[Experiment, Analysis, and Benchmark] Benchmarking RL-Enhanced Spatial Indices Against Traditional, Advanced, and Learned Counterparts

Guanli Liu
The University of Melbourne
guanli.liu1@unimelb.edu.au

Renata Borovica-Gajic
The University of Melbourne
renata.borovica@unimelb.edu.au

Hai Lan
The University of Queensland
h.lan@uq.edu.au

Zhifeng Bao
The University of Queensland
zhifeng.bao@uq.edu.au

Abstract—Reinforcement learning (RL) has recently been used to enhance index structures, giving rise to RL-enhanced spatial indices (RLESIs) that aim to improve query efficiency during index construction. However, their practical benefits remain unclear due to the lack of unified implementations and comprehensive evaluations, especially in disk-based settings.

We present the first modular and extensible benchmark for RLESIs. Built on top of an existing spatial index library, our framework decouples index training from building, supports parameter tuning, and enables consistent comparison with traditional, advanced, and learned spatial indices (LSIs).

We evaluate 12 representative spatial indices across six datasets and diverse workloads, including point, range, kNN, spatial join, and mixed read/write queries. Using latency, I/O, and index statistics as metrics, we find that while RLESIs can reduce query latency with tuning, they consistently underperform LSIs and advanced variants in both query efficiency and index-building cost. These findings highlight that although RLESIs offer promising architectural compatibility, their high tuning costs and limited generalization hinder practical adoption.

Index Terms—Reinforcement Learning, Spatial Index, Learned Index, Benchmark, Parameter Tuning

I. Introduction

The explosive growth of location-based services and spatial analytics has led to a surge in spatial data, often far exceeding the capacity of main memory. To ensure scalable query processing, traditional disk-based spatial indices—such as R-trees [1] and their variants [2]–[4]—have long been used in production systems [5]–[7]. However, increasing dataset sizes, complex workloads, and data skew present growing challenges to these traditional structures, particularly in maintaining high query performance across diverse scenarios.

Recent work has explored *learning-based spatial indexing methods*, which leverage machine learning (ML) or reinforcement learning (RL) to enhance index construction and query efficiency [8]–[15]. These methods can be broadly categorized based on their integration of learning models: (1) **Learned Spatial Indices (LSIs)**, such as ZM-index [8], RSMI [9], Flood [10], and LISA [11], employ machine learning (ML) models to construct entirely new index structures that directly capture the spatial data distribution. These indices require custom query and update mechanisms. (2) **RL-Enhanced Spatial Indices (RLESIs)**, such as BM-tree [12], Qd-tree [13], and RLR-tree [14], retain traditional structures and use RL models to optimize decision-making during index building, such as determining partition boundaries.

While both LSIs and RLESIs aim to accelerate spatial queries, they differ fundamentally in design philosophy: LSIs propose entirely new structures, whereas RLESIs optimize within existing ones. This distinction gives RLESIs a practical advantage in terms of deployability and ease of integration into existing systems.

Despite their promise, RLESIs remain insufficiently studied along two key dimensions: their *external* performance relative to diverse and competitive baselines, and their *internal* sensitivity to parameter choices. Existing evaluations are often conducted in isolated settings with fixed parameters and narrow baselines [12]–[14], limiting generalizable insights.

This raises two key research questions: *(1) How do RLESIs compare to traditional, advanced, and learned spatial indices across diverse workloads? (2) How do configuration parameters, such as training sample size and tuning frequency, affect the query performance of RLESIs?* Answering these questions is non-trivial due to the lack of a unified evaluation framework for systematic and comprehensive comparison.

To address *Question 1*, we develop the first unified benchmarking framework for evaluating RLESIs alongside traditional, advanced, and learned spatial indices. Our framework supports end-to-end evaluation by decoupling index learning from index building, enabling reusable implementations and consistent comparisons across indexing methods.

Existing benchmarks [16]–[19] primarily focus on one-dimensional learned indices [20]–[26], which differ significantly from RLESIs in terms of dimensionality, query types, and indexing strategies. While a benchmark for learned spatial indices exists [27], it targets in-memory LSIs and is not suitable for disk-based RLESIs. Additionally, fragmented implementations, e.g., RLR-tree [14] using standalone R-trees and BM-tree [12] relying on external environments, complicate fair evaluation.

Our benchmarking framework includes representative indices from three major spatial indexing paradigms: data partitioning, space partitioning, and mapping. These span traditional, learned, and RL-enhanced variants. We prioritize open-source implementations to ensure reproducibility.

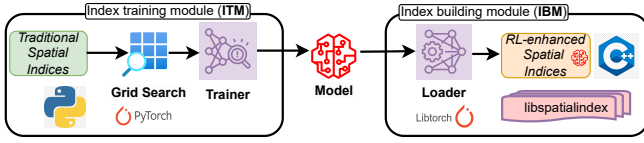As illustrated in Figure 1, the framework comprises two

Fig. 1: An overview of the benchmarking framework.

primary modules: the **Index Training Module (ITM)** and the **Index Building Module (IBM)**:

- **ITM** provides a standardized training environment for RLESIs through a *trainer* built on PyTorch [28], ensuring consistency in training, parameter tuning, and RL model generation. PyTorch is chosen for its compatibility with existing RLESI baselines, which are predominantly implemented in Python.

- **IBM** extends the disk-based indexing library *libspatialindex* [29] to support RLESIs, enabling the seamless integration into spatial systems. A core component of IBM is the *loader*, which leverages PyTorch's C++ API to load trained RL models produced by ITM, enabling integration with traditional disk-based indices.

To address *Question 2*, we incorporate a structured *grid search* mechanism within ITM for parameter tuning. This component systematically explores configuration spaces to minimize query costs while balancing index building overhead and query performance. Grid search is tailored for each dataset and workload, and constrained by a training cost threshold to ensure practical feasibility.

This paper makes the following key contributions:

① **A unified benchmark for comprehensive evaluation of RLESIs.** We present the first systematic evaluation of RL-enhanced spatial indices, comparing them with traditional, advanced, and learned baselines under diverse workloads. Our benchmark isolates learning effects from structural differences and fine-tuning, and quantifies RLESIs under diverse workloads. All components are released as open source[1] to facilitate reproducibility and future extensions.

② **A comprehensive and reusable evaluation setup for benchmarking spatial indices.** Our framework includes diverse datasets, workloads (point, range, and kNN queries), and metrics (latency at P50 and P99, I/O operations, index-level statistics) to reflect realistic spatial scenarios.

③ **Our extensive experiments yield several key observations (O): (1) Query Performance (O1-O5):** RLESIs provide modest gains over traditional indices but often underperform compared to advanced and learned baselines. **(2) Query Variation (O6-O8):** RLESIs exhibit high latency for range and kNN queries under varying query parameters, except for BM-tree [12]. **(3) Tail Latency (O9-O11):** RLESIs generally show increasing latency from P1 to P99, RLR-tree and Qd-tree suffer sharp spikes at high percentiles (e.g., P90 to P99). **(4) Insertion Performance (O12-O14):** Insertion latency is high for RLESIs, with BM-tree being the worst; however, they maintain query performance under heavy insertions. **(5) Index Building (O15-O18):** RLESIs incur high index

building costs due to model training overhead. Additionally, data partitioning-based indices incur higher building costs due to frequent I/O operations, while mapping-based indices are more space-efficient. **(6) Cardinality Variation (O19):** Most indices, including RLESIs, maintain performance scalability across dataset sizes and query types. **(7) Parameter Tuning (O20-O22):** Proper parameter optimization of RLESIs can lead to up to $120\times$ reductions in query latency.

④ **We systematically evaluate RLESIs against other baselines and identify challenges and limitations of RLESIs, e.g., high training costs.** Our study reveals that while RLESIs can outperform traditional indices in some cases, they suffer from high training costs and require careful parameter tuning. To mitigate training costs, we adopt advanced reward functions [30], achieving up to 27% training time reduction for BM-tree without compromising query performance.

II. SPATIAL QUERIES AND INDICES REVISITED

This section examines spatial queries and categorizes spatial indices into three primary types: data partitioning, space partitioning, and mapping [10], [31], [32], based on their building methods. In each category, we analyze its traditional, advanced, learned (LSI), and RL-enhanced (RLESI) versions. In this paper, we focus on points as the primary spatial data type because RLESIs are optimized for handling points.

A. Spatial Queries

Spatial queries enable the analysis and retrieval of spatial information based on the relationships and properties of geographic entities. Common query types include point queries, range queries, k nearest neighbor (kNN) queries, and spatial joins, each critical for spatial analysis.

Point queries retrieve the points that exactly match a specified query point. They can be viewed as a special case of range queries, where the query range is limited to a single point. A point query is formally defined as follows:

Definition 1 (Point Query (PQ)): Given a point $q \in E^d$, find all points p in dataset D that are equal to q:

$$PQ(q) = \{p \in D \mid p = q\}. \tag{1}$$

Range queries retrieve all points within a specified query range, also known as a query window, which is represented as a (hyper)rectangle. A range query is defined as follows:

Definition 2 (Range Query (RQ)): Given a d-dimensional interval $I^d = [l_1, u_1] \times [l_2, u_2] \times \cdots \times [l_d, u_d]$, find all points p in dataset D that are contained within I^d:

$$RQ(I^d) = \{p \in D \mid p \in I^d\}. \tag{2}$$

kNN queries generalize the nearest neighbor (NN) query (i.e., $k = 1$) by returning the k points in the dataset that are "nearest" to a given query point, as defined by a distance function $dist$. A kNN query is formally defined as follows:

Definition 3 (k Nearest-Neighbor Query (kNNQ)): Given a query point q and a distance function $dist$, return all points p in subset S ($|S| = k$) that have the minimum distance to q:

$$kNNQ(k, q) = \{p \in S \mid \forall p' \in D \setminus S : \text{dist}(p, q) \leq \text{dist}(p', q)\}. \tag{3}$$

---

[1] https://anonymous.4open.science/r/rl_spatial_benchmark

Spatial joins identify all pairs of points from two spatial datasets that satisfy a specified spatial predicate θ [33], [34]. Given two spatial datasets R and S, a spatial join query returns all pairs $(o, o^{'})$, where $o \in R$ and $o' \in S$ that satisfy a spatial predicate θ. Common predicates θ include `intersects`, `contains`, and `distance`.

Definition 4 (Spatial Join (SJ)): Given two spatial datasets R and S, and a spatial predicate θ, return all pairs of objects (o, o') such that:

$$SJ(R, S, \theta) = \{(o, o') \mid o \in R, o' \in S, \theta(o, o') = \text{true}\}. \quad (4)$$

B. Classification of Spatial Indices

Following the classification outlined in [9], [35], we categorize spatial indices into three primary categories: data partitioning-based, space partitioning-based, and mapping-based spatial indices.

1) Data Partitioning-based (DP-based): DP-based spatial indices partition data into smaller regions with minimal overlap to reduce query search space. By limiting overlap, these indices enable efficient query processing by retrieving fewer tree branches and nodes, leading to faster response times.

Traditional: *R-trees* [1] are widely used for spatial indexing [5]. An R-tree organizes data into a tree-structured nested rectangle, where each tree node represents a minimum bounding rectangle (MBR) that contains child nodes or data points. When a node exceeds its capacity, it is split into two smaller nodes based on criteria such as spatial area or margin, aiming to reduce query costs.

Advanced: Advanced versions of R-tree [1], such as R*-tree [2], revised R*-tree [36], and PR-tree [37], focus on reducing node overlap and optimizing node splitting to improve query performance. R*-tree minimizes the total area of MBRs and their overlap through a complex node-splitting strategy and reinsertion mechanism. STR-tree [3] avoids overlap through recursive sorting and tiling, efficiently partitioning data into spatially coherent groups.

Learned: AI+R-tree [38] integrates ML models to enhance traditional R-trees by directly predicting the target leaf node for queries, which avoids retrieving multiple leaf nodes. PLATON [15], despite incorporating RL, is categorized as an LSI because it introduces a novel index structure. It uses a learned partitioning policy based on packing R-tree tailored to specific data and query workloads.
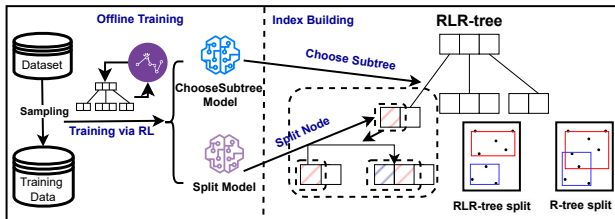

Fig. 2: An overview of RLR-tree building.

RL-enhanced: ACR-tree [39] and RLR-tree [14] refine the R-tree [1] structure rather than introducing new ones. As shown in Figure 2, RLR-tree [14] includes an offline training phase using sampled datasets. The trained models are then integrated into the index building phase of the R-tree [1]. During data insertion, RLR-tree [14] leverages the models to select subtrees and predict node splits when the target node is full. ACR-tree [39] employs an Actor-Critic [40] RL method, where the Critic estimates the long-term cost, and the Actor determines actions like splitting or packing.

2) Space Partitioning-based (SP-based): SP-based indices partition the entire space into smaller, non-overlapping regions, typically through a recursive approach. Unlike data partitioning, where overlaps can occur between the MBRs of partitioned data objects, space partitioning ensures no overlaps exist. Space partitioning continues until the spatial objects within an area are sufficiently reduced to fit into a leaf node. The design of these indices aims to reduce the number of comparisons needed to locate a spatial object.

Traditional: *KD-tree* [41], *Quadtrees* [42], and *Grid-Files* [43] are typical examples. KD-tree [41] partitions a multi-dimensional space by alternately choosing one dimension and finding the median to divide the space into two areas. Quadtree [42] recursively partitions a two-dimensional space into four quadrants until the contained data in each quadrant meets certain criteria. In contrast, *Grid-File* [43] partitions multidimensional space into a fixed grid of cells, wherein each cell links to a bucket that stores the records.

Advanced: KDB-tree [44] merges the dimension-wise splitting of a KD-tree [41] with the multi-way block storage of a B-tree [21]. Its inner nodes hold multiple partitions, optimizing data retrieval for disk-based storage. This approach reduces disk access, making it effective for large datasets. GKD-tree [13] heuristically adjusts partitions based on query patterns to enhance query performance.
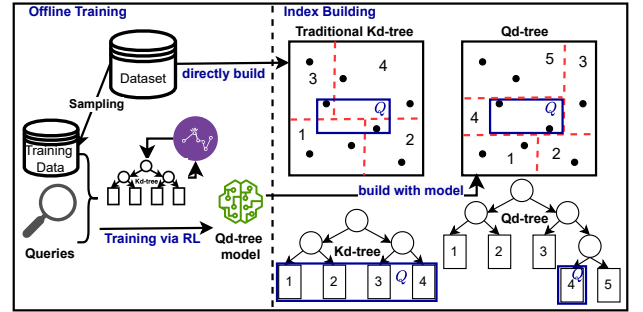

Fig. 3: An overview of Qd-tree building.

Learned: LISA [11] partitions the data space using a grid and maps data points based on a weighted aggregation of their coordinates. It utilizes learned functions to assign a shard ID to each point. SPRIG [45] is similar to LISA [11] in using a grid for partitioning, but uses bi-linear interpolation functions for prediction. Flood [10] and Tsunami [46] are memory-based, which partition a d-dimensional space using a (d-1)-dimensional grid, indexing the points by their coordinates in the last dimension using RMI [24]. CaseSpatial [35] and Spatial-LS [47] are in-memory indices leveraging different partition techniques. Waffle [48] is a workload-aware spatial index that combines the space and data partitioning.

RL-enhanced: The *Qd-tree* [13] uses RL to guide the space partitioning process of a KD-tree [41], which helps

minimize the number of data block accesses given a query workload (assuming that data points are grouped and stored in blocks). As shown in Figure 3, the Qd-tree [13] undergoes offline training with sampled datasets and queries to learn a partitioning strategy based on GKD-tree [13]. During each training episode, an intermediate Qd-tree [13] is incrementally built, making partitioning decisions until the tree is complete.

3) Mapping-based (MP-based): MP-based indices map multi-dimensional data objects into one-dimensional values. The key idea is to order spatial objects by their mapped values to ensure objects close in the multi-dimensional space remain close in the one-dimensional mapped values. Once ordered, the data can be efficiently packed into data blocks and indexed using *bulk-loading*, optimizing both storage and query performance.

Traditional: Space-filling curves (SFCs) are widely used for mapping purposes due to their effectiveness in preserving the locality of spatial objects. The most used SFCs are *Z-curve* [49] and *Hilbert-curve* [50], which map spatial coordinates into one-dimensional curve values. These mappings support the building of spatial indices, namely *ZR-tree* [4] and *HR-tree* [51]. The use of SFCs ensures that objects within a node are likely to be in proximity, thereby minimizing overlap and enhancing query performance.

Advanced: A key challenge in traditional SFCs is selecting an appropriate bit resolution. Low bit resolutions lead to excessive data clustering, while high resolutions result in overly fine-grained partitions, adding unnecessary complexity. To address this, *rank-space* [52] is proposed with a worst-case query guarantee. Instead of using actual values, this method calculates the curve value based on the rank of each dimension value. This approach can minimize the use of bits while ensuring each spatial object has a unique curve value. ZR-tree [4] that integrates rank-space is called ZRR-tree [52].

Learned: Learned MP-based indices leverage machine learning for improved mapping and indexing. *ZM-index* [8] maps data points according to their Z-curve [49] values for sorting, followed by indexing using Recursive Model Indexing (RMI) [24]. ML-Index [53] utilizes RMI for indexing and incorporates the *iDistance* technique [54] to map data points to one-dimensional values. RSMI [9] employs SFCs to create a hierarchy of space partitions, mapping data points to their corresponding partition IDs.
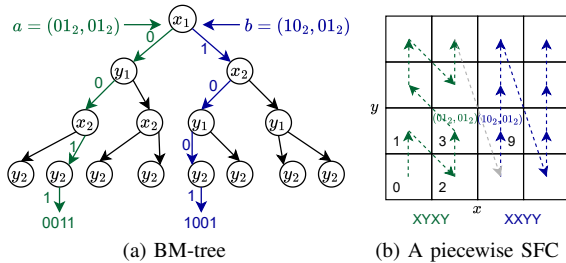

(a) BM-tree   (b) A piecewise SFC
Fig. 4: An example of a piecewise SFC (BM-tree).

RL-enhanced: BM-tree [12] and LMSFC [55] both learn space-filling curves (SFCs). LMSFC generates a global SFC via Bayesian optimization [56], while BM-tree constructs

piecewise SFCs by partitioning the data space with a quadtree [42], assigning each partition a specific SFC segment. Like Qd-tree [13] and RLR-tree [14], BM-tree requires offline training, whereas its output is not an RL model but a tree structure that encodes the learned SFC. In Figure 4a, the bits for $a = (01_2, 01_2)$ are $x_1 = 0$, $x_2 = 1$, $y_1 = 0$, and $y_2 = 1$. Traversing BM-tree from the root, one follows left for 0 and right for 1, resulting in a curve value of 3 for a (Figure 4b).

III. BENCHMARKING FRAMEWORK

Our benchmarking framework comprises two modules: index training module (ITM) and index building module (IBM). This design is more effective than a standalone module, as it avoids the complexity of integrating training (often in Python) and implementation (typically in C++) in a single pipeline, and also enables the reuse of trained RLESIs.

In Figure 5, ITM is a Python-based module designed to train RLESIs by selecting best-performing parameters. For each RLESI, we use configurations specific to the dataset and workload, testing various parameter combinations to find the best-performing setup. The traditional version, such as R-tree [1], serves as the input to the *trainer*. After an RLESI is trained, the trainer can output the trained models. For RLR-tree [14] and Qd-tree [13], their outputs are the PyTorch [28] models, while for BM-tree [12], the output is a tree structure that represents a piecewise SFC.

IBM supports disk-based spatial indices and implements them in C++. It extends from a well-known spatial index library libspatialindex [29], which already supports R-trees [1], R*-trees [2], and the skeleton of MP-based indices. Building on this foundation, our framework supports point, range, and kNN queries, as well as spatial joins. To integrate trained models from ITM, we incorporate a *loader* component using *LibTorch*, the C++ API of PyTorch [28]. LibTorch is only used for static model loading. The loader takes PyTorch models and adapts them for C++ programs, which ensures seamless integration into the indexing pipeline.

Algorithm 1: Tuning of RLESI

Input: Parameter space $\mathcal{P}$, build cost limit T_{build}
Output: Best-performing parameter configuration $\mathcal{P}_{opt}$

1   $min\_query\_cost \leftarrow \infty$, $\mathcal{P}_{opt} \leftarrow$ null;
2   /* $\mathcal{P}$: { "epoch": [8, ..., 12], "sample_size": [1k, ..., 16k] }*/
3   **for** $\mathbf{p} \in generate\_configurations(\mathcal{P})$ **do**
4   **if** $is\_build\_required(\mathbf{p},\ \mathcal{P}_{opt})$ **then**
5         $build\_cost, index \leftarrow$ train_index($\mathbf{p}$);
6   **if** $build\_cost < T_{build}$ **then**
7            $query\_cost \leftarrow$ run_queries($index$);
8   **if** $query\_cost < min\_query\_cost$ **then**
9               $min\_query\_cost \leftarrow query\_cost$, $\mathcal{P}_{opt} \leftarrow \mathbf{p}$;

10   **return** $\mathcal{P}_{opt}$;

A. Implementation of Parameter Tuning

Our benchmark employs a structured grid search process for (hyper-)parameter tuning of RLESIs, as outlined in Algorithm 1. This process explores key parameters in $\mathcal{P}$ (e.g.,

Fig. 5: Framework for benchmarking spatial indices: learning, deployment, and building in libspatialindex.

epochs, sample size) that impact both index building and query efficiency (Line 2). To manage computational costs, an early stopping threshold T_{build} halts the search if the build cost exceeds a set limit. The function `is_build_required()` compares two configurations $\mathbf{p}$ and $\mathcal{P}_{opt}$, returning `true` if $\mathbf{p}$ could be less expensive for training. When a configuration $\mathbf{p}$ results in a build cost below T_{build}, it is then compared with $\mathcal{P}_{opt}$. This method guarantees the selection of configurations that satisfy both build and query performance criteria.

B. Selection of Baselines

The vast number of spatial indices makes comparing them all unfeasible. Thus, we prioritize representative indices. Figure 6 summarizes the selected indices in chronological order and groups them by category using background colors: DP-based (blue), SP-based (green), and MP-based (orange). Beneath each index, we tag its type with A/L/R for Advanced / Learned / RL-enhanced, while traditional indices are left unmarked. For capabilities, we indicate native support for point, range, kNN, spatial join, and updates using ✓ and ✗.

Fig. 6: Chronological listing of selected indices

We ensure fairness and representativeness by selecting indices across all categories. For RLESIs, we choose one state-of-the-art representative per family: RLR-tree [14] (DP, only open-sourced option), Qd-tree [13] (SP, only RL-enhanced variant), and BM-tree [12] (MP, LMSFC [55] lacks open-source). Each RLESI is compared against its direct traditional counterpart: R-tree [1], KD-tree [41], and ZR-tree [4]. Advanced baselines extend these families: R*-tree [2] for R-tree, GKD-tree [13] for KD-tree, and ZRR-tree [52] for ZR-tree. For LSIs, we select strong representatives: PLATON [15] (DP), LISA [11] (SP), and ZM-index [8] (MP). This setup ensures coverage of traditional, advanced, learned, and RL-enhanced methods across DP-, SP-, and MP-based categories.

C. Implementation Details

We implement all indices in IBM to ensure consistency and fair comparison. For each RLESI, we reuse its original RL model and adopt the original query algorithms. In contrast, LSIs introduce entirely new index structures that require customized implementations. We thus follow their original implementation in libspatialindex.

1) Index Nodes: In IBM, the index structures of all baselines are categorized into two main components: *inner* nodes and *leaf* nodes. Inner nodes, including all non-leaf nodes from the root to the second last layer, do not store spatial objects but act as organizational structures to support search and traversal.

The maximum node capacity is set to 100 entries (i.e., $B = 100$), with each entry occupying 40 bytes: 32 bytes for the coordinates of an MBR and 8 bytes for either a pointer to disk storage (in non-leaf nodes) or a data ID (in leaf nodes). The remaining space is allocated to storing metadata, such as the level of the node and the actual number of entries. The block size is set to 4 KB to ensure that each node fits within a single block, following the default settings from existing works [52], [57]. For LISA [11] and ZM-index [8], we use a large leaf node capacity, i.e., $B = 10,000$ to boost space utilization. This avoids the inefficiency of space usage when using $B = 100$, which can cause utilization to drop below 10% due to recursive partitioning after reaching node limits.

2) Insertion Strategy: We implement the insertion strategy according to the original designs of each index. R-tree [1] is built using an insertion-based strategy, which we follow for RLR-tree [14] and PLATON [15]. R*-tree [2] enhances R-tree by incorporating a reinsertion technique for node splitting. For KD-tree [41], a point is inserted by performing a point query and adding the point to the target leaf node. If the leaf node is full, it is partitioned using the same procedure as in index building. Since no specific insertion methods are proposed for Qd-tree [13] and GKD-tree [13], we reuse the insertion method of KD-tree [41]. For MP-based indices, maintaining order during insertion is challenging due to their reliance on sorted spatial objects during index building. As a result, we reuse the insertion method of R-tree [1] for these indices, given their R-tree-based structure.

IV. EXPERIMENTAL SETUP

All experiments are executed on a server running 64-bit Ubuntu 20.04 with a 3.60 GHz Intel i9 CPU, 64 GB RAM, and a 1 TB HDD. We use *PyTorch* 1.4 [28] to train models for RLESIs based on the CPU and employ its C++ APIs to load models and invoke index building.

A. Datasets

We evaluate all baselines on six 2D datasets: three synthetic datasets and three real-world datasets derived from OpenStreetMap (OSM) [58], each containing 100M spatial points following configurations commonly used in prior work [9], [11], [14], [52]. Figure 7 illustrates the spatial distributions by sampling 10,000 points (blue) and overlaying 20 example

range queries (red), which are generated by randomly sampling 20 centers following the data distribution and wrapping them with axis-aligned rectangles. Histograms on the x and y axes further highlight the distinct distributional characteristics.
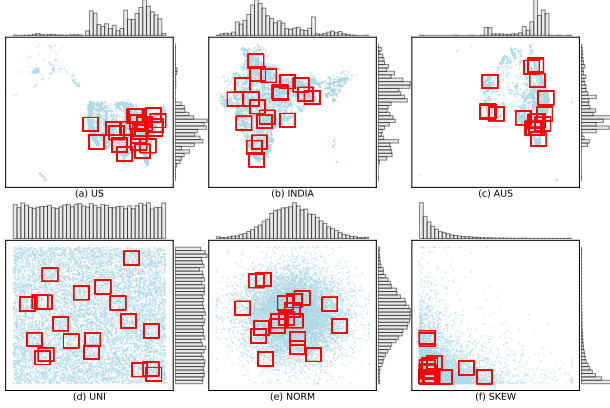

Fig. 7: Six datasets used in the experimental study.

The three synthetic datasets represent uniform, normal, and skewed distributions, which are widely used in recent studies [11], [12], [14], [15], [30], [52]. For example, PLATON, RLR-tree, and ZRR-tree evaluate across all three distributions, whereas LISA is limited to skewed and uniform datasets, and BM-tree reports results on normal and uniform distributions. We use 100M points for all to ensure consistency with the scale of our real datasets.

For real datasets, we select three OSM datasets (US, India, and Australia) to capture diverse geographic patterns. The US and India datasets have been widely used, for example, by PLATON (US and India), RLR-tree (India), and BM-tree (US). Here, the US dataset reflects dense metropolitan clusters with sparse rural coverage, the India dataset exhibits a more balanced spread across cities and towns. The Australia dataset has not been used in prior work, but it provides a more skewed distribution with points concentrated along coastal cities. Together, these datasets cover concentrated, balanced, and skewed real-world spatial distributions.

B. Workloads

We test all the indices on seven distinct workload types:

1) **Point Query-Only:** Executes 200,000 point queries sampled from the corresponding dataset, following [16].
2) **Range Query-Only:** Executes 1,000 range queries with centroids sampled like point queries. The default query shape is square, with edge lengths varying as {0.01%, 0.05%, **0.1%**, 0.5%, 1%} of the data space [12]. Additionally, aspect ratios vary across {1/16, 1/4, **1**, 4, 16}, with {1/4, 1, 4} matching settings in BM-tree [12], and 1/16, 16 added to explore skewed queries.
3) k**NN Query-Only:** Executes 1,000 kNN queries with query points sampled similarly to point queries. Values of k are chosen from {1, 5, **25**, 125, 625}, following configurations in [14].
4) **Spatial Join-Only:** Executes 1,000 spatial joins using the classical hierarchical join algorithm [33]. We use

range-based predicates, as they naturally define the spatial overlap conditions required for joins.

5) **Write-Only:** Inserts 10M randomly generated data points after the initial bulk-loading of 10M points.
6) **Write-Heavy:** Performs 90% inserts and 10% lookups after bulk-loading 10M random data points. For every 20 operations, 18 are inserts and 2 are lookups.
7) **Read-Heavy:** Performs 90% lookups and 10% inserts after bulk-loading 10M random data points. For every 20 operations, 2 are inserts and 18 are lookups.

Write-only, write-heavy, and read-heavy workloads follow the configurations in [17], [59], adapted for spatial data. In our experiments, the query range with a 0.1% edge length and an aspect ratio of 1 is the default setting for building RLESIs, GKD-tree, and PLATON. Default parameters across workloads are highlighted in **bold**.

C. Metrics

We use the following metrics to evaluate index performance:

1) **Index Building Cost:** Measures the total time required to build the index from reading the dataset to storing it on disk. For RLESIs, this includes model training.
2) **Average I/O per Search Query:** Captures the average number of disk blocks accessed for each query. This is divided into **leaf I/O**, representing data storage nodes, and **inner I/O**, which accounts for index nodes. This metric provides a detailed breakdown of I/O efficiency.
3) **Average Query/Insert Latency:** Reports the average response time for querying or inserting data into the index.
4) **Latency Distribution (P1 to P99):** Tracks latency percentiles from the 1st to the 99th, including **tail latency** (P99). This helps evaluate both typical and worst-case performance, as RLESIs may exhibit suboptimal behavior outside trained conditions.
5) **Index Statistics:** Includes tree height, index size, node utilization, and metrics specific to insertion, such as the number of node splits or adjustments. These statistics help explain performance outcomes, e.g., how frequent splits impact insertion latency.

V. EXPERIMENTAL RESULTS

To evaluate the effectiveness of RLESIs, we conduct a series of experiments that aim to answer the following key questions: **Q1:** How do RLESIs compare with traditional, advanced, and learned indices in terms of query performance? (Section V-A and Section V-B) **Q2:** How do all indices perform under worst-case scenarios, such as tail latency? (Section V-C) **Q3:** How well do the indices handle insertions, especially under write-heavy workloads? (Section V-D) **Q4:** How do key index statistics (e.g., size, height, splits) evolve during index building? (Section V-E) **Q5:** How does dataset cardinality affect performance across different index types? (Section V-F) **Q6:** To what extent can parameter tuning improve RLESI performance? (Section V-G)

For better visualization, indices in three categories are colored blue (DP-based), orange (MP-based), and green (SP-based). Within each group, four types (traditional, advanced,

learned, and RL-enhanced) are filled by different patterns, and the same type uses consistent patterns across groups.

A. Query Performance Study

We present latency and I/O results for range queries. For other query types, we report latency only, as I/O trends closely align with latency, allowing for brevity. All RLESIs have been tuned, with details outlined in Section V-G.
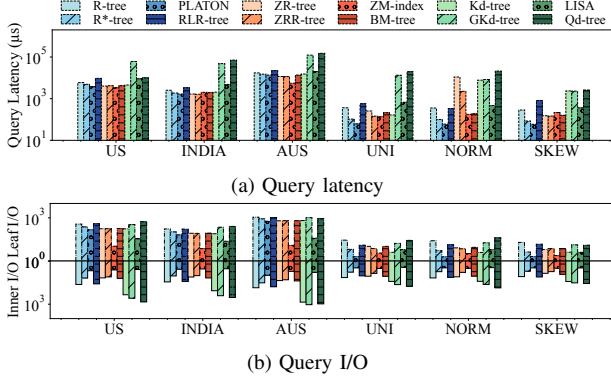

(a) Query latency

(b) Query I/O

Fig. 8: Range query

1) Range Query-Only: Figure 8 presents the range query latency and I/O count across six datasets. We observe that:

O1: Across all datasets, RLESIs show competitive query performance compared to traditional indices but fall short of advanced and learned baselines. RLR-tree achieves comparable performance to R-tree in terms of query latency and I/O. However, RLR-tree does not surpass R*-tree or PLATON in range query performance. This limitation arises because RLR-tree relies on R-tree for node splits, leading to suboptimal overlap management compared to R*-tree and PLATON. As shown in Figure 8b, R*-tree and PLATON achieve fewer leaf and inner I/Os than R-tree and RLR-tree.

BM-tree performs similarly to ZR-tree across all datasets, with better query performance on real datasets. This advantage is attributed to BM-tree learning from the mapped values of spatial data points, which can enhance the quality of the learned model. Compared to ZRR-tree and ZM-index, BM-tree exhibits comparable query latency and I/O on most datasets. However, BM-tree outperforms others in query latency on the NORM dataset, while the I/O cost remains relatively close. This suggests that NORM (see Figure 6e) can cause structural traversal differences for ZR-tree and ZRR-tree.

Qd-tree and GKD-tree show inferior query latency and I/O compared to KD-tree and LISA across most datasets. This is attributed to the RL model in Qd-tree failing to converge effectively, as indicated by training logs, which is also a challenge highlighted in [14]. For GKD-tree, the partitioning strategy heavily relies on query boundaries due to its greedy nature, which limits the generalizability.

O2: RLESIs show consistent alignment between query latency and I/O across datasets as well as other indices except for ZM-index and LISA. ZM-index and LISA show higher latency but lower I/O on all datasets due to the impact of skewed data on model prediction accuracy, which causes more computational costs. For ZM-index and LISA, inner I/O is

solely associated with recursive model predictions rather than tree traversal, leading to no overlap between these operations. Additionally, ZM-index and LISA benefit from large leaf node capacity, which reduces the number of leaf I/Os required.
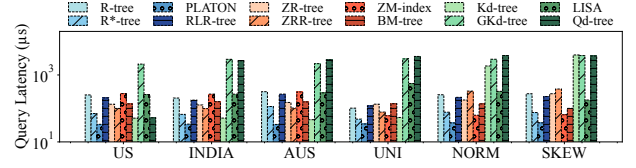
Fig. 9: Point query latency

2) Point Query-Only: Figure 9 illustrates the point query latency across six datasets. We observe that:

O3: Advanced and learned indices consistently outperform RLESIs in query latency across all datasets. RLESIs offer moderate improvements over traditional indices in specific cases. RLR-tree achieves slightly better query latencies than R-tree, while BM-tree shows competitive performance relative to ZR-tree, though not always superior. There is a special case where on the US dataset, Qd-tree shows low query latency, this is due to the tuning, which is also shown in Figure 22. In comparison, advanced indices, such as R*-tree and ZRR-tree, benefit from optimized structures, delivering balanced performance across various query types, including point queries. Among learned indices, PLATON achieves the best query latency. ZM-index performs exceptionally well on synthetic datasets due to their predictable patterns but incurs higher latency on real-world datasets, where data is more challenging to model [8], [9].
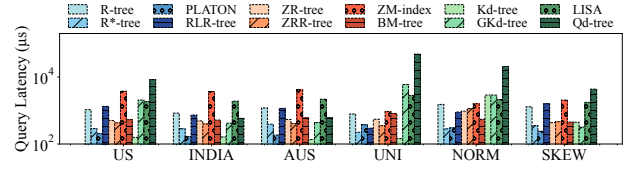

Fig. 10: kNN query latency

3) kNN Query-Only: Figure 10 shows the kNN query latency across different datasets. We observe that: **O4: Among RLESIs, BM-tree achieves the best kNN query latency across most datasets; however, KD-tree and PLATON still outperform all RLESIs on most datasets.** RLESIs fail to deliver competitive kNN query latency, particularly RLR-tree and Qd-tree, which do not effectively partition the data space. However, BM-tree achieves better performance by clustering closely located data points and packing them consecutively into blocks, optimizing query latency. Overall, RLESIs lag behind advanced and learned baselines in kNN query latency, which is consistent with observations in range queries. The superior performance of KD-tree and PLATON is attributed to minimal or no node overlap, reducing the need to visit unnecessary nodes during kNN queries. ZM-index underperforms for kNN queries due to the absence of a dedicated kNN algorithm, as it reuses the method from R-tree.

4) Spatial Join-Only: Figure 11 illustrates the spatial join latency across six datasets. We observe that:

O5: RLESIs are competitive with traditional indices and often, though not always, behind advanced and learned
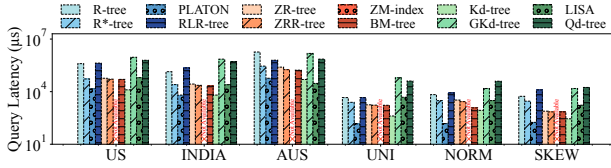
Fig. 11: Spatial join latency

indices. RLR-tree can only outperform R-tree on AUS dataset, while being slower on the other datasets. This stems from RLR-tree's suboptimal data partitioning, as illustrated also in Figure 8a. BM-tree performs comparably to ZR-tree and ZRR-tree, as BM-tree minimizes overlap and backtracking, yielding fewer overlaps. Qd-tree underperforms its counterparts as it derives partitions from a subset of queries, which can only learn substantial partitions.
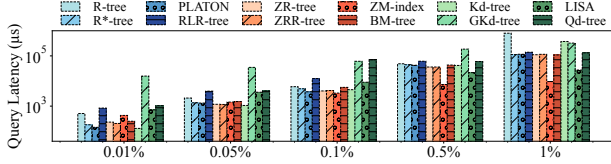
(a) Varying edge length of queries
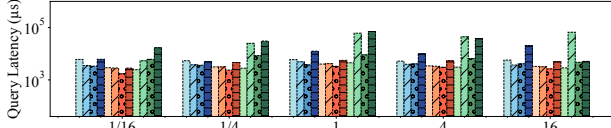
(b) Varying aspect ratio of queries
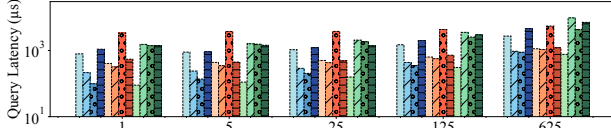
(c) Varying k

Fig. 12: Varying queries

B. Query Variation Study

1) Varying Query Range Size: We execute range queries with varying edge lengths on the US dataset, with query latency results shown in Figure 12a. We observe that:
O6: RLESIs and GKD-tree exhibit higher range query latency for small query ranges. As the query range increases, query latency rises for all indices due to the larger number of data points involved. RLESIs show inferior performance when the edge length is 0.1% or smaller, particularly for RLR-tree and Qd-tree. However, as the edge length grows, RLESIs demonstrate moderate latency improvements. In comparison, GKD-tree, an advanced baseline, exhibits worse latency across all ranges. Among LSIs, PLATON delivers consistently competitive latency across all edge lengths, while ZM-index shows outstanding performance as the edge length exceeds 0.1%.

2) Varying Query Range Aspect Ratio: We execute range queries with varying aspect ratios on the US dataset, presenting the query latency in Figure 12b. We observe that:
O7: RLESIs and GKD-tree are sensitive to query aspect ratios. RLESIs and GKD-tree continue to exhibit inferior query latency when the aspect ratio of query ranges varies.

RLR-tree outperforms R-tree when the aspect ratio is 1/4 and 1/16, but still falls short of R*-tree and PLATON. Notably, the query latency of RLR-tree increases significantly when the aspect ratio is 16. Similarly, BM-tree falls behind other MP-based counterparts, due to its quadtree partitioning, which is less effective for skewed queries.

3) Varying k: We execute kNN queries on the US dataset with k varying from 1 to 625 in Figure 12c. We observe that:
O8: Except for BM-tree, RLESIs exhibit the highest latency increase with larger k values, while KD-tree and PLATON remain efficient. As k increases, the query latency of all indices rises, and the result pattern does not change much. For RLESIs, RLR-tree and Qd-tree show the most pronounced increase in latency, indicating their inefficiency in handling large k values. This inefficiency arises because RLR-tree requires more computations and Qd-tree does not partition the data space effectively, requiring more leaf node accesses as k increases. In contrast, KD-tree and PLATON are efficient in handling kNN queries, while another LSI, ZM-index, shows inferior performance due to its missing implementation of kNN query method and the reuse of kNN query from R-tree.
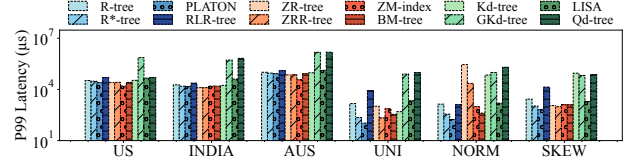
(a) Range query
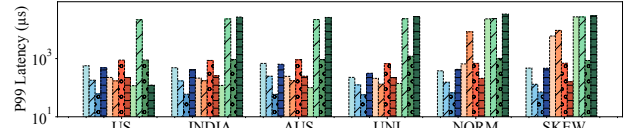
(b) Point query
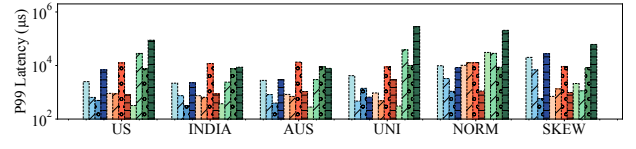
(c) kNN query

Fig. 13: P99 Query latency

C. Latency Study

1) Tail Latency: Figure 13 presents P99 latency for point, range, and kNN queries across all datasets. We observe that:
O9: RLESIs suffer from unstable P99 latency, especially RLR-tree and Qd-tree, while traditional indices maintain consistency. This variability comes from RLESI building. For instance, BM-tree performs relatively well as its MP-based structure balances the tree effectively and mitigates the impact of worst-case queries. Conversely, RLR-tree and Qd-tree often exhibit poor performance, with RLR-tree struggling in most scenarios and Qd-tree performing significantly worse. In contrast, traditional indices like KD-tree and R*-tree demonstrate more consistent P99 latency. This is attributed to their robust design, e.g., KD-tree uses recursive median splitting, and R*-tree employs reinsertion strategies to handle overflows.

These methods ensure effective partitioning of data and space, minimizing imbalances and maintaining stable query.
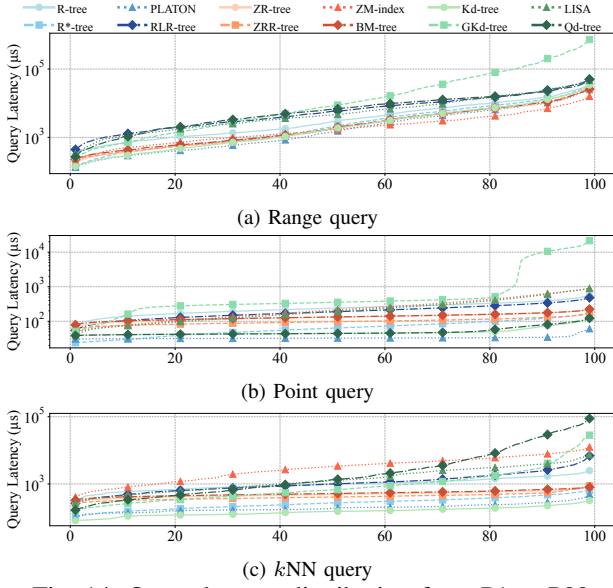

(a) Range query

(b) Point query

(c) kNN query

Fig. 14: Query latency distribution from P1 to P99

2) Latency Distribution: Figure 14 shows the P1 to P99 latency distribution on the US dataset. We observe that:

O10: RLESIs exhibit consistent range and point query latency increases from P1 to P99, comparable to traditional indices. Query latency varies due to differences in data density and node overlap across spatial regions. Denser areas lead to more node accesses and higher latency, especially when queries intersect overlapping partitions. RLR-tree and BM-treeexhibit similar latency distributions to traditional indices, demonstrating robust performance across spatial variations.

O11: RLESIs based on DP or SP structures suffer from latency increases at high percentiles for point and kNN queries. This is caused by query regions intersecting dense clusters or heavily overlapping nodes, which significantly increases the number of accessed nodes. For instance, RLR-tree and Qd-tree exhibit steep latency growth from P90 to P99, similar to traditional DP- and SP-based indices such as GKD-tree. In contrast, MP-based RLESIs like BM-tree maintain smoother latency curves, as their one-dimensional mappings reduce sensitivity to spatial skew.

D. Insertion Performance Study

1) Insertion Latency: Figure 15 shows insertion latency for ready-heavy, write-heavy, and write-only workloads. We observe that:

O12: All RLESIs exhibit high insertion latency, with BM-tree performing the worst. Insertions trigger node splits and new node creation. For RLESIs, which add extra costs due to model predictions for RLR-tree and Qd-tree. While BM-tree does not rely on model predictions, its densely packed leaf nodes (fill ratio 1.0) necessitate frequent node splits during insertions. In contrast, DP- and MP-based indices reserve more space in their leaf nodes, resulting in fewer node splits. Notably, LISA exhibits the highest latency as it creates a model

(a) Write-Only
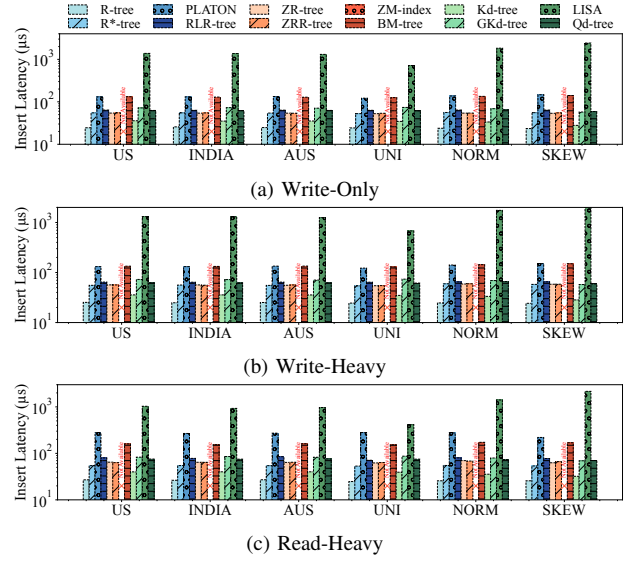

(b) Write-Heavy

(c) Read-Heavy

Fig. 15: Insertion latency for different workload types

node to handle overflow and reassign points to different leaf nodes. For PLATON, it exhibits the second highest latency due to its top-down partitioning strategy, which fully packs nodes and intensifies the cost of insertions.
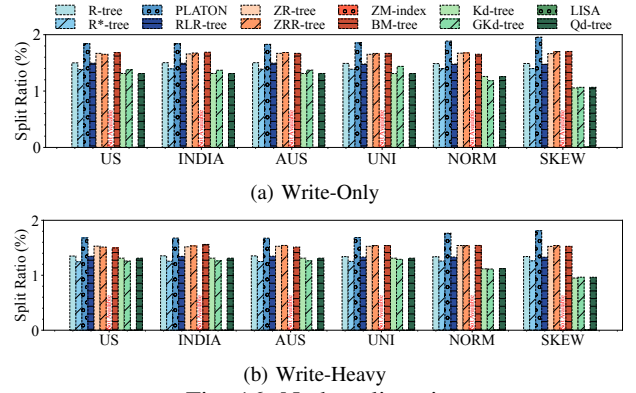

(a) Write-Only

(b) Write-Heavy

Fig. 16: Node split ratio

2) Node Split Ratio: We show the node split ratio of write-only and write-heavy workloads in Figure 16. We observe that:

O13: PLATON and MP-based indices exhibit the highest split ratio, while SP-based indices have the lowest, especially LISA. As explained for insertion latency, PLATON and MP-based indices fully utilize their leaf nodes, leaving no unused space. Consequently, inserting new points into these indices often requires a leaf node split with high probability. In contrast, SP-based indices and some DP-based indices retain unused space in their leaf nodes, which allocates more space for insertions. This significantly reduces the likelihood of node splits. Notably, LISA exhibits an extremely low split ratio. It is visually close to zero in Figure 16, though a small number of splits still occur.

3) Query Latency: Figure 17 shows query latency for write-heavy workload. Due to space constraints, read-heavy results are provided in the supplementary materials. We observe that:

O14: RLESIs exhibit comparable query latency under heavy insertions. RLR-tree and Qd-tree slightly outperform
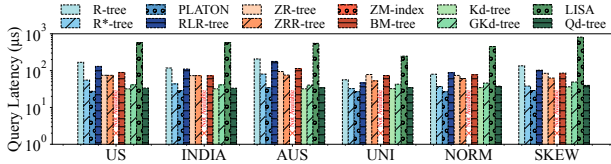
Fig. 17: Query latency during insertion (write-heavy)

R-tree and KD-tree, respectively, while BM-tree performs similarly to ZR-tree. This contrasts with the results of point queries alone (cf. Figure 9). LISA inserts more points into each leaf node, which increases the search cost at the leaf level, even though its index traversal remains efficient. Other SP-based indices show low query latency as insertion-related workloads use only one-tenth of the dataset, which reduces the index height for SP-based indices, thereby lowering query latency. In contrast, the height of DP- and MP-based indices remains largely unchanged, resulting in minimal latency variation.
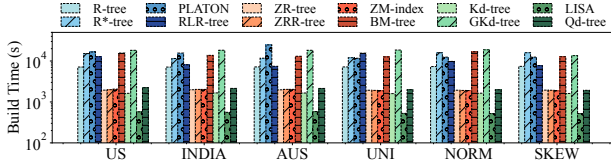
Fig. 18: Index build time (training time included)

TABLE I: Training "+" building time of RLESIs (minutes).

Index	US	INDIA	AUS	UNI	NORM	SKEW
RLR-tree	94 + 117	18 + 116	4 + 119	140 + 118	45 + 117	9 + 119
Qd-tree	6 + 30	5 + 29	4 + 29	5 + 28	4 + 27	4 + 27
BM-tree	257 + 3	226 + 2	214 + 2	206 + 2	277 + 2	209 + 2

E. Index Building Study

Figure 18 shows total build time, and Table I breaks down the training and building costs for RLESIs. We observe that:
O15: RLESIs, R*-tree, PLATON, and R*-tree incur high index building costs. RLESIs are time-consuming to build, due to the expensive training process. For example, RLR-tree incurs additional overhead from its training process, while both PLATON and BM-tree utilize Monte Carlo tree search (MCTS) for index training, which has high computational complexity. Notably, the time cost for parameter tuning in RLESIs, exceeding 30 hours per dataset for each index, is omitted here. For R*-tree, the reinsertion process is particularly time-intensive during index building. Similarly, GKD-tree involves computationally expensive operations to determine the optimal partition locations [13].

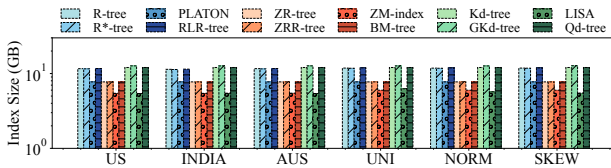*1) Index Size and Height:* Figure 19 shows the index sizes, and Figure 20 shows the index heights. We observe that:

Fig. 19: Index size

O16: The MP-based RLESI, BM-tree, is the most space-efficient among all indices, except for LISA and ZM-index. This is because MP-based indices, including BM-tree, tightly
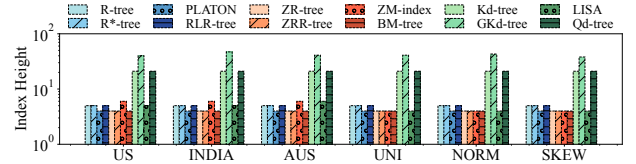
pack data in leaf nodes, with only the last node potentially partially filled. This leads to lower tree height and smaller index size, as shown in Figure 20. In contrast, DP-based RLESIs like RLR-tree use a lower fill ratio (0.4 in this study), leading to only 67%–73% node utilization. SP-based RLESIs, such as Qd-tree, also incur higher space costs, as internal nodes have fixed fanout and leaf nodes are not always fully utilized (around 95%). Notably, LSIs like LISA and ZM-index save space by storing more data in each leaf and using linear models instead of tree-based inner structures, thus reducing the number of intermediate nodes.

TABLE II: The number of node adjustments (million).

Index	US	INDIA	AUS	UNI	NORM	SKEW
R-tree	6.5	6.4	6.2	7.8	2.6	7.4
R*-tree	55.3	55.8	55.3	56.8	19.3	59.0
RLR-tree	7.4	7.1	7.1	8.5	8.4	8.9

(a) Range query
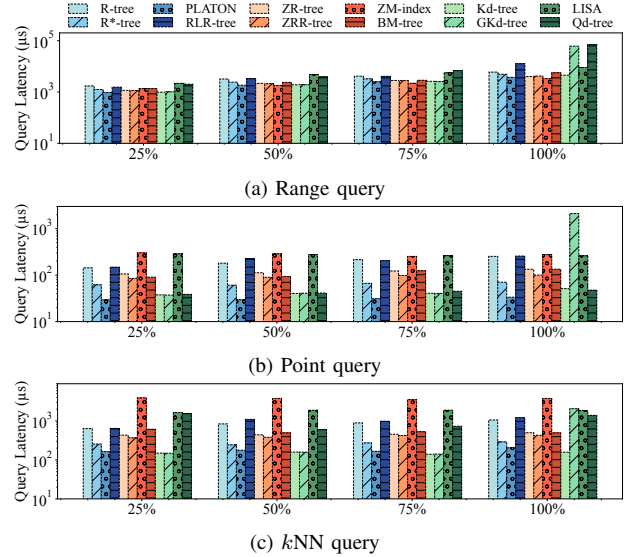

(b) Point query

(c) *k*NN query

Fig. 21: Varying dataset cardinality

O17: SP-based indices exhibit the highest tree height, except for LISA. This is because binary partitioning limits each internal node to two children, naturally leading to deeper trees. In contrast, MP-based indices such as BM-tree achieve the lowest tree height by fully packing nodes. However, tree height is not a direct indicator of performance, as it reflects a trade-off between depth and fanout. Although LISA appears shallower, it does not follow binary partitioning like typical SP-based indices and is used here as a representative LSI in the absence of a binary-partitioned learned index.

2) Node Adjustment in Building: Table II presents node adjustments for three DP-based indices. PLATON, SP-based, and MP-based indices are excluded as they write sequentially
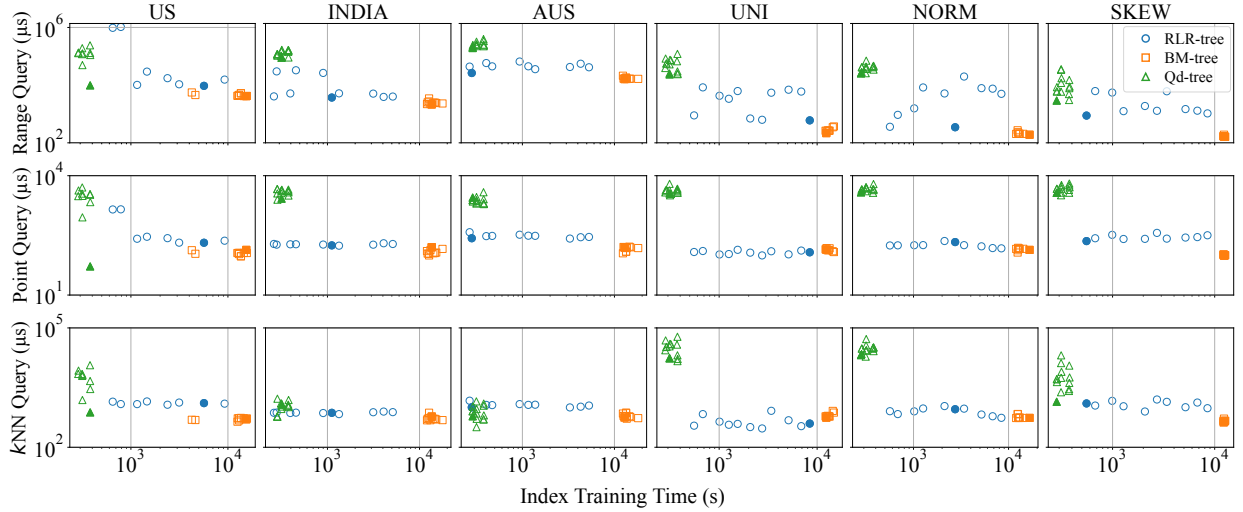

Fig. 20: Index height

Fig. 22: Query latency for range, point, and kNN queries vs. index training time across varying training configurations.

on storage without adjustments. We observe that:

O18: R*-tree requires significantly more splits compared to RLR-tree, while RLR-tree requires only slightly more splits than R-tree. As previously mentioned, R*-tree employs a reinsertion strategy, reinserting a portion of the entries when a node overflows. In contrast, R-tree and RLR-tree only adjust nodes during node splits. Additionally, R*-tree is more sensitive to skewed data, as evidenced in Table II, where it exhibits the highest number of node adjustments on the SKEW dataset. RLR-tree shows more node adjustments than R-tree because it optimizes node splits to minimize node size.

F. Cardinality Variation Study

Figure 21 shows query latency and index building cost as the cardinality of the US dataset varies. We observe that:

O19: RLESIs exhibit stable query latency with increasing dataset size, consistent with trends across most indices, highlighting strong scalability for all query types. This stability reflects the scalability of the indices, which preserve performance trends as data volume grows. These results highlight the robustness of RLESIs and LSIs in large-scale settings, consistently delivering reliable query performance across different query types.

G. Parameter Tuning Study

We tune all RLESIs using the configurations listed in Table III. While the search space remains limited, the chosen parameters have a substantial influence on training overhead and are sufficient to balance tuning efficiency against query performance. We measure parameter effectiveness using latency on the default range query workload, since all RLESIs define their training objectives on range access. Once chosen, the parameters are reused across all workloads on the same dataset, rather than re-tuned for each workload.

Figure 22 illustrates the relationship between training time and query latency for range, point, and kNN queries across different datasets. Configurations achieving the lowest range

TABLE III: Configurations for RLESIs.

Index	Configurations
RLR-tree	{epoch:[6,8,10,12,14], sample:[10k, 20k, 40k, 80k]}
Qd-tree	{query:[200, 400, 800], sample:[10k, 20k, 40k, 80k, 160k]}
BM-tree	{height:[8, 10, 12], sample:[10k, 20k, 40k, 80k]}

query latency are highlighted with filled markers. Overall, the tuning process requires a total of approximately $30 \times 3 \times 6$ hours (30 hours per baseline across 3 RLESIs and 6 datasets).

We use a grid-based search with query and build cost limits because, given the small number of parameters, it is simple to implement and easy to reuse. Moreover, tuning can be independent of the benchmark, thus users can have their own parameters and try alternative tuning methods.

O20: Parameter tuning improves range query performance by up to two orders of magnitude. On the US dataset, the query latency for RLR-tree shows a $120\times$ improvement between the best and worst-performing configurations. Significant enhancements are also observed across other datasets for RLR-tree. Similarly, for Qd-tree, tuning parameters on the US dataset achieves up to a $28\times$ reduction in range query latency, with approximately a $10\times$ improvement on the AUS and SKEW datasets. In contrast, BM-tree shows limited improvement, likely due to restricting the sample size to minimize training time. Larger sample sizes are avoided to prevent excessive training costs.

O21: Qd-tree exhibits consistent training costs, RLR-tree shows variability, while BM-tree is consistently slow but stable. Qd-tree uses a mirrored KD-tree structure for learning, where the fixed two-partition splitting minimizes the impact of parameter changes. Conversely, RLR-tree explores multiple partitioning strategies during R-tree building, leading to higher and more variable training costs. This sensitivity to parameter tuning results in scattered performance. In contrast, BM-tree incurs consistently high training costs as it builds intermediate indices and uses real queries to compute rewards, with parameter changes having minimal effect.

O22: Different query types require tailored configurations,

and no single RLESI excels across all query types, even with tuning. Qd-tree benefits from efficient training and delivers strong performance on the US dataset. RLR-tree demonstrates significant potential through tuning in expanded parameter space, achieving competitive performance for point and kNN queries across most datasets. BM-tree, despite its longer training time, offers consistent efficiency and stability across all query types.
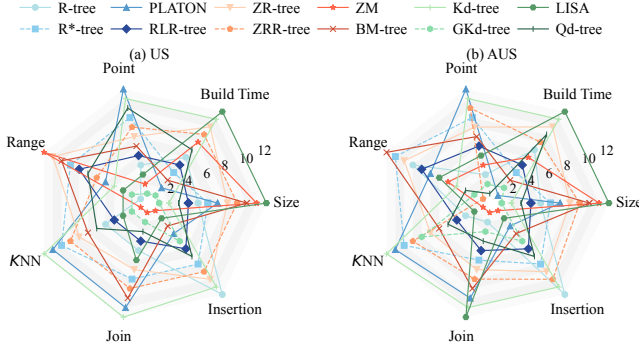

Fig. 23: Comparison across five workloads, build time, and index size on US and AUS datasets. Each axis shows a rank-based score (higher is better).

VI. INSIGHTS AND LIMITATIONS

A. Summary of Overall Evaluation

Figure 23 summarizes the overall performance of all baselines on two representative datasets, US and AUS, evaluated across seven metrics: index size, build cost, query latency for four query types, and insertion performance. Among all indices, LISA is the most efficient in both build time and index size. In contrast, RLESIs generally incur higher building costs due to training overhead and the need for hyperparameter tuning to avoid suboptimal configurations. In terms of query performance, PLATON, ZM-index, and KD-tree demonstrate strong performance for point, range, kNN, and join queries, respectively. However, RLESIs show unstable performance for point and kNN queries, as they are trained only on range queries and lack generalization. For insertion latency, R-tree is the most efficient. Its leaf nodes are underutilized compared to MP-based indices, and it avoids the reinsertion overhead seen in R*-tree. In contrast, RLESIs suffer from additional overhead during insertion due to model inference.

B. Challenges and Limitations of RLESIs

Challenges. As discussed in Section V-G, training RLESIs is costly, as each parameter combination requires a full *train → build → evaluate* cycle. Here, we train on sampled data, but build and evaluate the index on the full dataset to ensure accuracy. This mismatch means models often optimize policies on artificial geometries that fail to capture true data locality. For example, in the full dataset, 100 spatially close points may be spread across multiple pages, whereas in a sparse sample the same 100 points can fit on a single page. Therefore, the trained models do not capture data locality well.

Limitations. Within the cycle, the costs of building and evaluating are fixed, while training varies. From our tuning

study in Section III-A, we know that parameter selection leads to cost differences. Beyond that, reward calculation also affects training overhead. For instance, BM-tree's reward mechanism repeatedly evaluates intermediate partitions, resulting in high training cost (Figure 22).

C. Opportunities for Improvement

To address the limitation, we enhance the training process using advanced reward calculations. Building on the approach in [30], we utilize cost estimations to optimize BM-tree's performance by refining its original cost function, which we refer to as BM-tree-Impr. As shown in Figure 24, build time improves by 12% on the NORM and SKEW datasets, and up to 27% on the US dataset.

We also find two promising directions to reduce training cost: *incremental learning* and the use of *pre-trained models*. Incremental learning allows a model to update continuously as new data arrives, without retraining from scratch, while pre-trained models provide efficient initialization that reduces the need for long training. In the context of RLESIs, combining these approaches would allow the RL policy to adapt to new insertions and workload shifts by fine-tuning only the affected parts, making them more practical in dynamic settings.
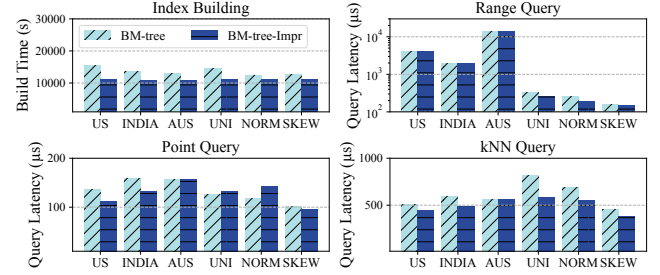

Fig. 24: Comparison between BM-tree and BM-tree-Impr.

VII. CONCLUSIONS

This paper presents the first unified benchmark for evaluating RL-enhanced spatial indices (RLESIs) alongside traditional, advanced, and learned spatial indices. Our framework decouples index training from index building via two modular components, enabling extensible and reproducible evaluation.

To ensure fair comparison, we implement parameter tuning for RLESIs and evaluate 12 representative indices across six datasets and six workloads using a comprehensive set of metrics. We find that while RLESIs can reduce query latency with proper tuning, they do not consistently outperform LSIs or advanced variants in query efficiency or index building time.

We identify key challenges and limitations of RLESIs, including high training overhead and sensitivity to parameter choices. We also address the high training cost for BM-tree by exploring cost-based reward functions and list potential opportunities to accelerate the RLESI tuning efficiency.

Overall, our study suggests that RLESIs are viable for systems requiring backward compatibility, but their practical adoption remains limited by high tuning costs and inconsistent performance across workloads.

VIII. AI-GENERATED CONTENT ACKNOWLEDGEMENT

The authors used ChatGPT in English language editing to improve the fluency, grammar, and clarity of text originally drafted by the authors. In addition, ChatGPT was also employed during code development to support basic syntax and implementation-related tasks. No experimental results or original research contributions were generated solely by AI tools. The authors take full responsibility for the accuracy, originality, and integrity of all content presented in this paper.

REFERENCES

[1] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD*, 1984, pp. 47–57.

[2] 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*, 1990, pp. 322–331.

[3] S. T. Leutenegger, J. M. Edgington, and M. A. López, "STR: A simple and efficient algorithm for R-Tree packing," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 1997, pp. 497–506.

[4] J. A. Orenstein, "Spatial query processing in an object-oriented database system," in *SIGMOD*, 1986, pp. 326–336.

[5] PostGIS. (2024) https://postgis.net/docs/manual-3.5/using_postgis_dbmanagement.html#spgist_indexes. Accessed: 2024-12-27.

[6] Microsoft SQL Document. (2023) https://learn.microsoft.com/en-us/sql/relational-databases/spatial/spatial-indexes-overview?view=sql-server-ver15. Accessed: 2024-12-27.

[7] Oracle Manual. (2024) https://docs.oracle.com/en/database/oracle/oracle-database/23/spatl/creating-spatial-index.html. Accessed: 2024-12-27.

[8] H. Wang, X. Fu, J. Xu, and H. Lu, "Learned index for spatial queries," in *MDM*, 2019, pp. 569–574.

[9] J. Qi, G. Liu, C. S. Jensen, and L. Kulik, "Effectively learning spatial indices," *PVLDB*, vol. 13, no. 11, pp. 2341–2354, 2020.

[10] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *SIGMOD*, 2020, pp. 985–1000.

[11] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "LISA: A learned index structure for spatial data," in *SIGMOD*, 2020, pp. 2119–2133.

[12] J. Li, Z. Wang, G. Cong, C. Long, H. M. Kiah, and B. Cui, "Towards designing and learning piecewise space-filling curves," *PVLDB*, vol. 16, no. 9, pp. 2158–2171, 2023.

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

[14] 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," *PACMMOD*, vol. 1, no. 1, pp. 63:1–63:26, 2023.

[15] J. Yang and G. Cong, "Platon: Top-down r-tree packing with learned partition policy," *PACMMOD*, vol. 1, no. 4, pp. 1–26, 2023.

[16] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska, "Benchmarking learned indexes," *PVLDB*, vol. 14, no. 1, pp. 1–13, 2020.

[17] H. Lan, Z. Bao, J. S. Culpepper, and R. Borovica-Gajic, "Updatable learned indexes meet disk-resident dbms - from evaluations to design choices," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, 2023.

[18] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang, "Are updatable learned indexes ready?" *PVLDB*, vol. 15, no. 11, p. 3004–3017, 2022.

[19] L. Bindschaedler, A. Kipf, T. Kraska, R. Marcus, and U. F. Minhas, "Towards a benchmark for learned systems," in *ICDEW*, 2021, pp. 127–133.

[20] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. Lomet, and T. Kraska, "ALEX: An updatable adaptive learned index," in *SIGMOD*, 2020, pp. 969–984.

[21] STX B+ Tree. (2007) https://panthema.net/2007/stx-btree. Accessed: 2023-05-31.

[22] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *SIGMOD*, 2019, pp. 1189–1206.

[23] P. Ferragina and G. Vinciguerra, "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," *PVLDB*, vol. 13, no. 8, pp. 1162–1175, 2020.

[24] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *SIGMOD*, 2018, pp. 489–504.

[25] J. Wu, Y. Zhang, S. Chen, J. Wang, Y. Chen, and C. Xing, "Updatable learned index with precise positions," *PVLDB*, vol. 14, no. 8, p. 1276–1288, 2021.

[26] A. Kipf, R. Marcus, A. an Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "RadixSpline: A single-pass learned index," in *Proceedings of the International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)*, 2020, pp. 5:1–5:5.

[27] Q. Liu, M. Li, Y. Zeng, Y. Shen, and L. Chen, "How good are multi-dimensional learned indices? an experimental survey," *CoRR*, vol. abs/2405.05536, 2024.

[28] PyTorch. (2016) https://pytorch.org. Accessed: 2023-05-31.

[29] M. Hadjieleftheriou. (2024) libspatialindex. https://github.com/libspatialindex/libspatialindex. Accessed: 2024-05-14.

[30] G. Liu, L. Kulik, C. S. Jensen, T. Li, R. Borovica-Gajic, and J. Qi, "Efficient cost modeling of space-filling curves," *PVLDB*, vol. 17, no. 13, pp. 4773–4785, 2024.

[31] V. Pandey, A. van Renen, A. Kipf, I. Sabek, J. Ding, and A. Kemper, "The case for learned spatial indexes," *CoRR*, vol. abs/2008.10349, 2020.

[32] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *NeurIPS 2019 Workshop on Machine Learning for Systems*, 2019.

[33] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using r-trees," in *SIGMOD*, 1993, p. 237–246.

[34] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, *R-Trees: Theory and Applications*. Springer Publishing Company, Incorporated, 2005.

[35] V. Pandey, A. van Renen, A. Kipf, I. Sabek, J. Ding, and A. Kemper, "The case for learned spatial indexes," *CoRR*, vol. abs/2008.10349, 2020.

[36] N. Beckmann and B. Seeger, "A revised R*-tree in comparison with related index structures," in *SIGMOD*, 2009, pp. 799–812.

[37] L. Arge, M. D. Berg, H. Haverkort, and K. Yi, "The Priority R-tree: A practically efficient and worst-case optimal R-tree," *ACM Transactions on Algorithms*, vol. 4, no. 1, pp. 9:1–9:30, 2008.

[38] A.-A. Abdullah-Al-Mamun, C. M. R. Haider, J. Wang, and W. G. Aref, "The "ai + r" - tree: An instance-optimized r - tree," in *MDM*, 2022, pp. 9–18.

[39] S. Huang, Y. Wang, and G. Li, "Acr-tree: Constructing r-trees using deep reinforcement learning," in *DASFAA*. Springer, 2023, pp. 80–96.

[40] I. Grondman, L. Busoniu, G. A. D. Lopes, and R. Babuska, "A survey of actor-critic reinforcement learning: Standard and natural policy gradients," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 6, pp. 1291–1307, 2012.

[41] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[42] R. FinkelJon and L. Bentley, "Quad trees: A data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[43] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The Grid File: An adaptable, symmetric multikey file structure," *TODS*, vol. 9, no. 1, pp. 38–71, 1984.

[44] J. T. Robinson, "The K-D-B-tree: A search structure for large multidimensional dynamic indexes," in *SIGMOD*, 1981, pp. 10–18.

[45] S. Zhang, S. Ray, R. Lu, and Y. Zheng, "SPRIG: A learned spatial index for range and kNN queries," in *SSTD*, 2021, pp. 96–105.

[46] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, "Tsunami: A learned multi-dimensional index for correlated data and skewed workloads," *PVLDB*, vol. 14, no. 2, p. 74–86, 2020.

[47] V. Pandey, A. van Renen, E. T. Zacharatou, A. Kipf, I. Sabek, J. Ding, V. Markl, and A. Kemper, "Enhancing in-memory spatial indexing with learned search," *CoRR*, vol. abs/2309.06354, 2023.

[48] M. H. Moti, P. Simatis, and D. Papadias, "Waffle: A workload-aware and query-sensitive framework for disk-based spatial indexing," *PVLDB*, vol. 16, p. 670–683, 2022.

[49] J. A. Orenstein and T. H. Merrett, "A class of data structures for associative searching," in *PODS*, 1984, pp. 181–190.

[50] J. K. Lawder and P. J. H. King, "Querying multi-dimensional data index using the hilbert space-filling curve," *SIGMOD Rec.*, vol. 30, no. 1, pp. 19–24, 2001.

[51] I. Kamel and C. Faloutsos, "Hilbert R-tree: An improved R-tree using fractals," in *PVLDB*, 1994, pp. 500–509.

[52] J. Qi, Y. Tao, Y. Chang, and R. Zhang, "Theoretically optimal and empirically efficient R-trees with strong parallelizability," *PVLDB*, vol. 11, no. 5, pp. 621–634, 2018.

[53] A. Davitkova, E. Milchevski, and S. Michel, "The ML-Index: A multidimensional, learned index for point, range, and nearest-neighbor queries," in *EDBT*, 2020, pp. 407–410.

[54] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search," *TODS*, vol. 30, no. 2, pp. 364–397, 2005.

[55] J. Gao, X. Cao, X. Yao, G. Zhang, and W. Wang, "Lmsfc: A novel multidimensional index based on learned monotonic space filling curves," *PVLDB*, vol. 16, no. 10, pp. 2605–2617, 2023.

[56] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *International Conference on Learning and Intelligent Optimization*, 2011, pp. 507–523.

[57] 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*, vol. 45, no. 3, 2020.

[58] OpenStreetMap Stats. (2024) https://www.openstreetmap.org/stats/data_stats.html. Accessed: 2024-07-22.

[59] H. Lan, Z. Bao, J. S. Culpepper, R. Borovica-Gajic, and Y. Dong, "A fully on-disk updatable learned index," in *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 2024, pp. 4856–4869. [Online]. Available: https://doi.org/10.1109/ICDE60146.2024.00369