PLATON: Top-down R-tree Packing with Learned Partition Policy

JINGYI YANG, Nanyang Technological University, Singapore
GAO CONG, Nanyang Technological University, Singapore

The exponential growth of spatial data poses new challenges to the performance of spatial databases. Spatial indexes like R-tree greatly accelerate the query performance and can be effectively constructed through packing, *i.e.*, loading all data into the index at once. However, existing R-tree packing methods rely on a set of fixed heuristic rules, which may not be suitable for different data distributions and workload patterns. To address the limitations of existing R-tree packing methods, we propose PLATON, a top-down R-tree packing method with learned partition policy that explicitly optimizes the query performance with regard to the given data and workload instance. We develop a learned partition policy based on Monte Carlo Tree Search and carefully make design choices for the MCTS exploration strategy and simulation strategy to improve algorithm convergence. We propose a divide and conquer strategy and two optimization techniques, early termination and level-wise sampling, to drastically reduce the MCTS algorithm's time complexity and make it a linear-time algorithm. Experiments on both synthetic and real-world datasets demonstrate the superior performance of PLATON over existing R-tree variants and recently proposed learned/workload-aware spatial indexes.

CCS Concepts: • **Information systems → Database management system engines**.

Additional Key Words and Phrases: Spatial index, Spatial query processing, Learned index, Monte carlo tree search

ACM Reference Format:
Jingyi Yang and Gao Cong. 2023. PLATON: Top-down R-tree Packing with Learned Partition Policy. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 253 (December 2023), 26 pages. https://doi.org/10.1145/3626742

1 INTRODUCTION

Spatial data is rapidly generated from mobile GPS sensors through location-based apps like Uber and Google Reviews, from embedded GPS sensors in vehicles, vessels, and airplanes, and from a vast number of Lidar sensors with the increasing adoption of self-driving technology. With the exponential growth of spatial data, there is an increasing need for database systems to efficiently manage and analyse spatial data. One of the key components of a spatial database is the spatial index, a data structure that facilitates spatial query processing. Under the ongoing trend of applying machine learning techniques to enhance database systems, several learning-based methods have been proposed to improve the spatial index performance. Existing work on machine learning for spatial indexes can be categorized into two classes: learned spatial indexes and ML-enhanced spatial indexes. Learned spatial indexes use machine learning models to map spatial coordinates to storage locations [40, 46, 47, 53], while ML-enhanced spatial indexes [10, 21, 26] learn to enhance certain operations, *e.g.*, insertion and search operations, of traditional spatial indexes like R-tree.

Authors' addresses: Jingyi Yang, jyang028@e.ntu.edu.sg, Nanyang Technological University, Singapore; Gao Cong, gaocong@ntu.edu.sg, Nanyang Technological University, Singapore.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2836-6573/2023/12-ART253
https://doi.org/10.1145/3626742

Learned spatial indexes have several limitations. Firstly, depending on the index design, some learned spatial indexes may produce approximate results for certain types of query, *e.g.*, KNN query [47, 53]. Secondly, existing learned spatial indexes only support point data, and cannot handle data with geometry. Thirdly, most learned indexes model the cumulative density function (CDF) of the data [40, 53], which makes them vulnerable to data poisoning attacks [35]. As a consequence, it would be a long way for real-world systems to integrate learned spatial indexes. In contrast, the ML-enhanced spatial indexes can be more easily deployed into real-world systems, as they are built on top of popular spatial indexes like R-tree, which are already widely used in DBMS. Moreover, as the basic structure and properties of R-tree are unchanged, existing query processing algorithms remain applicable.

Existing ML-enhanced spatial indexes [21, 27] construct an R-tree through one-by-one insertion, and use ML techniques to optimize the *chooseSubtree* and *splitNode* process. However, in real-world applications, R-tree packing, or bulk-loading is usually performed at index construction, because packing constructs an R-tree with better space utilization and query performance compared to one-by-one insertion. Moreover, packing is performed during the periodic reload of clustered indexes to optimize the index performance. As we will discuss, existing methods for R-tree packing suffer from two major limitations, which motivate the use of machine learning techniques to enhance R-tree packing.

Two classes of methods have been proposed for R-tree packing, bottom-up methods and top-down methods. Bottom-up [12, 15, 17, 30, 31, 33, 39, 48, 49] methods build the R-tree from leaf nodes upward by sorting the data objects in a heuristic order with a good clustering property. Top-down methods [25, 38] build the R-tree through recursive partitioning, with a partition policy that greedily optimizes a heuristic cost function, *e.g.*, the sum of the area of node minimum bounding rectangles (MBRs). The first limitation of existing methods is that both bottom-up and top-down methods are not adaptive to different data distributions and workload patterns, as they rely on a fixed set of heuristic rules. Bottom-up methods use a heuristic sort order, while top-down methods use a heuristic cost function. The use of heuristic rules results in the constructed R-tree only performing well for certain data distributions and workload patterns, while performing poorly for others. The second limitation is that existing top-down packing methods ignore the dependencies between the partition decisions on different nodes, which leads to a partition policy that only makes locally optimal partitions. We will discuss the two limitations in detail in Section 2.4.

In this work, we propose PLATON: top-down R-tree **P**acking with **L**earned p**A**r**T**iti**ON** policy, which addresses the two limitations of the existing packing methods. PLATON adopts a top-down packing framework and explicitly optimizes the query performance with regard to the given data and workload instance, so that it is adaptive to any data distribution and workload pattern.

We prove in Section 3 that optimal R-tree packing with regard to the given data and workload instance, is an NP-hard problem. While it is possible to design greedy partition policies that optimize a heuristic function with regard to the given data and workload, such an approach, like existing top-down methods, ignores the dependencies between the partition decisions on different nodes. To identify the partition policy that optimizes query performance while avoiding locally optimal partition actions, we propose to learn a partition policy using an effective and lightweight RL technique, Monte Carlo Tree Search (MCTS), which maximizes the long-term reward of the partition actions.

The partition problem setup, however, poses unique challenges to the design of the MCTS algorithm, specifically, to the design of the exploration strategy and simulation strategy in MCTS. As the return of a state across different rollouts usually has a high variance, the standard *Upper Confidence Bound for Trees* (UCT) selection policy no longer works well. Meanwhile, due to a long action sequence and a huge state space, the standard random rollout policy leads to slow

convergence. We carefully design these two policies to address the challenges, as we will discuss in Section 5.4.

Another main obstacle in learning a partition policy using MCTS is that the MCTS algorithm has a time complexity of $O(k \cdot N^2 \log N)$ (N is the size of data, k is the number of MCTS iterations), which is not scalable to large datasets. To address this challenge, we develop a divide and conquer strategy, which drastically reduces the size of the state space. We further propose two optimization techniques, early termination and level-wise sampling, to reduce the time complexity to $O(k \cdot N)$. Moreover, we consider the case when no training workload is available, and propose to learn from a synthetic query workload following the data distribution.

Contributions. This work makes the following contributions.

- We propose PLATON, a top-down R-tree packing method adaptive to different data distributions and workload patterns. We formulate the optimal R-tree packing problem with regard to the given data and workload instance, and prove its NP-hardness. (Sec. 3) To take into account the dependencies between the partition decisions at different nodes, we develop a learned partition policy based on Monte Carlo Tree Search and carefully designed the selection policy and rollout policy for our problem to achieve better convergence. (Sec. 5)
- We propose a divide and conquer strategy, and two optimization techniques to reduce the MCTS algorithm complexity from higher than quadratic time to linear time. (Sec. 6)
- When a workload is not available at the time of index construction, we propose to optimize the I/O cost of a query workload that follows the data distribution. (Sec. 7)
- We integrate PLATON into two real-world systems, libspatialindex and PostgreSQL, and conduct comprehensive experiments to show that PLATON significantly outperforms existing R-tree variants and recently proposed learned spatial indexes.

2 BACKGROUND

We first provide the preliminaries in Section 2.1. We then review two classes of existing methods for R-tree packing, bottom-up methods and top-down methods, in Section 2.2 and Section 2.3 respectively. We discuss the limitations of existing methods in Section 2.4.

2.1 Preliminaries

Consider a set $\mathcal{D} = \{R_1, ...R_N\}$ of N (hyper-)rectangle data objects in a d-dimensional Euclidean space. R-tree packing considers the problem of constructing a balanced R-tree T from $\mathcal{D}$ with a node capacity B.

We label the levels of the R-tree in a bottom-up manner, with leaf nodes being level 1 and root nodes being level $\lceil \log_B N \rceil$. Each node at level t indexes at most B^t data objects. For ease of presentation, we use $d = 2$ in the following discussion, but the discussion can be generalized to any $d > 2$.

2.2 Bottom-up R-tree Packing

Bottom-up methods [12, 15, 17, 31, 33, 39, 48, 49] start by packing data objects into leaf nodes, and then repeatedly build the tree upward by packing lower-level nodes into nodes one level higher. To achieve good query performance and create tree nodes with minimal overlap, bottom-up methods typically rely on a heuristic sort order with a good clustering property, *i.e.*, after sorting the data objects in the heuristic order, neighboring objects are spatially close. Example heuristic sort orders include Hilbert ordering [17, 31, 33], Sort-Tile-Recursive [39], as well as space-filling curve-based ordering in rank space [48]. Bottom-up methods first sort data objects in the heuristic sort order, and pack every B consecutive objects into a leaf node. This process then repeats over the Minimum

Bounding Rectangles (MBRs) of nodes at level t to create nodes at level $t+1$. Bottom-up packing finishes when the root node of the tree is built. We note that a special case is the Priority R-tree [12] (PR-tree), which does not rely on a sort order. However, the construction of PR-tree nodes also involves heuristic rules.

2.3 Top-down R-tree packing

Top-down methods [25, 38] start with the root node, and construct the R-tree downward by recursively partitioning higher-level nodes into nodes one level lower. To illustrate the top-down packing process, we first define several key concepts.

Definition 2.1 (Partially Constructed Tree). A partially constructed tree is a B-ary tree that represents an intermediate state during the top-down R-tree packing process. A partially constructed tree node n corresponds to one or more R-tree nodes and is described by two fields, $n.data$ and $n.level$. $n.data$ represents the set of data to be indexed, while $n.level$ represents the level of the corresponding R-tree nodes.

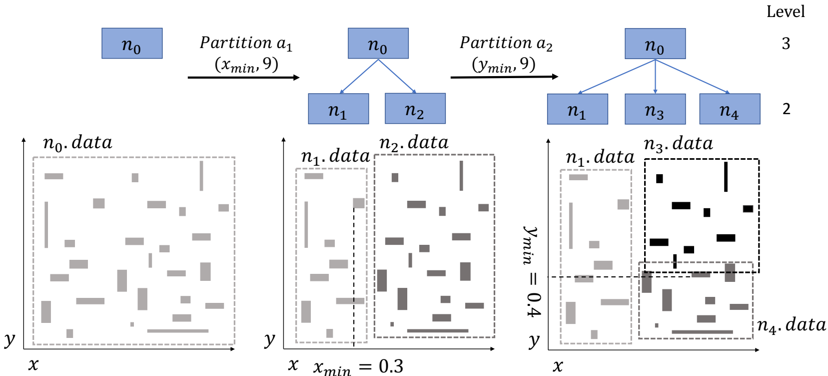

Fig. 1. Top-down R-tree packing example, $N = 27$, $B = 3$.

Definition 2.2 (Partition Action). A partition action a on a partially constructed tree node n partitions $n.data$ into two subsets, and creates two new nodes. Each partition action a is described by a pair of fields (dim, pos), where dim denotes the dimension along which to partition the data, and pos denotes the position of the partition. To build a balanced tree, at least one child node's size should be an integer multiple of the corresponding R-tree node's size, i.e., one of the child nodes n' satisfies $|n'.data| = i \times B^{n'.level}$, where i is a positive integer.

Example. Figure 1 shows the first three partially constructed trees during top-down packing. The initial partially constructed tree only consists of a single root node n_0. $n_0.data = \mathcal{D}$, and $n_0.level = 3$. The first partition acts on the root node n_0 of the initial state, with $dim = x_{min}$ and $pos = 9$. We first sort $n_0.data$ by their x_{min} values, and partition them into two sets. The first 9 objects with the smallest x_{min} form node n_1, while the rest 18 objects form node n_2. Both n_1 and n_2 are at level 2, as they correspond to R-tree nodes at level 2. The second partition acts on the node n_2 with $dim = y_{min}$ and $pos = 9$. Similarly, we create two new nodes n_3 and n_4. Both nodes are at the same level as n_2, and we remove n_2 from the tree.

Partition Policy. At each step, there are multiple candidate partition actions for the given partially constructed tree node n. Specifically, along each dimension, there are $\lceil \frac{|n.data|}{B^{n.level}} - 1 \rceil$ possible partition positions given the constraints in Definition 2. Assuming c possible dimensions, *e.g.*, x_{min}, y_{min}, to sort the data, this gives us a total of $c \cdot \lceil \frac{|n.data|}{B^{n.level}} - 1 \rceil$ candidate partition actions. The key to

top-down packing is the partition policy, which chooses a partition action among all the candidates at each step, so that the constructed R-tree has good query performance. Existing top-down packing methods like TGS [25] proposed partition policies that greedily minimize a heuristic cost function, *e.g.,* the sum of the area of node MBRs.

Top-down packing starts with a partially constructed tree of a single node n_0, which corresponds to the root of the R-tree. n_0 is initialized with $n_0.data = D$, $n_0.level = \lceil \log_B N \rceil$. The first partition action a_1 is chosen by the partition policy, and acts on the root n_0, which bi-partitions $n_0.data$ to form two nodes at the next level. We then recursively choose and perform partition actions on the newly created partially constructed tree nodes, until all leaf nodes of the R-tree are created.

2.4 Limitation of Existing Methods

Limitation 1: Not adaptive to different data distributions and workload patterns. One common limitation of both classes of packing methods is that they rely on heuristic rules during tree construction, which only work well for certain data distributions and workload patterns, and fail for the others.

As mentioned, bottom-up packing methods rely on a heuristic sort order with a good clustering property in order to work well. However, this is not always the case, as no heuristic sort order works for all data distribution and workload patterns. The left part of Figure 2 shows a packing example using ZR [48], the most recently proposed bottom-up packing method. It first maps the data objects from the euclidean space to the rank space, and then sort them using a space-filling curve, *e.g,* Z-curve. With the heuristic sort order, ZR packs $\{R_2, R_3, R_4\}$, $\{R_8, R_1, R_5\}$, $\{R_7, R_6, R_9\}$ into three leaf nodes. We note that the leaf node $\{R_8, R_1, R_5\}$ has a huge MBR that almost spans the entire data space, because although R_8 and R_1 are neighbors under the heuristic sort order, they are far apart spatially. As a result, this leaf node needs to be accessed for queries that do not intersect with any object in the node. For instance, given the query q, the node $\{R_8, R_1, R_5\}$ is accessed, although the query only intersects with R_2 and R_3, leading to 2 leaf node I/O. An R-tree with better performance on the given data and query is shown on the right of Figure 2, in which case the leaf node I/O is 1.

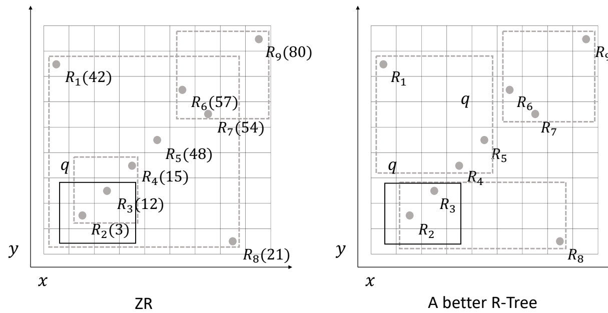

Fig. 2. Example failure of bottom-up packing, $N = 9$, $B = 3$.

Top-down methods explicitly optimize a heuristic cost function like the sum of the area of node MBRs. However, such heuristics may not be effective for all data distributions and workload patterns. As shown in Figure 3, certain data distributions and workload patterns can make the heuristic cost function fail. The left part of the figure shows the four leaf nodes created using TGS. As TGS makes its partition decisions by greedily minimizing the sum of the area of node MBRs, it packs the data points into leaf nodes with the shape of long and thin stripes. Given the query q, which spans a large range along the y-axis, the result R-tree has poor performance because every leaf node needs to be visited. An R-tree with better performance on the given data and query is

shown on the right of Figure 3. Only 1 leaf node I/O is needed for the same query q. Note that bottom-up methods can also fail due to the heuristic nature of the sort order used. A detailed example can be found in the appendix.

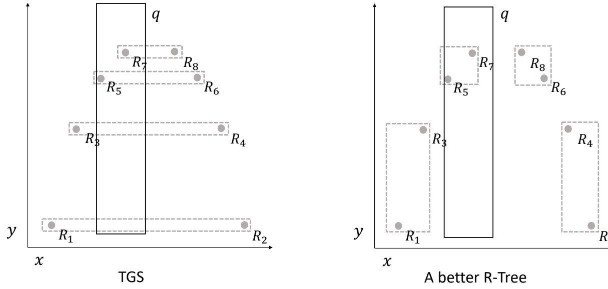

Fig. 3. Example failure of TGS, $N = 8$, $B = 2$.

Limitation 2: Ignoring the dependencies between partition decisions. Existing top-down packing methods greedily choose the partition that optimizes a heuristic cost function, and the partition decisions are independently made at each step. However, the partition decisions are not independent of one another, because the partition on a node can affect subsequent partitions on its lower-level child nodes. Ignoring the dependencies between the partitions on the parent nodes and its child nodes results in locally optimal partition decisions and R-tree structures that are not optimized with regard to the cost function.

Our work aims at addressing the limitations of existing methods and designing a packing strategy that optimizes the index performance for any given data distribution and workload patterns.

3 OPTIMAL R-TREE PACKING

In this section, we define the problem of optimal R-tree packing with regard to the given data and workload instance, and show the NP-hardness of the problem.

Problem: Optimal R-tree Packing with regard to data and workload. Given a set $\mathcal{D} = \{R_1, ...R_N\}$ of N (hyper-)rectangle data objects and a workload $\mathcal{W} = \{q_1, ...q_M\}$ of M window queries in a d-dimensional Euclidean space. Our goal is to construct a balanced R-tree T from $\mathcal{D}$ with a node capacity B, so that the query performance is optimized for workload $\mathcal{W}$. Specifically, we consider the case of constructing a disk-based R-tree, where each node is stored on a disk page. The I/O cost $C(T, \mathcal{W})$ is measured by the total number of nodes accessed by the queries.

Theorem 3.1. *Optimal R-tree packing with regard to the workload I/O cost $C(T, \mathcal{W})$ is NP-hard.*

Proof. To prove the theorem, we prove for the case of a 2-level R-tree. Optimal packing of a 2-level R-tree is equivalent to finding the optimal partitioning of $\mathcal{D}$ into leaf nodes $P = \{p_1, p_2, ..., p_l\}, B/2 \le |p_i| \le B$, such that for a given weight function $w\colon p_i \to \mathbb{R}^+$, the sum of the weights is minimized. In our case, $w(p) = \sum_{q \in \mathcal{W}} \mathbb{1}_{hasOverlap(p,q)}$, *i.e.*, the weight of a partition is the number of query windows it overlaps. We prove the NP-hardness by a reduction from the optimal partitioning problem without considering the workload [11], which shows that for the weight function $w(p) = area(MBR(p))$, the optimal partitioning problem is NP-hard.

An instance of the optimal partitioning problem [11] consists of a dataset $\mathcal{D}$ of N rectangles. The problem finds a partitioning $P = \{p_1, p_2, ..., p_l\}, B/2 \le |p_i| \le B$, that minimizes $\sum_{i=1}^{l} area(MBR(p_i))$. We assume that all rectangles have integer coordinates. We reduce this instance of optimal partitioning to our problem with the same dataset $\mathcal{D}$, and construct a workload such that we have an equivalent weight functions. We construct our query workload $\mathcal{W}$ to be the set of 1×1 squares that

packs the data space, *i.e.*, $\mathcal{W} = \{(x, y, x{+}1, y{+}1) \mid x \in [X_{min}, X_{max}{-}1], y \in [Y_{min}, Y_{max}{-}1], x, y \in \mathbb{Z}\}$, where $X_{min}, X_{max}, Y_{min}, Y_{max}$ are the minimal and maximal coordinates of the input data along each dimension. It is easy to see that $w(p) = \sum_{q \in \mathcal{W}} \mathbb{1}_{hasOverlap(p,q)} = area(MBR(p))$. As a result, a solution to the optimal partition problem [11] is a solution to the optimal partition problem with regard to data $\mathcal{D}$ and the constructed workload $\mathcal{W}$, and vice versa. This completes the proof.

4 OVERVIEW OF PLATON

PLATON aims at exploiting the advantage of the top-down R-tree packing framework, while addressing its limitations. We summarize the design goals and our solutions as follows.

1. **Goal: Optimizing query performance**. Optimal R-tree packing with regard to a given data and workload instance is NP-hard. While existing top-down packing methods propose partition policies that greedily optimize a heuristic cost function, such policies lead to locally optimal partitions.
   Solution: To address this issue, we frame top-down packing as an MDP and leverage an effective, lightweight RL technique, Monte Carlo Tree Search (MCTS) to learn a partition policy that optimizes the long-term reward.
2. **Goal: Designing exploration and simulation strategy for better convergence**. The partition problem setup makes it hard for MCTS to converge due to a huge state space and the high variance of simulation returns across different rollouts.
   Solution: We carefully design the MCTS exploration strategy and simulation strategy to achieve better convergence. We incorporate domain knowledge of R-tree packing into MCTS simulation by proposing a greedy rollout policy.
3. **Goal: Efficiently learning the partition policy**. While the learned partition policy achieves good query performance, the MCTS algorithm, despite being lightweight compared with many other RL techniques, still has a high time complexity, which makes it hard to scale to large datasets.
   Solution: We propose a divide and conquer strategy, and two optimization techniques, early termination and level-wise sampling, which drastically reduce the time complexity of the MCTS algorithm and makes it a linear-time algorithm.

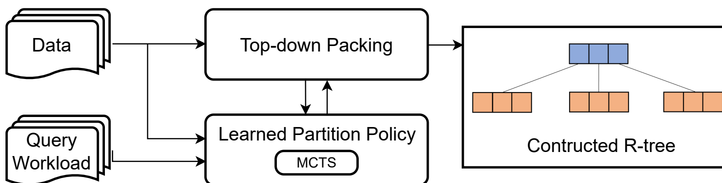

Fig. 4. Workflow of PLATON.

Workflow of PLATON. Figure 4 shows the high-level workflow of PLATON. The inputs include the dataset and a training workload representative of the workload patterns. We construct an R-tree on the data in a top-down manner. At each step, the partition action is chosen based on the learned policy using MCTS, which optimizes the query performance for the given data and workload instance. This process is repeated until the R-tree is fully constructed.

Insertion and Deletion. Like all prior packed R-trees, PLATON is compatible with existing R-tree insertion and deletion methods, *e.g.*, the methods used in R*-tree or learning-based methods like RLR-tree [26] .

5 LEARNING PARTITION POLICY

We first discuss the motivation for learning the partition policy in Section 5.1. We then discuss how to frame top-down packing as a MDP in Section 5.2. We propose to learn a partition policy using Monte Carlo Tree Search (MCTS) in Section 5.3. We discuss our design of MCTS exploration and simulation strategy in Section 5.4.

5.1 Motivation for Learned Partition Policy

Due to the NP-hardness of the optimal R-tree packing problem, one can only design approximate algorithms for the problem. To develop a partition policy that minimizes the I/O cost, a natural idea is to adopt an approach similar to TGS [25], which chooses the partition action that greedily optimizes a heuristic at each step. However, this approach ignores the dependencies between the partition decisions at different nodes. As the partition on a node can affect subsequent partitions on its lower-level child nodes, the greedy partition policy only makes locally optimal partition decisions by choosing the partition with the highest immediate reward.

We identify that choosing the partition action at each step is a sequential decision-making process. To avoid making locally optimal decisions, we consider a class of algorithms that optimizes the long-term rewards in sequential decision-making, reinforcement learning (RL) [52].

5.2 Top-down Packing as an MDP

As discussed in Section 5.1, choosing the partition action at each step is a typical sequential decision-making process. We now discuss how we can frame it as a Markov Decision Process (MDP). We define the components of our MDP as follows. **State space**: The state space S, is defined as the set of all possible partially constructed R-trees, and a state $s \in S$ is represented by a partially constructed R-tree. A state is *terminal* if the corresponding R-tree is fully constructed. **Action space**: Without loss of generality, we define the action space A for a state, *i.e.*, partially constructed R-tree, to be the set of possible partition actions on the left-most unpartitioned node. As there may be multiple unpartitioned nodes for a partially constructed R-tree, we fix a node to partition so that we reduce the action space. **Transition probability**: The state transition function P_a is deterministic. Taking an action a on a state s transits into a new partially constructed tree state s' with 100% probability.

Reward function: The reward function of our MDP should be designed in a way so that the objective of our MDP is equivalent to minimizing the I/O cost of the given workload. However, the I/O cost of an R-tree can only be computed when the R-tree is fully constructed. If the reward is only associated with the last action, the extremely sparse reward makes it hard to learn a good policy. To address this problem, we process a way to break down the objective into a series of rewards associated with each partition action. We first transform the objective of minimizing the workload I/O cost into the equivalent objective of maximizing the number of skipped page access. Consider an R-tree T and a workload $\mathcal{W}$, the number of skipped page access $S(T, \mathcal{W})$ refers to the total number of disk pages one avoid accessing due to the R-tree T when executing the workload $\mathcal{W}$.

$$S(T, \mathcal{W}) = \sum_{q \in W} \sum_{n \in T} \mathbb{1}_{\neg hasOverlap(n,q)} \tag{1}$$

where $hasOverlap(n, q)$ computes whether the MBR of a partially constructed tree node n overlaps query q. It is easy to see that maximizing the number of skipped page access $S(T, \mathcal{W})$ is equivalent to minimizing the I/O cost $C(T, \mathcal{W})$. While we can only calculate the I/O cost when the R-tree is fully constructed, we can define each partition action's contribution to the number of skipped page access.

To see how a partition action leads to skipped page access, we first look at an example in Figure 5. The partition action a_1 creates two partially constructed tree nodes, n_1 and n_2. n_1 overlaps with the

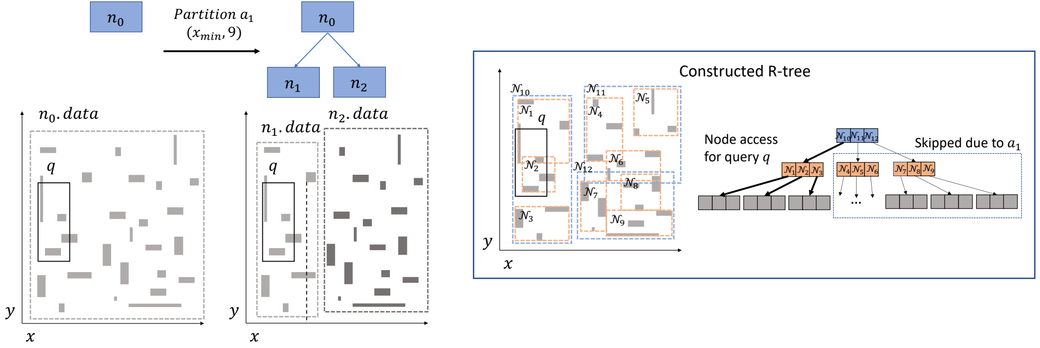

Fig. 5. How partition actions lead to skipped page access.

query q, while n_2 does not overlap with query q. Because n_2 does not overlap with query q, any node created by further partitioning n_2 does not overlap with query q either. As a result, we can skip all the subtrees that index $n_2.data$ when executing query q, *i.e.*, the two subtrees rooted at $\mathcal{N}_{11}$ and $\mathcal{N}_{12}$ in the constructed R-tree. In this case, a_1 leads to a total of $2 \times (1 + 3) = 8$ skipped disk pages.

Definition 5.1 (Reward of a partition action). Assume partition action a on partially constructed tree state s partitions node n, and creates two new nodes n_1 and n_2, the reward of the action is defined as:

$$
\begin{aligned}
Reward(s, a) = \sum_{q \in \mathcal{W}} & \mathbb{1}_{\neg hasOverlap(q,n)} \cdot (\mathbb{1}_{hasOverlap(q,n1)} \\
& \cdot pageSize(n_1) + \mathbb{1}_{hasOverlap(q,n2)} \cdot pageSize(n_2))
\end{aligned}
\tag{2}
$$

where $pageSize(n) = (|n.data|/B^{n.level}) \cdot \frac{B^{n.level}-1}{B-1}$ computes the number of disk pages to store the subtree(s) that indexes $n.data$ in the fully constructed tree. For each query, we sum up the size of nodes that do not overlap with the query. This value is then summed over all queries in the workload to produce the reward of a partition action. To avoid redundant reward computation, page skipping should only be counted if the node n before partitioning overlaps with the query.

The objective of an MDP is to find the optimal policy function π that maximizes the expected return $V_\pi(s)$ of a state s following the policy. The expected return $V_\pi(s)$ is the expected discounted sum of rewards with decay factor γ. Under our formulation, we set $\gamma = 1$, so maximizing the expected return is equivalent to maximizing the total number of skipped page access.

5.3 Monte Carlo Tree Search algorithm

To find the partition policy that optimizes the I/O cost while taking the dependencies between the partition decisions into consideration, we propose to use Monte Carlo Tree Search (MCTS), an effective and lightweight search algorithm. We first motivate our use of MCTS over other common families of RL techniques like dynamic programming, value-based RL (*e.g.* Q-learning), and policy gradient methods.

- **Efficiency and Effectiveness.** Due to the large state space, a dynamic programming algorithm for our problem has exponential complexity. Value-based (typically temporal difference methods) and policy gradient methods require neural networks to approximate the value/policy function. They are slow to train (several days for 1 million records) even with lightweight models and hardly converge for our problem. In contrast, MCTS is a statistical anytime algorithm [16], *i.e.*, it can work under any given amount of computing power, while more computing power leads to better solutions to the problem. MCTS achieves much better convergences under limited time and computing resources compared to value-based and policy gradient methods.

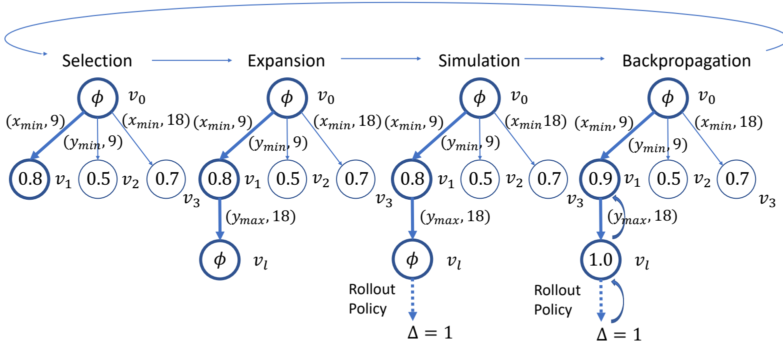

Fig. 6. Overview of MCTS.

- **Free of Tuning.** RL techniques that use neural networks to learn the value/policy function usually require hyper-parameter tuning during training. As the optimal set of hyper-parameters differs across datasets, the time-consuming tuning process makes it costly to apply these techniques to new data. In comparison, MCTS is free of hyper-parameter tuning, and therefore much easier to apply on new instances.

MCTS is a simulation-based search algorithm that explores the state space in the form of a search tree. Each search tree node v corresponds to a state s, *i.e.*, partially constructed R-tree. We use $v(s)$ to represent the search tree node corresponding to the state s, and $s(v)$ to represent the state corresponding to the search tree node v. Note that a search tree is not the R-tree being constructed. During the search process, MCTS leverages Monte Carlo methods to evaluate the action value function $Q(s, a)$ of state-action pairs, *i.e.*, the expected return from state s by taking action a. In our problem, the Q-values represent the expected return from a partially constructed R-tree by taking a particular partition action. The Q-values are approximated by averaging the returns from a number of simulated episodes, or rollouts. By leveraging the approximated Q-values of explored states and actions, MCTS progressively builds a search tree that focuses on the branches with the most promising states. We maintain two values for each search tree node v, $N(v)$ and $Q(v)$. $N(v)$ is the number of times a state is visited, and $Q(v)$ is the approximated return of the state from the Monte Carlo evaluations. Given that $v.parent$ transit to v with action a, $Q(v)$ effectively approximates the action value function $Q(s(v.parent), a)$.

MCTS involves an iterative process. In each iteration, the search tree grows by one node and the approximated returns of the explored states are updated. As the number of iterations increases, the approximated returns of the states gradually approach the expected returns. An iteration consists of four steps: Selection, Expansion, Simulation and Backpropagation. The number of iterations is dependent on the computational budget. We use the example in Figure 6 to show how the search tree evolves over an iteration. For simplicity, we only show $Q(v)$ for each search tree node v, and ϕ denotes no approximated return available.

- **Selection.** MCTS starts from the root v_0, and recursively descends into the child nodes following a $TreePolicy$ to find the most promising search tree node whose child nodes will be explored. In the figure, assuming the $TreePolicy$ simply finds the child node v with the largest approximated return $Q(v)$. MCTS starts from the search tree root, and descends into node v_1 with the partition action $(x_{min}, 9)$, which has the highest approximated return among the three child nodes. Since v_1 has unexplored child nodes, we select v_1 to be the node to expand.
- **Expansion.** MCTS randomly adds one of the selected node's child v_l to the search tree. In the figure, we choose a random partition action from the state $s(v_1)$, *e.g.*, $(y_{max}, 9)$, and act on it. We add the result state to the search tree as node v_l.

- **Simulation.** MCTS performs a rollout from the new node v_l to approximate its return. A rollout is a Monte Carlo evaluation of the node v_l, where a sequence of actions is taken following a *RolloutPolicy* until a terminal state is reached. The return of the actions from the rollout, Δ, is obtained. In the figure, we perform a sequence of random actions from v_l, until the R-tree is fully constructed. We record the return $\Delta = 1$.
- **Backpropagation.** After obtaining the return Δ from a rollout, we use it to update the $N(v)$ and $Q(v)$ values of nodes on the path from v_l to v_0. In the figure, we update the approximated return of the nodes v_l and v_1 with the return $\Delta = 1$ from the current rollout. $Q(v_l)$ becomes 1, and $Q(v_1)$ increases to 0.9.

An overview of PLATON with MCTS is shown in Algorithm 1. We start by initializing a partially constructed R-tree T with a single root node (line 1). We maintain three variables, *canSplit* tracks if the tree can be further partitioned, *currentLevel* tracks the tree level currently being partitioned, and *nextNode* records the next node to partition (line 2). We partition the tree level-by-level, starting from the top. As long as the tree can be partitioned (line 3), we iterate through nodes at *currentLevel*, and finds the left-most unpartitioned node *nextNode* (lines 4–7). If it is found, we run MCTS to find a good partition action (line 9). Specifically, we first create a search tree root with the current partially constructed R-tree T (line 15), and then run MCTS for k iterations (lines 16–19). After k iterations, the action that leads to the state with the highest approximate return is chosen and performed on *nextNode* (line 10). If no unpartitioned node is found at *currentLevel*, we decrease *currentLevel* by 1 (line 11-12). The algorithm returns the fully constructed R-tree when all bottom-level nodes are partitioned.

Algorithm 1 PLATON with MCTS

Input: data $\mathcal{D}$, workload $\mathcal{W}$, node capacity B, # of iterations k
1: $T \leftarrow \{n_0\}$, $n_0.data = \mathcal{D}$, $n_0.level = \lceil \log_B N \rceil$
2: $canSplit \leftarrow True, currentLevel \leftarrow \lceil \log_B N \rceil, nextNode \leftarrow \phi$
3: **while** canSplit **do**
4: $canSplit \leftarrow False$
5: **for** each node $n \in T$ at level $currentLevel$ **do**
6: **if** $|n.data| > B^{currentLevel}$ **then**
7: $canSplit \leftarrow True$, $nextNode \leftarrow n$
8: **if** canSplit **then**
9: a = MCTS(T, k)
10: perform partition a on $nextNode$
11: **else if** $currentLevel > 1$ **then**
12: $currentLevel \leftarrow currentLevel - 1$, $canSplit \leftarrow True$
13: **return** T
14: **procedure** MCTS(T, k)
15: Initialize search tree root v_0 with state T
16: **for** $i = 1$ to k **do**
17: $v_l \leftarrow TreePolicy(v_0)$
18: $\Delta \leftarrow RolloutPolicy(v_l)$
19: $BackUp(v_l, \Delta)$
20: **return** $a(BestChild(v_0))$

5.4 MCTS Policy Designs

Challenges. In order for MCTS to return good actions, the exploration strategy and simulation strategy, *i.e.*, selection policy and rollout policy, must be carefully designed. There are unique challenges in designing the policies for our problem step. For the selection policy, as the return of a state across different rollouts usually has a high variance, the standard *Upper Confidence Bound for Trees* (UCT) algorithm no longer works well. For the rollout out policy, due to a long action sequence and a huge state space, random rollouts lead to slow convergence. We now discuss in detail the design choices we make for the selection policy and rollout policy to address the challenges.

Selection Policy. At the selection stage, the algorithm follows a $TreePolicy$ to find the most promising search tree node to expand. The key challenge in designing the selection policy is the exploitation-exploration dilemma, *i.e.*, how to balance the choice between states of high approximated returns and unexplored states. A popular choice is the *Upper Confidence Bound for Trees* (UCT) algorithm. When selecting the search tree nodes, UCT considers both the average return from previous rollouts, which prioritize nodes with high returns, and an upper confidence bound term, which prioritize less frequently visited child states.

While the UCT policy works for many problems, using the average return in node selection does not work well for our problem setup. This is because the return of a state has a high variance across different rollouts, and a good partition choice followed by some bad partition choices can lead to an extremely low return. If the average return is used for node selection, rollouts with high returns may be averaged out by a few rollouts with low returns, making it hard to identify good partition actions. To address this issue, we propose a new selection policy.

$$v \leftarrow \underset{v' \in children\,of\,v}{\arg\max}\; Q_{max}(v') + c\sqrt{\frac{2\ln N(v)}{N(v')}} \tag{3}$$

The first term is the maximal return from previous rollouts, and the second term is a confidence bound that makes the selection to prioritize less frequently visited child states. With the new design, the selection policy considers the most successful previous experience instead of the average experience, which prevents promising states from being overlooked during selection. We present our $TreePolicy$ in Algorithm 2 .

Algorithm 2 $TreePolicy(v)$

```
1: while v is non-terminal do
2:     if v not fully expanded then
3:         randomly choose untried action a ∈ A(s(v))
4:         add a new child v' to v
5:         return v'
6:     else
7:         v ← arg max_{v' ∈ children of v} Q_max(v') + c·sqrt(2 ln N(v) / N(v'))
```

Subsequently, as our $TreePolicy$ makes the selection based on the maximum return from previous iterations, we record the maximum Q-value for each state during Backpropagation.

Rollout Policy. The standard choice of *RolloutPolicy* in MCTS is selecting a random action from the action space. However, due to a long action sequence and a huge state space, random rollouts lead to slow convergence. To address the problem, we propose to incorporate domain knowledge of R-tree packing into the simulation using a greedy rollout policy. The greedy rollout policy chooses the action that leads to the highest immediate reward, *i.e.*, the partition with the most page skipping. The greedy rollout policy makes better partitions than random rollouts, and the return obtained

from the rollouts more closely approximates the optimal return. Thus, we need fewer iterations for MCTS to converge and find a good solution. We present our *RolloutPolicy* in Algorithm 3.

Algorithm 3 $RolloutPolicy(v)$

```
1: Δ = CumulativeReward(v)
2: s = s(v)
3: while s is non-terminal do
4:     a_greedy = arg max_{a∈A} Reward(s, a)
5:     s ← f(s, a_greedy)
6:     Δ ← Δ + reward(s, a_greedy)
7: return Δ
```

Complexity. The complexity of PLATON (Algorithm 1) consists of two parts, the complexity of running the MCTS algorithm (line 15-20) at each partitioning step, as well as the complexity of the top-down construction of the index on disk given the learned partitions. The running time of the top-down construction of the tree is $O((N/B)\log_2 N)$[12]. We next focus on the time complexity of the MCTS algorithm. We first consider the complexity of a single MCTS iteration. The complexity of the selection and expansion step is $O(k)$ (line 17). For the simulation step (line 18), we need to construct the entire R-tree using the greedy rollout policy. Computing and performing the greedy partition action for a single partially constructed tree node n incurs a complexity of $O(B \cdot |n.data|)$. As we construct the entire tree, we add up this complexity for all nodes. We first add up the complexity for all nodes at the same level, which gives us a complexity of $O(B \cdot N)$ because the sum of the node sizes at the same level is $O(N)$. We then add up the complexity for all $\log_B N$ levels of partially constructed R-tree nodes, which leads to a time complexity of $O(B \cdot N \log N)$ for the simulation step. The complexity of the backpropagation step is $O(k)$ (line 19). As a result, the complexity of an MCTS iteration is dominated by that of the simulation step and is $O(N \log N)$. We then multiply the per-iteration complexity by the number of iterations k and the total number of actions N/B, which gives a time complexity of $O(k \cdot N^2 \log N)$ for the MCTS algorithm.

To summarize, PLATON's complexity is the sum of the complexity of the top-down construction of the index, $O((N/B)\log_2 N)$, and the complexity of the MCTS algorithm, $O(k \cdot N^2 \log N)$. We note that in practice, the total running time is dominated by the running time of the MCTS algorithm.

6 OPTIMIZATION OF MCTS

The MCTS algorithm has a running time that dominates the total running time. In this section, we propose to reduce the running time of MCTS with a divide and conquer strategy. Together with two optimization techniques, early termination and level-wise sampling, we are able to drastically reduce the time complexity of the MCTS algorithm to linear time.

The main reason of the high complexity is the long action sequence. For example, for a three-level R-tree with node size $B = 100$, there is a total of 9999 actions. To make the first partition decision, the MCTS algorithm needs to consider all 9998 follow-up actions, which results in the long running time of the rollouts. We first make the observation that the optimal partition decision on the two child nodes of a partially constructed tree node does not depend on each other. Formally speaking, if we denote the optimal total return from partitioning partially constructed tree node n and its child nodes as $V^*(n)$, we have the following:

$$V^*(n) = \max_{a \in A}(Reward(n, a) + V^*(n_1) + V^*(n_2)), \quad (4)$$

where n_1 and n_2 are the two partially constructed tree nodes created from a partition action a on node n. The equation above identifies the optimal substructure in our problem: in order to find the

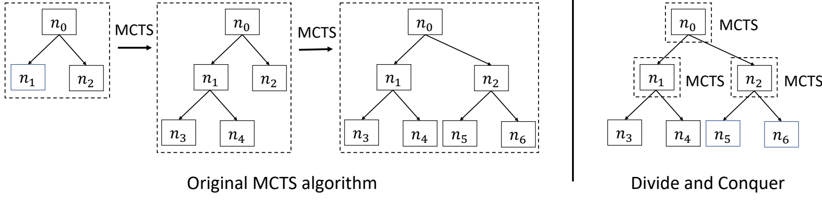

Fig. 7. Illustration of divide and conquer.

sequence of actions that maximizes the total return from a partially constructed tree node n, we only need to independently solve the subproblems of finding the sequence of actions that maximizes the total return from the two child nodes n_1 and n_2. In other words, to construct the subtree rooted at n with the maximal return, we break it down into two subproblems of constructing the subtrees rooted at n_1 and n_2 with the maximal return respectively.

With the optimal substructure identified, we can leverage a divide and conquer strategy to reduce the algorithm complexity. For the partition decision on a node n, instead of running MCTS on the original state space, we consider a new state space consisting of smaller trees, *i.e.*, the subtrees rooted at n. The new state space is equivalent to the state space we have for constructing an R-tree from $n.data$. To run MCTS on this new state space, we create a new partially constructed tree with a single root node n, initialize a search tree with this partially constructed tree state, and perform the MCTS iterations to find a good action, as if we are constructing a new R-tree from $n.data$. Figure 7 illustrates how divide and conquer helps make the MCTS algorithm run faster. The dashed boxes denote the states that form the search tree root of MCTS. Previously, we always run MCTS on the original partially constructed tree state space to find the partition actions for nodes n_1 and n_2. As a result, to find the partition action for n_1, MCTS also considers the partition actions on n_2 and its child nodes at the simulation step, leading to slow rollouts. With divide and conquer, to find the partition action on n_1/n_2, we only need to run MCTS algorithm on the new state space of subtrees rooted at n_1/n_2, and the action sequence is much shorter. This greatly accelerates the rollouts.

Algorithm 4 shows the pseudocode of MCTS with the divide and conquer strategy. We maintain a queue of unpartitioned nodes (line 2). We repeatedly pop nodes from the queue and perform partitions as long as the queue is not empty (line 3). Given a node $currentNode$, instead of running MCTS on the original state space, we run MCTS on the state space of subtrees rooted at $currentNode$ to get the partition action a (line 5). After obtaining the two newly created nodes n_1 and n_2 from the partition (line 6), we add them to the queue as long as they are not leaf nodes (line 7).

Early termination. With the divide and conquer strategy, to carry out a rollout, we still need to construct the entire subtree using the greedy policy, which involves a long action sequence if the partition is on a high-level node. To further speed up the rollouts, we propose early termination: instead of using the greedy policy to construct the entire subtree, $RolloutPolicy$ will terminate as long as all the next-level partially constructed tree nodes are created. For example, if we are finding the optimal partition action on a partially constructed tree node n at level t with size B^t, a rollout will terminate as long as all B child nodes at level $t-1$ are created. This further reduces the number of steps to run for each rollout.

Level-wise sampling. To accelerate the training of RL, we learn on a sample set of data. Intuitively, one can learn the R-tree partition policy on a data sample, where the size of the partially constructed tree nodes scales by the sample rate. However, the inherent structure of the R-tree makes it hard to directly learn from a data sample. Consider building an R-tree on a data set of size 1000,000, with node capacity $B = 100$. If we perform top-down packing on a sample of 5% of the data, then each leaf node on average contains $100 \times 5\% = 5$ data object. As a result, the leaf node MBRs are no

Algorithm 4 PLATON with Divide and Conquer

Input: dataset $\mathcal{D}$, workload $\mathcal{W}$, node capacity B, # of iterations k
Output: packed R-tree T

```
1: T ← {n_0}, n_0.data = D, n_0.level = ⌈log_B N⌉
2: Queue = {n_0}
3: while !Queue.empty() do
4:     currentNode = Queue.pop()
5:     a = MCTS(currentNode, k)
6:     n_1, n_2 ← Partition(currentNode, a)
7:     add n_1 and n_2 to Queue if not leaf node
8: return T
9: procedure MCTS(n, k)
10:    n' = Sample(n, rate = s/B^{n.level});
11:    T_new = {n'}
12:    Initialize search tree root v_0 with state T_new
13:    for i = 1 to k do
14:        v_l ← TreePolicy(v_0)
15:        Δ ← RolloutPolicy(v_l);                  # Early termination
16:        BackUp(v_l, Δ)
17:    return a(BestChild(v_0))
```

longer representative of the original leaf node MBRs when the R-tree is built on the full dataset, and the rewards derived from the samples may deviate significantly from that of the original data.

While a small sample rate makes leaf node MBRs no longer representative, for higher-level nodes, it is possible to apply a small sample rate. Under the same example, with a sample rate of 5%, nodes at level 2 on average contain $100^2 \times 5\% = 500$ data objects, which can still form node MBRs that are sufficiently representative of the original node MBRs. By taking into consideration the varying node sizes at different levels, we develop a level-wise sampling technique that applies a different sampling rate to partially constructed tree nodes at different levels when running the MCTS algorithm. The core idea of level-wise sampling is that after applying data sampling, bottom-level tree nodes at the terminal states have a fixed size s, which ensures that the result node MBRs are still representative of the original node MBRs. Level-wise sampling applies a sample rate of s/B^l when running the MCTS algorithm to partition a partially constructed R-tree node into child nodes at level l. Level-wise sampling makes the MCTS algorithm more efficient using samples, without compromising the accurate calculation of rewards.

Complexity. We now analyze the time complexity of the MCTS algorithm with the proposed optimization (Algorithm 4). Similar to the analysis in Section 5, we compute the complexity of the simulation step in an MCTS iteration (line 15). With early termination, we compute and perform the greedy partition actions for $O(B)$ steps. At each step, the size of a node is $O(B \cdot s)$ after level-wise sampling (line 10) , and therefore it takes $O(B^2 \cdot s)$ to compute and perform the greedy action for the sampled node. We multiply the two terms and obtain a complexity of $O(B^3 \cdot s)$ for the simulation step. The complexity of an MCTS iteration is therefore also $O(B^3 \cdot s)$. We then multiply the per-iteration complexity by the number of iterations k and the total number of actions N/B, which gives us a time complexity of $O(k \cdot B^2 \cdot s \cdot N)$. As discussed earlier, s is a constant in level-wise sampling, and therefore the complexity of the MCTS algorithm is effectively $O(k \cdot N)$. Note that the top-down construction of the tree in PLATON still has a complexity of $O((N/B) \log_2 N)$. However,

at the data scale this work concerns, the total running time is always dominated by that of the MCTS algorithm. Thus reducing the complexity of the MCTS algorithm to linear with regard to the input size N makes PLATON scalable to large datasets.

7 PACKING WITHOUT WORKLOAD

When a workload is not available at the time of index construction, we propose to learn a partition policy that optimizes the I/O cost of a workload that follows the data distribution. We first provide the intuition on why this can lead to a well-packed R-tree.

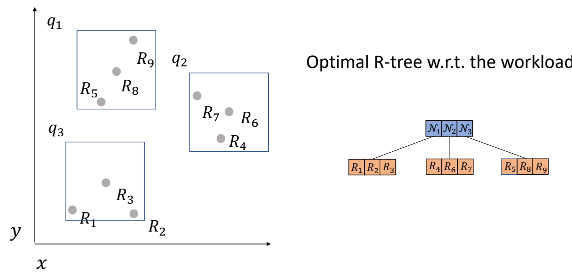

Fig. 8. Motivating example of optimizing for a synthetic query workload following the data distribution.

Intuition. Consider the example in Figure 8, where we need to pack 9 data points into a 2-level R-tree with node capacity $B = 3$. The data points are distributed over three clusters. We have a query workload $\mathcal{W} = \{q_1, q_2, q_3\}$ containing three window queries, and the centers of the query windows follow the data distribution. The R-tree that minimizes the I/O cost of the workload $\mathcal{W}$ is shown on the right of Figure 8, which leads to an I/O cost of 2 for each query. Note that this R-tree essentially packs the data objects in the same cluster into the same leaf node, which is the ideal R-tree given no specific assumption on the workload pattern.

The reason why our proposed objective works is as follows. An R-tree that minimizes the I/O cost of a query workload would pack data objects overlapping the same query window into the same leaf node or subtree. As we sample the queries from the data distribution, the constructed R-tree packs each data object into the same leaf node or subtree as its neighbors, achieving a good clustering property. Moreover, the generated workload covers the entire data space, therefore the constructed R-tree would perform well for queries accessing different regions of the data space.

Another problem is how to set the height and width of the query windows. A naive approach is to uniformly sample window height and width from a range to form query windows of different sizes. For example, to generate query window sizes between 0.0001% and 1% of the unit square, we can simply sample from the range $[0.001, 0.1]$. However, uniform sampling results in an unbalanced portion of large query windows. In fact, only 10% of samples are from $[0.001, 0.01]$, while 90% of samples are from $[0.01, 0.1]$. To remedy this, we sample the height and width from a log-uniform distribution, *i.e.*, the log of the sampled values are uniformly distributed. As a result, there will be as many samples from $[0.001, 0.01]$ as samples from $[0.01, 0.1]$, creating a much more balanced ratio between small query windows and large query windows.

8 EXPERIMENTS

8.1 Experimental Settings

Datasets. We use three synthetic datasets and three real-world datasets for our experiments. Each synthetic dataset contains 10 million data objects generated within a unit square. We generate synthetic datasets following three types of data distributions: Uniform, Skew, Gaussian.

- Uniform (UNI): rectangle data whose centers are uniformly distributed over the unit square. The height and width of the rectangles are uniformly sampled from $[0, 0.001]$.
- Skew: point data with a distribution that is highly skewed over one dimension. Following previous work on R-tree packing [12, 48], we first generate uniformly distributed points over the unit square, and then squeeze the y-dimension by raising y to y^9.
- Gaussian (GAU): rectangle data following Gaussian distribution. We generate the center of the rectangles by sampling each dimension's value from $\mathcal{N}(0, 1)$. We then normalize sampled values to ensure the centers fall within the unit square. The height and width of the rectangles are uniformly sampled from $[0, 0.001]$.

The real-world datasets are from US Census Bureau TIGER/Line Shapefiles [6] and OpenStreetMap [4]. A summary of the real-world datasets is in Table 1. The Area Water dataset and the OSM Parks dataset are pre-processed and published by SpatialHadoop [22]. We use four rectangle datasets and two point datasets.

Name	Type	Cardinality	Description
Area Water(AW)	Polygon	2.3M	US Area Hydrography
OSM Parks(PARK)	Polygon	10M	Parks around the world
OSM India(IND)	Point	100M	Landmarks in India

Table 1. Summary of real-world datasets.

Workloads. We generate workloads with different patterns to train PLATON and to evaluate different methods. For each workload pattern, we generate a training workload and a test workload, both of which contain 10000 queries. For the synthetic datasets, we consider two aspects of the workload patterns:

- *aspect ratio*: We generate 4 workloads with aspect ratio of 10, 100, 1000, and 10000. The queries follow the uniform distribution and have a fixed size of 0.001%.
- *size*: We generate 5 workloads with query size of 0.001%, 0.005%, 0.01%, 0.05% and 0.1% of the entire data space. The generated queries follow the uniform distribution and have random shapes.

For real-world datasets, we generate workloads with different patterns using the concept of *decimal degree* [1]. Decimal degree is an alternative unit for geographic coordinates, and different decimal degree scales have different interpretations in the physical world. For each real-world dataset, we generate four workloads with decimal degree scales of 0.001, 0.01, 0.1, and 1, which correspond to street level, town level, city level, and country level in the real world. The centers of the query windows are sampled from the data distribution. One advantage of generating workload in this way is that the queries have actual meaning. For example, the generated workloads contain queries that answer "How many rivers cross XXX town of US", or "Find all parks in XXX city".

Baseline Methods. We compare PLATON with two classes of methods: R-tree variants and learned/workload-aware spatial indexes. The R-tree variants include R-tree constructed from repeated insertion (R*-tree), and four R-tree packing methods: STR [39], TGS [25], PR-tree [12], and a rank space SFC-based method HRR [48]. The four methods are the best-performing methods reported in previous works. The four learned/workload-aware indexes[1] are learned index RSMI [47] and LISA [40], ML-enhanced index RLR-tree [26], and workload-aware index Waffle [44]. RLR-tree constructs an R-tree through repeated insertion instead of packing. Note that RSMI, LISA, and

[1]We do not include RW-tree in the experiments as the source code is not available after communication with the authors. We note that RW-tree reported up to 1.24× speedup over R*-tree, while PLATON achieves up to 3.17× speedup.

Waffle only support point data. We find that the released code of RSMI only works on the OSM India dataset with small data size, *i.e.*, 10 million. Therefore, we exclude RSMI from the experiments on other datasets and scalability. We use PLATON-D to denote the learned partition policy from the data distribution.

Implementation. We implement PLATON on top of two real-world systems: libspatialindex [2], a popular open source library of spatial indexes, as well as PostgreSQL 14.3 [9], with the PostGIS [8] extension for spatial index. Our implementations use C++ and Python. We also implement all R-tree variants on top of both libspatialindex and PostgreSQL. For learned/workload-aware indexes, we use the code released by the authors [3, 5, 7]. We aligned the implementation of PLATON and the baseline methods and used the same encoding to store the records. Specifically, for each record, we use 4 double (8 bytes each) to store the MBR coordinate values, and 4 bytes to store the address, which leads to a total of $4 \times 8 + 4 = 36$ bytes per record. All experiments are conducted on a Ubuntu server with a 20-core E5-2698 v4 @ 2.20GHz CPU.

Metric. We measure the performance of different methods based on both query I/O cost and query latency.

I/O Cost. For PLATON and R-tree variants, we obtain the I/O cost from the libspatialindex implementation. We include the I/O cost of the intermediate nodes when comparing PLATON with the R-tree variants. When comparing PLATON with learned/workload-aware indexes, we assume the intermediate nodes of the tree-based indexes, *i.e.*, PLATON, RLR and Waffle, are cached in the memory, because disk-based learned indexes, *i.e.*, RSMI and LISA, assume their models are cached in the memory. We normalize the I/O cost by an I/O lower bound, which is calculated by assuming the result data objects of each query are packed in a minimal subtree. The lower bound includes I/Os of the intermediate nodes if the I/O cost being compared includes intermediate nodes, and vice versa.

Query Latency. We obtain the query latency from PostgreSQL. We normalize the results by PLATON's query latency.

Parameters. We set the page size to 4KB by default. With 4KB pages, the libspatialindex implementation of PLATON and the R-tree variants, as well as all learned/workload-aware methods, have a node capacity B of $\lfloor \frac{4096}{36} \rfloor = 113$. For the PostgreSQL implementation of PLATON and the R-tree variants, as PostgreSQL reserves 44 bytes for metadata per page, the node capacity is given by $\lfloor \frac{4096-44}{36} \rfloor = 112$. We set the level-wise sampling parameter $s = B$.

8.2 Comparison with R-tree variants

We evaluate the performance of PLATON in comparison with existing R-tree variants for different types of workloads. Below we report several major findings from the results.

(F1) PLATON significantly outperforms all existing R-tree variants in terms of both query I/O and query latency across different workload patterns. Figure 9 shows the overall result of PLATON and existing R-tree variants, in terms of normalized I/O and normalized query latency, respectively. We observe that PLATON significantly outperforms all existing R-tree variants in terms of both query I/O and query latency. The performance improvement is a result of PLATON 's ability to adapt to both the data distributions and workload patterns. Compared to the best-performing bottom-up and top-down methods, *i.e.*, STR and TGS, PLATON achieves a speedup of up to 2.70× and 3.80× respectively in terms of query I/O. The query latency result in general aligns with the query I/O result, with PLATON achieving a speedup of up to 2.03× and 2.32× compared to STR and TGS. We notice that the performance gaps are slightly reduced in terms of query latency due to the existence of the database buffer during query execution. For the remaining experiment results, we follow previous work and only report query I/O. Figures 10 - 12 show the performance of

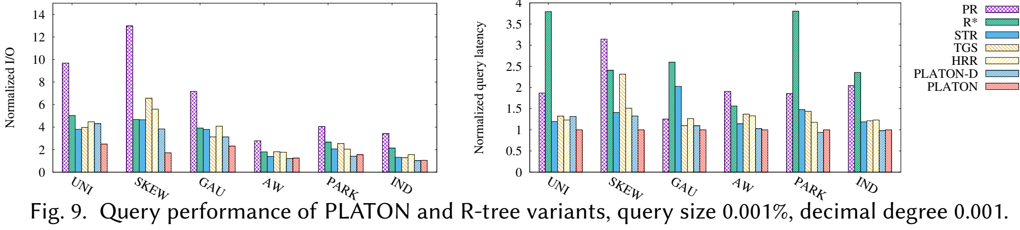

Fig. 9. Query performance of PLATON and R-tree variants, query size 0.001%, decimal degree 0.001.

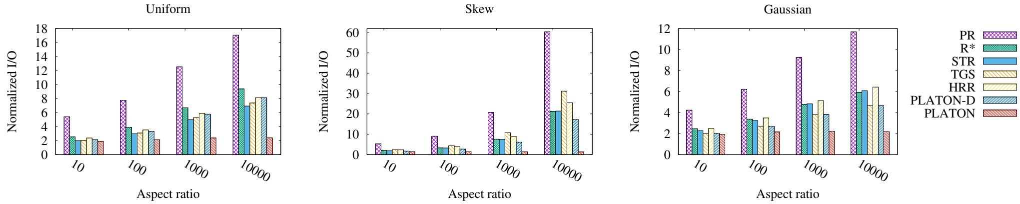

Fig. 10. Query I/O of PLATON and R-tree variants on workloads with varying query aspect ratios.

PLATON and existing R-tree variants on workloads with varying aspect ratios, sizes, and decimal degree scales respectively. We now discuss the observations for each type of workload pattern.
(1) *Window query with varying aspect ratios.* PLATON achieves the biggest performance improvement on workloads with a large aspect ratio. This shows the benefit of adapting to both the data distribution and workload distribution, especially when the workload has a special pattern. Compared to the best-performing baseline method, PLATON achieves a speedup of up to 2.90×, 16.31×, and 2.15× on the Uniform, Skew, and Gaussian dataset respectively.
(2) *Window query with varying sizes.* PLATON achieves the biggest performance improvement on workloads with a small query size. Compared to the best-performing baseline method, PLATON achieves a speedup of up to 1.52×, 2.70×, and 1.35× on the Uniform, Skew, and Gaussian dataset respectively.
(3) *Window query with varying decimal degree scales on real-world dataset.* We note that among the existing R-tree variants, the best-performing method is different for each dataset, which shows that no heuristic rule works well for all data distributions. Meanwhile, PLATON consistently outperforms all existing R-tree variants across the datasets. Compared to the best-performing baseline method, PLATON achieves a speedup of up to 1.18×, 1.31×, and 1.29× on the Area Water, OSM Parks, and OSM India dataset respectively.
(F2) If there is no workload available, by training from a synthetic workload following data distribution, PLATON still significantly outperforms all existing R-tree variants across different workload patterns. From Figure 9, we can see that compared to existing R-tree variants, PLATON-D achieves the best performance in 4 out of 6 datasets and top-2 performance in 5 out of 6 datasets in terms of query I/O. From Figure 10 and Figure 11, we observe that PLATON-D outperforms all existing R-tree variants on the Skew dataset and Gaussian dataset for different workload patterns. The performance of PLATON-D is most noticeable on real workload datasets. Figure 12 shows that PLATON-D achieves similar performance as PLATON across all three real-world datasets, even outperforming PLATON for smaller decimal degree scales. This demonstrates the effectiveness of optimizing for a synthetic query workload following data distribution.
(F3) PLATON significantly outperforms all existing R-tree variants for KNN queries. Figure 13 shows the performance of PLATON and R-tree variants for K-Nearest-Neighbor (KNN) queries. We sample the query points from the data distribution and conduct experiments for $K = 1, 5, 25, 125, 625$. We observe that although PLATON is optimized for window query performance, it outperforms all existing R-tree variants for KNN queries.

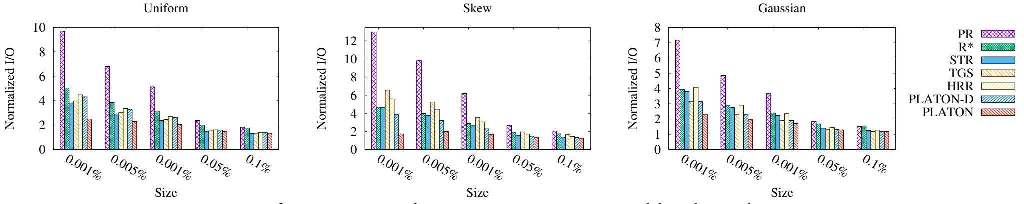

Fig. 11. Query I/O of PLATON and R-tree variants on workloads with varying query sizes.

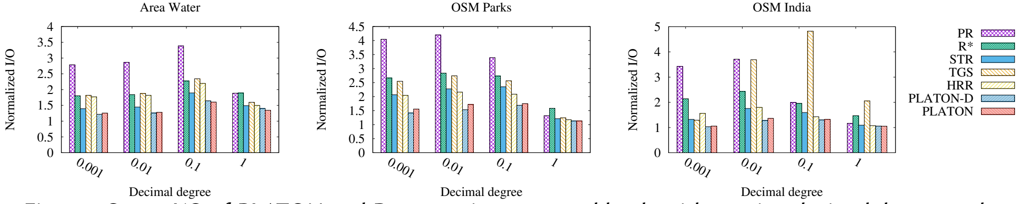

Fig. 12. Query I/O of PLATON and R-tree variants on workloads with varying decimal degree scales.

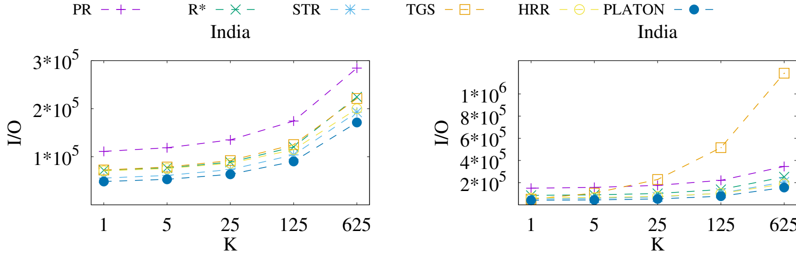

Fig. 13. Query I/O of PLATON and R-tree variants on KNN queries.

(F4) PLATON significantly outperforms all existing R-tree packing methods for spatial join queries. Table 2 shows the performance of PLATON and R-tree variants for spatial join queries. We perform spatial join over the OSM Parks and Area Water dataset to find all pairs of (park, water area) that intersects with each other. We implement two classic spatial join algorithms [32], index-nested loop join [23] and hierarchical traversal [28]. For the index-nested loop join, we only build an R-tree index on the OSM Parks dataset and loops over the Area Water dataset to find the parks intersecting each water area. For the hierarchical traversal method, we build R-tree index on both datasets. From the results, we observe that with Index-nested loop join, PLATON outperforms all existing R-tree variants; with hierarchical traversal, PLATON underperforms R*-tree but outperforms all existing R-tree packing methods. One interesting finding is that R*-tree outperforms all existing R-tree packing methods on spatial join.

Method	R*	PR	STR	TGS	HRR	PLATON
Index-nested loop ($\cdot 10^7$)	1.24	2.75	1.49	1.37	1.31	1.00
Hierarchical traversal ($\cdot 10^5$)	1.42	3.84	2.14	2.24	2.40	1.73

Table 2. Query I/O of PLATON and R-tree variants on spatial join queries.

8.3 Comparison with Learned/Workload-aware Spatial Indexes

Figure 14 shows the performance of PLATON in comparison with learned/workload-aware indexes on the two point datasets. Note that RSMI, LISA, and Waffle only support point data. We only

evaluate RSMI on the OSM India dataset as the released code does not work on the synthetic skew dataset. We observe that PLATON outperforms all baseline methods across all query sizes on the two datasets, with a speedup of up to 2.30× and 1.42× compared to the best performing baseline Waffle.

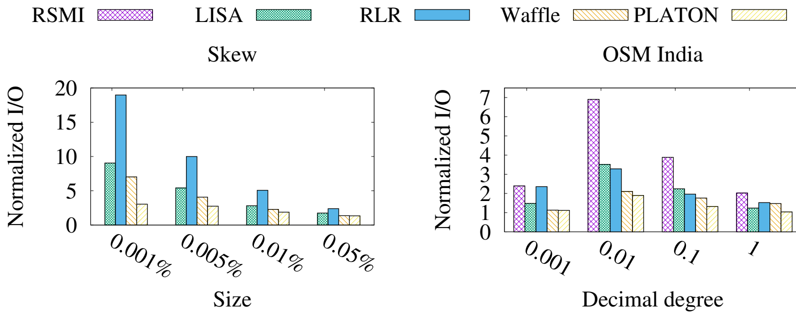

Fig. 14. Query I/O of PLATON in comparison with learned/workload-aware indexes.

8.4 Does MCTS optimizes the long-term reward?

We investigate whether MCTS optimizes the long-term reward through comparison with a greedy partition policy that maximizes the immediate reward at each step. Table 3 shows PLATON's speedup over the greedy partition policy on different datasets. We observe the PLATON significantly outperforms the greedy partition policy across all datasets, with the most speedup observed on large datasets. This shows the effectiveness of MCTS in avoiding locally optimal decisions and optimizing the long-term reward.

Dataset	UNI	SKEW	GAU	AW	PARK	IND
Speedup	1.04	1.13	1.22	1.07	1.17	1.69

Table 3. PLATON's speedup over the greedy partition policy.

8.5 Effect of Optimization Techniques

We conduct experiments to evaluate the effect of the proposed optimization techniques. The left plot in Figure 15 shows the log-log plot of the training time of PLATON with data sizes ranging from 100K to 100 million on the OSM India dataset. From the figure, we observe that the training time of PLATON w/o optimization scales super-linearly with regard to the dataset size. With the proposed optimization, the training time is drastically reduced, and becomes linear with regard to the data size.

8.6 Scalability

We conduct experiments to evaluate whether PLATON is scalable to larger datasets. We generate four data samples with sizes ranging from 25 million to 100 million from the OSM India dataset.

The right plot in Figure 15 shows the overall construction time (including training time) of PLATON in comparison with the learned index LISA. We observe that the construction of PLATON is much faster than LISA for large data sizes, and is only slower than LISA for 25 million records. The construction time of PLATON scales linearly with regard to the data size, which is consistent with our analysis in Section 6. We note that existing R-tree variants and workload-aware indexes like

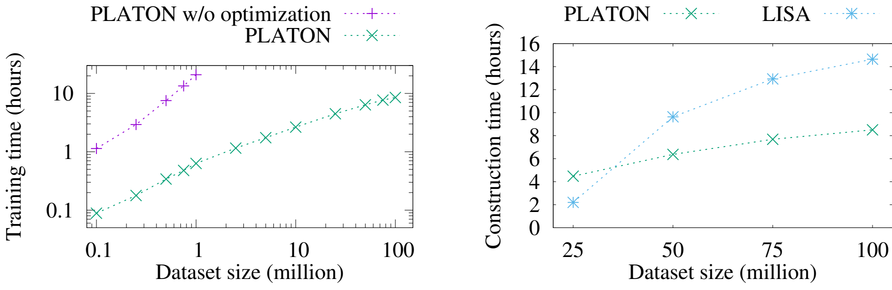

Fig. 15. Scalability results.

Waffle have a shorter construction time, *i.e.*, several minutes, on such data scales as they do not require training. As the bulk-loading of an index is typically performed in an offline manner, the training overhead of PLATON is acceptable.

We also conduct experiments to evaluate whether PLATON keeps the performance gain on larger datasets. Figure 16 shows the query I/O of PLATON and baseline methods with varying data sizes. From the left chart, we observe that PLATON consistently outperforms all R-tree variants, as well as all learned/workload-aware indexes as the data size increases.

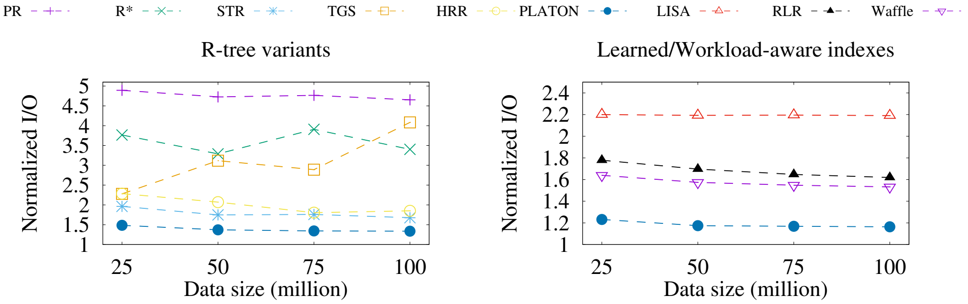

Fig. 16. Query I/O of PLATON and baseline methods with varying data sizes.

8.7 Effect of pages size

Figure 17 shows the performance of PLATON and baseline methods with varying page sizes. We conduct experiments on the OSM India dataset for page size of 4KB, 8KB, 12KB, and 16KB, which corresponds to node capacity of 113, 227, 341, and 455 respectively. From the results, we observe that PLATON consistently outperforms existing R-tree variants and learned/workload-aware indexes as the page size varies.

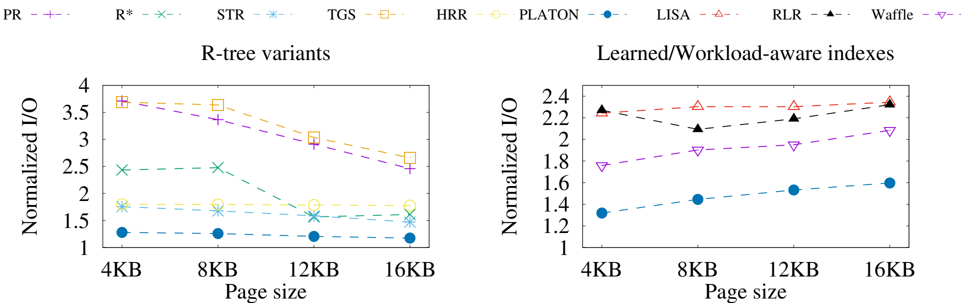

Fig. 17. Query I/O of PLATON and baseline methods with varying page sizes.

8.8 Effect of dimensionality

We conduct experiments to evaluate the applicability and scalability of PLATON in higher dimensions. We generate hyper-rectangle datasets of size 10 million in up to 6-dimensional space. The rectangle centers follow uniform distribution, and the edge length in each dimension is uniformly sampled from (0, 0.001]. The query windows have a fixed size of 0.001% of the data space and have random shapes. The left plot in Figure 18 shows the query I/O of PLATON and the R-tree variants as the dimension increases. We observe that while the performance of existing R-tree variants drastically degrades as the number of dimensions increases, PLATON remains highly efficient in higher dimensions. The right plot in the figure shows the training time of PLATON as the dimension varies. We observe that the training time of PLATON is linear with regard to the number of dimensions.

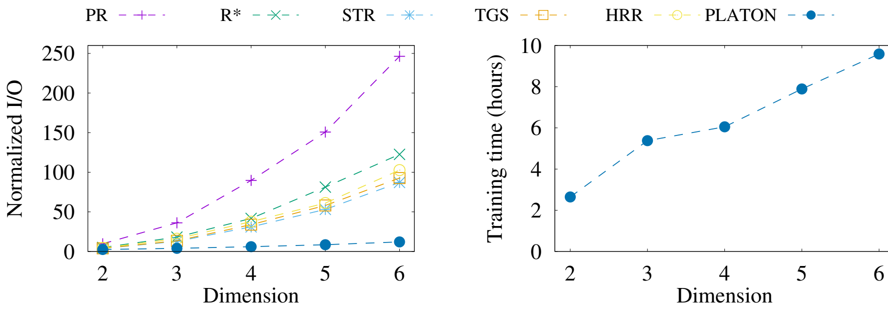

Fig. 18. Effect of dimensionality

8.9 Workload shifts

Figure 19 shows the performance of PLATON when the test workload shifts from the training workload in terms of workload characteristics. We evaluate two workload shift scenarios: shift in query aspect ratio and shift in query size. In both scenarios, we train PLATON on a default workload, *i.e.*, query aspect ratio of 10 and decimal degree scale of 0.1 respectively, and evaluate its performance on workloads with a range of aspect ratios and sizes. We denote PLATON trained on the default workload as PLATON-S. We compare the performance of PLATON under workload shifts with the best-performing baseline methods, the workload-aware index Waffle, as well as PLATON-D. From the left plot, we observe that PLATON trained on the default workload consistently outperform the baseline methods as the query aspect ratio changes, while PLATON-D underperforms Waffle. From the right plot, we observe that PLATON-D achieves the best performance for all query sizes. PLATON trained on the default workload slightly underperforms PLATON-D, but still outperforms Waffle. This shows that PLATON is robust to workload shifts.

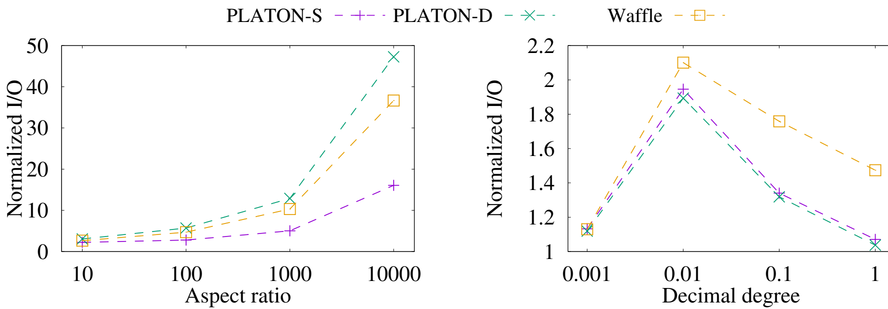

Fig. 19. Performance of PLATON under workload shifts

9 RELATED WORK

R-Tree Variants. R-tree [29] was first proposed by Antonin Guttman in 1984 and has since been extensively studied by the research community. R^+-tree [51], R^*-tree [13], and RR^*-tree [14] were proposed to improve the insertion and deletion operation and produce a better tree structure. Meanwhile, a number of works have studied the problem of R-tree packing [12, 15, 17, 25, 30, 31, 33, 38, 39, 48, 49], which constructs an R-tree with better space utilization and query performance. A detailed discussion of existing R-tree packing methods can be found in Section 2. Waffle [44] combines the concepts of data partitioning from R-tree with space partitioning and proposes an index structure with square-like, non-overlapping nodes.

Machine Learning for Spatial Index. The initial idea of learned index [37] was to replace 1-d index structures like B-tree with a machine learning model that learns the cumulative density function (CDF) of the data. The model then maps the search key to the storage id based on the learned CDF. Several variants [19, 24, 34, 41] have been proposed to achieve better efficiency and robustness. Learned spatial indexes extended the idea to multi-dimensional spatial data. ZM index [53] was proposed for 2-d spatial data points. It linearizes the spatial points using a space-filling curve, *e.g.*, Z-curve, and learns to model the CDF of the z-order values. RSMI [47] improved over ZM index by introducing a rank space-based ordering and a recursive partitioning strategy. Instead of relying on space-filling curves, LISA [40] was proposed to directly learn a mapping from spatial data points to 1-d value. Flood [45] proposed a grid-based index for multi-dimensional data that adapts to a particular dataset and workload. Tsunami [20] addressed the performance issue of Flood in case of correlated data and skewed workload. Note that Flood and Tsunami mainly focus on in-memory indexes, and they do not support spatial queries like KNN queries. Therefore, we do not compare Flood and Tsunami with the spatial indexes.

More recently, several methods were proposed to enhance traditional spatial indexes like R-tree with machine learning techniques. RLR-Tree [27] and RW-tree [21] were proposed to improve R-tree construction through one-by-one insertion, and they learn to optimize the *chooseSubtree* and *splitNode* process. "AI+R"-tree [10] was proposed to enhance the range query processing algorithm of R-tree for a given data and workload instance, by leveraging machine learning techniques.

Instance-optimized Databases. The concept of Instance-optimized Databases [36] refers to data systems that self-adjust to a given workload and data distribution to provide unprecedented performance. A number of instance-optimized data components have been proposed, including instance-optimized index [10, 20, 45], data layouts [18, 54], query optimizers [42, 43] and query scheduler [50].

10 CONCLUSION

This work considers the problem of R-tree packing, such that the query performance is optimized for a given data and workload instance. We propose PLATON, a top-down packing method that leverages Monte Carlo Tree Search to learn an optimal partition policy. We propose a divide and conquer strategy, and two optimization techniques to reduce the MCTS algorithm complexity and derive a linear-time algorithm. Extensive experiments on both synthetic and real-world datasets show that PLATON outperforms both existing R-tree variants and learned/workload-aware indexes.

ACKNOWLEDGMENTS

This research is supported in part by MOE Tier-2 grant MOE-T2EP20221-0015. We would like to acknowledge Alibaba-NTU Joint Research Institute, Interdisciplinary Graduate Programme, Nanyang Technological University, Singapore.

REFERENCES

[1] [n. d.]. Decimal Degrees. https://en.wikipedia.org/wiki/Decimal_degrees.
[2] [n. d.]. libspatialindex 1.9.3. https://libspatialindex.org/en/latest/index.html.
[3] [n. d.]. LISA Implementation. https://github.com/pfl-cs/LISA.
[4] [n. d.]. OpenStreetMap. https://www.openstreetmap.org.
[5] [n. d.]. RSMI Implementation. https://github.com/Liuguanli/RSMI.
[6] [n. d.]. TIGER/Line Shapefiles. https://www.census.gov/geographies/mapping-files/time-series/geo/tiger-line-file.html.
[7] [n. d.]. Waffle Implementation. https://gitlab.com/moinmoti/waffle.
[8] 2023. PostGIS. https://postgis.net/.
[9] 2023. PostgreSQL. https://www.postgresql.org/.
[10] Abdullah-Al Abdullah-Al-Mamun, Ch. Md. Rakin Haider, Jianguo Wang, and Walid G. Aref. 2022. The "AI + R" - tree: An Instance-optimized R - tree. In *2022 23rd IEEE International Conference on Mobile Data Management (MDM)*. 9–18. https://doi.org/10.1109/MDM55031.2022.00023
[11] Daniar Achakeev, Bernhard Seeger, and Peter Widmayer. 2012. Sort-based query-adaptive loading of r-trees. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 2080–2084.
[12] 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.
[13] 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.
[14] 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.
[15] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. 1998. Improving the query performance of high-dimensional index structures by bulk load operations. In *International Conference on Extending Database Technology*. Springer, 216–230.
[16] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.
[17] David J DeWitt, Navin Kabra, Jun Luo, Jignesh M Patel, and Jie-Bing Yu. 1994. Client-server paradise. In *VLDB*, Vol. 94. Citeseer, 558–569.
[18] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-optimized data layouts for cloud analytics workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 418–431.
[19] 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.
[20] 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).
[21] Haowen Dong, Chengliang Chai, Yuyu Luo, Jiabin Liu, Jianhua Feng, and Chaoqun Zhan. 2022. RW-Tree: A Learned Workload-aware Framework for R-tree Construction. In *2022 IEEE 38th International Conference on Data Engineering*. IEEE.
[22] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st international conference on Data Engineering*. IEEE, 1352–1363.
[23] R Elmasri, Shamkant B Navathe, R Elmasri, and SB Navathe. 2000. Fundamentals of Database Systems</Title>.
[24] 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, 8 (2020), 1162–1175.
[25] 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.
[26] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2021. A Reinforcement Learning Based R-Tree for Spatial Data Indexing in Dynamic Environments. *arXiv preprint arXiv:2103.04541* (2021).
[27] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2023. The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[28] Oliver Gunther. 1993. Efficient computation of spatial joins. In *Proceedings of IEEE 9th International Conference on Data Engineering*. IEEE, 50–59.
[29] 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.

[30] M Hammar, HJ Haverkort, et al. 2002. Box-trees and R-trees with near-optimal query time. *Discrete & Computational Geometry* 28, 3 (2002), 291–312.

[31] Herman Haverkort and Freek V Walderveen. 2008. Four-dimensional Hilbert curves for R-trees. *Journal of Experimental Algorithmics (JEA)* 16 (2008), 3–1.

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

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

[34] 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.

[35] Evgenios M Kornaropoulos, Silei Ren, and Roberto Tamassia. 2020. The Price of Tailoring the Index to Your Data: Poisoning Attacks on Learned Index Structures. *arXiv preprint arXiv:2008.00297* (2020).

[36] Tim Kraska. 2021. Towards instance-optimized data systems. *Proceedings of the VLDB Endowment* 14, 12 (2021).

[37] 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.

[38] Taewon Lee and Sukho Lee. 2003. OMT: Overlap Minimizing Top-down Bulk Loading Algorithm for R-tree.. In *CAISE Short paper proceedings*, Vol. 74. 69–72.

[39] 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.

[40] 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.

[41] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A High-Performance Learned Index on Persistent Memory. *arXiv preprint arXiv:2105.00683* (2021).

[42] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*. 1275–1288.

[43] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. In *VLDB*.

[44] Moin Hussain Moti, Panagiotis Simatis, and Dimitris Papadias. 2022. Waffle: A Workload-Aware and Query-Sensitive Framework for Disk-Based Spatial Indexing. *Proceedings of the VLDB Endowment* 16, 4 (2022), 670–683.

[45] 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.

[46] Varun Pandey, Alexander van Renen, Andreas Kipf, Ibrahim Sabek, Jialin Ding, and Alfons Kemper. 2020. The case for learned spatial indexes. In *Proceedings of the 2nd International Workshop on Applied AI for Database Systems and Applications (AIDB'20)*.

[47] 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.

[48] 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.

[49] 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.

[50] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1228–1242. https://doi.org/10.1145/3514221.3526158

[51] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB '87)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 507–518.

[52] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[53] 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.

[54] 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.

Received 16 April 2023; revised 20 July 2023; accepted 24 August 2023