ having distance sum of squares of dx and dy of 34, which is greater than 5.0992 and it will not be inserted in the $maxKHeap$. The same chance has the next pair $(QS.P[1], PS.P[0])$, having distance sum of squares of dx and dy of 146. The *for* loop (line 39) continues for $k = 0$ but no pair is inserted in $maxKHeap$ and it terminates because of the value of k ($k = -1$), and the local index value of j is incremented by 1, setting to the value of 2 ($j = 2$) (line 52). The condition $QS.P[2].x \leq stop\_run$ ($15 \leq 8$) now is false (line 36) and *while* loop terminates here. The value of the sentinel of the $PS.P$ array is reverted to ∞ (line 53), the value of $stop\_run$ is set to 15, and the flag run_setP is set to TRUE (line 54), $i = 3$, $j = 2$, and one main loop is finished.

The second iteration will be started (from the line 16) having the *active* run of the PS.P array, since $run\_setP = TRUE$. The *active* run of the $PS.P$ array consists of $PS.P[3], PS.P[4], PS.P[5]$ and $PS.P[6]$ because it is the last point of $PS.P$ before $QS.P[2]$. Each of the points of the *active* run should be compared with each of the current *comparison* points of $QS.P$ (in reverse X-order) which form the sequence $QS.P[j-1], \ldots, QS.P[leftq + 1]$ $(QS.P[0], QS.P[1])$. The conditions of the lines 17 and 18 are true while the condition of the line 19 is false (*rule 3* is checked). The *for* loop starts with $k = 1$ to $leftq + 1$ (0) (line 21), creating the pairs $(PS.P[3], QS.P[1])$ and $(PS.P[3], QS.P[0])$. For the first pair and because the $maxKHeap$ is full (line 25) $dx = 3$ distance is calculated (line 26), it is compared with the key $dist$ of the $maxKHeap.root = 5.099$ (line 27) which is smaller (*rule 2* is checked), and the sum of squares of dx and dy distances is compared with the square of the key $dist$ of the $maxKHeap.root$ (line 31). This sum is equal to 25 so the distance of this pair is calculated at $dist = 5$ (line 32) and it is inserted in the full $maxKHeap$ with its key $dist$ value (line 33) by deleting the previous root and setting the key $dist$ of the $maxKHeap.root$ to 5. The *for* loop (line 21) continues with the value of $k = 0$ processing the pair $(PS.P[3], QS.P[0])$ which is not inserted because of the distance $dist = 12.649$ value and it terminates due to the value of k ($k = -1$). The local index value of i is incremented by 1 ($i = 4$) (line 34). The next pair $(PS.P[4], QS.P[1])$ is not inserted because of the distance $dist = 11.705$. The following pair $(PS.P[4], QS.P[0])$ has the distance $dx = 5$ equal to the key $dist$ of the $maxKHeap.root$ (line 27), the *rule 2* is applied and the distance $dist$ is not calculated and the $leftq$ limit is updated to 0 (*rule 2*). This results the decision that the last *comparison* point from the $QS.P$ array may be the point on the right of the $leftq$ limit, the $QS.P[1]$ point. The *for* loop (line 21) is broken and the index i is incremented to 5. The next pair $(PS.P[5], QS.P[1])$ has a distance $dx = 6$, this value is larger than the key $dist$ of the $maxKHeap.root$ (5) and for this the left limit $leftq$ is advanced to $leftq = 1$. The value of the condition $j - 1 = leftq$ (1 = 1) is true (line 19) and the process will skip the examination of the last point of the *active* run (line 20) (applying the *rule 3*). Figure 2 shows the status of the algorithm right after the examination of $(PS.P[5], QS.P[1])$. Note that the *reference* point is $PS.P[5]$ and the key $dist$ of the $maxKHeap.root = 5.000$. The combination of $PS.P[6]$ to previous points of the $QS.P$ array is skipped because of the value of the left limit ($leftq = 1$) and the condition in line 19 $j - 1 = leftq$ (1 = 1) is true (the *rule 3* is applied), then i is updated to 7 and the *while* loop is broken ($i = 7$ and $j = 2$). Next, the algorithm establishes the $stop\_run$ to 23 ($PS.P[7].x$), the *active* run is now from $QS.P$ which consists of $QS.P[2], QS.P[3]$ and $QS.P[4]$ because they are the last points of $QS.P$ before $PS.P[7]$, and the *comparison* points are from $PS.P$ which form the sequence $PS.P[i-1], \ldots, PS.P[leftp+1]$ $(PS.P[6], \ldots, PS.P[0])$. And the execution of the algorithm continues at line 35 for the current *active* run.

Following analogous steps, the next iterations of the *while* loop at (line 16) examine the next *runs* as they depicted in Table 1, either inserting pairs in the $MaxKHeap$, or not inserting pairs in the $MaxKHeap$, due to their $dist$ distance, or not inserting pairs in the $MaxKHeap$, due to their dx values, advancing the left limits.

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. In the specific example described previously, the $CCPS$ algorithm would perform 9 *dist* distance calculations, 95 dx calculations, 9 $MaxKHeap$ insertions and would examine 62 pairs. $RCPS$ performed 6 *dist* distance calculations, 77 dx calculations, 6 $MaxKHeap$ insertions and examined 49 pairs.

4.2. Extension to εDistance Join Query

As for $\varepsilon CCPS$ for εDJQ, the adaptation of the $RCPS$ algorithm from KCPQ to εDJQ is not so difficult, and we will get *Reverse Run Circle Plane-Sweep* algorithm for εDJQ ($\varepsilon RCPS$). 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 31 and 49: **if** ($dist \geq \varepsilon_1$ **and** $dist \leq \varepsilon_2$)). That means the result of this query must not be ordered and the $MaxKHeap$ is unnecessary (lines 22, 23, 24 and 25; and lines 40, 41, 42, and 43 can be omitted), because for 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* (lines 27, 45, 31 and 49). As like $\varepsilon CCPS$, the data structure that holds the result set will be a file of records (*resultFile*), each one with three fields ($dist, PS.P[i], QS.P[j]$) and the modifications of this storage are in the lines 33 and 51, where we have to replace them by $resultFile.write(newPair)$. Finally, the proof of the correctness of $\varepsilon RCPS$ algorithm is similar to the proof of Theorem 4.1 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 [58]. For instance, to sort P on the X-axis, first P is partitioned in $\lceil P/B \rceil$ 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*. For each file, we maintain, two strips, PS and QS, 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, 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 datasets.

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**. Using the disk-page size, which, in this policy, is equal to the strip 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 datasets. To accomplish this, we use a part of main memory as a buffer (equal to one disk page for each set) which can hold a number of points from 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 two points of the two 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. Next,

i	0	1	2			3	4	5	6				7	8	
$P[i]$	(0,8)	(2,24)	(3,22)			(8,15)	(9,30)	(11,33)	(14,1)				(23,12)	(23,34)	
j				0	1					2	3	4			
$Q[j]$				(4,27)	(5,19)					(15,16)	(18.5,30)	(19,24)			

i	9	10	11					12	13				14	15	16
$P[i]$	(25,0)	(26,13)	(26,28)					(32,23)	(34,16)				(39,31)	(40,24)	$(\infty,-)$
j				5	6	7	8			9	10	11			12
$Q[j]$				(28,20)	(30,23)	(31,0)	(31,32)			(36,23)	(37,7)	(37,19)			$(\infty,-)$

TABLE 1. The points of 2 in X-sorted order.

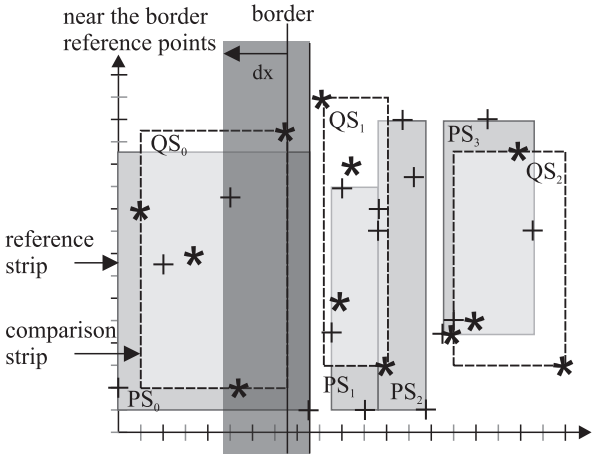

FIGURE 3. Applying the $FCCPS$ algorithm on two data sets partitioned in strips equally full (4 points/strip).

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 partioned to non-overlaping vertical *strips*, whaetver 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 almost equally full, 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 strip from the set $\mathcal{P}$ (PS_0) and one strip form the set $\mathcal{Q}$ (QS_0). 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. 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 24 of Algorithm 3). Next we examine the K closest pairs in these strips by using the $ClassicPlaneSweep$ ($CCPS$) algorithm at lines 8 and 26, during the first iteration of while-loop at lines 7 and 25, 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 ($\mathcal{Q}$) or 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 7-19 and 25-37 in the Algorithm 3.

In the third step, we will load in main memory 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 by lines 20-22 and 38-40 in the Algorithm 3.

We must also highlight that in Algorithm 3, TS

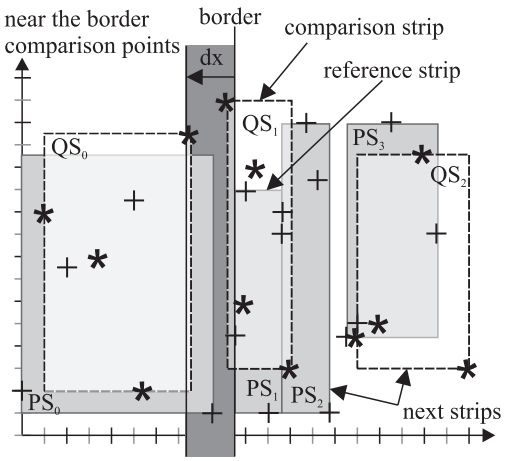

FIGURE 4. Applying the $FRCPS$ algorithm on two data sets partitioned in strips equally full (4 points/strip).

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 that δ from the (right) *border*.

*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.

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.

In the first step, we join the strips of the *current* pair (PS_1 and QS_1). From the *current* pai, we will set as $reference$ strip the strip which has the rightmost $first$ point (QS_1) and the other strip will be set as *comparison* strip (PS_1). This step is implemented by lines 17-22 and 29-34 of Algorithm 4.

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

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

In the second step, while the left limit is outside of the *comparison* strip, we will load the previous strip (PS_0) of the *currentcomparison* strip and we make the join between the strips QS_1 and PS_0. This loop will continue until the left limit will be reached inside the *comparison* strip. This step is implemented by lines 23-27 and 35-39 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 40-51 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 $\mathcal{Q}$. $oleftp$ and $oleftq$ are local variables that save the old values of $gleftp$ and $gleftq$ (previous strips). nPS and nQS are the next strips of (the current strips) PS and QS.

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 P and Q having 16 and 12 points, respectively. We also consider that the maximum number of points per strip is 4. The data sets and the separation into strips are shown in Figure 3 and in Tables 2 and 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 -1.

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

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 = {first, start, end, P[0..n − 1]}, QS = {first, start, end, P[0..m − 1]}, TS
2: border is a local variable to hold the right border of the calculated strip so far
3: Read from LRU Buffer strips PS and QS
4: while PS.first < N and QS.first < M do                ▷ if first point of the strip PS is on the left of left point of the strip QS
5:     if PS.P[0].x < QS.P[0].x then
6:         TS ← QS
7:         while TRUE do
8:             CCPS(PS, TS)
9:             TS.first += (TS.end + 1)                  ▷ prepare to get the next strip of the Q set
10:            if TS.first < M then                      ▷ if the points of strip Q are not finished
11:                border = TS.P[TS.end]                 ▷ set the border at the last x-coordinate
12:                if MaxKHeap is full then
13:                    check_near_border(border, PS)
14:                if PS.start ≤ PS.end then             ▷ if there are points of strip P near the border
15:                    Read from LRU Buffer the next strip of set Q in TS
16:                else
17:                    break                             ▷ all points are too far from the border
18:            else
19:                break                                 ▷ end of the set Q
20:        PS.first += (PS.end + 1)                      ▷ prepare to get the next strip of the P set
21:        if PS.first < N then                          ▷ if the points of strip P are not finished
22:            Read from LRU Buffer the next strip of set P in PS
23:    else                                              ▷ if PS.P[0].x ≥ QS.P[0].x
24:        TS ← PS
25:        while TRUE do
26:            CCPS(TS, QS)
27:            TS.first += (TS.end + 1)                  ▷ prepare to get the next strip of the P set
28:            if TS.first < N then                      ▷ if the points of strip P are not finished
29:                border = TS.P[TS.end]                 ▷ set the border at the last x-coordinate
30:                if MaxKHeap is full then
31:                    check_near_border(border, QS)
32:                if QS.start ≤ QS.end then             ▷ if there are points of strip Q near the border
33:                    Read from LRU Buffer the next strip of set P in TS
34:                else
35:                    break                             ▷ all points are too far from the border
36:            else
37:                break                                 ▷ end of the set P
38:        QS.first += (QS.end + 1)                      ▷ prepare to get the next strip of the Q set
39:        if QS.first < M then                          ▷ if the points of strip Q are not finished
40:            Read from LRU Buffer the next strip of set Q in QS
```

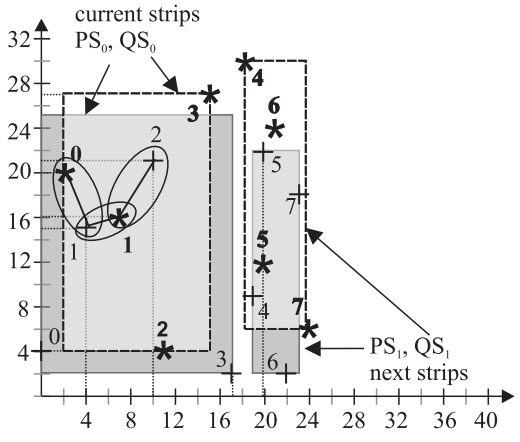

FIGURE 5. Join of strips PS_0 and QS_0 using the $FRCPS$ algorithm.

(regardless of the strip in which they are located), and values for left limits $leftp = 1$, $leftq = 2$.

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 $leftp = 1$ value from the previous iteration. Exiting from $RCPS$ function, no new pair is inserted into $maxKHeap$, but the left limits are updated to $leftp = 3$, $leftq = 2$. 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

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 $\mathcal{Q}$

Allocate memory for strips $PS = \{first, start, end, P[0..n-1]\}, QS = \{first, start, end, P[0..m-1]\}, TS, nPS, nQS$
$gleftp = oleftp = gleftq = oleftp = -1$ ▷ initialize the left limits
Read from LRU Buffer the first strip of set $\mathcal{P}$ in PS and from set $\mathcal{Q}$ in QS
if $PS.P[0].x < QS.P[0].x$ **then** ▷ if the first point of the strip PS is on the left of left point of the strip QS
  $tmp = PS.end + 1$
  while $tmp < N$ **and** $PS.P[PS.end].x < QS.P[0].x$ **do**
    Read from LRU Buffer the next strip of set $\mathcal{P}$ in PS
    tmp += $(PS.end + 1)$ ▷ $tmp = tmp + PS.end + 1$
else
  $tmp = QS.end + 1$
  while $tmp < M$ **and** $QS.P[QS.end].x < PS.P[0].x$ **do**
    Read from LRU Buffer the next strip of set $\mathcal{Q}$ in QS
    tmp += $(QS.end + 1)$ ▷ $tmp = tmp + QS.end + 1$
Read from LRU Buffer the next strip of set $\mathcal{P}$ in nPS and of set $\mathcal{Q}$ in nQS
while $gleftp < N - 1$ **and** $gleftq < M - 1$ **do**
  if $PS.P[0].x < QS.P[0].x$ **then**
    if $gleftp > PS.first$ **then**
      $PS.start = gleftp - PS.first + 1$ ▷ start from the next point of the left limit
      if $PS.start \leq PS.end$ **then**
        RCPS(PS, QS)
    else
      RCPS(PS, QS)
      $oleftp = gleftp \quad TS.first = PS.first \quad TS.P[0].x = PS.P[0].x$
      while $(TS.first > 0$ **and** $((MaxKHeap\ is\ not\ full)$ **or** $(QS.P[0].x - TS.P[0].x < key\ dist\ of\ MaxKHeap\ root))$ **do**
        Read from LRU Buffer the previous strip of set $\mathcal{P}$ in TS
        RCPS(TS, QS)
      $gleftp = olfetp$
  else ▷ if $PS.P[0].x \geq QS.P[0].x$
    if $gleftq > QS.first$ **then**
      $QS.start = gleftq - QS.first + 1$ ▷ start from the next point of the left limit
      if $QS.start \leq QS.end$ **then**
        RCPS(PS, QS)
    else
      RCPS(PS, QS)
      $oleftq = gleftq \quad TS.first = QS.first \quad TS.P[0].x = QS.P[0].x$
      while $(TS.first > 0$ **and** $((MaxKHeap\ is\ not\ full)$ **or** $(PS.P[0].x - TS.P[0].x < key\ dist\ of\ MaxKHeap\ root))$ **do**
        Read from LRU Buffer the previous strip of set $\mathcal{Q}$ in TS
        RCPS(PS, TS)
      $gleftq = olfetq$
  if $nPS.first = N$ **then** ▷ if the points of strip PS are finished
    if $nQS.first = M$ **then** ▷ if the points of strip QS are finished
      break ▷ end of sets, terminate the process
    $QS \leftarrow nQS$
    Read from LRU Buffer the next strip of set $\mathcal{Q}$ in nQS
  else ▷ the points of strip P are not finished
    if $nQS.first \neq M$ **and** $nQS[0].x < nPS[0].x$ **then**
      $QS \leftarrow nQS$
      Read from LRU Buffer the next strip of set $\mathcal{Q}$ in nQS
    else
      $PS \leftarrow nPS$
      Read from LRU Buffer the next strip of set $\mathcal{P}$ in nPS

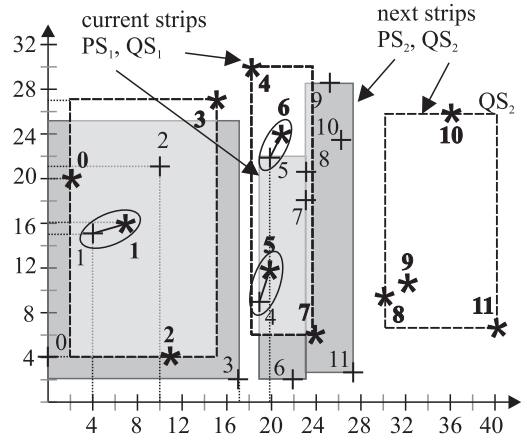

FIGURE 6. Join of strips PS_1 and QS_1 using the $FRCPS$ algorithm.

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 $leftp = 4$, $leftq = 4$. 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 $PS_2\{first = 12, start = 0, end = 3, P[\mathbf{12,13,14,15}]\}$, $QS_2\{first = 8, start = 0, end = 3, P[\mathbf{8,9,10,11}]\}$ are the *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 $leftq = 6$. 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 $leftq = 7$. 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 $leftp = 15$ and $leftq = 9$. 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 strips 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, QS_5 and QS_6).

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 two disk pages from secondary memory with points from the two datasets, we have to execute a *synchronization* process (through *sync_queues* function). This process determines the points in the two disk pages that form the respective two strips such 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 pages PS and QS. As it is shown in Figure 7 the PS_0 page has the array of points with indexes $PS_0.P = [0, 1, 2, 3]$ which are depicted with the symbol '+'. The QS_0 page has the array of points with indexes $QS_0.P = [0, 1, 2, 3]$ which are depicted with the symbol '*'. The $last$ point $QS_0.P[3]$ is on the left of the $last$ point $PS_0.P[3]$l ($QS_0.P[QS_0.end].x < PS_0.P[PS_0.end]$). 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).

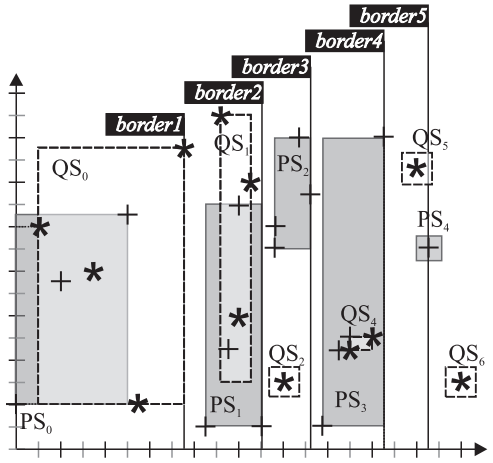

FIGURE 7. Applying the $SCCPS$ algorithm on two data sets partitioned in strips of variable *width*.

In the first step, the process is starts loading the first two pages PS_0 and QS_0. 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 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, we must check if the points of the strip have dx distance from the *border* smaller than the key $dist$ of $maxKHeap$ root, else we must check all points of the strip, with the points of the next strip of the other set, that is called *comparison* set. Now, 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, check the points of the PS_0 strip and if there are some

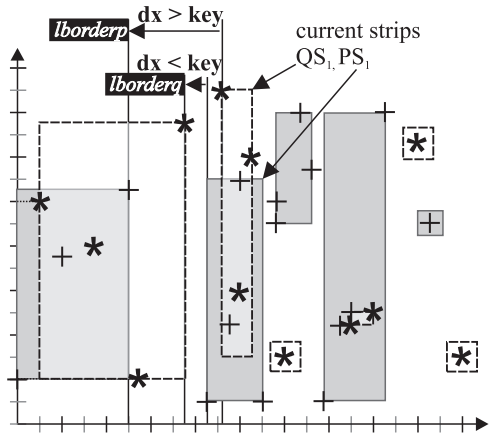

FIGURE 8. Applying the *SRCPS* algorithm on two data sets partitioned in strips of variable *width*.

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 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 7-45 of Algorithm 5.

The third step is to prepare the next pair of strips (PS_1 and QS_1) by loading two pages (one from each set) from secondary memory, *synchronizing* them and continuing from the first step as long as we have points for both strips. This step is implemented by lines 46-56 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 more 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 local variable that holds the right border of the current strips.

*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 the first strip which has *overlap* with the first strip of the other set or which is the last strip of the leftmost set that has no *overlap* with the first strip of the other set; lines 5-16 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 18-19 and 20-23 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 (PS) 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 (QS) 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 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) less than the *key dist of maxKHeap root* (line 24 of Algorithm 6). If all conditions are true then we call the subroutine *srcps_on_border* (Algorithm 7). 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 keft border smaller than the *key dist of maxKHeap root* (line 13 of Algorithm 7). 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 28-31 of Algorithm 6.

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 34-45 of Algorithm 6, where *srcps_get_next_strip* is called. 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 $\mathcal{Q}$.

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 $\mathcal{Q}$. The data sets and the separation into strips, having variable *width*, are shown in the Figure 8.

The algorithm *SRCPS* firstly reads the pages: $PS_0\{first = 0, start = 0, end = 3, P[0,1,2,3]\}$ and $QS_0\{first = 0, start = 0, end = 3, P[0,1,2,3]\}$. 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 -1. 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 $leftp = 1$, $leftq = 0$. 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 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 = {first, start, end, P[0..n − 1]}, QS = {first, start, end, P[0..m − 1]}, TS
2: Read from LRU Buffer strips PS and QS
3: border = sync_queues(PS, QS)
4: while PS.first < N and QS.first < M do
5:     if PS.end ≥ PS.start and QS.end ≥ QS.start then
6:         CCPS(PS, QS)
7:     if PS.end ≥ PS.start then
8:         cur_border = border
9:         if MaxKHeap is full then
10:            check_near_border(cur_border, PS)
11:        if PS.start ≤ PS.end then                    ▷ if there are points of strip PS near the border
12:            TS ← QS
13:            TS.start = QS.end + 1   TS.end = QS.P.m − 1   ▷ update the strip TS to check the rest points of QS
14:            while TRUE do
15:                CCPS(PS, TS)
16:                TS.first += (TS.end + 1)              ▷ prepare to get the next strip of the Q set
17:                if TS.first < M then                 ▷ if the points of strip QS are not finished
18:                    cur_border.x = TS.P[TS.end].x    ▷ set the border at the last x-coordinate
19:                    if MaxKHeap is full then
20:                        check_near_border(cur_border, PS)
21:                    if PS.start ≤ PS.end then        ▷ if there are points of strip PS near the border
22:                        Read from LRU Buffer the next strip of set Q in TS
23:                    else
24:                        break                        ▷ all points are too far from the border
25:                else
26:                    break                            ▷ end of the set P
27:    if QS.end ≥ QS.start then
28:        if MaxKHeap is full then
29:            check_near_border(border, QS)            ▷ cur_border instead of border
30:        if QS.start ≤ QS.end then                    ▷ if there are points of strip QS near the border
31:            TS ← PS
32:            TS.start = PS.end + 1   TS.end = PS.P.n − 1   ▷ update the strip TS to check the rest points of PS
33:            while TRUE do
34:                CCPS(TS, QS)
35:                TS.first += (TS.end + 1)              ▷ prepare to get the next strip of the P set
36:                if TS.first < N then                 ▷ if the points of strip PS are not finished
37:                    border.x = TS.P[TS.end]          ▷ set the border at the last x-coordinate
38:                    if MaxKHeap is full then
39:                        check_near_border(border, QS)
40:                    if QS.start ≤ QS.end then        ▷ if there are points of strip QS near the border
41:                        Read from LRU Buffer the next strip of set P in TS
42:                    else
43:                        break                        ▷ all points are too far from the border
44:                else
45:                    break                            ▷ end of the set Q
46:    PS.first += (PS.end + 1)                         ▷ prepare to get the next strip of the P set
47:    if PS.first ≥ N then                             ▷ if the points of strip PS are finished
48:        break
49:    QS.first += (QS.end + 1)                         ▷ prepare to get the next strip of the Q set
50:    if QS.first ≥ M then                             ▷ if the points of strip QS are finished
51:        break
52:    if PS.end ≥ PS.start then
53:        Read from LRU Buffer the next strip of set P in PS
54:    if QS.end ≥ QS.start then
55:        Read from LRU Buffer the next strip of set Q in QS
56:    border = sync_queues(PS, QS)
```

algorithm must prepare the next iteration. Therefore, the strip PS_0 will remain in main memory by setting the values of the indexes *start* and *end* to the value 3, $PS_0\{first = 0, start = 3, end = 3, P[0, 1, 2, \mathbf{3}]\}$ and the next QS page will be read from disk $QS_1\{first = 4, start = 0, end = 3, P[4, 5, 6, 7]\}$.

For the second iteration and after the *synchronization* process, the current strips are $PS_0\{first = 0, start = 3, end = 3, P[\mathbf{0,1,2,3}]\}$ 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_0 and QS_1) will be omitted (line 20 of the Algorithm 6). The *current* strip PS_0 has the point $PS_0.P[3]$ which is at the right border ($PS_0.start = PS_0.end$), 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_0 strip ($PS_0.P[\mathbf{3}].x = 17$) 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 $PS_0.P[\mathbf{3}]$ 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 $leftq$ is set to 2. In

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 $PS = \{first, start, end, P[0..N-1]\}, QS = \{first, start, end, P[0..m-1]\}$
2: $gleftp = oleftp = gleftq = oleftp = -1$ ▷ initialize the left limits
3: Read from LRU Buffer strips PS and QS
4: $lborderp = PS.P[0].x$, $lborderq = QS.P[0].x$
5: **if** $PS.P[0].x < QS.P[0].x$ **then** ▷ if the first point of the strip PS is on the left of left point of the strip QS
6: $tmp = PS.end + 1$
7: **while** $tmp < N$ **and** $PS.P[PS.end].x < QS.P[0].x$ **do**
8: $lborderp = PS.P[PS.end].x$
9: Read from LRU Buffer the next strip of set $\mathcal{P}$ in PS
10: $tmp \mathrel{+}= (PS.end + 1)$ ▷ $tmp = tmp + PS.end + 1$
11: **else** ▷ if first point of the strip QS is on the left of left point of the strip PS
12: $tmp = QS.end + 1$
13: **while** $tmp < M$ **and** $QS.P[QS.end].x < PS.P[0].x$ **do**
14: $lborderq = QS.P[QS.end].x$
15: Read from LRU Buffer the next strip of set $\mathcal{Q}$ in QS
16: $tmp \mathrel{+}= (QS.end + 1)$ ▷ $tmp = tmp + QS.end + 1$
17: **while TRUE do**
18: **if** $PS.end \neq -2$ **and** $QS.end \neq -2$ **then** ▷ if none STRIP is finished
19: **sync_queues**(PS, QS)
20: **if** $PS.start \leq PS.end$ **and** $QS.start \leq QS.end$ **then**
21: $gleftp = oleftp \quad gleftq = oleftq$
22: **RCPS**(PS, QS)
23: **swap**$(gleftp, oleftp)$, **swap**$(gleftq, oleftq)$
24: **if** $PS.start \leq PS.end$ **and** $(Q.start > 0$ **or** $QS.first > 0)$ **and** $((MaxKHeap\ is\ not\ full)$ **or** $(PS.P[PS.start].x - lborderq < key\ dist\ of\ MaxKHeap\ root))$ **then**
25: **srcps_on_border**$(PS, QS, lborderq, \mathcal{Q}, gleftq, MaxKHeap)$ ▷ CurS, ComS, lborder, X, left, MaxKHeap
26: **if** $gleftq > oleftq$ **then**
27: $oleftq = gleftq$
28: **if** $QS.start \leq QS.end$ **and** $(P.start > 0$ **or** $PS.first > 0)$ **and** $((MaxKHeap\ is\ not\ full)$ **or** $(QS.P[QS.start].x - lborderp < key\ dist\ of\ MaxKHeap\ root))$ **then**
29: **srcps_on_border**$(QS, PS, lborderp, \mathcal{P}, gleftp, MaxKHeap)$ ▷ CurS, ComS, lborder, X, left, MaxKHeap
30: **if** $gleftp > oleftp$ **then**
31: $oleftp = gleftp$
32: **if** $oleftp = N - 1$ **or** $oleftq = M - 1$ **then**
33: **break**
34: $tmp =$ **srcps_get_next_strip**$(PS, QS, \mathcal{P})$
35: **if** $tmp > -1$ **then**
36: $lborderp = tmp$ ▷ set the border at the last x-coordinate
37: **else**
38: **if** $tmp < -1$ **then**
39: **break** ▷ terminate the process, both sets are finished
40: $tmp =$ **srcps_get_next_strip**$(QS, PS, \mathcal{Q})$
41: **if** $tmp > -1$ **then**
42: $lborderq = tmp$ ▷ set the border at the last x-coordinate
43: **else**
44: **if** $tmp < -1$ **then**
45: **break** ▷ terminate the process, both sets are finished

Algorithm 7 SRCPS_ON_BORDER(CurS, ComS, lborder, X, left, MaxKHeap)

1: Allocate memory for strip $TS = \{first, start, end, P[0..N-1]\}$
2: **if** $ComS.start > 0$ **then**
3: $TS \leftarrow ComS$
4: $TS.start = 0$, $TS.end = ComS.start - 1$
5: **else**
6: Read from LRU Buffer the previous strip of set $\mathcal{X}$ in TS
7: **if** $left > TS.first$ **then**
8: $TS.start = left - TS.first + 1$ ▷ start from the next point of the left limit
9: **if** $TS.start \leq TS.end$ **then**
10: **RCPS**$(CurS, TS)$
11: **else**
12: **RCPS**$(CurS, TS)$
13: **while** $TS.first > 0$ **and** $((MaxKHeap\ is\ not\ full)$ **or** $CurS.P[CurS.start].x - TS.P[0].x < key\ dist\ of\ MaxKHeap\ root))$ **do**
14: Read from LRU Buffer the previous strip of set $\mathcal{X}$ in TS
15: **RCPS**$(CurS, TS)$

the third step, the algorithm must prepare the current strips for the next iteration. Therefore, the strip $PS_1\{first = 4, start = 0, end = 3, P[4, 5, 6, 7]\}$ is read and $QS_1\{first = 4, start = 0, end = -1, P[4, 5, 6, 7]\}$ is kept in main memory for the next iteration.

For the third iteration and after the *synchronization* process, the current strips are $PS_1\{first = 4, start = 0, end = 3, P[\mathbf{4,5,6,7}]\}$ 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 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 $leftp = 1, leftq = 4$. The *current* strip PS_1 has points (all points) at or on the left of the right border, the starting point of the *current* strip QS_1

Algorithm 8 SRCPS_GET_NEXT_STRIP(CurS, ComS, X)

```
1: if CurS.start ≤ CurS.end then
2:     CurS.start = (CurS.end + 1)
3:     if CurS.start < CurS.P.m then                ▷ if points of the strip CurS are not processed
4:         CurS.end = (CurS.P.n − 1)
5:     else
6:         CurS.first += (CurS.end + 1)             ▷ prepare to get the next strip of the X set
7:         if CurS.first < M then                   ▷ if the points of strip CurS are not finished
8:             Read from LRU Buffer the next strip of set X in CurS
9:         else
10:            if ComS.end = −2 then
11:                return -2                        ▷ terminate the process, both sets are finished
12:            CurS.end = −2                        ▷ mark that the X set is finished
13:            CurS.first -= CurS.start             ▷ revert the old value of the first id
14:    return CurS.P[ComS.end].x                    ▷ set the border at the last x-coordinate
15: else
16:    return -1                                    ▷ the Current STRIP must remain in the main memory
```

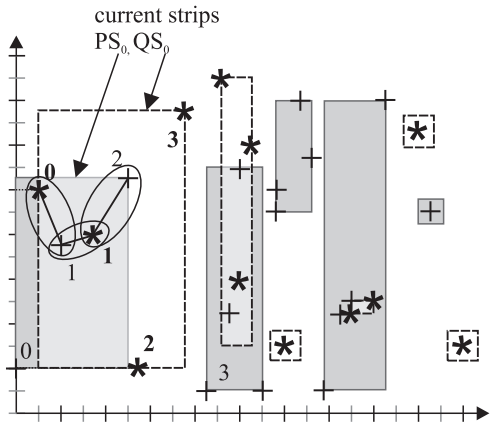

FIGURE 9. Join of strips PS_0 and QS_0 using the $SRCPS$ algorithm.

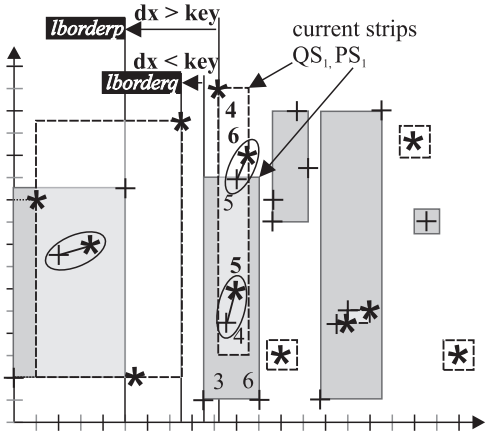

FIGURE 10. Join of strips PS_1 and QS_1 using the $SRCPS$ algorithm.

is not the first point of the set $\mathcal{Q}$ and the difference $PS_1.P[4].x - lborderq = 19 - 15 = 4 < 4.123$. Therefore, the second step will continue by checking the strips PS_1 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_1 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_0 (previous strip of the *current* strip PS_1). The $maxKHeap$ is not updated with new pairs, but the left limits of the sets are updated to the new values $leftp = 2$ and $leftq = 4$. In the third step, the algorithm must prepare the current strips for the next iteration. Therefore, the strip $PS_2\{first = 8, start = 0, end = 3, P[8, 9, 10, 11]\}$ is read and $QS_1\{first = 4, start = 0, end = 2, P[4, 5, 6, 7]\}$ remains in main memory for next iteration.

For the forth iteration and after the *synchronization* process, the current strips are $PS_2\{first = 8, start = 0, end = 0, P[\mathbf{8}, 9, 10, 11]\}$ and $QS_1\{first = 4, start = 3, end = 3, P[4, 5, 6, 7]\}$. 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 $leftp = 2$, $leftq = 4$. The *current* strip PS_2 has (one) point 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 $PS_2.P[8].x - lborderq = 23 - 21 = 2 < 4.123$. Therefore, the second step will continue by checking the strips PS_1 and QS_1 (previous points of the starting point of the *current* strip QS_1). The *current* strip QS_1 has (one) point at or on the left of the right border, the starting point of the *current* strip PS_1 is not the first point of the set $\mathcal{P}$ and the difference $QS_1.P[7].x - lborderp = 24 - 23 = 2 < 4.123$. 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 value $leftp = 4$. In the third step, the algorithm must prepare the current strips for the next iteration. Therefore, the strip $PS_2\{first = 8, start = 0, end = 3, P[8, 9, 10, 11]\}$ is kept and $QS_2\{first = 8, start = 0, end = 3, P[8, 9, 10, 11]\}$ is read from the disk for next iteration.

For the fifth iteration and after the *synchronization*

process, the current strips are $PS_2\{first = 8, start = 1, end = 3, P[8, 9, 10, 11]\}$ and $QS_2\{first = 8, start = 0, end = -1, P[8, 9, 10, 11]\}$. The value of index $QS_2.end$ is smaller than the index $QS_2.start$ and the first step (join between current strips PS_2 and QS_2) will be omitted (lines 20-23 of the Algorithm 6). The *current* strip PS_2 has three points 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 $PS_2.P[9].x - lborderq = 25 - 24 = 1 < 4.123$. Therefore, the second step will continue by checking the strips PS_2 and QS_1 (previous points of the starting point of the *current* strip QS_2). The second part of the second step will not be executed since the *current* strip QS_2 has no points at or on the left of the right border ($QS2.end < QS2.start$). The $maxKHeap$ is not updated with new pairs, but the left limit of the set $\mathcal{Q}$ updated to the new value $leftq = 6$. In the third step, the algorithm must prepare the current strips for the next iteration. Therefore, the strip $PS_3\{first = 12, start = 0, end = 3, P[12, 13, 14, 15]\}$ is read and $QS_2\{first = 8, start = 0, end = 3, P[8, 9, 10, 11]\}$ is kept in the main memory for the next iteration.

For the sixth iteration and after the *synchronization* process, the current strips are $PS_3\{first = 12, start = 0, end = 3, P[\mathbf{12,13,14,15}]\}$ and $QS_2\{first = 8, start = 0, end = 2, P[\mathbf{8,9,10}, 11]\}$. 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 $leftp = 14$, $leftq = 9$. The *current* strip PS_3 has all its four 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 $\mathcal{Q}$ but the difference $PS_3.P[12].x - lborderq = 29 - 24 = 5 > 2$. Therefore, the first part of the second step will be skipped. The *current* strip QS_2 has three points 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 $\mathcal{P}$ but the difference $QS_2.P[8].x - lborderp = 30 - 27 = 3 > 2$. 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 is finished and will be updated to the following values $PS_3\{first = 12, start = 4, end = -2, P[12, 13, 14, 15]\}$ and $QS_2\{first = 8, start = 0, end = 3, P[8, 9, 10, 11]\}$ 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 ($PS_3.end = -2 < 0$). The *current* strip QS_2 has (the last 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 $\mathcal{P}$ but the difference $QS_2.P[11].x - lborderp = 40 - 37 = 3 > 2$. 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 strips 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 3.1 and 4.1, 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 P or 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 $|\mathcal{P}|$ [59], 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 P and 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 that 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 26, the lines 12 and 30 must be removed because $MaxKHeap$ is not used, and $\varepsilon check\_near\_border(border, reference\_strip)$ should be called at lines 13 and 31.
From $SCCPS$ to get $\varepsilon SCCPS$ we should call $\varepsilon CCPS$ instead of $CCPS$ at lines 6, 15 and 34, the lines 9, 19, 28 and 38 must be removed ($MaxKHeap$ is not used), and $\varepsilon check\_near\_border(border, reference\_strip)$ should be called at lines 10, 20, 29 and 39.
From $FRCPS$ to get $\varepsilon FRCPS$ we should call $\varepsilon RCPS$ instead of $RCPS$ at lines 20, 26, 32 and 38. Line 24 should be replaced by **while**(TS.first > 0 **and** (QS.P[0].x − TS.P[0].x ≤ ε_2)) and line 36 by **while**(TS.first > 0 **and** (PS.P[0].x − TS.P[0].x ≤ ε_2)).
And from $SRCPS$ to get $\varepsilon SRCPS$ we should call $\varepsilon RCPS$ instead of $RCPS$ at line 22. Line 24 should be replaced by **if**(PS.start ≤ PS.end **and** (Q.start > 0 **or** QS.first > 0) **and** PS.P[PS.start].x − *lborderq* ≤ ε_2) and line 28 by **if**(QS.start ≤ QS.end **and** (Q.start > 0 **and** QS.first > 0) **and** QS.P[QS.start].x − *lborderp* ≤ ε_2). Moreover, we have to replace $RCPS$ by $\varepsilon RCPS$ in line 22, $maxKHeap$ is not used, $\varepsilon srcps\_on\_border$ is called in lines 25 and 29. Finally, in $\varepsilon srcps\_on\_border(CurS, ComS, lborder, \mathcal{X}, left, \varepsilon_2)$ we should call $\varepsilon RCPS$ instead of $RCPS$ at line 10, 12 and 15, and replace the line 13 by **while**(TS.first > 0 **and** (CurS.P[CurS.start].x − TS.P[0].x < ε_2)).

6. PERFORMANCE EVALUATION

This section provides the results of an extensive experimentation study aiming at measuring and evaluating the efficiency of the new algorithms for KCPQ and εDJQ when none inputs are indexed, which are proposed in Section 5. In particular, Section 6.1 describes the experimental settings and some implementation details. Section 6.2 compares the four proposed algorithms (FCCPS, SCCPS, FRCPS and SRCPS) with respect to the number of pairs (K) in the result of KCPQ. Sections 6.3 and 6.4 show the effect of the disk page size and the size of the strip over the execution of the algorithms, respectively. Section 6.5 examines the effect of the size of LRU buffer in terms of performance of the proposed algorithms. Section 6.6 presents a brief outline of an extensive set of experiments for the execution of εDJQ. Finally, in Section 6.7 a summary from the experimental results is reported.

6.1. Experimental Setup and Implementation Details

In order to evaluate the behavior of the proposed algorithms, we have used four large real spatial data sets of North America, representing cultural landmarks (NAcl consisting of 9203 points) and populated places (NApp consisting of 24493 points), roads (NArd) consisting of 569120 line-segments, and railroads (NArr) consisting of 191637 line-segments. To create large sets of points, we have transformed the MBRs of line-segments from NArd and NArr into points by taking the center of each MBR (i.e. |NArd| = 569120 points, |NArr| = 191637 points). Moreover, in order to get double amount of points from NArr and NArd we choose the two points (*min*, *max*) of the MBR of each line-segment (i.e. |NArdD| = 1138240, |NArrD| = 383274). The data of these 6 files were normalized in the range $[0,1]^2$ (NAclN, NAppN, NArrN, NArrND, NArdN and NArdND). We have also created 6 combinations of input sets ($NAppN \bowtie_K NArrN$, $NAppN \bowtie_K NArdN$, $NArrN \bowtie_K NArdN$, $NArrN \bowtie_K NArdND$, $NArrND \bowtie_K NArdN$ and $NArrND \bowtie_K NArdND$) for the 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. $125KC1N \bowtie_K 125KC2N$, $250KC1N \bowtie_K 250KC2N$, $500KC1N \bowtie_K 500KC2N$, and $1000KC1N \bowtie_K 1000KC2N$).

All experiments were performed on a PC with Intel Core 2 Duo, 2.2 GHz CPU with 4 GB of RAM and several GBs of secondary storage, with Ubuntu Linux v. 12.04 LTS (Linux OS), using the GNU C/C++ compiler (gcc). In this OS it is set the value $CLOCKS\_PER\_SEC = 10^6$, but the time interval for two sequential timeslots is 10^4 Tics. So in the time of two seconds we have $2 * 10^6/10^4 = 200$ Tics. The unit of the time measurement is 10 ms and all experiments have duration at least 2000 ms. Also every experiment

was executed for 5 times and as execution time we got the average between 3 values which was remained after subtraction the maximum and minimum value of 5 initial values.

In our previous paper [57], it is proven that the semi-circle variant of both algorithms *Classic Plane-Sweep* and *Reverse Run Plane-Sweep* has the better efficiency in execution time for KCPQ. For this, all experiments were executed for $CCPS$ and $RCPS$. For the KCPQ and for all (4) algorithms we set four questions. How the value of K, disk page size, size of the strips and size of the LRU buffer, affects the efficiency of the algorithms. For each question we have executed experiments for the previous 10 combinations of data sets.

The performance measurements to show the efficiency of our algorithms are:

1. The execution time of processing the DJQ (i.e. response time). The execution time was measured for overall execution time of the DJQ algorithms. This measure is reported in milliseconds (ms) and represents the overall CPU time consumed, as well as the I/O time performed by the algorithms (i.e. execution time = CPU time + I/O time).
2. The number of X-axis distance calculations (dx).
3. The number of disk accesses (disk-pages read).

To measure the effectiveness of our 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. It is just the opposite to the pruning ratio, and a pair selected occurs when a candidate pair from two strips is considered for processing according to its dx distance.

With respect to some implementation details, in our case, a *data file* is a sequence of records (points 2-dimensional). They are stored in binary files virtually divided into pages whose size is set at the file creation. As we know, in order to design algorithms for processing KCPQ (K must be fixed in advance), an extra data structure that holds the K closest pairs is needed. This data structure is organized as a (binary) max-heap [55], called $maxKHeap$, and holds pairs of points sorted according to their distances $dist$ (i.e. the key is $dist$). This data structure stores the K closest pairs of points with the smallest distance processed so far, and the pair of points with the largest distance resides on top of the $maxKHeap$ (the root), and we will discard the unnecessary pairs of points using its distance value. Initially, the $maxKHeap$ is empty and the distance of all elements inside this data structure is infinity (∞). The pairs of points are inserted in the $maxKHeap$ until it gets full. Then, when a new pair is discovered, if its distance $dist$ is smaller than the top of the $maxKHeap$, then the root is removed and this new pair is inserted in the $maxKHeap$, updating this data structure. Finally, for εDJQ, $maxKHeap$ is not needed and the $resultFile$ is used, which is another binary file whose description and processing has been shown in Section 3.3.

K	FCCPS	SCCPS	FRCPS	SRCPS	Total
1	38.78	33.72	25.32	**24.41**	122.23
10	45.41	44.24	**31.80**	35.71	157.16
100	57.08	58.57	**43.55**	51.28	210.48
1000	93.33	96.51	**78.08**	87.68	355.60
10000	227.41	232.22	**194.24**	203.33	857.20

TABLE 4. Execution time in ms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $NArrN \bowtie_K NArdND$.

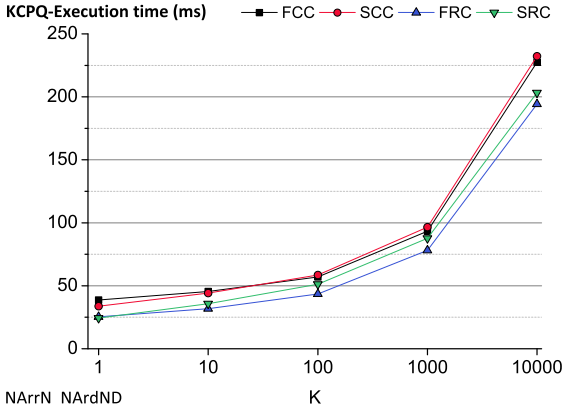

FIGURE 11. Execution time in ms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $NArrN \bowtie_K NArdND$.

6.2. The effect of the number of pairs (K)

In order to examine the effect of the number of pairs (K) of the algorithms in terms of performance, we set the value of K from the set of values (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 exection time

The results for the measure of execution time were similar for all input data sets. Table 4 and Figure 11 show the execution time in ms when KCPQ is executed by the algorithms FCCPS, SCCPS, FRCPS and SRCPS on $NArrN \bowtie_K NArdND$ data sets. As the value of K increases the execution time increases, but the rate of the increment continuously increases and the way of addiction for these quantities is complex. For example using FCCPS algorithm from $K = 1$ to $K = 10$ the time increased 17%, from $K = 10$ to $K = 10^2$ 26%, from $K = 10^2$ to $K = 10^3$ 64% and from $K = 10^3$ to $K = 10^4$ 144%.

In all experiments and for all data sets FCCPS wins 33-16 times vs SCCPS and, FRCPS wins 42-8 times vs SRCPS. Comparing the best result between FCCPS and SCCPS (variants of Classic Plane-Sweep algorithm) and the best result between FRCPS and SRCPS (variants of Reverse Run Plane-Sweep algorithm) for every combination of data sets, we can conclude that FRCPS is faster in all (50-0) cases. Figure 11 represents the values of the execution time of the queries for $K = 1, 10, 10^2, 10^3, 10^4$ of the Table 4.

K	FCCPS	SCCPS	FRCPS	SRCPS
1	31.73%	27.59%	20.72%	**19.97%**
10	28.89%	28.15%	**20.23%**	22.72%
100	27.12%	27.83%	**20.69%**	24.36%
1000	26.25%	27.14%	**21.96%**	24.66%
10000	26.53%	27.09%	**22.66%**	23.72%

TABLE 5. Fraction of execution time of each algorithm on the total time for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $NArrN \bowtie_K NArdND$.

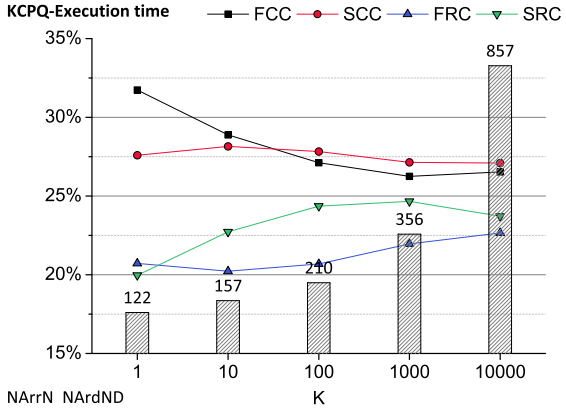

FIGURE 12. Execution time in fractions for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $NArrN \bowtie_K NArdND$.

Table 5 shows the values of the execution time of each query run by each algorithm as a fraction of the total time consumed by all algorithms for the same query. In the Figure 12 is represented the values of the Table 5. Each rectangle has height equal to the total time consumed by all algorithms for each query. It is shown that the FRCPS (blue line with blue triangles as markers) needed 20.23% up to 22.66% of the total time to execute the queries, so it is the fastest algorithm for all values of $K > 1$. Only for the case of $K = 1$, the SRCPS algorithm becomes slightly faster, executing the query in 24.41 ms which is the 19.97% of the total time consumed by all algorithms.

6.2.2. The number of the dx distance calculations

The results with respect to the number of dx distance calculations were similar for all input data sets. Table 6 and Figure 13 show the values of this metric when KCPQ is executed by the algorithms FCCPS, SCCPS, FRCPS and SRCPS on $NArrN \bowtie_K NArdND$ 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 to a ratio ranging between 1.83 and 2.66. For example using SRCPS algorithm from $K = 1$ to $K = 10$ the number of dx distance calculations increased 266%, from $K = 10$ to $K = 10^2$ 183%, from $K = 10^2$ to $K = 10^3$ 207% and from $K = 10^3$ to $K = 10^4$ 213%.

In all experiments and for all data sets SCCPS wins 32-18 times vs FCCPS and, SRCPS wins 44-6 times vs FRCPS. Comparing the best result between FCCPS and SCCPS and the best result between FRCPS and SRCPS for every combination of data sets, we can conclude that RR algorithm needs fewer dx distance calculations in all cases (50-0).

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 6. Number of dx distance calculations is millions ($\times 10^6$) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $NArrN \bowtie_K NArdND$.

K	FCCPS	SCCPS	FRCPS	SRCPS
1	43.9%	28.6%	13.7%	**13.7%**
10	32.7%	26.5%	20.4%	**20.4%**
100	28.1%	25.6%	23.1%	**23.1%**
1000	26.1%	25.3%	24.3%	**24.3%**
10000	25.5%	25.2%	24.7%	**24.7%**

TABLE 7. Fraction of number of dx distance calculations of each algorithm on the total number of dx distance calculations for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $NArrN \bowtie_K NArdND$.

Table 7 shows the values of the number of dx distance calculations of each query executed by each algorithm as a fraction of the total number of dx distance calculations needed by all algorithms for the same query. In the Figure 13 are represented the values of the Table 7. Each rectangle has height equal to the total number of dx distance calculations needed by all algorithms. It is shown that the SRCPS (green line with green triangles as markers) needed 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 is overwritten from the line of the SRCPS. The RR algorithm needs smaller number of dx distance calculations in all cases.

6.2.3. The number of the disk accesses (pages read)

The results for the metric of number of disk accesses (pages read) were 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 algorithms FCCPS, SCCPS, FRCPS and SRCPS on $NArrN \bowtie_K NArdND$ data sets. As the value of K increases the number of disk accesses increases. But the rate of this increment is too small. While the number K increases geometrically with a ratio of 10, the number of pages read also increases, for example, the FRCPS algorithm steps with 0.051%, 0.358%, 1.069%, 3.678%.

In all experiments and for all data sets FCCPS wins 50-0 times vs SCCPS and, FRCPS wins 50-0 times vs SRCPS. Comparing the best result between FCCPS and SCCPS and the best result between FRCPS and SRCPS for every combination of data sets, we can conclude that FRCPS needs fewer disk accesses in all cases (50-

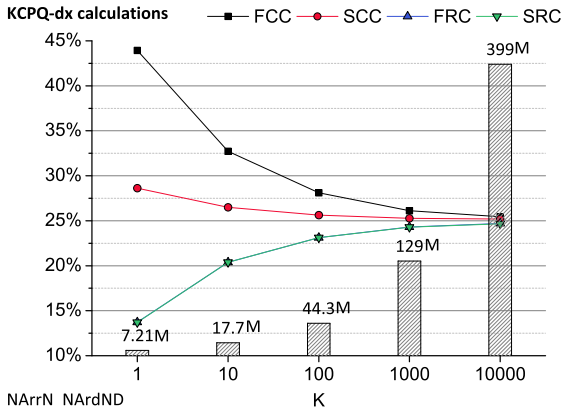

FIGURE 13. Fraction of number of dx distance calculations of each algorithm on total number of dx distance calculations for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $NArrN \bowtie_K NArdND$.

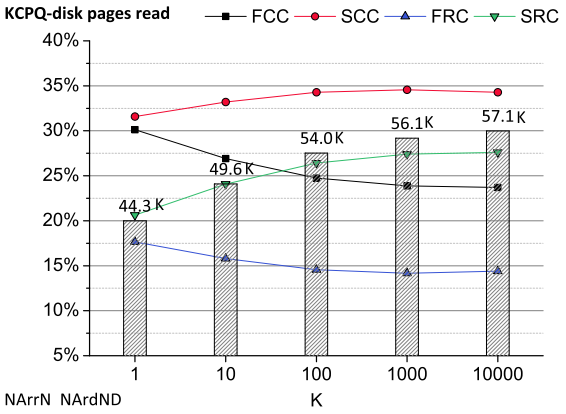

FIGURE 14. Fraction of number of disk accesses of each algorithm on total number of disk accesses for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS on $NArrN \bowtie_K NArdND$.

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 8. Number of disk accesses (in pages read) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $NArrN \bowtie_K NArdND$.

0). Table 9 shows the values of the number of disk accesses of each query executed by each algorithm as a fraction of the total number of disk accesses needed by all algorithms for the same query. Figure 14 represents the values of Table 9. Each rectangle has height equal to the total number of disk accesses needed by all algorithms for each query.

Summarizing the results of experiments on the effect of K in the execution time, the number of dx distance calculations and the number of pages needed to be read, we can say that the exponential growth of $K = 1, 10, 10^2, 10^3, 10^4$ causes: (1) Increase in the execution time but not geometrical. (2) The fastest algorithm proved to be the FRCPS. (3) Increase in the number of dx-distance calculations with a lower ratio (ranging from 1.8 to 3.7). (4) Economical algorithm proves to be the SRCPS. The number of disk accesses required by FCCPS and FRCPS algorithms increases with the growth of K, unlike SCCPS and SRCPS whose increment is more pronounced.

K	FCCPS	SCCPS	FRCPS	SRCPS
1	30.1%	31.6%	**17.7%**	20.6%
10	26.9%	33.2%	**15.8%**	24.1%
100	24.7%	34.3%	**14.6%**	26.4%
1000	23.9%	34.6%	**14.2%**	27.4%
10000	23.7%	34.3%	**14.4%**	27.6%

TABLE 9. Fraction of number of disk accesses of each algorithm on the total number of disk accesses for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $NArrN \bowtie_K NArdND$.

pg	FCCPS	SCCPS	FRCPS	SRCPS	Total
1	843	897	**627**	658	3024
2	834	881	**620**	640	2976
4	829	874	**616**	632	2951
8	828	870	**614**	628	2939
16	827	870	**613**	628	2938

TABLE 10. Execution time in ms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $1000KC1N \bowtie_K 1000KC2N$.

6.3. The effect of the disk page size (pg)

In order to examine the effect of the disk page size (pg) in terms of performance of the algorithms, we set the value of $K = 1000$; the size of disk page (pg) = 1, 2, 4, 8 and 16 KBytes; the size of strip is 16 KBytes; and there is no LRU buffer (its size is 0).

6.3.1. The execution time

The results for the measure of execution time were similar for all input data sets. Table 10 and Figure 15 show the execution time in ms when KCPQ is executed by the algorithms FCCPS, SCCPS, FRCPS and SRCPS on $1000KC1N \bowtie_K 1000KC2N$ 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 2.66%, from $pg = 2KB$ to $pg = 4KB$ 1.30%, from $pg = 4KB$ to $pg = 8KB$ 0.66% and from $pg = 8KB$ to $pg = 16KB$ 0.00%. In the Figure 15 is shown that RR algorithms are faster than the Classic ones.

Table 11 shows the execution time values as a fraction of each algorithm's time on the total execution time consumed by all algorithms and for all page sizes. In all experiments and for all data sets FCCPS wins 48-2 times vs SCCPS and, FRCPS wins 50-0 times vs SRCPS. Comparing the best result between FCCPS and SCCPS and the best result between FRCPS and SRCPS for every combination of data sets, we can conclude that

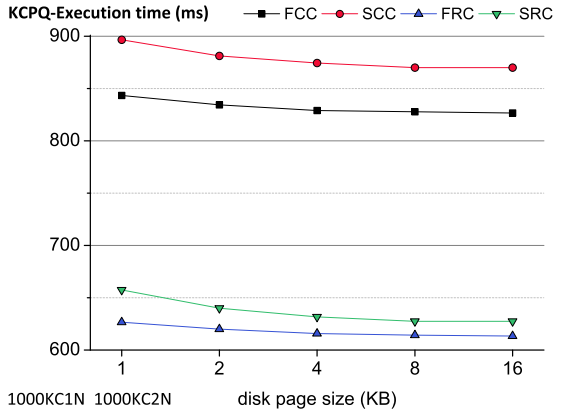

FIGURE 15. Execution time in *ms* for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $1000KC1N \bowtie_K 1000KC2N$.

pg	FCCPS	SCCPS	FRCPS	SRCPS
1	27.89%	29.65%	**20.72%**	21.74%
2	28.04%	29.61%	**20.84%**	21.51%
4	28.09%	29.63%	**20.87%**	21.41%
8	28.16%	29.60%	**20.89%**	21.35%
16	28.18%	29.62%	**20.88%**	21.36%

TABLE 11. Fraction of execution time of each algorithm on the total execution time for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $1000KC1N \bowtie_K 1000KC2N$.

FRCPS is the fastest in all cases. Figure 16 represents the values of the execution time of the queries for disk page having size 1, 2, 4, 8 and 16 KB which are shown in the Table 11. The rectangles represent the total time consumed by all algorithms to execute every query. It is shown that the increment of the disk page size, for sizes larger than 4 KB do not give advantage in query execution for any algorithm. Experiments with page sizes larger than 32 KB show that the execution becomes slightly slower.

The fastest algorithm is the FRCPS and the best disk page size is 8 (for real data sets) and 16 KB (for synthetic data sets), that is larger the physical I/O unit.

6.3.2. The number of dx distance calculations

The results with respect to the number of dx distance calculations are similar for all input data sets. In the Table 12, we can see the values of this metric when KCPQ is executed by the algorithms FCCPS, SCCPS, FRCPS and SRCPS on $1000KC1N \bowtie_K 1000KC2N$ data sets. As the value of disk page size increases, the number of dx distance calculations stays almost constant.

In all experiments and for all data sets SCCPS wins 30-20 times vs FCCPS and, SRCPS wins 44-6 times vs FRCPS. Comparing the best result between FCCPS and SCCPS and the best result between FRCPS and SRCPS for every combination of data sets, we can conclude that RR algorithms need fewer dx calculations in all cases (50-0).

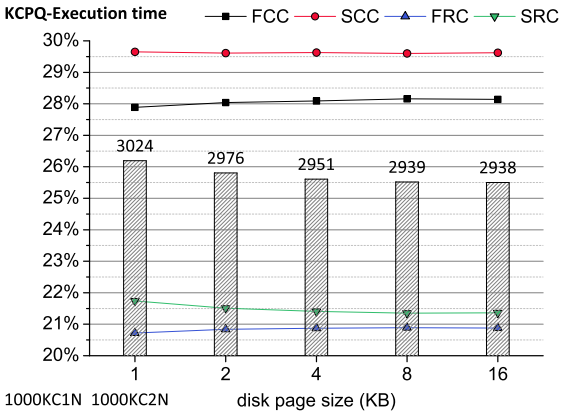

FIGURE 16. Execution time in fraction of the current value of each algorithm on the total execution time of all algorithms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $1000KC1N \bowtie_K 1000KC2N$.

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 12. Number of dx distance calculations in millions ($\times 10^6$) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $1000KC1N \bowtie_K 1000KC2N$.

Table 13 shows the values of the number of dx distance calculations of each query executed by each algorithm as a fraction of the total number of dx distance calculations needed by all algorithms for the same query. In the Figure 17 are represented the values of the Table 13. Each rectangle has height equal to the total number of dx calculations needed by all algorithms. It is shown that the SRCPS (green line with green triangles as markers) needed 20.63% up to 20.64% 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 is overwritten by the line of the SRCPS. The RR algorithms need smaller number of dx distance calculations in all cases.

6.3.3. The number of the disk accesses (pages read)

The results for the metric of number of disk accesses (pages read) were similar for all input data sets and this performance measure proved to be the most important

pg	FCCPS	SCCPS	FRCPS	SRCPS
1	28.63%	30.08%	20.66%	**20.63%**
2	28.66%	30.04%	20.67%	**20.63%**
4	28.66%	30.04%	20.67%	**20.63%**
8	28.65%	30.03%	20.68%	**20.64%**
16	28.65%	30.03%	20.68%	**20.64%**

TABLE 13. Fraction of number of dx distance calculations of each algorithm on the total number of dx distance calculations for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $1000KC1N \bowtie_K 1000KC2N$.

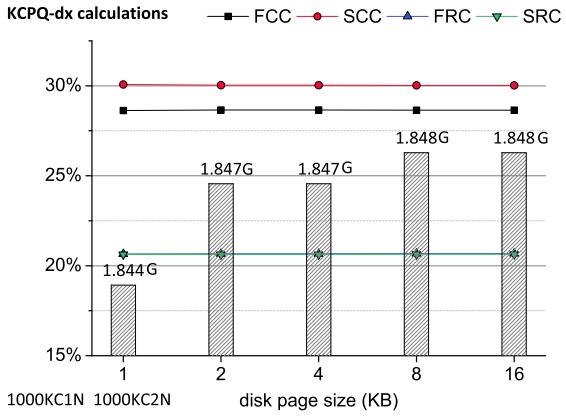

FIGURE 17. Fraction of number of dx distante calculations of each algorithm on the total number of dx distance calculations of all algorithms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $1000KC1N \bowtie_K 1000KC2N$.

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 14. Number of disk accesses (pages read) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $1000KC1N \bowtie_K 1000KC2N$.

factor that shaped the results. Table 14 shows the values of this metric when KCPQ is executed by the algorithms FCCPS, SCCPS, FRCPS and SRCPS on $1000KC1N \bowtie_K 1000KC2N$ 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 decrease smoothly, for example, the FRCPS algorithm steps with 50.74%, 50.00%, 49.78%, 49.99%.

In all experiments and for all data sets FCCPS wins 50-0 times vs SCCPS and, FRCPS wins 50-0 times vs SRCPS. Comparing the best result between FCCPS and SCCPS and, the best result between FRCPS and SRCPS for every combination of data sets, we can conclude that FRCPS needs fewer disk accesses in all cases (50-0). Table 15 shows the values of the number of disk accesses of each query executed by each algorithm as a fraction of the total number of disk accesses needed by all algorithms for the same query. In the Figure 18 are represented the values of the Table 15. Each rectangle has height equal to the total number of disk accesses needed by all algorithms.

Summarizing the results of experiments on the effect of disk page size in the execution time, the number of dx distance calculations and the number of pages needed to be read, we can say that the growth of $pg = 1, 2, 4, 8, 16$ causes: (1) Decrease in the execution time not larger than 15% on real data sets and not larger than 3% on

pg	FCCPS	SCCPS	FRCPS	SRCPS
1	19.77%	30.15%	**16.93%**	33.14%
2	19.82%	30.17%	**16.91%**	33.11%
4	19.82%	30.17%	**16.91%**	33.11%
8	19.84%	30.03%	**16.98%**	33.15%
16	19.84%	30.03%	**16.99%**	33.14%

TABLE 15. Fraction of number of disk accesses of each algorithm on the total number of disk accesses for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $1000KC1N \bowtie_K 1000KC2N$.

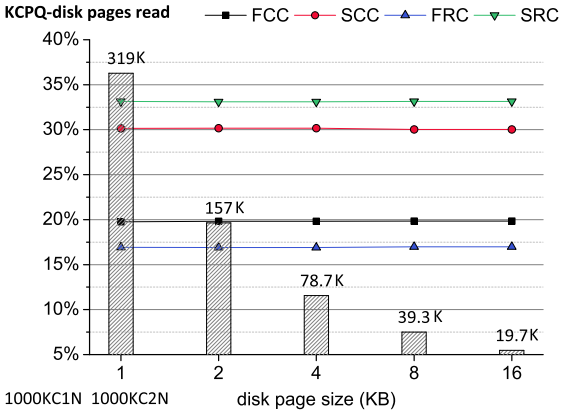

FIGURE 18. Fraction of number of disk accesses (pages read) of each algorithm on the total number of disk accesses of all algorithms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $1000KC1N \bowtie_K 1000KC2N$.

synthetic data sets. (2) The fastest algorithm proves the FRCPS. (3) The number of dx distance calculations remains quite stable. (4) Economical algorithm proves the SRCPS. The number of disk accesses required by FCCPS and FRCPS algorithms decreases hardly, but for the SCCPS and SRCPS this decreasing is not so hard.

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 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 15% 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 the metric of execution time were similar for all input data sets. Table 16 and Figure 19 show the execution time in ms when KCPQ is executed by the algorithms FCCPS, SCCPS, FRCPS and SRCPS on $250KC1N \bowtie_K 250KC2N$ data sets. As the strip size increases, the execution time is reduced, but the rate of

ss	FCCPS	SCCPS	FRCPS	SRCPS	Total
2	150.00	155.38	**118.04**	126.25	549.67
4	142.67	144.52	**111.67**	116.48	515.34
8	135.33	139.11	**108.95**	113.70	497.09
16	121.57	135.56	**107.89**	111.48	476.50
32	115.37	131.25	**108.25**	110.00	464.87

TABLE 16. Execution time in *ms* for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $250KC1N \bowtie_K 250KC2N$.

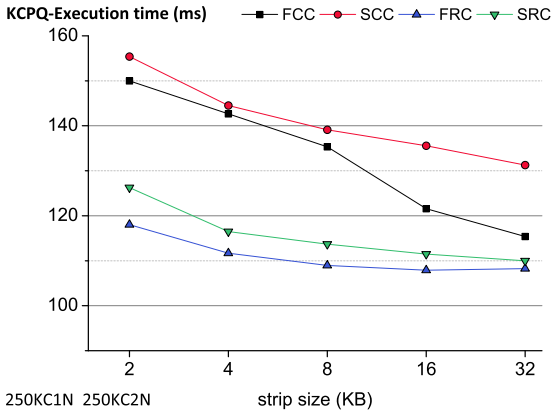

FIGURE 19. Execution time in *ms* for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $250KC1N \bowtie_K 250KC2N$.

decreasing continuously decreases. For example, using SCCPS algorithm from $ss = 2KB$ to $ss = 4KB$ the time decreased 6.99%, from $ss = 4KB$ to $ss = 8KB$ 3.74%, from $ss = 8KB$ to $ss = 16KB$ 3.18% and from $ss = 16KB$ to $ss = 32KB$ 3.01%. In the Figure 19 is shown that the RR algorithms are faster than the Classic ones.

Table 17 shows the execution time values as a fraction of each algorithm's time on the total execution time consumed by all algorithms and for all strip sizes. In all experiments and for all data sets FCCPS wins 46-4 times vs SCCPS and, FRCPS wins 50-0 times vs SRCPS. Comparing the best result between FCCPS and SCCPS and the best result between FRCPS and SRCPS for every combination of data sets, we can conclude that FRCPS is the fastest in all cases. Figure 20 represents the values of the execution time of the queries for strips having sizes 2, 4, 8, 16 and 32 KB which are shown in the Table 17. The rectangles represent the total time consumed by all algorithms to execute every query. It is shown that the increment of the strip size, for sizes larger than 32 KB do not give advantage in terms of query execution time for any algorithm. Experiments with strip sizes larger than 32 KB are shown that the execution becomes slower.

The fastest algorithm is FRCPS and the best strip size is 32 KB for real data sets and 16 KB for synthetic data sets, in all cases larger than the physical I/O unit.

ss	FCCPS	SCCPS	FRCPS	SRCPS
2	27.29%	28.27%	**21.47%**	22.97%
4	27.68%	28.04%	**21.67%**	22.60%
8	27.22%	27.98%	**21.92%**	22.87%
16	25.51%	28.45%	**22.64%**	23.40%
32	24.82%	28.23%	**23.29%**	23.66%

TABLE 17. Fraction of execution time of each algorithm on the total execution time for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $250KC1N \bowtie_K 250KC2N$.

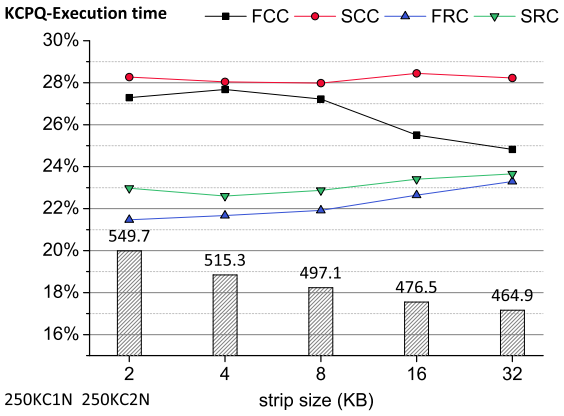

FIGURE 20. Execution time in fraction of the current value of each algorithm on the total execution time of all algorithms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $250KC1N \bowtie_K 250KC2N$.

6.4.2. The number of dx distance calculations

The results with respect to the number of dx distance calculations were similar for all input data sets. Table 18 and Figure 21 show the values of this metric when KCPQ is executed by the algorithms FCCPS, SCCPS, FRCPS and SRCPS on $250KC1N \bowtie_K 250KC2N$ data sets. As the value of strip size increases the number of dx distance calculations remains almost constant.

In all experiments and for all data sets SCCPS wins 30-20 times vs FCCPS and, SRCPS wins 37-13 times vs FRCPS. Comparing the best result between FCCPS and SCCPS and the best result between FRCPS and SRCPS for every combination of data sets, we can conclude that RR algorithm needs fewer dx calculations in all cases (50-0).

Table 19 shows the values of the number of dx distance calculations of each query executed by each algorithm as a fraction of the total number of dx distance calculations needed by all algorithms for the same query. In the Figure 21 are represented the

ss	FCCPS	SCCPS	FRCPS	SRCPS	Total
2	84.4	84.6	**63.6**	63.7	296.3
4	83.3	84.0	**63.5**	63.6	294.4
8	81.2	82.6	**63.2**	63.4	290.4
16	72.4	81.0	**63.1**	63.1	279.6
32	67.7	77.7	63.0	**62.1**	270.6

TABLE 18. Number of dx distance calculations in millions ($\times 10^6$) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $250KC1N \bowtie_K 250KC2N$.

ss	FCCPS	SCCPS	FRCPS	SRCPS
2	28.49%	28.54%	**21.47%**	21.50%
4	28.31%	28.52%	**21.56%**	21.61%
8	27.96%	28.44%	**21.77%**	21.83%
16	25.89%	28.95%	**22.58%**	22.57%
32	25.04%	28.71%	23.30%	**22.96%**

TABLE 19. Fraction of number of dx distance calculations of each algorithm on the total number of dx distance calculations for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $250KC1N \bowtie_K 250KC2N$.

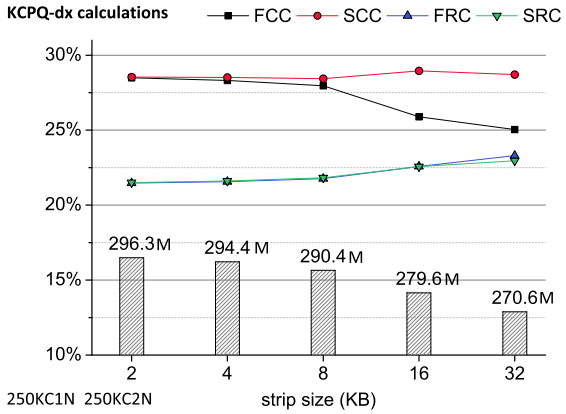

FIGURE 21. Fraction of number of dx distante calculations of each algorithm on the total number of dx distance calculations of all algorithms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $250KC1N \bowtie_K 250KC2N$.

values of the Table 19. Each rectangle has height equal to the total number of dx calculations needed by all algorithms. It is shown that the FRCPS (blue line with blue triangles as markers) needed 21.47% up to 23.30% of the total number of dx distance calculations needed to execute the queries. The SRCPS algorithm has almost equal number of dx distance calculations so its line is overwritten from the line of the FRCPS. The RR algorithm needs smaller number of dx distance calculations in all cases.

6.4.3. The number of the disk accesses (pages read)
The results for the metric of number of read accesses (pages read) were similar for all input data sets and this performance measure proved to be the most important factor that shaped the results. Table 20 shows the values of this metric when KCPQ is executed by the algorithms FCCPS, SCCPS, FRCPS and SRCPS on $250KC1N \bowtie_K 250KC2N$ 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, the SRCPS algorithm steps with 60.17%, 56.13%, 52.99%, 51.58%.

In all experiments and for all data sets FCCPS wins 50-0 times vs SCCPS, and FRCPS wins 50-0 times vs SRCPS. Comparing the best result between FCCPS and SCCPS and the best result between FRCPS and SRCPS for every combination of data sets, we can conclude that FRCPS needs fewer disk accesses in all cases (50-0). Table 21 shows the values of the number of disk accesses of each query executed by each algorithm as a fraction of the total number of disk accesses needed by all algorithms for the same query. In the Figure 22 are represented the values of the Table 21. Each rectangle has height equal to the total number of disk accesses needed by all algorithms for each query.

ss	FCCPS	SCCPS	FRCPS	SRCPS	Total
2	11992	17484	**10223**	18854	58553
4	4620	6878	**4015**	7510	23023
8	1973	2971	**1723**	3295	9962
16	908	1377	**797**	1549	4631
32	436	659	**381**	750	2226

TABLE 20. Number of disk accesses (pages read) for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $250KC1N \bowtie_K 250KC2N$.

ss	FCCPS	SCCPS	FRCPS	SRCPS
2	20.48%	29.86%	**17.46%**	32.20%
4	20.07%	29.87%	**17.44%**	32.62%
8	19.81%	29.82%	**17.30%**	33.08%
16	18.61%	29.73%	**17.21%**	33.45%
32	19.59%	29.60%	**17.12%**	33.69%

TABLE 21. Fraction of number of disk accesses of each algorithm on the total number of disk accesses for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $250KC1N \bowtie_K 250KC2N$.

Summarizing the results of experiments on the effect of strip size in the execution time, the number of dx distance calculations and the number of pages needed to be read, we can say that the exponential growth of $ss = 2^1, 2^2, 2^3, 2^4, 2^5$ causes: (1) Decrease in the execution time not larger than 15% for all, real and synthetic data sets. (2) The fastest algorithm proves the FRCPS. (3) The number of dx distance calculations remains quite stable. (4) Economical algorithm proves the SRCPS. The number of disk accesses needed by all algorithms decreases notably, but the best behaviour for this performance measure is for FRCPS.

6.5. The effect of the LRU buffer (bs)

In order to examine the effect of the size of the LRU buffer (bs) in terms of performance of the algorithms, we set the value of $K = 1000$; the size of disk page (pg) = 16 KBytes; the size of strip (ss) = 16 KBytes; and the size of LRU buffer (bs) = 0, 128, 256, 512, 1024 KBytes.

Regarding the effect of the LRU-buffer on the performance of the algorithms, it is expected that only the execution time will be affected (possibly reduced), as a result of finding in RAM (and not reading from disk) some of the strips that are needed for processing. Nevertheless, there is a cost for the management of the LRU-buffer. It is not expected that the LRU-buffer will 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.

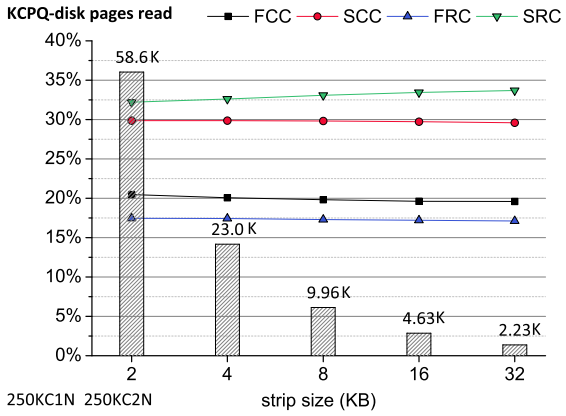

FIGURE 22. Fraction of number of disk accesses (pages read) of each algorithm on the total number of disk accesses of all algorithms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $250KC1N \bowtie_K 250KC2N$.

$bs(KB)$	FCCPS	SCCPS	FRCPS	SRCPS	Total
0	120.00	122.55	**108.25**	117.65	468.45
128	119.80	121.96	**111.05**	115.56	468.37
256	120.20	122.16	**111.11**	115.19	468.66
512	121.37	123.53	**112.22**	117.22	474.34
1024	127.92	131.04	**118.63**	122.75	500.34

TABLE 22. Execution time in ms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $NArrND \bowtie_K NArdND$.

6.5.1. The execution time

The results for the metric of execution time were not similar for all input data sets. As a general note, we can say that for each algorithm as the size of the LRU buffer increases, the execution time was also increased. Table 22 and Figure 23 show the execution time in ms when KCPQ is executed by the algorithms FCCPS, SCCPS, FRCPS and SRCPS on $NArrND \bowtie_K NArdND$ data sets. We show this increment of the time between the execution without buffering and the buffer, having the maximum size (1 MB) values up to 23.25% were obtained (for the combination of datasets $NAppN \bowtie_K NArdN$ and for FRCPS algorithm). For the combinations of the synthetic data sets, the maximum increment had value up to 4.5%. We have selected to show the results about the execution time for the largest combination of real data sets, $NArrND \bowtie_K NArdND$, in the Table 22 and Figure 23.

Figure 23 shows that the FRCPS algorithm is the fastest. In all experiments and for all data sets FCCPS wins 46-4 times vs SCCPS and, FRCPS wins 50-0 times vs SRCPS. Comparing the best result between FCCPS and SCCPS and the best result between FRCPS and SRCPS for every combination of data sets, we can conclude that FRCPS is the fastest in all cases.

6.5.2. The number of strips found in LRU buffer

The results for the metric of the number of strips found in the LRU buffer were similar for all input data sets. For this, we present the values of this number according to the size of the buffer in the Table 23.

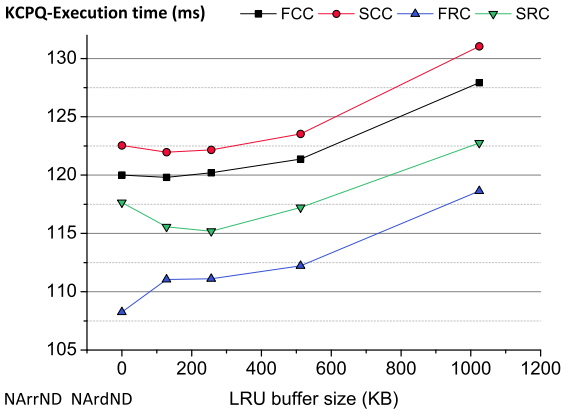

FIGURE 23. Execution time in ms for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $NArrND \bowtie_K NArdND$.

$bs(KB)$	FCCPS	SCCPS	FRCPS	SRCPS	Total
0	0	0	0	0	0
128	1038	1581	**38**	2088	5793
256	1061	1581	**51**	2174	4867
512	1087	1581	**51**	2176	4895
1024	1122	1581	**51**	2177	4931

TABLE 23. Number of the strips found in the LRU buffer for KCPQ using FCCPS, SCCPS, FRCPS and SRCPS for $NArrND \bowtie_K NArdND$.

Table 23 shows that exist one maximum number of strips that must be found in the LRU buffer not depending from the size of the buffer. For FRCPS algorithm this number is very small, up to 51 strips, and was appeared from the size of the buffer equal or larger than 256 KB. We can see that this number is only the 0.87% relative to the total number of strips found from all algorithms in the LRU buffer having the same size. For the combination of $NArrND \bowtie_K NArdND$ data sets the FRCPS algorithm does not found any strip in the LRU buffer except the largest buffer of 1MB in which was found 1 from 873 strips which was read. The FRCPS algorithm has no need to use the LRU buffer, but the results on the execution time depicted this algorithm as the absolute winner for all combinations of data sets and for all sizes of LRU buffer. The fastest execution was proved without any buffering.

6.6. Experimental results for εDJQ

In this section, we are going to study the effect of the increment of distance threshold (ε) for εDJQ. In order to examine the effect of such distance in the performance of the εDJQ algorithms, we set the value of ε_1 equal to zero ($\varepsilon_1 = 0$) and $\varepsilon_2 = \varepsilon$ from the set of values $\{(0, 1.25, 2.5, 5, 10) \times 10^{-5}\}$; the size of disk page (pg) = 4 KBytes; the size of strip (ss) = 16 KBytes; and there is no LRU buffer (its size is 0).

$\varepsilon \times 10^{-5}$	εFCCPS	εSCCPS	εFRCPS	εSRCPS	Total
0	39.32	36.52	26.35	**23.65**	125.89
1.25	74.09	83.72	**63.06**	77.13	298.00
2.5	101.96	112.15	**92.35**	106.81	413.27
5	157.65	168.43	**150.34**	165.58	642.00
10	269.37	281.46	**266.34**	282.85	1100.02

TABLE 24. Execution time in ms for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS for $NArrN \bowtie_\varepsilon NArdND$.

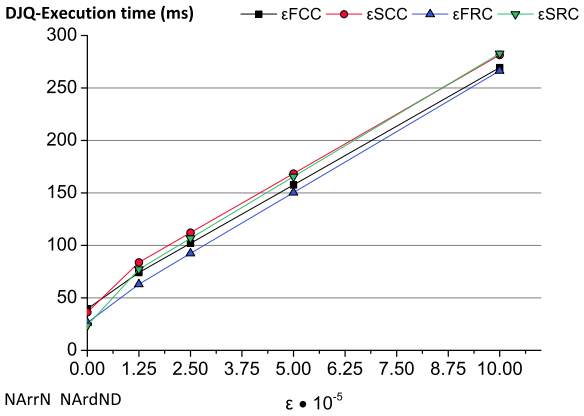

FIGURE 24. Execution time in ms for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS for $NArrN \bowtie_\varepsilon NArdND$.

6.6.1. *The execution time*

The results for the measure of execution time are similar for all input data sets. Table 24 and Figure 24 show the execution time in ms when the εDJQ is executed by the algorithms εFCCPS, εSCCPS, εFRCPS and εSRCPS on $NArrN \bowtie_\varepsilon NArdND$ data sets. As the value of ε increases the execution time grows, and the rate of the increment continuously grows. For example, using εFCCPS algorithm from $\varepsilon = 0$ to $\varepsilon = 1.25 \times 10^{-5}$ the time increased 209%, from $\varepsilon = 1.25 \times 10^{-5}$ to $\varepsilon = 2.5 \times 10^{-5}$ 37%, from $\varepsilon = 2.5 \times 10^{-5}$ to $\varepsilon = 5 \times 10^{-5}$ 53% and from $\varepsilon = 5 \times 10^{-5}$ to $\varepsilon = 10 \times 10^{-5}$ 70%. In the Figure 24 is shown that the εFRCPS algorithm is the fastest.

In all experiments and for all data sets εFCCPS wins 40-10 times vs εSCCPS and, εFRCPS wins 45-5 times vs εSRCPS. 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 εFRCPS is the fastest in the most (34-16) cases. Figure 24 represents the values of the execution time of the queries for $\varepsilon = \{(0, 1.25, 2.5, 5, 10) \times 10^{-5}\}$ of the Table 24.

Table 25 shows the values of the execution time of each query run by each algorithm as a fraction of the total time consumed by all algorithms for the same query. Figure 25 represents the values of Table 25. Each rectangle has height equal to the total time consumed by all algorithms for each query. It is shown that the εFRCPS (blue line with blue triangles as markers) needed 20.93% up to 24.21% of the total time to execute the queries, so it is the fastest algorithm for all values

$\varepsilon \times 10^{-5}$	εFCCPS	εSCCPS	εFRCPS	εSRCPS
0	31.39%	28.89%	20.93%	**18.79%**
1.25	24.83%	28.09%	**21.16%**	25.88%
2.5	24.67%	27.14%	**22.35%**	25.85%
5	24.56%	26.24%	**23.42%**	25.79%
10	24.49%	25.59%	**24.21%**	25.71%

TABLE 25. Fraction of execution time of each algorithm on the total execution time for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS for $NArrN \bowtie_\varepsilon NArdND$.

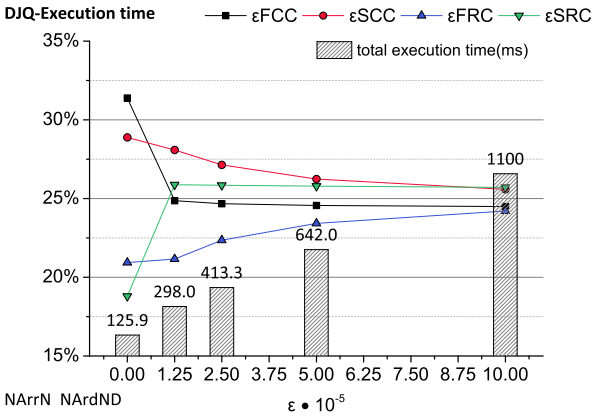

FIGURE 25. Execution time in fraction of the current value of each algorithm on the total execution time of all algorithms for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS for $NArrN \bowtie_\varepsilon NArdND$.

of $\varepsilon > 0$. For the value of $\varepsilon = 0$ the εSRCPS algorithm was faster since the fraction of the time was 18.79%.

6.6.2. *The number of dx distance calculations*

The results with respect to the number of dx distance calculations were similar for all input data sets. Table 26 and Figure 26 show the values of this metric when εDJQ is executed by the algorithms εFCCPS, εSCCPS, εFRCPS and εSRCPS on $NArrN \bowtie_\varepsilon NArdND$ 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 ratio ranging between 1.57 and 2. For example using εSRCPS algorithm from $\varepsilon = 1.25 \times 10^{-5}$ to $\varepsilon = 2.5 \times 10^{-5}$ the number of dx distance calculations increased 99%, from $\varepsilon = 2.5 \times 10^{-5}$ to $\varepsilon = 5 \times 10^{-5}$ 99%, and from $\varepsilon = 5 \times 10^{-5}$ to $\varepsilon = 10 \times 10^{-5}$ 100%.

In all experiments and for all data sets εSCCPS wins 50-0 times vs εFCCPS and, εFRCPS wins 34-16 times vs εSRCPS. Comparing the best result between

$\varepsilon \times 10^{-5}$	εFCCPS	εSCCPS	εFRCPS	εSRCPS	Total
0	2.40	1.20	0.33	**0.33**	4.35
1.25	25.22	24.13	**23.16**	23.16	95.67
2.5	48.04	46.96	**45.99**	45.99	186.98
5	93.71	92.65	**91.68**	91.68	369.72
10	184.99	183.97	183.00	**183.00**	734.95

TABLE 26. Number of dx distance calculations in millions ($\times 10^6$) for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS for $NArrN \bowtie_\varepsilon NArdND$.

$\varepsilon \times 10^{-5}$	εFCCPS	εSCCPS	εFRCPS	εSRCPS
0	55.07%	29.76%	7.61%	**7.56%**
1.25	26.36%	25.22%	**24.21%**	24.21%
2.5	25.69%	25.11%	**24.60%**	24.60%
5	25.35%	25.06%	**24.80%**	24.80%
10	25.17%	25.03%	**24.90%**	24.90%

TABLE 27. Fraction of number of dx distance calculations of each algorithm on the total number of dx distance calculations for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS for $NArrN \bowtie_\varepsilon NArdND$.

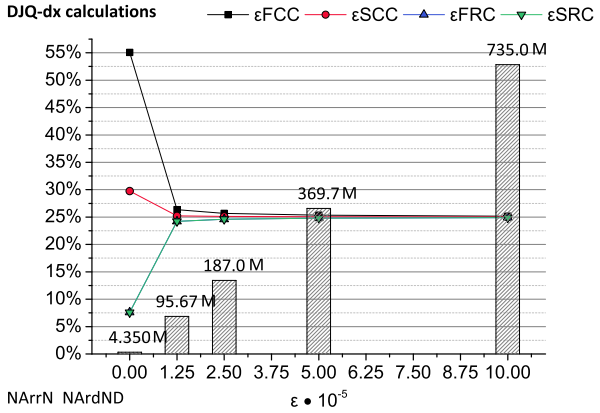

FIGURE 26. Fraction of number of dx distante calculations of each algorithm on the total number of dx distance calculations of all algorithms for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS for $NArrN \bowtie_\varepsilon NArdND$.

εFCCPS and εSCCPS and the best result between εFRCPS and εSRCPS for every combination of data sets, we conclude that RR algorithm needs fewer dx distance calculations in all cases (50-0).

Table 27 shows the values of the number of dx distance calculations of each query executed by each algorithm as a fraction of the total number of dx distance calculations needed by all algorithms for the same query. In the Figure 26 are represented the values of the Table 27. Each rectangle has height equal to the total number of dx calculations needed by all algorithms. It is shown that the εFRCPS (blue line with blue triangles as markers) needed 7.61% for the case of $\varepsilon = 0$ and for the other cases from 24.21% up to 24.90% of the total number of dx distance calculations needed to execute the queries. The εSRCPS algorithm has a little fewer dx distance calculations than εFRCPS only in the case $\varepsilon = 0$ and in all other cases almost equal number of dx distance calculations so its line is overwritten from the line of the εFRCPS. The RR algorithm needs smaller number of dx calculations in all cases.

6.6.3. The number of the disk accesses (pages read)
The results for the metric of number of disk accesses (pages read) were similar for all input data sets and this performance measure proved to be the most important factor that shaped the results. Table 28 shows the values of this metric when εDJQ is executed by the algorithms εFCCPS, εSCCPS, εFRCPS and εSRCPS on $NArrN \bowtie_\varepsilon NArdND$ data sets. As the value of ε increases, the number of disk accesses increases. But the rate of this increment is too small. While the number ε increases geometrically with a ratio of 2, the number of pages read increases with a lower ratio, for example, the εFRCPS algorithm steps with 0.971%, 1.468%, 2.345% and 4.924%.

$\varepsilon \times 10^{-5}$	εFCCPS	εSCCPS	εFRCPS	εSRCPS	Total
0	13340	13215	**7824**	7824	42203
1.25	13380	19223	**7900**	15220	55723
2.5	13420	19439	**8016**	15460	56335
5	13516	10535	**8204**	15720	56975
10	13704	19587	**8608**	16156	58055

TABLE 28. Number of disk accesses (pages read) for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS for $NArrN \bowtie_\varepsilon NArdND$.

$\varepsilon \times 10^{-5}$	εFCCPS	εSCCPS	εFRCPS	εSRCPS
0	31.61%	31.31%	**18.54%**	18.54%
1.25	24.01%	34.50%	**14.18%**	27.31%
2.5	23.82%	34.51%	**14.23%**	27.44%
5	23.72%	34.29%	**14.40%**	27.59%
10	23.61%	33.74%	**14.83%**	27.83%

TABLE 29. Fraction of number of disk accesses of each algorithm on the total number of disk accesses for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS for $NArrN \bowtie_\varepsilon NArdND$.

In all experiments and for all data sets εFCCPS wins 46-4 times vs εSCCPS and εFRCPS wins 40-10 times vs εSRCPS. 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 εFRCPS needs fewer disk accesses in all cases (50-0). Table 29 shows the values of the number of disk accesses of each query executed by each algorithm as a fraction of the total number of disk accesses needed by all algorithms for the same query. Figure 22 represents the values of Table 29. Each rectangle has height equal to the total number of disk accesses needed by all algorithms for each query.

Summarizing the results of experiments on the effect of ε (for εDJQ) in the execution time, the number of dx distance calculations and the number of pages needed to be read, we can say that the exponential growth of $\varepsilon = \{(0, 1.25, 2.5, 5, 10) \times 10^{-5}\}$ causes: (1) Increase in the execution time but not geometrical. (2) The fastest algorithm proves to be the εFRCPS. (3) Increase in the number of dx distance calculations with a lower ratio (ranging from 1.57 to 2). (4) Economical algorithm proves to be the εFRCPS. The number of disk accesses required by εFCCPS and εFRCPS algorithms increases with the growth of ε, unlike εSCCPS and εSRCPS whose increment is more pronounced.

The experiments was continued in the same form as in sections 6.3, 6.4 and 6.5 for KCPQ made, in order to study the effect of the disk page size (pg), the size of strip (ss) and the size of the LRU-buffer (bs). In general the results were similar between KCPQ and εDJQ. Fastest in execution time and reading fewer

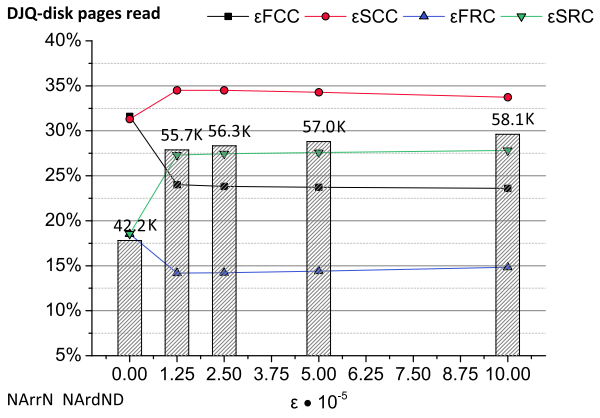

FIGURE 27. Fraction of number of disk accesses (pages read) of each algorithm on the total number of disk accesses of all algorithms for εDJQ using εFCCPS, εSCCPS, εFRCPS and εSRCPS for $NArrN \bowtie_{\varepsilon} NArdND$.

pages from the disk proved the εFRCPS algorithm for εDJQ. We can remark only that, for the case of $\varepsilon = 0$ the RRPS algorithms is much better than the Classic ones. For the cases that $\varepsilon > 0$ the results for FRCPS are better than εFRCPS. This result is explained if we accept that the most important factors that improve the performance of an algorithm for KCPQ are (1) how quickly will be entered in the $maxKHeap$ pairs with very short distances, and (2) how efficiently the algorithm will manage the largest distance of these pairs. In contrast to KCPQ, εDJQ does not need the fastest finding pairs with short distance since the maximum acceptable distance is consistently defined by the user beforehand (ε). There remains only the smart and economical management of the given distance on the final performance of an algorithm and its characterization in terms of profitability.

6.7. Effectiveness study

To study the effectiveness of the proposed algorithms we will use the *selection ratio*, and remember that it is the fraction of pairs considered by the 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 measure is the opposite to the pruning ratio, and therefore the smaller the selection ratio, the higher the power of pruning of the algorithm.

As to study the efficiency, we can allow for different parameters as K, pg, ss and bs. Now, we are going to consider only the increment of K. Tables 30 and 31 reports 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 combination of real data there are $191,637,138,240 = 218{,}128{,}898{,}880$ possible pairs ($2.18 \times 10^1 1$) and for the synthetic data we have $10^1 2$

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 30. 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 $NArrN \bowtie_K NArdND$.

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 31. 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 $1000KC1N \bowtie_K 1000KC2N$.

possible pairs. We can observe that an increasing K makes the selection ratio of the proposed algorithms that continuously increases. 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 RR and Classic plane-sweep algorithms (we can mainly see this fact on real data) in terms of selection ratio.

From the tables, we can 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). It means that FRCPS sacrifices slightly effectiveness for efficiency, in the use of the partitioning technique. And an interesting conclusion from this effectiveness measure is that the best algorithms in pruning are the RRPS ones, and this conclusion is according to the efficiency. Moreover, since the selection ratio depends on the dx distance, it is the most representative measure for pruning and for effectiveness.

And we have to highlight that the average performance in pruning (for all combinations) for real and synthetic data follows the same trend. Such as behaviour is shown in Tables 32 and 33, where SRCPS is the winner in most of the cases, but FRCPS is very close to it.

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 32. 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 33. 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.

6.8. Conclusions from the experiments

In our previous work [57] was proved that the RRPS algorithms are faster than the Classic ones for KCPQ when the data is stored and processed in the main memory. It is due to that Classic PS algorithms always process the data sets from left to right and the runs of the two sets are generally interleaved. On the other hand, RRPS algorithms process pairs of points in opposite X-orders, 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 this, the pruning distance (*key dist of MaxKHeap root*) is expected to be updated more quickly, the query processing cost of RRPS algorithms will be smaller and it will become faster.

From the experiments presented here and when the data are stored on disk, we can 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 the disk to main memory; (3) Volumes of memory required and their management; and (4) How quickly the *maxKHeap* is filled up with pairs having small distances, and how fast the pruning distance (*key dist of MaxKHeap root*) is progressively reduced (it is important for KCPQ, unlike the εDJQ), because the lower its value is, the greater the power of pruning. Every one of these factors affects differently the final result, but FRCPS is the fastest for the increment of K, disk page size (pg), size of strips (ss) and size of LRU buffer (bs), although SRCPS requires less memory volume compared to FRCPS.

With respect to the *number of dx distance calculations*, SRCPS algorithm seems to be better (lower number of calculations) in most of the cases, although FRCPS is quite close (i.e. the difference compared to SRCPS in total calculations is rather small). It is due to that the for RRPS 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.

For the case of the number of disk accesses, FRCPS needs less disk accesses in all experiments (K, pg and ss). It is due to the combination of RRPS 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 algorithms). This smaller number of strips leads to smaller number of strips read from disk, and then smaller number of disk accesses. In addition, the number of disk accesses seems to be the most influential factor governing the algorithm efficiency in execution time, and the difference between SRCPS and FRCPS becomes significant for this performance measure, in which FRCPS is totally dominating, and thus, faster.

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

1. a smaller number of strips that partition the space,
2. a smaller number of strips read from disk,
3. a more consistent application of RRPS processing in the management of the 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 RRPS algorithms are the most effective algorithm for pruning, highlighting SRCPS slightly over FRCPS.

Finally, from this extensive experimental study we can conclude that RRPS algorithms are the most efficient and effective for K-CPQ and εDJQ, and we should highlight the FRCPS variant.

7. CONCLUSIONS AND FUTURE WORK

This paper has presented several efficient and effective algorithms (FCCPS, SCCPS, FRCPS and SRCPS) for K-CPQ 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 propose 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 the four algorithms (FCCPS, SCCPS, FRCPS and SRCPS) for KCPQ and εDJQ are proposed, without the use of an index on each data set saving on disk (neither inputs are indexed). They employ a combination of the *plane-sweep* algorithms and space partitioning techniques to join the data sets. Finally, we present results of an extensive experimental study, where efficiency and effectiveness measures are compared for the proposed algorithms. That performance study conducted on long spatial data sets (real and synthetic), when neither inputs are indexed, we can conclude that RRPS algorithms are the most efficient and effective for K-CPQ and εDJQ, and we should highlight that FRCPS is the best variant, which combines RRPS processing with uniform filling partition technique. For the future work, we plan to further investigate to adapt the new plane-sweep-based algorithms, when neither input is

indexed, to other related DJQs (as Iceberg Distance Join Query [49] and K Nearest Neighbour Join query [50]). Moreover, it would be interesting to study approximate implementations of proposed algorithms by using the distance-based approximate techniques presented in [42].

ACKNOWLEDGEMENTS

Work 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.

REFERENCES

[1] Güting, R. H. (1994) An introduction to spatial database systems. *VLDB J.*, **3**, 357–399.

[2] Shekhar, S. and Chawla, S. (2003) *Spatial databases - a tour*. Prentice Hall.

[3] Samet, H. (2007) *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, San Francisco, CA.

[4] Gaede, V. and Günther, O. (1998) Multidimensional access methods. *ACM Computing Surveys*, **30**, 170–231.

[5] Corral, A., Manolopoulos, Y., Theodoridis, Y., and Vassilakopoulos, M. (2000) Closest pair queries in spatial databases. *SIGMOD Conference*, May, pp. 189–200.

[6] Corral, A., Manolopoulos, Y., Theodoridis, Y., and Vassilakopoulos, M. (2004) Algorithms for processing k-closest-pair queries in spatial databases. *Data Knowl. Eng.*, **49**, 67–104.

[7] Preparata, F. P. and Shamos, M. I. (1985) *Computational Geometry - An Introduction*. Springer.

[8] Hinrichs, K., Nievergelt, J., and Schorn, P. (1988) Plane-sweep solves the closest pair problem elegantly. *Information Processing Letters*, **26**, 255–261.

[9] Jacox, E. H. and Samet, H. (2007) Spatial join techniques. *ACM Trans. Database Syst.*, **32**, 7.

[10] Shin, H., Moon, B., and Lee, S. (2003) Adaptive and incremental processing for distance join queries. *IEEE Trans. Knowl. Data Eng.*, **15**, 1561–1578.

[11] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. (1990) The r*-tree: An efficient and robust access method for points and rectangles. *SIGMOD Conference*, May, pp. 322–331.

[12] Jacox, E. H. and Samet, H. (2003) Iterative spatial join. *ACM Trans. Database Syst.*, **28**, 230–256.

[13] Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T., and Vitter, J. S. (1998) Scalable sweeping-based spatial join. *VLDB Conference*, August, pp. 570–581.

[14] Gurret, C. and Rigaux, P. (2000) The sort/sweep algorithm: A new method for r-tree based spatial joins. *SSDBM Conference*, July, pp. 153–165.

[15] Hjaltason, G. R. and Samet, H. (1998) Incremental distance join algorithms for spatial databases. *SIGMOD Conference*, June, pp. 237–248.

[16] Mamoulis, N. and Papadias, D. (2001) Multiway spatial joins. *ACM Trans. Database Syst.*, **26**, 424–475.

[17] Orenstein, J. A. (1986) Spatial query processing in an object-oriented database system. *SIGMOD Conference*, May, pp. 326–336.

[18] Rotem, D. (1991) Spatial join indices. *ICDE Conference*, April, pp. 500–509.

[19] Brinkhoff, T., Kriegel, H.-P., and Seeger, B. (1993) Efficient processing of spatial joins using r-trees. *SIGMOD Conference*, May, pp. 237–246.

[20] Hoel, E. G. and Samet, H. (1995) Benchmarking spatial join operations with spatial output. *VLDB Conference*, September, pp. 606–618.

[21] Huang, Y.-W., Jing, N., and Rundensteiner, E. A. (1997) Spatial joins using r-trees: Breadth-first traversal with global optimizations. *VLDB Conference*, August, pp. 396–405.

[22] Guttman, A. (1984) R-trees: A dynamic index structure for spatial searching. *SIGMOD Conference*, June, pp. 47–57.

[23] Bae, W. D., Alkobaisi, S., and Leutenegger, S. T. (2010) *IRSJ*: incremental refining spatial joins for interactive queries in gis. *GeoInformatica*, **14**, 507–543.

[24] Lo, M.-L. and Ravishankar, C. V. (1994) Spatial joins using seeded trees. *SIGMOD Conference*, May, pp. 209–220.

[25] Lo, M.-L. and Ravishankar, C. V. (1998) The design and implementation of seeded trees: An efficient method for spatial joins. *IEEE Trans. Knowl. Data Eng.*, **10**, 136–152.

[26] Papadopoulos, A., Rigaux, P., and Scholl, M. (1999) A performance evaluation of spatial join processing strategies. *SSD Conference*, July, pp. 286–307.

[27] Mamoulis, N. and Papadias, D. (2003) Slot index spatial join. *IEEE Trans. Knowl. Data Eng.*, **15**, 211–231.

[28] Patel, J. M. and DeWitt, D. J. (1996) Partition based spatial-merge join. *SIGMOD Conference*, June, pp. 259–270.

[29] Koudas, N. and Sevcik, K. C. (1997) Size separation spatial join. *SIGMOD Conference*, May, pp. 324–335.

[30] Dittrich, J.-P. and Seeger, B. (2000) Data redundancy and duplicate detection in spatial join processing. *ICDE Conference*, February, pp. 535–546.

[31] Lo, M.-L. and Ravishankar, C. V. (1996) Spatial hash-joins. *SIGMOD Conference*, June, pp. 247–258.

[32] Smid, M. (2000) Closest-point problems in computational geometry. In Sack, J.-R. and Urrutia, J. (eds.), *Handbook of Computational Geometry*, chapter 20, pp. 877–935. Elsevier.

[33] Corral, A. and Almendros-Jiménez, J. M. (2007) A performance comparison of distance-based query algorithms using r-trees in spatial databases. *Inf. Sci.*, **177**, 2207–2237.

[34] Kim, Y. J. and Patel, J. M. (2010) Performance comparison of the r*-tree and the quadtree for knn and distance join queries. *IEEE Trans. Knowl. Data Eng.*, **22**, 1014–1027.

[35] Gutiérrez, G. and Sáez, P. (2013) The k closest pairs in spatial databases when only one set is indexed. *GeoInformatica*, **17**, 543–565.

[36] Weber, R., Schek, H.-J., and Blott, S. (1998) A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. *VLDB Conference*, August, pp. 194–205.

[37] Pincheira, M., Gutiérrez, G., and Gajardo, L. (2010) Closest pair query on spatial data sets without index. *SCCC Conference*, November, pp. 178–182.

[38] Koudas, N. and Sevcik, K. C. (2000) High dimensional similarity joins: Algorithms and performance evaluation. *IEEE Trans. Knowl. Data Eng.*, **12**, 3–18.

[39] Chan, E. P. F. (2003) Buffer queries. *IEEE Trans. Knowl. Data Eng.*, **15**, 895–910.

[40] Yang, C. and Lin, K.-I. (2002) An index structure for improving nearest closest pairs and related join queries in spatial databases. *IDEAS Conference*, April, pp. 140–149.

[41] Angiulli, F. and Pizzuti, C. (2005) An approximate algorithm for top-k closest pairs join query in large high dimensional data. *Data Knowl. Eng.*, **53**, 263–281.

[42] Corral, A. and Vassilakopoulos, M. (2005) On approximate algorithms for distance-based queries using r-trees. *The Computer Journal*, **48**, 220–238.

[43] Shan, J., Zhang, D., and Salzberg, B. (2003) On spatial-range closest-pair query. *SSTD Conference*, July, pp. 252–269.

[44] Papadopoulos, A. N., Nanopoulos, A., and Manolopoulos, Y. (2006) Processing distance join queries with constraints. *Comput. J.*, **49**, 281–296.

[45] Qiao, S., Tang, C., Peng, J., Li, H., and Ni, S. (2008) Efficient k-closest-pair range-queries in spatial databases. *WAIM Conference*, July, pp. 99–104.

[46] U, L. H., Mamoulis, N., and Yiu, M. L. (2008) Computation and monitoring of exclusive closest pairs. *IEEE Trans. Knowl. Data Eng.*, **20**, 1641–1654.

[47] Cheema, M. A., Lin, X., Wang, H., Wang, J., and Zhang, W. (2011) A unified approach for computing top-k pairs in multidimensional space. *ICDE Conference*, April, pp. 1031–1042.

[48] Zhang, W., Xu, J., Liang, X., Zhang, Y., and Lin, X. (2012) Top-k similarity join over multi-valued objects. *DASFAA Conference*, April, pp. 509–525.

[49] Shou, Y., Mamoulis, N., Cao, H., Papadias, D., and Cheung, D. W. (2003) Evaluation of iceberg distance joins. *SSTD Conference*, July, pp. 270–288.

[50] Böhm, C. and Krebs, F. (2004) The *k*-nearest neighbour join: Turbo charging the kdd process. *Knowl. Inf. Syst.*, **6**, 728–749.

[51] Zhang, J., Mamoulis, N., Papadias, D., and Tao, Y. (2004) All-nearest-neighbors queries in spatial databases. *SSDBM Conference*, June, pp. 297–306.

[52] Chen, Y. and Patel, J. M. (2007) Efficient evaluation of all-nearest-neighbor queries. *ICDE Conference*, April, pp. 1056–1065.

[53] Zheng, K., Zhou, X., Fung, G. P. C., and Xie, K. (2012) Spatial query processing for fuzzy objects. *VLDB J.*, **21**, 729–751.

[54] Braunmüller, B., Ester, M., Kriegel, H.-P., and Sander, J. (2001) Multiple similarity queries: A basic dbms operation for mining in metric databases. *IEEE Trans. Knowl. Data Eng.*, **13**, 79–95.

[55] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009) *Introduction to Algorithms (3. ed.)*. MIT Press.

[56] Bryan, B., Eberhardt, F., and Faloutsos, C. (2008) Compact similarity joins. *ICDE Conference*, April, pp. 346–355.

[57] Roumelis, G., Vassilakopoulos, M., Corral, A., and Manolopoulos, Y. (2014) A new plane-sweep algorithm for the k-closest-pairs query. *SOFSEM Conference*, January, pp. 478–490.

[58] Graefe, G. (1993) Query evaluation techniques for large databases. *ACM Comput. Surv.*, **25**, 73–170.

[59] Aggarwal, A. and Vitter, J. S. (1988) The input/output complexity of sorting and related problems. *Commun. ACM*, **31**, 1116–1127.

[60] Leutenegger, S. T., Edgington, J. M., and Lopez, M. A. (1997) Str: A simple and efficient algorithm for r-tree packing. *ICDE Conference*, April, pp. 497–506.

[61] Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A. N., and Theodoridis, Y. (2006) *R-Trees: Theory and Applications*. Springer.

[62] Roussopoulos, N., Kelley, S., and Vincent, F. (1995) Nearest neighbor queries. *SIGMOD Conference*, May, pp. 71–79.