# A Two-layer Partitioning for Non-point Spatial Data

Dimitrios Tsitsigkos[†] Konstantinos Lampropoulos[†] Panagiotis Bouros[§] Nikos Mamoulis[†] Manolis Terrovitis[#]
[†]University of Ioannina, [§]Johannes Gutenberg University Mainz, [#]Athena Research Center
{dtsitsigkos, klampropoulos, nikos}@cse.uoi.gr, bouros@uni-mainz.de, mter@imis.athena-innovation.gr

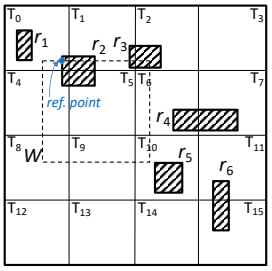
***Abstract*—Non-point spatial objects (e.g., polygons, linestrings, etc.) are ubiquitous and their effective management is always timely. We study the problem of indexing non-point objects in memory. We propose a secondary partitioning technique for space-oriented partitioning indices (e.g., grids), which improves their performance significantly, by avoiding the generation and elimination of duplicate results. Our approach is novel and of a high impact, as (i) it is extremely easy to implement and (ii) it can be used by any space-partitioning index. We show how our approach can be used to boost the performance of spatial range queries. We also show how we can avoid performing the expensive refinement step of a range query for the majority of objects and study the efficient processing of numerous queries in batch and in parallel. Extensive experiments on real datasets confirm the superiority of space-oriented partitioning over data-oriented partitioning and the advantage of our approach against alternative duplicate elimination techniques.**

## I. Introduction

The management of spatial data has been extensively studied for at least four decades [20]. Nowadays, memories have become much bigger and cheaper and in most applications, spatial object collections can easily fit in the memory of even a commodity machine. In addition, modern processors have multiple cores and facilitate parallel query processing. Although a number of distributed systems for spatial data have been developed in the past decade [1], [10], [32], [34], [24], the problem of in-memory management of large-scale spatial data has received relatively little attention.

In this paper, we study the problem of indexing non-point spatial objects (e.g., polygons, linestrings, etc.) in memory, for the efficient single- and multi-threaded evaluation of spatial range queries. Large volumes of non-point data are ubiquitous, hence, their effective management is always timely. Besides Geographic Information Systems, domains that manage big volumes of such data include graphics (e.g., management of huge meshes [13]), neuroscience (e.g., building and indexing a spatial model of the brain [25]), and location-based analytics (e.g., managing spatial influence regions of mobile users in order to facilitate effective POI recommendations [7]).

**Motivation.** Spatial access methods can be divided into two categories; *space-oriented partitioning* (SOP) and *data-oriented partitioning* (DOP) approaches. Indices of the first category divide the space into *spatially disjoint* partitions. As a result, objects that overlap with multiple partitions need to be replicated (or clipped) in each of them. DOP methods allow the extents of the partitions to overlap and ensure that their contents are disjoint (i.e., each object is assigned to exactly one partition). For disk-resident data, DOP approaches (such as the R-tree [12] and its variants) are considered to be the best, because they avoid data replication and they have a balanced structure. However, SOP approaches (especially grids) are gaining ground due to their efficiency in search and updates in main memory [21], [14], [35], [22], [29], [27]. In addition, query evaluation over grids is embarassingly parallelizable and SOP is widely used in distributed spatial data management systems [10], [32], [34].




| Tile | 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. 1. Example of partitioning and object classes

In this paper, we focus on improving SOP indices by addressing an inherent problem they have: potential duplicate query results. In particular, a range query may overlap multiple partitions which may include multiple replicas of the same object. For example, consider the six rectangular objects depicted in Figure 1, partitioned using a 4×4 grid. Some objects are assigned to multiple tiles. Given a query range (e.g., $W$), a replicated object (e.g., $r_2$) may be identified as query result multiple times (e.g., at tiles $T_0$, $T_1$, $T_4$, and $T_5$).

The classic approach to eliminate duplicates is to hash the query results and identify duplicates at each bucket. This method is very expensive, especially when the number of results is large. An improved hashing technique for spatial data that limits the size of the hash table was proposed in [2]. The state-of-the-art technique for duplicate elimination, used in most big spatial data management systems [24], is the *reference point* approach [9]. For each query result $r_i$, found in a tile $T$, this approach computes a reference point of the intersection between $r_i$ and the query window $W$ (e.g., the upper-left corner in Fig. 1). If the reference point is inside $T$, then $r_i$ is reported, otherwise it is ignored. Since the reference point can only be inside one tile, no duplicate results are reported. Although this method avoids hashing, we still have to bear the cost of retrieving duplicate copies of the same object and computing the reference point for each copy.

**Contributions.** In Sec. III, we propose a secondary par-

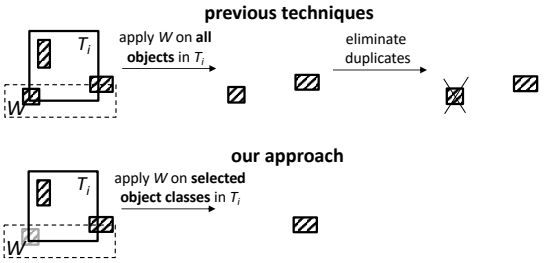


Fig. 2. Comparison between our approach and previous work

titioning technique for SOP indices, which improves their performance significantly, by avoiding the generation and elimination of duplicate results. Our approach is novel and of a high impact, as (i) it is extremely easy to implement, (ii) it can be used by any SOP index, and (iii) it can be directly implemented in big spatial data management systems [24]. In a nutshell, we divide the objects which are assigned to each partition $T$ into four classes $A, B, C, D$. Objects in class $A$ begin inside $T$ in both dimensions, objects in class $B$ start inside $T$ in dimension $x$ only, objects in class $C$ start inside $T$ in dimension $y$ only, and objects in class $D$ start before $T$ in both dimensions. Fig. 1 exemplifies how the objects are divided into classes. For example, in tile $T_1$, object $r_2$ belongs to class $C$, because $r_2$ starts before $T_1$ in the $x$ dimension and starts inside $T$ in the $y$ dimension. During query evaluation, for each partition $T$ which intersects the query range, we access *only* the object classes in $T$ that are guaranteed not to produce duplicate results. For example, in tile $T_1$ of Fig. 1, we will not access class $C$, because query $W$ starts before $T_1$ in dimension $x$; i.e., any object in class $C$ of $T_1$ that intersects $W$ should also intersect $W$ in the previous tile $T_0$. Hence, we avoid verifying whether $r_2$ intersects $W$ before realizing that it is a duplicate result. In Sec. IV-A, we explain in detail how range queries are evaluated by our scheme and show how redundant computations and duplicate checks can be avoided overall. Fig. 2 illustrates the difference between our approach and the deduplication process followed by previous work [9], [2]; while all previous approaches evaluate queries on *all* objects of each partition and then eliminate possible duplicates, we process only a subset of objects in each partition that cannot be duplicates and we do not perform any deduplication.

Besides proposing a secondary partitioning technique for duplicate avoidance in range queries, we show how to reduce the number of required comparisons per rectangle to at most one per dimension (Sec. IV-B). Furthermore, we propose a data decomposition approach which further reduces the number of comparisons (Sec. IV-C). Next, we focus on non-rectangular objects, which are approximated and indexed using their minimum bounding rectangles (MBRs). In Sec. V, we show that for such objects, the expensive query refinement step can be avoided in most cases by a simple post-filtering test on the object MBRs. In Sec. VI, we investigate the evaluation of multiple range queries in batch and in parallel, using our secondary partitioning approach.

In Sec. VII, we evaluate our proposal experimentally using large publicly available real datasets and synthetic ones of the same scale as those used in recent work [24], [18], [26]. Our experiments (with workloads of queries and updates) show that main-memory grids are superior to alternative SOP and state-of-the-art DOP indices, which justifies our focus to improve SOP indexing. More importantly, we show that when we replace the state-of-the-art duplicate elimination technique [9] by our secondary partitioning technique, the performance of grid-based indexing is improved by up to a few times. Overall, a grid index equipped with our secondary partitioning technique is up to one order of magnitude faster compared to the best performing DOP index (an in-memory R-tree implementation from boost.org) for range queries of varying sizes, achieving an impressive throughput of tens of thousands of queries per second. We also show that our (directly parallelizable) approach scales gracefully with the number of cores (i.e., threads in a multi-core machine), making it especially suitable for shared-nothing parallel environments where tree-based indices are hard to deploy. Finally, we demonstrate that in-memory spatial indexing can be orders of magnitude faster compared to distributed spatial data management systems for the scale of data used in our experiments.

## II. Related Work

In this section, we review related work on spatial indices and distributed spatial data management systems.

### A. Spatial Indexing

Spatial queries on non-point data are typically processed in two steps [20], following a *filtering-and-refinement* framework. During the *filtering* step, the query is applied on the MBRs, which approximate the objects. During the *refinement* step, the exact representations of the candidates are accessed and tested against the query predicate. Spatial indices are typically designed for the filtering step; hence, they manage MBRs instead of exact geometries.

Depending on the nature of the partitioning, spatial indices can be classified into two classes [23]. Indices based on *space-oriented partitioning* (SOP) divide the space into disjoint partitions and were originally designed for point data. A grid [5], which divides the space into cells (partitions) using axis-parallel lines, is the simplest SOP index. Hierarchical indices that fall in this category are the kd-tree [4] and the quad-tree [11]. SOP can also be used for non-point objects; in this case, objects whose extent overlaps with multiple partitions need to be replicated (or clipped) in each of them [28].

Due to object replication, the same query results may be detected in multiple partitions and deduplication techniques should be applied. Aref and Samet [2] improve the baseline hash-based duplicate elimination technique by processing the partitions in a specific order, which guarantees that duplicates may appear only in a subset of partitions (called *active border*). The size of the active border determines the space requirements of the hash table. The state-of-the-art deduplication technique by Dittrich and Seeger [9] avoids the use of a hash table; it computes a *reference point* of the intersection area between each result and the query range. If the reference point

is inside the partition, then the result is reported, otherwise it is eliminated as a duplicate. All deduplication approaches in the literature, first compute the results and then check whether they are duplicates. Our proposal is radically different; the objects in each partition are divided into four classes (during indexing). For each query, we process at each partition only the object classes that cannot produce duplicates. Hence, we avoid (i) needless comparisons to objects that would be duplicates and (ii) the cost of deduplication checks.

Indices based on *data-oriented partitioning* (DOP) allow the extents of the partitions to overlap and ensure that their contents are disjoint (i.e., each object is assigned to exactly one partition); hence, there is no need for result deduplication. Variants of the R-tree [12] (e.g., the R*-tree [3]) are the most popular methods in this class. The R-tree is a height-balanced tree, which generalizes the $B^+$-tree in the multi-dimensional space and hierarchically groups object MBRs to blocks. Each block is also approximated by an MBR, hence the tree defines a hierarchy of MBR groups. The R-tree was originally proposed for disk-resident data and the key focus is minimizing the I/O cost during query processing. The CR-tree [16] is an optimized R-tree for the memory hierarchy. BLOCK [23] is a recently proposed main-memory DOP index, which uses a hierarchy of grids.

Recently, following the trend for relational data, *learned indices* for spatial data have been proposed [31], [18], [26]. The main idea is to learn the spatial distribution of the objects, and then define a lightweight index, where search is guided by models instead of a sparse index. Wang et al. [31] first map the data to a 1D space, using their Z-order, and then construct a multi-staged learned index for 1D data. In LISA [18], the data are organized using a grid; the 1D order of the cells and the data distribution determines the grouping of cells and the corresponding learned models. RSMI [26] suggests a rank space based ordering, which becomes scalable by a recursive partitioning and learning strategy. These indices are not directly comparable to our work, because they are designed for point data (with no obvious extension to non-point data) and their primary goal is to minimize the I/O cost.

### B. Parallel and Distributed Data Management

With the advent of Hadoop, research on spatial data management has shifted to the development of distributed systems for spatial data [6], [1], [10], [33], [32], [34]. Spatial data in *Hadoop-GIS* [1] are partitioned using a hierarchical grid, wherein high density tiles are split to smaller ones. The nodes of the cluster share a *global tile index* which can be used to find the HDFS files where the contents of the tiles are stored. Spatial queries are implemented as MapReduce workloads. In the *SpatialHadoop* system [10], different options for partitioning based on different spatial indices are possible (i.e., grid based, R-tree based, quad-tree based, etc.) A global index for each dataset is stored at a Master node, indexing for each HDFS file block the MBR of its contents. A local index is built at each physical partition and used by map tasks.

Spark-based implementations of spatial data management systems [33], [32], [34] apply similar partitioning approaches. The main difference to Hadoop-based implementations is that data, indices, and intermediate results are shared in the memories of all nodes in the cluster as *resilient distributed datasets* (RDDs) and can be made persistent on disk. Unlike SpatialSpark [33] and GeoSpark [34] which are built on top of Spark, Simba [32] has its own native query engine and query optimizer, however, Simba does not support non-point geometries. Pandey et al. [24] conduct a comparison between in-memory spatial analytics systems and find that they scale well in general, although each one has its own limitations.

Distributed spatial data management systems focus on data partitioning and not on query evaluation at each partition. In other words, emphasis is given on scaling out (i.e., making the cost anti-proportional to the number of nodes), rather than on per-node scalability (i.e., reducing the computational cost per node) and multi-core parallelism. On the other hand, we focus on in-memory spatial data management and scaling up spatial query evaluation, by reducing the computational cost and exploiting multi-core parallelism.

## III. Two-layer Spatial Partitioning

In this section, we present our secondary partitioning approach for SOP spatial indices. Even though our approach can be used in any SOP index, we will present it in the context of a regular grid index, which divides the space into $N \cdot M$ disjoint spatial partitions, called *tiles*.[1] An object $o$ is assigned to a tile $T$ iff $MBR(o)$ and $T$ intersect. For each tile $T$, we keep a list of (MBR, object-id) pairs that are assigned to $T$. On the other hand, the actual geometry of each object is stored only once in an array or a hash-map and retrieved on-demand, given the object's id. Since the spatial distribution of objects may not be uniform, there could be empty tiles. If the percentage of empty tiles is very large, to save memory, we can use a hash-table to map each non-empty tile to the set of rectangles assigned to it. The above storage scheme is quite effective for main-memory data because it supports queries and updates quite fast.

**Secondary Partitioning.** We propose that the set of MBRs at each tile is further divided into four classes $A$, $B$, $C$, and $D$ (which are physically stored separately in memory). Each MBR $r$ can be represented by an interval of values at each dimension. Let $r.x = [r.x_l, r.x_u]$ be the projection of rectangle $r$ on the $x$ axis and $r.y = [r.y_l, r.y_u]$ $r$'s $y$-projection. Now, consider a rectangle $r$ which is assigned to tile $T$.

- $r$ belongs to class $A$, if for every dimension $d \in \{x, y\}$, the begin value $r.d_l$ of $r$ falls into projection $T.d$, i.e., if $T.d_l \leq r.d_l$.
- $r$ belongs to class $B$ if $r.x$ begins inside $T.x$ and $r.y$ begins before $T.y$, i.e., if $T.x_l \leq r.x_u$ and $T.y_l > r.y_l$.
- $r$ belongs to class $C$ if $r.x$ begins before $T.x$ and $r.y$ begins inside $T.y$, i.e., if $T.x_l > r.x_l$ and $T.y_l \leq r.y_l$.

[1] In Table V, we test how the quad-tree can benefit from our secondary partitioning.

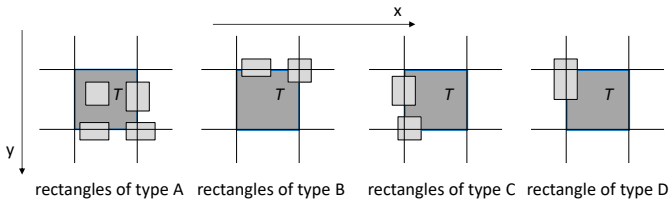



Fig. 3. The four classes of rectangles assigned to a tile $T$.

- $r$ belongs to class $D$ if both its $x$- and $y$-projections begin before $T$, i.e., if $T.x_l > r.x_l$ and $T.y_l > r.y_l$.

Figure 3 illustrates examples of rectangles in a tile $T$ that belong to the four different classes.[2] During data partitioning, for each tile $T$ a rectangle $r$ is assigned to, we identify its class and place it to the corresponding division. Note that a rectangle can belong to class $A$ of just one tile, while it can belong to other classes (in other tiles) an arbitrary number of times. We denote the secondary partitions of tile $T$ which store the MBRs of classes $A$, $B$, $C$, and $D$, by $T^A$, $T^B$, $T^C$, and $T^D$, respectively. Table I summarizes the notation used frequently in the paper.

TABLE I
TABLE OF NOTATIONS

| Notation | Description |
|---|---|
| $W$ | query window |
| $r.d = [r.d_l, r.d_u]$ | projection of rectangle $r$ at dimension $d \in \{x, y\}$ |
| $T^X$ | secondary partition of tile $T$ holding MBRs in class $X \in \{A, B, C, D\}$ |
| $prev(T, d)$ | previous tile to $T$ in dimension $d \in \{x, y\}$ |
| $L^X_{d_l}$ ($L^X_{d_u}$) | decomposed table holding $\langle r.d_l, id\rangle$ ($\langle r.d_u, id\rangle$) pairs of class $X$ and dimension $d \in \{x, y\}$ |

## IV. Range Query Evaluation

In this section, we show how the secondary partitions at each tile $T$ can be used to avoid the generation and elimination of duplicate query results. We first consider rectangular range queries $W$ (window queries). For now, we focus on the *filtering step* of the query, i.e., the objective is to just find the object MBRs which intersect $W$. The refinement step will be discussed in Section V.

First, the tiles which intersect $W$ in a $N \times M$ regular grid can be found in $O(1)$ time, by algebraic operations. Specifically, assuming that tile $T_{i,j}$ is at the $i$-th row and at the $j$-th column of the grid, the tiles which intersect $W$ are all tiles $T_{i,j}$, for which $\lfloor W.x_l/N \rfloor \leq i \leq \lfloor W.x_u/N \rfloor$ and $\lfloor W.y_l/M \rfloor \leq j \leq \lfloor W.y_u/M \rfloor$. We now explain in detail, for each tile $T$ that intersects $W$, which classes of rectangles should be accessed and which computations are necessary for determining whether each rectangle $r$ intersects $W$. Our goal is not only to avoid accessing irrelevant secondary partitions, but also to minimize the computational cost for finding the query results in the relevant partitions of $T$.

[2]We conventionally assume that the $x$ dimension is from left to right and the $y$ dimension is from top to bottom.

### A. Selecting relevant classes

For a tile $T$, let $prev(T, d)$ denote the tile which is right before $T$ in dimension $d$ and has exactly the same projection as $T$ in the other dimension. For example, in Figure 4, $prev(T, x)$ (resp. $prev(T, y)$) is the tile right before $T$ in dimension $x$ (resp. $y$). Given a window query $W$, the following lemmas determine the classes of rectangles in $T$ which should be disregarded, because they can only produce duplicate results.

*Lemma 1:* If the query range $W$ intersects tile $T$ and $W$ starts before $T$ in dimension $x$, then secondary partitions $T^C$ and $T^D$ should be disregarded.

*Proof.* Consider a rectangle $r$ in class $C$ or class $D$ of tile $T$, i.e., $r \in T^C$ or $r \in T^D$. Rectangle $r$ should also be assigned to the previous tile $prev(T, x)$ to $T$ in dimension $x$, because it belongs to class $C$ or $D$ of $T$. If $r$ intersects $W$ in $T$, then $r$ should also intersect $W$ in $prev(T, x)$, because $W$ also starts before $T$ in dimension $x$. Hence, examining and reporting $r$ in tile $T$ would produce a duplicate, since the same result can also be identified in tile $prev(T, x)$. □

*Lemma 2:* If $W$ intersects tile $T$ and $W$ starts before $T$ in dimension $y$, then secondary partitions $T^B$ and $T^D$ should be disregarded.

Lemma 2 can be proved by replacing $x$ by $y$ and $C$ by $B$ in the proof of Lemma 1. The two lemmas are combined to exclude all classes $B$, $C$, and $D$ if $W$ starts before $T$ in both dimensions. To illustrate the lemmas, consider tile $T$ in Figure 4. In addition, consider the MBRs of objects $o_1$ and $o_2$, which belong to secondary partitions $T^B$ and $T^C$, respectively. $MBR(o_1)$ should be ignored when processing $T$ because it belongs to class $B$ and $W$ starts before $T$ in dimension $y$ (Lemma 2). Indeed, $MBR(o_1)$ intersects $W$ also in tile $prev(T, y)$ which is right above $W$. On the other hand, $W$ does not start before $T$ in dimension $x$, i.e., Lemma 1 does not apply for tile $T$. This means that $MBR(o_2) \in T^C$ will be found to intersect $W$. Figure 4 shows, in the top-left corner of each tile $T$ intersected by $W$, the object classes in $T$ that should be examined (the remaining classes can be disregarded). Observe that we have to consider all objects in just one tile (the one containing point $(W.x_l, W.y_l)$). For the majority of tiles, we only have to examine secondary partition $T^A$.

### B. Minimizing the number of comparisons

We now turn our attention to *minimizing the number of comparisons* needed for each secondary partition that *has to* be checked (i.e., those not eliminated by Lemmas 1 and 2). For a rectangle $r$ in a tile $T$ to intersect the query window $W$, $r.x$ should intersect $W.x$ and $r.y$ should intersect $W.y$. Hence, to test whether $x$ intersects $W$, we need at most four comparisons (i.e., $r$ and $W$ do not intersect, iff $r.x_u < W.x_l$ or $r.x_l > W.x_u$ or $r.y_u < W.y_l$ or $r.y_l > W.y_u$).

A direct observation that saves comparisons is that, if a tile $T$ is covered by the window $W$ in a dimension $d$, then we do not have to perform intersection tests in dimension $d$ for all rectangles in the relevant secondary partitions in $T$. In the example of Figure 4, we need to examine partitions $T^A$

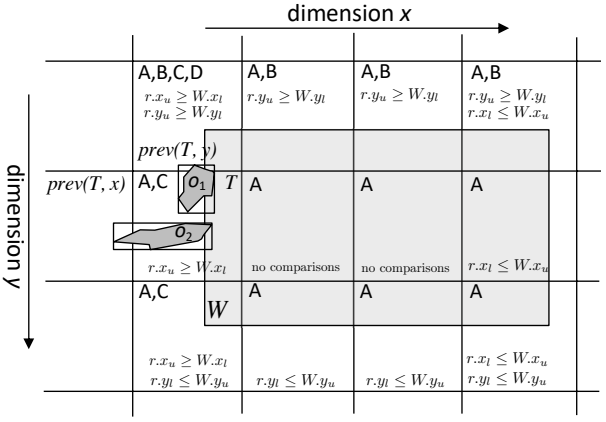



Fig. 4. Examples of object classes and comparisons

and $T^C$ of tile $T$ (Lemma 2). For each rectangle $r$ in these partitions, we only have to verify if projections $r.x$ and $W.x$ intersect, because $r.y$ and $W.y$ definitely intersect (since $T.y$ is covered by $W.y$).

For the dimension(s) where $T$ is not covered by $W$, the following lemmas can be used to further reduce the necessary comparisons.

*Lemma 3:* If $W$ ends in tile $T$ and starts before $T$ in dimension $d$, then for a rectangle $r \in T$, $r$ intersects $W$ in dimension $d$ iff $r.d_l \leq W.d_u$.

Symmetrically, we can show:

*Lemma 4:* If $W$ starts in tile $T$ and ends after $T$ in dimension $d$, then for a rectangle $r \in T$, $r$ intersects $W$ in dimension $d$ iff $r.d_u \geq W.d_l$.

For example, in tile $T$ of Figure 4, we only have to test intersection in dimension $x$, as already explained. The intersection test can be reduced to a simple comparison, i.e., $r$ intersects $W$ iff $r.x_u \geq W.x_l$, due to Lemma 4. To demonstrate the impact of Lemmas 3 and 4, in each tile of the figure, we show the necessary comparisons. For the two tiles in the center, no comparisons are required because all MBRs (in class A) are guaranteed to intersect $W$. For the remaining two tiles, which intersect the border of $W$, we only have to perform at most one comparison per dimension, because $W$ either starts or ends at these tiles (and some of these tiles are totally covered by $W$ in one dimension). Contrast this to the four comparisons required in the general case for testing whether two rectangles (e.g., $r$ and $W$) intersect. Therefore, for range queries that cover multiple tiles, we have:

*Corollary 1:* For a window query $W$ that intersects more than one tile per dimension, the number of required comparisons per rectangle in each relevant tile is at most two.

### C. Storage decomposition

Conventionally, each MBR $r$ is stored as a quintuple $\langle id, r.x_l, r.x_u, r.y_l, r.y_u \rangle$. To further reduce the query cost and improve the data access locality, we propose the representation of each MBR $r$ by four pairs: $\langle r.x_l, id \rangle$, $\langle r.x_u, id \rangle$, $\langle r.y_l, id \rangle$, $\langle r.y_u, id \rangle$, following the Decomposition Storage Model (DSM) [8], [30]. Hence, for each of the secondary partitions $T^X \in \{T^A, T^B, T^C, T^D\}$, we can define four decomposed tables $L^X_{x_l}, L^X_{x_u}, L^X_{y_l}, L^X_{y_u}$, which store the four pairs of each rectangle in $T^X$, respectively. The tables are sorted by their first column and used to evaluate fast queries on tiles, where just one endpoint of each MBR needs to be compared (according to Lemmas 3 and 4).

TABLE II
REQUIRED DECOMPOSED TABLES FOR EACH SECONDARY PARTITION

| partition | required tables |
|---|---|
| $T^A$ | $L^A_{x_l}$, $L^A_{x_u}$, $L^A_{y_l}$, $L^A_{y_u}$ |
| $T^B$ | $L^B_{x_l}$, $L^B_{x_u}$, $L^B_{y_u}$ |
| $T^C$ | $L^C_{x_u}$, $L^C_{y_l}$, $L^C_{y_u}$ |
| $T^D$ | $L^D_{x_u}$, $L^D_{y_u}$ |

In particular, for each tile $T$ satisfying Lemma 3 in dimension $d$, we can perform *binary search* on each of its relevant tables $L^X_{d_l}$, having the $\langle r.d_l, id \rangle$ tuples, to find the largest $r.d_l$, which satisfies $r.d_l \leq W.d_u$. All rectangles in the table up to this value are guaranteed to satisfy the condition and can be reported without any comparison.[3] Symmetrically, we can reduce the comparisons for rectangles in a tile $T$, which satisfies Lemma 4, by taking advantage of the sorted tables $L^X_{d_u}$. For example, for the tile $T$ in Figure 4, we only have to access and perform binary search to tables $L^A_{x_u}$ and $L^C_{x_u}$, which store the $\langle r.x_u, id \rangle$ decompositions of the rectangles in secondary partitions $T^A$ and $T^C$, respectively. If we have to perform two comparisons in a tile (e.g., $r.x_u \geq W.x_l$ and $r.y_u \geq W.y_l$), we choose one of the two relevant decomposed tables (e.g., $L_{x_u}$ or $L_{y_u}$) to perform the search; then, for each qualifying rectangle according to the selected comparison, we verify the other comparison by accessing the entire MBR. We select the table in the dimension which is covered the least by $W$, in order to minimize the necessary verifications.

Finally, we observe that, for some object classes, it is not necessary to store all decompositions. For example, the only possible comparisons that can be applied to rectangles of class $D$ are $r.x_u \geq W.x_l$ and $r.y_u \geq W.y_l$, because all MBRs of class $D$ start before the tile in both dimensions and they are only compared with $W$ in the tile that includes the start point of $W$ in both dimensions (Lemma 4). Hence, we only need to keep tables $L^D_{x_u}$ and $L^D_{y_u}$ for each secondary partition $T^D$. Overall, we can reduce the storage requirements for the decomposed tables as shown in Table II.

The decomposed data representation not only reduces the number of comparisons but also accesses only the necessary data for each verified comparison. In particular, rentangle coordinates which are not relevant to the required verification are not accessed at all, while in a record-based representation irrelevant data are fetched to the memory cache. On the other hand, the decomposed representation requires additional storage and is more expensive to update (unless a batch update strategy is employed); hence, it is mostly appropriate for indexing static spatial object collections.

[3]Alternatively, we can scan from the beginning of the table until the condition is violated.

### D. Overall approach

Algorithm 1 describes the steps of window query evaluation. Given a window $W$, we first identify the tiles $\mathcal{T}$ that intersect $W$ by simple algebraic operations, as discussed in the beginning of this section. Then, for each tile $T \in \mathcal{T}$, we identify the secondary partitions $\mathcal{P}_T$ that would not produce duplicates, using Lemmas 1 and 2. For each such secondary partition $T^X$, we find all rectangles that intersect $W$, by applying the techniques of Sec. IV-B to reduce the necessary computations. We also use the decomposed tables presented in Sec. IV-C. Note that the operations at each tile $T$ (and each secondary partition in $T$) are *totally independent* to each other and they can be parallelized without the need of any synchronization.

**Algorithm 1** Window query evaluation (filtering step)

**Require:** grid $\mathcal{G}$, query window $W$
1: $\mathcal{T}$ = tiles in $\mathcal{G}$ that intersect $W$
2: **for** each tile $T \in \mathcal{T}$ **do**
3: $\quad\mathcal{P}_T$ = sub-partitions of $T$ relevant to $W$ ▷ Lemmas 1 & 2
4: $\quad$**for** each sub-partition $T^X \in \mathcal{P}_T$ **do**
5: $\quad\quad$find all $r \in T^X$ that intersect $W$ ▷ Sec. IV-B & IV-C
6: $\quad$**end for**
7: **end for**

Although we focus on indexing 2D MBRs in this paper, our secondary partitioning scheme can directly be used for minimum bounding boxes (MBBs) of arbitrary dimensionality $m$. In a nutshell, we need $2^m$ classes to re-partition an $m$-dimensional tile $T$, which indexes $m$-dimensional MBBs. For each tile $T$, if MBB $r$ intersects $T$, there are two cases for each dimension $d$: either $r$ begins inside $T$ ($r.d_l \geq T.d_l$) or before $T$ ($r.d_l < T.d_l$). Hence, there are $2^m$ cases (classes) in total. Lemmas 1 and 2 can be generalized to a lemma that prunes all classes corresponding to cases of MBBs that begin before $T$ in each dimension $d$, if $W$ begins before $T$ in that dimension. Lemmas 3 and 4 apply to any dimensionality and the decomposition approach of Section IV-C can directly be used to split each $m$-dimensional MBB to $2 \cdot m$ pairs. These pairs can be used to search fast in tiles that are on the boundary of the query range.

### E. Non-rectangular ranges

Window queries are the most popular range queries. Still, not all query ranges are rectangular. A characteristic non-rectangular range query is the *disk* (or *distance*) range query, where the objective is to find all objects with (minimum) distance to a given query point $q$ at most $\epsilon$. To evaluate a disk query on our two-layer partitioned dataset, we apply a similar method to Algorithm 1; we first find the set of tiles $\mathcal{T}$ that intersect with the disk (using algebraic/trigonometric operations) and then find the objects in them that satisfy the query predicate. As in window queries, for each tile $T \in \mathcal{T}$, we check whether $prev(T, d)$ in each dimension $d$ is also in $\mathcal{T}$. If yes, then we disregard the corresponding class of rectangles in $T$. Hence, if $prev(T, x) \in \mathcal{T}$, then classes $B$ and $D$ are disregarded, whereas if $prev(T, y) \in S$, then classes $C$ and $D$ are disregarded. Figure 5 shows an example of a disk query

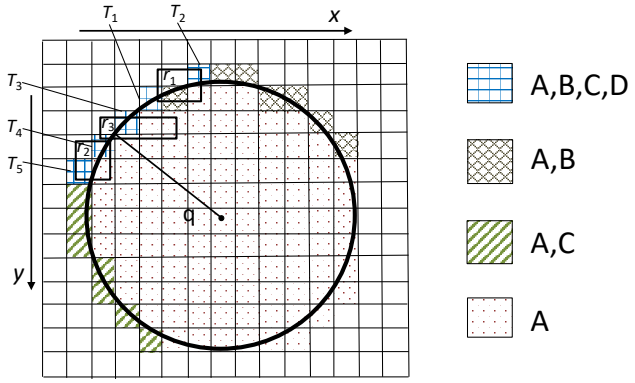



Fig. 5. Example of disk query evaluation

centered at q. The tiles which intersect the disk are shown by different patterns depending on the classes of rectangles in them that have to be checked. For example, in tile $T_5$ all four classes will be examined (we call $T_5$ an $ABCD$ tile, in the context of the disk query). Note that for the majority of tiles which intersect the disk range, we only have to examine rectangles in class $A$.

A subtle point here is that if we simply examine all rectangles in the classes that correspond to each tile, we may end up examining duplicates. For example, consider rectangle $r_1$, which will be examined in both tiles $T_1$ (in class $B$) and $T_2$ (in class $C$). To avoid such duplicates, for each rectangle in an $ABCD$ tile $T$, if the tile is closer to $q$ in the $y$-dimension compared to the $x$-dimension, we ignore rectangles $r$ in classes $C$ and $D$, for which $r.y_u > T.y_u$ (these will be handled in another tile). The case where $T$ is closer to $q$ in the $x$-dimension is handled symmetrically.

For tiles which are totally covered by the disk range, we do not verify any distances between the objects assigned to them and $q$, as these are guaranteed to be query results. Distance verification only has to be performed for objects in tiles which partially intersect the disk.

The method described above for disk queries can be generalized for any non-rectangular query. We first find the set of tiles $\mathcal{T}$ which intersect the query range. Then, for each tile $T \in S$, we determine which classes of objects need to be examined (i.e., exclude classes that would produce duplicates). For each tile which is totally covered by the query region, we just report its contents in the relevant classes as results and for the remaining tiles we conduct an intersection test for each rectangle before determining whether it is a result.

## V. Refinement Step

We now discuss the evaluation of the refinement step of range queries using our secondary partitioning scheme. We begin by a general and important lemma, which is independent to our approach.

*Lemma 5 (Secondary filtering):* Given a candidate object whose MBR $r$ intersects the query range, if at least one side of $r$ is inside the query range, then the object is guaranteed to intersect the range and no refinement step for the object is necessary.

The lemma is trivial to prove, based on the definition of MBR. If one side of the MBR is inside the query range, then there should be at least one point of the object inside the query range, i.e., the object and the range intersect. For rectangular query ranges $W$, we can simplify the test by checking whether at least one of the projections $r.x$ or $r.y$ of $r$ is covered by the corresponding projection $W.x$ or $W.y$ of $W$. If this is true, given that $r$ intersects $W$, at least one point of the object corresponding to $r$ should be inside $W$. This test costs at most four comparisons. For a disk query range, we can check whether there are at least two corners of $r$ whose distances to the disk center are smaller than or equal to the disk radius (in this case at least one side of $r$ should be inside the disk). This test costs at most four distance computations.

**Efficient secondary filtering.** We now show how we can use our two-layer partitioning approach to reduce the cost of applying the post-filtering tests (Lemma 5). Specifically, for each $T$ that intersects a query range $W$ and for each dimension $d$, we consider two cases: (i) $W$ starts before $T$ in dimension $d$, i.e., $W.d_l < T.d_l$ and (ii) $W.d_l \geq T.d_l$. In the first case, due to Lemmas 1 and 2, only classes of rectangles that start inside $T$ in dimension $d$ are considered, which means for each rectangle $r \in T$ which is found to intersect $W$, we already know that $W.d_l < r.d_l$. Hence, we only have to test if $r.d_u \leq W.d_u$ to confirm whether $r$ is covered by $W$ in dimension $d$. On the contrary, for the case where $W.d_l \geq T.d_l$, we should apply the complete coverage test (i.e., $W.d_l \leq r.d_l \wedge r.d_u \leq W.d_u$) in dimension $d$.

For example, in Figure 4, we only have to perform the complete coverage test for all rectangles that intersect $W$ in the top-left tile ($prev(T, y)$). In the tiles, where we access classes $A$ and $B$, we save one comparison in the $x$ dimension, in the tiles, where we access classes $A$ and $C$, we save one comparison in the $y$ dimension, and in all other tiles (where we access only class $A$), we save one comparison per dimension.

## VI. Batch Query Processing

In the previous sections, we presented how our two-layer index handles single query requests. Real systems however receive and need to evaluate a large number of concurrent queries. Under this, we next discuss how to efficiently process batches of spatial range queries. Although our focus is primarily in a single-threaded processing environment, parallel query processing in modern multi-core hardware can also benefit from the ideas discussed in this section. To this end, our experimental analysis includes both single-threaded and multi-threaded experiments.

**Queries-based approach.** A straightforward approach for processing a workload of concurrent spatial range queries is to directly evaluate every query independently. In a parallel processing environment, we can easily adopt this approach by assigning the queries to the available threads in a round robin fashion. We call this simple approach queries-based. Its main shortcoming is that it is cache agnostic; as every issued query $q$ typically overlaps multiple tiles of the grid, the computation of $q$ requires accessing data structures in different parts of the main memory, i.e., the memory access pattern is prone to cache misses. The problem is present also in parallel query processing, as every thread goes through multiple rounds of *content switching*.

TABLE III
Real-world datasets used in the experiments

| dataset | type | card. | avg. $x$-extent | avg. $y$-extent |
|---|---|---|---|---|
| ROADS | linestrings | 20M | 0.00001173 | 0.00000915 |
| EDGES | polygons | 70M | 0.00000491 | 0.00000383 |
| TIGER | mixed | 98M | 0.00000740 | 0.00000576 |

**Tiles-based approach.** To address this shortcoming of queries-based, we design a cache-conscious two-step approach. Given a large batch of queries $Q$, for each tile, we first accumulate the subtasks of all queries in $Q$ that intersect the tile. Each subtask corresponds to accessing and processing (the relevant to the query) secondary partitions in the tile. Then, in a second step, we initiate one process at each tile, which evaluates the corresponding subtasks. Essentially, query processing is no longer driven by the queries, but from the grid tiles and therefore, we call this approach tiles-based. This method is favored by parallel processing, since each thread (corresponding to a tile) can benefit from the processor's cache while processing the subtasks assigned to it. As we demonstrate in Section VII the tiles-based approach scales better with the number of parallel threads compared to queries-based.

## VII. Experimental Evaluation

Our analysis was conducted on a machine with 384 GBs of RAM and a dual Intel(R) Xeon(R) CPU E5-2630 v4 clocked at 2.20GHz running CentOS Linux 7.6.1810. All methods were implemented in C++, compiled using `gcc` (v4.8.5) with flags `-O3`, `-mavx` and `-march=native`. For our parallel processing tests, we used OpenMP and activated hyper-threading, allowing us to run up to 40 threads.

**Datasets.** We experimented with publicly available Tiger 2015 datasets [10], summarized in Table III. The third dataset resulted by merging all polygon and linestring Tiger 2015 objects, excluding zip codes, counties and states. The objects in each dataset were normalized so that the coordinates in each dimension take values inside $[0, 1]$. The last two columns of the table are the relative (over the entire space) average length for every object's MBR at each axis. In order to test the robustness of our index, we also experimented with synthetically generated datasets with rectangles of uniform and zipfian spatial distribution. Table IV shows the parameters used in data generation. The coordinates in each dimension take values inside $[0, 1]$ and all generated rectangles in a dataset have the same area. The width to height ratio of each rectangle was generated randomly in the range $[0.25, 4]$ in order to avoid unnaturally narrow rectangles.

**Methods**. We implemented our secondary partitioning approach as part of a main-memory regular grid spatial index. We designed two variants of the index. In the first variant,

TABLE IV
SYNTHETIC DATASETS (MBRS) USED IN THE EXPERIMENTS

| parameter | values | default |
|---|---|---|
| cardinality | 1M, 5M, 10M, 50M, 100M | 10M |
| area | $10^{-\infty}$, $10^{-14}$, $10^{-12}$, $10^{-10}$, $10^{-8}$, $10^{-6}$ | $10^{-10}$ |
| distribution | Uniform or Zipfian ($\alpha = 1$) | — |

TABLE V
COMPARED METHODS AND THEIR THROUGHPUT (WINDOW QUERIES)

| type | index | details | throughput [queries/sec] | |
|---|---|---|---|---|
| | | | ROADS | EDGES |
| SOP | 2-layer | Section III | 30981 | 9406 |
| | 2-layer$^+$ | Section IV-C | 36444 | 10855 |
| | 1-layer | grid with [9] | 12597 | 4403 |
| | quad-tree | [11] using [9] | 10949 | 3640 |
| | quad-tree, 2-layer | [11] + Section III | 16883 | 5831 |
| DOP | R-tree | [17] (boost.org) | 7888 | 2011 |
| | R*-tree | [3] (boost.org) | 6415 | 1610 |
| | BLOCK | [23] | $< 1$ | $< 1$ |
| | MXCIF quad-tree | [15] | 8 | 2 |

termed 2-layer, for each tile $T$ of the grid, we divide the (MBR, id) pairs assigned to $T$ into four secondary partitions ($T^A$, $T^B$, $T^C$, $T^D$), as discussed in Section III. In the second variant, termed 2-layer$^+$, each secondary partition $T^X$ is further divided into decomposed tables, as discussed in Section IV-C.

We considered both SOP and DOP competitors to our 2-layer and 2-layer$^+$, summarized in Table V. First regarding SOPs, the 1-layer index is an in-memory grid with identical primary partitioning as our 2-layer, but uses the reference point approach [9] to perform duplicate elimination. Comparing 1-layer to 2-layer and 2-layer$^+$ shows the benefit of our secondary partitioning scheme and the techniques we propose in Section IV for duplicate avoidance and minimization of comparisons. The second SOP competitor is a quad-tree implementation, which assigns each object MBR to all quadrants it intersects. As soon as the contents of a quadrant exceed a predefined maximum *capacity* (set to 1000, after tuning), the quadrant is split to four; the rectangles are then re-distributed in the four generated children and *replicated* if they span the division borders. In order to avoid extensive splitting of quad-tree nodes in the case of extremely skewed data, a *maximum tree depth* (=12) is set. The reference point approach [9] is also used for duplicate elimination. We also implemented a version of quad-tree that uses our approach instead of [9]. Regarding DOPs, we used two implementations of in-memory R-trees from the highly optimized Boost. Geometry library (boost.org)[4]; an STR-bulkloaded [17] (denoted for simplicity as R-tree) and an R*-tree [3]. Both trees have a fanout of 16 for inner and leaf nodes; this configuration is reported to perform the best (we also confirmed this by testing). The next DOP competitor is BLOCK; the implementation was kindly provided by the authors of [23]. Finally, we also implemented and tested the MXCIF quad-tree for non-point data [15], which does not replicate objects that span quadrants, but stores each object at the lowest-level quadrant which covers the object. All compared methods are listed in Table V.

[4]Recent benchmarks [19] showed the superiority of Boost.Geometry R-tree implementations over the ones in libspatialindex.org

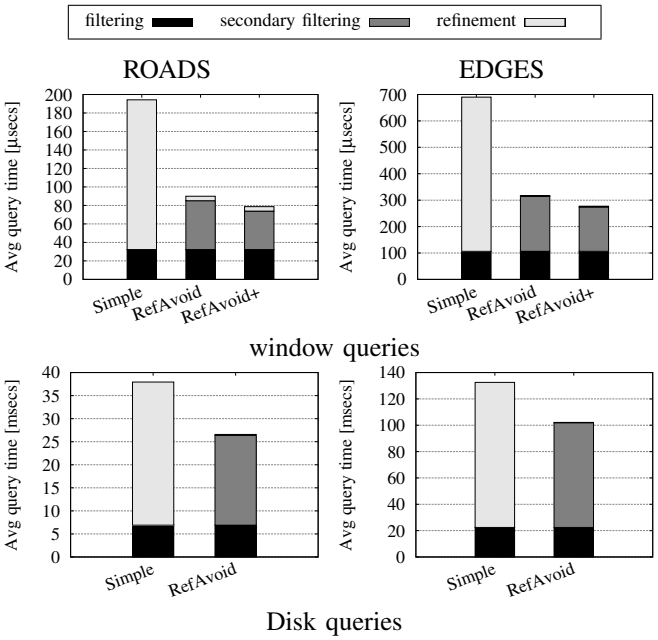



Fig. 6. Time breakdown in two-layer indexing

**Queries**. We experimented with both window and disk queries which apply on non-empty areas of the map (i.e., they always return results). We vary their relative area as a percentage of the entire data space, inside the $\{0.01, 0.05, 0.1, 0.5, 1\}$ value range (default value $0.1\%$ of the area of the map). Queries on synthetic data follow the same spatial distribution as the data.

### A. Filtering vs. Refinement

In the first experiment we evaluate the effectiveness of our extra pre-refinement filtering (Lemma 5). We used our 2-layer index and considered three variants of query evaluation; filtering is identical in all three variants. Under Simple, all candidates identified by the filtering step are passed to the refinement step; RefAvoid employs Lemma 5 to reduce the number of candidates to be refined; last, RefAvoid$^+$ enhances RefAvoid by using our secondary partitioning, as discussed at the end of Section V. Figure 6 breaks down the average execution time for 10000 window and disk queries; note that for disk queries RefAvoid$^+$ is not applicable. The pre-refinement filter is very effective; both RefAvoid and RefAvoid$^+$ significantly reduce the number of candidates to be refined by over 90%. To achieve this however, they apply extra comparisons using the MBRs; these comparisons are more expensive in the case of disk queries because they involve costly distance computations between the disk center and the corners of object MBRs. When our secondary filtering technique is used, the bottleneck of window queries is in the filtering step; hence, in the subsequent experiments, we focus on the filtering step.

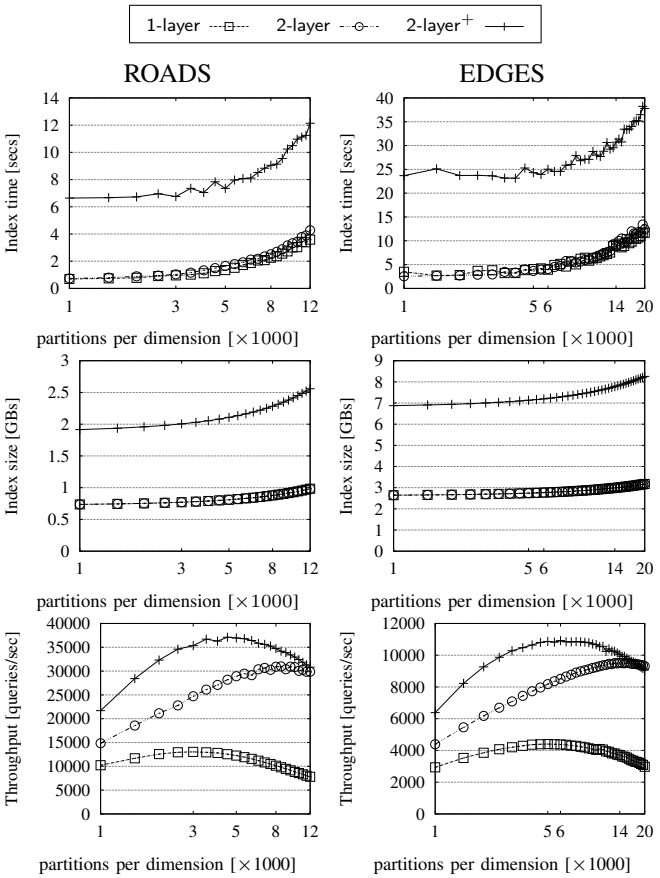



Fig. 7. Building and tuning grid-based indices (window queries)

### B. Indexing and Tuning

We next investigate the index building cost and tuning. The first four plots of Figure 7 compare the indexing times and the sizes of the three grid-based indices on ROADS and EDGES datasets, while varying the granularity of the grid partitioning. Naturally, the indexing cost for all three indices rises as we increase the granularity of the grid. As expected, 1-layer and 2-layer have the same space requirements; regardless of employing secondary partitioning or not, both indices store exactly the same number of object MBRs (originals and replicas). Note that the index sizes do not grow too much with the grid granularity, which means that MBR replication is not excessive. In terms of indexing time, 2-layer is only slightly more expensive than 1-layer. On the other hand, the indexing cost of 2-layer$^+$ is higher than both 1-layer and 2-layer indices, because 2-layer$^+$ essentially stores a second (decomposed) copy of the rectangles inside every tile. The construction costs for the two quadtrees (not shown) are 7s and 28.2s, respectively, and their sizes are similar to those of the corresponding 1-layer indices. The sizes of the packed R-trees (not shown) are about the same as the sizes of the corresponding 1-layer (and 2-layer) indices, indicating that the replication ratio of our indices is low. In addition, the bulk loading costs of the R-trees are 5.2s and 19.5s for the two datasets, respectively, i.e., about 20% lower compared to the construction cost of 2-layer$^+$.

The last two plots of Figure 7 compare the window query throughputs of 1-layer, 2-layer, and 2-layer$^+$ for different grid granularities. The three methods achieve their best throughputs when several thousands of partitions per dimension are used. Observe that employing our secondary partitioning significantly enhances query processing; 2-layer and 2-layer$^+$ always outperform 1-layer by a wide margin (2x–3x). It is worth noting that 1-layer uses the comparisons reduction optimization described in Section IV-B, meaning that the performance gap is due to our secondary partitioning and the storage decomposition (by 2-layer$^+$). Specifically, our approach outperforms the state-of-the-art reference point method for result deduplication [9] used by 1-layer by a factor of at least 2. For a wide range of granularities (i.e., 1000 to 10000 partitions per dimension), the throughput of all three methods does not change significantly meaning that finding the best granularity is not crucial to query performance. The fastest index is 2-layer$^+$ as it trades the extra used space for better query performance We observed similar trends on the TIGER and on the synthetic datasets (not shown due to lack of space). For the rest of our analysis, we used the best granularity for 1-layer, 2-layer and 2-layer$^+$.

### C. Query and Update Performance

We now compare all indices in terms of query throughput (window and disk queries), evaluate batch and parallel query processing, and finally measure their update costs.

**Window queries**. First, we report in Table V the throughput (queries/sec) achieved by each index for 10K window queries (of average relative area 0.1% of the map) on ROADS and EDGES. 2-layer and 2-layer$^+$ outperform the competition by a wide margin. Note that the performance of quad-tree is greatly improved by our secondary partitioning, which confirms that other SOPs besides grids can benefit from our approach. However, as 2-layer is consistently more efficient than the quad-tree using our approach, we do not consider the latter in the rest of the experiments. R-tree is the most efficient DOP competitor, outperforming R*-tree (from the same library). BLOCK takes seconds to evaluate range queries on our datasets, which can be attributed to the fact that it is implemented for 3D objects. Similar, MXCIF quad-tree is orders of magnitude slower than the R-tree. Under these, in the rest of the tests we only include 1-layer, R-tree and quad-tree indices as the key competitors to our 2-layer and 2-layer$^+$.

The first two columns of Figure 8 show the throughput of the five competitors for window queries of varying relative area and selectivity on the three real datasets. For the experiments of the second column, we collected the runtimes of all queries (regardless of their areas) and averaged them after grouping them by selectivity. Naturally, query processing is negatively affected by both factors. We also observe that 2-layer and 2-layer$^+$ are consistently much faster than the competition on all datasets and query areas; in addition, the relative difference between 2-layer and 2-layer$^+$ is stable. For each query, 2-layer and 2-layer$^+$ access the relevant partitions very fast (without the need of traversing a hierarchical index) and manage to drastically reduce the required number of

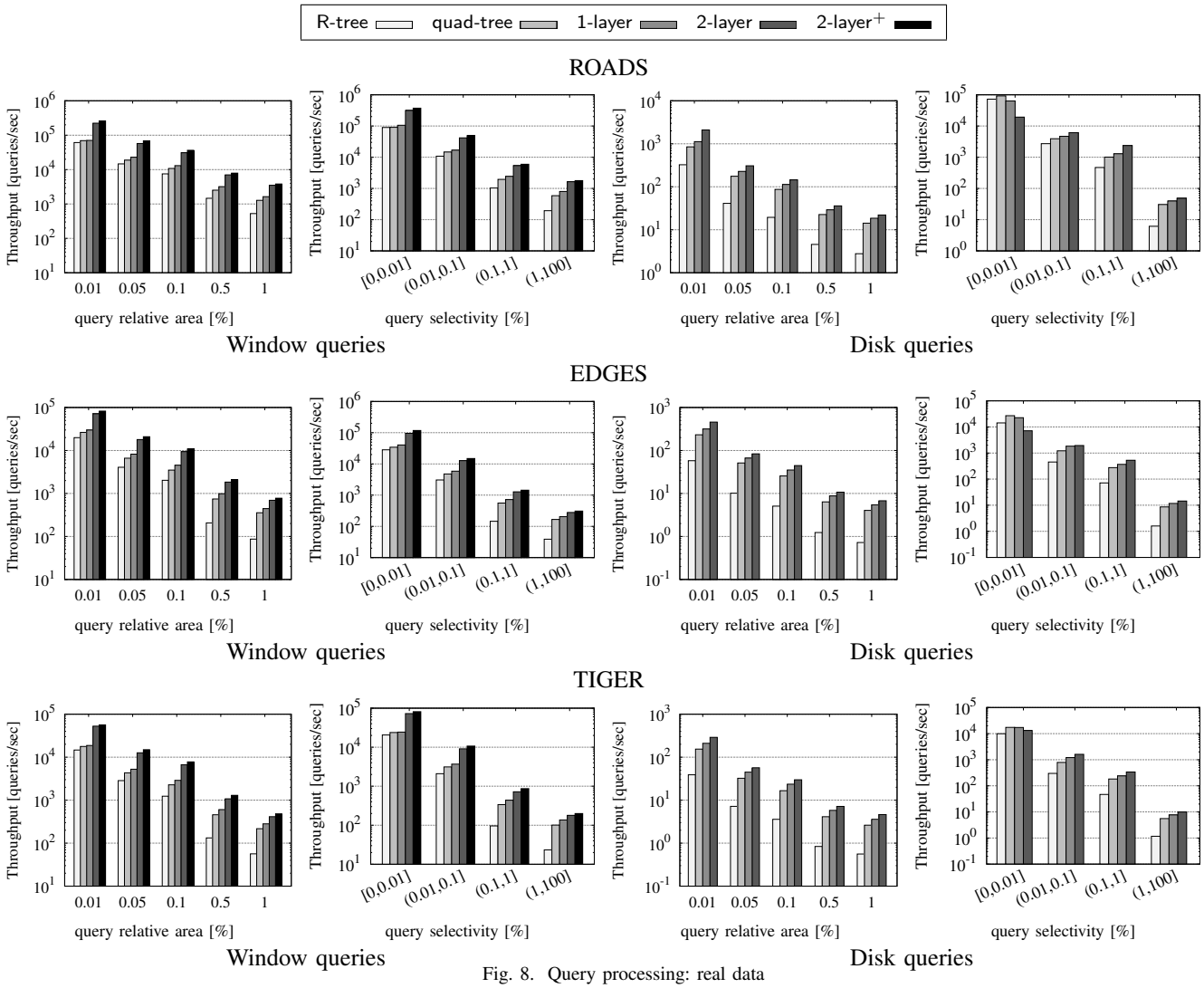



Fig. 8. Query processing: real data

computations. Figure 9 compares all methods for window queries on the synthetic datasets. In these experiments, we additionally vary the database size and the areas of the data objects. In terms of query throughput w.r.t. query area and selectivity, the trends are similar as those for the real data. In addition, the data cardinality does not affect the relative performance of the methods. Finally, we observe that 2-layer and 2-layer$^+$ are more robust to the area of the data objects compared to the competition. As the area grows, the replication to tiles increases, and so 1-layer and the quad-tree have to compute and eliminate more duplicate results. In contrast, 2-layer and 2-layer$^+$, with the help of our secondary partitioning, completely avoid the generation and elimination of duplicate results. On the other hand when the data area shrinks ($10^{-\infty}$ represents the case of extremely small rectangles that resemble points), the replication to tiles decreases. 1-layer and the quad-tree still need to perform the extra comparison of the de-duplication reference test, which explains the stable advantage of 2-layer and 2-layer$^+$.

**Disk range queries**. For disk range queries, we report results on the real data in the last two columns of plots in Figure 8. 2-layer$^+$ is not included in the comparison, because storage decomposition cannot improve distance computations. For disk queries on 1-layer and quad-tree, we cannot use the reference point technique to eliminate duplicate results (and duplicate elimination using hashing is too expensive). Thus, we implemented disk queries on them as follows. We executed a window query using the MBR of the query range and eliminated any duplicates intersecting the window. For all tiles/quadrants inside the disk range, we just reported all window query results there as disk query results. For all other tiles and quadrants we performed distance tests before confirming and reporting the results. The plots show once again the superiority of the 2-layer index.

**Batch and Parallel Query Processing**. Figure 10 compares the two approaches (queries-based and tiles-based), discussed in Section VI, for batch window query processing (10K queries or 1% relative area, per batch) on ROADS and EDGES.[5]

[5]Similar findings are observed for TIGER, but the results are omitted due to lack of space.

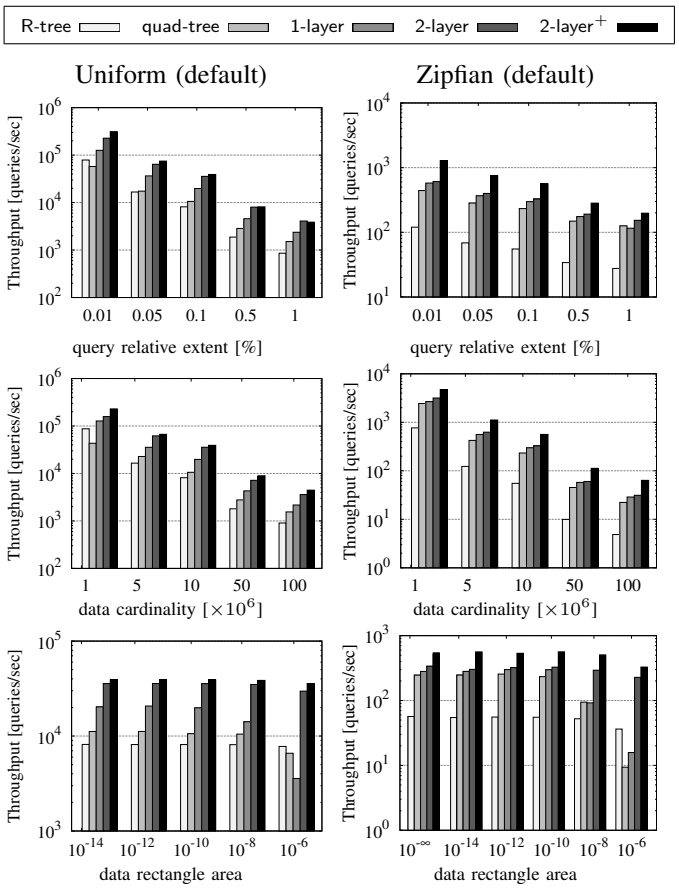



Fig. 9. Query processing: synthetic data (window queries)

A general observation from the plots is that tiles-based is superior to queries-based when the dataset is large (i.e., dense) and the queries are relatively large. In this case, the sizes of the dedicated tables for each class per tile are large and cache conscious tiles-based approach makes a difference. On the other hand, the overhead of finding and accumulating the subtasks per tile does not pay off when the number of queries on each tile is too small or when the tiles do not contain many rectangles. The advantage of tiles-based becomes more prominent in parallel query processing. Figure 11 shows the speedup of batch query evaluation on the two largest datasets (again, 10K queries per batch) as a function of the number of parallel threads. Note that tiles-based scales gracefully with the number of threads (up to about 25 threads, where it starts being affected by hyperthreading). On the other hand, queries-based scales poorly due to the numerous cache misses.

**Updates.** To confirm the superiority of grid indices in updates, we conducted an experiment using the real datasets, where we first constructed the index by loading 90% of the data in batch and then measuring the cost of incrementally inserting the last 10% of the data. Table VI compares the total update costs of the competitor indices. R-tree is two orders of magnitude slower than the baseline 1-layer index and the cost of updates on 2-layer is only a bit higher compared to the update cost on 1-layer. Updates on quad-tree are also slower compared to

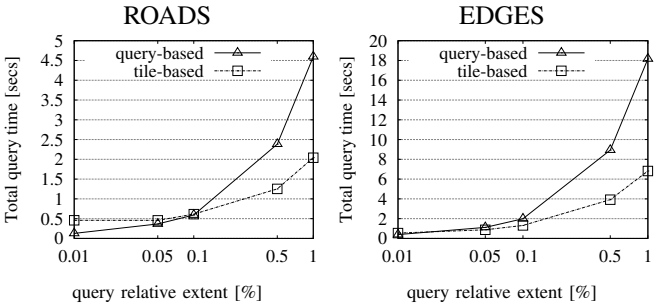



Fig. 10. Batch query processing (window queries)

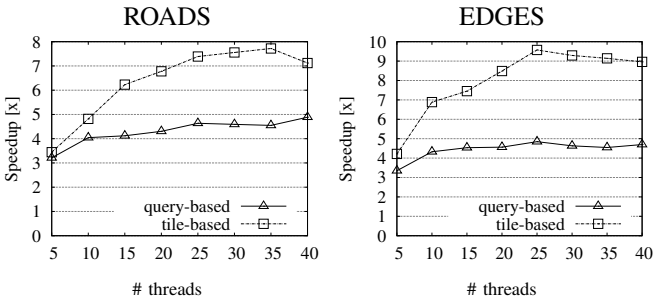



Fig. 11. Batch query parallel processing (window queries)

1-layer and 2-layer, due to the tree traversal.

### D. Comparison with GeoSpark

Finally, we compare our proposed 2-layer grid index with GeoSpark [34], one of the best-performing distributed spatial data management systems according to [24]. Our goal is to show that, for the scale of benchmarking data [10] that we and recent papers [24], [18], [26] use, in-memory indexing in a multi-core processing machine is superior to using a system designed for cluster computing. As our implementation is designed to run on a single machine, we run GeoSpark in client mode, meaning that the driver and Spark applications are both on the same machine. In addition, we used R-tree indexing in GeoSpark, which performs best in query evaluation. We compared GeoSpark with 2-layer that uses a grid granularity of 1000x1000 and tested both single and multi-threaded versions for range queries. The experiments were conducted using the ROADS dataset on a machine with 64 GBs of RAM and a Intel(R) Core i7-4930K CPU clocked at 3.40GHz.[6] For each method, we average the cost of 100 (end-to-end) window queries, where the area of each query is 0.1% of the area of the map. Figure 12 shows that 2-layer always outperforms GeoSpark in terms of query performance by at least three orders of magnitude. These results are consistent with the findings of [24], where distributed spatial data management systems are shown to have a throughput of at most several hundred range queries per minute on data of similar scale. In order to compare the two methods in a multi-threaded scenario on equal terms, our approach evaluates the queries independently (i.e., not in batch). We observe the same trend as the number of cores increases.

[6]We do not own the platform where we ran the previous experiments, so we could not install GeoSpark on that machine.

TABLE VI
TOTAL UPDATE COST (SEC)

| dataset | R-tree | quad-tree | 1-layer | 2-layer |
|---|---|---|---|---|
| ROADS | 5.34 | 0.76 | 0.059 | 0.068 |
| EDGES | 19.8 | 2.89 | 0.267 | 0.382 |
| TIGER | 33.91 | 4.63 | 0.459 | 0.634 |

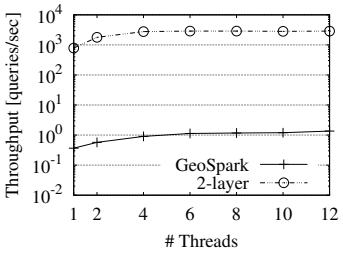



Fig. 12. Window query performance comparison

## VIII. CONCLUSIONS

We presented a secondary partitioning approach that can be applied to SOP indices, such as grids, and divides the MBRs within each spatial partition to four classes. Our approach reduces the number of comparisons during range query evaluation and avoids the generation (and elimination) of duplicate results. In addition, we proposed a secondary filtering technique for spatial range queries, avoids a refinement step for the majority of query results. Finally, we investigated techniques for evaluating numerous range query requests in batch and in parallel. Our experimental findings confirm the superiority of our approach compared to the state-of-the-art duplicate result elimination method [9]. We also show that a grid equipped with our method outperforms other indices (such as the quad-tree and R-tree) by up to one order of magnitude and showed its scalability to multiple query evaluation in parallel. A direction for future work is the application of our approach to distributed spatial data management systems. Moreover, we will study the evaluation of other popular query types, such as nearest neighbor queries and spatial joins using SOP indices that employ our secondary partitioning scheme.

## ACKNOWLEDGMENTS

Partially funded by EU's Horizon 2020 programme (Grant Agreement No. 957345), the European Regional Development Fund - GSRT (project MIS 5002437/3) and the Greek national funds, under the Research-Create-Innovate call (projects: T1EDK-04810 and T2EDK-02848). The authors gratefully acknowledge the computing time on the supercomputer Mogon at Johannes Gutenberg University Mainz (hpc.uni-mainz.de).

## REFERENCES

[1] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz. Hadoop-GIS: A high performance spatial data warehousing system over mapreduce. *Proc. VLDB Endow.*, 6(11):1009–1020, 2013.

[2] W. G. Aref and H. Samet. Hashing by proximity to process duplicates in spatial databases. In *CIKM*, pages 347–354, 1994.

[3] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[5] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.

[6] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on processing spatial data with mapreduce. In *SSDBM*, pages 302–319, 2009.

[7] C. Cheng, H. Yang, I. King, and M. R. Lyu. Fused matrix factorization with geographical and social influence in location-based social networks. In *AAAI*, 2012.

[8] G. P. Copeland and S. Khoshafian. A decomposition storage model. In *SIGMOD*, pages 268–279, 1985.

[9] J. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *ICDE*, pages 535–546, 2000.

[10] A. Eldawy and M. F. Mokbel. SpatialHadoop: A mapreduce framework for spatial data. In *ICDE*, pages 1352–1363, 2015.

[11] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[13] H. Hoppe. Progressive meshes. In *SIGGRAPH*, pages 99–108, 1996.

[14] D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distributed and Parallel Databases*, 15(2):117–135, 2004.

[15] G. Kedem. The quad-cif tree: A data structure for hierarchical on-line algorithms. In *DAC*, pages 352–357, 1982.

[16] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *SIGMOD*, pages 139–150, 2001.

[17] S. T. Leutenegger, J. M. Edgington, and M. A. López. STR: A simple and efficient algorithm for r-tree packing. In *ICDE*, pages 497–506, 1997.

[18] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan. LISA: A learned index structure for spatial data. In *SIGMOD*, pages 2119–2133, 2020.

[19] M. Loskot and A. Wulkiewicz, 2019. https://github.com/mloskot/spatial_index_benchmark.

[20] N. Mamoulis. *Spatial Data Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[21] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, 2004.

[22] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, pages 634–645, 2005.

[23] M. Olma, F. Tauheed, T. Heinis, and A. Ailamaki. BLOCK: efficient execution of spatial range queries in main-memory. In *SSDBM*, pages 15:1–15:12, 2017.

[24] V. Pandey, A. Kipf, T. Neumann, and A. Kemper. How good are modern spatial analytics systems? *Proc. VLDB Endow.*, 11(11):1661–1673, 2018.

[25] M. Pavlovic, D. Sidlauskas, T. Heinis, and A. Ailamaki. QUASII: query-aware spatial incremental index. In *EDBT*, pages 325–336, 2018.

[26] J. Qi, G. Liu, C. S. Jensen, and L. Kulik. Effectively learning spatial indices. *Proc. VLDB Endow.*, 13(11):2341–2354, 2020.

[27] S. Ray, R. Blanco, and A. K. Goel. Supporting location-based services in a main-memory database. In *IEEE MDM*, pages 3–12, 2014.

[28] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[29] D. Sidlauskas, S. Saltenis, C. W. Christiansen, J. M. Johansen, and D. Saulys. Trees or grids?: indexing moving objects in main memory. In *SIGSPATIAL/ACM-GIS*, pages 236–245, 2009.

[30] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[31] H. Wang, X. Fu, J. Xu, and H. Lu. Learned index for spatial queries. In *MDM*, pages 569–574, 2019.

[32] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *SIGMOD*, pages 1071–1085, 2016.

[33] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *CloudDB, ICDE Workshops*, pages 34–41, 2015.

[34] J. Yu, Z. Zhang, and M. Sarwat. Spatial data management in apache spark: the geospark perspective and beyond. *GeoInformatica*, 23(1):37–78, 2019.

[35] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, pages 631–642, 2005.