Efficient Distance Queries on Non-point Data

ACHILLEAS MICHALOPOULOS, Computer Science and Engineering, University of Ioannina, Ioannina, Greece and Archimedes Unit, Athena Research Center, Marousi, Greece
DIMITRIOS TSITSIGKOS, Computer Science and Engineering, University of Ioannina, Ioannina, Greece
PANAGIOTIS BOUROS, Computer Science, Johannes Gutenberg Universitat Mainz, Mainz, Germany
NIKOS MAMOULIS, Computer Science and Engineering, University of Ioannina, Ioannina, Greece and Archimedes Unit, Athena Research Center, Marousi, Greece
MANOLIS TERROVITIS, Athena Research Center, Marousi, Greece

Distance queries, including distance-range queries, k-nearest neighbors search, and distance joins, are very popular in spatial databases. However, they have been studied mainly for point data. Inspired by a recent approach on indexing non-point objects for rectangular range queries, we propose a secondary partitioning approach for space-partitioning indices, which is appropriate for distance queries. Our approach classifies the contents of each primary partition into 16 secondary partitions, taking into consideration the begin and end values of objects with respect to the spatial extent of the primary partition. Based on this, we define algorithms for three popular spatial query types, that avoid duplicate results and skip unnecessary computations. We compare our approach to the previous secondary partitioning method and to state-of-the-art data-partitioning indexing and find that it has a significant performance advantage.

CCS Concepts: • **Information systems → Geographic information systems;**

Additional Key Words and Phrases: Spatial indexing, distance queries, nearest neighbor search, distance join, non-point data

ACM Reference Format:
Achilleas Michalopoulos, Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2024. Efficient Distance Queries on Non-point Data. *ACM Trans. Spatial Algorithms Syst.* 11, 1, Article 1 (November 2024), 37 pages. https://doi.org/10.1145/3698194

1 Introduction

Large collections of spatial objects with extent, such as polygons, polylines, and so on, are available in many scientific and application domains, including Geographic Information Systems, graphics

Funded by the Hellenic Foundation for Research and Innovation (HFRI) under the "2nd Call for HFRI Research Projects to support Faculty Members and Researchers" (Project No. 2757).
Authors' Contact Information: Achilleas Michalopoulos, Computer Science and Engineering, University of Ioannina, Ioannina, Epirus, Greece and Archimedes Unit, Athena Research Center, Marousi, Attica, Greece; e-mail: amichalopoulos@cs.uoi.gr; Dimitrios Tsitsigkos, Computer Science and Engineering, University of Ioannina, Ioannina, Epirus, Greece; e-mail: dtsitsigkos@cse.uoi.gr; Panagiotis Bouros, Computer Science, Johannes Gutenberg Universitat Mainz, Mainz, Germany; e-mail: bouros@uni-mainz.de; Nikos Mamoulis, Computer Science and Engineering, University of Ioannina, Ioannina, Epirus, Greece and Archimedes Unit, Athena Research Center, Marousi, Attica, Greece; e-mail: nikos@cs.uoi.gr; Manolis Terrovitis, Athena Research Center, Marousi, Attica, Greece; e-mail: mter@imis.athena-innovation.gr.

ACM 2374-0353/2024/11-ART1
https://doi.org/10.1145/3698194

[22], medical science [32], and location-based services [9]. The problem of indexing spatial objects which are not points has been studied for decades and has reached a high level of maturity [27]. Given a collection of non-point spatial objects, their ***minimum bounding rectangles*** (**MBRs**) are computed and added to a spatial index (i.e., spatial access method), such as the R-tree [18]. Spatial queries are then processed in two steps: in the *filter* step, the index is used to eliminate objects that cannot be part of the query result; in the *refinement* step, the exact geometries of non-filtered objects are accessed and the query is evaluated on them.

Nevertheless, the data management model has changed over the years. Up until 1–2 decades ago, the dominant model was to store the data on the hard disk of a single machine and use hierarchical disk-based indices for MBRs (e.g., the R-tree [18]). Nowadays, given the fact that memory chips have become much bigger and cheaper, we can store and index large spatial data collections in memory [41, 42]. In addition, given the advent of cloud computing, we can spatially partition big spatial data collections to multiple machines and store/index them in their main memories [1, 14, 31, 43, 46]. In this article, we focus on in-memory spatial indexing and query evaluation.

Space-oriented partitioning (**SOP**) indices (e.g., grid, quad-tree) divide the space into *spatially disjoint* partitions. ***Data-oriented partitioning*** (**DOP**) indices allow overlapping partitions, such that each object is assigned to exactly one partition. SOP indexing (especially grids) are more preferable to DOP for large-scale indexing, because partitions relevant to queries can be identified very fast; hence, searches and updates are much faster compared to DOP indices [23, 29, 30, 35, 40, 47]. In addition, query evaluation over SOP partitions can be easily parallelized; thus, SOP has become the de facto approach in distributed spatial data management systems [14, 43, 46, 48].

If the indexed data collection consists of points only, disjoint space partitioning by SOP is guaranteed to assign each point object to exactly one partition. In this article, we focus on non-point objects, whose geometry is extended. For example, such objects are represented as polygons, linestrings, and so on. In a **disjoint space partitioning** (**SOP**) it might not be possible to guarantee that the MBR of each object is enclosed by a single partition. Therefore, each object MBR that intersects multiple partitions needs to be replicated to all these partitions. For example, consider the four rectangles r_1, r_2, r_3, and r_4, depicted in Figure 1, which represent the MBRs of parks in a city. Assume that the data space is divided by a 4×4 grid, as shown in the picture, which defines 16 rectangular tiles T_1–T_{16}. The tiles do not overlap each other and their spatial union is the data space (i.e., they form a SOP index). Three out of the four rectangles (r_1, r_2, r_4) overlap more than one tile, so they are assigned to multiple tiles (e.g., r_2 is assigned to T_5 and T_6). If a spatial range query intersects multiple partitions, then it is possible that the same object is identified as a query result multiple times. For example, a range query that covers both T_5 and T_6 would produce r_2 as a result in both these tiles. Hence, possible duplicates in query results should be eliminated [12, 48]. Recently, Tsitsigkos et al. [41, 42] proposed a *secondary partitioning approach* for the *filter step* of spatial range queries on non-point objects. This approach divides the object MBRs inside each SOP partition into four classes based on whether they begin inside the partition or not in each of the two space dimensions. Given a range query, only a subset of object classes in each partition is selected, such that the query result is guaranteed not to have duplicates. Besides avoiding duplicate results, this approach also reduces the number of comparisons in each partition significantly, improving the overall performance.

In this article, we focus on distance queries over non-point data. Such queries include nearest neighbor search, ϵ-range queries, and ϵ-distance joins. Distance queries are fundamental in **geographical information systems** (**GIS**), location-based services, transportation, healthcare, and many other fields. For example, distance range queries facilitate finding residential areas within a specific radius from a pollution source and distance joins find water bodies within a certain distance of industrial facilities. If the spatial objects are indexed by a SOP data structure, distance queries may produce duplicate results. For instance, consider query point q in Figure 1, which may

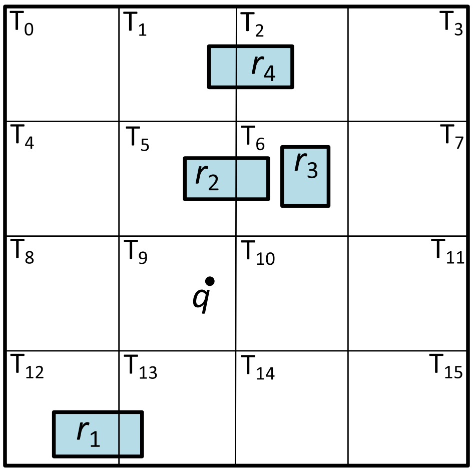


Fig. 1. Distance queries and duplicate results in SOP.

correspond to a mobile user location. Assume that the user is interested in finding the nearest park to its location; such a **nearest neighbor** (**NN**) query would have as a result object r_2. This query would search for the nearest neighbors of q in the nearby tiles to T_9 (which includes q) and would find some objects multiple times (e.g., r_1 would be found in T_{12} and T_{13} and r_2 would be found in both T_5 and T_6). The previously proposed approach for handling duplicates [41, 42] fails in the case of distance-based queries such as NN search because its secondary partitioning is *asymmetric*, i.e., the four classes are determined based on the begin point of the rectangle's projection at each dimension, whereas the end point of the projection is ignored. A workaround for distance range queries (called disk queries in [41, 42]) was proposed, but k-NN queries and distance joins were not studied at all.

Contributions. In this article, we improve upon the secondary partitioning of [41, 42], by defining four classes per dimension (instead of two), which capture more precisely the position of a rectangle with respect to all directions. The goal is to address distance-based spatial queries (nearest neighbor queries, range distance queries, distance joins), which cannot be handled by the scheme of [41, 42]. As a result, in the 2D space, we end up with 16 classes of rectangles. This does not bring any overhead for rectangular range queries, because each of the 16 classes can be mapped to one of the original four classes in [41, 42] and the rectilinear range query algorithms can be readily applied. On the other hand, the definition of the 16 classes can bring a significant performance improvement for distance queries. Under this premise, we design novel algorithms for the filter step of both incremental and traditional k-NN search on top of a grid index which adopts our 16-classes secondary partitioning scheme. For each query, we show which partitions need to be considered at each tile, such that duplicates can be avoided. In addition, for k-NN queries, we propose a 2-step evaluation approach, which takes advantage of the class cardinalities at each tile to avoid distance computations for objects that are definitely in the k-NN set. Under the same principle, we present a solution also for the distance range or disk queries, another popular distance query type.

We also study the filter step for the computation of ϵ-distance joins: given two sets of spatial objects R and S, the objective is to find all pairs of objects (r, s), $r \in R$ and $s \in S$, such that $dist(r, s) \leq \epsilon$, where $dist$ is the Euclidean distance function. Our objective is to efficiently retrieve the object MBRs which are within distance ϵ from each other. Given the replication of objects in multiple partitions of a SOP index, the main challenge is to avoid producing duplicate pairs. For example, in Figure 1 if the distance bound ϵ equals the length of the side of a tile, then r_2 and r_4 are near each other, but this pair could be reported when examining tile combinations (T_1, T_5), (T_1, T_6), (T_2, T_5), (T_2, T_6). Our objective is to avoid considering all object classes when examining a pair of tiles that would lead to duplicate avoidance and minimize the necessary computations at the same time. We study ϵ-distance joins under two settings. When none of the inputs are indexed in advance, we show how to compute the join by first building two temporary secondary partitioning schemes on top of two grids with the same granularity and then breaking down the query into small class-to-class join tasks. Otherwise, if both inputs are already indexed by our secondary partitioning (i.e., to answer typical NN or disk queries) but on top of grids with different granularities, we devise an online transformation to avoid re-indexing one of the inputs. This transformation aligns the grids of the two schemes and then uses our class-to-class break down solution.

Finally, we compare a grid index that implements our secondary partitioning approach against the state-of-the-art SOP index [41, 42] and the best performing DOP index (an in-memory R-tree implementation from boost.org) using real data to demonstrate its superiority for incremental NN and k-NN search, ϵ-range and ϵ-distance join queries.

The contributions of this work can be summarized as follows:

— We propose a secondary partitioning technique for SOP indices over non-point data that is appropriate for distance queries.
— We design algorithms for the filter step of three popular distance queries that take advantage of our object classification at each tile for efficient query evaluation with duplicate result avoidance. We prove the correctness of our approaches.
— We implemented algorithms for traditional and incremental k-NN, ϵ-range, and two different algorithms for ϵ-distance joins, based on their partition granularities. We conducted extensive experiments with state-of-the-art competitors using real data, and demonstrated the superiority of our approach.

Comparison to previous work. This article extends our previous work in [28], which focused solely on the nearest neighbor search, i.e., incremental NN and traditional k-NN queries, toward two directions. Specifically, we consider (1) the computation of another popular type of distance queries on non-point objects, i.e., the *disk* or ϵ-range queries (Section 4.3), and (2) the computation of ϵ-distance joins (Section 5).

Outline. The rest of the article is organized as follows. Section 2 reviews related work. Section 3 presents our secondary partitioning scheme that defines 16 classes of rectangles per tile. In Section 4, we propose algorithms for incremental NN, k-NN, and ϵ-distance queries using our scheme and Section 5 addresses ϵ-distance joins. In Section 6, we evaluate our methods and, finally, Section 7 concludes the article with a discussion about future work.

2 Background and Related Work

In this section, we review related work on spatial access methods, secondary partitioning in SOP [42], and distance-based queries.

2.1 Spatial Access Methods

The problem of defining data structures for geometric objects emerged from computational geometry [33]; later it became a popular subject in database research [15]. The first access methods, including grid files, kd-trees and quad-trees, were designed for range queries over multidimensional point data [6]. Later, the focus shifted to the more difficult case of indexing extended objects (e.g., polygons, linestrings). These data structures, such as the point-region quad-tree, are typically extensions of point-data structures that *clip* objects spanned by partition boundaries [37]; this is unavoidable if partition boundaries are fixed and partitions are disjoint.

A major departure from this model were DOP methods, which adapt the partition boundaries to the data distribution and enable overlap between the spatial extents of partitions. The dominant method in this category are the R-tree [18] and its variants (e.g., the R*-tree [4]), which hierarchically group the MBRs of objects into nodes to form a multi-way search tree. The primary focus of indexing in the 80's and 90's was to minimize the search I/O cost in range search (and other spatial queries), given that the data structures do not fit in memory and they are disk based.

More recently, the focus has shifted to main-memory indexing, due to the increase of memory capacities compared to the data volumes to be indexed. Indeed, most collections for non-point objects (e.g., geographic data) are not huge in terms of numbers of objects, but they could occupy a lot of space, mainly due to the potentially high complexity of each individual object (e.g., a polygon representation of a single geographic object could have thousands of edges). This means that the MBR approximations of objects typically fit in the memory of a commodity machine and main-memory indexing is feasible. Given this, the state-of-the-art DOP approaches such as efficient R-tree implementations (e.g., see www.boost.org) may lose their advantage over SOP approaches, such as grids [30, 40], which are designed for speedy searches and updates.

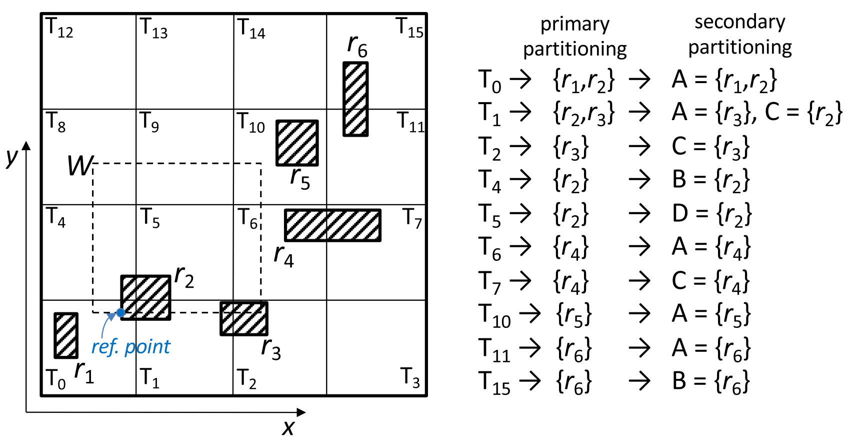
Following the recent trend on machine learning, learned indices for spatial data were proposed [10, 13, 16, 17, 25, 34, 44]. Al-Mamun et. al. [2] present an extended survey about the multi-dimensional learned indices. Most of these methods do not support k-NN queries and those which do apply on *point data* only, whereas the focus of our research is on non-point data. In [13] authors proposes a learning approach that improves the construction of an R-tree, based on the query workloads; they support range and approximate k-NN queries. RLR-tree [17] propose machine learning models that improve the construction of an R-tree in the presense of workloads with large number of insertions, where bulkloading is not effective. PLATON [44] is a top down tree packing method that leverages Monte Carlo tree search to discover an optimal partition policy. These indices are not directly comparable to our work, as their goal is to minimize the I/O cost.

2.2 Secondary Partitioning for SOP

SOP approaches are designed for point data, where each data object (point) is assigned to exactly one partition. When applied on non-point data (e.g., MBRs) some objects may have to be replicated (or clipped) to multiple partitions. This introduces space overhead, which is not high in real applications (e.g., GIS), where the partition extents are significantly larger than the extents of data objects. Still, an important issue in having duplicates is that a query may potentially retrieve the same result twice if the object is included in multiple partitions.

For example, consider the six MBRs $r_1 \dots r_6$ partitioned by a spatial grid as shown in Figure 2; each rectangle is assigned all tiles that intersect it (e.g., r_3 is assigned to tiles T_1 and T_2). The range query defined by rectangle W should retrieve all MBRs that intersect it (e.g., r_2 and r_3). However, if the query is processed independently in each of the 9 tiles that intersect it, r_2 will be retrieved four times and r_3 will be retrieved two times.

Duplicate results can be avoided by the definition and use of a *reference point* for each rectangle r_i that intersects W [12]. For example, the reference point could be the smallest x and y values of the rectangle which is the common area of r_i and W. Then, each tile T_j processes the query independently, and for each result r_i, it computes the reference point. If the reference point is *inside* T_j, then r_i is reported; otherwise, it is ignored. This ensures that there will be no duplicate results, because the reference point is guaranteed to be inside a single tile only. For example, r_2 will be reported only by tile T_0.



	primary partitioning	secondary partitioning
T_0 →	$\{r_1, r_2\}$ →	$A = \{r_1, r_2\}$
T_1 →	$\{r_2, r_3\}$ →	$A = \{r_3\}$, $C = \{r_2\}$
T_2 →	$\{r_3\}$ →	$C = \{r_3\}$
T_4 →	$\{r_2\}$ →	$B = \{r_2\}$
T_5 →	$\{r_2\}$ →	$D = \{r_2\}$
T_6 →	$\{r_4\}$ →	$A = \{r_4\}$
T_7 →	$\{r_4\}$ →	$C = \{r_4\}$
T_{10} →	$\{r_5\}$ →	$A = \{r_5\}$
T_{11} →	$\{r_6\}$ →	$A = \{r_6\}$
T_{15} →	$\{r_6\}$ →	$B = \{r_6\}$

Fig. 2. Primary and secondary partitioning [41, 42].

Still, duplicate avoidance by the reference point approach does not save the cost of computing query results before eliminating them. For example, T_1 will have to first identify r_2 as result, then compute the reference point, and finally ignore r_2 because the reference point is not inside T_1. Tsitsigkos et al. [41, 42] proposed a *secondary partitioning* technique, which divides the MBRs in each tile into four classes based on whether they begin inside or before the tile in each dimension. For example, r_2 in T_4 is classified as class B rectangle because it begins inside T_4 in dimension x but before T_4 in dimension y. For each (rectangular) range query W, the query evaluation approach of [42] selects for each tile T_j intersecting W, only the classes that would not produce duplicates. For example, in Figure 2, for tile T_4, only rectangles of classes A and C will be compared to W, because any rectangle in classes B and D that intersects W would be a duplicate result (e.g., r_2 will be reported in tile T_0). This approach not only avoids duplicate results, but also saves the effort of computing them.

2.3 Distance-based Queries

In this article, we focus on distance-based queries, including distance-range queries (e.g., find all objects within distance at most ϵ to my current location), k nearest-neighbor (k-NN) queries (e.g., find the 10 nearest objects to my current location), and ϵ-distance joins (e.g., find pairs of lakes and forests having distance at most ϵ). Below, we provide formal definitions of these queries.

Definition 1 (ϵ-range Query). Given a collection of spatial objects R, a query point q, and a distance bound ϵ, the ϵ-range query retrieves the subset R' of objects in R, such that $\forall r \in R', dist(q,r) \leq \epsilon$ and $\forall r \in R \setminus R', dist(q,r) > \epsilon$. $dist(q,r)$ refers to the minimum Euclidean distance between q and object r.

Definition 2 (k-NN Query). Given a collection of spatial objects R, a query point q, and a positive integer k, such that k is smaller than the number of objects in R, the k-nearest neighbor query (kNN query) retrieves a subset R' of k objects in R, such that $\forall r \in R', r' \in R - R', dist(q,r) \leq dist(q,r')$. $dist(q,r)$ refers to the minimum Euclidean distance between q and object r.

Definition 3 (ϵ-distance Join). Given two collections of spatial objects R and S and a distance bound ϵ, the ϵ-distance join query retrieves the subset J of $R \times S$, such that for every pair of objects $(r,s) \in J$, $dist(r,s) \leq \epsilon$ and for every pair of objects in $(r,s) \in R \times S \setminus J$, $dist(r,s) > \epsilon$. $dist(r,s)$ refers to the minimum Euclidean distance between objects r and object s.

Previous work on distance-based queries has mainly focused on point data, although it can be extended for non-point objects. A distance range query can be processed similarly to a rectangular range query; the partitions of the spatial index having minimum distance greater than the distance bound ϵ to the query object q are pruned, whereas the others are explored. In the case of a hierarchical index (e.g., R-tree), search is conducted in a depth-first manner, following the index entries (partitions) having distance at most ϵ from q. Since the MBR $M(r)$ of an object r spatially contains r, we can easily show that $dist(q, M(r)) \leq dist(q, r)$. Hence, spatially indices can effectively be used to prune objects r such that $dist(q, M(r)) > \epsilon$, as they may not be query results; for the retrieved objects having $dist(q, M(r)) \leq \epsilon$, a refinement step may be necessary to find whether the object is within distance ϵ from q. Still, if the **minimum maximum distance** (**MINMAXDIST**) [36] between q and an MBR is at most ϵ, the objects is guaranteed to be a result without a refinement step.

For k-NN queries, we can extend the depth-first search algorithm to explore the space using the index while keeping in a k-maxheap H the k-NNs found so far [36]. If the distance of an index entry (partition) is greater than the distance of the k-th NN, the entry (partition) can be eliminated. This approach was extended to form the popular *distance browsing* algorithm [20], which organizes the entries (partitions) to be examined in a priority queue Q (i.e., a minheap), such that the entry whose extent has the smallest minimum distance to q is at the top of Q. At each iteration, the top entry e of Q is de-heaped; if e is a partition containing objects (e.g., a leaf node of the R-tree) or smaller partitions (e.g., a non-leaf node of the R-tree), its contents (objects or extents of smaller partitions) are added to Q; otherwise, e is an object and it is reported as the next neighbor of q. This approach can readily be used for *incremental* NN search, where k is not specified and the user/application would like to retrieve objects in increasing distance to q until a satisfactory number of objects are retrieved or any other condition holds. For example, a user may retrieve the nearest restaurant, examine it, decide that it is not appropriate, continue to get the next nearest restaurant, and so on until the user is satisfied. In both cases of k-NN queries and incremental NN queries, objects can be examined in order of their distances between their MBRs and q and a refinement step can be applied as necessary to find the nearest neighbors according to the actual object geometry.

The algorithm of [20] has been widely used in many contexts, most notably for similarity queries in multi-dimensional spaces [8]. It has also been used in combination with fast-to-update SOP indexing for k-NN queries over moving objects. Specifically, Mouratidis et al. [30] proposed the use of a spatial grid to monitor the locations of moving objects. To answer a k-NN query, the tiles of the grid are accessed incrementally based on their distances to the query location q, until the k-NNs are guaranteed to be found (i.e., the minimum distances tiles not accessed yet to q is greater than the next neighbor's distance). To avoid computing the minimum distance from q to a large number of tiles, the tiles are hierarchically grouped into stripes and entire tile groups can be eliminated. This approach, like the majority of previous work on NN search, focuses on point data; in the next sections, we show how we can apply our secondary partitioning scheme on SOP indices to efficiently perform distance-based search on extended objects.

ϵ-distance join algorithms extend previous work on spatial intersection joins [7] by replacing the intersection condition between partitions by a minimum distance condition [38]. Grid-based evaluation of ϵ-distance joins was investigated (among other works) in [5]. Finally, an alternative definition of distance joins, where the objective is to retrieve the k-closest pairs was studied in [11, 19].

In the rest of the article, we present our methodology for enhancing SOP indices to effectively support evaluation of distance queries over objects with spatial extent (i.e., non-point objects). Table 1 presents the frequently used notation in the article.

Table 1. Table of Notations

notation	description
R, S	collections of spatial objects
r, s	spatial objects
d	dimensionality
$\mathcal{T}$	grid index
T	partition (tile) in a grid index
q	query location
ϵ	distance bound (real number)
k	number of required neighbors (positive integer)
T_q	partition (tile) that includes query point q
G	group of partitions (tiles)
Q	Minheap of objects/tiles/groups in NN algorithms
H	k-maxheap with kNN results so far
δ	width/length of a tile
λ	level around T_q with possible kNN results
μ	level around T_q with guaranteed kNN results

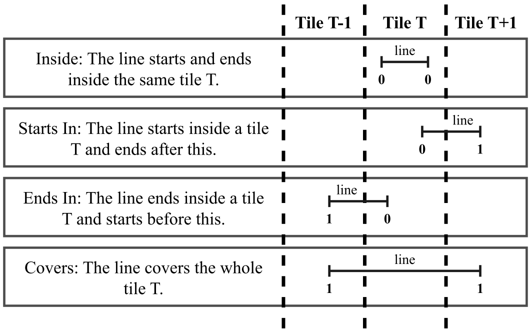

Tile T-1
Tile T
Tile T+1
Inside: The line starts and ends inside the same tile T.
line
0
0
Starts In: The line starts inside a tile T and ends after this.
line
0
1
Ends In: The line ends inside a tile T and starts before this.
line
1
0
Covers: The line covers the whole tile T.
line
1
1

Fig. 3. Class decomposition per axis.

3 Class Decomposition

The secondary partitioning approach proposed in [41, 42] was mainly designed for rectangular range queries and not for distance-based queries. Recall, from Section 2.2, that this scheme divides the contents of a tile in four classes, depending on whether the *begin* points (i.e., values) of the object's projections in each axis begin inside or before the tile. On the other hand, the *end* points of the projections are not considered. As a result, given a tile T and the class of an object o which is assigned to T, we may not be able to infer a tight lower bound for the distance between o and a query object q, if the tile is *before* q in one or both dimensions.

Given this, we propose to define secondary partitions of a tile T (or an SOP partition in general) based on both the *begin* and *end* points of the objects assigned to T. Figure 3 shows the four cases of an object's projection w.r.t. the projection of a tile T whereto o is assigned. The object could be inside T, start inside T and end after T, start before T and end inside T, or start before and end after T. The four cases are encoded by two bits, as shown in the figure. The first bit position refers to the begin point of the object's projection and the second to the end point of the object's projection. Bit value 0 denotes that the begin (end) point is inside T, whereas bit value 1 means that the end point is before (after) T.

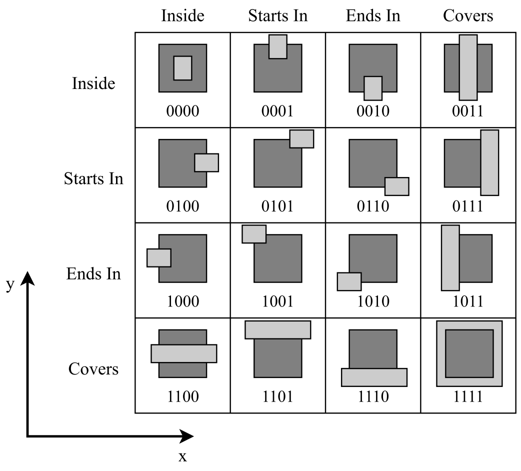


Fig. 4. Class decomposition in a 2D tile.

To encode divisions of a tile (partition) T in the d-dimensional space, we need $2 \cdot d$ bits, i.e., two per dimension, because we have d projections of T (and the objects); this results into 2^{2d} partitions. Hence, in the 2D space, we have $2^4 = 16$ classes of objects (i.e., secondary partitions of a tile T), as illustrated in Figure 4. In this figure, a tile T is denoted by the dark gray square and each of the 16 examples show a case of an object o (light gray rectangle), which belongs to each of the 16 divisions. For example, the upper-left corner shows an object belonging to class `0000`, as the object is inside the tile in both dimensions; class `0001` means that the object is inside the tile in dimension x, but starts inside and ends outside the tile in dimension y, and so on.

Given the above classification, our proposed indexing scheme partitions the objects (object MBRs) using an SOP index (e.g., a grid), such that each object o is assigned to *all* partitions (e.g., tiles) whose spatial extent intersects o. Within each tile, the assigned objects, are re-partitioned to classes (i.e., 16 classes in the 2D space). We next show how our re-partitioning scheme can be used to evaluate distance-based queries efficiently and without producing duplicate results. In a nutshell, while processing a query, we (1) minimize the number of classes to be considered at each tile (and, hence, minimize the number of objects to be considered overall in each tile), (2) avoid duplicate results, and (3) use the class cardinalities to overall avoid comparisons for some query results.

4 Nearest Neighbor and ϵ-Range Queries

In the next two sections, we discuss how to use our secondary partitioning scheme to efficiently answer distance queries. This section focuses on queries with a single set of rectangles as input; specifically, it presents algorithms for NN search and distance range queries. We start with an algorithm for *incremental* NN search (a.k.a. distance browsing [20]), where the number k of desired NNs is not specified *a priori* and the user would like to retrieve the objects in increasing distance from the query object q, until an ad-hoc stopping condition is met. Then, we propose an algorithm that takes advantage of a user-specified number k of NNs, to reduce the number of distance computations and en-heaping/de-heaping operations. The processing of ϵ-distance range queries directly follows from the k-NN evaluation algorithm. Throughout the presentation, we assume that the query object q is a point; however, the distance-range and NN algorithms can be applied also for non-point query objects, in a straightforward manner.

⬟ Type 0	⬠ Type 1	● Type 2	◇ Type 3	○ Type 4	○ Type 5	◆ Type 6	□ Type 7	■ Type 8
`X0X0`	`XXX0`	`0XX0`	`X0XX`	`XXXX`	`0XXX`	`X00X`	`XX0X`	`0X0X`

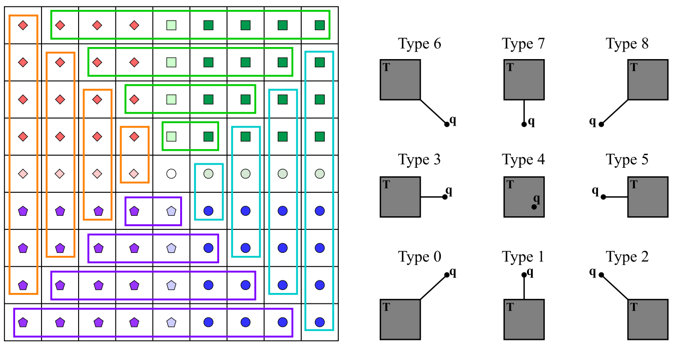


Fig. 5. Tile types; grouping and distribution in space.

Although the algorithms can be adapted for all SOP indices (e.g., arbitrary grids, quad-trees, k-d-trees), we will assume that the set R of data objects (i.e., object MBRs) are indexed by a regular grid, where the domain of each dimension is divided into N equi-width partitions. Hence, we assume a $n \times n$ grid $\mathcal{T}$, where each object is assigned to all tiles (cells) that intersect it and the objects in each tile are re-partitioned based on our classification scheme, presented in Section 3.

4.1 Incremental NN Search

Consider a query location q and assume that we want to retrieve the NNs of q in R incrementally, in increasing distance. Our incremental NN search algorithm extends the method of [30], which was originally proposed for point objects. In a nutshell the tiles of the grid are divided into groups based on their orientation w.r.t. tile T_q, which includes the query object q, as shown by the oblongs in Figure 5 (left). T_q forms a group by itself. The 8 neighboring tiles of T_q are grouped to 4 groups, each having 2 tiles; the next level of 16 tiles are grouped to 4 groups of 4 tiles each, and so on. In general, the nth level of tiles around T_q includes $8n$ tiles, which are split to four tile groups on the top, bottom, left, and right of T_q.

The incremental NN search algorithm (Algorithm 1) initializes a priority queue Q (minheap) and adds all objects in T_q to Q, which organizes them in increasing order of their *minimum* Euclidean distance to q. Obviously, objects that contain q (if any) have distance 0. The algorithm also adds to Q the 4 tile-groups G that are neighbors to T_q (level-1 tile-groups), using as key minimum distance between q and the extent of the tile-group (Function $NeighboringGroups$). It then runs a while-loop, which incrementally yields the NNs of q. At each iteration, the top entity c in the heap Q is de-heaped. If c is an object, then c is returned as the next NN of q (Line 16). If c is a group of tiles, then we add to Q (1) the neighboring group g of c at the next level ($NextLevelGroup(c)$) and (2) each tile $T_c \in c$ (Lines 7–10). The neighboring (i.e., next-level) group of c is the one having the same direction as c w.r.t. q (i.e., to the left, right, top, or bottom) and it is one level beyond c w.r.t. q.

Lastly (Lines 11–14), if c is a tile, for *some* of the object classes, all objects of these classes in c are en-heaped to Q. Figure 5 (right) shows how to define 9 types of the tiles depending on their relative direction w.r.t. the tile T_q which contains q. For each of these nine types of tiles, different classes are considered as shown in the legend of the figure (top). For example, if c is a tile of type 6, only objects of c in classes `0000`, `0001`, `1000`, `1001` are en-heaped; we encode this set of classes using the `X00X` notation, where “`X`” represents a bit equal to either `0` or `1`. The reason for this

ALGORITHM 1: Incremental NN Search

```
Input  : query point $q$, grid index $\mathcal{T}$ of rectangle set $R$
Output : next NN of $q$ in $R$
1  $T_q \leftarrow$ tile of $\mathcal{T}$ that contains $q$;
2  $G \leftarrow NeighboringGroups(T_q)$;
3  $Q \leftarrow$ new Min-Heap; add to $Q$ (1) all rectangles of $T_q$ and (2) all $g \in G$;
4  while $Q$ is not empty do
5      $c \leftarrow Q.pop()$;                    // $c$ is next nearest heap element to $q$
6      if $c$ is a group of tiles then
7          $g \leftarrow NextLevelGroup(c)$;
8          $Q.push(g)$;
9          foreach tile $T_c \in c$ do
10             $Q.push(T_c)$;
11     else if $c$ is a tile then
12         $rectArr$ = get rectangles in $c$ based on tile type;
13         foreach rectangle $r \in rectArr$ do
14             $Q.push(r)$;
15     else // $c$ is an object
16         yield $c$;                              // $c$ is the next NN
```

is that the objects in other classes are also included in neighboring tiles that are *closer* to q and therefore these objects would have been added to Q earlier. If we add them again to Q, then we could produce duplicate results in the NN set. The selection of classes for each tile c based on the relative direction of c w.r.t. T_q is established by the following theorem:

THEOREM 1. *For each dimension d, let BE be the pair of bits characterizing the rectangle classes in a tile c w.r.t. d. If tile c is before (after) T_q in dimension d, then all rectangle classes in c for which E =1 (B =1) should not be en-heaped to Q in Lines 12–14 of Algorithm 1, because they will produce duplicates.*

PROOF. If a class in tile c has E =1 (B =1), this means that all the rectangles in the class also appear in the tile c_{next} that follows (precedes) c in dimension d. However, since c is before (after) T_q in dimension d, c_{next} is guaranteed to be closer to q compared to c, meaning that the rectangles will be accessed before by the NN search when c_{next} is processed. Hence, including the rectangles of this class to Q would produce duplicate results in the NN search. □

Overall, the value of the object classification in each tile that we have proposed in Section 3 is twofold: (1) avoid the need to detect and eliminate duplicate results in NN queries, and (2) minimize the number of objects added to Q, as each object is guaranteed to be en-heaped at most once.

4.2 k-NN Search

(k-NN query) The k-NN is an extension of the NN query, where we include a parameter, which specifies the number of nearest neighbors to be retrieved.

Knowing the number k of desired neighbors allows us to prune objects and tiles, and to control the size of the priority queue Q. This is done by keeping track in a maxheap H the set of the k-NN objects so far. The top $H.top$ of H holds the object with the kth distance to q found so far. Distance $H.top.dist$ between $H.top$ and q can be used as a bound for eliminating tiles and tile-groups. Hence, we do not add to Q any objects, but only tiles and tile-groups which have minimum

ALGORITHM 2: k-NN Search

```
Input  : query point $q$, grid index $\mathcal{T}$ of rectangle set $R$
Output : $k$-NNs of $q$ in $R$
1  $T_q \leftarrow$ tile of $\mathcal{T}$ that contains $q$;
2  $H \leftarrow$ create $k$-maxheap and push to $H$ all rectangles of $T_q$;
3  $G \leftarrow NeighboringGroups(T_q)$;
4  foreach tile-group $g \in G$ do
5      if $|H| < k$ or $dist(g, q) < H.top.dist$ then
6          $Q.push(g)$;
7  while $Q.nonempty()$ and ($|H| < k$ or $Q.top.dist < H.top.dist$) do
8      $c \leftarrow Q.pop()$;
9      if $c$ is a group of tiles then
10         $g \leftarrow NextLevelGroup(c)$;
11         if $|H| < k$ or $dist(g, q) < H.top.dist$ then
12             $Q.push(g)$;
13         foreach tile $T_c \in c$ do
14             if $|H| < k$ or $dist(t, q) < H.top.dist$ then
15                 $Q.push(T_c)$;
16     else if $c$ is a tile then
17         $rectArr$ = get rectangles in $c$ based on tile type;
18         foreach rectangle $r \in rectArr$ do
19             if $|H| < k$ or $dist(r, q) < H.top.dist$ then
20                 $H.push(r)$;
21 return $H$;
```

distance to q smaller than or equal to $H.top.dist$. This greatly reduces the size of Q and may help to compute the k-NN result faster than the approach of applying the incremental NN algorithm until k objects have been returned. Algorithm 2 is a pseudo-code of our k-NN search procedure, which uses two heaps. Note that we start pruning groups/tiles as soon as the size $|H|$ of H reaches k.

4.2.1 *Avoiding Comparisons in k-NN Search.*

The performance of our k-NN search algorithm can be further improved, by taking advantage of existing statistics that we keep in the grid $\mathcal{T}$. Specifically, for a certain set of tiles, we can guarantee that all objects in them are k-NN results (considering only the relevant classes for each such tile, according to Theorem 1 and as shown in Figure 5). Hence, we do not need to compute distances between q and these objects and en-heap them into H.

For this, we first compute the *level* λ of tiles around tile T_q (which contains the query location q) up to which it is necessary to search until we can guarantee the computation of the k-NN set (T_q is at level 0). Starting from T_q, we count the number of objects in each tile at the levels around T_q, in the order examined by the k-NN algorithm. At each such level, we compute the number of objects, excluding classes that could produce duplicates, based on the direction of the tiles and according to Theorem 1. We continue counting until the number of rectangles counted including the currently examined level λ becomes at least k.

Given λ, we apply Theorem 2 to determine the level μ that guarantees all the rectangles in all tiles up to that level (inclusive) are certain query results.

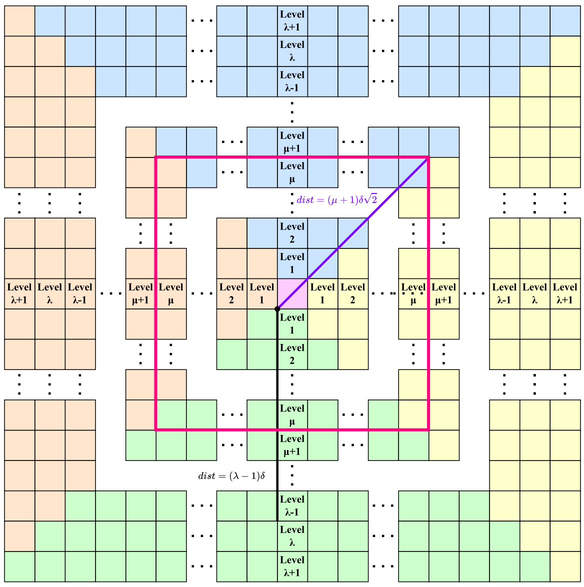


Fig. 6. k-NN search based on μ-optimization.

Theorem 2. *All rectangles, considering only the relevant classes of the tiles w.r.t. to T_q shown in Figure 5, up to level $\mu = \lfloor(\lambda-1)\sqrt{2}/2-1\rfloor$ are certain k-NN results.*

Proof. Assume that the smallest number of levels around T_q which include at least k rectangles is λ, as illustrated in Figure 6. For each rectangle r_i in any partition at level λ, considering only the classes specified by Theorem 1, we have:

$$(\lambda-1)\delta \leq dist(r_i, q), \tag{1}$$

where δ is the width/height of a tile in grid $\mathcal{T}$.

Now, consider the tile $T \in \mathcal{T}$ at a level $\mu < \lambda$, which is the furthermost to q (in Figure 7, it is the rightmost tile from the top group at the μ-th level). For this tile, we have $maxdist(T,q) = (\mu+1)\delta\sqrt{2}$, hence for each rectangle r_j at level μ, we have:

$$dist(r_j, q) \leq (\mu+1)\delta\sqrt{2} \tag{2}$$

Based on Equations (1) and (2), all rectangles of tile T are definitely in the k-NN result if:

$$(\mu+1)\delta\sqrt{2} \leq (\lambda-1)\delta \Rightarrow \mu \leq \frac{\sqrt{2}}{2}(\lambda-1)-1$$

Hence, the maximum value of μ for which all rectangles in tiles up to level μ are definite k-NN results and no distance computations and comparisons are needed for them is $\lfloor\frac{\sqrt{2}}{2}(\lambda-1)-1\rfloor$. □

Assume that the number of sure results at all levels up to μ is k'. After reporting all these results, we retrieve all rectangles (just the relevant classes) from levels $\mu+1$ to λ, compute their distances to q and keep track of the $(k-k')$ nearest ones in heap H. Unlike Algorithm 2, we avoid the use and maintenance of the priority queue Q up to level λ. Finally, we initialize Heap Q with the tile-groups of level $\lambda+1$ and follow the process described in Algorithm 2, for $k-k'$.

Algorithm 3 shows the steps of the k-NN algorithm based on the computation of levels λ and μ. Figure 7 illustrates an example of a 20-NN search. We begin from the tile containing the query

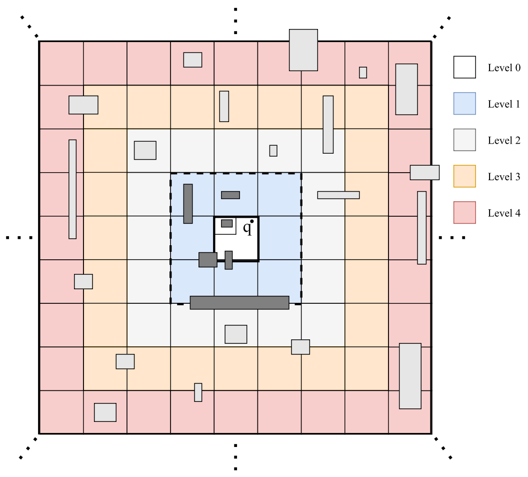


Fig. 7. 20-NN search based on μ-optimization.

ALGORITHM 3: Enhanced k-NN Search

```
Input  : query point $q$, grid index $\mathcal{T}$ of rectangle set $R$
Output : $k$-NNs of $q$ in $R$
1  $\lambda \leftarrow ComputeLambda(q, \mathcal{T})$;
2  $\mu \leftarrow \lfloor(\lambda - 1)\sqrt{2}/2 - 1\rfloor$;
3  $Res \leftarrow$ all rectangles in tiles up to level $\mu$;
4  $k \leftarrow k - |Res|$;
5  $T_q \leftarrow$ tile of $\mathcal{T}$ that contains $q$;
6  $H \leftarrow$ create $k$-maxheap and push to $H$ all rectangles at levels $\mu + 1$ to $\lambda$;
7  $G \leftarrow NeighboringGroups(T_q, \lambda)$;                // groups at level $\lambda + 1$
8  $Q \leftarrow$ new Min-Heap; add to $Q$ all $g \in G$;
9  Lines 7–20 of Algorithm 2;
10 return $Res \cup H$;
```

point q and expand up to level $\lambda = 4$, which is the smallest level that includes at least 20 objects (specifically, 26). According to Theorem 2, the $k' = 6$ objects up to level $\mu = 1$ (dark gray rectangles in the dashed square) can be promptly reported as results without requiring distance verification. Then, k is updated to $k - k' = 14$. For objects belonging to levels 2 to 4, their distances to point q are computed and added to heap H. Additionally, the tile groups at level $\lambda + 1 = 5$ are inserted into the priority queue Q, as it is possible that some of the k-NNs are in tiles at a level beyond λ. From thereon, the algorithm proceeds like Algorithm 2 to compute the 14-NN set beyond level $\mu = 1$.

Theorem 2 is not directly applicable for non-uniform grids, as it computes μ algebraically based on λ. However, we can apply the same idea behind Theorem 2 on a non-uniform grid to identify cells that contain objects in the kNN set without distance computations. Specifically, in the case of non-uniform grids the level λ may not be the same in all directions around T_q, i.e., the tile that includes q. To compute the number of levels at each direction, which are guaranteed to include at least k objects, we expand one level at the time at the currently nearest direction (up, down, left, or right) to q, until the number of cells in the expansion area include at least k unique object MBRs. In the end, this will form an *outer rectangular area* that includes the kNN results. Let D be the smallest distance from q to any of the boundaries of this rectangular area. Then, the *inner rectangular area* with guaranteed kNN results (corresponding to level μ in Theorem 2) is computed by gradually expanding around T_q and including levels in each direction while the largest distance from q to any of the corners of the inner rectangular area is smaller than D.

4.3 ϵ-range Queries

Distance range queries, where the objective is to retrieve all objects having distance at most ϵ from a query point q, can easily be evaluated, given our class decomposition. Specifically, we directly report the rectangles in all tiles whose maximum distance is at most ϵ from q. These tiles include all those in the rectangle Φ that comprises of the tiles having maximum x-distance and y-distance from q at most $\epsilon/\sqrt{2}$. Hence, for all such tiles we directly report all rectangles in all tiles in Φ as results (considering only the relevant classes as dictated by Theorem 1 and shown in Figure 5). For all other tiles T intersecting the query range, (1) if $maxdist(T, q) \leq \epsilon$, we directly report all rectangles in T as results, (2) otherwise, for each $r \in T$, if $dist(r, q) \leq \epsilon$, we report r as result. Note that only the relevant rectangle classes are considered at each tile T, according to Theorem 1, as already discussed.

5 ϵ-distance Joins

In this section, we explore the efficient evaluation of ϵ-distance joins using our proposed secondary partitioning scheme. Recall that this problem takes as input two collections R and S of spatial objects and a distance bound ϵ and the objective is to find the object pairs $(r, s) \in R \times S$, such that $dist(r, s) \leq \epsilon$. As before, we focus on the filter step of the query, meaning that we are seeking for MBR pairs having minimum distance from each other at most ϵ.

We study the problem under different indexing assumptions for the input sets R and S. The first case to handle is when both input sets R, S are already indexed by the same grid. For this case, we show in Section 5.1 how to exploit our secondary partitioning scheme in order to perform the distance join efficiently and without production of duplicates. In a nutshell, when we examine the objects in two tiles to find pairs in them within distance ϵ, we consider only object classes in the tiles that would not produce duplicates, based on the relative orientation of the tiles.

In the case where no index exists for the two datasets, we partition R and S on-the-fly using a regular grid, where each tile has extent $\epsilon/2 \times \epsilon/2$, create secondary partitions at each tile, and then apply our approach. In the case where one of the two datasets is already indexed by a grid with secondary partitions, we choose to create an index for the second dataset using an identical space partitioning (i.e., the same grid lines).

Finally, in Section 5.2, we study the case where both datasets are indexed by non-identical grids. If the two grids have coinciding lines but the grids are of different granularity, we propose an online transformation technique, which maps the finer granularity to the coarser one before proceeding with the ϵ-distance join. This technique exploits the existing indices and avoids the expensive re-partitioning of the data. In the case of different and misaligned grids for the two datasets, we select one of the two input datasets and create a new temporary index with the same granularity as the grid of the other dataset.

Similar to Section 4, our join techniques can be applied to other SOP indices as well, but for illustration purposes, we assume regular $n \times n$ grids are used for the primary partitioning of the space.

5.1 Identical Grids

To simplify the notation for the rest of this subsection, we assume that both input sets are primarily indexed under the same grid $\mathcal{T}$. We first discuss the setting where the projection of the grid tiles

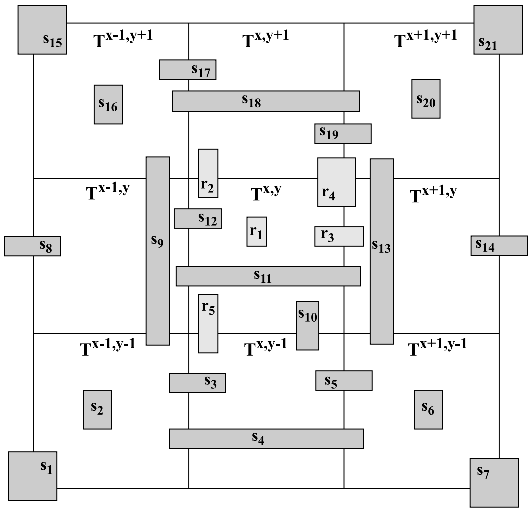


Fig. 8. Illustration of a distance join: joining $T^{x,y}$ tile to itself and the 8 surrounding tiles.

in each dimension is equal to or greater than the query distance threshold ϵ; later, we will discuss the case when these projections are lower than ϵ.

With grid $\mathcal{T}$ in place, inputs R, S are partitioned into sets of primary partitions $\mathcal{T}_R$ and $\mathcal{T}_S$, while each partition in these sets is further divided in 16 classes as proposed in Section 3. Now, to evaluate an ϵ-distance join efficiently, it suffices to traverse grid $\mathcal{T}$ and compute a series of partition-to-partition joins for each tile. Specifically, consider tile $T^{x,y}$. We perform a distance join between the partition $T_R^{x,y}$ from $\mathcal{T}_R$ and *nine* partitions in $\mathcal{T}_S$ (if they exist): $T_S^{x-1,y-1}$, $T_S^{x-1,y}$, $T_S^{x-1,y+1}$, $T_S^{x,y-1}$, $T_S^{x,y}$, $T_S^{x,y+1}$, $T_S^{x+1,y-1}$, $T_S^{x+1,y}$, $T_S^{x+1,y+1}$; these S-partitions correspond to tile $T^{x,y}$ and its eight surrounding tiles; we also refer to these tiles as joinable. As the projection of each grid tile is equal to or greater than the query threshold ϵ, the rest of the partitions in $\mathcal{T}_S$ contain object rectangles from S located farther than ϵ from any R object in partition $T_R^{x,y}$, by definition of the grid $\mathcal{T}$.

Figure 8 illustrates an example of a tile $T^{x,y}$ (in the center) which corresponds to a partition $T_R^{x,y}$ having five rectangles r_1 to r_5. The figure also shows the nine partitions from S which include all rectangles from S that will be joined with $T_R^{x,y}$. Simply joining $T_R^{x,y}$ to all nine partitions from S would produce duplicate results. For example, pair (r_1, s_{12}) will be reported in both $T_R^{x,y} \bowtie T_S^{x,y}$ and $T_R^{x,y} \bowtie T_S^{x-1,y}$, as s_{12} appears in both $T_S^{x,y}$ and $T_S^{x-1,y}$. We next elaborate on the spatial distance join between two partitions T_R and T_S, from $\mathcal{T}_R$ and $\mathcal{T}_S$ respectively, which *avoids* producing duplicate results using the rectangle classes by our secondary partitioning (Section 3).

The $T_R^{x,y} \bowtie T_S^{x,y}$ case. We first discuss the case where T_R and T_S correspond to the same tile T, i.e., $T_R = T_R^{x,y}$, $T_S = T_S^{x,y}$, and $T = T^{x,y}$. Since there are 16 classes in each of T_R and T_S, the total number of possible distance joins between the corresponding partition subdivisions is $16 \times 16 = 256$. Out of these *class-to-class joins*, we skip 112 joins, whereas from the remaining 144 joins, 95 of them reduce to cross products which simply enumerate join results without distance computations; the remaining 49 joins are evaluated by verifying the distances between objects, as in a typical distance join. Figure 9 shows which pairs of classes correspond to each case. The following lemmas determine the case whereto each possible class-class join belongs.

Lemma 1 (Pruned Class Pairs). *Assume that T_R and T_S correspond to the same tile T. For each pair of rectangle classes C_R from T_R and C_S from T_S for which there exists a dimension d such that for*

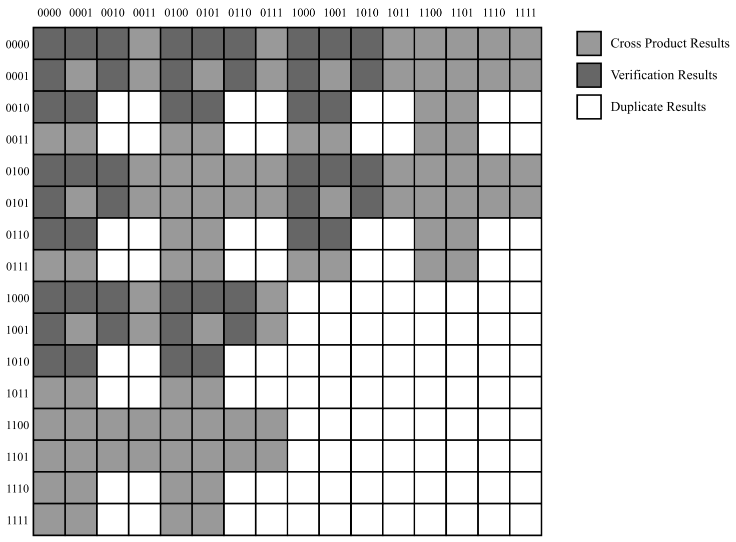


Fig. 9. Pairs of classes checked for $T_R^{x,y} \bowtie T_S^{x,y}$.

both C_R *and* C_S *the first bit is set to* 1 *in* d*, join pair* (C_R, C_S) *should be skipped, otherwise duplicate join results could be reported.*

Proof. If both C_R and C_S have their first bit set in dimension d, this means that *all* rectangles in C_R and C_S begin before T in dimension d. Therefore, any join result should be reported in the previous tile to T in dimension d. □

For example, in the 2D space consider class pair $C_R = \texttt{0010}$ and $C_S = \texttt{0010}$. In Figure 8, partitions $T_R^{x,y}$ and $T_S^{x,y}$ in the central tile include r_5 and s_{10} in classes C_R and C_S, respectively. Both classes have 1 at position 2, which indicates that all rectangles in C_R and all rectangles in C_S begin before tile $T^{x,y}$ in dimension y. So, all join pairs $(r_R, r_S), r_R \in C_R, r_S \in C_S$, such as (r_5, s_{10}) should not be reported by $T_R^{x,y} \bowtie T_S^{x,y}$ as they will be considered in $T_R^{x,y-1} \bowtie T_S^{x,y-1}$. Such class pairs are called *pruned class pairs.*

Lemma 2 (Cross-product Class Pairs (Case 1)). *Assume that* T_R *and* T_S *correspond to the same tile* T*. For each non-pruned pair of rectangle classes* C_R *from* T_R *and* C_S *from* T_S *for which there exists a dimension* d *such that for* both C_R *and* C_S *the two bits are* 01*, all* $(r_R, r_S), r_R \in C_R, r_S \in C_S$ *can immediately be reported as results without any comparisons.*

Proof. Consider two rectangles $r_R \in C_R$ and $r_S \in C_S$. In dimension d, both r_R end after T in dimension d, crossing the boundary of T. Hence the minimum distance between them should be their distance in the other dimension. As the distance in one dimension cannot exceed ϵ based on the definition of the grid $\mathcal{T}$, pair (r_R, r_S) is definitely a join result. □

As an example, consider again rectangles r_5 and s_{10} of Figure 8 in partitions $T_R^{x,y-1}$ and $T_S^{x,y-1}$, respectively, which belong to classes $C_R = \texttt{0001}$ and $C_S = \texttt{0001}$ in these partitions. When $T_R^{x,y-1} \bowtie T_S^{x,y-1}$ is computed, pair (r_5, s_{10}) is directly reported as a result, because the x-distance between the two rectangles is guaranteed not to exceed ϵ and their y-distance is guaranteed to be zero.

Lemma 3 (Cross-product Class Pairs (Case 2)). *Assume that* T_R *and* T_S *correspond to the same tile* T*. For each non-pruned pair of rectangle classes* C_R *from* T_R *and* C_S *from* T_S *for which there exists* *a dimension d such that for* one *of C_R and C_S the two bits are* 11*, all $(r_R, r_S), r_R \in C_R, r_S \in C_S$ can immediately be reported as results without any comparisons.*

Proof. Consider two rectangles $r_R \in C_R$ and $r_S \in C_S$. W.l.o.g. assume that the two bits of C_R in d are 11. This means that r_R starts before and end after tile T in d. Hence, the minimum distance between any $r_S \in C_S$ and r_R is their distance in the other dimension, which cannot exceed ϵ and (r_R, r_S) is a definite a join result. □

For example, in the central tile of Figure 8, consider class pair C_R = `0000` of $T_R^{x,y}$ containing rectangle r_1 and C_S = `1100` of $T_S^{x,y}$ containing rectangle s_{11}. All rectangles in C_S, such as s_{11} begin before and end after T in dimension x. For every rectangle $r_R \in C_R$, such as r_1, the distance between r_R and all rectangles in C_S cannot exceed their y-distance which is at most ϵ.

The $T_R^{x,y} \bowtie T_S^{x',y'}$ case. Now consider the join between two neighboring partitions T_R and T_S, which do not correspond to the same tile T. Once again, we avoid performing all 256 class-to-class joins as duplicate results could be produced by them. In a nutshell, we do not perform joins between classes whose rectangles extend beyond their tile (e.g., T_R) in the direction of the other tile (e.g., T_S) in *any* dimension. For example, in the join $T_R^{x,y} \bowtie T_S^{x-1,y}$ of Figure 8, we skip class pair C_R = `0000`, C_S = `0100`, because the rectangles in C_S (e.g., s_{12}) also appear in partition $T_S^{x,y}$ and if they join with any rectangle in $T_R^{x,y}$ the pair (e.g., (r_1, s_{12})) will be found during join $T_R^{x,y} \bowtie T_S^{x,y}$. The following lemma establishes which class pairs can be skipped when joining partitions corresponding to different tiles.

Lemma 4. *Assume that T_R and T_S do not correspond to the same tile T. For each pair of rectangle classes C_R from T_R and C_S from T_S for which there exists a dimension d such that there is a* 1 *bit in one of C_R, C_S in the direction of the tile where the other class belongs, join pair (C_R, C_S) should be skipped, otherwise duplicate join results could be reported.*

Proof. Consider two rectangles $r_R \in C_R$ and $r_S \in C_S$. W.l.o.g. assume that tile T_R of C_R is before tile of T_S of C_S in dimension d. If the first bit of C_S is 1, then r_S is also in tile T_S^{-1} *before* T_S in dimension d; T_S^{-1} is guaranteed to be also joined with T_R because it will have the same position as T_R in dimension d. Hence, if (r_R, r_S) is a join pair it will be reported in $T_R \bowtie T_S^{-1}$. Symmetrically, if T_R of C_R is after tile of T_S of C_S in dimension d, if (r_R, r_S) is a join pair it will be reported in $T_R^{-1} \bowtie T_S$. □

Based on Lemma 4, when joining two partitions T_R and T_S corresponding to different tiles, out of the 256 class-class joins, we conduct (1) 48 joins if the tiles are next to each other (e.g., $T_R^{x,y} \bowtie T_S^{x-1,y}$), and (2) just 16 joins if the tiles are diagonally to each other (e.g., $T_R^{x,y} \bowtie T_S^{x-1,y-1}$).

Summary and correctness. Figure 10 summarises the join process for identical grids, depicting the pairs of classes checked for the $T_R^{x,y}$ partition and its 9 joinable S-partitions in $\mathcal{T}_S$. The grid tile for $T_R^{x,y}$ is highlighted in dark gray and the 8 surrounding tiles by a red border, colored in light gray. Each subfigure corresponds to one of the 16 C_R classes of the $T_R^{x,y}$ partition. Moreover, for each partition T_S, we utilize the notation introduced in Figure 5 to show the sets of C_S classes to be examined.

We start with the $T_R^{x,y} \bowtie T_S^{x,y}$ case. The contents of the dark grayed tile for each subfigure in Figure 10 are determined according to Lemma 1:

(1) Figure 10(a), 10(b), 10(e), and 10(f) depict the sub-case when a C_R class has `0` in the first bit on both dimensions, i.e, the `0000`, `0001`, `0100` and `0101` classes, respectively. These $T_R^{x,y}$ classes are joined with all classes from $T_S^{x,y}$; we use the Type 4 notation from the table in Figure 5 to compactly denote all C_S classes.

⬟ Type 0	⬠ Type 1	● Type 2	◇ Type 3	○ Type 4	○ Type 5	◆ Type 6	□ Type 7	■ Type 8
`X0X0`	`XXX0`	`0XX0`	`X0XX`	`XXXX`	`0XXX`	`X00X`	`XX0X`	`0X0X`

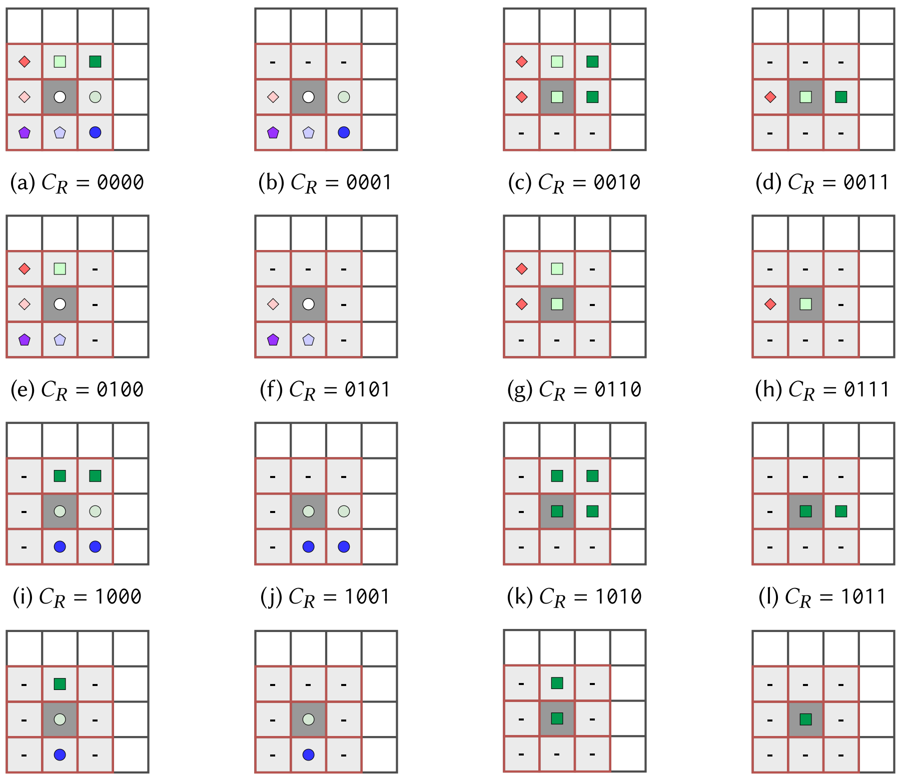

(a) C_R = `0000` (b) C_R = `0001` (c) C_R = `0010` (d) C_R = `0011`
(e) C_R = `0100` (f) C_R = `0101` (g) C_R = `0110` (h) C_R = `0111`
(i) C_R = `1000` (j) C_R = `1001` (k) C_R = `1010` (l) C_R = `1011`
(m) C_R = `1100` (n) C_R = `1101` (o) C_R = `1110` (p) C_R = `1111`

Fig. 10. All sets of classes checked for joining partition $T_R^{x,y}$ to partitions $T_S^{x,y}$ and $T_S^{x',y'}$ partitions from the 8 surrounding tiles.

(2) Figure 10(k), 10(l), 10(o), and 10(p) show when a C_R class has 1 in the first bit on both dimensions, i.e., the `1010`, `1011`, `1110` and `1111` classes, respectively. Such classes are joined only with the 4 classes `0000`, `0001`, `0100`, `0101` from $T_S^{x,y}$ that have a zero on these bits, i.e., denoted as the Type 8 classes from Figure 5.

(3) Figure 10(c), 10(d), 10(g), and 10(h) illustrate when a C_R class has 1 in the first bit on the y-dimension. Each of these classes are joined with a set of 8 classes with a `0` at the first bit in the y-dimension, i.e., denoted by Type 7 in Figure 5.

(4) Figure 10(i), 10(j), 10(m), 10(n) illustrate when a C_R class has 1 in the first bit on the x-dimension. Similar to the sub-case above, each of these classes is joined with a set of 8 C_S classes denoted by Type 5 in Figure 5.

We continue with the $T_R^{x,y} \bowtie T_S^{x',y'}$ case. We use Lemma 4 to determine which sets of classes in the 8 surrounding tiles (light gray, in red border) are joined with the each class of the $T_R^{x,y}$ partition. Again, we use the notation from Figure 5 to compactly summarise these sets of classes. Note that the S-partitions (and their classes) involved vary according to the current class in $T_R^{x,y}$, to avoid reporting duplicate results. For the C_R = `0000` in Figure 10(a), all 8 surrounding S-partitions are needed for the join, albeit a different set of classes (type) for each partition. In contrast, for C_R = `0001` in Figure 10(b), we ignore the partitions (and their classes) in the top row of the surrounding 8 S-partitions. Consider for example r_2 in Figure 8, which falls under the `0001` class in the $T_R^{x,y}$ partition. Assume for the sake of the example that $dist(r_2, s_{18}) \le \epsilon$ holds. The result pair (r_2, s_{18}) will be identified both when examining the $T_R^{x,y}$ partition and for $T_R^{x,y+1}$, where r_2 is classified in the `0010` class. Hence, to guarantee that (r_2, s_{18}) and all result pairs which include r_2, will be reported only once, we omit all three $T_R^{x,y} \bowtie T_S^{x-1,y+1}$, $T_R^{x,y} \bowtie T_S^{x,y+1}$ and $T_R^{x,y} \bowtie T_S^{x+1,y+1}$ joins.

Finally, we discuss the correctness of our method. To this end, we introduce the following theorem:

Theorem 3. *The ϵ-distance join algorithm that joins each tile $T_R \in \mathcal{T}_R$ with its (up to) nine neighboring tiles of $\mathcal{T}_S$, skips redundant class-join pairs and reduces class-join pairs, according to Lemmas 1–4 computes the correct ϵ-distance join result without producing any duplicate rectangle pairs.*

Proof. The correctness of the individual tile-tile joins follows directly from the proofs of Lemmas 1–4. No join results will be missed because any pair of rectangles (r_R, r_S), $r_R \in T_R, r_S \in T_S$ in non-neighboring tiles T_R and T_S having distance no more than ϵ, would expand into tiles T'_R and T'_S which are neighbors and will be identified by the respective $T'_R \bowtie T'_S$ join. □

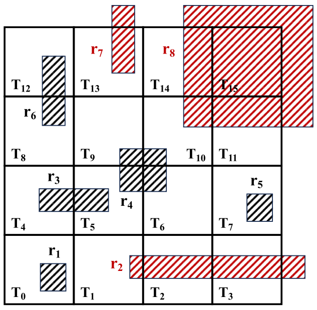
Class-to-class spatial distance join. We also elaborate on the computation of the ϵ-distance join between two classes C_R and C_S in two partitions $T_R^{x,y}$ and $T_S^{x',y'}$, respectively. If the two partitions correspond to the same tile (i.e., $x' = x$ and $y' = y$), then we expect most rectangle pairs to form query results. Hence, we verify the distances between all pairs $(r, s) \in C_R \times C_S$ and report the pairs having distance at most ϵ. If $x = x', y \neq y'$ then it must be $|y - y'| = 1$, i.e., the two tiles are one above the other. In this case, we apply an adaptation of the plane-sweep algorithm for intersection joins [3, 7], after y-sorting the contents of each of C_R and C_S; the adaptation identifies efficiently the pairs of objects having y-distance at most ϵ and verifies each such pair by computing their Euclidean distance. In all other cases, we apply the plane sweep adaptation after x-sorting and sweeping along the x-axis.

Extensions. We finally discuss potentially extensions or modifications to the original setting. Specifically, the above joining process can be also applied on grids of identical granularity, but where the extents of the tiles are lower than ϵ in some dimension. Intuitively, the key difference in this case is that we may have to consider additional levels around each tile of R and we may replace some joins by cross-products. Finally, our ideas are also directly applied on distance *self*-joins [38], where the objective is to find objects from the same dataset which are near each other.

5.2 Non-identical Grids

If the pre-existing grids on the input sets have different granularities, then our approach in Section 5.1 is not directly applicable. A straightforward solution is to re-index one of the input datasets, e.g. R, by creating a temporary secondary partitioning on top of a grid with a granularity that matches the one in S; a similar approach was employed in [45]. In this context, a key question that naturally arises is which input we should re-index. The rule of thumb is to select the input with either the smallest cardinality or the smallest average object extent, because such a decision is expected to incur the lowest online indexing cost. Our experiments in Section 6 highlight the importance of this decision; a poor selection of which input to re-index will drastically increase the total execution time.

Alternatively however, we can completely avoid re-indexing one of the inputs and the decision-making process (i.e., to determine which input to re-index) while still achieving a good



Rectangles	T_S	T_S Class	T'_S Class
r_1	T_0	0000	0000
r_2	T_1	0100	0100
r_3	T_4	0100	0000
r_4	T_5	0101	0000
r_5	T_7	0000	0000
r_6	T_8	0001	0000
r_7	T_{13}	0001	0001
r_8	T_{10}	0101	0101

Fig. 11. Online grid transformation: defining class `0000` for new partition T'_S.

performance for the $R \bowtie S$ distance join, by means of an online transformation. Assume that input R is primarily indexed by an $n \times n$ grid, resulting in the set of primary partitions $\mathcal{T}_R$, and S, by an $m \times m$ one, resulting in set $\mathcal{T}_S$, with $n < m$. We assume $m \mod n = 0$ holds; a simple workaround this is to define the grid granularity in a power of 2. Under this premise, we can define an online transformation of the finer grid to the coarser, by standard window range queries. Essentially, after overlaying the two grids, every partition T_R in $\mathcal{T}_R$ overlaps a collection of exactly $(m/n)^2$ adjacent partitions from $\mathcal{T}_S$. Consider for example Figure 12. The tiles defined by the bold lines represent the *coarser* 4×4 grid for the $\mathcal{T}_R$ partitions, while the thinner lines define tiles of the *finer* 16×16 grid for $\mathcal{T}_S$; for the moment, ignore the symbols for the sets of classes (or types) and the rest of the coloring. Observe how every T_R partition overlaps with exactly $(16/4)^2 = 16$ partitions from $\mathcal{T}_S$.

Now, to transform our secondary partitioning scheme of the $m \times m$ $\mathcal{T}_S$ grid onto a new grid $\mathcal{T}_S'$ that matches the granularity of the $n \times n$ $\mathcal{T}_R$ grid, it suffices to

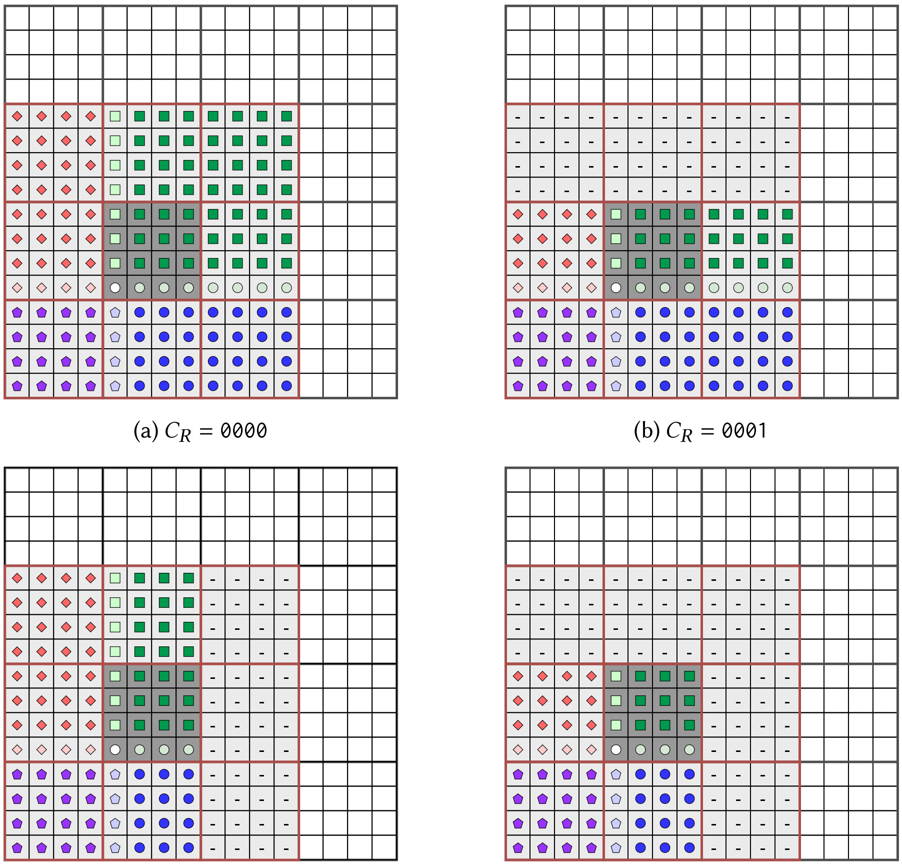
(1) evaluate a window range query for each partition $T'_S \in \mathcal{T}_S'$ to determine the overlapping $(m/n)^2$ partitions from $\mathcal{T}_S$, and
(2) define the contents of each T'_S class based on the class contents of these overlapping partitions.

While determining the overlapping partitions T_S is straightforward, defining the contents of all $(m/n)^2$ classes in T'_S is challenging.

To clarify the challenges, we elaborate on the formation of the `0000` class in a new T'_S partition. For each of the $(m/n)^2$ overlapping T_S partitions, only rectangles from the `0000`, `0001`, `0100`, `0101` classes should be considered in the formation of the class, i.e., those that start inside every overlapping T_S partition on both axes which by definition also start inside T'_S. On the one hand, the case of the `0000` rectangles in all these T_S partitions is straightforward; every rectangle that starts and ends inside the `0000` class of a T_S, also starts and ends inside the new T'_S partition, by definition. On the other hand though, this is not the case for the remaining `0001`, `0100`, `0101` classes in the overlapping T_S partitions, as their rectangles are not guaranteed to also end inside T'_S. We exemplify this issue in Figure 11. The tiles in the figure correspond to the 16 overlapping T_S partitions, $T_0 \dots T_{15}$, for a T'_S partition. The r_2, r_7 and r_8 rectangles all start inside the T'_S partition by definition, as they belong to the `0100` class in T_1, the `0001` in T_{13} and the `0101` in T_{10}, respectively. However, these rectangles cannot be part of the `0000` class of T'_S as they extend beyond its bounds.

Similar issues arise when determining the contents of the other 15 classes in T'_S as well. Essentially, a series of additional comparisons and checks are required for this purpose, which will affect the total execution time of the join. To avoid these checks, we devise an alternative approach

⬟ Type 0	⬠ Type 1	● Type 2	◇ Type 3	○ Type 4	○ Type 5	◆ Type 6	□ Type 7	■ Type 8
`X0X0`	`XXX0`	`0XX0`	`X0XX`	`XXXX`	`0XXX`	`X00X`	`XX0X`	`0X0X`


(a) C_R = `0000` (b) C_R = `0001`

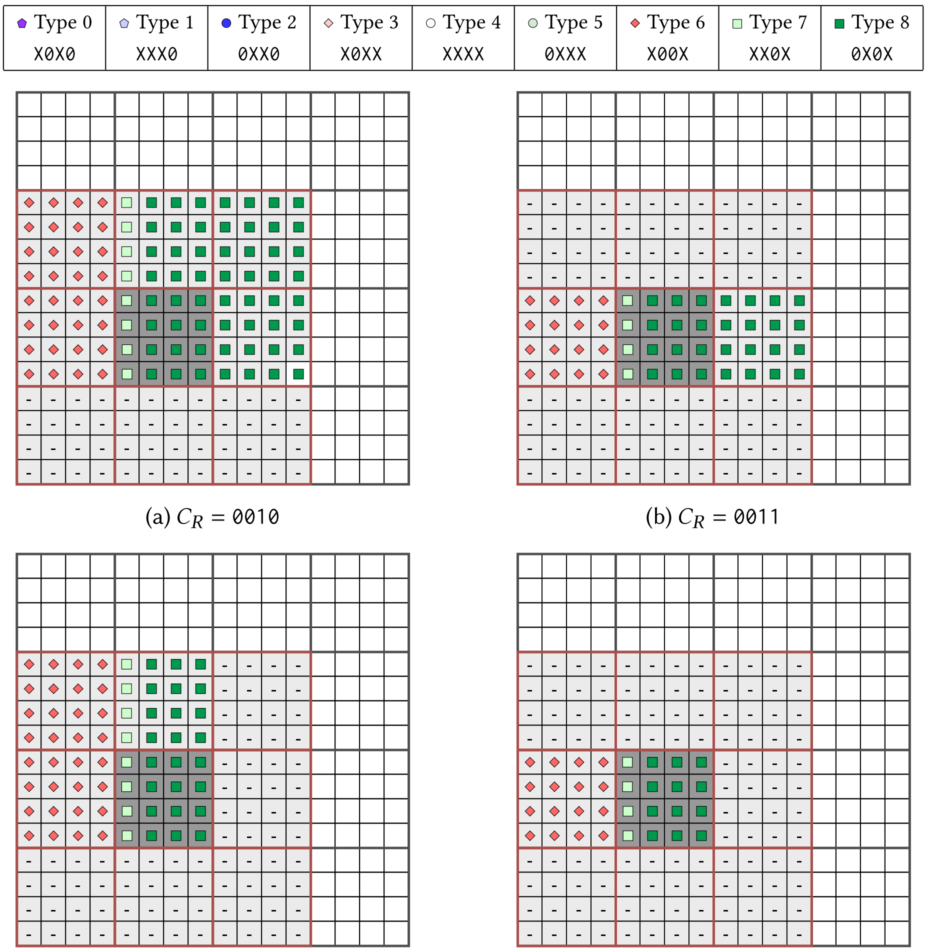
(c) C_R = `0100` (d) C_R = `0101`

Fig. 12. Online grid transformation: the case of $0x0x$ classes in T_R.

for the online transformation that does not require to determine the contents of each new T'_S partition. Intuitively, the key idea is to directly employ the join process described in Section 5.1 which utilizes the secondary partitionings on $\mathcal{T}_R$ and $\mathcal{T}_S'$ under an identical $n \times n$ grid. We adjust the process in Section 5.1 such that the pairs of classes checked for every $T_R \bowtie T'_S$ partition-to-partition distance join are re-defined based on the $(m/n)^2$ partitions from the original $m \times m$ $\mathcal{T}_S$ on input S that overlap each T'_S partition. We illustrate this approach for the online transformation in Figures 12–15.

Intuitively, Figures 12–15 extend their Figure 10 counterparts based on the principle of the online transformation:

— Figure 12(a), 12(b),12(c) and 12(d) extend Figure 10(a), 10(b), 10(e) and 10(f), respectively;
— Figure 13(a), 13(b), 13(c) and 13(d) extends Figure 10(c), 10(d), 10(g) and 10(h), respectively;
— Figure 14(a), 14(b), 14(c) and 14(d) extends Figure 10(i), 10(j), 10(m) and 10(n), respectively;
— Figure 15(a), 15(b), 15(c) and 15(d) extends Figure 10(k), 10(l), 10(o) and 10(p), respectively.



(a) C_R = `0010` (b) C_R = `0011`

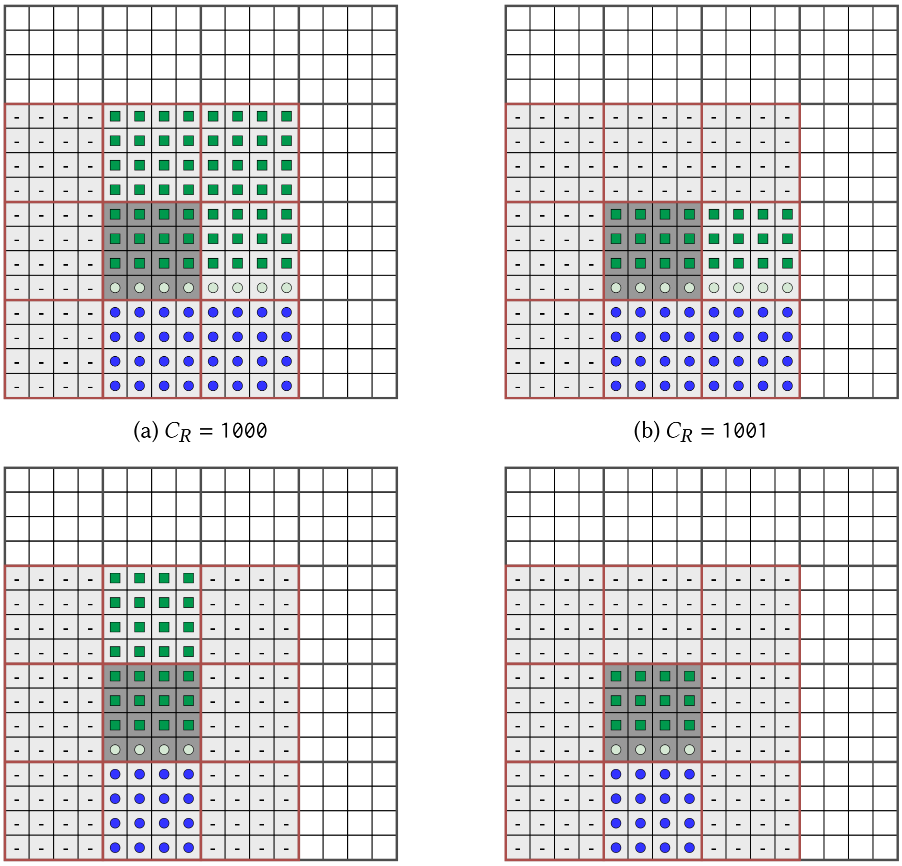
(c) C_R = `0110` (d) C_R = `0111`

Fig. 13. Online grid transformation: the case of $0x1x$ classes in T_R.

Every tile in Figure 10 is essentially overlayed on top of 16 tiles from the $\mathcal{T}_S$ grid in Figures 12–15. Again, we highlight the current partition T_R and its 16 overlayed T_S partitions in dark gray. Without loss of generality, we assume that the distance threshold ϵ is smaller than the projection of the 4×4 tiles in both dimensions, similar to Figure 10. Therefore, T_R is joined to the T'_S partition from the same tile and the S partitions from the 8 surrounding tiles, highlighted in red. For each class in T_R, the figures show which T_S partitions should be consider and specifically, which sets of classes in them; similar to Section 5.1 and Figure 10, we utilize the notation from Figure 5 to compactly mark these sets of classes.

To explain the join process, consider class `0000` for T_R and juxtapose Figure 12(a) to Figure 10(a). For the S partitions at the four corners of the joinable area (i.e., the 9 tiles highlighted in red), the join process is a straightforward extension of the process in Figure 10(a). Specifically, the `0000` class from T_R will be joined with Type 6 (`X00X`), Type 8 (`0X0X`), Type 2 (`0XX0`) and Type 0 (`X0X0`) classes from the overlapping T_S partitions, similar to Figure 10(a); we report the tiles in clockwise order starting from the top-left corner. On the other hand, for the S partitions overlapping T_R

⬟ Type 0	⬠ Type 1	● Type 2	◇ Type 3	○ Type 4	○ Type 5	◆ Type 6	□ Type 7	■ Type 8
X0X0	XXX0	0XX0	X0XX	XXXX	0XXX	X00X	XX0X	0X0X



(a) C_R = 1000 (b) C_R = 1001

(c) C_R = 1100 (d) C_R = 1101

Fig. 14. Online grid transformation: the case of $1x0x$ classes in T_R.

and the tiles over, below, to the left and to the right of T_R, we cannot use the same class types specified in Figure 10(a); such an approach will incur duplicates. In fact, the class types from Figure 10(a) are applicable only for the T_S partitions located in rows or columns at the borders of these tiles. Let us denote T_R partition as $T_R^{x,y}$. Type 3 (X0XX) classes are only used for the bottom row of the T_S partitions defining partition $T_S'^{x-1,y}$; Type 7 (XX0X) classes are only used for the left column of partitions for $T_S'^{x,y+1}$; Type 5 (0XXX) classes are used only for the bottom row in $T_S'^{x+1,y}$; and Type 1 (XXX0), only for the left column of the $T_S'^{x,y-1}$ partition. For the remaining T_S partitions which define the $T_S'^{x-1,y}$, $T_S'^{x,y+1}$, $T_S'^{x+1,y}$, $T_S'^{x,y-1}$ partitions, we need to restrict the number of classes to be joined with 0000 from $T_R^{x,y}$ (or simply T_R in our example) by specifying one extra bit in their binary encoding, again to avoid duplicate results. Finally, regarding the T_S partitions overlapping with T_R, only the partition for bottom-left corner tile of the S grid directly inherits the type of classes specified in Figure 10(a), i.e., Type 4 (XXXX); for the rest we need to restrict the classes to be join with 0000 from T_R either by specifying one or two extra bits. The rest of the cases in Figure 12 and Figures 13–15 are determined in a similar fashion,

⬟ Type 0	⬠ Type 1	● Type 2	◇ Type 3	○ Type 4	○ Type 5	◆ Type 6	□ Type 7	■ Type 8
`X0X0`	`XXX0`	`0XX0`	`X0XX`	`XXXX`	`0XXX`	`X00X`	`XX0X`	`0X0X`

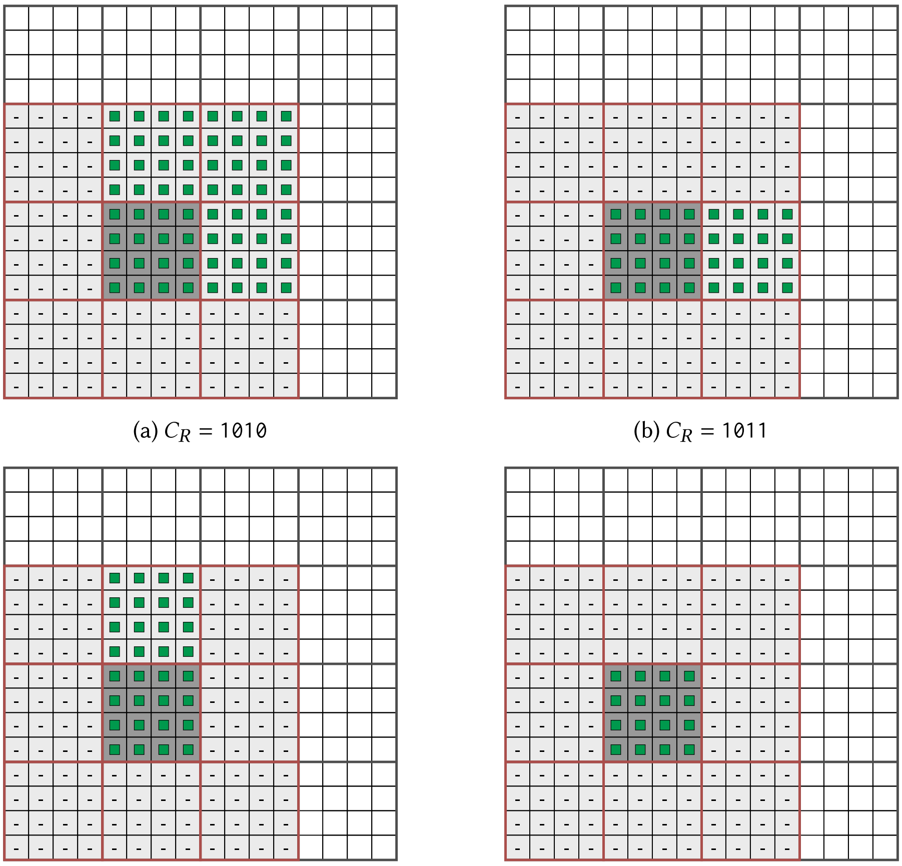

(a) C_R = `1010` (b) C_R = `1011`
(c) C_R = `1110` (d) C_R = `1111`

Fig. 15. Online grid transformation: the case of $1x1x$ classes in T_R.

appropriately adjusting the cases in Figure 10. Observe, for instance, how the first row of T'_S (and their overlapping T_S) partitions are ignored in Figure 12(b) for class `0001` in T_R, similar to Figure 10(b).

6 Experimental Evaluation

We conducted our analysis on a dual Intel(R) Xeon(R) CPU E5-2630 v4, clocked at 2.20 GHz with 384 GBs of RAM, AlmaLinux 8.5. All indices were implemented in C++, compiled using `gcc` (v8.5.0) with flags `-O3`, `-mavx`, and `-march=native`.

6.1 Setup

We experimented with the ZCTA5, RAILS, ROADS and EDGES Tiger 2015[1] from [14]; Table 2 summarizes their characteristics. The objects in each dataset were normalized so that the coordinates

[1]http://spatialhadoop.cs.umn.edu/datasets.html

Table 2. Datasets Used in the Experiments

TIGER 2015 dataset [14]	**type**	**cardinality**	**size**	**avg. relative [%]**	
				x-extent	**y-extent**
ZCTA5	polygons	33 K	1.1 MB	1.7	2.052
RAILS	linetrings	157 K	5.3 MB	0.012	0.05
ROADS	linetrings	17 M	564 MB	0.004	0.015
EDGES	polygons	51 M	1.7 GB	0.002	0.007

in each dimension take values inside $[0, 1]$. The last two columns of Table 2 are the relative (over the entire space) average length for every object's MBR at each axis.

We implemented our secondary partitioning scheme as part of a main-memory regular grid spatial index; we denote our scheme as 2-layer16. Although our approach can also be applied on other SOP indices, such as the PMR quadtree [21], we do not include such an approach in the comparison, as quadtrees were shown inferior to grids with and without two-layer partitioning in [42]. As its main competitor, we considered the state-of-the-art DOP index: an in-memory STR-bulkloaded [24] R-tree, from the highly optimized Boost Geometry library (boost.org).[2, 3] The inner and leaf nodes of the constructed trees have a capacity 16; this configuration is reported to perform the best (we also confirmed this by testing). For ϵ-range queries, we also compared to the secondary partitioning scheme from [42], which divides each partition into 4 classes (see Section 2.2); we denote this scheme as 2-layer4.

To study the computation of NN and ϵ-range queries, we measure the average execution time of 10 K queries over 10 runs. For NN queries, we vary the number of returned neighbors; in incremental NN search, the neighbors are progressively determined as described in Section 4.1, while in k-NN search, we set k accordingly. For the ϵ-range queries and the ϵ-distance joins, we vary ϵ as a percentage of the entire data space, inside the $\{0.01\%, 0.05\%, 0.1\%, 0.5\%, 1\%\}$ value range.

6.2 Index Tuning

We first investigate the best granularity for a grid that uses our partitioning scheme. For this purpose, we considered only the incremental NN search, for browsing the first 100, 1,000, and 10,000 neighbors. Figure 16 reports the total time for 10 K queries for different grid granularities, i.e., number of partitions per dimension. In all cases, the search initially accelerates as the grid becomes finer but slows down when the grid becomes too detailed. Based on the plots, we set the number of partitions per dimension to 160 for ZCTA5, 400 for RAILS, 3,000 for ROADS and to 4,000 for EDGES, where the best performance is observed.

6.3 Query Processing

We now turn our focus to processing distance queries.

6.3.1 NN Search. Figures 17 and 18 compare the throughput of our 2-layer16 partitioning against the R-tree, for incremental NN and k-NN search. At each experimental instance, for each of the two compared methods, we report the average, minimum, and maximum performance (throughput) for the tested query workload (shown as error bar). For the latter, 2-layer16 is equipped with the μ-optimization described in Section 4.2.1.

[2]Recent benchmarks [26] showed the superiority of Boost.Geometry R-tree implementations over the ones in libspatialindex.org.

[3]The analysis in [42] showed that R-tree is the most efficient DOP competitor, outperforming R*-tree (from the same library).

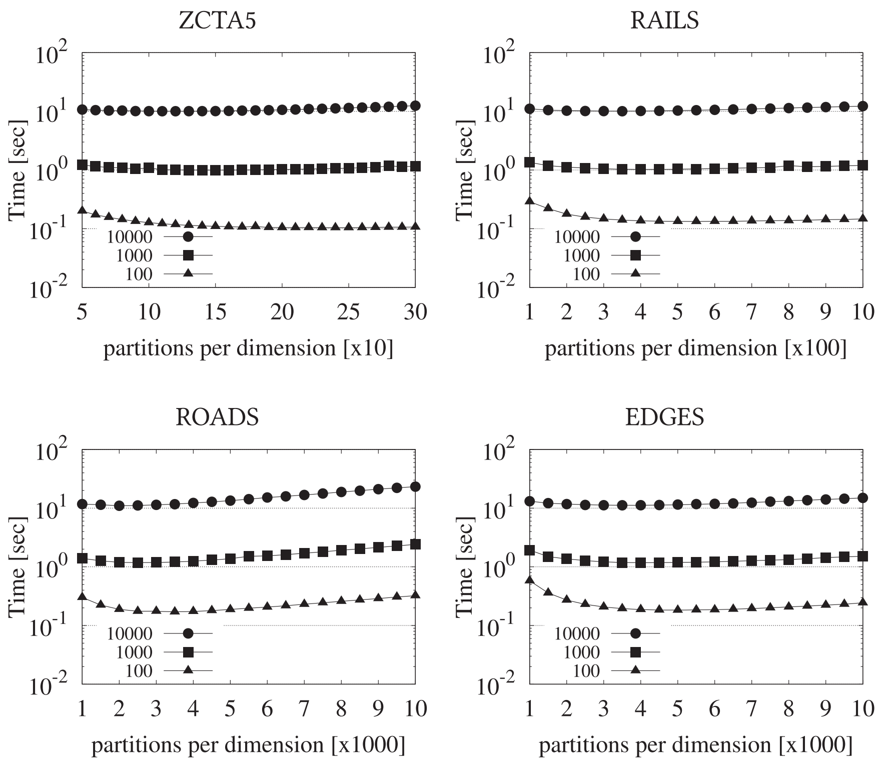


Fig. 16. Determining best grid granularity: incremental NN search for 100, 1,000 and 10,000 browsing neighbors.

We observe that for both types of NN search, 2-layer16 steadily outperforms the R-tree; the latter is competitive only when a small number of neighbors are browsed or requested on EDGES. Notice that the performance gap also increases by the number of browsed neighbors up to one order of magnitude, rendering 2-layer16 significantly more efficient for complex query plans which require pipelining a large number of NNs to the next operator. As another observation, the k-NN 2-layer16 algorithm is faster than the incremental NN one. As we discussed in Section 4.2, knowing the number of neighbors in advance allows the k-NN algorithm to prune objects and tiles, and even returning a portion of the results without conducting any comparisons, using the μ-optimization from Section 4.2.1. We also observe that for for small values of k 2-layer16 has higher variability in its performance compared to the R-tree, while for large values of k the opposite holds. To interpret this, we need to examine how the algorithms operate in both best and worst scenarios. The worst scenarios for 2-layer16 occur when the query belongs to a neighborhood with very sparse data (where the tile has no nearby tiles with spatial objects) or to a tile with dense data and a small k. In both scenarios, 2-layer16 method performs poorly: in the first case, it needs to expand the grid space to find results, and in the second, it needs to calculate all distances from the spatial objects within the tile. The worst-case scenario for an R-tree occurs when the value of k is large, as it requires visiting numerous nodes to obtain the desired result. The best-case scenario for 2-layer16 occurs when there are at least k objects within the query's tile or its immediate neighbors. Meanwhile, the optimal scenario for an R-tree is when k is smaller than the size of its leaf nodes, allowing it to find the result immediately with just one traversal. The above justify the behavior of the two algorithms with respect to their best-case and worst-case performance.

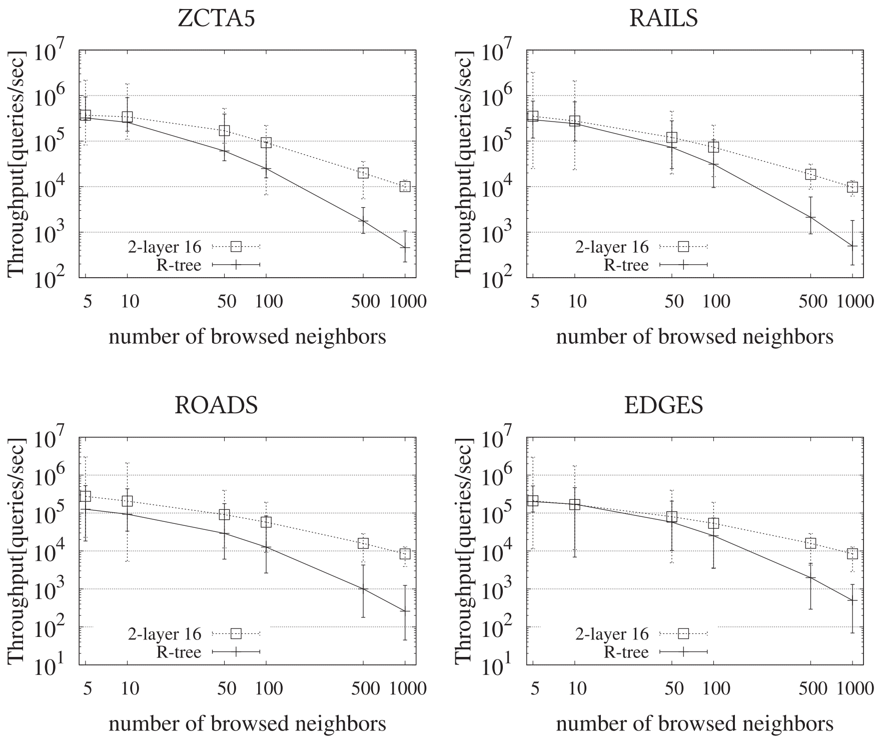


Fig. 17. Incremental NN search.

Finally, we also investigate the scalability of the indexing methods. We considered for this purpose the incremental NN search; we observed similar findings for k-NN. Figure 19 reports the results on our largest datasets ROADS and EDGES from Table 2; we ran our experiments over random samples comprising 20%, 40%, 60%, 80%, and 100% of the objects. The plots demonstrate that both methods exhibit good scalability when the dataset size varies, but 2-layer16 outperforms the R-tree in all cases.

6.3.2 ϵ-range Queries. Next, we study the computation of ϵ-range queries. We compare our 2-layer16 partitioning against the secondary partitioning from [42] and the R-tree. Figure 20 reports the throughput of each query processing method while varying the ϵ parameter. Our analysis confirms the results of [42], i.e., 2-layer4, always outperforms the R-tree. Most importantly though, 2-layer16 is the fastest method, steadily outperforming 2-layer4. This finding clearly shows the benefit of the new class decomposition into 16 classes, which, (1) as opposed to [42], facilitates NN and distance join queries and (2) is superior to [42] for ϵ-range queries.

6.3.3 ϵ-distance Joins. We lastly investigate the computation of distance joins. For this purpose, we consider two alternative settings; when neither of the R, S inputs is indexed and when both are.

Under the first setting, the computation of an $R \bowtie S$ distance join becomes a two-step process. Initially, the inputs are temporarily partitioned by two identical grids on top of which our secondary partitioning scheme is built, and then, the process described in Section 5.1 is directly applied. We experiment with four pairs of inputs to define our join queries; ZCTA5 $\bowtie$ ROADS, ZCTA5 $\bowtie$ EDGES, RAILS $\bowtie$ ROADS and RAILS $\bowtie$ EDGES. Figure 21 reports the total processing

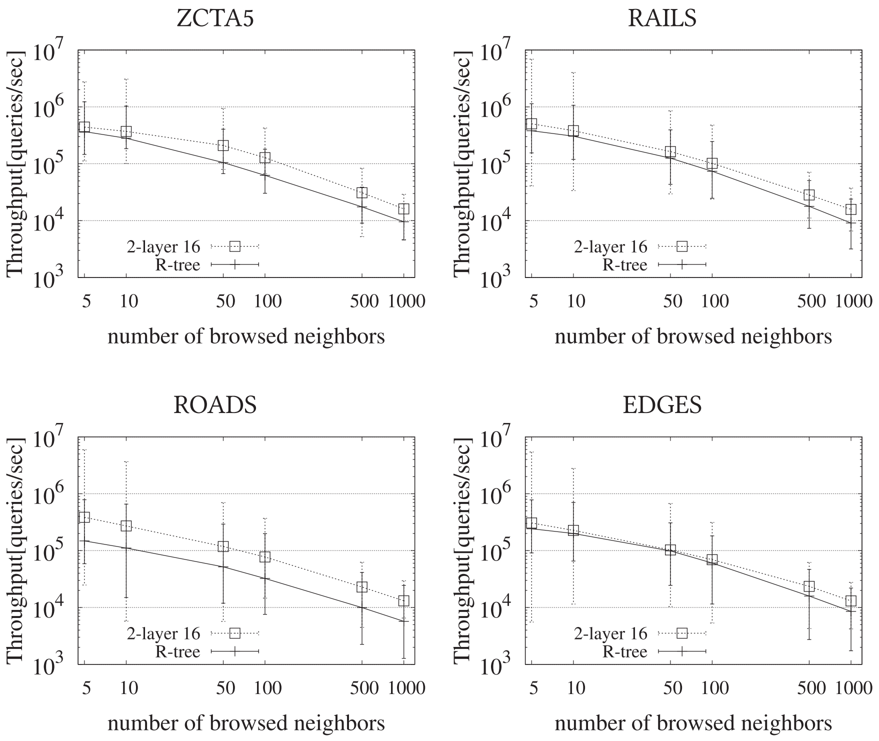


Fig. 18. k-NN search.

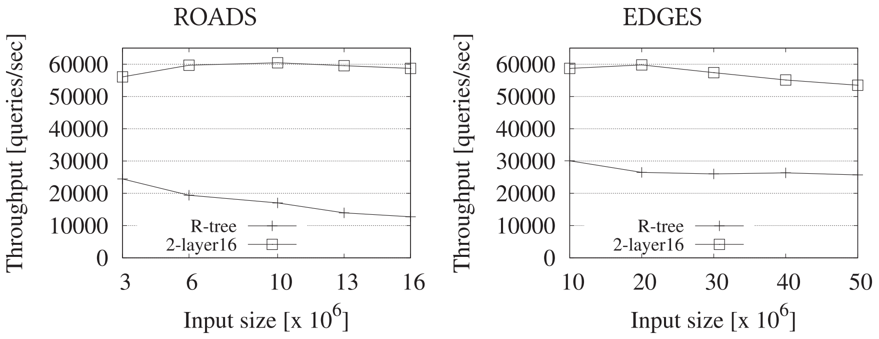


Fig. 19. Incremental NN search; scalability experiment.

time for different values of the ϵ threshold, while varying the granularity of the online grid; at the top of every plot, we also report the total indexing cost. Naturally in all tests, this total indexing cost always rises with the increase of the grid granularity due to the higher replication factor. On the other hand, all joins initially benefit from utilizing a finer grid and therefore, the joining time initially drops, but later on the joining process becomes increasingly more expensive, due to the large number of partition-to-partition and class-to-class joins defined in grids of higher granularity. The optimal grid granularity, i.e., the optimal number of partitions per dimension, is dictated by the input whose indexing cost is more likely to be drastically affected by increasing the grid granularity. In case of RAILS $\bowtie$ ROADS and RAILS $\bowtie$ EDGES, building our two-layer partitioning

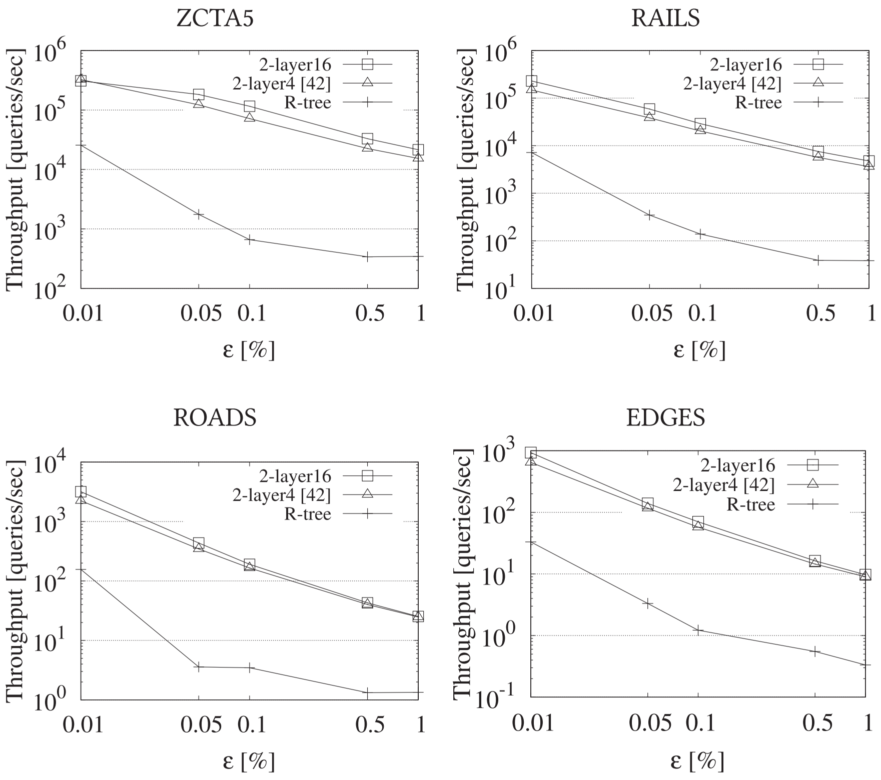


Fig. 20. ϵ-range queries.

scheme takes longer on ROADS and EDGES compared to RAILS. Therefore, the best performance is observed roughly for a 2000 × 2000 grid, which is very close to the optimal grid granularity used individually in ROADS and EDGES for NN search and ϵ-range queries, see Figure 16. In contrast, for ZCTA5 ⋈ ROADS and ZCTA5 ⋈ EDGES, the indexing of ZCTA5 is severely affected by the increase of the grid granularity because of replicating its large rectangles to increasingly more grid tiles. Observe how the total query processing is completely dominated by the indexing costs in the top row of Figure 21 after the 500 × 500 grid. In fact, the best performance is observed for this 500 × 500 grid, which is similar to the 160 × 160 optimal grid applied in ZCTA5 for NN search and ϵ-range queries.

We next study the second setting, when both R, S inputs are indexed in advance for answering frequent distance queries such as NN and ϵ-range. We distinguish between two cases based on the existing indexing; either an R-tree exists on each input dataset, which is the most dominant DOP index, or a two-layer partitioning scheme. For the former, we modify the classic R-tree overlap join algorithm in [7] to compute ϵ-distance joins. For the latter, we assume that the granularity of the underlying grid is set to achieve the best performance for NN or ϵ-range queries and employ our methods proposed in Section 5. We experimented again with the ZCTA5 ⋈ ROADS, ZCTA5 ⋈ EDGES, RAILS ⋈ ROADS, and RAILS ⋈ EDGES distance joins. In all four queries, the underlying grids for the inputs are built on different granularities according to Figure 16. Therefore, we consider three approaches to compute an $R \bowtie S$ distance join:

(1) build a temporary two-layer scheme for R on top of a grid that matches the granularity of S and then apply the join process discussed in Section 5.1 for identical grids;

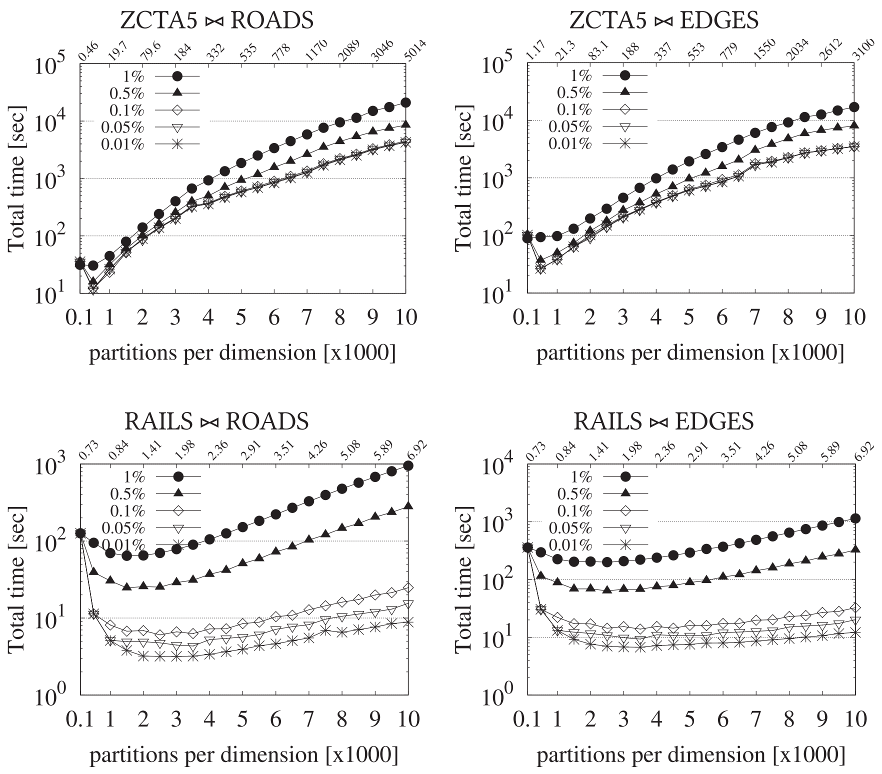


Fig. 21. ϵ-distance joins: the case of unindexed inputs; plot lines show the total querying time and number at the top, the total indexing cost.

(2) build a temporary two-layer scheme for S on top of a grid that matches the granularity of R and then apply the join process discussed in Section 5.1 for identical grids;
(3) use the online transformation described in Section 5.2.

For the third approach, we slightly modify the granularity of ROADS grid to be 3200 instead of 3000.

Figure 22 reports the total time of each approach on the four join queries, while varying the value of the ϵ distance threshold. In addition, Table 3 reports the online indexing costs that occur when a temporary two-layer scheme is built. With respect to the existing indexing, the first observation is that R-trees are either the least or second to last efficient indices for the distance join queries. In combination to our tests on NN search and ϵ-range queries, we can overall discard R-tree as an efficient indexing for distance queries. Regarding our two-layer partitioning, the experiments back up our arguments from Section 5.2. Specifically, they showcase how critical is the decision which input should be re-indexed, in other words for which input a temporary two-layer scheme should be built. When the correct decision is made, i.e., re-indexing ROADS, EDGES for ZCTA5 ⋈ ROADS, ZCTA5 ⋈ EDGES, and RAILS ⋈ ROADS, RAILS ⋈ EDGES, our join algorithm on top of the secondary partitioning achieves the best performance. However, if a poor decision is made, the performance of the secondary partitioning can be very similar to that of R-tree join even orders of magnitude worst, as she we observe in the case of re-indexing ZCTA5. Due to its large rectangles, the online indexing costs of ZCTA5 to a grid that matches the granularity of ROADS or EDGES are extremely high as the replication fact drastically increases (see Table 3); note that

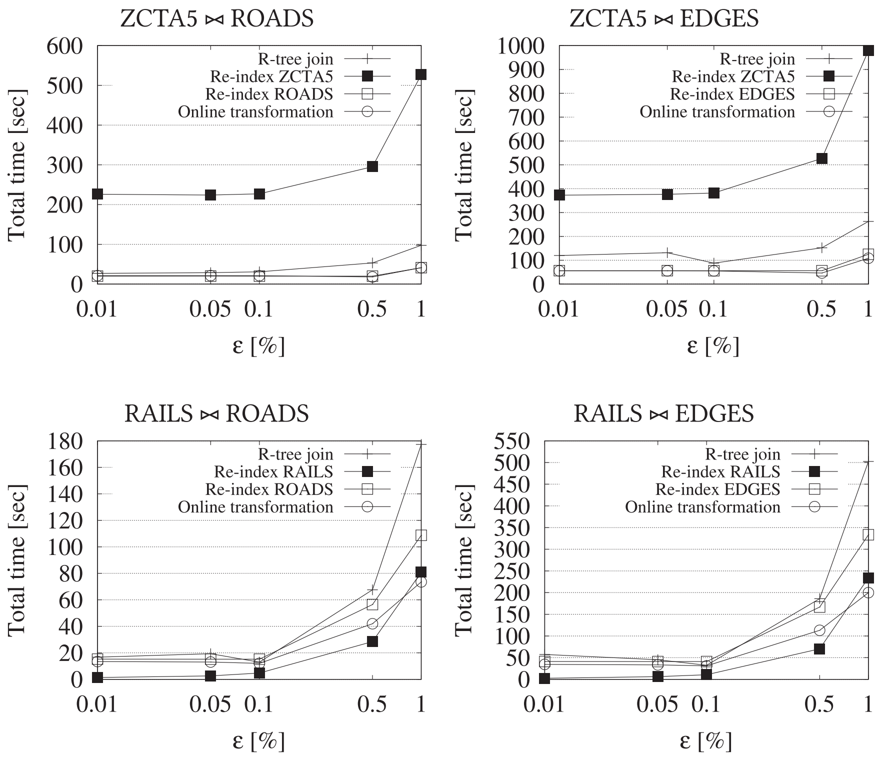


Fig. 22. ϵ-distance joins: the case of indexed inputs.

Table 3. ϵ-distance Joins: Online Indexing Costs for the Case of Indexed Inputs [Secs]

query $R \bowtie S$	re-index R	re-index S
ZCTA5 ⋈ ROADS	215	0.41
ZCTA5 ⋈ EDGES	337	1.27
RAILS ⋈ ROADS	0.23	0.41
RAILS ⋈ EDGES	0.35	1.36

the best grid granularity for ZCTA5 is 160, while for ROADS and EDGES is 3,000 and 4,000, respectively.

On the other hand, the plots also show that we can always ensure fast join computation using the online transformation. The performance of the transformation matches the performance of the re-index approach with the best decision or even exceeds it in case of high values of ϵ. An additional advantage of the online transformation is in terms of space; no additional space is required as none of the inputs are re-indexed.

6.4 Index Building, Size, and Maintenance

Figure 23 reports on the index building costs for the tested datasets. We compare the indexing time of our proposed partitioning scheme 2-layer16 against the 2-layer4 scheme from [42]. For reference, we also include the bulk-loading cost for the R-tree. As expected, due to decomposing every partition into 16 classes instead of 4, the new 2-layer16 scheme exhibits a higher indexing time. Nev-

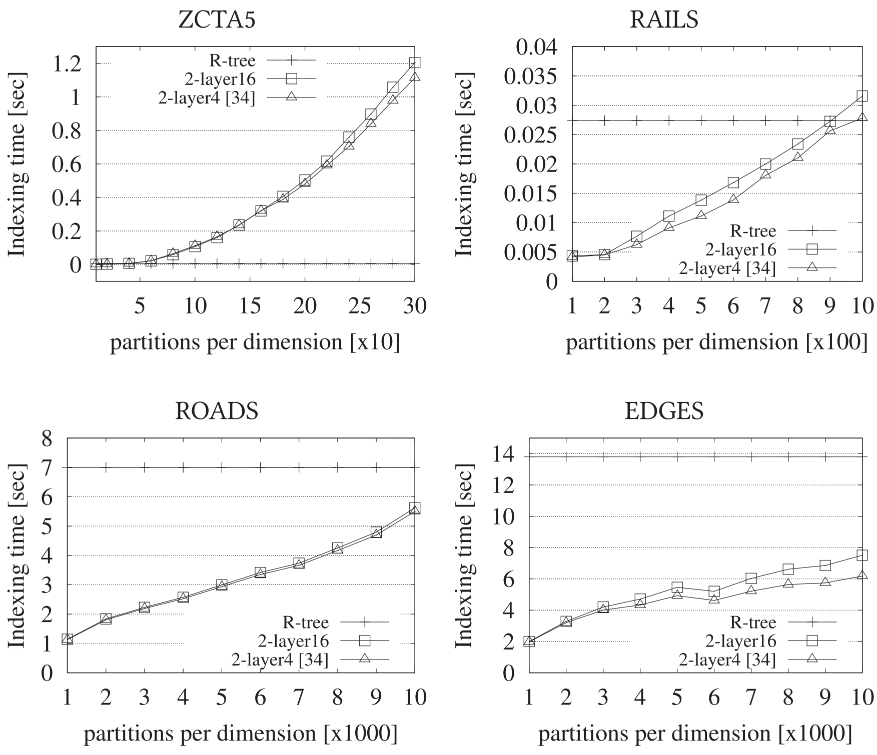


Fig. 23. Indexing cost and tuning.

ertheless, the building cost of 2-layer16 is significantly lower than that of the R-tree in most cases with the exception of ZCTA5, which contains very few rectangles. Note that the index size of the two 2-layer schemes is identical as the total number of entries inside the classes remains the same.

Figure 24 presents the memory space required by each index. As observed, the R-tree requires less memory because it does not replicate the data. Both 2-layer16 and 2-layer4 exhibit the same memory usage, as they replicate the same amount of data. The memory usage increases in the 2-layer16 versions as the grid size increases, which is expected because larger grids entail more data replication. Despite this, grids are preferred to R-trees due to their superior query performance as shown in the previous experiments for distance queries and in [42] for simple range queries.

Finally, regarding index maintenance, we expect a similar trend to index building. The analysis in [42] showed that 2-layer4 has much faster updates than the R-tree. We expect also 2-layer16 to outperform the R-tree for updates but exhibit slightly higher time compared to 2-layer4, due to maintaining more classes.

7 Conclusions

We presented a secondary partitioning technique for space-oriented partitioning indices, which divides the contents of each partition into 16 classes based on the begin and end points of the object MBRs with respect to the partition boundaries. We proposed algorithms for distance queries (NN, distance range, and distance joins) that take advantage of our technique to compute query results efficiently and without producing duplicates. Our evaluation using real datasets confirms the superiority of our approach compared to previous work. Note that our approach can be generalized

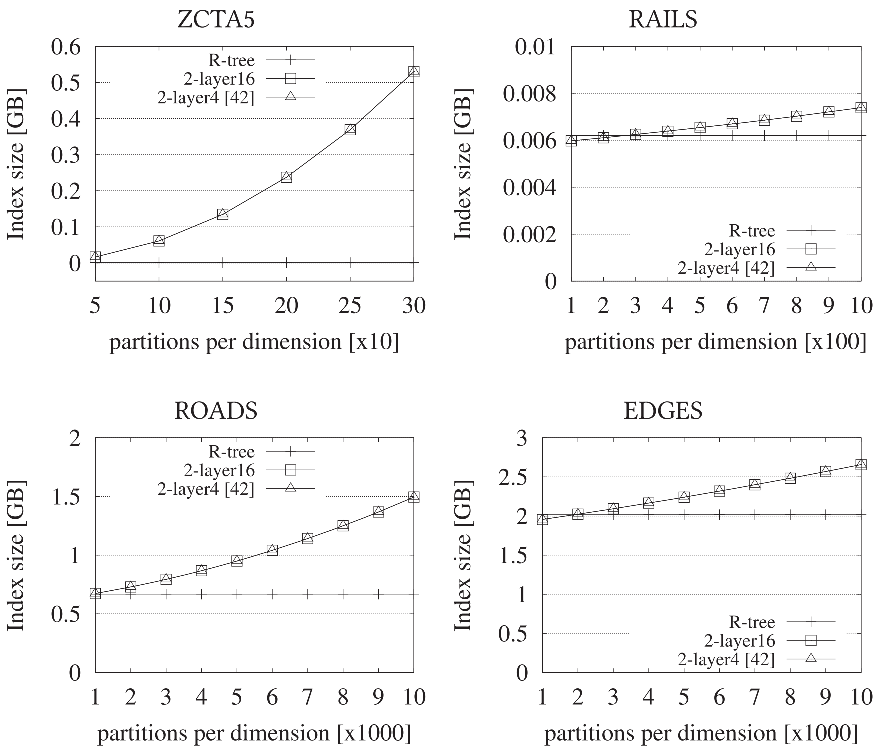


Fig. 24. Memory usage experiment.

to apply on d-dimensional spaces, for $d > 2$. For d-dimensional bounding boxes, indexed by a d-dimensional grid, the number of classes in each cell is 4^d and the classes that are selected in each cell depend on the direction of the cell w.r.t. q in each dimension. Still, in higher dimensionalities than 2, we seldom find applications with non-point objects. In the future, we plan to investigate the applicability of our approach for more advanced distance-based queries, such as closest-pair queries [11] and iceberg distance joins [39].

References

[1] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz. 2013. Hadoop-GIS: A high performance spatial data warehousing system over MapReduce. *Proceedings of the VLDB Endowment International Conference on very Large Data Bases* 6, 11 (2013), 1009–1020. DOI: https://doi.org/10.14778/2536222.2536227

[2] Abdullah Al-Mamun, Hao Wu, Qiyang He, Jianguo Wang, and Walid G. Aref. 2024. A survey of learned indexes for the multi-dimensional space. arXiv:2403.06456. Retrieved from https://arxiv.org/abs/2403.06456

[3] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. 1998. Scalable sweeping-based spatial join. In *Proceedings of 24rd International Conference on Very Large Data Bases, August 24–27, 1998, New York City, New York, USA*. Morgan Kaufmann, 570–581. Retrieved from http://www.vldb.org/conf/1998/p570.pdf

[4] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23–25, 1990*. ACM, 322–331. DOI: https://doi.org/10.1145/93597.98741

[5] Alexandros Belesiotis, Dimitrios Skoutas, Christodoulos Efstathiades, Vassilis Kaffes, and Dieter Pfoser. 2018. Spatio-textual user matching and clustering based on set similarity joins. *The VLDB Journal* 27, 3 (2018), 297–320. DOI: https://doi.org/10.1007/S00778-018-0498-5

[6] Jon Louis Bentley and Jerome H. Friedman. 1979. Data structures for range searching. *ACM Computing Surveys* 11, 4 (1979), 397–409. DOI: https://doi.org/10.1145/356789.356797

[7] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient processing of spatial joins using R-trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26–28, 1993.* ACM, 237–246. DOI: https://doi.org/10.1145/170035.170075

[8] Lu Chen, Yunjun Gao, Xuan Song, Zheng Li, Yifan Zhu, Xiaoye Miao, and Christian S. Jensen. 2023. Indexing metric spaces for exact similarity search. *ACM Computing Surveys* 55, 6 (2023), 128:1–128:39. DOI: https://doi.org/10.1145/3534963

[9] Chen Cheng, Haiqin Yang, Irwin King, and Michael R. Lyu. 2012. Fused matrix factorization with geographical and social influence in location-based social networks. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence, July 22–26, 2012, Toronto, Ontario, Canada.* AAAI Press, 17–23. DOI: https://doi.org/10.1609/AAAI.V26I1.8100

[10] Dalsu Choi, Hyunsik Yoon, Hyubjin Lee, and Yon Dohn Chung. 2022. Waffle: In-memory grid index for moving objects with reinforcement learning-based configuration tuning system. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2375–2388. DOI: https://doi.org/10.14778/3551793.3551800

[11] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. 2000. Closest pair queries in spatial databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16–18, 2000, Dallas, Texas, USA.* ACM, 189–200. DOI: https://doi.org/10.1145/342009.335414

[12] Jens-Peter Dittrich and Bernhard Seeger. 2000. Data redundancy and duplicate detection in spatial join processing. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28–March 3, 2000.* IEEE Computer Society, 535–546. DOI: https://doi.org/10.1109/ICDE.2000.839452

[13] Haowen Dong, Chengliang Chai, Yuyu Luo, Jiabin Liu, Jianhua Feng, and Chaoqun Zhan. 2022. RW-tree: A learned workload-aware framework for R-tree construction. In *Proceedings of the 38th IEEE International Conference on Data Engineering, Kuala Lumpur, Malaysia, May 9–12, 2022.* IEEE, 2073–2085. https://doi.org/10.1109/ICDE53745.2022.00201

[14] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *Proceedings of the 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13–17, 2015.* IEEE Computer Society, 1352–1363. DOI: https://doi.org/10.1109/ICDE.2015.7113382

[15] Volker Gaede and Oliver Günther. 1998. Multidimensional access methods. *ACM Computing Surveys* 30, 2 (1998), 170–231.

[16] Jian Gao, Xin Cao, Xin Yao, Gong Zhang, and Wei Wang. 2023. LMSFC: A novel multidimensional index based on learned monotonic space filling curves. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2605–2617. DOI: https://doi.org/10.14778/3603581.3603598

[17] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2023. The RLR-tree: A reinforcement learning based R-tree for spatial data. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 63:1–63:26. DOI: https://doi.org/10.1145/3588917

[18] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the Annual Meeting, Boston, Massachusetts, USA, June 18–21, 1984.* ACM, 47–57. DOI: https://doi.org/10.1145/602259.602266

[19] Gísli R. Hjaltason and Hanan Samet. 1998. Incremental distance join algorithms for spatial databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 2–4, 1998, Seattle, Washington, USA.* ACM, 237–248. DOI: https://doi.org/10.1145/276304.276326

[20] Gísli R. Hjaltason and Hanan Samet. 1999. Distance browsing in spatial databases. *ACM Transactions on Database Systems* 24, 2 (1999), 265–318. DOI: https://doi.org/10.1145/320248.320255

[21] Gísli R. Hjaltason and Hanan Samet. 2002. Speeding up construction of PMR quadtree-based spatial indexes. *VLDB J.* 11, 2 (2002), 109–137. DOI: https://doi.org/10.1007/S00778-002-0067-8

[22] Hugues Hoppe. 1996. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1996, New Orleans, LA, USA, August 4–9, 1996.* ACM, 99–108. Retrieved from https://dl.acm.org/citation.cfm?id=237216

[23] Dmitri V. Kalashnikov, Sunil Prabhakar, and Susanne E. Hambrusch. 2004. Main memory evaluation of monitoring queries over moving objects. *Distributed Parallel Databases* 15, 2 (2004), 117–135. DOI: https://doi.org/10.1023/B:DAPD.0000013068.25976.88

[24] Scott T. Leutenegger, J. M. Edgington, and Mario Alberto López. 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of the 13th International Conference on Data Engineering, April 7–11, 1997, Birmingham, UK.* IEEE Computer Society, 497–506. DOI: https://doi.org/10.1109/ICDE.1997.582015

[25] Jiangneng Li, Zheng Wang, Gao Cong, Cheng Long, Han Mao Kiah, and Bin Cui. 2023. Towards designing and learning piecewise space-filling curves. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2158–2171. DOI: https://doi.org/10.14778/3598581.3598589

[26] Mateusz Loskot and Adam Wulkiewicz. 2019. Retrieved from https://github.com/mloskot/spatial_index_benchmark

[27] Nikos Mamoulis. 2011. *Spatial Data Management.* Morgan and Claypool Publishers. DOI: https://doi.org/10.2200/S00394ED1V01Y201111DTM021

[28] Achilleas Michalopoulos, Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2023. Efficient nearest neighbor queries on non-point data. In *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2023, Hamburg, Germany, November 13–16, 2023*. ACM, 33:1–33:4. DOI: https://doi.org/10.1145/3589132.3625609

[29] Mohamed F. Mokbel, Xiaopeng Xiong, and Walid G. Aref. 2004. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13–18, 2004*. ACM, 623–634. DOI: https://doi.org/10.1145/1007568.1007638

[30] Kyriakos Mouratidis, Marios Hadjieleftheriou, and Dimitris Papadias. 2005. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14–16, 2005*. ACM, 634–645. DOI: https://doi.org/10.1145/1066157.1066230

[31] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How good are modern spatial analytics systems? *Proceedings of the VLDB Endowment* 11, 11 (2018), 1661–1673. DOI: https://doi.org/10.14778/3236187.3236213

[32] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2018. QUASII: QUery-aware spatial incremental index. In *Proceedings of the 21st International Conference on Extending Database Technology, Vienna, Austria, March 26–29, 2018*. OpenProceedings.org, 325–336. DOI: https://doi.org/10.5441/002/EDBT.2018.29

[33] Franco P. Preparata and Michael Ian Shamos. 1985. *Computational Geometry - An Introduction*. Springer. DOI: https://doi.org/10.1007/978-1-4612-1098-6

[34] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively learning spatial indices. *Proceedings of the VLDB Endowment* 13, 11 (2020), 2341–2354. Retrieved from http://www.vldb.org/pvldb/vol13/p2341-qi.pdf

[35] Suprio Ray, Rolando Blanco, and Anil K. Goel. 2014. Supporting location-based services in a main-memory database. In *Proceedings of the IEEE 15th International Conference on Mobile Data Management, Brisbane, Australia, July 14–18, 2014 - Volume 1*. IEEE Computer Society, 3–12. DOI: https://doi.org/10.1109/MDM.2014.7

[36] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. 1995. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22–25, 1995*. ACM, 71–79. DOI: https://doi.org/10.1145/223784.223794

[37] Hanan Samet. 1990. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley.

[38] John C. Shafer and Rakesh Agrawal. 1997. Parallel algorithms for high-dimensional similarity joins for data mining applications. In *Proceedings of 23rd International Conference on Very Large Data Bases, August 25–29, 1997, Athens, Greece*. Morgan Kaufmann, 176–185. Retrieved from http://www.vldb.org/conf/1997/P176.PDF

[39] Yutao Shou, Nikos Mamoulis, Huiping Cao, Dimitris Papadias, and David W. Cheung. 2003. Evaluation of iceberg distance joins. In *Advances in Spatial and Temporal Databases, 8th International Symposium, SSTD 2003, Santorini Island, Greece, July 24–27, 2003, Proceedings*. Lecture Notes in Computer Science, Vol. 2750, Springer, 270–288. DOI: https://doi.org/10.1007/978-3-540-45072-6_16

[40] Darius Sidlauskas, Simonas Saltenis, Christian W. Christiansen, Jan M. Johansen, and Donatas Saulys. 2009. Trees or grids? Indexing moving objects in main memory. In *17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2009, November 4–6, 2009, Seattle, Washington, USA, Proceedings*. ACM, 236–245. DOI: https://doi.org/10.1145/1653771.1653805

[41] Dimitrios Tsitsigkos, Panagiotis Bouros, Konstantinos Lampropoulos, Nikos Mamoulis, and Manolis Terrovitis. 2024. Two-layer space-oriented partitioning for non-point data. *IEEE Transactions on Knowledge and Data Engineering* 36, 3 (2024), 1341–1355. DOI: https://doi.org/10.1109/TKDE.2023.3297975

[42] Dimitrios Tsitsigkos, Konstantinos Lampropoulos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2021. A two-layer partitioning for non-point spatial data. In *Proceedings of the 37th IEEE International Conference on Data Engineering, Chania, Greece, April 19–22, 2021*. IEEE, 1787–1798. DOI: https://doi.org/10.1109/ICDE51399.2021.00157

[43] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 1071–1085. DOI: https://doi.org/10.1145/2882903.2915237

[44] Jingyi Yang and Gao Cong. 2023. PLATON: Top-down R-tree packing with learned partition policy. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 253:1–253:26. DOI: https://doi.org/10.1145/3626742

[45] Simin You, Jianting Zhang, and Le Gruenwald. 2015. Large-scale spatial join query processing in cloud. In *Proceedings of the 31st IEEE International Conference on Data Engineering Workshops, Seoul, South Korea, April 13–17, 2015*. IEEE Computer Society, 34–41. DOI: https://doi.org/10.1109/ICDEW.2015.7129541

[46] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. 2019. Spatial data management in apache spark: The GeoSpark perspective and beyond. *GeoInformatica* 23, 1 (2019), 37–78. DOI: https://doi.org/10.1007/S10707-018-0330-9

[47] Xiaohui Yu, Ken Q. Pu, and Nick Koudas. 2005. Monitoring K-nearest neighbor queries over moving objects. In *Proceedings of the 21st International Conference on Data Engineering, 5–8 April 2005, Tokyo, Japan*. IEEE Computer Society, 631–642. DOI: https://doi.org/10.1109/ICDE.2005.92

[48] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. 2009. SJMR: Parallelizing spatial join with MapReduce on clusters. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31–September 4, 2009, New Orleans, Louisiana, USA*. IEEE Computer Society, 1–8. DOI: https://doi.org/10.1109/CLUSTR.2009.5289178

Received 29 February 2024; revised 8 July 2024; accepted 9 September 2024