Effectively Learning Spatial Indexes with a Support for Updates

Tu Gu[1], Kaiyu Feng[1], Gao Cong[1], Cheng Long[1], Zheng Wang[1], Sheng Wang[2]
[1]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[2]DAMO Academy, Alibaba Group
{gutu0001,zheng011}@e.ntu.edu.sg,{kyfeng,gaocong,c.long}@ntu.edu.sg,sh.wang@alibaba-inc.com

ABSTRACT

Learned indices have been proposed to replace classic index structures like B-Tree with machine learning (ML) models. They require to replace both the indices and query processing algorithms currently deployed by the databases, and such a radical departure is likely to encounter challenges and obstacles. In contrast, we propose a fundamentally different way of using ML techniques to build a better R-Tree without the need to change the structure or query processing algorithms of traditional R-Tree. Specifically, we develop reinforcement learning (RL) based models to decide how to choose a subtree for insertion and how to split a node when building and updating an R-Tree, instead of relying on hand-crafted heuristic rules currently used by the R-Tree and its variants. Experiments on real and synthetic datasets with up to more than 100 million spatial objects show that our RL based index outperforms the R-Tree and its variants in terms of query processing time.

1 INTRODUCTION

To support efficient processing of spatial queries, such as range queries, KNN queries and spatial join queries, spatial databases have relied on delicate indices. The R-Tree [14] is arguably the most popular spatial index that prunes irrelevant data for query processing and is currently deployed by a number of databases such as PostgreSQL. R-Trees have attracted extensive research interests [1–3, 5, 7, 12, 18, 19, 23, 24, 27, 30, 32–34, 36, 38, 40, 43].

The learned index has been proposed in [22], which proposes a recursive model index (RMI) for indexing 1-dimensional data by learning a cumulative distribution function (CDF) to map a search key to a rank in a list of ordered data objects. The idea of learned indices is also extended for spatial data [24, 29, 32, 40] and multi-dimensional data [5, 7, 27]. They usually map spatial data points in a dataset to a uniform rank space (e.g., using a space filling curve), and then learn the CDF for this dataset. Despite the success of these learned indices in improving the efficiency of processing some types of queries, they still have various limitations. (1) To the best of our knowledge, existing learned indices [5, 7, 24, 27, 32, 40] can only handle point objects, but not other types of spatial objects. (2) Some of them [5, 7, 27] do not consider KNN queries or spatial join queries, which are important spatial queries, while some others [32, 40] do not return accurate query results. (3) Some learning based spatial data indices [7, 27] do not consider updates while some others [24, 32] need to frequently retrain their models. Furthermore, they need to replace both the index structures and query processing algorithms currently used by the spatial database systems, and such a radical departure, together with these limitations, would make it difficult for them to be deployed in current database systems. Section 5 covers more discussions on them.

In this work, rather than learning a CDF for spatial data, we consider a fundamentally different approach, i.e., to use machine learning (ML) techniques to construct a better R-Tree in a data-driven way for better query efficiency in a dynamic environment where updates occur frequently and bulk loading is not viable. Specifically, we propose to build ML models for the two key operations of building and updating an R-Tree, i.e., ChooseSubtree and Split, which currently rely on hand-crafted heuristic rules. Our proposed method has several salient features. (1) The learning based index can handle any spatial object, such as rectangular objects. (2) The R-Tree structure is not modified and thus all the currently deployed query processing algorithms will remain applicable. This would make it easier for the learning based index to be deployed by current databases. (3) The learning based index returns accurate query results. (4) The learning based index is designed for dynamic environments and can readily handle updates.

To motivate our idea, we next revisit ChooseSubtree and Split. When inserting a new spatial object, the ChooseSubtree operation is invoked iteratively, i.e., choosing which child node to insert the new data object, until a leaf node is reached. If the number of entries in a node exceeds the capacity, the Split operation is invoked to divide the entries into two groups. Many R-Tree variants have been proposed, which mainly differ in their ChooseSubtree and Split algorithms. Almost all these strategies are based on hand-crafted heuristics. For example, in R-Tree [14], ChooseSubtree utilizes the heuristic rule of inserting a new data object into a tree node whose minimum bounding rectangle (MBR) needs the least area enlargement while R*-Tree [2] adopts a more complicated heuristic rule by taking into account both area enlargement and overlap increment. However, no single heuristic strategy dominates the others in terms of query performance. We would like to use Split as an example to illustrate this. We generate a dataset with 1 million uniformly distributed data points and construct four R-Tree variants using four different Split strategies, namely linear [14], quadratic [14], Greene's [13] and R*-Tree [2]. We run 1,000 random range queries and rank the four indices based on the query processing time of each individual query. We observe that no single index has the best performance for all the queries. For example, Greene's Split has the best query performance among 50% of the queries while R*-Tree Split is the best for 49% of the queries.

This observation motivates us to handle the ChooseSubtree and Split operations by replacing or enhancing their heuristic strategies with machine learning models. Furthermore, we observe that the two operations are invoked in a sequential process when inserting a data object, so we model them as two Markov decision processes (MDPs) [31] and propose to use reinforcement learning (RL) to learn models for the two operations.

Challenges and Proposed Solutions. While this idea seems interesting, a few challenges have to be overcome to make it work.

The first challenge is how to formulate ChooseSubtree and Split as MDPs each of which consists of states, actions and reward signals.

Specifically, (1) Designing the action space. A straightforward idea could be to define the action space as a set of existing heuristic strategies. However, this idea did not work, as our experiments showed that different actions often lead to the same ChooseSubtree/Split decision, leaving us very little room for improvement. Our solution is to use the top k decision candidates to form the action space which excludes candidates that cause significant area increase for ChooseSubtree and candidates that have large total areas for Split. As bad actions are filtered out, exploration in the learning process gets more efficient. (2) Designing the state. A straightforward idea is to use the extreme values of the bounding box of the current node as the state metrics. However, our preliminary experiments showed that this state design did not work well [1]. One possible reason is that this spatial information does not allow the agent to learn the impact of each action. As a result, the agent is unable to learn to pick the favourable action. In our solution, we let the state directly reflect the impact of each decision candidate if it is chosen. Using RL Split as an example, the area and the perimeter of each of the 2 nodes resulted from each Split decision candidate form the state. Such a state design allows the RL agents to know the impact of each action and enables them to learn a good policy. (3) Designing the reward signal. It is nontrivial to design a good reward function to train the RL agents to take "good" actions. In order for the agents to learn to distinguish between "good" and "bad" actions, as a learning based index is built during the training process, we run random queries periodically and let the reward function reflect the difference in query performance between the learning based tree and an R-Tree which is used as a reference of comparison.

The second challenge is how to learn policies for the defined MDPs. Specifically, (1) In the construction of an R-Tree, node overflow (and thus the Split operation) does not occur frequently. Therefore, only a few state transitions for Split are generated, making model training inefficient. To address this issue, we first build a tree, the nodes of which are mostly full before RL Split training starts. It allows node overflow to occur frequently which then improves the training efficiency. (2) A previous decision (ChooseSubtree or Split) may affect the tree structure in the future. Therefore, a "good" action may receive a bad reward due to past bad actions. To address this problem, during model training, we always synchronize the tree that is used as the reference to the learning based tree after each reward computation. This approach enables impacts from past actions to be minimized and allows fair evaluation of actions.

Contributions. In summary, we make the following contributions:

(1) We propose to train machine learning (ML) models to replace heuristic rules in the construction and update of R-Tree to improve on its query efficiency in a dynamic environment where updates occur frequently and bulk loading is not viable. To the best of our knowledge, this is the first work that uses ML to build a better R-Tree without modifying its structure; Therefore, all currently deployed query processing algorithms are still applicable and the proposed index can be easily deployed by current databases.

(2) We model the ChooseSubtree and the Split operations as two MDPs, and carefully design their states, actions and reward signals. To shed light on the rationales behind our models, we present not only the final designs but also some unsuccessful trials of designing.

(3) We design an effective and efficient learning process that learns good policies to solve the MDPs. The learning process enables us to apply our RL models trained with a small dataset to build an R-Tree for up to more than 100 million spatial objects.

(4) Extensive experiments on both real and synthetic datasets show: (a) RLR-Tree achieves up to 95% better query performance for range queries, 74% for KNN queries and 70% for spatial join queries than R-Tree and its variants, and up to 53.8% better query performance for range queries, 44.8% for KNN queries and 57.1% for spatial join queries than LISA [24], which is the only disk based learned spatial index that returns exact results for range and KNN queries. (b) The RLR-Tree outperforms R*-Tree and RR*-Tree by up to 89.3% and 79.2% respectively, when handling updates of up to 100 million new data objects of the same distribution. It is remarkable that the RLR-Tree's performance gets better as more data objects are inserted. Furthermore, the RLR-Tree consistently outperforms R*-Tree and RR*-Tree significantly even when handling new data with changes in distribution. Additionally, when we use the RLR-Tree model trained on one real-life dataset to build an RLR-Tree on another real-life dataset, the performance of the RLR-Tree only drops slightly compared with the RLR-Tree trained and used on the same dataset. (c) RLR-Tree scales well with dimensions and its advantage becomes more significant as the number of data dimensions increases.

2 PRELIMINARY AND PROBLEM

2.1 Preliminary

R-Tree [14] is a balanced tree for indexing multi-dimensional objects, such as coordinates and rectangles and is widely deployed in databases such as PostgreSQL. Each tree node can contain at most M entries. Each node (except the root node) must also contain at least m entries. Each entry in a non-leaf node consists of a reference to a child node and the minimum bounding rectangle (MBR) of all entries within this child node. Each leaf node contains entries, each of which consists of a reference to an object and the MBR of that object. Therefore, a query that does not intersect with the MBR cannot intersect any of the contained objects.

The algorithms for building and updating an R-Tree comprise two key operations, **ChooseSubtree** and **Split**. To insert an object into an R-Tree, starting from the root node, ChooseSubtree is iteratively invoked to decide in which subtree to insert the object, until a leaf node is reached. The object is inserted into the leaf node and its corresponding MBR is updated. If the number of entries in a leaf node exceeds M, the Split operation is invoked to divide the objects into two groups: one remains in the original leaf node and the other will become a new leaf node. The Split operation may be propagated upwards as an entry referring to the new leaf node is added to its parent node, which may overflow and need to be split.

The query performance of an R-Tree highly depends on how the R-Tree is built and updated and many different R-Tree variants have been proposed. Section 5 covers more discussions on them.

2.2 Problem Statement

As discussed in Section 1, R-Tree variants mostly adopt hand-crafted ChooseSubtree and Split strategies, and no strategy can build and

[1]Due to space constraint, we do not report the details of the experiments.

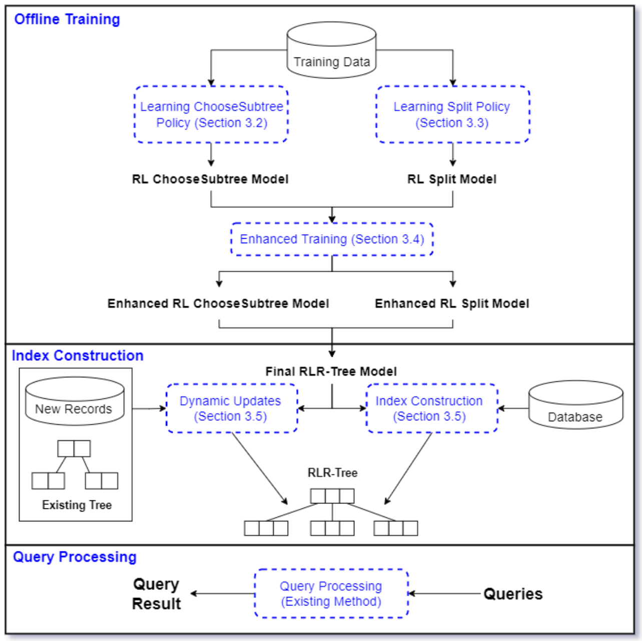

Offline Training
Training Data
Learning ChooseSubtree Policy (Section 3.2)
Learning Split Policy (Section 3.3)
RL ChooseSubtree Model
RL Split Model
Enhanced Training (Section 3.4)
Enhanced RL ChooseSubtree Model
Enhanced RL Split Model
Index Construction
Final RLR-Tree Model
New Records
Existing Tree
Dynamic Updates (Section 3.5)
Index Construction (Section 3.5)
Database
RLR-Tree
Query Processing
Query Result
Query Processing (Existing Method)
Queries

Figure 1: RLR-Tree Overview

update an R-Tree with dominant query performance in all cases. Motivated by this, we aim to learn to build and update an R-Tree, i.e., using RL models to make decisions for ChooseSubtree and Split. The new index is called RLR-Tree.

3 RLR-TREE

3.1 Overview

The process of inserting a new object into an R-Tree is essentially a combination of two typical sequential decision making processes. In particular, starting from the root, it needs to make a decision on which child node to insert the new object at each level in a top-down traversal (ChooseSubtree). It also needs to make a decision on how to split an overflowing node and divide the entries in a bottom-up traversal (Split). Reinforcement learning (RL) has been proven to be effective in solving sequential decision making problems. Therefore, we propose to model the insertion of a new object as a combination of two Markov decision processes (MDPs) and adopt RL to learn the optimal policies for ChooseSubtree and Split operations. Since databases such as PostgreSQL currently use the R-Tree [14] to index spatial data, our proposed method is readily applicable in these implementations.

Figure 1 depicts an overview of the proposed solution to build and update an RLR-Tree, as well as using the RLR-Tree to answer queries. In offline training, we propose new solutions to train RL ChooseSubtree and RL Split models using a small dataset. This is the focus of this work. The trained model can be integrated into the algorithms for R-Tree construction to build the RLR-Tree and can also be used by an existing R-Tree to handle dynamic updates. Finally, any existing query processing algorithm designed for the R-Tree family can be used for RLR-Tree to answer different spatial queries.

Next, we focus on the offline training and present our designs for the two MDPs. We present RL ChooseSubtree and its model training in Section 3.2, and RL Split and its model training in Section 3.3. We present how to train the two models together in Section 3.4. We briefly introduce how to integrate the trained models into existing algorithms to construct the RLR-Tree and then to handle updates in Section 3.5.

3.2 ChooseSubtree

To insert a new object into the R-Tree, we need to conduct a top-down traversal starting from the root. In each node, we need to decide which child node to insert the new object. To choose a subtree with RL, we formulate this problem as an MDP. We proceed to present how to train a model to learn a policy for the MDP.

3.2.1 MDP Formulation. An MDP consists of four components, namely states, actions, transitions, and rewards. We proceed to explain how states and actions are represented in our model, and then present the reward signal design, which is particularly challenging.

MDP: State Space. A state captures the environment that is taken into account for decision making. For ChooseSubtree, it is a natural idea that a state is from the tree node whose child nodes are to be selected for inserting a new object. The challenging question is: what kind of information should we extract from the tree node to represent the state?

Intuitively, as we need to decide which child node to insert the new object, it is necessary to incorporate the change of the child node if we add the new object into it for each child node. Possible features that capture the change of a child node N include: (1) $\Delta Area(N, o)$, which is the area increase of the MBR of N if we add the new object o into N; (2) $\Delta Peri(N, o)$, which is the perimeter increase of the MBR of N if we add o into N, and (3) $\Delta Ovlp(N, o)$, which is the increase of the overlap between N and other child nodes after o is inserted into N. Furthermore, it is helpful to know the occupancy rate of the child node, denoted by $OR(N)$, which is the ratio of the number of entries to the capacity. A child node with a high occupancy rate is more likely to overflow in the future. Our feature design is inspired by the classic work on designing R-Tree and its variants, and we choose features that are likely to have a significant impact on the query performance [2].

As we have presented several features to capture the properties of a tree node, a straightforward idea is that we compute the aforementioned features for every child node and concatenate them to represent the state. However, the number of child nodes varies across different nodes, making it difficult to represent a state with a vector of a fixed length. An idea to address this challenge is to do padding, i.e., to append zeros to the features of the child nodes to get a $4 \cdot M$ dimensional vector, as there are four features and there are at most M child nodes. However, the padded representations are likely to have many zeros which will add noises and mislead the model, resulting in poor performance. This is confirmed by our preliminary experiments.

To address the challenge, we propose to only use a small part of child nodes to define the state. This is because most of the child nodes are not good candidates for hosting the new object, as inserting the new object may greatly increase their MBRs. Here we aim to prune unpromising child nodes from the state space, and our RL agent will not consider them for representing a state. Our design of state representation is as follows: We first retrieve the top-k child nodes in ascending order of area increase. We also consider other alternative criteria including perimeter, overlap and occupancy rate, and they perform worse than area increase. Then for each retrieved child node, we compute four features $\Delta Area(N, o)$, $\Delta Peri(N, o)$, $\Delta Ovlp(N, o)$, and $OR(N)$. We concatenate the features of the k child nodes to get a $4 \cdot k$ dimensional vector to represent a state. k is a parameter to be set empirically. Note that to make the representation of different states comparable, the increases of area, perimeter and overlap are normalized by the maximum corresponding value among all k child nodes.

Remark. It is a natural idea that we can include more features to represent a state. For instance, we can include global information, such as the tree depth and the size of the tree, and local information, such as the depth of the tree node, the coordinates of the boundary of the MBR. However, our experiments show that these features do not improve the performance of our model while making the model training slower. The four features that we use are sufficient to train our model to make good subtree choices as shown in our experiments. It would be a useful future direction to design and evaluate other state features.

MDP: Action Space. As many R-Tree variants have been proposed with different ChooseSubtree strategies, such as minimizing the increase of area, perimeter, or overlap. A straightforward idea is to make the different cost functions from the strategies mentioned above form the action space. After trying different combinations of these cost functions and different state space designs, this idea is proven to be ineffective by our experimental results. Table 1 depicts the average relative I/O cost for processing 1,000 random range queries on three datasets of different distributions, namely Zipf, Gaussian and Uniform. For each query, the relative I/O cost of an index is computed by the ratio of the I/O cost for it to answer the query to the I/O cost for an R-Tree to answer the same query. Smaller relative I/O costs indicate better query performance compared with the R-Tree. We observe from Table 1 that compared with the R-Tree, an RL model with the cost functions as actions only achieves an improvement of less than 2% in terms of query processing time. We find from our experiments that in 90% of the nodes, different cost functions end up with the same subtree choice which gives us very little room for improvement.

Table 1: Performance of Cost Function Based Action Space.

	Relative I/O cost		
	Zipf	**Gaussian**	**Uniform**
Use cost functions	0.98	0.98	1.00
Our final design	0.20	0.18	0.56

As a result, we propose a new idea of training the RL agent to decide which child node to insert the new object directly. Based on the idea, one design is to have all child nodes to comprise the action space. However, this incurs two challenges: (1) the number of child nodes contained by different nodes is usually different, and (2) the action space is large. Considering all child nodes as the actions leads to a large action space with many "bad actions". The bad actions make the exploration during model training ineffective and inefficient. To address the challenges, we use the similar idea as we use for designing state space. Recall that in designing the state space, we propose to retrieve top-k child nodes in terms of the increase of area to represent a state. To make the action space and the state representation consistent, we define the action space $\mathcal{A} = \{1, \dots, k\}$, where action $a = i$ means the RL agent chooses the i-th retrieved child node to be inserted with the new object. The experiment on the impact of the value of k on the performance of our proposed method can be found in Section 4.2.5.

MDP: Transition. In the process of the ChooseSubtree operation, given a state (a node in the R-Tree) and an action (inserting the new object into a child node), the RL agent transits to the child node. If the child node is a leaf node, the agent reaches a terminal state.

MDP: Reward Signal. A reward associated with a transition corresponds to some feedback indicating the quality of the action taken at a given state. A larger reward indicates a better quality. Since our objective is to learn to build and to update an R-Tree that processes query efficiently, the reward signal is expected to reflect the improvement of query performance.

In the process of ChooseSubtree, it is challenging to directly evaluate if an action taken at a state is good, because the new object has not been fully inserted into the tree yet. A straightforward idea is after the new object has been inserted, we use the R-Tree to process a set of random range queries. The inverse of the cost (e.g., the number of accessed nodes) for processing the queries is set as the reward shared by all of the state-action pairs encountered in the insertion of the new object. The agent seems to be encouraged to take the actions to build a tree that can process range queries by accessing as few nodes as possible. However, this is not the case due to the following reasons: (1) A previous action may affect the tree structure and hence the query performance in the future. Therefore, a "good" action may receive a poor reward due to some bad actions that were made previously. (2) More importantly, as we aim to learn to construct an R-Tree that outperforms the competitors, we are interested in knowing what actions make the resulting tree better than a competitor and what actions make it worse. The aforementioned reward signal cannot distinguish the two types of actions, making it ineffective for the agent to learn a good policy to outperform the competitors. (3) As more objects are inserted into the R-Tree, the average number of accessed nodes naturally increases. Therefore, the reward signal becomes weaker and weaker, which makes it difficult for the model to learn useful information in the late stage of the training.

Inspired by the observations, we design a novel reward signal for ChooseSubtree. The high level idea is that we maintain a reference tree with a fixed ChooseSubtree and Split strategy. The reference tree serves as a competitor and can be any existing R-Tree variant. The reward signal is computed based on the gap between costs for processing random queries with the reference tree and the RLR-Tree. Specifically, the design of the reward signal is as follows:

(1) We maintain an RLR-Tree, that uses RL for ChooseSubtree, and adopts a pre-specified Split strategy.

(2) We maintain a reference tree which adopts a pre-specified ChooseSubtree strategy and the same Split strategy as RLR-Tree.

(3) We synchronize the reference tree with the RLR-Tree, so that they have the same tree structure.

(4) Given p new objects $\{o_1, \dots, o_p\}$, we insert them into both the reference tree and the RLR-Tree and then generate p range queries of predefined sizes whose centers are at the p objects, respectively.

(5) The p range queries are processed with both the reference tree and the RLR-Tree. We compute the normalized node access

rate, which is defined as $\frac{\text{\# acc. nodes}}{\text{Tree height}}$ and is the number of accessed nodes for answering a range query over the tree height. Let R and R' be the normalized node access rate of the RLR-Tree and the reference tree, respectively. We compute $r = R' - R$ as the reward signal. The higher r is, the fewer nodes RLR-Tree needs to access to process the range queries than the reference tree.

(6) All the transitions encountered in the insertion of the p objects share the same reward r.

With the idea, we are able to distinguish the good actions from the bad actions: A positive reward means that the RLR-Tree processes the recent p insertions well as it requires fewer nodes accesses to process the queries compared with the reference tree. Moreover, as the reference tree is periodically synchronized with the RLR-Tree, we can avoid the effect of previous actions. Therefore, maximizing the accumulated reward is equivalent to encouraging the agent to take the actions to outperform the reference tree. R-Tree is chosen as the reference because RLR-Tree reduces to R-Tree when the action space size is set to be 1. It is therefore easy to observe the improvement our proposed RL method achieves. We also tried R*-Tree as the reference tree and obtained similar results. Due to space constraint, we do not share the detailed results.

3.2.2 *Training the Agent for* ChooseSubtree.

Deep-Q-Network (DQN) Learning. Deep Q-learning is a commonly used model-free RL method. It uses a Q-function $Q^*(s, a)$ to represent the expected accumulated reward that the agent can obtain if it takes action a in state s and then follows the optimal policy until it reaches a terminal state. The optimal policy takes the action with the maximum Q-value in any state. Deep-Q-Network [26] has been proposed to approximate the Q-function $Q^*(s, a)$ with a deep neural network $Q(s, a; \Theta)$ with parameters Θ. In our model, we adopt the deep Q-learning with experience replay [26] for learning the Q-functions.

Given a batch of transitions (s, a, r, s'), parameters in $Q(s, a; \Theta)$ is updated with a gradient descent step by minimizing the mean square error (MSE) loss function, as shown in Equation 1.

$$L(\theta) = \sum_{s,a,r,s'} [(r + \gamma max_{a'} \hat{Q}(s', a'; \Theta^-) - Q(s, a; \Theta))^2], \quad (1)$$

where γ is the discount rate, and $\hat{Q}(; \Theta^-)$ is frozen target network.

Training the Agent. We present the RL ChooseSubtree training process in Algorithm 1. We first initialize the main network $Q(s, a; \Theta)$ and the target network $\hat{Q}(s, a; \Theta^-)$ with the same random weights (line 3). In each epoch, it first resets the replay memory (line 5). Then it involves a sequence of insertions of the objects in the training dataset (lines 6–20). Specifically, for every p objects $\{o_1, \ldots, o_p\}$, we synchronize the structure of T_r with T_{rl} (line 7). For each o_i of the p objects, we first insert it into the reference tree (line 9). Then a top-down traversal on the RLR-Tree is conducted (lines 10–15). At each level, we compute the state representation (line 12) and use ϵ-greedy to choose the action based on their Q-values (line 13), until we reach a terminal state (leaf node). The transitions are stored in SA (line 14). At the leaf node, we insert the new object and use the same Split strategy as the reference tree in a bottom-up scan to ensure no node overflows (line 15). Meanwhile, we generate a range query with a predefined size centered at o_i and add the query to RQ (line 16). When the p objects have been inserted, we

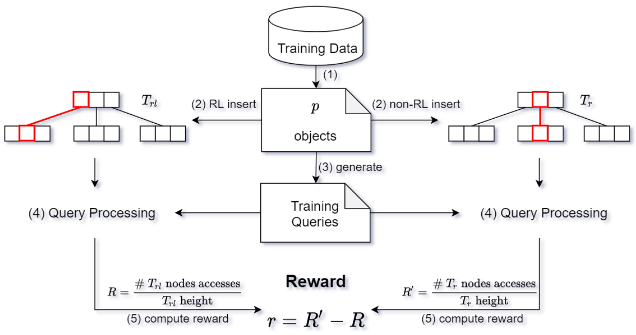


Figure 2: RL ChooseSubtree **Reward Computation**

compute the reward with the queries in RQ (line 17). The reward computation process is illustrated in Figure 2. All transitions encountered in the insertions of the p objects share the same reward r and are pushed into the replay memory (line 18). Then we draw a batch of transitions randomly from the replay memory and use the batch to update the parameters in the main network $Q(; \Theta)$ as DQN does (line 19). The parameters in the target network $\hat{Q}$ are periodically synchronized with Q (line 20).

Remark. The new object to be inserted may be fully contained in one of the child nodes. If we add the new object into such a child node, the MBRs of all child nodes are not affected. When such cases happen, we do not pass the state representation to the model, but choose the child node that contains the new object directly. The benefits of using heuristic rules in such "simple and special" cases are two-fold. Firstly, the RL model can be trained more efficiently as there is no exploration of potential bad actions. Secondly, model performance becomes better as potential bad actions are eliminated. Table 2 shows the query performances of the RL ChooseSubtree models with and without the heuristic rules respectively, on three different datasets, i.e., Zipf, Gaussian and Uniform. We observed that using heuristic rules in special cases improved the performance of RL ChooseSubtree by up to 13%. Note that the heuristic rules used in these special cases are different from the core heuristic rules used in the R-Tree and its variants, such as always inserting a data object into the node with the least MBR enlargement.

Table 2: Impact of Heuristic Rules on RL ChooseSubtree**.**

	Relative I/O cost		
	Zipf	**Gaussian**	**Uniform**
Without Heuristic Rules	0.22	0.21	0.62
With Heuristic Rules	0.20	0.18	0.56

3.2.3 *Time Complexity.*

In our analysis, the additional computation cost associated with the use of neural networks in an RLR-Tree is deemed constant. Assume the RLR-Tree has a size of S and a height of h. Inserting an object into the RLR-Tree encounters $h - 1$ states. At each state, it takes $O(k \cdot M)$ time to retrieve the top-k child nodes and $O(M)$ to compute the features for each child node. Therefore, the overall time complexity is $O(h \cdot k \cdot M)$. As a comparison, it takes $O(h \cdot M)$ time for ChooseSubtree in the R-Tree.

3.3 Split

The top-down traversal ends up at a leaf node. If the leaf node overflows, it will be split into two nodes and the Split operation

Algorithm 1: DQN Learning for ChooseSubtree

```
Input: A training dataset;
Output: Learned action-value function $Q(s, a; \Theta)$;
Initialize $Q(s, a; \Theta)$, $\hat{Q}(s, a; \Theta^-)$;
for $epoch = 1, 2, \ldots$ do
    Replay memory $\mathcal{M} \leftarrow \emptyset$;
    for every $p$ objects $\{o_1, \ldots, o_p\}$ in dataset do
        $T_r \leftarrow T_{rl}$, $SA \leftarrow \emptyset$, $RQ \leftarrow \emptyset$;
        for $o_i \in \{o_1, \ldots, o_p\}$ do
            Insert $o_i$ into $T_r$;
            $N \leftarrow$ the root of $T_{rl}$;
            while $N$ is non-leaf do
                $s \leftarrow$ state representation of $N$ and $o_i$;
                $a \leftarrow$ an action selected by $\epsilon$-greedy based on $Q$-values;
                $N \leftarrow a$, $SA \leftarrow SA \cup \{(s, a)\}$;
            Insert $o_i$ into $N$, split until no node overflows;
            $RQ \leftarrow RQ \cup$ {a range query centered at $o_i$};
        $r \leftarrow$ compute reward with queries in $RQ$;
        Add $(s, a, r, s')$ for every $(s, a) \in SA$ into memory;
        Draw samples from memory and update $\Theta$ in $Q(; \Theta)$;
        Periodically synchronize $\hat{Q}(; \Theta^-)$ with $Q(; \Theta)$;
```

may be propagated upwards. Next, we present how to model Split as an MDP and how to train the model.

3.3.1 MDP Formulation. We have also explored different ideas to design the state, action, transition and reward signal of the MDP for Split. Due to space limitation, we only present the final design.

MDP: State Space. For Split, it is natural that a state comes from an overflowing node. A straightforward idea is to make the representation of a state capture the goodness of all the possible splits, so that the agent can decide how to split the node. However, since an overflowing node contains $M + 1$ entries, there are $(2^{M+1} - 2)$ possible splits in total. It is impractical to reflect all of these splits in the state representation.

In order to avoid considering so many possible splits, we adopt a similar idea as R*-Tree [3] as follows: We first sort the entries with respect to their projection to each dimension. For each sorted sequence, we consider the split at the i-th element ($m \leq i \leq M + 1 - m$), where the first i entries are assigned to the first group and the remaining $M + 1 - i$ entries are assigned to the second group. This means for each sorted sequence, we only consider $M + 2 - 2 \cdot m$ splits. Next, we further discard the splits which create two nodes that overlap with each other. We pick the top-k splits from the remaining splits in ascending order of total area and construct the representation of the state. Specifically, for each split, we consider four features: the areas and the perimeters of the two nodes created by the split. We concatenate the features of all k splits and generate a $4k$-dimensional vector to represent the state. The areas and perimeters are normalized by the maximum area and perimeter among all splits, so that each dimension of the state representation falls in $(0, 1]$.

MDP: Action Space. Similar to ChooseSubtree, in order to make the action space consistent with the candidate splits that are used to represent the state, we define the action space as $\mathcal{A} = \{1, \ldots, k\}$, where k is the number of splits used to represent the state. An action $a = i$ means that the i-th split is adopted.

MDP: Transitions. In Split, given a state (a node in the R-Tree) and an action (a possible split), the agent transits to the state that represents the parent of the node. If the node does not overflow, it is the terminal state.

MDP: Reward Signal. The reward signal for Split is similar to that of ChooseSubtree. We maintain a reference tree that is periodically synchronized with the RLR-Tree. We use the difference of the normalized node access rate as the two trees process training queries as the reward signal. Note that the RLR-Tree adopts the same ChooseSubtree strategy as the reference tree and uses the RL agent to decide how to split an overflowing node.

Algorithm 2: DQN Learning for Split

```
Input: A training dataset;
Output: Learned action-value function $Q(s, a; \Theta)$;
Initialize $Q(s, a; \Theta)$, $\hat{Q}(s, a; \Theta^-)$;
for $epoch = 1, 2, 3, \ldots$ do
    for $j = 1, 2, 3, \ldots (parts - 1)$ do
        training part $\leftarrow \emptyset$;
        Construct $T_{base}$ with the first $\frac{j}{parts}$ of the objects (initial part);
        for $o$ in the remaining objects do
            if Inserting $o$ into $T_{base}$ causes a split then Add $o$ to training part;
            else Add $o$ to fill part and insert into $T_{base}$;
        for every $p$ objects $\{o_1, \ldots, o_p\} \subseteq$ training part do
            $T_{rl} \leftarrow T_{base}$, $T_r \leftarrow T_{base}$, $SA \leftarrow \emptyset$, $RQ \leftarrow \emptyset$;
            for $o_i \in \{o_1, \ldots, o_p\}$ do
                Insert $o_i$ into $T_r$;
                Top-down traversal on $T_{rl}$ to a leaf node $N$;
                if $N$ overflows then $RQ \leftarrow RQ \cup$ {training query};
                while $N$ overflows do
                    $s \leftarrow$ state representation of $N$;
                    $a \leftarrow$ an action selected by $\epsilon$-greedy based $Q$-values;
                    $N \leftarrow N$'s parent, $SA \leftarrow SA \cup \{(s, a)\}$
            $r \leftarrow$ compute reward with queries in $RQ$;
            Add $(s, a, r, s')$ for all $(s, a) \in SA$ to memory;
            Draw samples from memory and update $Q(; \Theta)$;
            Periodically synchronize $\hat{Q}(; \Theta^-)$ with $Q(; \Theta)$;
```

3.3.2 Training the Agent for Split. Compared with ChooseSubtree, training the agent for Split is a more challenging task. To insert an object into the R-Tree, ChooseSubtree is iteratively invoked at each level in the top-down traversal. On the other hand, Split operation is invoked only when a node overflows. Therefore, only a few transitions for Split are available for training, making it difficult for the agent to learn a good policy in reasonable time.

To tackle this challenge, we design a new method for the agent to interact with the environment, so that Split operation is frequently invoked. Theoretically, if all the nodes are full, the insertion of a new object definitely causes a tree node to split. Inspired by this, we propose to first build a tree in which most of the nodes are full, so that node splits can be frequently encountered. We generate

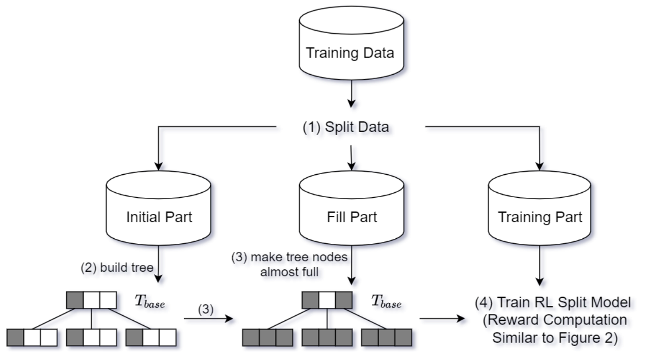


Figure 3: RL Split **Pre-Training Preparation**

such R-Trees with different sizes, and use the transitions caused by inserting the remaining objects in the dataset for training.

The procedure is presented in Algorithm 2. In each epoch, we repeat $parts - 1$ iterations to train the agent (lines 5–24). In particular, in the i-th iteration, the first $\frac{i}{parts}$ of the training dataset forms the initial part which is used to construct an R-Tree T_{base} (line 7). The remaining data are then divided into 2 parts, i.e., the fill part containing objects that will not cause node overflow in T_{base} and the training part containing objects not in the fill part. Objects in the fill part are inserted into T_{base} while objects in the training part will be used to trigger splits for training later (line 9–10). In this way, most of the nodes in T_{base} are likely to be full and the objects in the training part are likely to cause splits. This pre-training preparation process is illustrated in Figure 3. After T_{base} is constructed, we start training with objects in the training part (lines 11–24). For every p objects, we first synchronize T_r and T_{rl} with T_{base} (line 12). This makes T_r and T_{rl} have the same structure and are almost full. Then for each of the p objects, we insert it into the reference tree T_r directly with pre-specified ChooseSubtree and Split strategies (line 14). For the RLR-Tree, we use the same ChooseSubtree strategy as the reference tree to reach a leaf node N (line 15). If N overflows, we generate a range query with a predefined size centered at o_i and add the query to RQ (line 16). Then we iteratively split N and move to its parent until N does not overflow (lines 17–20). For each node N, we compute the state representation and use ϵ-greedy to select an action based on their Q-values (lines 18–19). The transitions are stored in SA (line 20). When the p objects have been processed, we compute the reward with the queries in RQ (line 21). The transitions encountered in processing the p objects share the same reward (line 22). Then we draw a batch of random transitions from the replay memory and use it to update the parameters in the main network (line 23). The parameters in the target network $\hat{Q}(;\Theta^-)$ are periodically synchronized with the main network $Q(;\Theta)$ (line 24).

Remark. When we split a node to two, the ideal case is that the two nodes do not overlap with each other, so that the number of node accesses for processing a query can be reduced. It is more challenging when there are many candidate splits that generate two nodes with zero overlap, as we need to carefully consider how to break the tie. As a result, we consider such a special case in the exploration of the agent. Specifically, if there exists at most one candidate split that generates two non-overlapping nodes, we simply select the split with the minimum overlap. We only use RL to decide how to split the node when more than one split generates non-overlapping nodes. Using heuristic rules in such "simple and special" cases improved the performance of RL Split by up to 10%. Due to the space constraint, we do not report the details of the experiment.

The training process for Split has two advantages. Firstly, using different fractions of objects to build an "almost-full" base tree helps to learn a more general model. Secondly, by building the "almost-full" base tree and periodically resetting T_r and T_{rl} to the base tree ensures that the Split operation is consistently invoked at a high frequency. This makes the training process more efficient.

3.3.3 Time Complexity. Similar to ChooseSubtree, the additional computation cost associated with the use of neural networks is deemed constant. If a node overflows, at most $O(h)$ Split operations are invoked. In each Split operation, we first sort the entries along each dimension, which takes $O(M \log M)$ time. Then it takes $O(k \cdot (M - 2m) \cdot M)$ time to retrieve the top k splits with minimum total area. Finally, it takes $O(k \cdot M)$ time to compute the four features for the k splits. Therefore, the overall time complexity for RLR-Tree Split is $O(h \cdot k \cdot (M - 2m) \cdot M)$. As a comparison, Split operation takes $O(h \cdot M^2)$ time in the classic R-Tree.

3.4 The Combined Model

A straightforward way is to train RL ChooseSubtree and RL Split separately, and then to use the learned policy to build an RLR-Tree. However, we expect the two agents to be able to help each other achieve better performance since they have the same goal. Recall that we specially design a learning process of Split, as node overflow occurs infrequently in the construction of an R-Tree. Motivated by this, we propose an enhanced training process to train the two agents alternately. Specifically, in odd epochs, we train the RL agent for ChooseSubtree and the agent for Split is fixed to be the Split strategy for the RLR-Tree. In even epochs, we train the agent for Split and the agent for ChooseSubtree is fixed to be the ChooseSubtree strategy for RLR-Tree.

3.5 RLR-Tree Construction & Dynamic Updates

With the learned models for ChooseSubtree and Split, we incorporate the models into the insertion algorithm of R-Tree to build the RLR-Tree as follows. For each object to be inserted, in the top-down traversal, we iteratively compute the state representation of a node and use the model trained for ChooseSubtree to select the subtree corresponding to the action with the maximum Q-value until the object is inserted into a leaf node. When a node overflows, we compute its state representation and use the model trained for Split to choose the split corresponding to the action with the maximum Q-value. For dynamic updates, new data records can be inserted into an existing tree using the trained models. Our experimental results (Section 4.2.4) show that the trained models do not experience significant performance deterioration even when it is not retrained frequently.

4 EXPERIMENTS

4.1 Experimental Setup

Datasets. We use 3 synthetic datasets (1-3) used in previous work on spatial indices [1, 24, 32, 33], and 2 large real-life datasets (4-5).

(1) Zipf: It consists of small squares of a fixed size. The x and y coordinates of the squares centers are randomly generated from

a Zipf distribution where the shape parameter is set to be $a = 4$ by default;

(2) Gaussian (GAU): It consists of small squares of a fixed size. The coordinates of the center of a square are (x, y) where x and y are randomly generated from a Gaussian distribution with mean $\mu = 0.5$ and standard deviation $\sigma = 0.2$;
(3) Uniform (UNI): It consists of small squares of a fixed size. The x and y coordinates of the centers of the squares are randomly generated from a uniform distribution in the range $[0, 1]$;
(4) OSM China (CHI): It contains more than 98 million locations in China extracted from OpenStreetMap;
(5) OSM India (IND): It contains more than 100 million locations in India extracted from OpenStreetMap.

Note that the centers of all spatial data objects in synthetic datasets fall within the unit square.

Queries. For model training, we run range queries over both the reference R-Tree and the RLR-Tree. Range queries of different sizes are generated. During model training, when a range query is generated, we first set its center the same as the center of the last inserted object; Its length to width ratio is then randomly selected in the range $[0.1, 10]$. For evaluation, we randomly generate 1,000 range queries for each query size ranging from 0.005% to 0.5% of the whole region [33]. Note that queries used for training and testing are generated separately and hence different.

Baselines. We compare with R-Tree and its variants that are designed for dynamic environments where updates may occur frequently. Baselines used in the experiments include R-Tree [14], which is also the reference tree used for model training, R*-Tree [2], RR*-Tree [3] which is reported to have the best query performance among R-Tree variants built using one-by-one insertion for supporting queries in dynamic environments. Note that PostgreSQL uses R-Tree. We also compare with LISA [24] which is the only disk based learned index that returns exact results for range queries and KNN queries. Note that LISA only supports point data and is not designed for dynamic environments. We test it on the two real-life datasets, which contain point objects, so as to show where RLR-Tree stands compared to this recently proposed learned spatial data index. We do not focus on any comparison with packing R-Trees that are designed for the static databases [18] or other learned indices as they are not designed for dynamic environments as discussed in Section 5. However, we include STR [23] as a reference to show what packing R-Tree can achieve with all the data available. Moreover, we conduct experiments to show that RLR-Tree can be used to maintain packing R-Trees in a dynamic environment and outperforms previous methods.

Measurements. For measurements of query performance, we consider both running time and the I/O cost. We find that both measures yield qualitatively consistent results (as exemplified in Figures 4 and 5). We follow [24] to report the average relative I/O cost mainly. For each query, the relative I/O cost of an index is computed by the ratio of the I/O cost for it to answer the query to the I/O cost for an R-Tree to answer the same query. Smaller relative I/O costs indicate better query performance compared with the R-Tree. R-Tree is chosen as the reference because the RLR-Tree reduces to the R-Tree when the action space size for both RL ChooseSubtree and RL Split is set to be 1. It is therefore easy to observe the improvement our proposed method achieves. Note that this choice does not affect the reporting of the experimental results as RLR-Tree is eventually compared with all the baselines.

Parameter settings. Table 3 shows a list of parameters and their corresponding values tested in our experiments. The default settings are bold. For all R-Tree variants evaluated in this paper, we follow [3] and maintain a maximum of 50 and a minimum of 20 child nodes per tree node.

Table 3: Parameters and Values

Parameters	Values
Data distribution	Zipf, **GAU**, UNI
Dataset size (million)	1, **25**, 50, 75, 100
Training set size (thousand)	25, 50, **100**, 200
Training query size (%)	0.005, **0.01**, 0.5
Testing query size (%)	0.005, **0.01**, 0.05, 0.1, 0.5
Action space size k	**2**, 3, 5, 10
Number of dimensions	**2**, 4, 6, 8, 10

The DQN models for both ChooseSubtree and Split contain 1 hidden layer of 64 neurons with SELU [21] as the activation function. In the training process, the learning rate is set to be 0.003 for RL ChooseSubtree and 0.01 for RL Split. The initial value of ϵ is set to be 1 and the decay rate is set to be 0.99. The value of ϵ is never allowed to be less than 0.1 in order to maintain a certain degree of exploration throughout model training. The replay memory can contain at most 5,000 (s, a, r, s') tuples. Network update is done by first sampling a batch of 64 tuples from the replay memory. Then Θ is updated by using gradient descent of the MSE loss function to close the gap between the Q-value predicted by Θ and the optimal Q-value derived from Θ^-. The discount factor is set to be 0.95 for RL ChooseSubtree and 0.8 for RL Split. Synchronization of Θ^- with Θ is done once every 30 network updates.

During the model training for RL ChooseSubtree (resp. RL Split), the deterministic splitting (resp. insertion) rules are set to be the same as that used by the reference tree which is minimum overlap partition (resp. minimum node area enlargement). We train the RL ChooseSubtree and Split models for 20 and 15 epochs, respectively, and set *parts* in Algorithm 2 to be 15, i.e., the training dataset is divided into 15 equal parts. The action space size k for both RL ChooseSubtree and RL Split is set to be 2 by default. Note that the trivial case of $k = 1$ simply gives us the reference tree.

We train our models on NVIDIA Tesla V100 SXM2 16 GB GPU using PyTorch 1.3.1. All indices are coded using C++.

4.2 Experimental Results

Our experiments aim to find out:

(1) Can RL ChooseSubtree and RL Split individually build better R-Trees (Section 4.2.1 and Section 4.2.2)?
(2) Can RLR-Tree outperform the baselines for range queries, KNN queries and spatial join queries (Section 4.2.3)?
(3) How well RLR-Tree handles updates in dynamic environments (Section 4.2.4)?
(4) The effect of different parameters on performance such as the training dataset size, the action space size of the RL models and the training query size (Section 4.2.5);

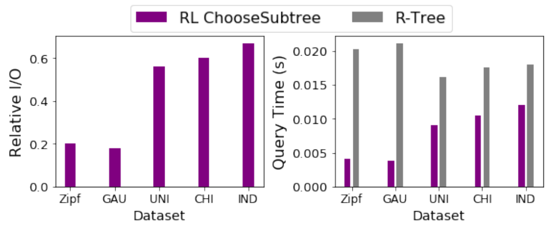


Figure 4: Performance of RL ChooseSubtree

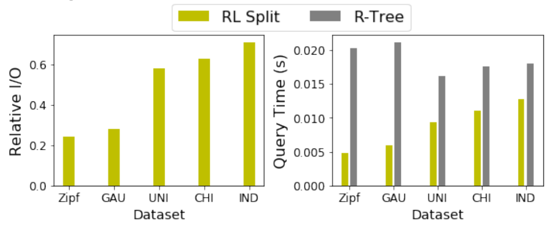


Figure 5: Performance of RL Split

(5) How well RLR-Tree can be incorporated into a packing R-Tree, how well RLR-Tree scales with dimensions and how large RLR-Tree's construction time and size are (Section 4.2.6)?

4.2.1 RL ChooseSubtree. Figure 4 reports the average relative I/O cost of the RL ChooseSubtree on all the 5 datasets. In this set of experiments, we set the dataset size of the 3 synthetic datasets to be 100 million so that all the 5 datasets have similar sizes. RL ChooseSubtree achieves significant improvements and outperforms R-Tree consistently on all datasets. The best relative I/O cost is 0.18 and observed on the GAU dataset. It is remarkable that although the training dataset size is only 100,000, the trained model can be applied on large datasets successfully.

Query Processing Time. Figure 4 also reports the query processing time of RL ChooseSubtree. We observe that the ratio of the query time of RL ChooseSubtree to that of R-Tree is generally consistent with the relative I/O cost on all the 5 datasets. For example, on the Zipf dataset, the relative I/O cost for RL ChooseSubtree is 0.2 while the query time for RL ChooseSubtree is about 1/5 of that for R-Tree. Therefore, for the remaining of the paper, we follow [24] and report only relative I/O cost as a measure of query performance.

4.2.2 RL Split. To evaluate the performance of RL Split, Figure 5 reports results on all the 5 synthetic datasets. Similar to Section 4.2.1, we also set the dataset size of the 3 synthetic datasets to be 100 million. RL Split achieves significant improvements and outperforms the R-Tree consistently on all the 5 datasets. The best relative I/O cost is 0.24 and observed on the Zipf dataset.

Query Processing Time. Figure 5 also reports the query processing time of RL Split. Similar to RL ChooseSubtree, I/O cost and query time show consistent results.

4.2.3 RLR-Tree. This set of experiments is to evaluate the performance of RLR-Tree, which is constructed from a combined RL ChooseSubtree and RL Split model.

The Enhanced Training Process. We first compare the performance of RL ChooseSubtree, RL Split, Naive RLR-Tree, which is obtained by directly applying RL ChooseSubtree and RL Split, and RLR-Tree, which uses the enhanced training process (Section 3.4), on all the 5 datasets in terms of relative I/O cost. As shown in Figure 6, by applying the enhanced training process, RLR-Tree has the best performance on all datasets. Specifically, the enhanced

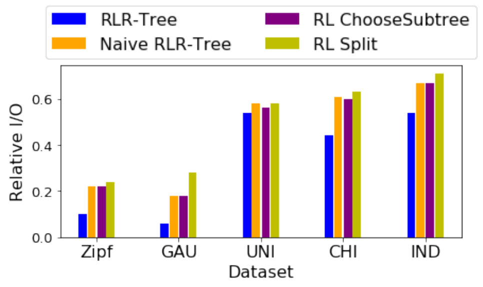


Figure 6: RL ChooseSubtree, RL Split and RLR-Tree

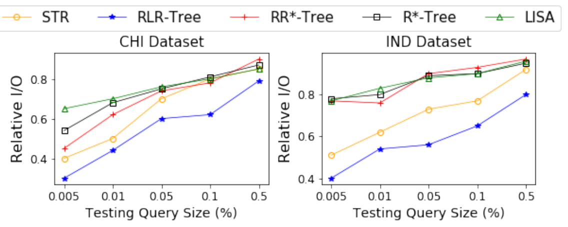


Figure 7: RLR-Tree Performance (Range Queries)

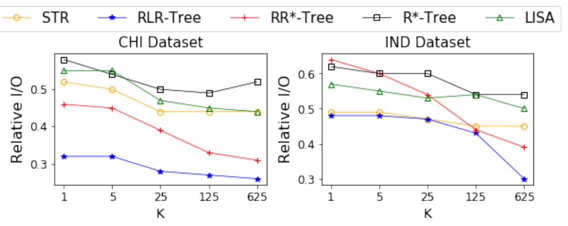


Figure 8: RLR-Tree Performance (KNN Queries)

training process achieves up to 66.7% of improvements in query performance compared to Naive RLR-Tree. The most significant improvement is observed on the GAU dataset.

Range queries. We evaluate the query performance of the RLR-Tree on both real datasets using range queries of different sizes and report the results in Figure 7. Note that we also include STR [23] to observe where RLR-Tree stands, although packing R-Trees are not designed for dynamic environments. We make 2 main observations. Firstly, RLR-Tree consistently outperforms the baselines, including the R*-Tree, the RR*-Tree, LISA and even STR. Specifically, RLR-Tree outperforms R*-Tree, RR*-Tree, LISA and STR by up to 48.7%, 48%, 53.8% and 25.0% respectively. Secondly, RLR-Tree's advantages over baselines get increasingly significant as query size decreases. This is because a larger query intersects with a larger number of tree nodes, and more tree nodes will then be traversed when answering it. In this case, all R-Tree based indices need to visit a larger portion of the data and the difference between different indexing techniques will diminish. Similar results are also reported in [33]. Consider an extreme case where a range query covers the entire data space. Then all tree nodes will be traversed, irrespective the index used, and thus relative I/O cost will be close to 1.

KNN queries. To evaluate the performance of the RLR-Tree for types of queries that are not used in the model training process, we look into the K-Nearest-Neighbor (KNN) queries. A KNN query returns the K nearest objects to a given query point. In our experiments, we consider different K values, i.e. $K \in \{1, 5, 25, 125, 625\}$.

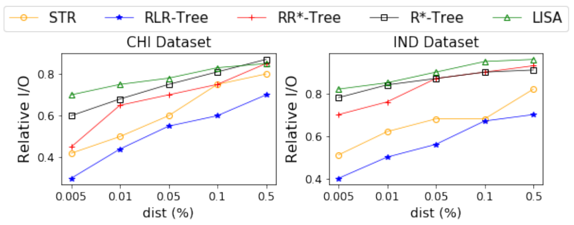


Figure 9: RLR-Tree Performance (Spatial Join Queries)

For each K value, 1,000 uniformly distributed query points are randomly generated in the data space. We use the method proposed in [16], which is the state-of-the-art KNN query processing algorithm to compute KNN queries accurately. We would like to highlight that we also tested the algorithm proposed in [35] and observed similar results. Due to space constraint, we do not report the details of the experiments results obtained using the method proposed in [35]. Note that we also include STR [23] for reference. Figure 8 shows that the RLR-Tree consistently outperforms all the baselines for all values of K on both real datasets. We also observe that the relative query performance of RLR-Tree to R-Tree gets better for larger K values. The finding that the RLR-Tree also outperforms the baselines is particularly interesting — the RLR-Tree is designed and trained to optimize the performance of range queries, rather than KNN queries. Additionally, we compute the reward and design the state features for the RLR-Tree in a way such that the model is trained to minimize the number of nodes accesses when answering range queries and not KNN queries. However, despite those design features that do not favour KNN queries, RLR-Tree still has the best performance for KNN queries.

Spatial join queries. In this experiment, we evaluate the performance of the RLR-Tree on both real datasets for the processing of spatial join queries. For a query set QS and a dataset DS, a spatial join query returns all pairs of objects (qo, do), where $qo \in QS$ and $do \in DS$, such that the Euclidean distance between qo and do is less than a given threshold $dist$ [17]. In our experiments, we consider different $dist$ values (as a percentage of the size of the data space along dimension x), i.e. $dist \in \{0.005\%, 0.01\%, 0.05\%, 0.1\%, 0.5\%\}$. For each $dist$ value, we consider a query set containing 1,000 random query points. To process spatial join queries, we use the index nested-loop join algorithm described in [17] where the dataset is indexed and the query set is not indexed. Figure 9 shows that RLR-Tree consistently outperforms the baselines and its advantage is more significant for smaller values of $dist$. Note that these trends are generally consistent with that observed for range queries.

4.2.4 *Dynamic Updates.*

We aim to evaluate the performance of the RLR-Tree when handling updates in dynamic environments.

Updates without change in Data Distribution. In the first set of experiments, we follow previous work such as [43] and assume that future unseen data follow the same distribution as existing data. We first train and build 2 RLR-Trees of size 100 thousand using the GAU dataset and the Zipf dataset with their default settings respectively. Then up to 100 million data objects from the same data distributions are inserted into the respective trees using the corresponding trained models. As shown in Figure 10, RLR-Tree consistently outperforms the baselines and RLR-Tree's advantage over the baselines remain significant throughout. Specifically, the RLR-Tree outperforms R*-Tree and RR*-Tree by up to 89.3% and

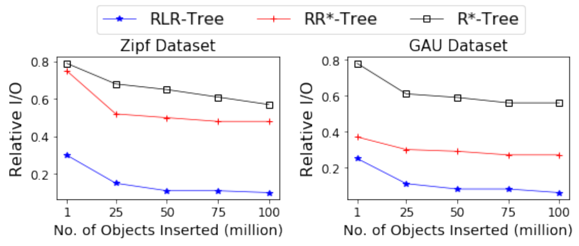


Figure 10: Updates without Change in Data Distribution

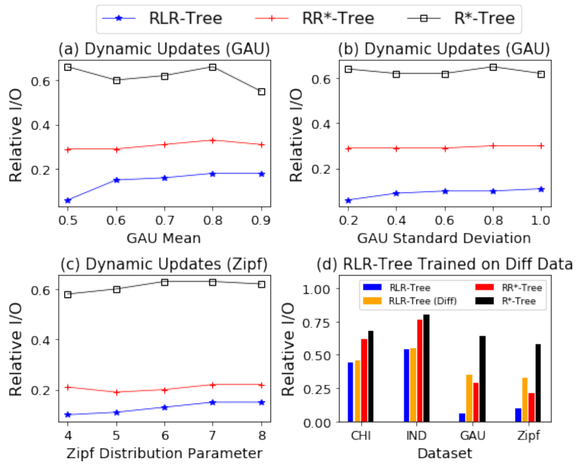


Figure 11: Updates with Change in Data Distribution

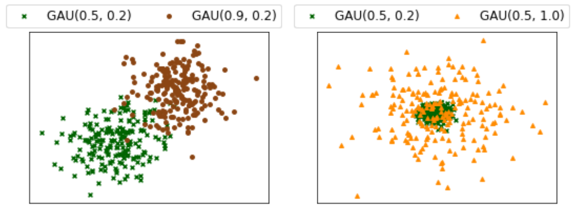


Figure 12: Illustration of Change in Gaussian Distribution

79.2% respectively. It is remarkable that the model performance gets better as more data objects are inserted. This could be because as more data objects are inserted, the index tree becomes larger and more RL ChooseSubtree and RL Split operations occur. Hence, the accumulated benefits from these RL operations enable RLR-Tree to outperform the R-Tree more significantly.

Updates with change in Data Distribution. As mentioned earlier, previous work mostly assumes that future unseen data follow the same distribution as existing data. However, we would like to take into consideration a certain degree of change in data distribution so as to show the robustness of our proposed method. In the second set of experiments, we first train and build 2 RLR-Trees of size 100 thousand using the GAU ($\mu = 0.5$, $\sigma = 0.2$) dataset and the Zipf ($a = 4$) dataset with their default settings respectively. For the RLR-Tree built using the GAU dataset, we use the GAU-trained model to insert up to 100 million spatial objects from GAU distribution with different mean ($\mu \in \{0.5, 0.6, 0.7, 0.8, 0.9, 1\}$) and standard deviation ($\sigma \in \{0.2, 0.4, 0.6, 0.8, 1\}$), and report the average relative I/O cost of 1,000 random queries in Figure 11a and Figure 11b, respectively. For the RLR-Tree built using the Zipf dataset, we use the Zipf-trained model to insert up to 100 million spatial objects from Zipf distribution with different distribution parameter ($a \in \{4,5,6,7,8\}$), and report the average relative I/O cost of 1,000 random queries in Figure 11c. The scatter plots in Figure 12 illustrates the changes in data distribution for the default GAU dataset. It shows that significant changes in data density and data locations are considered in the experiments. The results show that, even without retraining the RL model, it consistently outperforms the baselines despite slight deterioration of performance amid changes in data distributions. In contrast, LISA [24] needs to retrain its model when the number of data objects to insert is 4 times the size of the initial dataset even when there is no change in data distribution. One possible reason for the resilient performance of the RLR-Tree is that it does not rely on any learned CDF, and some of the knowledge initially learned by the model remains useful even if the data distributions are changed.

Applying Trained Models on Different Dataset. We attempt to train RL ChooseSubtree and RL Split models on one dataset and use the trained models to build RLR-Tree on a different dataset. In Figure 11d, for the 2 real datasets, IND and CHI, the yellow bars represent the models trained on the IND (resp. CHI) dataset and then applied on the CHI (resp. IND) dataset; for the 2 synthetic datasets, GAU and Zipf, the yellow bars represent the models trained on the GAU (resp. Zipf) dataset and then applied on the Zipf (resp. GAU) dataset. For the 2 real datasets, although the models are trained on a different real dataset, the constructed RLR-Trees still outperform all baselines. Compared with the models that are trained and applied on the same real dataset, these RLR-Trees only experience a small degree of performance deterioration. This is perhaps because these real datasets share common features as they mostly consist of "clusters" of high data density which represent developed regions surrounded by vast regions of low data density which represent rural areas. For the 2 synthetic datasets, as we train the models on a different synthetic dataset, the constructed RLR-Trees outperform R*-Tree but are slightly outperformed by RR*-Tree. It shows that the query performance of the trained models tends to deteriorate more significantly when data changes are more drastic.

4.2.5 *The Effects of Parameters.*

The tuning of the parameters is done on a training dataset without touching any testing dataset. Specifically, we evaluate specific parameters values by using the trained RL models to build an RLR-Tree on the training dataset and then comparing with the reference tree, which is an R-Tree. Note that after we tune the parameters on the training dataset, we use the same parameters values for all the other datasets in the experiment, i.e., we do not tune parameters for each dataset. Small deviations from the default values do not have a significant impact on the model performance.

Training Dataset Size. This experiment is to evaluate the effect of training dataset size on the performance of the RLR-Tree. We would expect better query performance if we train the RL ChooseSubtree and RL Split models on the full dataset. However, the training is slow on large datasets. Instead, we propose to use a small training dataset. Experimental results are shown in Figure 13. First, we observe that the training time of RL ChooseSubtree model for different data distributions is similar, and increases significantly with the size of the training dataset. As expected, the query performance of the trained models improves as the training dataset size increases from 25,000 to 100,000. However, the query performance becomes

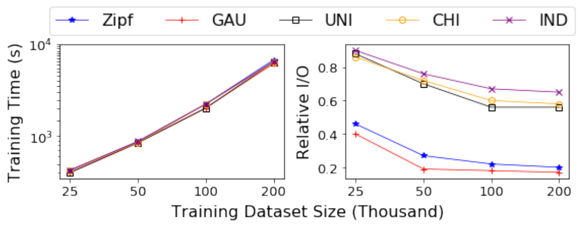
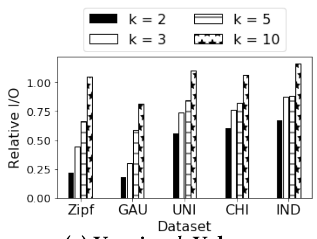
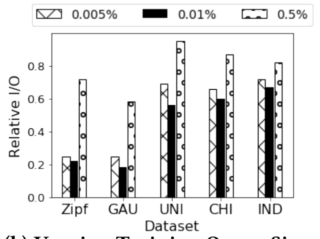


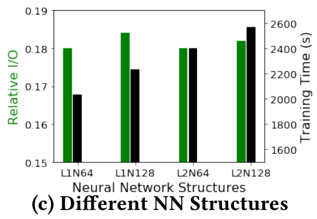
Figure 13: Effect of Varying Training Dataset Sizes



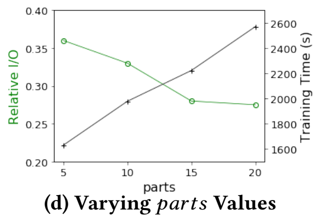
(a) Varying k Values



(b) Varying Training Query Sizes



(c) Different NN Structures



(d) Varying *parts* Values

Figure 14: The Effects of Parameters

stable after the dataset size reaches 100,000, and the improvement of using training dataset of size 200,000 over 100,000 is not significant. The results for RL Split are qualitatively similar to those for RL ChooseSubtree. Therefore, we set the training dataset size to be 100,000 by default which achieves a good tradeoff between training time (generally no more than 40 minutes) and query performance. Note that RLR-Tree only needs to be trained once on a small training dataset and the learned models can be used on large dataset to build index and then to handle updates.

The Value of k. We would expect the action space size k to have a direct impact on RLR-Tree's performance. On the one hand, as more candidates are shortlisted to form the action space, more actions are available to be selected for the trained model. On the other hand, model performance can be adversely affected when the action space is large as the trained model may not do a good job to filter out "bad" candidates. To find a good k value, we test different values of k on all the 5 datasets, using RL ChooseSubtree as an example. From Figure 14a, we observe qualitatively similar trends on all the 5 datasets. We make 2 observations. Firstly, recall that $k = 1$ is the trivial case that generates the reference tree. By including one additional candidate in the action space, i.e., when $k = 2$, we achieve significant query performance improvement for RL ChooseSubtree. The best performance improvement is 82% on the GAU dataset. Secondly, RL ChooseSubtree has the best result at $k = 2$ for all the 5 datasets. As k value increases, model performance deteriorates gradually which is expected. When k value approaches and exceeds 10, the RL ChooseSubtree model starts to fail to outperform the R-Tree. Similar trends are also observed for RL Split.

Training Query Size. This experiment is to evaluate the effect of the training query size on the performance of RLR-Tree. Figure 14b reports the average query performance of RL ChooseSubtree for each training query size for each dataset. We observe rather poor performance of RL ChooseSubtree when using the largest training query size, i.e. 0.5%. For example, on the GAU dataset, the model performance with training query size of 0.5% is more than 3 times poorer than that with our default setting of 0.01%. When using training query size of 0.005%, model performance shows slightly poorer results on all 5 datasets. Therefore, we set the training query size to be 0.01% by default. We observe similar trends for RL Split.

Neural Network Structure. This experiment is to evaluate the effect of the neural network (NN) structure of the RL model on the performance of RLR-Tree. Figure 14c reports the query performance (green bars) and the model training time (black bars) of RL ChooseSubtree for different NN structures for the default GAU dataset. Note that L*a*N*b* represents a neural network that consists of *a* hidden layers of *b* neurons. We observe quite similar model performance for different NN structures. However, as the NN structure gets more complicated, model training time increases significantly. Therefore, we set the NN structure to be L1N64 by default. We observe similar trends for RL Split.

The Value of *parts*. We would expect parameter *parts* (Section 3.3.2, Algorithm 2) to have a direct impact on the performance and the training time of RL Split. When *parts* is larger, there are more base trees (T_{base}) in each training epoch. As the model is trained using a larger number of base trees of different sizes, we could expect model training to be more effective. However, building more base trees (T_{base}) leads to a larger overhead in model training time. Figure 14d reports the query performance (green line) and the model training time (black line) of RL Split for different *parts* values for the default GAU dataset. We observe that as *parts* increases from 5 to 15, the performance of RL Split improves steadily. As *parts* increases from 15 to 20, there is no more significant performance improvement. We also observe significantly larger model training time for larger values of *parts*. Therefore, we set the value of *parts* to be 15 by default, which achieves a good balance between model performance and training time.

4.2.6 Other Experiments.

Using RLR-Tree Models to Handle Updates on a Packing R-Tree. Packing methods can be used to build an R-Tree when data is available at the time of building an index. However, to maintain a packing R-Tree in the presence of updates, we still need to use one-by-one insertion methods. Furthermore, database systems such as PostgreSQL use R-Tree [14] to index spatial data. As a result, one-by-one insertion methods for updates remain relevant and vital. This experiment is to investigate how well the RLR-Tree model can handle updates on an existing packing R-Tree. We first use the STR packing method [23] to build 2 trees of size 1 million using the default GAU and Zipf datasets respectively. Then using corresponding RLR-Tree models, we insert up to 100 million spatial objects from corresponding datasets to the 2 trees, and report the average relative I/O cost of 1,000 random queries, respectively in Figure 15. On the same packing R-Tree, we compare different one-by-one insertion methods, i.e., RLR-Tree, RR*-Tree and R*-Tree, that are used for dynamic updates. In this experiment, we also compare with STR [23] as a reference. We observe that compared with other one-by-one insertion methods for updates, the advantage of RLR-Tree gets more significant as more data objects are inserted.

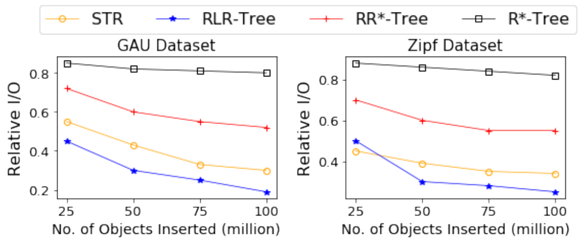


Figure 15: Incorporation of RLR-Tree into Packing R-Tree

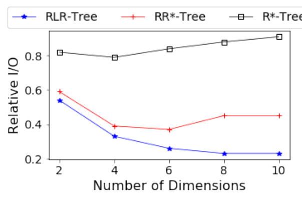


Figure 16: Changing Dimensions

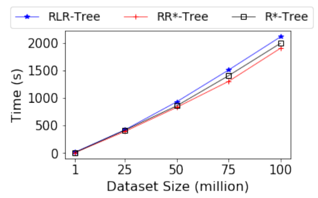


Figure 17: Index construction time

Table 4: Index size for GAU datasets

Dataset size (million)	1	25	50	75	100
RLR-Tree size (MB)	39	975	1950	2925	3900

Number of Dimensions. This experiment is to show how well RLR-Tree scales with dimensions. We vary the number of dimensions from 2 to 10. For each case, we generate a synthetic dataset with 25 million objects from the Uniform distribution, which is commonly used for high dimensional synthetic datasets in previous works [3, 27]. For each dataset, we report the average relative I/O cost for answering 1,000 random queries in Figure 16. The average data selectivity of the query workload for each dataset is kept constant at 0.01%. We observe that RLR-Tree consistently outperforms all baselines all the time. Moreover, its advantage becomes more significant as the number of dimensions increases, which illustrates the robustness of RLR-Tree w.r.t. the number of dimensions.

Index Construction Time and Size. The index construction time is similar for different data distributions of the same size, and we use the GAU dataset as an example. As shown in Figure 17, index construction time increases almost linearly with increase in dataset size. The construction time of RLR-Tree is comparable to that of R*-Tree and RR*-Tree. And we expect RLR-Tree construction time to be shorter when using better GPU devices. For index size, the RLR-Tree and other baselines are similar. Therefore, we report the RLR-Tree sizes for the GAU dataset of different sizes. As shown in Table 4, RLR-Tree size increases linearly as dataset size increases.

5 RELATED WORK

5.1 Spatial Indices

We next introduce three categories of spatial indices.

R-Tree and its variants. The category of indices partitions the dataset into subsets and indexes the subsets. Typical examples include the R-Tree and its variants, such as R*-Tree [2], R^{+}-Tree [38], and RR*-Tree [3]. There also exist works that explore the query workload such as QUASII [30] and "AI+R"-Tree [15]. QUASII [30] is a main memory based spatial data index that is built by optimizing on a known query workload. Note that QUASII only indexes the subset of the data queried by the given query workload. "AI+R"-Tree[15] does not build a better R-Tree. Instead, it takes as input a traditional R-Tree and a given query workload, and proposes to use ML techniques to improve the query processing algorithm for range queries. In contrast, our proposed method aims to build a better R-Tree (not to propose any new query processing algorithm), and the RLR-Tree does not need query workloads as input.

Packing. An alternative way of R-Tree construction is to pack data objects into leaf nodes and build the tree bottom up. The original R-Tree and almost all of its variants are designed for a dynamic environment, being able to handle updates, while packing methods are for static databases [18]. Packing needs to have all the data before building indices, which is not always available. The existing packing methods explore different ways of sorting spatial objects to achieve better ordering of the objects and eventually better packing. Some ordering methods [1] are based on the coordinates of objects, such as STR [23], TGS [12] and the lowx packed R-Tree [36] and other ordering methods are based on space filling curves such as z-ordering [28, 33], Gray coding [8] and the Hilbert curve [9].

Other spatial indices. In space partitioning based indices, such as kd-Tree [4] and Quad-Tree [11], the space is recursively partitioned until the number of objects in a partition reaches a threshold.

5.2 Learned Indices

Learned one-dimensional indices. The idea behind learned indices [6, 10, 20, 22, 42] is to learn a function that maps a search key to the storage address of a data object. The idea of learned indices is first introduced by [22], which proposes the Recursive Model Index (RMI) to learn the data distribution. It essentially learns a cumulative distribution function (CDF) using a neural network to predict the rank of a search key.

Learned spatial indices. Several learned spatial indices have been proposed. The Z-order model [40] extends RMI to spatial data by using a space filling curve to order data points and then learning the CDF. Recursive spatial model index (RSMI) [32] further develops the Z-order idea [40]. LISA [24] is a disk-based learned index. It partitions the data space with a grid, numbers the grid cells, and learns a data distribution based on this numbering. Similar to Z-order model [40] and RSMI [32], Flood [27] also maps a dataset to a uniform rank space before learning a CDF. Differently, it utilizes workload to optimize the learning of the CDF, and it learns the CDF of each dimension separately. Tsunami [7] overcomes the limitation of Flood in handling skewed workload and correlated data. The ML-Index [5] maps point objects to a one-dimensional space and then learns the CDF. Zacharatou et al. [45] shares the vision of incorporating distance-bounded spatial approximations in spatial databases, which has great potential to improve the query processing of learned spatial indices.

Remark. These learned indices all aim to learn a CDF for a particular data to replace the traditional indices. However, RLR-Tree is fundamentally different - Instead of learning any CDF, we train RL models to handle ChooseSubtree and Split operations. Furthermore, these learned indices have the following limitations compared with our solution: First, they can only handle spatial point objects while our proposed method can handle any spatial object, such as rectangular objects. Second, they all need customized algorithms to handle each type of query. They focus on certain types of queries and it is not clear how they can process other types of queries. For example, some of them [5, 7, 27] do not consider KNN or spatial join queries, which are important spatial queries. Some learned indices [24] extend their algorithm for range queries to handle KNN queries by issuing a series of range queries until k points are found. However, the query performance largely depends on the size of the region used. In contrast, RLR-Tree simply uses existing query processing algorithms for R-Tree to handle different types of queries. Third, some of these learned indices [32, 40] return approximate query results while our query results are accurate. Fourth, both Flood and Tsunami need the query workload as the input while RLR-Tree does not assume the query workload to be known. Fifth, updates are not discussed for Flood, Tsunami or the ML-Index. Although RSMI [32] and LISA [24] can handle updates, their models have to be retrained periodically to retain good query performances. In contrast, RLR-Tree readily handles updates without the need to frequently retrain the model. Finally, unlike existing learned indices, our proposed method makes no modification to the R-Tree structures currently deployed by databases, and hence is readily applicable in their implementations.

5.3 Applications of Reinforcement Learning

To the best of our knowledge, no previous work uses RL to build a better R-Tree. RL has been successfully used to solve database problems, such as tuning tasks [25, 39, 46], similarity search [41], join order selection [44], index selection [37], and data partitioning [43]. In particular, QD-Tree [43] is proposed to partition data into blocks such that the size of each block is larger than a predefined size b and the data skipping ratio for a given query workload is maximized. QD-Tree is unsuitable for spatial data indexing for the following reasons. (1) QD-Tree does not have a mechanism that generates nodes/blocks of a fixed size to build a balanced tree, and maintains the fixed-size nodes in the presence of updates, which is desirable for database indices. (2) The complexity of training a QD-Tree is prohibitively expensive if it is used for indexing. In the worst case, its complexity is $O(e*n^2/b+e*f+e*|Q|*n/b)$, where n is the data size, b is the block size, e is the number of training episodes, f denotes the cost of one neural network forward pass, and $|Q|$ is the size of the query workload. Since QD-Tree is designed to partition data into large blocks each of which contains more than 100k tuples [43], the complexity might be acceptable. In our preliminary experiments of using QD-Tree to process range queries, if we follow [43] and set $b = 100$k, the performance is orders of magnitude worse than R-Tree. On the other hand, if we set $b = 50$ as other indices considered in this paper, on the IND dataset which contains 100 million objects, QD-Tree was trained for more than 72 hours and no convergence was observed. (3) QD-Tree relies on a known query workload, which may not be available. The query workload determines the action space of QD-Tree and there is no guarantee that there always exists an action to partition an internal node. As a result of these differences, the MDP design of RLR-Tree is fundamentally different from that of QD-Tree.

6 CONCLUSIONS AND FUTURE WORK

We propose an RL based method to construct a better R-Tree, i.e., the RLR-Tree. Experimental results show that RLR-Tree consistently outperforms the R-Tree variants for range queries, KNN queries and spatial join queries. We would like to highlight that RLR-Tree has especially remarkable performance on larger datasets and is able to handle dynamic updates amid certain changes in data distribution.

This work takes a first step to use machine learning to build a better R-Tree, and we believe it would open a few directions for future work: 1) further explore and refine the designs of the states, the actions and the reward signal; 2) extend the idea to index enriched spatial data, such as spatial-temporal data, moving objects, and spatial-textual data; 3) explore the application of other machine learning models in the domain of spatial data indexing.

REFERENCES

[1] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. 2008. The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Transactions on Algorithms (TALG)* 4, 1 (2008), 1–30.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 322–331.

[3] Norbert Beckmann and Bernhard Seeger. 2009. A revised r*-tree in comparison with related index structures. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 799–812.

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

[5] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries.. In *EDBT*. 407–410.

[6] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

[7] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *arXiv preprint arXiv:2006.13282* (2020).

[8] Christos Faloutsos. 1986. Multiattribute hashing using gray codes. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*. 227–238.

[9] Christos Faloutsos and Shari Roseman. 1989. Fractals for secondary key retrieval. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 247–252.

[10] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 10 (2020), 1162–1175.

[11] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4, 1 (1974), 1–9.

[12] Yván J García R, Mario A López, and Scott T Leutenegger. 1998. A greedy algorithm for bulk loading R-trees. In *Proceedings of the 6th ACM international symposium on Advances in geographic information systems*. 163–164.

[13] Diane Greene. 1989. An implementation and performance analysis of spatial data access methods. In *Proceedings. Fifth International Conference on Data Engineering*. IEEE Computer Society, 606–607.

[14] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[15] Ch Md Rakin Haider, Jianguo Wang, Walid G Aref, et al. 2022. The "AI+ R"-tree: An Instance-optimized R-tree. In *2022 23rd IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 9–18.

[16] Gisli R Hjaltason and Hanan Samet. 1999. Distance browsing in spatial databases. *ACM Transactions on Database Systems (TODS)* 24, 2 (1999), 265–318.

[17] Edwin H Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM Transactions on Database Systems (TODS)* 32, 1 (2007), 7–es.

[18] Ibrahim Kamel and Christos Faloutsos. 1993. On packing R-trees. In *Proceedings of the second international conference on Information and knowledge management*. 490–499.

[19] Kothuri Venkata Ravi Kanth, Divyakant Agrawal, Ambuj K Singh, and Amr El Abbadi. 1997. Indexing non-uniform spatial data. In *Proceedings of the 1997 International Database Engineering and Applications Symposium (Cat. No. 97TB100166)*. IEEE, 289–298.

[20] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–5.

[21] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. 2017. Self-normalizing neural networks. *Advances in neural information processing systems* 30 (2017), 971–980.

[22] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[23] Scott T Leutenegger, Mario A Lopez, and Jeffrey Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings 13th International Conference on Data Engineering*. IEEE, 497–506.

[24] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2119–2133.

[25] Xi Liang, Aaron J Elmore, and Sanjay Krishnan. 2019. Opportunistic view materialization with deep reinforcement learning. *arXiv preprint arXiv:1903.01363* (2019).

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[27] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 985–1000.

[28] Jack A Orenstein. 1986. Spatial query processing in an object-oriented database system. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*. 326–336.

[29] Varun Pandey, Alexander van Renen, Andreas Kipf, Ibrahim Sabek, Jialin Ding, and Alfons Kemper. 2020. The case for learned spatial indexes. *arXiv preprint arXiv:2008.10349* (2020).

[30] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2018. QUASII: query-aware spatial incremental index. In *21st International Conference on Extending Database Technology (EDBT)*.

[31] Martin L Puterman. 2014. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

[32] Jianzhong Qi, Guanli Liu, Christian S Jensen, and Lars Kulik. 2020. Effectively learning spatial indices. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2341–2354.

[33] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2018. Theoretically optimal and empirically efficient r-trees with strong parallelizability. *Proceedings of the VLDB Endowment* 11, 5 (2018), 621–634.

[34] Kenneth A Ross, Inga Sitzmann, and Peter J Stuckey. 2001. Cost-based unbalanced R-trees. In *Proceedings Thirteenth International Conference on Scientific and Statistical Database Management. SSDBM 2001*. IEEE, 203–212.

[35] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. 1995. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. 71–79.

[36] Nick Roussopoulos and Daniel Leifker. 1985. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*. 17–31.

[37] Zahra Sadri, Le Gruenwald, and Eleazar Leal. 2020. Online index selection using deep reinforcement learning for a cluster database. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 158–161.

[38] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects*. Technical Report.

[39] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 1153–1170.

[40] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 569–574.

[41] Zheng Wang, Cheng Long, Gao Cong, and Yiding Liu. 2020. Efficient and effective similar subtrajectory search with deep reinforcement learning. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2312–2325.

[42] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *arXiv preprint arXiv:2104.05520* (2021).

[43] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 193–208.

[44] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1297–1308.

[45] Eleni Tzirita Zacharatou, Andreas Kipf, Ibrahim Sabek, Varun Pandey, Harish Doraiswamy, and Volker Markl. 2020. The case for distance-bounded spatial approximations. *arXiv preprint arXiv:2010.12548* (2020).

[46] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 415–432.