RW-Tree: A Learned Workload-aware Framework for R-tree Construction

Haowen Dong[1], Chengliang Chai[1*], Yuyu Luo[1], Jiabin Liu[1], Jianhua Feng[1], Chaoqun Zhan[2]
[1]*Department of Computer Science, Tsinghua University,* [2]*Alibaba*
{dhw21@mails.,ccl@,luoyy18@mails.,liujb19@mails., fengjh@}tsinghua.edu.cn, lizhe.zcq@alibaba-inc.com

***Abstract*—R-tree is a popular index which supports efficient queries on multi-dimensional data. The performance of R-tree mostly depends on how the tree structure is built if new data instances are inserted, which has been studied for years. Existing works can be categorized into two groups. One is the bulk-loading approaches that insert data instances in batch, but they cannot support real-time insertion. Hence, our focus is on the other one that inserts each data instance individually, and thus fresh data can be instantly queried. However, existing methods do not consider the workload information, which leads to limited potential optimization opportunity. Therefore, it is important to study workload-aware R-tree construction for efficient multi-dimensional data access. There are several challenges. First, how to represent the query workload is a challenge. Second, given a workload, it is challenging to accurately measure the benefit of a data insertion choice. Third, both range queries and kNN queries should be considered in the workload.**

To address these challenges, we propose a novel framework that leverages a learning-based method to solve the workload-aware R-tree construction problem. First, by extracting the query workload features, we learn a distribution for the workload using the space partition. Second, considering the distribution, we design a cost model to describe the benefits ($i.e.$, query execution time) of different insertion choices and select the best one. Third, we convert the kNN queries to range search ones, so as to support the workload including both types of queries. Experimental results show that on OpenStreetMap real datasets, compared with baselines, we improve the query efficiency by $1.17\times$.

I. Introduction

R-tree is a popular tree data structure used for spatial data, i.e., indexing multi-dimensional data like geographical coordinates, rectangles or polygons, which are represented by hierarchical minimum bounding rectangles (MBRs). The query efficiency mostly depends on the structure of an R-tree, and most research works [3], [4], [7], [14]–[16], [31] in this field focus on how to construct a well-organized R-tree leveraging the operation of data insertion.

Limitations of existing methods. Generally speaking, there are two lines of works to address this problem. One is the bulk-loading approaches [1], [2], [17], [19], [23], [28] that directly pack all data instances to be inserted into leaf nodes. The limitation is that they cannot support real-time data insertion, and thereby one cannot instantly query the data instances that have just been inserted. Hence, in this paper, we focus on the other line of works [4], [14]–[16], [31] that update the R-tree by inserting each data instance individually, and thus one can access fresh data at any time. However, current works in this line just leverage heuristic methods, $e.g.$, inserting an data instance based on the area enlargement of an MBR without considering the workload characteristics, and thus the potential optimization opportunity is limited. The reason is that the workload in real world always follows a certain distribution. Given such a workload, we should build an R-tree such that queries following the same distribution as the workload can be efficiently executed on the tree. Therefore, in this paper, we aim to optimize the R-tree construction by taking the historical query workload information into consideration.

Challenges. At a high level, there are several challenges for the problem of workload-aware R-tree construction. First, how to capture the features of the query workload and represent it for effective and efficient optimization is the first main challenge (**C1**). Second, given a workload, how to measure the benefit of an insertion choice to the workload is challenging (**C2**). Traditional measurements ($e.g.$, area enlargement) are not appropriate because they cannot well represent the query execution time. Third, spatial queries are not limited to the range search query, and kNN query is also a significant one, and thus how to consider both queries in the workload is another challenge (**C3**).

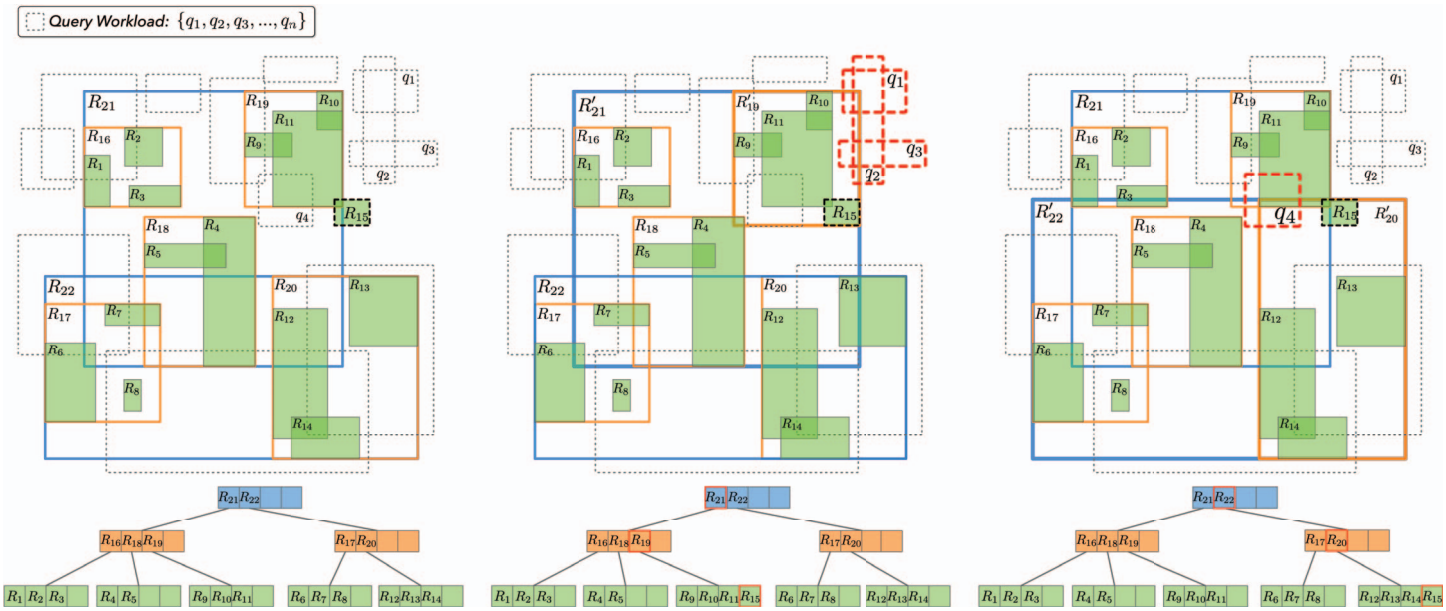
Our proposed method. To address the above challenges, we propose a learned workload-aware framework, `RW-tree` that builds the R-tree that supports efficient spatial data access considering the workload information. To be specific, we first extract several important features from the query workload and learn a distribution to represent the workload, such that the characteristics of the workload are well captured when building the R-tree (for **C1**). Then, given the workload representation, we need to measure how a data insertion choice performs on the workload, so as to discover the best choice. To this end, we propose a cost model as the measurement, which can accurately approximate the real query execution time (for **C2**). Third, when the workload contains both range search queries and kNN queries, we propose to transform kNN queries to the former ones and leverage learned data distribution to answer these queries (for **C3**).

Contributions. To summarize, we make the following contributions in this paper.

(1) We propose a learned workload-aware framework for R-

* Chengliang Chai is the corresponding author.

2375-026X/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDE53745.2022.00201



(a) An example of R-tree with query workload. (b) An example of data insertion using heuristics. (c) An example of workload-aware data insertion

Fig. 1. Examples of data insertion strategies.

tree construction, so as to optimize the query efficiency considering the workload information.

(2) We learn the workload distribution through space partition, and propose a cost model to effectively and efficiently capture the performance of each data insertion choice based on the learned distribution.

(3) We also support to optimize the R-tree built based on the workload mixed with range search and kNN queries, by transforming the kNN queries to range ones.

(4) Experimental results show that our method significantly outperforms existing R-tree construction approaches, improving the efficiency by $1.17\times$ in real-life datasets.

Paper organization. First, we define the typical data insertion problem and introduce the cost model in Section II. Then we introduce our workload-aware framework to address the proposed problem using the cost model in Section III. Next, we introduce model training and inference in Section IV. In Section V, we illustrate how to support kNN queries in the workload. We compare our method with other state-of-the-art methods in Section VI. We review related works in Section VII and conclude in Section VIII.

II. Preliminary

A. Problem Definition

As we know, R-tree is a data structure that groups nearby data instances, represents them as an MBR, and organizes these MBRs as a balanced search tree. Given an R-tree, the searching algorithm is rather simple, which uses the bounding boxes to decide whether to search inside a subtree (i.e., an MBR) or not. Therefore, the tree structure has a large impact on the search performance, which is determined by the data insertion operation. To insert a data instance, the tree is traversed recursively from the root node, where two key steps should be considered iteratively.

(1) [*Choosing the insertion subtree.*] During the recursive process, at each tree level, we have to choose which subtree the node should be inserted into. Then this step repeats until reaching a leaf node.

(2) [*Splitting an overflow node.*] When the data instance is inserted, causing a node to exceed the storage limitation, we should split this node into two parts or re-insert some data instances.

Limitations of typical heuristics. To address the above steps, traditional strategies adopt some heuristics considering the area of MBR. Taking the typical R-tree [16] as an example, for choosing the subtree, if an instance is to be inserted, it will select the node with the minimal area enlargement. For overflow treatment, it splits an overflow node based on the distance between children nodes in the area. However, the above heuristics do not consider the characteristics of query workloads, leading to the performance degradation.

Example 1: For example, Fig. 1 (a) shows an R-tree associated with a historical query workload, denoted by dashed rectangles. The R-tree is shown in colored solid rectangles. We can observe that the queries are dense in the upper part of the spatial space, and sparse in the lower part. Suppose that R_{15} is going to be inserted into the R-tree, and firstly, we should choose from R_{21} and R_{22}. If we adopt the traditional method as shown in Fig. 1 (b), R_{15} will first be inserted into R_{21} because the area enlargement of R_{21} is less than that of R_{22}. However, considering the query distribution, if we insert into R_{21}, given such a workload, q_1, q_2 and q_3 will lead to extra scans because they have overlap with R_{21}. If we insert into R_{22}, only q_4 needs an extra scan, which is more efficient than the choice of inserting into R_{21}.

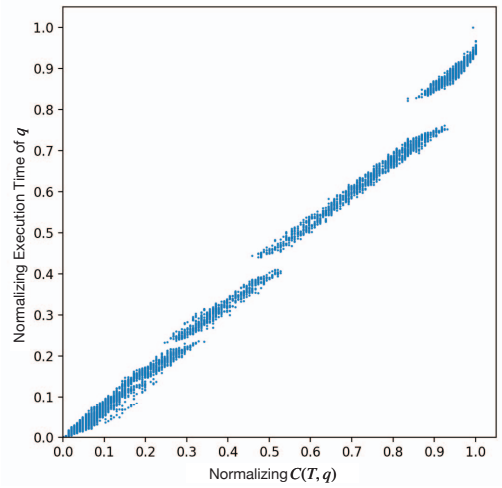


Fig. 2. Relationship between $\mathcal{C}$ and query execution time.

Workload-aware data insertion. Based on the example, we can see that considering the query workload, simply relying on the area enlargement to choose the subtree for insertion may not be a good choice. Ideally, given a query workload, a data instance, and several candidate MBRs to be inserted, if we can accurately obtain the performance change of the data instance being inserted into each MBR for the workload, we naturally have the ability to select the "*best*" MBR that leads to the most performance improvement to insert. However, the performance change is rather hard to derive because (1) the entire distribution of the query workload (including the future) is hard to predict, and (2) the performance is unavailable unless the workload is executed. For the former one, we make a reasonable assumption that the workload remains stable over a long period of time, and thus it is feasible to build the R-tree based on a historical workload. For the latter one, we propose to use a learning-based method to estimate the performance change for data insertion.

Remark. The above example mainly discusses the limitation of heuristic method w.r.t. the choosing subtree step. For overflow treatment, the heuristic method based on the distance is also hard to find the optimal solution for the query workload (we omit the example due to the space limitation). But fortunately, both steps can be solved by the learning-based method that will be introduced later.

***Cost*: # of scanned nodes.** For the widely-used range search query (dashed rectangles in Fig. 3), the search algorithm is to recursively scan the nodes whose MBRs overlap with the query MBP. Note that R-tree is balanced, so the scanning time on each node is similar, and thereby the number of scanned nodes is proportional to the query execution time. Since the actual execution time of a query in workload is hard to obtain, we propose to use the total number of scanned nodes (we also name the scanned number as *cost*, denoted by $\mathcal{C}$) during the entire workload as a measurement of the performance. Thus, optimizing the cost is basically to optimize the execution time, as shown in Fig. 2. $\mathcal{C}(T, q)$ is the actual number of nodes that scanned during the execution process of query q on R-tree T.

Example 2: As shown in Fig. 3, we consider a workload

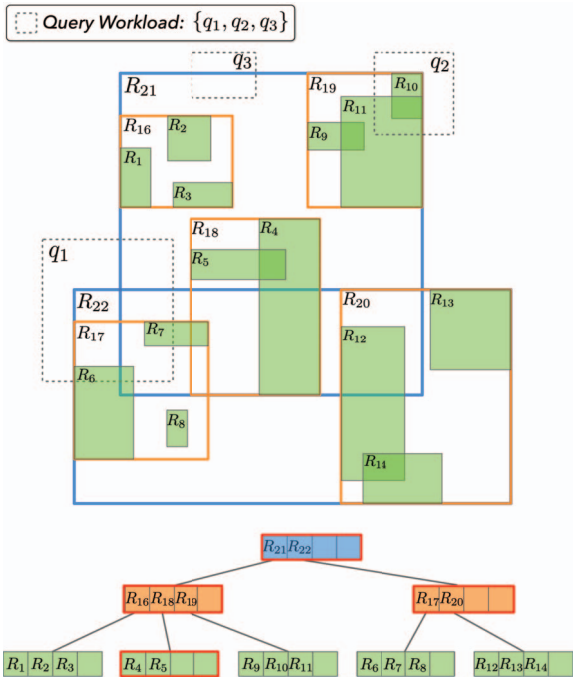
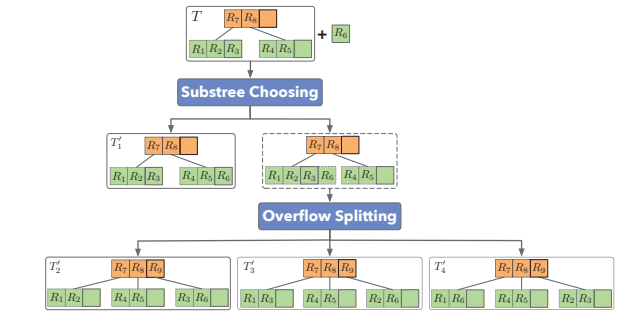
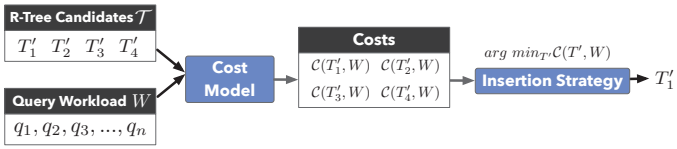


Fig. 3. An example of the cost metric

W $\{q_1, q_2, q_3\}$ and a R-tree T rooted at the node R_T (R_T is rectangle including all nodes and not shown in the Figure). The node of R-tree is $\{R_T, R_{16}, R_{17}, \cdots R_{22}\}$ and the data is $\{R_1, R_2, \cdots, R_{15}\}$. For the search of q_1, $R_T, R_{17}, R_{21}, R_{22}$ overlap with the q_1 and each of them will be scanned once, so the cost of $\{q_1\}$ is 4, denoted by $\mathcal{C}(q_1) = 4$. For the whole workload, R_T and R_{21} will be scanned in q_1, q_2, q_3, R_{22} and R_{17} will be scanned in q_1 and R_{19} will be scanned in q_2, while other internal nodes will not be scanned. Hence, the total cost of the entire workload executing on the tree is $3+3+1+1+1 = 9$, denoted by $\mathcal{C}(T, W) = 9$.



(a) An example of candidate selection



(b) An example of the insertion strategy

Fig. 4. An example of insertion optimization problem

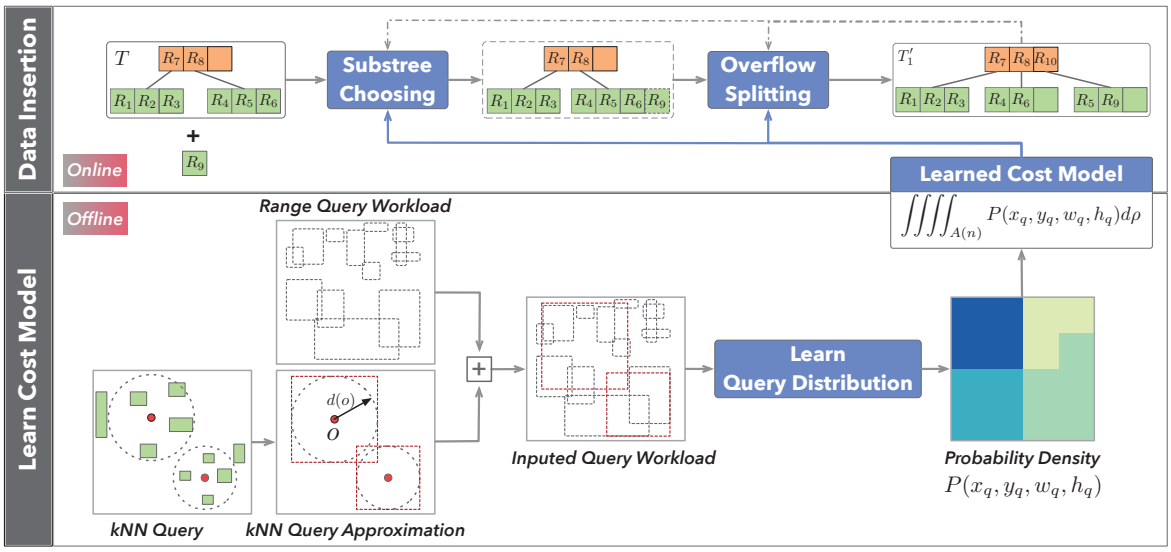


Fig. 5. Overall Framework of RW-tree

Insertion optimization using the *cost*. Recap that given an R-tree T as well as a query workload W, $\mathcal{C}(T, W)$ measures how T performs over the workload W. Then, given a data instance R to be inserted into T, different choices of the above two steps (i.e., subtree choosing and node splitting) will lead to different candidate tree structures, denoted by the set $\mathcal{T} = \{T'\}$, where each T' denotes another R-tree in which d has been inserted. Hence, $\mathcal{C}(T', W)$ denotes the cost of T' over the workload W.Then, we can formally propose the insertion optimization problem as follows.

Definition 1: Given W, T and a data instance R to be inserted, a number of R-tree candidates can be generated, denoted by $\mathcal{T} = \{T'\}$, where each element corresponds to an insertion strategy (including choose which subtree and how to the split overflow node). The insertion optimization problem is to choose the strategy that leads to the least cost, i.e., $\arg\min_{T'} \mathcal{C}(T', W)$.

Example 3: As Fig. 4(a) shows, considering an R-tree T (minimal capacity of a node is $m = 2$ and the maximal capacity is $M = 3$) and a data instance R_6, the task is to insert R_6 into T. To this end, we should first choose the subtree to insert from R_7 and R_8. If R_7 is chosen, we have T'_1. If R_8 is chosen, the node will have 4 children and need to split. There are $C_4^2/2 = 3$ splitting choices, corresponding to T'_2, T'_3 and T'_4. Hence, $\mathcal{T} = \{T'_1, T'_2, T'_3, T'_4\}$. Then given a query workload W, the optimization problem is to compute the $\arg\min_{T'} \mathcal{C}(T', W)$ using a cost model as shown in Fig. 4(b).

At the following, we will show the difficulty of computing the cost, and then overview our proposed framework, where a learning-based model for cost computation is proposed.

III. Framework

In this section, we first introduce the challenges with respect to the above problem, and overview the RW-tree framework to solve the problem (Section III-A). Then we briefly introduce two key components (learned cost model and kNN approximation) in our framework in Section III-B and III-C respectively.

A. Overview of RW-tree

Challenges. First, based on the problem definition, in order to choose the most cost-effective strategy, we need to compute $\mathcal{C}(T', W)$ for each R-tree candidate. However, given W and T', it is non-trivial to compute the cost, *i.e.*, the number of scanned nodes during the workload execution. The reason is that computing the exact number is equivalent to really executing the query, which is rather time-consuming. Hence, the first challenge is how to estimate the cost efficiently and accurately.

Second, to obtain the optimal strategy, we also need to explore the search space of R-tree candidates, *i.e.*, $|\mathcal{T}|$. However, $|\mathcal{T}|$ is proportional to the number of leaf nodes of the R-tree when only choosing subtree is needed. When there has to be the splitting operation, the search space will be much larger (an example is given in Fig. 4(a)). Therefore, how to efficiently explore the search space is another challenge.

Third, the above examples only discuss the workload of range search queries. Besides, the k Nearest Neighbor (kNN) query is also a typical query type in spacial data. Hence, when the workload contains kNN queries, how to adapt our framework to build an R-tree for such workload, such that queries in the workload can be efficiently executed.

RW-tree Framework. The overall framework of RW-tree is shown in Fig. 5. First, given the input T, W and R, instead of enumerating all R-tree candidates in $\mathcal{T}$, we adopt the typical best-first strategy to explore the search space of R-tree. To be specific, we maintain a priority queue while traversing T from top to bottom, and first explore the subtree that is likely to lead to a tree structure with a low cost, so as to reduce the search space.

During the best-first search, given the query workload W, we build a learned cost model to estimate the cost of an R-tree candidate, *i.e.*, $\mathcal{C}(T', W)$. To be specific, for training, the model learns the distribution of queries in W. Then, for

inferring the cost of T', as an R-tree consists of a number of nodes, we take as input a node as well as W, and predicts the number of scanned times of this node when executing queries in W using integration over the learned query distribution. Then $\mathcal{C}(T', W)$ can be computed by summing the number of scans of all nodes in T'. We will overview this step in Section III-B.

For kNN queries, we propose to transform them to range queries, and use the above trained model to construct the R-tree (see Section III-C for overview).

B. Learned Cost Model

In this part, we will overview how to train a cost model to estimate the cost of executing a query workload W over an R-tree T', and then introduce how to use it.

Model Training. At a high level, to achieve the workload-aware spatial data insertion, we have to capture the query distribution, which is significant for estimating the cost. Hence, in this part, we take as input the workload W and learn its distribution. Since each query q in the workload corresponds to an MBR, denoted by $B(q)$, which can be represented by the coordinate of the center (x_q, y_q), the width (w_q) and height (h_q) of the MBR of search boundary. Therefore, the query distribution can be represented in the form of 4-dimension probability density $P(x_q, y_q, w_q, h_q)$. We will show the details of learning the distribution in Section IV.

Model Inference. Recap that $\mathcal{C}(T', W)$ is computed by summing the number of scans of all nodes in T'. In this part, we take as input the query distribution and nodes in R-tree to predict the number of scans $i.e.$, $\mathcal{C}(n, W)$, of each node n. Hence, $\mathcal{C}(T', W)$ can be calculated. Specifically, $\mathcal{C}(n, W)$ equals the number of queries whose search boundary intersects with the MBR of n. Intuitively, it can be computed by density integral over the *intersection area*, which is defined as follows. Before that, we first introduce the intersection condition of two MBRs. Consider the MBR of node $B(n)$, denoted by its center (x_r, y_r), width (w_r) and height (h_r). If the following condition is satisfied, then $B(n)$ and $B(q)$ will be intersected.

$$B(n) \text{ intersects } B(q), \text{if} \begin{cases} |x_r - x_q| \leq w_r + w_q \\ |y_r - y_q| \leq h_r + h_q \end{cases}$$

Given the intersection condition, we can easily obtain the *intersection area* (denoted by $A(n)$) of a node n, where queries in this area will scan the node. $A(n)$:

$$A(n) = \begin{cases} x_r - w_r - w_q \leq x_q \leq x_r + w_r + w_q \\ y_r - h_r - h_q \leq y_q \leq y_r + h_r + h_q \\ w_q \geq 0, h_q \geq 0 \end{cases}$$

Therefore, given the distribution and integral area , the number of scans of a node in a given workload W can be obtained with integration as follows:

$$\mathcal{C}(n, W) = \iiiint_{A(n)} P(x_q, y_q, w_q, h_q) d\rho$$

Apparently, directly computing the 4-dimension integration is time-consuming, and thus we will illustrate how to conduct efficient integration computation in Section IV-A.

C. Learned kNN Approximation

In this part, we will showcase the key idea of how to handle the workload including kNN queries using the above framework. For any given query point $o = (x_o, y_o)$, we use $k\text{NN}(o)$ to denote the set of k nearest neighbor to o. Suppose that the k-th nearest data instance of o is o', and the distance between o and o' is denoted by $d(o)$. Since kNN query is implemented with the best first search algorithm, the scan nodes of $k\text{NN}(o)$ are the same to range query with a circular area with o as the center and $d(o)$ as the radius [29]. Then, we can transform the kNN query to range search query, and use the aforementioned framework to solve the problem. Hence, the core problem is how to compute $d(o)$.

$d(o)$ Prediction. To predict the $d(o)$, firstly, we need to learn the distribution of these data instances, denoted by $Q(x, y)$. Then, we use $E(o, d(o)) = \iint_{c(o,d(o))} Q(x, y) d\delta$ to denote the excepted number of data instances inside the search boundary, $i.e.$, $c(o, d(o))$, the circle with o as the center and $d(o)$ as the radius. Hence, solving the function $E(o, d(o)) = k$, we can derive the corresponding $d(o)$ (see Section V for details).

IV. Learned Cost Model

In this section, we will introduce the *learned cost model* proposed by us for predicting the cost $\mathcal{C}$ for any given R-tree T and workload W. In this section, we first introduce given the model ($i.e.$, the query distribution P), how to use it to infer the estimated cost ($i.e.$, the integration computation). Then we introduce the training process to learn query distribution. Finally, given the model training and inference methods, we illustrate how to conduct the data insertion algorithm using the cost model.

A. Cost Model Inference

As discussed in Section III-B, to estimate the cost of an R-tree T with a given workload W, the model will first break down to estimate the cost (number of scans) of each node, $i.e.$, $\mathcal{C}(n, W)$ in an R-tree, and then sum them up. To be specific, $\mathcal{C}(n, W)$ can be obtained with a 4-dimensional integration when the query distribution $P(x_q, y_q, w_q, h_q)$ is learned. However, directly computing the 4-dimensional integration is apparently time-consuming. Therefore, in this section, we will illustrate how to conduct efficient integration computation as follows.

In real workload, the query distribution over the entire spatial space is likely to be complicated, but a reasonable assumption is that the distribution over a small local area can be approximated as a *uniform distribution*. Based on that, the integration computation can be much accelerated using these uniform distributions.

More specifically, this basic idea consists of three steps: (1) Divide the entire space into several partitions, each of which can be approximately described by the uniform distribution.

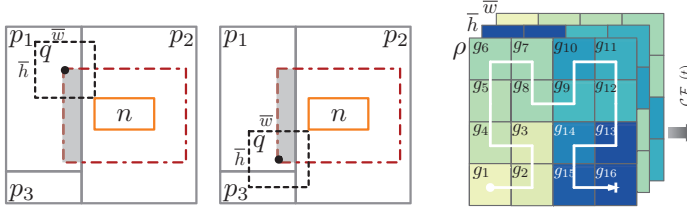


Fig. 6. Inference with space partitions

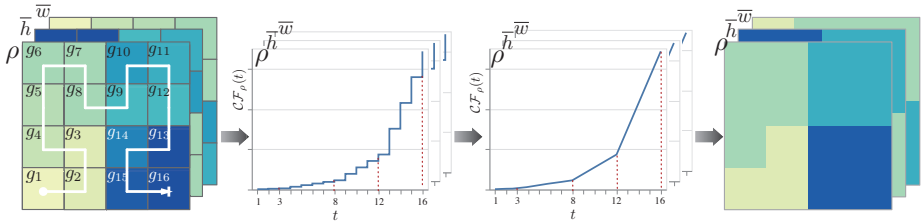


Fig. 7. An example of learning to cluster the grids

(2) Compute the integration over each uniform distribution of each partition. (3) Compute the $\mathcal{C}(n, W)$ by summing the result of each partition up. The first step is done in the phase of *cost model training*, which will be discussed in Section IV-B. Here, we introduce the uniform distribution and how to compute the cost based on it.

Uniform Distribution. We first assume an extreme case that the entire space can be regarded as a uniform distribution, where $P(x_q, y_q, w_q, h_q)$ follows a 4-dimension uniform distribution, and (w_q, h_q) is always independent to (x_q, y_q) in practice. In this case, the density of query in the entire data space is a constant, denoted as ρ. $\mathcal{C}(n, W)$ is integration of density over the area of intersection area $A(n)$. Therefore, $\mathcal{C}(n, W)$ can be simplified as follows:

$$\mathcal{C}(n, W) = \rho \iint_{A(n)} P(w_q, h_q) d\delta$$

Note that $P(w_q, h_q)$ is the probability density of a 2 dimension uniform distribution, which can be expressed as $U(w_l, w_u, h_l, h_u)$ ($[w_l, w_u]$ is the interval of w and $[h_l, h_u]$ is the interval of h). The expected width is denoted by $\overline{w}$ and the excepted height is denoted by $\overline{h}$. By expanding the integral area $A(n)$, $\mathcal{C}(n, W)$ can be expressed as follows:

$$\begin{aligned}
&\mathcal{C}(n, W) \\
=&\rho \int_{w_l}^{w_u} \int_{h_l}^{h_u} \frac{1}{w_u - w_l} \frac{1}{h_u - h_l} (w_r + w)(h_r + h) dh dw \\
=&\rho (w_r + \frac{1}{2}(w_l + w_u))(h_r + \frac{1}{2}(h_l + h_u)) \\
=&\rho (w_r + \overline{w})(h_r + \overline{h})
\end{aligned}$$

In this way, $\mathcal{C}(n, W)$ can be computed by the simple yet efficient mathematical operations over the density of query ρ, the average width $\overline{w}$ and the average height $\overline{h}$, which is much more efficient than directly computing the multi-dimensional integration.

Inference with space partitions. However, in real case, it is not reasonable to approximate the whole space into as a uniform distribution. Instead, we divide the space into some partitions (see Section IV-B in detail), each of which can be approximated as a uniform distribution.

To be specific, suppose that the space is is divided into N rectangle partitions, denoted by $\{p_1, p_2, \cdots, p_N\}$, queries in each of which are similar to each other. Note that not all partitions should be considered for computing $\mathcal{C}(n, W)$ because they are far away from the node n, and thus the queries in these partitions have no chance to influence the cost of n. Next, we will introduce the partitions that should be considered, in which queries may intersect with the node n when they are executed. More concretely, we observe that given the node n, whether a partition will influence $\mathcal{C}(n, W)$ can be determined with the boundary of partition, the height and weight of the queries inside the partition. For ease of computation, we use the average height and width to represent all the queries in the same partition. Specifically, we denote the partition p as $(x_p, y_p, w_p, h_p, \overline{w}, \overline{h})$, where (x_p, y_p, w_p, h_p) is the boundary of p. Therefore, the influence condition can be formally described as below:

$$p \text{ influences } \mathcal{C}(n, W), \text{if } \begin{cases} |x_r - x_p| \leq w_r + w_p + \overline{w} \\ |y_r - y_p| \leq h_r + h_p + \overline{h} \end{cases}$$

Then, we aim to compute which queries of each such partition will intersect with n, and $\mathcal{C}(n, W)$ can be computed through integration over these intersection areas. To this end, we observe that by extending the MBR of n, we can easily discover these intersection areas. We denote $S(n, p_i)$ as the intersection area of p_i and the extended MBR of node n, and ρ_i as the query density in p_i. Then $\mathcal{C}(n, W)$ can be computed by the following equation:

$$\mathcal{C}(n, W) = \sum_{i=1}^{N} \rho_i S(n, p_i)$$

Example 4: As Fig. 6 shows, given a workload W, we assume the space is divided into 3 partitions $\{p_1, p_2, p_3\}$ after the model training process. Now given a node n, we need to compute the number of scans of node over the workload, *i.e.*, $\mathcal{C}(n, W)$ based on the space partitions. Since all partitions may influence $\mathcal{C}(n, W)$, we take p_1 as an example. The three features of p_1 are denoted as $\{\rho_1, \overline{w}_1, \overline{h}_1\}$. For ease of representation, we assume that all the width and height of queries in p_1 are $(\overline{w}_1, \overline{h}_1)$. To compute $\mathcal{C}(n, W)$, we need to determine which queries in p_1 will scan n, and compute the intersection area of the space of influence condition (the red dotted line) and the space of p_1, shown as the grey area. Therefore, the influence of p_1 to the cost is $\rho_1 S(n, p_1)$, where $S(n, p_1)$ is the area of the grey part. The influence of other two

partitions p_2, p_3 is same as p_1. By summing up the influence of all these three partitions, $\mathcal{C}(n, W)$ is computed.

In this way, the time complexity to compute $\mathcal{C}(n, W)$ is $O(N)$. The reason is that we iterate all partitions to check if their boundaries intersect with the extended MBR of n.

Acceleration with R-tree. When the number of space partitions is large, the above method is not efficient even with the linear time complexity. To address this, we can organize the space partitions into an in-memory R-tree. Specifically, to check whether a space partition p influences $\mathcal{C}(n, W)$, we can extend the boundary of p to $(x_p, y_p, w_p + \overline{w}, h_p + \overline{h})$ and detect whether the MBR of n intersects with this extended boundary. Then we can build an R-tree with all the extended boundary of all space partitions. Afterwards, to search the partitions that will influence $\mathcal{C}(n, W)$, we can issue a range query, $i.e.$, the MBR of n, and obtain these partitions. Then we reduce the expected average cost estimation time complexity from $O(N)$ to $O(\log N)$.

B. Cost Model Training

Recap that in Section III, the training step takes as input the workload W and learns the query distribution from W. In order to predict the cost in an efficient way, we need to divide the space into some space partitions, in which the distribution of queries approximately follows a uniform distribution. To this end, during the model training process, `RW-tree` aims to learn the distribution of queries from given workload W, based on which we conduct the aforementioned space partition.

At a high level, this phase consists of 3 steps. First, we preprocess the workload by dividing the entire space into grids, mapping the queries into their corresponding grids and then compute the statistics of these queries. Second, we cluster these grids along with the space-filling curve, where queries in the same cluster have similar statistics. Third, we compute the space partitions based on these clusters. Next, we illustrate the above three steps respectively.

Workload Preprocessing. Since our task is to divide the space into several partitions, each of which follows an uniform distribution (Section IV-A), we propose to first divide the entire space into a number of small equal-size grids and then cluster some of them based on the query statistics.

Recap that in Section III-B, the range query is represented by a quadruple (x_q, y_q, w_q, h_q), so we consider three types of statistics for each grid g_i. Specifically, (1) the density of queries whose centers are inside g_i, denoted by ρ_i, which equals to the number of queries inside g_i divided by the area of g_i; (2) the average width $\overline{w_i}$ of the above queries; and (3) the average height $\overline{h_i}$ of the above queries. Next, we aim to cluster these grids based on the above features.

Learning to cluster the grids. Since we aim to use a uniform distribution to describe each partition, the grids in the partition should both be spatially continuous and have similar features. First, to achieve the spatial continuity, we sort all the grids by the position of each grid along with a space-filling curve, producing a sequence $G = [g_1, g_2, ..., g_{|G|}]$. As discussed above, each g_i contains three features, denoted by $f_i = \{\rho_i, \overline{w_i}, \overline{h_i}\}$. In this way, given an arbitrary continuous interval on the curve, the grids corresponding to the interval constitute a continuous space area [18]. Therefore, if we divide the sequence of sorted grids into several fragments, each of which corresponds to a spatially continuous partition. Next, we will study how to divide the sequence to ensure that each fragment has grids with similar features inside.

Since we have three dimension features to be clustered, in this part, we first study how to cluster one of them, and in the next `space partition` part, we will show how to combine these three types of clusters.

Taking the feature of ρ as an example, following the sequence of grids, we have a sequence $G_\rho = [\rho_1, \rho_2, ...\rho_{|G|}]$, based on which we first define a cumulative function $\mathcal{CF}_\rho(t) = \sum_{i=1}^{t} \rho_i$, as shown in Fig. 7. Second, we compute a piecewise linear function $\mathcal{CF}_\rho^*$ to approximate the cumulative function, which can well represent the distribution of ρ. Recap that we aim to require each space partition that can be described with a uniform distribution, $i.e.$, ρ of grids in each partition should be similar. Obviously, the grids of each piece of $\mathcal{CF}_\rho^*$ have the same gradient, so we cluster these grids into the same partition and these clusters constitute a collection of grids, denoted by $\mathcal{C}_\rho$.

Example 5: As Fig. 7 shows, when we cluster the grids by the density feature, we get a sequence $G_\rho = [\rho_1, \rho_2, ...\rho_{16}]$ along with the space-filling curve shown in Fig. 7(a). We can compute $\mathcal{CF}_\rho(t), t \in \{1, 2, \cdots, 16\}$ based on the definition of cumulative function $\mathcal{CF}_\rho(t) = \sum_{i=1}^{t} \rho_i$ ($e.g.$, $\mathcal{CF}_\rho(3) = \rho_1 + \rho_2 + \rho_3$). Based on that, we can learn a peicewise linear function $\mathcal{CF}_\rho^*$, as shown in Fig. 7(c). Since when the grids have similar ρ like ρ_1, ρ_2, ρ_3, the gradients in the corresponding interval of $\mathcal{CF}_\rho^*$ are the same. So according to our algorithm, we will cluster these grids into the same partition. In this example, $\mathcal{CF}_\rho^*$ is divided into four pieces, and thus the collection of clusters based on density ρ is $\mathcal{C}_\rho = \{\{g_1, g_2, g_3\}, \{g_4, g_5, g_6, g_7, g_8\}, \{g_9, g_{10}, g_{11}, g_{12}\}, \{g_{13}, g_{14}, g_{15}, g_{16}\}\}$, as shown in Fig. 7(d).

Similarly, we will conduct the same steps for features $\overline{w}$ and $\overline{h}$. Next, we introduce how to combine them to produce the final partition.

Adjust collections to space partition. In the above paragraph, given three sequences G_ρ, $G_{\overline{w}}$ and $G_{\overline{h}}$, we compute three collections denoted by $\mathcal{C}_\rho$, $\mathcal{C}_{\overline{w}}$ and $\mathcal{C}_{\overline{h}}$. Next, we show how to adjust these collections to produce the final space partitions such that in each partition, the above three features of grids are similar. At a high level, the adjustment consists of 2 steps. (1) [*Collections combination.*] We first need to combine the three collections into a single one, denoted by $\mathcal{C}_G$, and each cluster in $\mathcal{C}_G$ have similar features.

To be specific, for each collection, we give each cluster a unique ID ($e.g.$, $\{c_\rho^1, c_\rho^2, ...\}$) and then assign the each ID to the corresponding grids in each cluster. For example, as shown in Fig. 8(a), the collection of clusters based on density ρ is $\mathcal{C}_\rho = \{\{g_1, g_2, g_3\}, \{g_4, g_5, g_6, g_7, g_8\}, \{g_9, g_{10}, g_{11}, g_{12}\},$

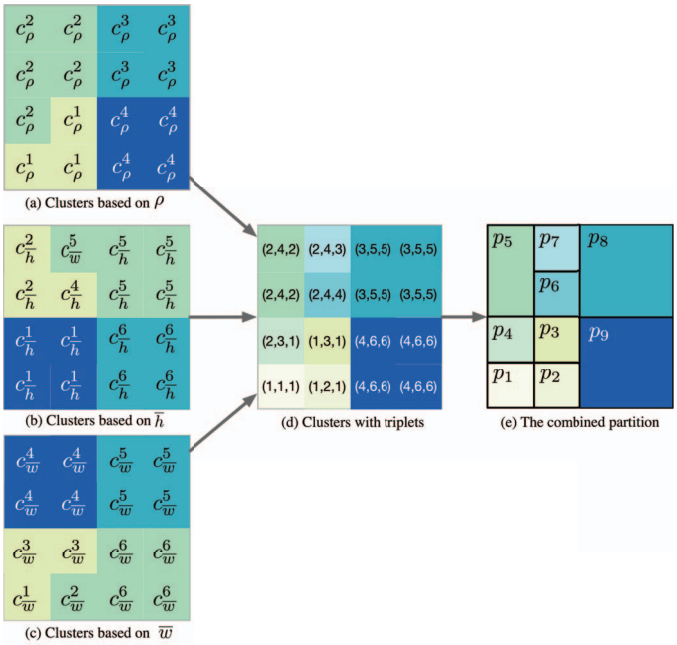


Fig. 8. Adjust collections to space partitions.

$\{g_{13}, g_{14}, g_{15}, g_{16}\}\}$. Then grids in the first cluster $\{g_1, g_2, g_3\}$ will be assigned to c_ρ^1. Then each grid will be assigned with three IDs, denoted by a triplet $[c_\rho^i, c_{\overline{w}}^j, c_{\overline{h}}^k]$. If two grids have the same triplet, they have same query features, and thereby they will be in the same partition.

Example 6: As Fig. 8 shows, we need to merge the three given collections $\mathcal{C}_\rho, \mathcal{C}_{\overline{w}}, \mathcal{C}_{\overline{h}}$ into a combined one. Note that two grids in the same cluster of a collection have the same feature and will be assigned to the same ID. Taking two grids g_1, g_2 as an example, they are both in c_ρ^1 cluster, so we can infer that $\rho_1 = \rho_2$. As Fig. 8 (a) (b) (c) shows, each grid will be assigned with 3 IDs, we merge the 3 IDs into a triplet. As a example, the three feature IDs of g_2 are $c_\rho^1, c_{\overline{w}}^2, c_{\overline{h}}^1$, so the triplet of g_2 is $(1, 2, 1)$. Along with the space-filling curve, we can divide the space based on the triplet of grids(shown in (d)) to the combined space partition $\mathcal{C}_G$(shown in (e)).

(2) [*Adjust $\mathcal{C}_G$ to rectangular partitions.*] Recap that in the inference step, each partition should be a rectangle for improving the efficiency. Hence, we should adjust $\mathcal{C}_G$ to rectangular partitions. To this end, for each partition in $\mathcal{C}_G$, if it is not a rectangle, we use a greedy strategy to split it to several rectangles. Specifically, the greedy method splits the space partition composed of arbitrary adjacent grids into several partitions with rectangle boundary. First, we set all the grids as "non-clustered"; Then, we repeat the process of finding the maximum rectangle area from the "non-cluster" grids and dividing them into a new partition. Since each time we will at least transform a "non-clustered" to a "clustered" node. So after a limited number of iteration, all the grids in original partition will be clustered into several partitions with a rectangle boundary.

V. KNN QUERY WORKLOAD IN RW-TREE

As discussed in Section III-C, in order to support kNN queries in the query workload, we transform them to range search queries, and use the aforementioned framework to optimize the R-tree insertion algorithm with given workload W. The transformation process requires the k-th nearest distance $d(o)$. Note that $d(o)$ is predicted by solving the function $E(o, d(o)) = \iint_{c(o,d(o))} Q(x, y)d\delta = k$, but solving the function with multi-dimensional integration is a time-consuming task. To this end, we use the basic idea of partition in Section IV-A: accelerating the integration computation by dividing the space into several partitions based on the data distribution in R-tree. At a high level, the prediction process consists of two steps: (1) Partition the space based on the data distribution learned from R-tree; (2) Solve the function of $d(o)$ based on the data partition.

A. Partition the space with learned data distribution.

This step takes as input a set $\mathcal{R}$ of data instances in the R-tree and learns the data distribution from $\mathcal{R}$, and then divides the space into some space partitions, in which the distribution of data instances approximately follows a uniform distribution. At a high level, this process is similar to the training process in Section IV-B, which considers three features to partition. Here, we just use one feature ($i.e.$, the density) to conduct this because only this feature will mostly influence the computation of $d(o)$. The reason is as follows. Typically, all the data instances in R-tree can be represented by their MBRs, denoted as (x_r, y_r, w_r, h_r). Note that in the real applications, $d(o)$ is usually much larger than the width and height of data in an R-tree. So we use the distance between o and the center of an instance to approximate the real distance between o and the boundary of the instance, so as to simplify the computation of $d(o)$. Based on that, all the data instances in R-tree can be represented by the coordinates of their centers, denoted as (x_r, y_r). Then, by reusing the training process of the above learned cost model, we obtain a space partition $\mathcal{C}_D$. In $\mathcal{C}_D$, each partition have similar data density.

B. Computing $d(o)$ using the partitions

Given $\mathcal{C}_D$, the query point $o = (x_r, y_r)$ and k, we aim to compute $d(o)$ by solving the equation $E(o, d(o)) = \iint_{c(o,d(o))} Q(x, y)d\delta = k$. To this end, we first consider a general case that $c(o, d(o))$ is included by a single partition p_i because the area of our partition is always much larger than $c(o, d(o))$. In this case, we define the density of p_i as ρ_i. Then we can transform the above equation as follows:

$$E(o, d(o)) = \rho_i \pi d(o)^2$$

where $\pi d(o)^2$ is the area of $c(o, d(o))$. From the above equation, we can compute that $d(o) = \sqrt{\frac{k}{\rho\pi}}$.

For more complicated cases that the circle covers different partitions, we can also compute $d(o)$ by considering the area covered by these partitions associated with different density.

VI. EXPERIMENT

The key questions that we aim to answer in the experiment are: (**Exp-1**) How well does `RW-tree` perform in the range query by varying some parameters? (**Exp-2**) Whether `RW-tree` can outperform the state-of-the-art `R-tree` and its variants? (**Exp-3**) What is the effect of our kNN approximation in the end-to-end evaluation? , and (**Exp-4**) What is the performance of our learned cost model?

A. Experimental Setup

Datasets. We used two synthetic datasets and two real-world datasets to evaluate `RW-tree`.

- **Synthetic Uniform (Syn-Uni)**. This dataset consists of $10k$ rectangles ($i.e.$, data instances) with a fixed size. The centers of rectangles in workload are synthetically generated following the uniform distribution. The center of each rectangle is denoted as (x_r, y_r), both of them follow the uniform distribution and are independent of each other. Some parameters can be configured to tune the statistic features of this workload. In the experiments, the width and height of each data instance are both fixed to 1. Both x_r and y_r follow the uniform distribution $U(0, 1000)$, and thus x_r and y_r are in the range $[0, 1000]$.
- **Synthetic Skew (Syn-Skew)**. This dataset consists of $100,000$ rectangles with various sizes. The centers of the rectangles in workload are synthetically generated, following a 2-dimensional normal distribution. The centers of data instances are first generated by a standard 2-dimensional distribution and then resized by affine transformation. Some parameters can be configured to tune the statistic features of this workload. In the experiments, the width and height of each data instance are both generated following the uniform distribution $U(0, 2)$. Therefore, the average width and height of rectangles are both 1. We let (x_r, y_r) follow the normal distribution and then resize it by affine transformation. Both the range of x_r and y_r are $[0, 1000]$.
- **OSM-China**. This dataset is a real-world dataset that contains the MBRs of buildings in China extracted from OpenStreetMap (OSM). The buildings in this dataset are expressed by the four boundary values, which are expressed by latitude and longitude. We can produce the center (x_r, y_r), width and height ($i.e.$, (w_r, h_r)) of each data instance based on the four boundary values. The average width ($\overline{w}_r$) and height ($\overline{h}_r$) of data instances are 4.39×10^{-5} and 8.51×10^{-5}, respectively. The x_r and y_r are in range $[73.53, 134.78]$ and $[15.78, 53.50]$, respectively. The number of data instances in this workload is more than $1.744M$.
- **OSM Central-America (OSM-CA)**. Similar to **OSM-China**, we obtained this real-world dataset from OpenStreetMap (OSM). This dataset has $2.00M$ MBRs of buildings in Central America. Similarly, we produce the center point ($i.e.$, (x_r, y_r)) and width and height ($i.e.$, (w_r, h_r)) of each data instance based on the four boundary values. The average width ($\overline{w}_r$) and height ($\overline{h}_r$) of data instances are 1.61×10^{-5} and 4.00×10^{-5}, respectively. The x_r and y_r are in range $[-92.24, -59.42]$ and $[5.54, 27.26]$, respectively.

https://www.openstreetmap.org
https://www.openstreetmap.org

TABLE I
STATISTICS OF DATASETS

Datasets	Syn-Uni	Syn-Skew	OSM-China	OSM-CA
#-Rectangles	10k	10k	1.744M	2.00M
$\overline{w_r}$	1	1	4.39×10^{-5}	1.61×10^{-5}
$\overline{h_r}$	1	1	8.51×10^{-5}	4.00×10^{-5}
$[\min x_r, \max x_r]$	[0, 1000]	[0, 1000]	[73.53, 134.78]	[-92.24, -59.42]
$[\min y_r, \max y_r]$	[0, 1000]	[0, 1000]	[15.78, 53.50]	[5.54, 27.26]

In summary, the statistics of experimental datasets are shown in Table I.

Workloads. Since `RW-tree` is designed to optimize the data insertion operation based on historical workloads, some characteristics of the workload may impact the performance of `RW-tree`. In addition to some regular statistics ($e.g.$, x_r, y_r, $\overline{w}_r$, and $\overline{h}_r$) of the workload, we also took the *skew* and *selectivity* of the workload into consideration:

- *skew*: the larger value of the ratio between the width and height of queries, $i.e.$, $\max(\frac{w_q}{h_q}, \frac{h_q}{w_q})$.
- *selectivity*: the ratio of the area of the query to the whole data space.

We obtained one query workload for each dataset, and thus we utilized four workloads to evaluate the performance.

For two synthetic datasets, we generated workload by vaying the *skew* and *selectivity*.

For two real-world datasets, we extracted a series of map queries corresponding to each OSM dataset. For evaluating the performance of `RW-tree` in different *skew*, we clustered all queries into three groups by their $skew$: $[1, 4)$, $[4, 8)$, $[8, \infty)$ and randomly sampled the queries in each group to generate the workload in the experiments.

We randomly sampled $100k$ for each workload to perform the experiments, for four datasets. Since the queries are sampled randomly, the average execution time will not change when we increase the number of queries in the workload proportionally.

Methods. Since `RW-tree` optimizes the workloads with frequently dynamic data insertion, we compared `RW-tree` with traditional R-tree and its variants that are suitable for such cases. There are a set of R-tree [16] and well-known variants, $i.e.$, R-tree with Greene's [14] and R*-tree [4]. As we all know, R*-tree [4] significantly outperforms original R-tree [16] and R-tree with Greene's [14]. Therefore, we utilized the R*-tree [4] as the strong baseline. We do not compare `RW-tree` with bulk-loading R-tree variants [1], [2], [17], [19], [23], [28] because we focus on the dynamic data insertion.

- **`RW-tree`**: a learned workload-aware R-tree variant. `RW-tree` learns the distribution of queries from historical workloads and optimizes the data organization in R-tree based on the learned cost model.

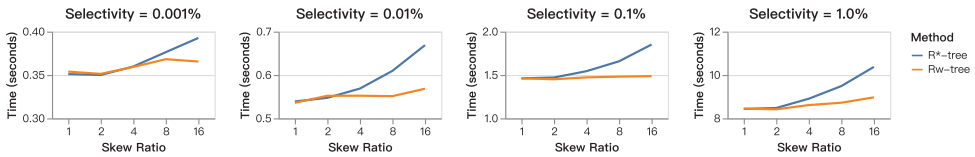


Fig. 9. Exp-1: Execution Time (seconds) *v.s.* Skew (Syn-Uni Dataset)

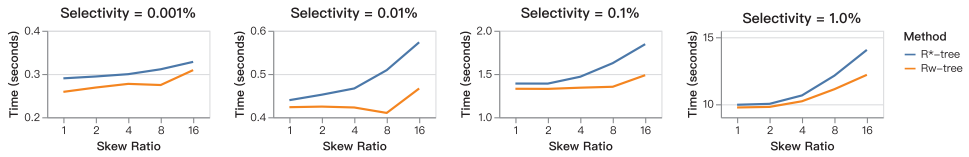


Fig. 10. Exp-1: Execution Time (ms) *v.s.* Skew (Syn-Skew Dataset)

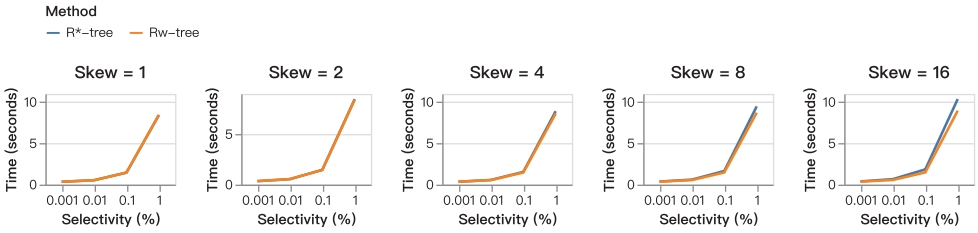


Fig. 11. Exp-1: Execution Time (seconds) *v.s.* Selectivity (Syn-Uni Dataset)

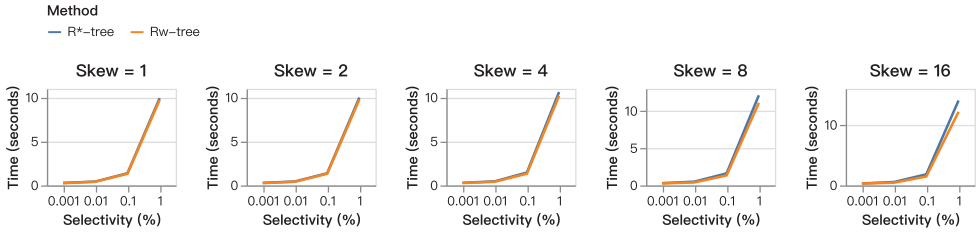


Fig. 12. Exp-1: Execution Time (seconds) *v.s.* Selectivity (Syn-Skew Dataset)

- **R*-tree** [4]: a classic R-tree variant which incorporates a combined optimization of area, margin, and overlap of enclosing rectangles of nodes. In real-world dynamic workload, it can usually achieve the state-of-art performance.

Parameters. The minimal and the maximum capacity of the R-tree's node are the key parameters of the R-tree [16]. We assumed the page size of the operating system is $4k$ bytes, and we used an 8-byte integer to identify each page. Therefore, each element in each node of the R-tree will conclude an integer pointed to the child page and four double values to express the boundary of this child node, which takes up 40 bytes of space in a page. Thus, we can compute that the maximum capacity of a node as 100. From experience, the minimal capacity should be about 30% to 40% of maximum capacity, so we set the minimum capacity of a node to be 40.

We trained the *learned cost model* using a set of real-world and synthetic workloads with $10k$ queries in total. Note that the training workloads and the testing workloads are different.

Experimental Environment. All algorithms are implemented by C++. All experiments are conducted on a Ubuntu server with 80 cores of Intel(R) Xeon(R) Gold 6242R CPU @ 3.10GHz and 256GB RAM.

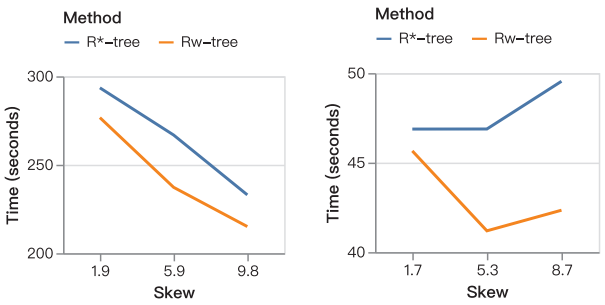
B. Experimental Results

Exp-1: Overall Performance of `RW-tree`.

In the first group of experiments, we evaluate end-to-end performance of `RW-tree` by varying the *skew* and *selectivity* of the query workloads.

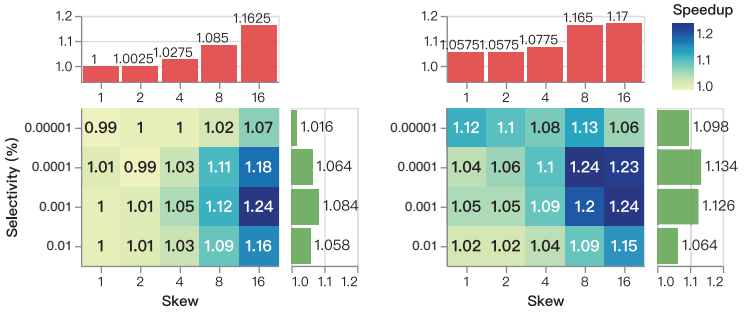
Vary the Skew. For the two synthetic datasets ($i.e.$, Syn-Uni and Syn-Skew), we vary the *skew* in $\{1, 2, 4, 8, 16\}$. Fig. 9 and Fig. 10 show the total execution time of the workload for each dataset, by varying the *skew*. With the increase of *skew*, the gap between `RW-tree` and R*-tree tends to be bigger, since R*-tree tends to organize data instances using square and thus leads to the higher query cost. Thanks to the learned cost model from the historical workloads, `RW-tree` can effectively organize the data instances and thus achieve efficient queries.

Vary the Selectivity. For the two synthetic datasets ($i.e.$, Syn-Uni and Syn-Skew), we also vary the *selectivity* in $\{1 \times 10^{-5}, 1\times10^{-4}, 1\times10^{-3}, 1\times10^{-2}\}$. Fig. 11- 12 depict the total execution time of the workload for each dataset, by varying the *selectivity*. With the increase of *selectivity*, it takes more



(a) OSM-China Dataset (b) OSM-CA Dataset

Fig. 13. Exp-1: Execution Time (seconds) *v.s.* Skew



(a) Speed-up on Syn-Uni Dataset (b) Speed-up on Syn-Skew Dataset

Fig. 14. Exp-2: Speed-up Ratio *v.s.* Skew and Selectivity (%)

execution time for all methods as a larger *selectivity* leads to more data instances to be visited. However, RW-tree still achieves better results by comparing with R*-tree.

Similar observations can be derived by experiments on the two real-world datasets (Fig. 13). We omit to discuss due to space limitations.

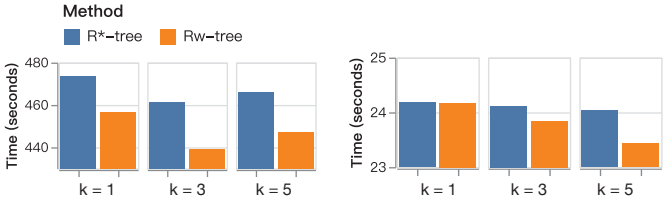
In summary, we make two observations from this group of experiments:

- Compared with R*-tree, RW-tree achieves competitive performance under various *skew* and *selectivity*. The larger the *skew* of the workloads, the better the RW-tree.
- In general, RW-tree is not sensitive to the *selectivity* of the workloads, but RW-tree slightly outperforms R*-tree when the *selectivity* is larger than 0.1%.

Exp-2: Comparison with the State of the art.

Our main purpose in this group of experiments is to test the performance improvement of RW-tree against R*-tree under various *skew* and *selectivity* of the workloads. We compute the speed-up ratio of RW-tree by dividing the workload execution time of R*-tree to RW-tree's.

Fig. 14 summarizes the evaluation results. More concretely, the heatmap shows the speed-up ratio under different *skew* and *selectivity* settings. The bar charts colored in red report the average speed-up ratio by *skew*, while the green bar charts show the average speed-up ratio across different selectivity settings. Next, we will elaborate on the evaluation results. For the synthetic dataset with uniform distribution (Syn-Uni), as shown in the heatmap of Fig. 14(a), we can see that the speed-up ratio is larger than 1.0 in almost all cases. For the synthetic dataset with skew distribution (Syn-Skew), we can see that the speed-up ratio of RW-tree is larger than 1.0 in all cases (Fig. 14(b)). Next, we give a closer look at the two heatmaps. We can see that the larger the skew and selectivity, the larger the speed-up ratio of our method. Both in the Syn-Uni dataset and the Syn-Skew dataset, the maximum speed-up ratio of RW-tree is 1.24×. Furthermore, by comparing the red bar charts and green bar charts, we can make the observations that the *skew* will contribute more to the speed-up ratio of RW-tree. In other words, the higher *skew* of the dataset, the more efficient of RW-tree. Overall, RW-tree shows its



(a) kNN Workload on OSM-CA Dataset (b) kNN Workload on Syn-Uni Dataset

Fig. 15. Exp-3: The performance of kNN Approximation

effectiveness by achieving the speed-up ratio of 1.06 and 1.11 on average on the Syn-Uni dataset and the Syn-Skew dataset, respectively.

Exp-3: Performance of kNN Approximation.

In this set of experiments, we test the performance of the kNN approximation of RW-tree.

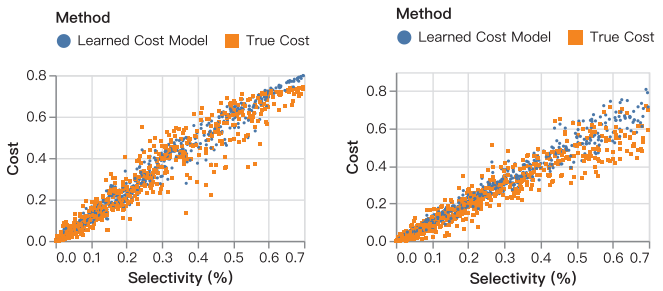
First, we generate kNN workloads from four range query workloads as introduced in Section VI-A. We take the center of the range query as the search point of the kNN query. Next, we run four kNN workloads on the corresponding datasets.

Fig. 15 shows the total execution time of the kNN workload for different datasets. Overall, RW-tree outperforms R*-tree in all testing cases. Moreover, we observe that the performance improvement in the OSM-CA dataset is better than the Syn-Uni dataset. The reason is that the kNN queries will first be transformed into square-based range queries for training the cost model of RW-tree, and thus the dataset following the uniform distribution, the improvement of RW-tree is limited due to the low *skew*.

Exp-4: Performance of Learned Cost Model.

Since the *learned cost model* contributes a lot to the overall performance of RW-tree, we evaluate the performance of our *learned cost model* by comparing it with the true cost of executing the workload.

We use a series of boundaries with different *selectivity* as testing queries. We run the testing queries with RW-tree and get the cost estimated by the *learned cost model*. We execute the testing queries and take the number of actual scanned times of these boundaries as the true cost. We repeat five times to



(a) OSM-China Dataset (b) OSM-CA Dataset

Fig. 16. Exp-4: Performance of Learned Cost Model

cover more cases.

Fig. 16 shows the relationship between the cost estimated by the *learned cost model* and the *true cost* after normalization. We can see that they follow a similar distribution, which indicates our learned cost model works well.

VII. Related Work

We classify the related work of multi-dimensional spatial index into three categories: (1) R-tree and its variants; (2) Packing R-tree indices; (3) Learned Spatial index.

A. R-tree and its Variants

R-tree with Dynamic Data Insertion. Different from the other traditional spatial indexes such as [6], [30] which organize the data instances by space partition, R-tree is a multi-dimensional spatial index that clusters the data instances into a hierarchical structure based on their boundaries. The R-tree [16] and its variants such as R*-tree [4], [5], [14], [31] will cluster the data instances based on some heuristic metrics such as the area enlargement or the overlap of MBRs. These methods focus on the case of dynamic data insertion. As discussed in Section II, these heuristic metrics may lead to a higher time consumption over the workload. This motivates us to find a method to estimate the time cost of a given workload on an R-tree, which leads to the idea of workload-aware R-tree construction.

R-tree with Bulk-loading Insertion. A different line of research works consider to construct R-tree by directly packing the data instances into the leaf nodes instead of inserting each instance individually. In other words, the entire R-tree is bulk-loaded in a bottom-up way. R-tree built with the bulk-loading algorithms such as [17], [19], [23] mostly relies on the order of the data instances, and thus such techniques will suffer a high I/O cost when ordering. Besides, some other packing R-trees consider to order data instances with other factors like features of queries [1], [2]. The query-adaptive idea is similar to the idea of workload-aware way. However, the above two works only consider simple query types and do not support dynamic data insertion.

B. Learned Spatial Index

Learned 1D Index. The key idea of the learned index is to learn the Cumulative Distribution Function (CDF) of the search key. We can directly estimate the position of stored data based on the position of the search key based on the CDF. The learned index is first proposed in RMI [21], which utilizes a *recursive model index* to learn the CDF, then it computes the position of the search key based on the CDF, and thus the stored position of corresponding data can be acquired efficiently. The framework of RMI [21] only supports 1-dimensional data and does not support dynamic data updates. Several learned indexes on 1-dimensional data [10], [12], [13], [20], [36] have been proposed to support dynamic data updates. Although such techniques overcome the weakness of dynamic data updates of RMI, they do not support multi-dimensional data.

Learned Spatial Index. From learned 1D index to learned spatial index, the key challenge is how to effectively organize multi-dimensional data instances. One alternative method is to utilize the space-filling curve to order all the data instances, which is studied by [27], [34]. Different from that, LISA [25] partitions the whole data space using grids, and then learns the data distribution based on these grids. Flood [26] takes the workload into consideration to optimize the learning process. Tsunami [11] extends Flood [26] by overcoming the limitations of skewed workload and correlated datasets. ML-Index [9] maps point instances to a 1-dimensional space and then learns the CDF, which also follows the idea of mapping multi-dimensional data to one-dimensional space. Although the above methods achieve good performance on static workload, the main limitation is that they do not support workload with dynamic data insertion. Besides, ML models can also be utilized to improve the performance of the database [8], [22], [38], [39], such as query optimization [24], [32], [33], [33], [35], [37], [40].

VIII. Conclusion

In this paper, we study the problem of workload-aware R-tree construction. Given a workload, we design an R-tree construction framework such that queries with the same distribution as the workload can be efficiently executed on the R-tree. We propose a learning-based method to learn the distribution of the workload. Then we propose a cost model to measure how a data insertion strategy performs and select the best one. Also, we support both range search and kNN queries in the workload. Experimental results show that our method outperforms the baseline in terms of the query efficiency.

Acknowledgment

This work is supported by NSF of China (61925205, 62102215, 62072261), Huawei, TAL education, China National Postdoctoral Program for Innovative Talents (BX2021155), China Postdoctoral Science Foundation (2021M691784), Shuimu Tsinghua Scholar and Zhejiang Lab's International Talent Fund for Young Professionals.

REFERENCES

[1] D. Achakeev, B. Seeger, and P. Widmayer. Sort-based query-adaptive loading of r-trees. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2080–2084, 2012.

[2] D. Achakeev, M. Seidemann, M. Schmidt, and B. Seeger. Sort-based parallel loading of r-trees. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 62–70, 2012.

[3] L. Arge, M. D. Berg, H. Haverkort, and K. Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. *TALG*, 4(1):1–30, 2008.

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

[5] N. Beckmann and B. Seeger. A revised r*-tree in comparison with related index structures. In *SIGMOD Conference 2009*, pages 799–812, 2009.

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

[7] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The x-tree: An index structure for high-dimensional data. In *Very Large Data-Bases*, pages 28–39, 1996.

[8] C. Chai, J. Wang, Y. Luo, Z. Niu, and G. Li. Data management for machine learning: A survey. *TKDE*, pages 1–1, 2022.

[9] A. Davitkova, E. Milchevski, and S. Michel. The ml-index: A multidimensional, learned index for point, range, and nearest-neighbor queries. In *EDBT*, pages 407–410, 2020.

[10] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, et al. Alex: an updatable adaptive learned index. In *SIGMOD Conference 2020*, pages 969–984, 2020.

[11] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *arXiv preprint arXiv:2006.13282*, 2020.

[12] P. Ferragina and G. Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB*, 13(8):1162–1175, 2020.

[13] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1189–1206, 2019.

[14] D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of the Fifth International Conference on Data Engineering, February 6-10, 1989, Los Angeles, California, USA*, pages 606–615. IEEE Computer Society, 1989.

[15] T. Gu, K. Feng, G. Cong, C. Long, Z. Wang, and S. Wang. The rlr-tree: A reinforcement learning based r-tree for spatial data. *arXiv preprint arXiv:2103.04541*, 2021.

[16] A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[17] H. Haverkort and F. V. Walderveen. Four-dimensional hilbert curves for r-trees. *JEA*, 16:3–1, 2008.

[18] D. Hilbert. Über die stetige abbildung einer linie auf ein flächenstück. In *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes*, pages 1–2. Springer, 1935.

[19] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. Technical report, 1993.

[20] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. Radixspline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–5, 2020.

[21] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

[22] H. Lan, Z. Bao, and Y. Peng. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering*, 6(1):86–101, 2021.

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

[24] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.*, 12(12):2118–2130, 2019.

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

[26] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In *SIGMOD Conference 2020*, pages 985–1000, 2020.

[27] J. Qi, G. Liu, C. S. Jensen, and L. Kulik. Effectively learning spatial indices. *PVLDB*, 13(12):2341–2354, 2020.

[28] J. Qi, Y. Tao, Y. Chang, and R. Zhang. Packing r-trees with space-filling curves: Theoretical optimality, empirical efficiency, and bulk-loading parallelizability. *TODS*, 45(3):1–47, 2020.

[29] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD Conference 1995*, pages 71–79, 1995.

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

[31] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. Technical report, 1987.

[32] J. Sun and G. Li. An end-to-end learning-based cost estimator. *Proc. VLDB Endow.*, 13(3):307–319, 2019.

[33] J. Sun, J. Zhang, Z. Sun, G. Li, and N. Tang. Learned cardinality estimation: A design space exploration and A comparative evaluation. *Proc. VLDB Endow.*, 15(1):85–97, 2021.

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

[35] J. Wang, C. Chai, J. Liu, and G. Li. FACE: A normalizing flow based cardinality estimator. *PVLDB*, 15(1):72–84, 2021.

[36] J. Wu, Y. Zhang, S. Chen, J. Wang, Y. Chen, and C. Xing. Updatable learned index with precise positions. *arXiv preprint arXiv:2104.05520*, 2021.

[37] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement learning with tree-lstm for join order selection. In *ICDE*, pages 1297–1308, 2020.

[38] C. Zhan and M. S. et al. Analyticdb: Real-time OLAP database system at alibaba cloud. *PVLDB*, 12(12):2059–2070, 2019.

[39] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. *IEEE TKDE*, 34(3):1096–1116, 2022.

[40] X. Zhou, G. Li, C. Chai, and J. Feng. A learned query rewrite system using monte carlo tree search. *PVLDB*, 15(1):46–58, 2021.