Efficient Nearest Neighbor Queries on Non-point Data

Achilleas Michalopoulos
University of Ioannina, Greece
amichalopoulos@cse.uoi.gr

Dimitrios Tsitsigkos
University of Ioannina, Greece
dtsitsigkos@cse.uoi.gr

Panagiotis Bouros
Johannes Gutenberg University
Mainz, Germany
bouros@uni-mainz.de

Nikos Mamoulis
University of Ioannina, Greece
nikos@cs.uoi.gr

Manolis Terrovitis
Athena RC, Athens, Greece
mter@imis.athena-innovation.gr

ABSTRACT

Nearest neighbor (NN) queries are ubiquitous in spatial databases, but have been studied mainly for point data. Inspired by recent work on indexing non-point objects for range queries, we propose a secondary partitioning scheme for space-partitioning indices, tailored to NN search. Our scheme classifies the contents of each primary partition into 16 secondary partitions, considering the begin and end of objects with respect to the spatial extent of the primary partition. Based on this, we design algorithms for both incremental NN and k-NN search that avoid duplicate results and skip unnecessary computations. We compare our scheme to the state-of-the-art indexing and find that it has a significant performance advantage.

CCS CONCEPTS

• **Information systems → Geographic information systems**.

KEYWORDS

Spatial indexing, distance queries, non-point data

ACM Reference Format:
Achilleas Michalopoulos, Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2023. Efficient Nearest Neighbor Queries on Non-point Data. In *The 31st International Conference on Advances in Geographic Information Systems (SIGSPATIAL '23), November 13-16, 2023, Hamburg, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3589132.3625609

1 INTRODUCTION

Large collections of non-point data are available in many scientific and application domains, including Geographic Information Systems, graphics [7], medical science [15], and location-based services [2]. Indexing non-point spatial data has been studied for decades and has reached a high level of maturity [11]. However, the data management model has shifted over the years. Up until 1-2 decades ago, the dominant model was to store the data on disk in a single machine and use hierarchical disk-based indices (e.g., the R-tree [5]). Nowadays, with memory chips being much bigger and cheaper, we can store and index large spatial data collections in memory [18]. 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, 4, 14, 19, 20].

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 [8, 12, 13, 16, 17, 21]. Further, query evaluation over SOP partitions can be easily parallelized; thus, SOP is the de facto approach in distributed spatial data management systems [4, 19, 20, 22].

Two-layer partitioning. However, disjoint space partitioning by SOP requires objects that intersect multiple partitions to be replicated. Under this, the same object can be identified as a query result in multiple partitions and so possible duplicates in query results should be eliminated [3, 22]. Recently, Tsitsigkos et al. [18] proposed a *secondary partitioning approach* for the *filter-step* of spatial range queries on non-point objects. This approach divides the object MBRs in 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, so that the result is guaranteed not to have duplicates. Besides avoiding duplicate results, this approach also reduces the number of comparisons in each partition significantly.

However, the aforementioned approach is not appropriate for distance-based queries on non-point objects, e.g., k nearest neighbor (kNN) queries. The problem in the secondary partitioning proposed in [18] is that it 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 [18]) is proposed, whereas kNN queries are not studied at all.

Contributions. We improve upon the secondary partitioning of [18], by defining four classes per dimension (instead of two), which capture more precisely the position of a rectangle w.r.t. all directions. As a result, in the 2D space, we end up with 16 classes of rectangles. This does not bring any overhead in rectangular range queries, as each of the 16 classes can be mapped to one of the original four classes in [18] and the rectilinear range query algorithms can

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGSPATIAL '23, November 13–16, 2023, Hamburg, Germany
© 2023 Association for Computing Machinery.
ACM ISBN 979-8-4007-0168-9/23/11…$15.00
https://doi.org/10.1145/3589132.3625609

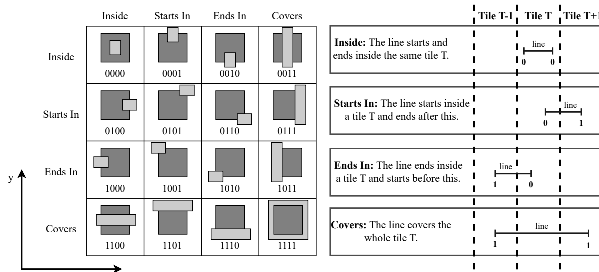


Figure 1: Illustration of class decomposition

readily be applied. In contrast, the definition of the 16 classes brings a significant performance improvement in evaluating NN queries.

We design algorithms for the filter step of NN queries on a grid index which adopts our 16-classes secondary partitioning scheme. For each query, we show which partitions are at each tile, such that duplicates are avoided. We compare this enhanced grid index against the best performing DOP index (the in-memory R-tree from boost.org) using real data and demonstrate its superiority.

2 CLASS DECOMPOSITION

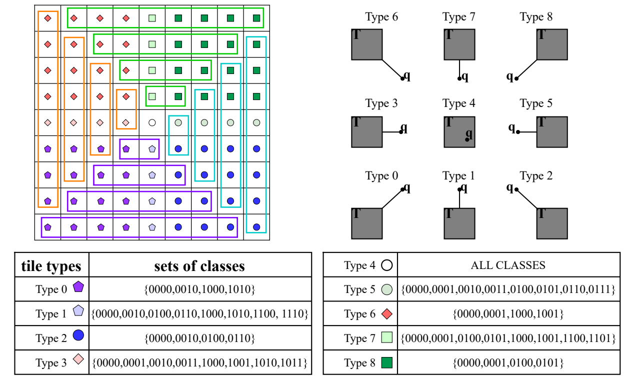
The two-layer partitioning in [18] was mainly designed for rectangular range queries and not for distance-based, such as NN. Given this, we next propose an extension which partitions a tile T (or a SOP partition in general) based on both the *begin* and *end* points of the objects assigned to T. Figure 1 (left) shows the four cases for the projection of an object o w.r.t. the projection of a tile T whereto o is assigned. The object can 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,. The first bit refers to the begin point of the object's projection and the second to the end point of the projection. Bit value 0 denotes that the begin (end) point is inside T; bit value 1 means that the end point is before (after) T.

To encode the divisions of a tile (partition) T in the d-dimensional space, we need $2 \cdot d$ bits, i.e., two per dimension, since we have d projections of T (and the objects); this results into 2^{2d} partitions. So, in the 2D space, we have $2^4 = 16$ classes of objects (i.e., secondary partitions of a tile T), as shown in Figure 1 (right). In this figure, a tile T is denoted by the darkgrey square and each of the 16 examples show a case of an object o (lightgrey rectangle), which belongs to each of the 16 divisions. For example, the upper-left corner shows an object 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, etc.

Given the above classification, our proposed indexing scheme partitions the objects (object MBRs) using a 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 into classes (i.e., 16 classes in the 2D space).

3 NEAREST NEIGHBOR QUERIES

We now discuss how to efficiently evaluate NN queries, without producing duplicate results. In the following, we assume that the query object q is a point, but our NN methods can be applied also for non-point query objects, in a straightforward manner. Our methods work with all SOP indices (e.g., arbitrary grids, quad-trees, k-d-trees, etc.), but for illustration purposes, we assume the input



tile types	sets of classes
Type 0	{0000,0010,1000,1010}
Type 1	{0000,0010,0100,0110,1000,1010,1100, 1110}
Type 2	{0000,0010,0100,0110}
Type 3	{0000,0001,0010,0011,1000,1001,1010,1011}

tile types	sets of classes
Type 4	ALL CLASSES
Type 5	{0000,0001,0010,0011,0100,0101,0110,0111}
Type 6	{0000,0001,1000,1001}
Type 7	{0000,0001,0100,0101,1000,1001,1100,1101}
Type 8	{0000,0001,0100,0101}

Figure 2: Tile types; grouping and distribution in space

set R of objects (i.e., object MBRs) is indexed by a regular grid. 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 in Section 2.

Incremental NN search (distance browsing in [6]). In this case, the number k of desired NNs is not specified a priori. The user retrieves the objects in increasing distance from the query object q. Our incremental NN algorithm extends [13], which targeted only point data. In a nutshell, the grid tiles are divided to groups based on their orientation w.r.t. tile T_q, which includes q, as shown by the oblongs in Figure 2 (top-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, etc. In general, the n-th 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.

Our incremental NN algorithm (Algorithm 1) initializes a priority queue Q (minheap) with all objects in T_q. Q is organized in increasing order of the *minimum* Euclidean distance to q. Naturally, 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 the 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 element c in 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. Last (Lines 11-14), if c is a tile, for *some* of the classes, all their objects in c are en-heaped to Q.

Figure 2 (top-right) shows how to define the 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 cell types, different classes are considered as shown at the bottom of the same figure. For example, if tile c is of type 6, only its objects in classes {0000,0001,1000,1001} are en-heaped. This is because the objects in other classes are also included in neighboring tiles which are *closer* to q, and hence these objects have been added to Q earlier. Re-adding these objects to Q will produce duplicate results. Class selection for each tile c based on the relative direction of c w.r.t. T_q is established by the following:

ALGORITHM 1: Incremental NN Search

```
Input  : query point q, grid index T of rectangle set R
Output : next NN of q in R
1  T_q ← tile of T that contains q;
2  G ← NeighboringGroups(T_q);
3  Q ← new Min-Heap; add to Q (i) all rectangles of T_q and (ii) all g ∈ G;
4  while Q is not empty do
5      c ← Q.pop();                    // c is next nearest heap element to q
6      if c is a group of tiles then
7          g ← NextLevelGroup(c);
8          Q.push(g);
9          foreach tile T_c ∈ 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 ∈ rectArr do
14             Q.push(r);
15     else // c is an object
16         output c;                                   // c is the next NN
```

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, all classes in c with $E = 1$ ($B = 1$) should not be en-heaped to Q (Lines 12-14, Algorithm 1) to avoid duplicates.*

Proof. If a class in tile c has $E = 1$ ($B = 1$), all the rectangles in the class also appear in tile c_{next} that follows (precedes) c in dimension d. But, as c is before (after) T_q in dimension d, c_{next} is guaranteed to be closer to q than c, which means the rectangles will be already accessed when c_{next} is processed. So, adding the rectangles of this class to Q would produce duplicate results. □

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

k-NN search. Knowing the number k of desired neighbors allows us to prune objects and tiles, and to control the size of the priority queue Q. For this, we maintain maxheap H with the k-NN objects so far. The top $H.top$ of H holds the object with the k-th distance to q found so far. Distance $H.top.dist$ between $H.top$ and q can be used as a bound for pruning. Hence, Q contains only tiles and tile-groups with a minimum distance to q lower or equal to $H.top.dist$. This greatly reduces the size of Q and may help to compute the k-NN result faster than 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.

ALGORITHM 2: k-NN Search

```
Input  : query point q, grid index T of rectangle set R
Output : k-NNs of q in R
1  T_q ← tile of T that contains q;
2  G ← NeighboringGroups(T_q);
3  H ← create k-maxheap and push to H all rectangles of T_q;
4  foreach tile-group g ∈ G do
5      if |H| < k or dist(g, q) < H.top.dist then
6          Q.push(g);
7  while Q is not empty and (|H| < k or Q.top.dist < H.top.dist) do
8      c ← Q.pop();
9      if c is a group of tiles then
10         g ← NextLevelGroup(c);
11         if |H| < k or dist(g, q) < H.top.dist then
12             Q.push(g);
13         foreach tile T_c ∈ 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 ∈ rectArr do
19             if |H| < k or dist(r, q) < H.top.dist then
20                 H.push(r);
21 return H;
```

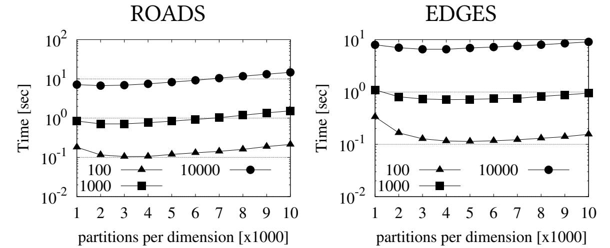


Figure 3: Determining best grid granularity: incremental NN search for 100, 1000 and 10000 browsing neighbors

4 EXPERIMENTAL EVALUATION

We conducted our analysis on a Intel(R) Core(TM) i9-10900K CPU, clocked at 3.70GHz with 32 GB of RAM, running Ubuntu Linux 20.04.1. All indices were implemented in C++, compiled using gcc (v9.4.0) with flags -O3, -mavx, and -march=native.

Setup. We used the ROADS and EDGES Tiger 2015 datasets [4].[1] ROADS contains 19M linestrings occupying 538MB; the projection of an object covers 0.008% of the x-axis and 0.015% of the y-axis, on average. EDGES contains 69M polygons occupying 1.6GB; each object covers 0.004% of the x-axis and 0.007% of the y-axis. The object coordinates in both datasets were normalized inside $[0, 1]$.

We implemented our secondary partitioning over a main-memory regular grid; we denote our scheme as 2-layer16. As a competitor, we considered the state-of-the-art DOP index: an in-memory STR-bulkloaded [9] R-tree, from the Boost Geometry library (boost.org).[2] The nodes of the constructed trees have a capacity of 16; this configuration was reported to perform the best and confirmed through testing. To assess the performance, we measure the average execution time of 10K queries over 10 runs, while varying the number of neighbors. For incremental NN search, the neighbors are progressively determined (Section 3); in k-NN search, we set k accordingly.

Index tuning. We first study 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, 1000, and 10000 neighbors. Figure 3 reports the total time for 10K 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. Under this, we set the number of partitions per dimension to 3000 for ROADS and to 4000 for EDGES, where the best performance was observed.

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

[2] The benchmarks in [10] showed the Boost.Geometry R-tree outperforms the one in libspatialindex.org. Also, the analysis in [18] showed that R-tree is the most efficient DOP competitor, outperforming R*-tree (both from the Boost library).

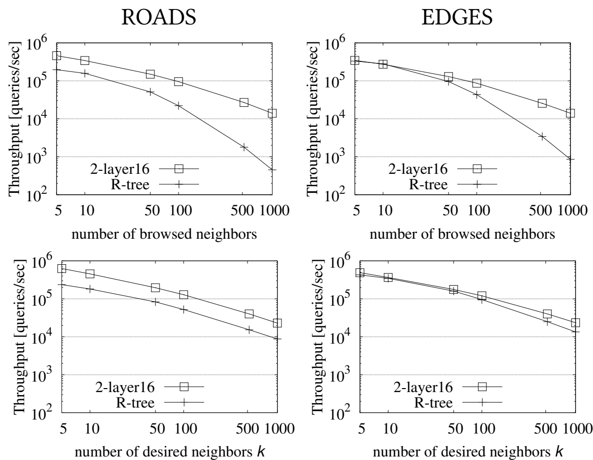


Figure 4: Incremental (top) and k-NN search (bottom)

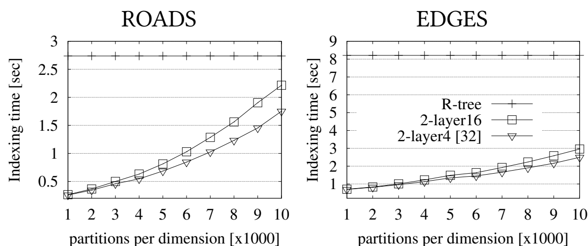


Figure 5: Indexing cost

Query processing. Figure 4 compares the throughput of our 2-layer16 partitioning against the R-tree, again for ROADS and EDGES. The plots at the top row report on incremental search while at the bottom, on k-NN. 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 pipe-lining a large number of NNs to the next operator. As another observation, the k-NN 2-layer16 algorithm is slightly faster than the incremental NN one. As we discussed in Section 3, knowing the number of neighbors in advance allows the k-NN algorithm to prune objects.

Indexing and maintenance. Figure 5 compares the indexing time of our partitioning scheme 2-layer16 against the scheme in [18], which we denote by 2-layer4. 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; 18% on average. Nevertheless, the building cost of 2-layer16 is significantly lower than that of the R-tree, especially for the default 3K-4K partitions and for the larger EDGES dataset. 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. Last, regarding index maintenance, we expect a similar trend to index building. Based on the analysis in [18], we expect 2-layer16 to outperform the R-tree for updates but exhibit slightly higher time compared to 2-layer4, due to maintaining more classes.

5 CONCLUSIONS

We presented a secondary partitioning technique for space-oriented partitioning indices that divides 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 NN search (both incremental and k-NN) that take advantage of our technique to compute query results efficiently and without producing duplicates. Our evaluation on real datasets confirms the superiority of our scheme compared to previous work. In the future, we plan to investigate more advanced distance-based queries, such as ϵ-distance joins, closest-pair queries and iceberg distance joins.

ACKNOWLEDGMENTS

Funded by the Hellenic Foundation for Research and Innovation (HFRI) under the "2nd Call for HFRI Research Projects to support Faculty Members & Researchers" (Project No. 2757) and EU's Horizon 2020 programme, MORE project (Grant Agreement No. 957345)

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. *Proc. VLDB Endow.* 6, 11 (2013), 1009–1020.
[2] 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 *AAAI*.
[3] Jens-Peter Dittrich and Bernhard Seeger. 2000. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *IEEE ICDE*. 535–546.
[4] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *IEEE ICDE*. 1352–1363.
[5] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *ACM SIGMOD*. 47–57.
[6] Gísli R. Hjaltason and Hanan Samet. 1999. Distance Browsing in Spatial Databases. *ACM Trans. Database Syst.* 24, 2 (1999), 265–318.
[7] Hugues Hoppe. 1996. Progressive Meshes. In *ACM SIGGRAPH*. 99–108.
[8] Dmitri V. Kalashnikov, Sunil Prabhakar, and Susanne E. Hambrusch. 2004. Main Memory Evaluation of Monitoring Queries Over Moving Objects. *Distributed and Parallel Databases* 15, 2 (2004), 117–135.
[9] Scott T. Leutenegger, J. M. Edgington, and Mario A. López. 1997. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *IEEE ICDE*. 497–506.
[10] Mateusz Loskot and Adam Wulkiewicz. 2019. https://github.com/mloskot/spatial_index_benchmark.
[11] Nikos Mamoulis. 2011. *Spatial Data Management*. Morgan & Claypool Publishers.
[12] Mohamed F. Mokbel, Xiaopeng Xiong, and Walid G. Aref. 2004. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *ACM SIGMOD*. 623–634.
[13] Kyriakos Mouratidis, Marios Hadjieleftheriou, and Dimitris Papadias. 2005. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *ACM SIGMOD*. 634–645.
[14] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How Good Are Modern Spatial Analytics Systems? *Proc. VLDB Endow.* 11, 11 (2018), 1661–1673.
[15] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2018. QUASII: QUery-Aware Spatial Incremental Index. In *EDBT*. 325–336.
[16] Suprio Ray, Rolando Blanco, and Anil K. Goel. 2014. Supporting Location-Based Services in a Main-Memory Database. In *IEEE MDM*. 3–12.
[17] Darius Sidlauskas, Simonas Saltenis, Christian W. Christiansen, Jan M. Johansen, and Donatas Saulys. 2009. Trees or grids?: indexing moving objects in main memory. In *SIGSPATIAL/ACM-GIS*. 236–245.
[18] Dimitrios Tsitsigkos, Konstantinos Lampropoulos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2021. A Two-layer Partitioning for Non-point Spatial Data. In *IEEE ICDE*. 1787–1798.
[19] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In *ACM SIGMOD*. 1071–1085.
[20] 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.
[21] Xiaohui Yu, Ken Q. Pu, and Nick Koudas. 2005. Monitoring K-Nearest Neighbor Queries Over Moving Objects. In *IEEE ICDE*. 631–642.
[22] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. 2009. SJMR: Parallelizing spatial join with MapReduce on clusters. In *CLUSTER*. 1–8.