A Survey on Spatial Indexes and Range Queries under Constraints

Ka Lok MO
HKUST
Hong Kong
klmoaa@connect.ust.hk

Abstract

Spatial indexes are the foundation of spatial databases. In this survey, we review several fundamental spatial indexes and one recent work on the emerging learned spatial indexes. The second half of the survey discusses range query processing with spatial relations and obstacles. This survey aims to serve as an entry point for researchers with little background but a strong interest in spatial indexes and spatial query processing.

1 Introduction

Spatial databases are used in a wide array of applications, such as Geographic Information System (GIS), computer-aided design (CAD), public health, and more. To store and retrieve spatial data efficiently, a substantial number of spatial indexes have been proposed and developed throughout the past few decades, and the topic remains a hot research area today among the spatial database community. In addition, spatial databases are capable of processing queries involving spatial objects, such as points, lines, and polygons, and spatial relations. Spatial queries often require to take different constraints into account, making the processing of queries more complex.

This survey is organized as follows: In Section 2, we first review some of the traditional spatial indexes in detail, followed by covering one of the novel learned spatial indexes. In Section 3, we put an emphasis on range query processing involving spatial relations and obstacles. We conclude the survey in Section 4.

2 Spatial Indexing

Since spatial data is typically two-dimensional or three-dimensional, and can even incorporate temporal information in spatial-temporal databases, multidimensional access methods with hierarchical structures are commonly employed in spatial indexes [1]. The three key performance requirements for indexes are query performance (fast query processing), storage efficiency (compact data representation), and update efficiency (efficient updates for dynamic data).

Spatial indexes can be classified in various ways:

(1) Firstly, spatial indexes can be categorized into point access methods and spatial access methods [2]. While point access methods are capable of handling multi-dimensional point data, spatial access methods are utilized to manage extended objects such as lines and polygons.
(2) Another dimension of comparison is space partitioning and data partitioning. The former builds the tree by dividing the entire space into non-overlapping partitions at the root level and recursively subdividing each partition into lower levels. The latter builds the tree by grouping nearby spatial objects into clusters, which may potentially overlap with other clusters. A hybrid approach that combines both techniques is also possible [3].
(3) Spatial indexes can be further distinguished in terms of in-memory indexes and disk-based indexes. In [4], the authors also surveyed newer spatial indexes specifically designed for modern storage devices, including Solid State Disks.
(4) Furthermore, spatial indexes can be grouped into traditional indexes and learned indexes. With the advancement of machine learning, learned spatial indexes have gained considerable interest in the research community [5]. Compared with conventional indexes, learned indexes enable faster query processing and smaller space consumption for internal and data nodes. Typically, a learned index replaces comparison-based searches within nodes with the inference of linear models or artificial neural networks (ANN), serving as a function approximator to sub-partition space, and selects the child node [6, 7]. Meanwhile, there are also learned indexes proposed without employing these models in intermediate nodes, but instead using deep neural networks as an auxiliary function estimator for index construction.

This section surveys the following spatial indexes: Quad-Tree, KD-Tree, R-Tree, and RLR-Tree. Table 1 presents the classification of these four spatial indexes.

2.1 Quad-Tree

In [8], the Quad-Tree, often known as the Point Quad-Tree, maintains a hierarchical structure where each node subdivides its space into four smaller quadrants. In this approach, each node stores a record and can have up to four child nodes corresponding to the four quadrants: NE (quadrant one), NW (quadrant two), SW (quadrant three), and SE (quadrant four), as shown in Figures 1 and 2. By convention, quadrants one and three are considered closed and quadrants two and four

Table 1. Classification of the Surveyed Spatial Indexes

Index	Spatial Data Type	Partitioning Approach	Type of Storage Device	Learned Index
Point Quad-Tree [8]	Point Access Method	Space-partitioned	In-memory	No
KD-Tree [9]	Point Access Method	Space-partitioned	In-memory	No
R-Tree [10]	Spatial Access Method	Data-partitioned	Disk-based	No
RLR-Tree [7]	Spatial Access Method	Data-partitioned	Disk-based	Yes

are considered open. This means that points lying on the quadrant line will be classified into the closed quadrants. Additionally, to handle data point collisions, a pointer to a linear list of records can be used to replace a single record in the node.

When performing a range query in the Quad-Tree, the search is conducted recursively starting from the root node. The algorithm first checks if the current node falls within the query range and reports the corresponding record if it does. Then, it iteratively evaluates the four quadrants of the current node. If any quadrant overlaps with the query region and has a defined child node, the algorithm recursively calls itself with that child node. The search continues until all quadrants have been evaluated.

It is worth noting that the subdivision of the plane in the Quad-Tree is determined by the sequence of point insertions into the tree, which also affects the tree's height. In the worst case, the order of insertion can lead to an unbalanced tree structure, resembling a linear list. To address this issue, an optimized version of the bulk-loading algorithm is suggested in [8] for static datasets. The dataset is sorted in a lexicographical order primarily by the x-coordinate and secondarily by the y-coordinate, in the two-dimensional case. The median of the sorted collection is chosen as the root node, and the sorted list is divided into two halves. The algorithm is invoked recursively on each half, resulting in a tree with a maximum path length of $\lfloor \log_2(n) \rfloor$, where n is the number of nodes. The algorithm has a time complexity of $O(n \log n)$.

In a comprehensive survey of Quad-Trees [11], the author further classified Quad-Trees variants based on the type of data they supported. For example, the Region Quad-Tree is designed for region data, the Matrix (MX) Quad-Tree, Point-Region (PR) Quad-Tree, and Point Quad-Tree are tailored for point data, and so on. The author also provided practical applications of different representations in various domains, including GIS, image processing, computer graphics, computer vision, and more.

2.2 KD-Tree

The KD-Tree partitions the search space in a hierarchical structure and is designed to fit in main memory. In the original paper [9], the author proposed the homogeneous KD-Tree, which is a multi-dimensional binary search tree, as depicted in Figures 3 and 4. In this version, each record in a

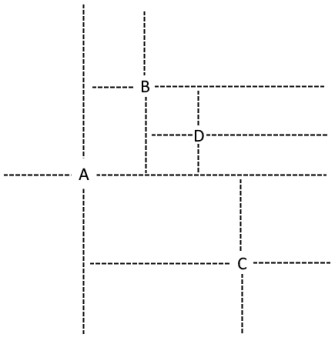

Figure 1. Point Quad-Tree: Space View

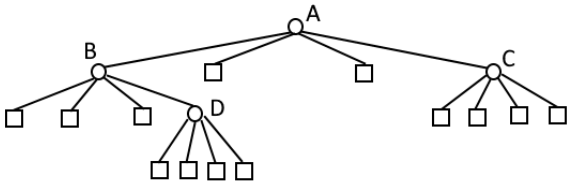

Figure 2. Point Quad-Tree: Tree View

data file resides in a node, whether it is a leaf or a non-leaf, within the tree. Each node, denoted as x, contains k keys (K_1 to K_k), corresponding to k dimensions. In addition to the list of keys, x contains optional data fields, two child pointers to the left and right subtrees, and a discriminator j, where j is just an integer between 1 to k, inclusive. For each node y under the left subtree of the node x, the j-th key K_j of y is less than x's j-th key. Similarly, in the right subtree of node x, all nodes have j-th key values greater than x's j-th key.

In a subsequent paper [12], the author also introduced the non-homogeneous KD-Tree. The internal nodes in the non-homogeneous version contain only the discriminator j, the discriminator key value K, and two child pointers. All records in this tree are stored in leaf nodes.

There are several variations in the choice of discriminator at each node. The original version, proposed by the author, is the cyclic method [9], where the discriminator is chosen in a round-robin manner: At each level i of the tree, the discriminator j is determined by $j = (i \mod k) + 1$. Another

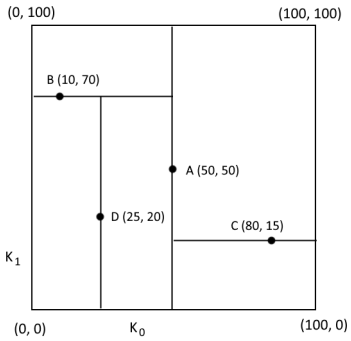

Figure 3. Homogeneous KD-Tree: Space View

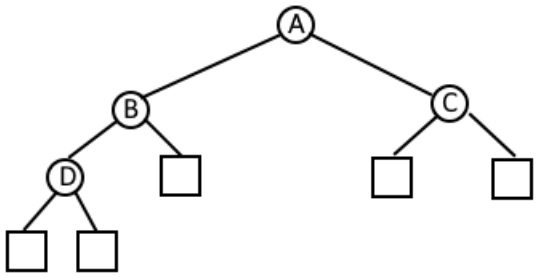

Figure 4. Homogeneous KD-Tree: Tree View

approach for choosing the discriminator j is based on the spread of key values in the subtree [12]. The spread of key values in each dimension can be measured by variance or range of values ($maximum - minimum$), and the goal is to choose j with the greatest spread. Under this selection criteria, the partitioning value is typically chosen as the median of the key values in that dimension. The goal of this scheme is to achieve balanced partitions, where subdivisions evenly distribute the data points. In contrast, if the dimension with the smallest spread is picked, data points are concentrated in a narrow range, and it tends to result in heavily skewed partitions.

2.3 R-Tree

The R-Tree index structure [10] partitions spatial objects into layers of nodes, with each node corresponding to a disk page. It is a height-balanced tree where the leaf nodes contain pointers to relation records of spatial objects. Every spatial object, such as a point, line, and polygon, is characterized by a minimum bounding rectangle (MBR) fully containing itself. The children in non-leaf nodes are represented by tuples. Each tuple contains a pointer to the lower level node and an MBR covering all the MBRs in that node. Figures 5 and 6 illustrate the structure of an R-Tree.

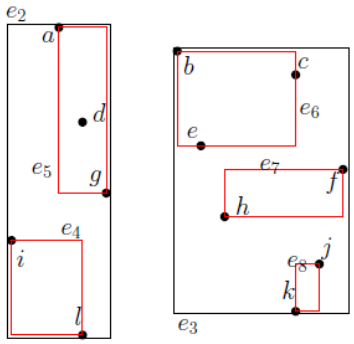

Figure 5. R-Tree: Space View

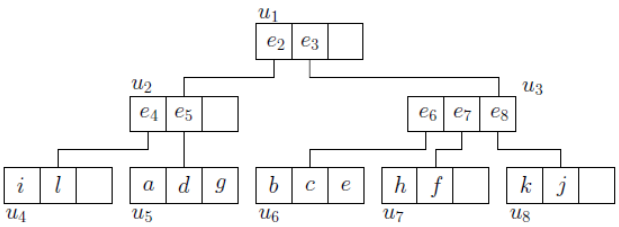

Figure 6. R-Tree: Tree View

In an R-Tree, every node, except the root, must satisfy the following space utilization requirement: Given that M is the maximum number of entries which can fit into a node, a hyperparameter $m \leq M/2$ specifies the minimum number of entries. Then, the worst-case space utilization of a node except the root will be m/M, and the R-Tree indexing N spatial objects will have a height bounded by $O(\log_m(N))$.

For querying in an R-Tree, a two-step approach is employed: a filter step followed by a refinement step. To perform a range query, the search begins at the root node and sequentially evaluates its children's MBRs. If the child's MBR overlaps with the search region, the algorithm will be invoked recursively in the corresponding subtree. When the query reaches a leaf level node, all the index entries whose MBRs overlap with the desired range will be retrieved. These retrieved records, which could include 'false-hits', undergo evaluation using computer geometry techniques in the refinement step. Owing to the data partitioning nature of R-Trees, it is possible for multiple MBRs in the same level to overlap, impacting the number of subtrees traversed and the query efficiency.

2.4 RLR-Tree

The RLR-Tree is a novel reinforcement learning-based R-Tree proposed to enhance the query performance of the classic R-Tree without making any modifications to the structure or query processing algorithms of the classic R-Tree [7], as shown in Figure 7. The author employed the deep Q-learning framework for training a *ChooseSubtree* policy and a *Split*

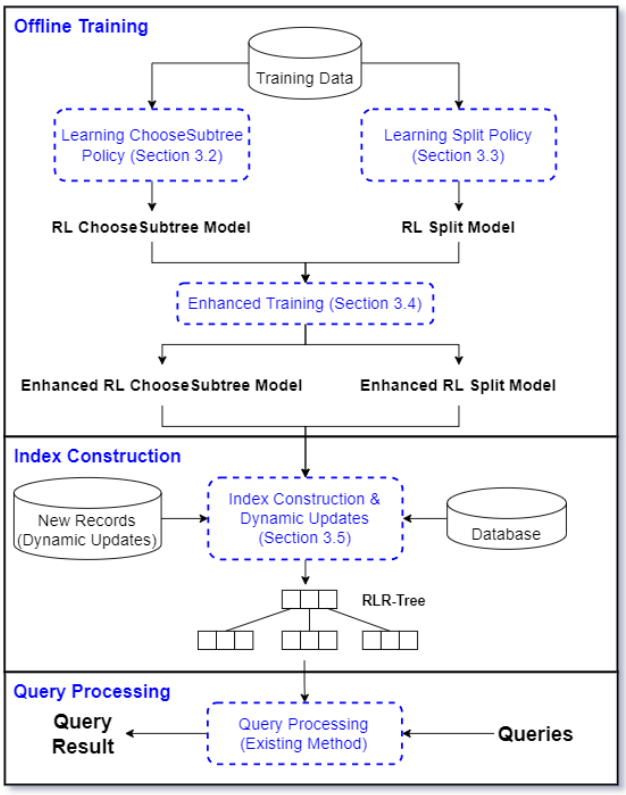

Figure 7. RLR-Tree: Pipeline [7]

policy, which are then utilized for R-Tree index construction and dynamic updates, eliminating the need for heuristic rules used in R-Tree and its variants. Unlike learned indexes like [5], RLR-Tree does not replace the underlying R-Tree structure and can leverage existing algorithms for query processing. Furthermore, the reinforcement learning models can be trained on small datasets with a single distribution and applied to construct trees on larger-scale datasets with varying distributions. The author also claimed the RLR-Tree can achieve at most 95% better query performance than the R-Tree and other variants designed for dynamic data.

In the deep-Q learning framework, the Markov Decision Process (MDP), composed of states, actions, transitions, and rewards, is formulated to enable the model to learn an effective policy.

In the MDP, a state represents the relevant features of the current environment for decision making. The action space defines the list of possible actions (decisions) that can be taken at each step. Transitions refer to the changes in state caused by the selected actions. Rewards indicate the gains obtained from transitioning to a new state and are used to drive the agent towards actions that maximize the expected cumulative reward, where they are usually determined by the quality of action taken (the larger the reward, the better the action taken).

Specifically, at a given node, the *ChooseSubtree* policy aims to choose the best subtree for inserting a new object. Its MDP is modeled as follows:

(1) **State space.** The training framework computes four metrics for a child node N: the increase in MBR area $\Delta Area(N, o)$ after inserting the new object o, the increase in MBR perimeter $\Delta Peri(N, o)$, the increase of overlap between N and other child nodes $\Delta Ovlp(N, o)$, and the occupancy rate of the child node $OR(N)$. Based on empirical findings, to optimize the performance of the policy, the framework first identifies the top-k child nodes in ascending order of $\Delta Area(N, o)$. Then, for each child, the framework computes the four aforementioned metrics, and it concatenates the metrics into a $4 * k$-dimensional tuple to represent the current state S.

(2) **Action space.** Based on empirical findings, the action space is defined as $A = 1, \ldots, k$, where each element i in the set A represents the action of inserting the new object o in the i-th child node. These k child nodes are exactly the same as the top-k child nodes we discussed above in the state space.

(3) **Transition.** Given the current state S and a selected action $i \in A$, the agent transitions to the selected child node i. In the case where the child node i is a leaf node, the agent is said to reach the terminal state.

(4) **Reward.** In the high level, the framework maintains a reference tree, which can be any R-Tree variants serving as a competitor, with a fixed *ChooseSubtree* and *Split* strategy. Meanwhile, the RLR-Tree shares the same *Split* strategy as the reference tree but differs in the *ChooseSubtree* strategy, which is being trained. The reward r is computed based on the difference in I/O costs for processing random queries between the RLR-Tree and the reference tree.
Next, during each iteration, p new objects will be inserted into both trees. p range queries, whose centers are the p objects, are then generated and processed on both trees. The key metric, normalized node access rate defined as $\frac{\#\text{nodes accessed}}{\text{Tree height}}$, is computed for both trees. The reward is then calculated as $r = R' - R$, where R and R' represent the access rates of the RLR-Tree and reference tree respectively. The higher the reward r, the larger the difference in I/O costs between the two trees, and the better query performance RLR-Tree has compared with the reference tree. After that, all transitions, encountered in the insertion of the aforementioned p objects, will be rewarded the same reward r. The last step is the most important: The reference tree will be synchronized with the RLR-Tree periodically, to avoid the influence of previous actions.

The next policy *Split* aims to determine the best way to split an overflowing node containing $M + 1$ entries. Its MDP is modeled as follows:

(1) **State space.** Following a similar idea from R*-Tree [13], the framework first sorts the $M + 1$ entries with respect to their projection for each dimension. For each sorted sequence, consider splitting at the i-th entry, the first i elements will be allocated into the first node, and the remaining $M + 1 - i$ elements will be allocated into the second node. To satisfy the space utilization requirement in R-Tree, a constraint $m \leq i \leq M + 1 - m$ is imposed to ensure that both nodes contain at least m entries. This reduces the number of possible splits to be considered to $M + 2 - 2 * m$ per sorted sequence. The framework then disregards all the splits that would result in creating two overlapping nodes. Next, the framework identifies the top-k splits in ascending order of the total area (of the two nodes created by the split) and compute the four state metrics for each split. The four state metrics are the areas and the perimeters of the two resulting nodes, and a $4 * k$-dimensional tuple is used to represent the current state.
(2) **Action space.** Similar to *ChooseSubtree*, the action space is defined as $A = 1, 2, \ldots, k$, where $i \in A$ represents the i-th split considered above in the state space. Choosing i means adopting the i-th split.
(3) **Transition.** Given the current state S and an action $i \in A$, the agent transitions to the next state, which represents the parent node of the current node. If the parent node is not overflowing, the agent is said to be reaching the terminal state, and no splitting is required in that node.
(4) **Reward.** Similar to *ChooseSubtree*, a reference tree is maintained, and the framework uses the same reward, which is the difference in normalized node access rate. The only difference is that now the RLR-Tree will have the same *ChooseSubtree* strategy as the reference tree but employs a different *Split* policy to be trained. The reference tree will also be synchronized periodically.

Since the underlying index structure is R-Tree, RLR-Tree can be categorized as a data-partitioning, disk-based, spatial access method.

3 Spatial Query Processing

In this section, we will first discuss the three major types of spatial relations commonly used in geographic applications, followed by range queries involving these relations. In the last part, we will cover range queries involving obstacles in a spatial database.

3.1 Spatial Relations

The three primary types of spatial relations used in geographical applications include topological relations, direction relations, and distance relations.

(1) **Topological relations.** Topological relations describe the relative positions between two spatial objects. In [14], the author proposed the four-intersection model, characterizing four intersections of the boundaries and interiors of two sets. The resulting set of eight semantics of region relations include: disjoint, touch, equal, inside, covered by, contain, cover, and overlap.
(2) **Direction relations.** For simplicity, this survey only considers absolute direction. Such direction relations take the cardinal direction of a referenced object, such as North, East, South, West, and more, into account.
(3) **Distance relations.** Distance relations specify concepts including near, far, and about [15]. For example, two objects are said to be 'near' if the object-object distance is within a given threshold.

3.2 Range Queries Involving Spatial Relations

[15] suggested a framework of mapping spatial queries into range queries using B-Tree and R-Tree. The paper provides a rigorous definition of a subset of the three major relations, based on the coordinates of the MBRs of the two objects. It is worth noting that, in the case of R-Tree, the author pointed out that spatial relations between MBRs do not necessarily equate the spatial relations between the bounded spatial objects. For one, if the primary object meets the referenced objects, the MBRs of the two objects can be related by any topological relations except disjoint, such as touch, equal, inside, and so on. The author also discusses how enforcing range constraints on the MBR endpoints may or may not be sufficient for retrieving the desired spatial objects.

To simplify the problem, at the leaf level, the author defined the constraints on the endpoints of MBRs for the retrieval of spatial objects, for a subset of spatial relations. These constraints serve as a preliminary filter, where MBRs failing to satisfy the constraints will be eliminated. Followed by that, in the refinement step, the set of candidate objects passing the filter step will be processed using computational geometry techniques. Any 'false hits' will be removed in this stage. The same logic can be generalized to intermediate nodes containing these MBRs.

In the high level, the query processing framework can be decomposed into three major steps:

(1) Firstly, for non-leaf nodes, nodes that cannot enclose the MBRs of potential 'hits' will be eliminated at the first place, based on a set of constraints enforced on the endpoints of the intermediate nodes' MBRs.
(2) Next, in the leaf node, only the MBR which potentially enclose the 'hits' will be selected. This is again

determined by a set of predefined constraints on the endpoints of the leaf nodes' MBRs.

(3) As the last step, a refinement step is performed to remove all 'false hits'.

3.3 Range Queries Involving Obstacles

To answer the query 'find all objects within r meters from a point q in the presence of obstacles', one has to consider the obstructed distance, rather than the Euclidean distance, between the query point q and the candidate object. The problem can be formulated as a shortest path problem with the aid of a visibility graph [16, 17], followed by evaluating the obstructed distance of candidate objects from q and remove the 'false hits'.

To reduce the I/O costs, at the beginning, the search space can be pruned by eliminating the candidate objects and obstacles which do not intersect with the query region, because the Euclidean distance of these objects from the query point q already exceeds r meters, which imply the obstructed distance will also exceed r, based on the Euclidean lower-bound property [18].

To construct a visibility graph $G = (V, E)$, which stores the connectivity information of mutually visible nodes (pairs of nodes which are not blocked by obstacles), we first define the set of vertices V as the collection of the query point q, the set of candidates (assumed to be point objects) within the query range P', and vertices of the set of obstacles O' intersecting with the query range. Next, to obtain the set of edges E, the 'plane-sweeping' technique is applied iteratively on every node $u \in V$. For each node u, the 'plane-sweeping' algorithm traverses through the set of vertices V in an anti-clockwise manner and stores a set of blocking edges it currently swept through in a priority queue H. If the sweeping line passes through a node $v_i \in V$, the node v_i is said to be visible to the target node u, if the Euclidean distance between u and v_i is less than or equal to the distance between u and the closest edge in the H, and the edge (u, v_i) does not overlap with any obstacle $o \in O'$. If the node v_i is visible, the edge (u, v_i) will be added to the result set.

Lastly, the obstructed distances between q and candidate objects are evaluated on the visibility graph. This can be accomplished easily using a traversal method similar to Dijkstra's algorithm [19]. The resulting candidates are those whose obstructed distances are within r.

4 Conclusion and Future Work

The survey introduces several traditional spatial indexes that forms the basis of many state-of-the-art indexes and covers an example of the emerging learned indexes. The processing of range queries involving spatial relations and obstacles are also discussed in the second half of the paper. In terms of future work, the following topics are worth exploring:

The R-Tree Family. The constantly evolving family of R-Tree has accommodated to become efficient in bulk-loading and dynamic updating. Further investigation on the evolution of different variants of R-Tree, such as the R*-Tree [13], is undeniably beneficial for readers to gain a more thorough understanding on spatial indexing.

Learned Spatial Index. The development of learned spatial indexes has led to faster query processing and more efficient index construction. Conducting more surveys in this emerging field would contribute to a better understanding of its progress.

References

[1] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, jun 1998.

[2] Hans-Peter Kriegel, Michael Schiwietz, Ralf Schneider, and Bernhard Seeger. Performance comparison of point and spatial access methods. In Alejandro P. Buchmann, Oliver Günther, Terence R. Smith, and Yuan-Fang Wang, editors, *Design and Implementation of Large Spatial Databases*, pages 89–114, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[3] K. Chakrabarti and S. Mehrotra. The hybrid tree: an index structure for high dimensional feature spaces. In *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)*, pages 440–447, 1999.

[4] Anderson Chaves Carniel and Cristina Dutra de Aguiar. Spatial index structures for modern storage devices: A survey. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–20, 2023.

[5] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. Lisa: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2119–2133, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Haowen Dong, Chengliang Chai, Yuyu Luo, Jiabin Liu, Jianhua Feng, and Chaoqun Zhan. Rw-tree: A learned workload-aware framework for r-tree construction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2073–2085, 2022.

[7] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. A reinforcement learning based r-tree for spatial data indexing in dynamic environments, 2021.

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

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

[10] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, jun 1984.

[11] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, jun 1984.

[12] J.L. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, SE-5(4):333–340, 1979.

[13] Norbert Beckmann and Bernhard Seeger. A revised r*-tree in comparison with related index structures. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 799–812, New York, NY, USA, 2009. Association for Computing Machinery.

[14] Max Egenhofer and Robert Franzosa. Point set topological spatial relations. *International journal of geographical information systems*, 5:161–174, 04 1991.

[15] Yannis Theodoridis and Dimitris Papadias. Range queries involving spatial relations: A performance analysis. In Andrew U. Frank and

Werner Kuhn, editors, *Spatial Information Theory A Theoretical Basis for GIS*, pages 537–551, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[16] Jun Zhang, Dimitris Papadias, Kyriakos Mouratidis, and Manli Zhu. Query processing in spatial databases containing obstacles. *International Journal of Geographical Information Science*, 19:1091–1111, 11 2005.

[17] Micha Sharir(and Amir Schorr. On shortest paths in polyhedral spaces. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, page 144–153, New York, NY, USA, 1984. Association for Computing Machinery.

[18] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial network databases. In Johann-Christoph Freytag, Peter Lockemann, Serge Abiteboul, Michael Carey, Patricia Selinger, and Andreas Heuer, editors, *Proceedings 2003 VLDB Conference*, pages 802–813. Morgan Kaufmann, San Francisco, 2003.

[19] E. W. Dijkstra. A note on two problems in connection with graphs. *Numer. Math.*, 1(1):269–271, dec 1959.