The "AI+R"-tree: An Instance-optimized R-tree

Abdullah-Al-Mamun
Purdue University
mamuna@purdue.edu

Ch. Md. Rakin Haider
Purdue University
chaider@purdue.edu

Jianguo Wang
Purdue University
csjgwang@purdue.edu

Walid G. Aref
Purdue University
aref@purdue.edu

2022 23rd IEEE International Conference on Mobile Data Management (MDM) | 978-1-6654-5176-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/MDM55031.2022.00023

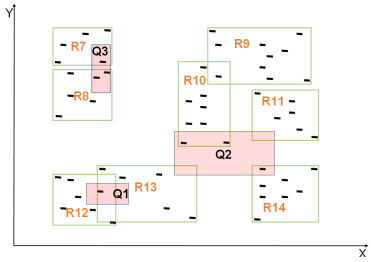
***Abstract*—The emerging class of instance-optimized systems has shown potential to achieve high performance by specializing to a specific data and query workloads. Particularly, Machine Learning (ML) techniques have been applied successfully to build various instance-optimized components (e.g., learned indexes). This paper investigates to leverage ML techniques to enhance the performance of spatial indexes, particularly the R-tree, for a given data and query workloads. As the areas covered by the R-tree index nodes overlap in space, upon searching for a specific point in space, multiple paths from root to leaf may potentially be explored. In the worst case, the entire R-tree could be searched. In this paper, we define and use the overlap ratio to quantify the degree of extraneous leaf node accesses required by a range query. The goal is to enhance the query performance of a traditional R-tree for high-overlap range queries as they tend to incur long running-times. We introduce a new AI-tree that transforms the search operation of an R-tree into a multi-label classification task to exclude the extraneous leaf node accesses. Then, we augment a traditional R-tree to the AI-tree to form a hybrid "AI+R"-tree. The "AI+R"-tree can automatically differentiate between the high- and low-overlap queries using a learned model. Thus, the "AI+R"-tree processes high-overlap queries using the AI-tree, and the low-overlap queries using the R-tree. Experiments on real datasets demonstrate that the "AI+R"-tree can enhance the query performance over a traditional R-tree by up to 500%.**

***Index Terms*—ML for Database Systems, Spatial Indexing, Instance-optimized components, Learned Indexes**

I. Introduction

Traditional spatial indexes have been used successfully over the years as an efficient access method for location data. In the area of spatial databases, the R-tree [1] is a widely used index structure. In the multi-dimensional space, the R-tree is analogous to the one-dimensional index structure B^+-tree [2]. These traditional index structures, e.g., the B^+-tree or the R-tree, do not make any assumptions about the underlying data distribution. They are designed to work on a variety of data and query workloads. As a result, an index is not necessarily optimized for a particular data and query workloads.

Recently, there is an emerging class of instance-optimized systems proposed to optimize system performance for a specific data and query workloads, e.g., [3], [4]. Following the same direction, we target to design an index for a particular data and query workloads, i.e., an instance-optimized index; a learned index that has better search and lower space requirements than their traditional counterparts [3], [5], [6]. Particularly, ML techniques have been successfully applied to build instance-optimized system components [4], [5]. Although ML models are normally trained to generalize over a variety of datasets, in the context of designing instance-optimized components, overfitting of ML models can be desired if the models learn only from a known dataset [3].



(a) An example of leaf nodes in an R-tree

Query	Visited leaf nodes	True leaf nodes
Q1	R12, R13	R12
Q2	R10, R11, R13, R14	R10, R14
Q3	R7, R8	R7, R8

(b) List of the accessed leaf nodes during query processing

Fig. 1: An example of R-tree range query processing

In this paper, we focus on answering range and point queries over an R-tree due to their wide applicability in spatial databases [7]. In the R-tree, objects are stored using Minimum Bounding Rectangles (MBRs). Notice that in the B^+-tree, nodes do not overlap in space. However, the MBRs of non-leaf and leaf nodes of an R-tree can overlap in space. Figure 1 illustrates the impact of node overlap in an R-tree to answer a range query. Only the MBRs of the leaf nodes are displayed in Figure 1. Notice that the number of accessed leaf nodes directly impacts the query response time of an R-tree [7]. For a disk-based R-tree, descending multiple paths in the R-tree incurs high I/O cost [8]. The leaf nodes of the R-tree are labelled R7-R14. Consider Range Queries Q1, Q2, and Q3 in Figure 1. To process Q1, the R-tree searches both R12 and R13, but the output data object is only present in R12. Hence, accessing R13 is wasted. Similarly, to process Q2, the R-tree searches R10, R11, R13 and R14, but the output data entries are only in R10 and R14. In both Q1 and Q2, the R-tree accesses 50% more leaf nodes than the true number of leaf nodes containing the data objects. In contrast, for Query Q3, the R-tree searches both R7 and R8, and data objects are exactly found in both nodes.

In this case, the number of visited leaf nodes by the R-tree matches the true number of leaf nodes required to answer Q3. Thus, in terms of the number of leaf node accesses, we can identify Q1 and Q2 as high-overlap queries and Q3 as a low-overlap query. Observe that the R-tree searches extraneous leaf nodes to answer Q1 or Q2 but performs optimally for Q3. We

2375-0324/22/$31.00 ©2022 IEEE
DOI 10.1109/MDM55031.2022.00023

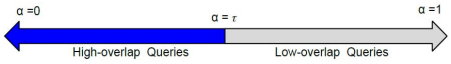


Fig. 2: Spectrum of the overlap ratio α with Threshold τ to identify high- and low-overlap queries.

define an overlap ratio α to quantify the degree of extraneous leaf node accesses required by a query. Specifically, for a range query, we divide the number of true leaf nodes by the total number of visited leaf nodes to estimate α, e.g., in Figure 1, to answer Q2, the number of visited leaf nodes is 4 while the number of true leaf nodes is 2 making $\alpha = 0.50$. Similarly, for Q1 and Q3, α is 0.50 and 1, respectively. Notice that the number of true leaf nodes cannot exceed the number of visited leaf nodes. Hence, α ranges from $[0 - 1]$.

For the purposes of this paper, the high- and low-overlap queries are determined as follows: Based on a pre-defined threshold τ, queries with overlap ratio $\alpha \leq \tau$ (i.e., closer to 0) are high-overlap while queries with $\alpha > \tau$ (i.e., closer to 1) are low-overlap. The spectrum of the of the overlap ratio α with Threshold τ is shown in Figure 2. ***To process high-overlap queries, we propose to find the true R-tree leaf nodes using Multi-label Classification***; a supervised ML task, where an input object can be classified into one or multiple categories at once [9]. For example, classifying a research paper into a Systems, Theory, or ML paper is a multi-label classification task as a paper can be both a Systems and ML paper. Analogously, we can cast answering a range query over the R-tree, as a multi-label classification task, where the classes are the R-tree leaf nodes, and we need to find these nodes that overlap the range query and that contain the output objects to the query.

Motivated by the benefits of instance-optimized components (e.g., learned indexes) and considering the issue of node overlap in the R-tree, the following important questions arise: *Which workloads degrade the performance of range query processing in a traditional R-tree? Can we leverage ML techniques to make R-tree range query processing faster?*

We propose to use the overlap ratio α to identify the high-overlap queries for which an R-tree accesses many extraneous leaf nodes. Moreover, we propose to build an ML-enhanced R-tree, termed the AI-tree, that leverages multi-label classification techniques [9]. Finally, we adopt a hybrid structure, termed the "AI+R"-tree, to avail the benefits of both the AI-tree and the traditional R-tree (refer to Figure 3). The ideas behind the AI-tree is as follows: First, we perform a preprocessing step to assign IDs to the leaf nodes of the R-tree. Then, we treat the queries as input and the corresponding true leaf node IDs as class labels. In the example R-tree in Figure 1, for $Q1, Q2$, and $Q3$, the corresponding class labels are the IDs $\{R12\}$, $\{R10, R14\}$, and $\{R7, R8\}$, respectively. Moreover, we prepare training data by processing each of the queries in a traditional R-tree and storing the corresponding true leaf node IDs as their class labels. Then, a multi-label classifier is constructed based on this training data. Motivated by the benefits of using multiple ML models (instead of a single model) [5], we adopt a multi-model approach that indexes the learned ML models in a grid-based structure.

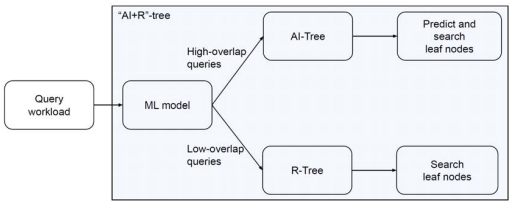


Fig. 3: Workflow of the "AI+R"-tree leveraging both the AI-tree and R-tree

To realize the "AI+R"-tree, we leverage a binary classification technique [10] to learn the value of the overlap ratio α given an input range query. This enables the "AI+R"-tree to differentiate between high- and low-overlap queries. Notice that the AI-tree is likely to perform better for high-overlap queries while a traditional R-tree is expected to perform better for low-overlap queries (due to the fact that there is limited scope for improvement). Notice further that the AI-tree performs exact (i.e., not approximate) range query processing. Thus, the "AI+R"-tree leverages the benefits of both the AI-tree and the R-tree.

The contributions of this paper are as follows:

1) We introduce an instance-optimized AI-tree that transforms the R-tree search operation into a multi-label classification task. While learned indexes are centered around the idea of *learning the index*, the AI-tree adds to that the idea of *indexing the learned models*.
2) We leverage ML to differentiate between high- and low-overlap range queries. This gives rise to the "AI+R"-tree that processes both query types efficiently.
3) For fixed query workloads, experiments on real spatial data demonstrate that the "AI+R"-tree enhances the performance of a traditional R-tree by up to 500%.

The remainder of this paper proceeds as follows: Section II presents the problem formulation. Section III introduces the AI-tree. Section IV introduces the hybrid "AI+R"-tree. Section V presents the experimental results. Section VI gives an overview of the related work. Finally, Section VII presents concluding remarks and suggestions for future research.

II. Background and Problem Formulation

A. The R-tree: An Overview

The R-tree [1] is a balanced hierarchical index for multi-dimensional objects. Each leaf or non-leaf node of the R-tree contains at least m and at most M entries. A rectangular range query is expressed as follows: Q (X_{min}, Y_{min}, X_{max}, Y_{max}), where (X_{min}, Y_{min}) and (X_{max}, Y_{max}) represent the bottom-left and top-right points of the query rectangle, respectively. To process a range query Q, we start from the root of the tree, and check the MBR for each child of the root against Q to test which child nodes overlap Q. In case of an overlap, we

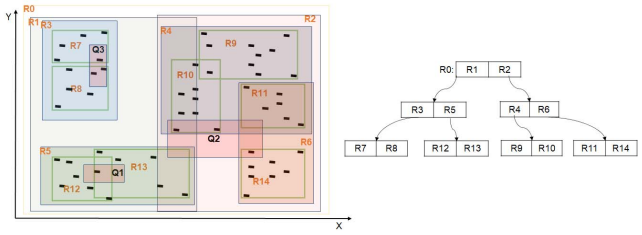


Fig. 4: An example of R-tree with overlapping nodes

search the sub-tree rooted at the corresponding child. When we reach a leaf node, we report the objects that overlap Q.

B. Classification: A Supervised Machine Learning Technique

Classification [10] is a commonly used ML technique. It can be divided into the following three categories: *(1) Binary Classification,* where the number of classes is restricted to two, *(2) Multi-class Classification,* where the number of classes n is more than 2, and the goal is to classify an object into exactly one of the n classes, and *(3) Multi-label Classification* [9], [11], where we also have $n(>2)$ classes, but the goal is to classify an object into c classes, where $0 \leq c \leq n$.

C. Problem Formulation

Refer to Figure 4 for illustration. Consider Queries Q1-Q3. For Q1, we search the R-tree down the 2 paths (from root to leaf): $R1 \rightarrow R5 \rightarrow R12$ and $R1 \rightarrow R5 \rightarrow R13$ while only the latter path contains actual query results. For Q2, we search the R-tree along 4 paths: $R1 \rightarrow R5 \rightarrow R13$, $R2 \rightarrow R4 \rightarrow R10$, $R2 \rightarrow R6 \rightarrow R11$, and $R2 \rightarrow R6 \rightarrow R14$ while only the 2nd and 4th paths contain output data objects. Notice that, for Query Q3, the R-tree searches two paths: $R1 \rightarrow R3 \rightarrow R7$ and $R1 \rightarrow R3 \rightarrow R8$, where both of them will contain output data objects. Thus, for processing Q1 and Q2, the R-tree searches extraneous leaf nodes. Hence, we formulate our problem as follows: **Given a range query $Q(X_{min}, Y_{min}, X_{max}, Y_{max})$, we need to predict the true leaf nodes of the R-tree that contain output data objects, and only access these nodes without accessing extraneous ones.**

We propose to formulate this problem as a multi-label classification task. For example, assume that an R-tree has four leaf nodes with unique IDs 1–4. For a range query, the R-tree may have to access any number of leaf nodes out of these four leaf nodes. We transform this problem into a multi-label classification task by treating the leaf node IDs as the class labels. At query time, the trained multi-label classifier predicts the true leaf node IDs that contain data entries that fall inside the query region. Hence, we only need to access the predicted leaf nodes to process the query.

III. The AI-tree

A. The Preprocessing Phase

1) Assigning Unique Identifiers to the R-tree Leaf Nodes: In the preprocessing step, each R-tree node is assigned a unique integer identifier (ID) based on a Depth First Search (DFS) order. Thus, all sibling leaf nodes of the R-tree will have consecutive integers as their IDs.

2) Definition of the Overlap Ratio α: We define an overlap ratio α to quantify the degree of extraneous leaf node accesses required by a range query. Given a range query Q, to calculate the value of α, we use two metrics: the true number of leaf-node accesses required to process Q (TN(Q), for short), and the number of leaf nodes visited by the R-tree index to answer Q (VN(Q), for short). For the range query Q, the definition of α is as follows (the value of α is in the interval $[0, 1]$):

$$\alpha = \frac{TN(Q)}{VN(Q)}$$

3) Query Workload Categorization: Given a query workload, we categorize each query based on its selectivity. After identifying the selectivity of a query, the overlap ratio α of the query is calculated to further categorize the queries based on their value of α. This is achieved by executing the query during the preprocessing phase, computing the query's selectivity, the leaf nodes being touched, and the true leaf nodes.

4) Preparing Training Data: This is a two-step process. In the first step, all queries in the query workload are executed one at a time on the constructed R-tree over the given dataset. For each executed query, we collect the following information: The IDs of the leaf nodes that the R-tree visits to answer the query, and the true leaf node IDs that contain the output data objects that are actually inside the query region.

TABLE I: Step-1 of Training Data Preparation

Query	Visited Nodes	True Nodes
$Q1$	R12,R13	R13
$Q2$	R13,R10,R11,R14	R10,R14
$Q3$	R7,R8	R7,R8

Assume that the ID assignment for the R-tree presented in Figure 4 is as follows: R7 and R8 have IDs 1 and 2, R12 and R13 have IDs 3 and 4, R9 and R10 have IDs 5 and 6, and R11 and R14 have IDs 7 and 8. For Query Q1, the visited leaf nodes are R12 and R13 but the true leaf node is R13. Thus, for training purposes, for Q1, we set the ID of R13, i.e., 3, as the output label for the multi-label classifier problem. Similarly, for Query Q2, we have the ID of R10 and R14 (6 and 8) as the labels for the multi-label classifier. Moreover, for Q3, we have the ID of R7 and R8 (1 and 2) as the labels. These steps are summarized in Tables I and II. In Table I, for each query, we list the visited and the true leaf nodes. In Table II, we list the leaf node IDs as the class labels for each of the queries.

TABLE II: Step-2 of Training Data Preparation

Query	Input Feature	Labels
$Q1$	$(X_{min}, Y_{min}, X_{max}, Y_{max})$	3
$Q2$	$(X_{min}, Y_{min}, X_{max}, Y_{max})$	6, 8
$Q3$	$(X_{min}, Y_{min}, X_{max}, Y_{max})$	1, 2

5) Feature Representation: For an input range query Q, we use the values $(X_{min}, Y_{min}, X_{max}, Y_{max})$ of the query rectangle as input features to the ML model without any additional transformation. Thus, the same input can be processed seamlessly by both the AI-tree and the R-tree. Moreover, for multi-label classification, the output labels are encoded using one-hot encoding, where we represent the class labels using binary values, which is suitable for training the multi-label classifier, e.g., in Table II, for query $Q1, Q2$ and $Q3$, the labels will be encoded as $00100000, 00000101$ and 11000000.

B. Learning the R-tree Index: ML Model Training and Testing

Refer to Figure 5. The workflow for training and testing the multi-label classifier is as follows. (1) While the given

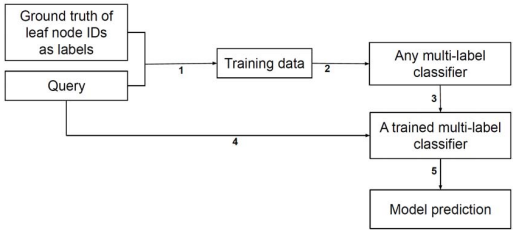


Fig. 5: Workflow of model training and testing

queries are executed by the R-tree, for each query, the IDs of the visited and true leaf nodes are captured. Thus, following the approach in Section III-A4 (i.e., using the feature representation of the queries and the true leaf node IDs as labels), the training data is prepared for a particular data and query workloads. (2) Then, the training data is used to train the multi-label classifier. Because the goal is to enhance the performance of the R-tree search operation for a fixed data and query workloads, the ML models are intentionally overfitted on the training data. In this paper, we use a multi-label decision tree classifier [11], [12] due to its ability to overfit the training data. Notice that the choice of the multi-label classifier is not limited to the family of decision tree classifiers. Because R-tree search has been cast as a multi-label classification problem, we have the opportunity to use any suitable multi-label classifier [13]. (3) A trained multi-label classifier will be created after the training phase. (4) As the AI-tree is optimized for a fixed query workload, the queries will be re-used as input for both the training phase of the multi-label classifier and the testing phase. This approach is similar to the previous works that leverage overfitting to build instance-optimized systems components [3], [5]. (5) At query time, the pre-trained multi-label classifier is invoked to directly predict the true leaf node IDs that contain the query result.

Indexing the Learned Models: A Multi-model Approach

The idea of indexing multiple learned models using a traditional index structure has been used in the context of music retrieval [14] and in handwritten and time series data [15]. Moreover, the benefit of indexing the learned models using a recursive model index is also shown in [5]. Notice that for exact range query processing, our goal is to perfectly (i.e., 100% prediction accuracy) fit the ML models to a particular data and query workloads. However, even with overfitting, it might not be possible to train a single ML model to capture the entire underlying distribution of the training data [5]. As a result, to achieve high prediction accuracy on the training dataset, multiple ML models are trained, e.g., several multi-label decision tree classifiers instead of a single ML model.

In the AI-tree, we use a simple index structure, e.g., a coarse grid to partition the training queries. Then, we train a separate ML model over queries inside each grid partition. The grid serves as an index to the localized learned ML models. As a result, at query time, we only invoke the ML models whose grid cell overlap the query rectangle. This concept is illustrated in Figure 6. The steps are as follows: Initially, the underlying

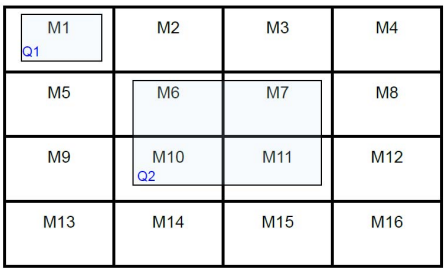


Fig. 6: Indexing the learned models

space is divided into equally sized grid cells. Then, during the training phase, if a query overlaps a single grid cell, the ML model corresponding to that cell is trained for that query. Similarly, if a query overlaps multiple grid cells, all the cells' corresponding ML models are trained for that query. Notice that if no query overlaps a particular grid cell, we do not train any ML model for that grid cell. For example, for a $10X10$ grid, it is not always the case that 100 ML models need to be trained. Finally, during query processing, only the ML models whose grid cells overlap the range query are executed.

For example, in Figure 6, the space is partitioned using a $4X4$ grid. Thus, at most 16 ML models (M1 to M16) can be trained using the grid. In the training phase, we incrementally search for the grid size that produces the best fit over the training data [16]. In Figure 6, as Query Q1 falls completely inside the top left grid cell, Model M1 is trained for Q1. On the other hand, as Query Q2 overlaps four grid cells, the four models M6, M7, M10, and M11 are trained for Q2. If multiple models are trained for a particular query, during query processing, we aggregate their prediction results from all the overlapping cells. This aggregated result is produced by performing a union of the predictions of the individual ML models.

C. Query Processing

Given a range query, first, the ML models are identified whose grid cells overlap the range query. Then, only the designated ML models are executed to process the query. After the prediction of the ML models, the results are aggregated in

terms of leaf node IDs. Then, only the leaf nodes whose IDs have been predicted by the ML models are accessed. Finally, all the data entries inside these leaf nodes are scanned to check which entries are actually contained within the input query rectangle. This ensures that the AI-tree never produces a false-positive result. **Notice that we only access the predicted leaf nodes without traversing the R-tree or accessing the non-leaf nodes or the extraneous leaf nodes. Thus, if we predict the leaf nodes accurately, we access the minimum number of leaf nodes needed to answer a query. This reduces the number of disk I/Os for processing a range query**.

Notice that in rare cases, the multi-label classifier (Section II-B) might not predict any label for a particular query. In other words, the classifier might predict no leaf node ID for a particular query. For the AI-tree, if the set of predicted leaf node IDs is empty for a particular query, we invoke a regular R-tree search operation. Moreover, if an ML model predicts a leaf node that does not contain any data object that is qualified in the result (i.e., mispredicts) of the given range query, we may resolve to search the regular R-tree. Thus, the AI-tree performs exact query processing by combining both the multi-model approach and the regular R-tree.

IV. The "AI+R"-tree

To achieve the best of both the AI-tree and the R-tree, we adopt a hybrid approach that we term the "AI+R"-tree (see Figure 3). We process the high-overlap queries using the AI-tree and the low-overlap queries using the traditional R-tree. However, this is non-trivial because the overlap ratio α of a query is unknown until we process the query. Hence, we leverage ML techniques to learn how to distinguish between high- and low-overlap queries. Specifically, the problem of classifying the range queries based on the value of α and the threshold τ can be formulated as a binary classification task II-B. In order to prepare the training data for a particular dataset, we combine the queries for each of the α values. Then, we assign Label 0 for the queries whose α value is less than or equal to the threshold τ, and assign Label 1 for the queries whose α value is greater than the threshold τ. Next, a binary classifier is trained on the training data. Finally, we can use the trained binary classifier to classify an incoming range query into either a high- or a low-overlap query.

A. Range Query Processing in the "AI+R"-tree

Given a range query Q, the binary classifier is invoked first (see Figure 3) to predict whether the incoming query Q is high- or low-overlap. If Q is classified as a high-overlap query, the AI-tree processes the query. Otherwise, the R-tree processes the query. Notice that query processing using the "'AI+R"-tree incurs a prediction cost before accessing the leaf nodes. Hence, the cost of query processing of the "AI+R"-tree is: ML model prediction cost + I/O cost. Thus, we expect to get the benefit of the AI-tree for processing the high-overlap queries whose α value is closer to zero. On the other side of the spectrum of α (Figure 2), for the queries with α closer to one, the R-tree is expected to perform better than the AI-tree.

To demonstrate query processing in the "AI+R"-tree, consider the three queries in Figure 1. For Queries Q1 and Q2, the overlap ratio $\alpha = 0.50$. If the "AI+R"-tree can accurately predict the leaf nodes, 50% less number of leaf nodes will be accessed to answer the query. Notice that we have room for improvement to process Q1 and Q2 using the AI-tree component of the "AI+R"-tree. In contrast, for Q3, $\alpha = 1$. Thus, both the visited leaf nodes contain data entries that fall inside the query rectangle. Thus, it is not possible for the AI-tree to process the query using less leaf node accesses than the R-tree. Thus, we use the R-tree in this case.

V. Evaluation

We run all experiments on an Ubuntu 18.04 with Intel Xeon Platinum 8168 (2.70GHz) and 3TB of total available memory.

A. Datasets

We use two datasets from the UCR Spatio-Temporal Active Repository, namely UCR-STAR [17]. Specifically, we use two real-world datasets with two-dimensional location data (in the form of longitude and latitude). The Tweets location dataset contains the locations of real tweets, and the other dataset contains the locations of Chicago crimes. Moreover, we have preprocessed the datasets before using them for the experiments. First, we eliminate the duplicate and missing values from both datasets. For the Tweets location dataset, we create a processed dataset containing the first 2 million tweet locations. On the other hand, after removing the duplicate values from the dataset of Chicago crimes locations, we get a processed dataset containing $872,127$ records.

B. Parameter Settings

1) R-tree Parameters: All R-tree variants attempt to reduce the amount of node overlap. However, with dynamic updates, the shape of the R-tree deteriorates over time. Thus, we construct the R-tree using a one-at-a-time tuple insertion method to replicate the scenario of a dynamic environment. When constructing the R-tree, we set the minimum leaf node size m to 50% of the maximum leaf node size M. Another parameter of the R-tree is the node-splitting algorithm. In the experiments, we use the linear node-splitting algorithm.

2) Query Selectivity and Values of α: For a particular dataset, to demonstrate the query performance for a particular value of α, (at most) 1000 synthetic range queries are used in the experiments with a fixed selectivity. For example, in the case of the Tweets location dataset, a range query with Selectivity 0.00001 returns approximately 20 objects, and a query with Selectivity 0.00005 returns approximately 100 objects. In the experiments, the selectivity varies between 0.00001 and 0.00005. We categorize the queries into five different values of $\alpha[0.1, 0.25, 0.5, 0.75, 1.0]$. Thus, for each dataset, we use up to 5000 queries with various values of α.

3) The "AI+R"-tree Parameters: The "AI+R"-tree has two parameters: The size of the grid (see Section III-B) and the choice of the threshold τ (see Figure 2). Similar to the idea of hyperparameter tuning [16] for ML models, we start

from a grid size $2X2$ and increase the size (e.g., $4X4$) to get the best fit for the training data. In all the experiments, we have achieved the best fit over the training data with a maximum grid size of 20X20. Notice that using the multi-model approach and invoking the regular R-tree in case of a misprediction, the AI-tree can achieve 100% prediction accuracy over the training data. As a result, both the AI-tree and the "AI+R"-tree can perform exact (i.e., not approximate) range query processing.

On the other hand, for a query Q with $\alpha = 0.75$, the $\frac{TN(Q)}{VN(Q)}$ can be e.g., $\frac{15}{20}$. Thus, there is room for improvement unless $\alpha = 1$. Thus, we set Threshold $\tau = 0.75$. In other words, for an incoming range query, if $\alpha \leq 0.75$, it is identified as a high-overlap query. If $\alpha > 0.75$, it is considered low-overlap.

C. Choosing the ML Models

1) The Multi-label Classifier: A decision-tree classifier [12] has the ability to overfit the training data, and hence can achieve high prediction accuracy for the training dataset. Moreover, a decision-tree classifier is simple and explainable. As a result, we use a multi-label decision tree model as the multi-label classifier [11]. However, with proper training, any multi-label classifier [13] can be used in the "AI+R"-tree. For the ML models, we use the standard scikit-learn python library [18]. We use the default parameters for the decision tree classifier except the maximum-depth that is set to 30. This maximum-depth is set to a high value to allow the decision tree classifier to overfit the training data.

2) The Binary Classifier: For binary classification, the goal is to train an ML model to classify an incoming query as high- or low-overlap. Notice that the goal is not to overfit but rather to generalize, and the same learned model will be able to classify high- vs. low-overlap queries across different query workloads. We use a random forest classifier [19] as the binary classifier. However, with proper training, any binary classifier can be used in the "AI+R"-tree. The training process of the binary random forest classifier is as follows: For a particular data and query workloads, we combine the queries for each $\alpha[0.1, 0.25, 0.5, 0.75, 1.0]$. For a particular dataset with fixed selectivity queries, we will have up to 5000 queries in total. For the binary classifier, we create the training data as follows: We assign Label 0 for queries where $\alpha \leq \tau$ (e.g., $\alpha \leq 0.75$), and Label 1 for queries where $\alpha > \tau$ (e.g., $\alpha > 0.75$). Moreover, we split the training data where we use 80% for training and 20% for testing. We use the scikit-learn python library [18], and use the default scikit-learn settings for the random forest binary classifier. The prediction accuracy of the binary random forest classifier is around 80% over all values of α.

D. Implementation and Measurements

We realize the "AI+R"-tree using an open-source python library for the R-tree available on Github [1]. We integrate the "AI+R"-tree inside the library and run the experiments using Python Version 3.6.9. On the other hand, for a disk-based R-tree index realized inside a practical system, in most of the cases, only the leaf nodes are stored in disk pages, and the internal nodes are kept in-memory. As a result, the performance of a query depends on both the CPU cost and the number of leaf node accesses. In the experiments, we assume that the required number of disk I/Os is equivalent to the number of leaf node accesses [20]. For a query, we measure the CPU time, and count the number of leaf node accesses. Then, we multiply the number of leaf node accesses by a standard disk I/O access time. Finally, we sum the CPU and disk I/O times to report the average query processing time (in milliseconds). In the experiments, we use a disk I/O access time of thirteen milliseconds [21]. This approach is similar to the experimental setup of a previous work [20].

Also, we report the size of the R-tree and the size of ML models [2]. The size of the ML models contains the summation of the sizes of both the multi-label and the binary classifiers.

Notice that for a particular query workload with fixed selectivity, to demonstrate the performance for each value of α, we run each experiment individually for each value of α. This enables us to report the average query processing time and the size of the ML models for each value of α.

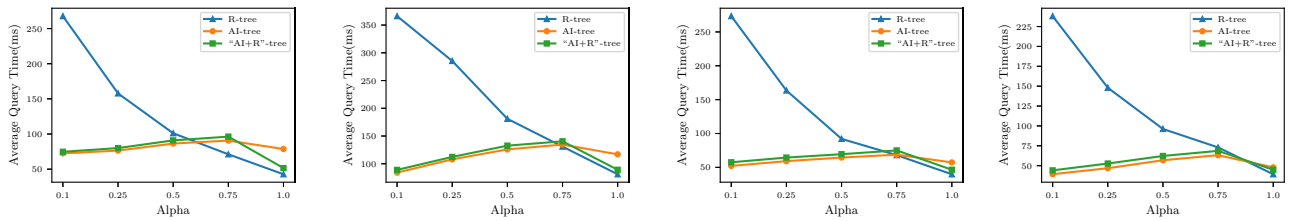
E. Experimental Results

1) Tweet Locations Dataset: We construct an R-tree using the minimum leaf capacity $m = 100$ and maximum leaf capacity $M = 200$. The selectivities of the synthesized queries iare: 0.00001, and 0.00005. As a result, for each range query, the result contains approximately 20 and 100 data points, respectively. Moreover, the value of the threshold τ is set to 0.75. In each of the figures for this dataset, we show the value of overlap ratio α in the X-axis and the average query processing time (in milliseconds) in the Y-axis. Also, we report the average query processing time taken by the standard R-tree, the AI-tree, and the "AI+R"-tree.

2) Effect of Selectivity for the Tweets Location Dataset: Figure 7a gives the results for Selectivity 0.00001. From the figure, both the AI-tree and the "AI+R"-tree enhance the performance of the R-tree by up to 3.69X and 3.58X, respectively. Notice that the performance loss is minimal between the AI-tree and the "AI+R"-tree, where the latter exhibits a hybrid approach to indexing. This same pattern of performance gains for both trees applies for the cases of $\alpha = 0.25$. For $\alpha = 0.25$, the AI-tree and the "AI+R"-tree enhance the performance of the R-tree up to 2.06X and 1.97X, respectively. Moreover, the "AI+R"-tree performs better than the R-tree up to $\alpha = 0.50$. After that the R-tree starts to perform better. Notice that the hybrid approach reduces the query processing time of the AI-tree when the $\alpha = 1$. **In summary, the "AI+R"-tree gets the best of both worlds. In the case of high-overlap (low α value), the "AI+R"-tree performs similar to the AI-tree, while in the case of low-overlap (high α value), the "AI+R"-tree performs similar to the standard R-tree.**

[1] https://github.com/sergkr/rtreelib

[2] https://docs.python.org/3.6/library/sys.html



(a) R-tree (M=200, m=100), and query selectivity = 0.00001

(b) R-tree (M=200, m=100), and query selectivity = 0.00005

(c) R-tree (M=400, m=200), and query selectivity = 0.00001

(d) R-tree (M=800, m=400), and query selectivity = 0.00001

Fig. 7: Results on Tweet locations dataset

Figure 7b gives the results for the same setup but for a selectivity of 0.00005. As a result, each of the queries returns approximately 100 points. From the figure, the AI-tree and the "AI+R"-tree exhibit the similar trend in performance gains.

3) The Effect of Node Capacity for the Tweets Location Dataset: In the next experiment, we vary the leaf node capacity. We cover the cases for $M = 200, 400$, and 800. We fix the selectivity of the synthesized queries to: 0.00001 (The query result will contain approximately 20 data points). The AI-tree can perfectly fit the training data with a $10X10$ grid size for R-tree with node capacity $M = 400$, and 800.

Figures 7a, 7c, and 7d give the performance results of the AI-tree and the "AI+R"-tree for maximum leaf node capacities of 200, 400, and 800, respectively. The performance trends are the same. Overall, the performance gains of the AI-tree and the "AI+R"-tree over the R-tree increase as the node capacities increase. The reason is that as the node capacity increases, any additional extraneous leaf nodes retrieved by the traditional R-tree will be very expensive due to the refinement step. Basically, as the node capacity increases, more leaf data objects will need to be checked against the query range to refine the results and form the actual output data objects from among the ones in the leaf node. In other words, due to the higher leaf node capacities, the penalty of an unnecessary scan inside an extraneous leaf node reduces the R-tree performance in contrast to the AI-tree and the "AI+R"-tree. In Figure 7d, for node capacity 800, the AI-tree enhances the performance of the R-Tree up to 6.06X for $\alpha = 0.10$. Also, the "AI+R"-tree does not decrease the performance of the AI-tree by a large margin. To be precise, the "AI+R"-tree enhances the performance of the R-tree up to 5.39X for $\alpha = 0.10$.

TABLE III: The R-tree and ML model sizes for the "AI+R"-tree for each α (in MBs) for the Tweets Location dataset

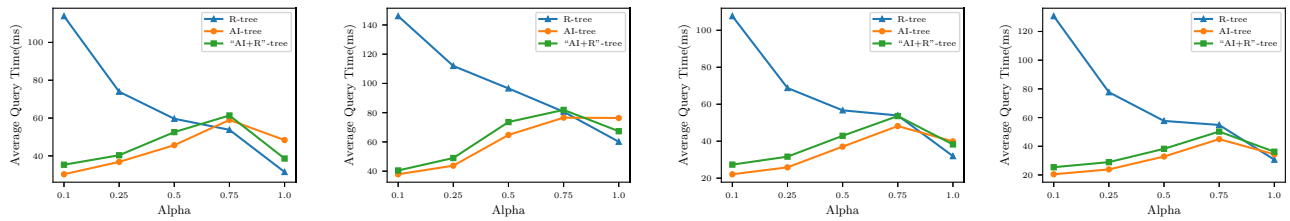
			"AI+R"-tree with various values of α				
Selectivity	Max Entries	R-tree	0.10	0.25	0.50	0.75	1.0
0.00001	200	978.05	9.50	9.50	9.51	11.38	9.50
0.00005	200	978.05	9.44	9.50	9.52	9.52	9.51
0.00001	400	972.05	1.97	2.02	2.97	2.96	2.01
0.00001	800	969.52	1.02	2.07	1.55	1.07	1.06

4) Space Consumption of the ML Models for Tweets Location Dataset: Table III lists the sizes of the R-tree and the ML models of the "AI+R"-Tree in Megabytes (MB, for short). Notice that the reported ML model size includes the sizes of both the multi-label and the binary classifiers. The space requirements of the ML models for the "AI+R"-tree with larger leaf capacity (e.g., M=400, and M=800) are less than those for the R-tree with leaf capacity 200. The reason is that the number of leaf nodes is less for the larger node capacities. Also, notice that a grid of size $10X10$ is sufficient for "AI+R"-tree with larger node capacity. Hence, less models are likely to be trained to fit the data. Thus, the size of the ML models is even less than the cases of using a larger grid of size $20X20$.

5) The Chicago Crimes Dataset: Overall, the Chicago Crimes dataset reflects the same performance trends as those for the Tweets Location dataset in favor of the "AI+R"-tree over the R-tree. We give the performance results for the the Chicago Crimes dataset below. For the Chicago crimes dataset, initially, we construct an R-tree using the maximum leaf capacity $M = 200$, and minimum leaf capacity $m = 100$. Moreover, the selectivity of the synthesized queries is: 0.00001 and 0.00005. For each range query, the result contains (approximately) 9 and 44 data points, respectively. Moreover, the AI-tree can perfectly fit the training data for this query workload with a grid size $20X20$. τ is set to 0.75.

6) Effect of Selectivity for the Chicago Crimes Dataset: Figures 8a and 8b give the performance results for the AI-tree, the "AI+R"-tree, and the R-tree for Selectivities 0.00001 and 0.00005. The performance gains of the "AI+R" over the R-tree is up to 3.6X for high overlap queries ($\alpha = 0.10$) while is very close to the R-tree for $\alpha = 1.0$. This is consistent for both selectivity values.

7) Effect of Node Capacity for Chicago Crimes Dataset: We vary leaf node capacity to cover for $M = 200, 400$, and 800. We fix query selectivity to: 0.00001. Moreover, the AI-tree can perfectly fit the training data for a $10X10$ grid size and R-tree with node capacity $M = 400$ and 800. Figures 8a, 8c, and 8d give the performance results of the AI-tree and the "AI+R"-tree for the Chicago Crimes dataset for maximum leaf node capacities $M = 200$, 400, and 800. The performance trends are consistent with those of the Tweets Location dataset. Overall, the performance gains of the AI-tree and the "AI+R"-



(a) R-tree (M=200, m=100), and query selectivity = 0.00001

(b) R-tree (M=200, m=100), and query selectivity = 0.00005

(c) R-tree (M=400, m=200), and query selectivity = 0.00001

(d) R-tree (M=800, m=400), and query selectivity = 0.00001

Fig. 8: Results on Chicago crimes dataset

tree over the R-tree increase as the node capacities increase, e.g., for $M = 800$, the "AI+R"-tree enhances the performance over the R-tree by 5.14X for $\alpha = 0.10$. Also, the "AI+R"-tree reduces the query performance of the AI-tree for $\alpha = 1$ to be close to that of the R-tree.

8) Space Consumption of the ML Models for the Chicago Crimes Dataset: Table IV lists the sizes of the R-tree and the ML models of the "AI+R"-Tree (in MBs). The reported ML model size includes the sizes of both the multi-label and the binary classifiers. The space requirements of the ML models for the "AI+R"-tree with larger leaf capacity (e.g., M=400, and M=800) are less than for the R-tree with $M = 200$. A grid of size $10X10$ is found to be sufficient for the "AI+R"-tree in these cases. Hence, less models are likely to be trained to fit the data. Also, the size of the ML models are even less than the cases for using a larger grid of size $20X20$.

TABLE IV: The R-tree and ML model sizes for the "AI+R"-tree for each α (in MBs) for the Chicago Crimes dataset.

			"AI+R"-tree with various values of α				
Selectivity	Max Entries	R-tree	0.10	0.25	0.50	0.75	1.0
0.00001	200	426.46	3.39	2.57	3.40	3.40	3.40
0.00005	200	426.46	2.52	2.57	2.58	3.41	3.41
0.00001	400	423.86	0.48	0.53	0.53	0.53	0.53
0.00001	800	422.84	0.27	0.32	0.33	0.33	0.32

*9) **Discussion**:* From Figures 7a and 8a, the R-tree performs better than the AI-tree for $\alpha = 0.75$. As we set the threshold $\tau = 0.75$, the "AI+R"-tree also degrades in performance because it uses the AI-tree to process these queries with $\alpha = 0.75$. In both cases, the R-tree has relatively small leaf capacity (i.e., $M = 200$). As the leaf capacity increases, for the same value of threshold $\tau = 0.75$, the "AI+R"-tree performance enhances (see Figure 7d for the Tweets Location dataset and Figure 8d for Chicago Crimes dataset).

For each α, the ML models increase the space requirement of the R-tree by no more than 1.1% (see Table III). Also, **the space overhead of the ML models for all values of α does not increase the size of the R-tree by more than** 5.04% (Table III). Thus, the space requirement of the ML models can be as low as 0.37% of the R-tree size (Table IV).

VI. Related Work

Many variants of the R-tree have been introduced, e.g., see [1], [7], [22]–[25]. The R$^+$-tree [22] recognizes the problem of node overlap in the R-tree, and creates an R-tree so that no two nodes overlap in space. The R*-tree [23] reduces node overlap by introducing the forced re-insertion of entries. The RR*-tree [24] is a further improvement over the R*-tree for dynamic data. The RR*-tree improves over the R*-tree by restricting the insertion to a single path and dropping the idea of re-insertion. In [26], the Clipped Bounding Box (CBB) based R-tree further improves the I/O performance of the R*-tree. The priority R-tree [27] can answer a query with an asymptotically optimal number of I/Os. The Hilbert R-tree [28] leverages the Hilbert space-filling curve to impose an ordering on the R-tree nodes to achieve good space utilization. A worst-case optimal R-tree packing strategy that uses space-filling curves can be found in [29]. Notice that regardless of the type of the R-tree, all R-tree variants attempt to reduce the amount of node overlap. However, with dynamic updates, the shape of an constructed R-tree deteriorates. As a result, our design principles for "AI+R"-tree can be applied to other R-tree variants.

The concept of separating objects into partitions based on their size and indexing each partition with a space filling curve can be found in [30], [31]. However, in the case of "AI+R"-tree, we do not partition the objects based on their size, but rather we group the queries using the grid to train multiple ML models. Moreover, we do not use any space filling curve (i.e., as a projection function).

The initial research on learned indexes [5], [32] has introduced the idea that "Indexes are models" by proposing a Recursive Model Index (RMI, for short) for read-only workloads. Many followup research has been conducted that is inspired by RMI both in the single and Multi-dimensional space [6], [33]–[35].

In the case of multi-dimensional indexes, some initial effort to extend the idea of RMI into the multi-dimensional space can be found in [36]. In [37], Z/Morton order is used to project the data into the one dimensional space. Then, an RMI-like structure can be used to build the learned index. However, learning the projection function, e.g., Z-order [38], [39], from the multi-dimensional space to the one-dimensional space is

hard. Thus, it has been proposed to choose a layout that is easy to learn by an ML model. An efficient scaling method has been proposed in [40]. In our proposed "AI+R"-tree, we avoid using a projection function, and operate directly on the original multi-dimensional representation of the spatial data objects. In [41], an in-memory learned multi-dimensional index, termed Flood, is introduced to efficiently support queries for a particular dataset and (read-only) query workloads. An extension to Flood has been proposed that can adapt to changes in the query workload [42]. Reinforcement Learning has been used to build an efficient data layout [4], [43] for a particular dataset and query workload. A learned spatial index for disk-based systems can be found in [8]. In [44], another disk-based spatial index, termed RSMI, leverages a rank-space-based transformation. The transformation has been used to get an easily learnable CDF. Notice that the goal of the above mentioned learned multi-dimensional indexes is to replace a traditional index. However, in the case of the "AI+R"-tree, our target is not to replace the existing index structure rather to enhance its performance using ML models.

The idea of using helper ML models inside traditional indexes to enhance their performance have been presented in the multi-dimensional space, e.g., see [45]–[47]. In [45], interpolation-based learned spatial indexes are proposed. In [46], techniques from [41] have been applied to five traditional multi-dimensional indexes. Recently, a disk-based ML-enhanced index to process k-nearest-neighbor queries over high-dimensional time-series data has been proposed [47]. The goal of the proposed method [47] is to re-organize the access order of the leaf nodes. In the context of ML-enhanced multi-dimensional indexes, the focus of the above mentioned techniques is not on analyzing (i.e., high- vs. low-overlap queries) and optimizing the index for a given query workload. Notice that, in the case of the "AI+R"-tree, the focus is on analyzing the query workload to identify the queries for which a traditional disk-based spatial index (in this case, the R-tree) does not perform well. Moreover, we propose to adopt a hybrid approach to leverage the benefit of both the proposed AI-tree, and the traditional R-tree.

Surveys on the topic of learned data structures can be found in [33], [48]. Recently, several tutorials related to learned indexes have been presented in different venues [6], [34], [35], [49], [50].

VII. Conclusion

In this paper, we leverage machine learning techniques to build an instance-optimized R-tree for a given data and query workloads. Although the paper focuses on the R-tree, the proposed design principles in the paper apply to other spatial indexes as long as node overlaps exist, and hence multiple tree paths are explored during search. Notice that we avoid using a projection function, and operate directly on the original representations of the spatial data objects. Also, because the "AI+R"-tree operates at the leaf node level of an R-tree, the proposed method can support different types of objects (e.g., objects with extension). Additionally, we adopt a multi-model approach and index the learned ML models using a grid-based structure. We further leverage ML techniques to train a binary classifier to differentiate between high- and low-overlap queries. Finally, we advocate for a hybrid approach, namely the "AI+R"-tree by combining both the traditional R-tree structure and the learned R-tree (i.e., the AI-tree) structure to maximize query processing performance. In the future, we plan to investigate alternative choices for the ML models, and how to support k-NN query and spatial join using the proposed "AI+R"-tree. As we maintain a hybrid structure inside the "AI+R"-tree, we will be able to sustain updates using its R-tree component. However, propagating the updates to the AI-tree component is an interesting future research direction. Finally, we plan to investigate challenges related to the integration of the proposed "AI+R"-tree into practical database systems.

Acknowledgments

Walid Aref acknowledges the support of the U.S. National Science Foundation under Grant Numbers: III-1815796 and IIS-1910216.

References

[1] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the ACM SIGMOD international conference on Management of data*, 1984, pp. 47–57.
[2] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.
[3] T. Kraska, "Towards instance-optimized data systems," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, p. 3222–3232, 2021.
[4] J. Ding, U. F. Minhas, B. Chandramouli, C. Wang, Y. Li, Y. Li, D. Kossmann, J. Gehrke, and T. Kraska, "Instance-optimized data layouts for cloud analytics workloads," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2021, pp. 418–431.
[5] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2018, pp. 489–504.
[6] A. Al-Mamun, H. Wu, and W. G. Aref, "A tutorial on learned multi-dimensional indexes," in *Proceedings of the 28th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2020, pp. 1–4.
[7] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, *R-trees: Theory and Applications*. Springer Science & Business Media, 2010.
[8] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "Lisa: A learned index structure for spatial data," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2020.
[9] F. Herrera, F. Charte, A. J. Rivera, and M. J. Del Jesus, "Multilabel classification," in *Multilabel Classification*. Springer, 2016, pp. 17–31.
[10] C. C. Aggarwal, "Data classification," in *Data Mining*. Springer, 2015, pp. 285–344.
[11] E. Gibaja and S. Ventura, "A tutorial on multilabel learning," *ACM Computing Surveys (CSUR)*, vol. 47, no. 3, pp. 1–38, 2015.
[12] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.
[13] P. Szymanski and T. Kajdanowicz, "Scikit-multilearn: a scikit-based python environment for performing multi-label classification," *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 209–230, 2019.
[14] H. Jin and H. Jagadish, "Indexing hidden markov models for music retrieval." in *ISMIR*, 2002.
[15] W. Aref, D. Barbará, and P. Vallabhaneni, "The handwritten trie: Indexing electronic ink," in *SIGMOD Rec.*, 1995, p. 151–162.
[16] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," *Advances in neural information processing systems*, vol. 24, 2011.
[17] S. Ghosh, T. Vu, M. A. Eskandari, and A. Eldawy, "Ucr-star: The ucr spatio-temporal active repository," *SIGSPATIAL Special*, vol. 11, no. 2, pp. 34–40, 2019.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[20] A. R. Mahmood, W. G. Aref, A. M. Aly, and S. Basalamah, "Indexing recent trajectories of moving objects," in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2014, pp. 393–396.

[21] Y. Deng, "What is the future of disk drives, death or rebirth?" *ACM Computing Surveys (CSUR)*, vol. 43, no. 3, pp. 1–27, 2011.

[22] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects." Tech. Rep., 1987.

[23] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: an efficient and robust access method for points and rectangles," in *Proceedings of the ACM SIGMOD international conference on Management of data*, 1990, pp. 322–331.

[24] N. Beckmann and B. Seeger, "A revised r*-tree in comparison with related index structures," in *Proceedings of the ACM SIGMOD International Conference on Management of data*, 2009, pp. 799–812.

[25] H. Samet, *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[26] D. Sidlauskas, S. Chester, E. T. Zacharatou, and A. Ailamaki, "Improving spatial data processing by clipping minimum bounding boxes," in *34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 425–436.

[27] L. Arge, M. D. Berg, H. Haverkort, and K. Yi, "The priority r-tree: A practically efficient and worst-case optimal r-tree," *ACM Transactions on Algorithms (TALG)*, vol. 4, no. 1, pp. 1–30, 2008.

[28] I. Kamel and C. Faloutsos, "Hilbert r-tree: An improved r-tree using fractals," in *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994, p. 500–509.

[29] J. Qi, Y. Tao, Y. Chang, and R. Zhang, "Packing r-trees with space-filling curves: Theoretical optimality, empirical efficiency, and bulk-loading parallelizability," *ACM Transactions on Database Systems (TODS)*, vol. 45, no. 3, pp. 1–47, 2020.

[30] R. Zhang, J. Qi, M. Stradling, and J. Huang, "Towards a painless index for spatial objects," *ACM Transactions on Database Systems (TODS)*, vol. 39, no. 3, pp. 1–42, 2014.

[31] N. Koudas and K. C. Sevcik, "Size separation spatial join," in *Proceedings of the ACM SIGMOD international conference on Management of data*, 1997, pp. 324–335.

[32] R. Marcus, E. Zhang, and T. Kraska, "Cdfshop: Exploring and optimizing learned index structures," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2020.

[33] P. Ferragina and G. Vinciguerra, "Learned data structures," *Recent Trends in Learning From Data*, pp. 5–41, 2020.

[34] S. Idreos and T. Kraska, "From auto-tuning one size fits all to self-designed and learned data-intensive systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2019, pp. 2054–2059.

[35] I. Sabek and M. F. Mokbel, "Machine learning meets big spatial data," in *36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1782–1785.

[36] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, J. Ding, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan, "Sagedb: A learned database system," in *9th Biennial Conference on Innovative Data Systems Research*, 2019.

[37] H. Wang, X. Fu, J. Xu, and H. Lu, "Learned index for spatial queries," in *20th International Conference on Mobile Data Management (MDM)*. IEEE, 2019, pp. 569–574.

[38] H. Sagan, *Space-filling Curves*. Springer Science & Business Media, 2012.

[39] M. F. Mokbel, W. G. Aref, and I. Kamel, "Analysis of multi-dimensional space-filling curves," *GeoInformatica*, vol. 7, no. 3, pp. 179–209, 2003.

[40] A. Davitkova, E. Milchevski, and S. Michel, "The ml-index: A multidimensional, learned index for point, range, and nearest-neighbor queries." in *International Conference on Extending Database Technology (EDBT)*, 2020, pp. 407–410.

[41] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2020, pp. 985–1000.

[42] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, "Tsunami: a learned multi-dimensional index for correlated data and skewed workloads," *Proceedings of the VLDB Endowment*, vol. 14, no. 2, pp. 74–86, 2020.

[43] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P.-Å. Larson, D. Kossmann, and R. Acharya, "Qd-tree: Learning data layouts for big data analytics," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2020, pp. 193–208.

[44] J. Qi, G. Liu, C. S. Jensen, and L. Kulik, "Effectively learning spatial indices," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2341–2354, 2020.

[45] A. Hadian, A. Kumar, and T. Heinis, "Hands-off model integration in spatial index structures," in *Proceedings of the 2nd International Workshop on Applied AI for Database Systems and Applications*, 2020.

[46] V. Pandey, A. van Renen, A. Kipf, I. Sabek, J. Ding, and A. Kemper, "The case for learned spatial indexes," *arXiv preprint arXiv:2008.10349*, 2020.

[47] R. Kang, W. Wu, C. Wang, C. Zhang, and J. Wang, "The case for ml-enhanced high-dimensional indexes," in *Proceedings of the 3rd International Workshop on Applied AI for Database Systems and Applications*, 2021.

[48] X. Zhou, C. Chai, G. Li, and J. Sun, "Database meets artificial intelligence: A survey," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[49] G. Li, X. Zhou, and L. Cao, "Ai meets database: Ai4db and db4ai," in *Proceedings of the ACM SIGMOD international conference on Management of data*, 2021, pp. 2859–2866.

[50] K. Echihabi, K. Zoumpatianos, and T. Palpanas, "New trends in high-d vector similarity search: al-driven, progressive, and distributed," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 3198–3201, 2021.