Compact data structures for efficient processing of distance-based join queries

Guillermo de Bernardo[1[0000−0002−6020−7092]], Miguel R. Penabad[1[0000−0001−5455−6088]], Antonio Corral[2[0000−0002−0069−4642]], and Nieves R. Brisaboa[1[0000−0001−8025−3048]]

[1] Universidade da Coruña, Centro de investigación CITIC, 15071 A Coruña, Spain.
E-mail: {guillermo.debernardo,miguel.penabad,nieves.brisaboa}@udc.es
[2] Dept. of Informatics, University of Almeria, 04120 Almeria, Spain.
E-mail: acorral@ual.es

Abstract. Compact data structures can represent data with usually a much smaller memory footprint than its plain representation. In addition to maintaining the data in a form that uses less space, they allow us to efficiently access and query the data in its compact form. The k^2-tree is a self-indexed, compact data structure used to represent binary matrices, that can also be used to represent points in a spatial dataset. Efficient processing of the Distance-based Join Queries (DJQs) is of great importance in spatial databases due to its wide area of application. Two of the most representative and known DJQs are the K Closest Pairs Query (KCPQ) and the ε Distance Join Query (εDJQ). These types of join queries are executed over two spatial datasets and can be solved by plane-sweep algorithms, which are efficient but with great requirements of RAM, to be able to fit the whole datasets into main memory. In this work, we present new and efficient algorithms to implement DJQs over the k^2-tree representation of the spatial datasets, experimentally showing that these algorithms are competitive in query times, with much lower memory requirements.

Keywords: k^2-tree · K Closest Pairs · ε Distance Join · Spatial Query Evaluation

1 Introduction

The efficient storage and management of large datasets has been a research topic for decades. Spatial databases are an example of such datasets. Some of the methods used to efficiently manage and query them include distributed algorithms, streaming algorithms, or efficient secondary storage management [9], frequently accompanied by the use of indexes such as R^*-trees to speed up queries.

Compression techniques, on the other hand, are focused on minimizing the storage needs of such datasets, but classical techniques (for example, any Huffman-based compressor) had a strong drawback: the datasets must usually be decompressed from the beginning in order to access any specific item of data. Therefore, compression has been mainly used for archival purposes, or when the whole

dataset must be processed sequentially from beginning to end (for example, combining decompression with streaming algorithms to process the data).

Compact data structures [9] try to combine low space usage and processing efficiency. They store the information in a *compact* (compressed) form, thus requiring less space, and manage (query) it also in its compact form, without having to first decompress it. In this way, it is possible to store and process (query, navigate, and optionally modify) much larger datasets in main memory. This has the additional benefit of the higher speeds of higher levels of the memory hierarchy (more data in processor caches, for example). An example of such a compact data structure is the k^2-tree [1], which will be further discussed in Section 2.2. Initially designed to represent and query web graphs, the k^2-tree has proved to be a powerful tool to represent other kinds of graphs [4], being especially efficient when the graph is clustered. Spatial point datasets, as well as other raster spatial data, can also be managed by a k^2-tree [2].

Distance join queries (DJQs) have received considerable attention from the database community, due to their importance in numerous applications, such as spatial databases and GIS, location-based systems, continuous monitoring, etc. [8]. DJQs are costly queries because they combine two datasets taking into account a distance metric. Two of the most used DJQs are the K Closest Pair Query (KCPQ) and the ε Distance Join Query (εDJQ) [3]. Given two point datasets $\mathcal{P}$ and $\mathcal{Q}$, the KCPQ finds the K closest pairs of points from $\mathcal{P} \times \mathcal{Q}$ according to a certain distance function (e.g., Manhattan, Euclidean, Chebyshev, etc.). The εDJQ finds all the possible pairs of points from $\mathcal{P} \times \mathcal{Q}$ that are within a distance threshold 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 datasets that represent the hotels and the monuments in a touristic city. A KCPQ ($K = 10$) can discover the *10* closest pairs of hotels and monuments, sorted in increasing order by distance. On the other hand, an εDJQ ($\varepsilon = 200$) could return all possible pairs (hotel, monument) that are within *200* meters of each other.

DJQs have been extensively studied, and algorithms exist to answer KCPQ, εDJQ, and other similar queries over plain data [10], as well as taking advantage of indexes such as R-trees or Quadtrees [7]. In this paper, we explore the advantages of representing spatial data using a compact data structure, the k^2-tree, to implement DJQs, and test it with two of the most used DJQs: KCPQ and εDJQ. Thus, the most important contributions of this paper are the following:

- A detailed description of the algorithms to answer KCPQ and εDJQ over large datasets of points, using k^2-trees as the underlying representation for both datasets.
- The execution of a set of experiments using large real-world datasets for examining the efficiency and the scalability of the proposed strategy, considering performance parameters and measures.

This paper is organized as follows. In Section 2 we present preliminary concepts related to DJQs and k^2-trees, as well as previous contributions in these areas. In Section 3, the new algorithms for KCPQ and εDJQ using k^2-trees are

proposed. In Section 4, we present the main results of our experiments, using large real-world datasets. Finally, in Section 5, we provide the conclusions arising from our work and discuss related future work directions.

2 Background and related work

In this section, we review some basic concepts about DJQs and the k^2-tree compact data structure, as well as a brief survey of the most representative contributions in both fields in the context of spatial query processing.

2.1 Distance-based Join Queries - KCPQ and εDJQ

Distance-based Join Queries are special joins where two datasets are combined, taking into account a distance metric (*dist*). DJQs can be very costly when the size of the joined datasets is large, and for this reason, they have lately been thoroughly investigated. Two of the most representative and known DJQs are the K Closest Pairs Query (KCPQ) and the ε Distance Join Query (εDJQ)

The KCPQ discovers the K pairs of data formed from the elements of two datasets having the K smallest distances between them (i.e., it reports only the top K pairs from the combination of two datasets). Formally:

Definition 1. (K **Closest Pairs Query,** K**CPQ**)
Let $\mathcal{P} = \{p_1, p_2, \cdots, p_n\}$ and $\mathcal{Q} = \{q_1, q_2, \cdots, q_m\}$ be two set of points, and a number $K \in \mathbb{N}^+$. Then, the result of the K Closest Pairs Query is an ordered collection, $KCPQ(\mathcal{P}, \mathcal{Q}, K)$, containing K different pairs of points from $\mathcal{P} \times \mathcal{Q}$, ordered by distance, with the K smallest distances between all possible pairs: $KCPQ(\mathcal{P}, \mathcal{Q}, K) = ((p_1, q_1), (p_2, q_2), \cdots, (p_K, q_K)),\ (p_i, q_i) \in \mathcal{P} \times \mathcal{Q},\ 1 \leq i \leq K$, such that for any $(p, q) \in \mathcal{P} \times \mathcal{Q} \setminus KCPQ(\mathcal{P}, \mathcal{Q}, K)$ we have $dist(p_1, q_1) \leq dist(p_2, q_2) \leq \cdots \leq dist(p_K, q_K) \leq dist(p, q)$.

Three properties of KCPQ are: (i) it is symmetric (i.e., $KCPQ(\mathcal{P}, \mathcal{Q}, K) = KCPQ(\mathcal{Q}, \mathcal{P}, K)$); (ii) the cardinality of the query result is known beforehand $|KCPQ(\mathcal{P}, \mathcal{Q}, K)| = K$; and (iii) the distance of the K closest pairs of points is unknown a priori.

On the other hand, the εDJQ reports all the possible pairs of spatial objects from two different spatial objects datasets, $\mathcal{P}$ and $\mathcal{Q}$, having a distance not greater than a threshold ε of each other. Formally:

Definition 2. (ε **Distance Join Query,** ε**DJQ**)
Let $\mathcal{P} = \{p_1, p_2, \cdots, p_n\}$ and $\mathcal{Q} = \{q_1, q_2, \cdots, q_m\}$ be two set of points, and a distance threshold $\varepsilon \in \mathbb{R}_{\geq 0}$. Then, the result of the εDJQ is the set, $\varepsilon DJQ(\mathcal{P}, \mathcal{Q}, \varepsilon) \subseteq \mathcal{P} \times \mathcal{Q}$, containing all the possible different pairs of points from $\mathcal{P} \times \mathcal{Q}$ that have a distance of each other smaller than, or equal to ε: $\varepsilon DJQ(\mathcal{P}, \mathcal{Q}, \varepsilon) = \{(p_i, q_j) \in \mathcal{P} \times \mathcal{Q} : dist(p_i, q_j) \leq \varepsilon\}$

The εDJQ can be considered as an extension of the KCPQ, where the distance threshold of the pairs (ε) is known beforehand and the processing strategy (e.g., plane-sweep technique) can be the same as in the KCPQ for generating the candidate pairs of the final result.

If both $\mathcal{P}$ and $\mathcal{Q}$ are non-indexed, the KCPQ between two point sets that reside in main-memory can be solved using *plane-sweep-based* algorithms [10]. The *Classic plane-sweep* algorithm for KCPQ consists of two steps: (1) sorting the entries of the two points sets, based on the coordinates of one of the axes (e.g. X) and (2) combining the reference point of one set with all the comparison points of the other set satisfying that their distance on the X-axis is less than δ (distance of the Kth closest pair found so far), and choosing those pairs whose point distance is smaller than δ. A faster variant called *Reverse-Run plane-sweep* algorithm is based on the concept of *run* (a continuous sequence of points of the same set that does not contain any point from the other set) and the *reverse* order of processing of the comparison points with respect to the reference point. To reduce the search space and considering the reference point, three methods are applied in these two plane-sweep algorithms: Sliding Strip (δ on X-axis), Sliding Window and Sliding Semi-Circle. These DJQs have been recently designed and implemented in SpatialHadoop and LocationSpark, that are Hadoop-based and Spark-based distributed spatial data management systems (Big Spatial Data context) [5], respectively.

The problem of DJQs has also received research attention by the spatial database community in scenarios where at least one of the datasets is indexed. If both $\mathcal{P}$ and $\mathcal{Q}$ are indexed using 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 [3]. In [7], an extensive experimental study comparing the R*-tree and Quadtree-like index structures for DJQs together with index construction methods was presented. In the case that only one dataset is indexed, in [6] an algorithm is proposed for KCPQ, whose main idea is to partition the space occupied by the dataset without an index into several cells or subspaces (according to a grid-based data structure) and to make use of the properties of a set of distance functions defined between two MBRs [3].

2.2 k^2-tree

A k^2-tree [1] is a compact data structure used to store and query a binary matrix that can represent a graph or a set of points in discretized space. Figure 1 shows in (a) a set of points in a 2D discrete space, and its direct translation into a binary matrix in (b). For the k^2-tree representation, choosing $k = 2$, (c) is the conceptual tree and (d) the actual bitmaps that are stored. T represents the "tree" part (non leaf nodes), and L the leaves of the conceptual tree.

Conceptually, the k^2-tree can be seen as an unbalanced tree, where each node has a bitmap of k^2 bits and up to k^2 children. This conceptual tree is built as follows: its root node corresponds to the full matrix, which is divided into $k \times k$ equal-sized submatrices (for $k = 2$, the matrix is decomposed in $k^2 = 4$ quadrants). For each submatrix, if there is at least one 1 in its cells, the

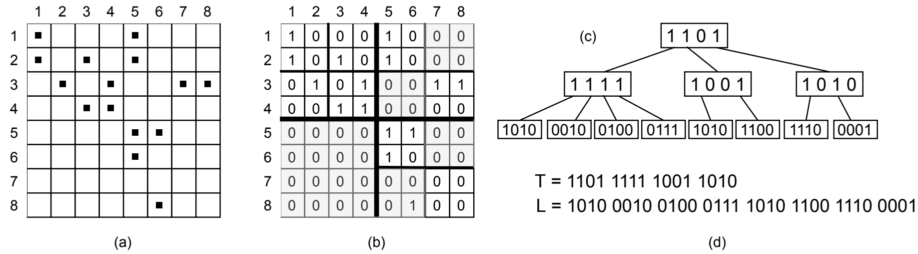

Fig. 1. A 2D-space model with its binary matrix and k^2-tree representations

corresponding bit in the conceptual tree node is set to 1, and the submatrix is included as a child of the node. If the submatrix is empty (there are no 1's) then the corresponding bit is set to 0 and the submatrix is discarded. See, for example, that the root node in Figure 1 is 1101 because quadrants 1, 2 and 4 have 1's, but the third quadrant is full of 0's. The process continues recursively for all non empty submatrices until the individual cells (that correspond to leaves in the k^2-tree) are reached. The actual k^2-tree is just the bitmap that corresponds to the breadth-first traversal of the conceptual tree (usually split in two fragments T and L, as shown in Figure 1(d)).

Points and regions of the original matrix can be easily found traversing the branches of the conceptual k^2-tree from the root node. This is achieved in practice using just the bitmaps by computing $getChild(T, i) = rank(T, i) \times k^2$, that returns, for the node at position i in T, the position in T where its children are located. The $rank(T, i)$ operation (which counts the number of 1's up to position i in T) can be computed in constant time by enhancing T with a small structure of counters.

Many variants and improvements have been proposed over the basic k^2-tree representation introduced here [1, 2]. In this paper we focus on the basic k^2-tree representation described in this section, in order to provide a clear description of the algorithms.

Regarding the use of k^2-trees in the field of DJQs, in the recent work [11], K Nearest Neighbors Query and KCPQ are proposed for k^2-trees representing points of interest. It is the first algorithm in the literature for KCPQ using k^2-trees (ALBA-KCPQ). One of the main drawbacks of their paper is that the authors used very small synthetic and real datasets in the experimentation, because the maximum number of points is only 1 million for each synthetic dataset and for real data the combination was 76451×20480 and 4499454×196902. Moreover, the total response time of the KCPQ experiments is questionable because their implementation of (Classic) plane-sweep-based KCPQ algorithm needed hours to solve the query, when it should be answered in an order of *ms*. These surprising performance results also make KCPQ implementations on k^2-trees and their results questionable. Note also that their implementation, basically of the same algorithm, is in Java and it is not publicly available, and the high-level

pseudo-codes provided in the journal paper omit low-level details that are key to performance. This makes it difficult to accurately reproduce their results from the available information. Finally, to the best of our knowledge, εDJQ has never been tackled using k^2-trees. Here we present efficient implementations of KCPQ and εDJQ to show the interest of the strategy of using k^2-trees to represent spatial data. Our algorithms were coded in C++ and are available to the community, and they were tested with large real-world datasets, comparing them with *Classic* and *Reverse-Run* plane-sweep DJQs on main memory.

3 Our approach to DJQs using k^2-trees

We describe in this section the new algorithms for KCPQ (Algorithm 1) and εDJQ (Algorithm 2) using k^2-trees. For all the distance calculations, we have used the Euclidean distance.

The input for Algorithm 1 consists on two matrices A and B, stored as k^2-trees (they would correspond to the $\mathcal{P}$ and $\mathcal{Q}$ datasets in the definitions in Section 2.1), and the number of pairs of closest points (although we use the name *NumPairs* instead of K in the pseudocode to avoid confusion with the k parameter of k^2-trees), We denote $A[i]$ the ith bit value of the bitmap for A. $A.lastLevel$ is the last level of the tree, corresponding to its leaves. Levels range from 0 to $\lceil lg(n) \rceil - 1$, being n the width of the original matrix.

The following data structures are used by Algorithm 1:

- A priority Queue *PQueue* that stores pairs of nodes to be processed. Each entry in *PQueue* contains:
  - A (sub)matrix of A, including its top-left coordinate and the offset of the associated node in the $T|L$ bitmap of the k^2-tree.
  - A (sub)matrix of B with the same information.
  - The level both matrices belong to in the k^2-tree conceptual trees.
  - The minimum possible distance between the points of A and B. It is computed as shown in Algorithm 3.

  PQueue is a min priority queue over the distance (that is, pairs with lower minimum distance come first). It uses the standard methods: $isEmpty()$, $enqueue()$ and $dequeue()$.
- An ordered list *OutList* with capacity for $NumPairs$ elements (we actually use a max binary heap to manage these elements), each one storing a pair of points (one coming from each input matrix), and the distance between them. The elements (pairs of points) are sorted according to their distance. It uses the methods: $length()$, $maxDist()$ and $insert()$.
- *MinDist(pA,pB,size)*, shown in Algorithm 3, obtains the minimum possible (Euclidean) distance between points of the matrices A and B that have their origins in pA and pB and are squares of $size \times size$.

  Note that the pairs of (A,B) matrices that are generated in Algorithm 1 have a special property: their origin coordinates are always multiples of $size$. This allows us to compute the minimum distance more efficiently, but it would not work to get the minimum distance between two matrices in general.

The idea behind Algorithm 1 is to recursively partition the matrices A and B into k^2 submatrices each and compare each possible pair of submatrices (down to when they are not actually submatrices but really individual cells or points). One of the strong points of this algorithm is that, at some point, we can stop without processing all the remaining pairs of submatrices. This happens when the required number of closest pairs has already been obtained, and the largest distance between them is not larger than the minimum possible distance between the pairs of submatrices not yet processed.

The algorithm follows a Best-First (BF) traversal. It starts by enqueuing the whole matrices (which correspond to the level 0 of the k^2-tree and have 0 as the minimum possible distance between them). The output list *OutList* is also initialized, with room for at most $NumPairs$ elements.

Then, the priority queue is processed until it is empty, or the stop criteria are reached. Lines $6 - 8$ check if the output list already has the target $NumPairs$ elements. If so, and the minimum distance of the current pair of matrices is at least the maximum distance in *OutList*, we can be sure that the current and remaining matrices can be safely discarded, and the algorithm returns the current output list.

In other case, the current matrices are partitioned (lines $11 - 20$), but only if they have children (which is tested by directly using the k^2-tree bitmaps in lines 12 for matrix A and 15 for matrix B). For each pair of child submatrices, if they are in the last level of the k^2-tree then they are actually points. So, if there is room in *OutList* or its maximum distance is greater than the distance between the current pair of points, then the pair and its distance are inserted in order in *OutList* (lines $21 - 25$). Recall that the *insert* operation may need to remove the element with the largest distance if the output list already contains *NumPairs* elements.

If the submatrices are not in the last level of the k^2-tree, and if they meet the conditions to contain candidate pairs of points (*OutList* is not full or the minimum distance between the matrices is less than the maximum distance in *OutList*) they are enqueued in the priority queue (lines $28 - 30$).

Algorithm 2 (εDJQ) uses the same scheme as the previous one, but with some key differences. The input consists now of the two k^2-trees A and B, plus the distance threshold ε. Since the algorithm does not limit the number of output pairs, *OutList* is now an unlimited-size, unordered list. For the same reason, the algorithm does not have an "early exit", and it exits only after the priority queue is empty. Additionally, each element in the priority queue stores not only the minimum possible distance between the matrices, but also the maximum possible distance, computed by the function *MaxDist* (shown in comments in the pseudocode of Algorithm 3). The initial $MaxDist$ for the whole matrices is $\sqrt{2}n$, where n is the width of each matrix.

The partitioning is done the same way, but for every pair of child submatrices the process is different:

- At leaf level of the k^2-trees (lines $18 - 21$) the pair of nodes is inserted in *OutList* if the distance between them is at most ε.

Algorithm 1 GetKCPQ: Get the $NumPairs$ closest points.

```
1: function GetKCPQ(A, B, NumPairs)
2:     PQueue.enqueue({ {(0,0), 0}, {(0,0), 0}, 0, 0})
3:     OutList = new OrderedList(NumPairs)
4:     while not PQueue.isEmpty() do
5:         Node = PQueue.dequeue()
6:         if OutList.length() == NumPairs
7:                 and Node.minDist ≥ OutList.maxDist() then
8:             return OutList
9:         chLevel = Node.Level + 1
10:        chSize = n/k^chLevel
11:        for i = 0 to k^2 − 1 do                    ▷ Directly access the k^2-tree bitmap
12:            if A[Node.A.ptr + i] == 1 then
13:                chPtrA = getChild(A, Node.A.ptr + i)
14:                for j = 0 to k^2 − 1 do
15:                    if B[Node.B.ptr + j] == 1 then
16:                        chPtrB = getChild(B, Node.B.ptr + j)
17:                        childA = {(Node.A.x + chSize · (i mod k),
18:                                Node.A.y + chSize · ⌊i/k⌋), chPtrA}
19:                        childB = {(Node.B.x + chSize · (j mod k),
20:                                Node.B.y + chSize · ⌊j/k⌋), chPtrB}
21:                        if chLevel == A.lastLevel then            ▷ Leaf nodes
22:                            distance = Dist((childA.x, childA.y), (childB.x, childB.y))
23:                            if OutList.length() < NumPairs
24:                              or OutList.maxDist() > distance then
25:                                OutList.insert((childA.x, childA.y), (childB.x, childB.y), distance)
26:                        else
27:                            minDist = MinDist((childA.x, childA.y), (childB.x, childB.y), chSize)
28:                            if OutList.length() < NumPairs
29:                              or OutList.maxDist() > minDist then
30:                                PQueue.enqueue({childA, childB, chLevel, minDist})
31:    return OutList
```

- If the maximum distance ($MaxDist$) between the two matrices is at most ε, then all the combinations of points between the two matrices meet the criteria. We use the *rangeQuery* operation of the k^2-trees to get the points and insert all possible pairs into $OutList$ (lines $23 - 31$).
- Otherwise, if the minimum distance is at most ε, we enqueue the submatrices with the minimum and maximum distances between them (lines $32 - 33$).

4 Experimental Results

We have tested our *DJQ* algorithms using the following real-world 2D point datasets, obtained from OpenStreetMap[3]: *LAKES* (L), that contains boundaries of water areas (polygons); *PARKS* (P), that contains boundaries of parks or green areas (polygons); *ROADS* (R), which contains roads and streets around the world (line-strings); and *BUILDINGS* (B), which contains boundaries of all buildings (polygons). For each source dataset, we take all the points extracted from the geometries of each line-string to build a large point dataset. Additionally, we round coordinates to 6 decimal positions, in order to be able to transform these values to k^2-tree coordinates in a consistent manner. Table 1 summarizes the characteristics of the original datasets and the generated point sets obtained

[3] Available at `http://spatialhadoop.cs.umn.edu/datasets.html`

Algorithm 2 εDJQ: Get all pairs with a distance threshold of ε

```
1: function εDJQ(A, B, ε)
2:     PQueue.enqueue({ {(0,0), 0}, {(0,0), 0}, 0, 0, √(2)n})
3:     OutList = new List()
4:     while not PQueue.isEmpty() do
5:         Node = PQueue.dequeue()
6:         chLevel = Node.Level + 1
7:         chSize = n/k^chLevel
8:         for i = 0 to k^2 − 1 do                    ▷ Directly access the k^2-tree bitmap
9:             if A[Node.A.ptr + i] == 1 then
10:                chPtrA = getChild(A, Node.A.ptr + i)
11:                for j = 0 to k^2 − 1 do
12:                    if B[Node.B.ptr + j] == 1 then
13:                        chPtrB = getChild(B, Node.B.ptr + j)
14:                        childA = {(Node.A.x + chSize · (i mod k),
15:                                 Node.A.y + chSize · ⌊i/k⌋), chPtrA}
16:                        childB = {(Node.B.x + chSize · (j mod k),
17:                                 Node.B.y + chSize · ⌊j/k⌋), chPtrB}
18:                        if chLevel == A.lastLevel then                ▷ Leaf nodes
19:                            distance = Dist((childA.x, childA.y), (childB.x, childB.y))
20:                            if distance ≤ ε then
21:                                OutList.insert((childA.x, childA.y), (childB.x, childB.y), distance)
22:                        else
23:                            minDist = MinDist((childA.x, childA.y), (childB.x, childB.y), chSize)
24:                            maxDist = MaxDist((childA.x, childA.y), (childB.x, childB.y), chSize)
25:                            if maxDist ≤ ε then
26:                                ▷ All pairs in the range satisfy the distance condition
27:                                rangeA = A.rangeQuery(A.x, A.x+chSize-1, A.y, A.y+chsize-1)
28:                                rangeB = B.rangeQuery(B.x, B.x+chSize-1, B.y, B.y+chsize-1)
29:                                for pA ∈ rangeA do
30:                                    for pB ∈ rangeB do
31:                                        OutList.insert(pA, pB, Dist(pA, pB))
32:                            else if minDist ≤ ε then
33:                                PQueue.enqueue({childA, childB, chLevel, minDist, maxDist})
34:    return OutList
```

Algorithm 3 MinDist/MaxDist: min/max possible distance between 2 matrices

```
function MinDist(pA,pB,size)
    ▷ Also MaxDist(pA,pB,size)
    if pA.x = pB.x then
        hdist = 0
    else
        hdist = |pA.x − pB.x| − (size − 1)
        ▷ For MaxDist: hdist = |pA.x − pB.x| + (size − 1)
    if pA.y = pB.y then
        vdist = 0
    else
        vdist = |pA.y − pB.y| − (size − 1)
        ▷ For MaxDist: vdist = |pA.y − pB.y| + (size − 1)
    return √(hdist^2 + vdist^2)
```

from them. Note that all the datasets represent worldwide data, and points are stored as ($longitude, latitude$) pairs.

Table 1. Source datasets and generated point sets

	Source dataset		Generated dataset	
Name	#Records (M)	Size (GiB)	#Points (M)	Size (GiB)
LAKES (L)	8.4	8.6	345	8.6
PARKS (P)	10	9.3	305	7.5
ROADS (R)	72	24	682	17
BUILDINGS (B)	115	26	615	14

The main performance measures that we have used in our experiments are the space required by the data structure vs. the plain representation, and the total execution time to run a given DJQ. We measure elapsed time, and only consider the time necessary to run the query algorithm. This means that we ignore time necessary to load the files, as well as time required to sort the points for the plane-sweep algorithms.

All experiments were executed on an HP ProLiant DL380p Gen8 server with two 6-core Intel® Xeon® CPU E5-2643 v2 @ 3.50GHz processors with 256GiB RAM (Registered @1600 MHz), running Oracle Linux Server 7.9 with kernel Linux 4.14.35 (64bits). Our algorithms were coded in C++ and are publicly available[4]. For the k^2-tree algorithms, the SDSL-Lite[5] library was used.

First, we build the k^2-tree for each dataset. We use the simplest variant of k^2-tree with no optimizations.In order to insert the points in the k^2-tree, they are converted to non-negative integer values. Since we are considering worldwide coordinates in degrees, with 6 decimal places, each coordinate (x, y) is converted to matrix coordinates (r, c) using $(r, c) = ((x + 180) \cdot 10^6, (y + 90) \cdot 10^6)$. In this way, the points fit into a binary matrix with 360 million rows and 160 million columns, that is finally stored as a k^2-tree.

Table 2. Space required by k^2-tree representations

Dataset	Plain (GiB)	Binary (GiB)	k^2-tree (GiB)	Compression ratio
LAKES (L)	8.6	2.7	1.8	0.67
PARKS (P)	7.5	2.4	1.6	0.67
ROADS (R)	17	5.3	2.3	0.43
BUILDINGS (B)	14	4.8	2.3	0.48

Table 2 shows the space required by the k^2-tree representation of each dataset. We display as a reference the plain size of the dataset, as well as a "binary size"

[4] Available at `https://gitlab.lbd.org.es/public-sources/djq/k2tree-djq`
[5] Available at `https://github.com/simongog/sdsl-lite`

estimated considering that each coordinate can be represented using two 32-bit words. Note that each coordinate component can be stored using 28–29 bits for our datasets, but this would make data access slower, so we consider 32 bits to be the minimum cost for a reasonable plane-sweep algorithm that works with uncompressed data. We also display the compression ratio of the k^2-tree relative to the binary input size. Results show that the k^2-tree representation is able to efficiently represent the collection, and the compression obtained improves with the size of the dataset. Notice also that the k^2-tree version we use in our experiments does not include any of the existing optimizations for the k^2-tree to improve compression.

We compared the performance of our KCPQ algorithm with 4 different implementations based on plane-sweep: two implementations of *Classic* plane-sweep, with Sliding Strip (PS-CS) and with Sliding Semi-Circle (PS-CC) respectively, and the equivalent implementations of *Reverse-Run*, with Sliding Strip (PS-RRS) and Sliding Semi-Circle (PS-RRC). We performed our experiments checking all the pairwise combinations of our datasets. Due to space constraints, we display only the results for some combinations, denoted LxP, LxB, PxR, RxB, and PxB. The remaining combinations yielded similar comparison results. For each combination of datasets, we run the KCPQ algorithm for varying $K \in \{1, 10, 10^2, 10^3, 10^4, 10^5\}$.

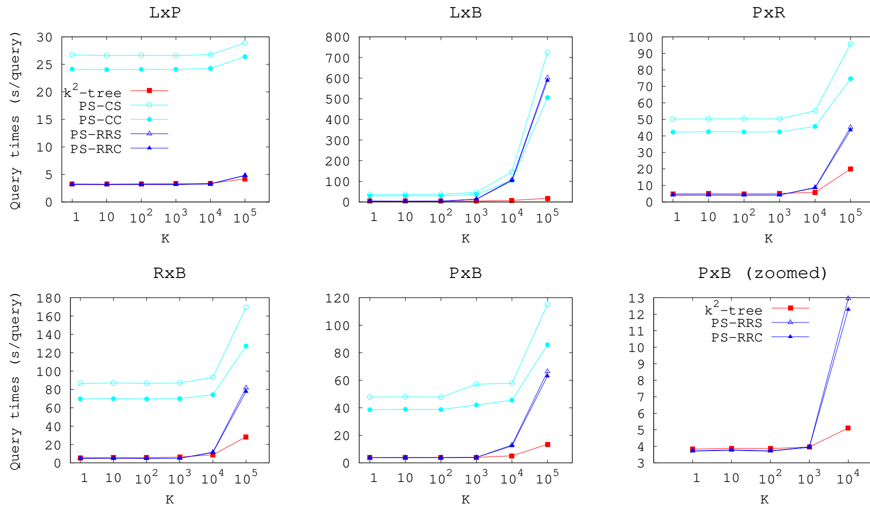

Fig. 2. Query times for KCPQ in k^2-trees and plane-sweep variants, changing K.

Figure 2 displays the query times obtained by our algorithm and the four variants of plane-sweep studied.The first five plots display the results for all variants for 5 different dataset combinations. Results clearly show that the *Classic* variants (PS-CS and PS-CC) are much slower than the other alternatives in

all cases (as in [10]). Therefore, we will focus on the comparison between our proposal and the *Reverse-Run* variants that are competitive with it.

The point datasets used have a significantly different amount of points, and correspond to different features, which leads to very different query times among the plots in Figure 2. However, results show that our algorithm always achieves the best query times for large values of K. Particularly, for $K = 10^5$, our algorithm is between 1.15 and 33 times faster than the best alternative, PS-RRC, depending on the joined datasets. Additionally, we are always the fastest option for $K \geq 10^4$, and in some datasets from $K = 10^3$. For smaller K, our proposal is competitive but slightly slower than the *Reverse-Run* plane-sweep algorithms. The lower right chart of Figure 2 shows a subset of the results for the PxB join, to better display the differences in performance for these smaller values of K. Results are similar in the remaining experiments: for smaller K, the k^2-tree algorithm is 3–15% slower than PS-RRC, depending on the dataset. This evolution with K is due to the characteristics of our algorithm: independently of K, we need to traverse a relatively large number of regions in both k^2-trees, even if many of these regions are eventually discarded, so the base complexity of our algorithm is comparable to that of *Classic* plane-sweep. On the other hand, this means that many candidate pairs have already been expanded and enqueued, so they can be immediately processed if more results are needed, making our algorithm more efficient for larger values of K.

Next, we compare our algorithm for εDJQ with two plane-sweep variants, *Classic* plane-sweep with Sliding Strip (εDJQ-CS) and *Reverse-Run* with Sliding Semi-Circle (εDJQ-RRC). We select a representative subset of joined datasets, namely LxP, PxR, RxB and PxB. In order to measure the scalability of the algorithms, we perform tests for varying $\varepsilon \in (7.5, 10, 25, 50, 75, 100) \times 10^{-5}$ (these values of ε are associated with the original coordinates in degrees, but recall that in the k^2-tree coordinates are scaled to integer values, so values of ε are also scaled accordingly).

Figure 3 displays the results obtained for each join query. Our algorithm based on k^2-trees is slower for LxP, but much faster in most cases for PxR, RxB and PxB (notice the logarithmic scale for query times). We attribute this difference mainly to the size of the datasets: LxP joins the two smallest datasets, whereas the remaining configurations involve one or two of the larger datasets. For these 3 larger joins, our algorithm is always much faster for the smaller values of ε. In this case, our algorithm does not improve for larger ε, as for KCPQ, because no early stop condition exists: we must traverse all candidate pairs as long as their minimum distance is below ε, and for very large ε the added cost to traverse the k^2-trees to expand many individual pairs makes our proposal slower, even if we are able to efficiently filter out many candidate regions. These queries with smaller values of ε, in which we are much faster than plane-sweep algorithms, are precisely the ones that would most benefit from our approach based on compact data structures, since the number of query results increases with ε: for $\varepsilon = 100 \times 10^{-5}$ we obtain over 10^9 results, and these results would become the main component of memory usage. Notice that, in practice, in our

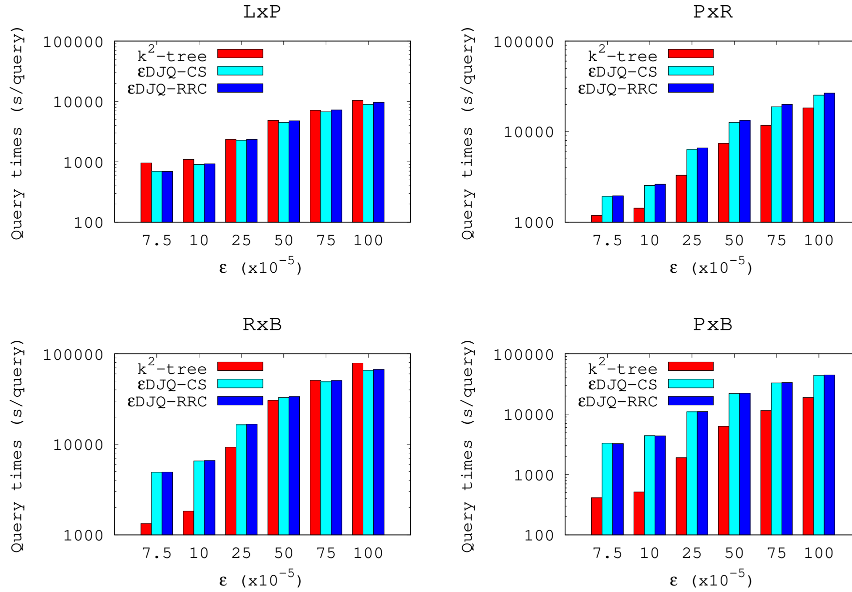

Fig. 3. Query times for εDJQ in k^2-trees and plane-sweep variants, changing ε.

experiments we measure the time to retrieve and count the query results, but do not store them in RAM to avoid memory issues in some query configurations.

5 Conclusions and Future Work

We have introduced two algorithms to solve DJQs on top of the k^2-tree representation of point datasets. Our proposal takes advantage of the compression and indexing capabilities of the k^2-tree to efficiently answer KCPQ and εDJQ queries in competitive time and with significantly lower memory requirements. Our results show that our algorithms for KCPQ queries are competitive with the alternatives for small K, but become much faster than plane-sweep algorithms for larger values of K. Our algorithm for εDJQ also achieves competitive query times and is especially faster when the join query involves the largest datasets.

As future work, we plan to test the performance of our algorithms with other variants of the k^2-tree, that are able to obtain similar query times but require much less space [1]. Particularly, our algorithms can be adjusted to work with hybrid implementations of the k^2-tree, that use different values of k, as well as variants that use statistical compression in the lower levels of the conceptual tree. Another interesting research line would be the application of these DJQ algorithms based on k^2-tree in Spark-based distributed spatial data management systems, since they are more sensitive to memory constraints. Finally, we plan

to explore other DJQ and similar algorithms that may also take advantage of the compression and query capabilities of k^2-trees.

Acknowledgments

Guillermo de Bernardo, Miguel R. Penabad and Nieves R. Brisaboa are partially funded by: MCIN/AEI [PDC2021-121239-C31 (FLATCITY-POC), PDC2021-120917-C21 (SIGTRANS, NextGenerationEU/PRTR), PID2020-114635RB-I00 (EXTRACompact), PID2019-105221RB-C41 (MAGIST)]; ED431C 2021/53 (GRC), GAIN/Xunta de Galicia; and as CITIC members are also partially funded by ED431G 2019/01 (CSI), Xunta de Galicia, FEDER Galicia 2014-2020. The work by Antonio Corral was partially funded by the EU ERDF and the Andalusian Government (Spain) under the project UrbanITA (ref. PY20_00809) and the Spanish Ministry of Science and Innovation under the R&D project HERMES (ref. PID2021-124124OB-I00)

References

1. Brisaboa, N.R., Ladra, S., Navarro, G.: Compact representation of web graphs with extended functionality. Information Systems **39(1)**, 152–174 (2014)
2. Brisaboa, N.R., Cerdeira-Pena, A., de Bernardo, G., Navarro, G., Óscar Pedreira: Extending general compact querieable representations to GIS applications. Information Sciences **506**, 196–216 (2020)
3. Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M.: Algorithms for processing k-closest-pair queries in spatial databases. Data & Knowledge Engineering **49**(1), 67–104 (2004)
4. Álvarez García, S., Brisaboa, N., Fernández, J.D., Martínez-Prieto, M.A., Navarro, G.: Compressed vertical partitioning for efficient RDF management. Knowl. Inf. Syst. **44**(2), 439–474 (aug 2015)
5. García-García, F., Corral, A., Iribarne, L., Vassilakopoulos, M., Manolopoulos, Y.: Efficient distance join query processing in distributed spatial data management systems. Information Sciences **512**, 985–1008 (2020)
6. Gutiérrez, G., Sáez, P.: The k closest pairs in spatial databases - when only one set is indexed. GeoInformatica **17**(4), 543–565 (2013)
7. Kim, Y.J., Patel, J.M.: Performance comparison of the R*-tree and the quadtree for kNN and distance join queries. IEEE Transactions on Knowledge and Data Engineering **22**(7), 1014–1027 (2010)
8. Mamoulis, N.: Spatial Data Management. Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2012)
9. Navarro, G.: Compact Data Structures: A Practical Approach. Cambridge University Press, USA (2016)
10. Roumelis, G., Vassilakopoulos, M., Corral, A., Manolopoulos, Y.: A new plane-sweep algorithm for the k-closest-pairs query. In: SOFSEM. pp. 478–490 (2014)
11. Santolaya, F., Caniupán, M., Gajardo, L., Romero, M., Torres-Avilés, R.: Efficient computation of spatial queries over points stored in k^2-tree compact data structures. Theoretical Computer Science **892**, 108–131 (2021)