LASE: A Learned Spatial Index for Dynamic Workloads

Pan Cai[1,2], Chaohong Ma[2], Zongze Lu[1,2], Peiran Liu[1,2], Pengju Liu[1,2], Cuiping Li[1,2(✉)], and Hong Chen[1,2]

[1] Key Laboratory of Data Engineering and Knowledge Engineering, Renmin University of China, Beijing, China
[2] School of Information, Renmin University of China, Beijing, China
{2020000878,chaohma,licuiping}@ruc.edu.cn

Abstract. Recently, several learned spatial indexes have been proposed to improve fast spatial queries, however, they are ineffective when facing highly dynamic scenarios with different read/write ratios. In this paper, we propose LASE, a Learned spatial index for dynamic workloads. We first empirically demonstrate that the poor performance for dynamic workloads comes from high-cost model training. Therefore, LASE takes advantage of both the traditional indexing techniques and the learned indexes to reduce model retraining costs. Specifically, LASE designs effective structural modification operations to support update operations and develops efficient query algorithms. Two aspects are considered in reducing update costs: 1) reduce the scale of model retraining; 2) achieve better locality to accelerate the query performance within leaf nodes. We conduct extensive experiments on various dynamic workloads and different datasets, and the experimental results demonstrate that LASE achieves high-performance updates and queries compared with applicable competitors.

Keywords: Learned index · Spatial index · Dynamic workloads

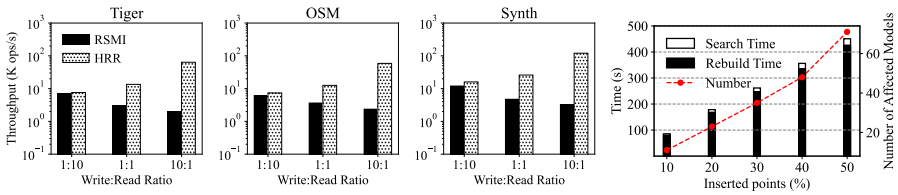
1 Introduction

Spatial data and queries are now ubiquitous, driven by the widespread use of location-based services like Google Maps, which generate vast amounts of spatial data enriched with detailed location information. To efficiently process spatial data and queries, various tree-based spatial indexing techniques, such as the R-tree [10]and its variants [2,25], have been developed. However, in the era of big data, the rapid growth of spatial data volume causes these indexes to expand into large tree structures, leading to costly updates and degraded query performance in dynamic workloads due to extensive structural reorganization [19].

Recently, several advanced learned spatial indexes [27,4,24,18,20] have been designed for fast spatial queries, which adopt the idea of one-dimensional learned indexes [13], replacing traditional indexing structures with machine learning models by learning a cumulative distribution function (CDF) to map a search

key to a position in an ordered array. Learned indexes improve query efficiency while achieving lower memory overhead than traditional indexes like R-tree [10]. Despite advancements, existing learned spatial indexes still have several limitations: (1) Most approaches [27,4,20] can only handle static queries and lack support for update operations, a critical limitation that renders learned spatial indexes unsuitable for dynamic, read-write workloads commonly encountered in practice. (2) The few methods [18,24] that support updates are ineffective in highly dynamic workloads due to the expensive update costs.

For example, RSMI [24] is an efficient learned spatial index that facilitates fast spatial query processing compared with the advanced traditional spatial index HRR [23] (a variant of R-tree) and can support update operations. We empirically evaluate RSMI and HRR in dynamic workloads with varying ratios of insertion and window queries on three datasets (described in Section 5), the results are presented in Fig. 1(a). As observed from the results, RSMI performs worse than HRR, particularly with higher insertion ratios. The throughputs of RSMI decline as the insert ratio increases. To further investigate why RSMI exhibits poor performance with frequent updates, we conduct another experiment to analyze its insertion performance within a write-only workload (without any query operations) in Fig. 1(b). The results clearly demonstrate that the rebuild time (i.e., model retraining) is the dominant factor in the overall cost of RSMI, accounting for approximately 92.46% of the total time. Moreover, it is consistent with the number of affected models, which further proves that model retraining degrades the performance of RSMI in dynamic workloads.



(a) Throughput comparison (b) Insertion cost

Fig. 1: Insertion performance analysis of RSMI.

To address the above limitations, we propose LASE, a Learned Spatial Index efficiently supports dynamic workloads with different read/write ratios. LASE is a tree structure, consisting of inner layers and a leaf layer. The pivotal design of LASE is to reduce the update costs by minimizing the overhead of model retraining while maintaining efficient query performance. Specifically, our approach benefits from both traditional indexing techniques and the learned indexes. For dynamic updates, we use an in-place insertion method, reserving free space in the leaf layer to accommodate new data. To reduce the overhead of model retraining during updates, LASE introduces lightweight structure modification operations (SMOs, described in Section 3.1). For query efficiency, we reorganize the under-

lying data at the leaf layer and use model-based insertion/searching to avoid the last mile search overhead caused by the inherent inaccuracy of the model. Furthermore, we develop efficient algorithms to support fast query processing, including window queries and KNN queries.

In summary, the main contributions are as follows:

(1) We propose LASE, a learned spatial index that effectively supports dynamic workloads, combining the strengths of both traditional and learned indexes.

(2) We design efficient insertion procedures and develop lightweight SMOs to support structural updates in LASE.

(3) We provide optimized query processing algorithms based on LASE to effectively answer window and KNN queries.

(4) We conduct extensive experiments on both real-world and synthetic data under dynamic workloads, demonstrating the superiority of LASE.

The rest of the paper is organized as follows. Section 2 provides an overview of LASE. Section 3 describes its update process. The query processing based on LASE is outlined in Section 4. Section 5 presents the experimental evaluation, followed by a review of related work in Section 6. Section 7 concludes the paper.

2 LASE Overview

Given a set of n points, $P = \{p_1, p_2, \ldots, p_n\}$, all situated in a d-dimensional Euclidean space, our goal is to build LASE on P to efficiently support dynamic updates and spatial queries, which involve insertion, window queries, and KNN (k-nearest neighbor) queries. We concentrate on 2-dimensional points to facilitate a focused and detailed analysis. It is important to note, however, that the methodologies we propose have the potential for wider applicability in $d \geq 3$.

2.1 Design Principles

Based on the observations in Fig. 1, we outline three guiding principles for the design of LASE:

P1. Model-Based Operations (Insert and Query). The larger leaf nodes can reduce tree height. Model-based operations help LASE efficiently locate the data associated with a search key, avoiding the full scan in leaf nodes.

P2. Lightweight Structural Modification Operations (SMOs). When data distribution shifts, SMOs in existing learned approaches require costly retraining of a large number of affected models to maintain query efficiency. LASE needs to minimize the scale of affected models to reduce training costs.

P3. Better search performance. Existing learned spatial indexes suffer from last-mile search overhead caused by inaccurate model prediction. LASE should provide effective strategies to facilitate last-mile search.

2.2 Design Highlights

LASE designs judicious structures to meet the above principles, which consist of the leaf layer and inner layers. First, we employ large-capacity leaf nodes and apply linear models to organize the data in the leaf layer, leading to model-based insertion and lookup (**P1**). Secondly, the inner layers are model-free, constructed without using models, and the large-capacity leaf nodes reduce the overall number of leaf nodes. As a result, model retraining during updates is confined to the leaf layer and with a limited number of models, rather than the entire index structure, effectively reducing SMOs costs (**P2**). Finally, to maintain query efficiency, we use the space-filling curves at the leaf level to map spatial data into one-dimensional key-value pairs, followed by the Fastest Minimum Conflict Degree (FMCD) algorithm [28] to construct learned models based on these pairs. This approach preserves spatial locality in the one-dimensional space, eliminates last-mile search overhead, and accelerates search performance within the leaf nodes (**P3**).

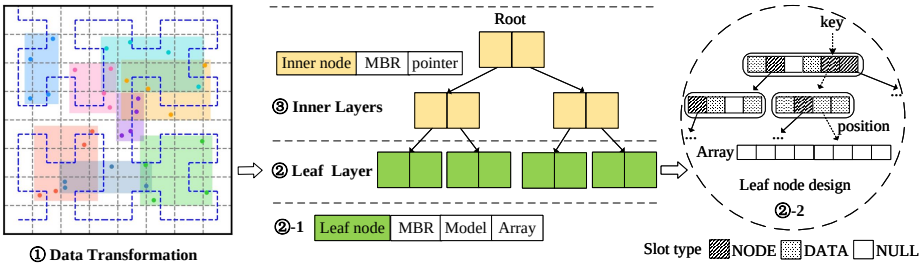


Fig. 2: LASE Index Design Framework

2.3 Index Structure

Fig. 2 illustrates the framework of the LASE index, which consists of three main components: data transformation, inner layer layout, and leaf layer layout. The construction process of LASE is as follows:

Data Transformation. The learned index [13] requires the underlying data to be sorted. However, unlike one-dimensional data, which can be directly sorted and indexed, spatial data do not exhibit the natural order. The advanced space-filling curves, such as Hilbert [6] or Z-order curves [22], which are commonly used in studies like [27,24]. These curves can linearize spatial data while maintaining better proximity. Due to the computational efficiency of the Hilbert curve, we utilize it to convert spatial data into one-dimensional curve values and sort the data accordingly in LASE, as described in Fig. 2 ①.

Leaf layer layout. The leaf layer consists of multiple packed leaf nodes, each with a larger capacity that includes linear models, a Minimum Bounding

Rectangle (MBR) that encloses all contained points belonging to this leaf, and an array of predifined-and-fixed-size, as shown in Fig. 2 ②-1. The sorted points based on Hilbert curve values are sequentially packed into these fixed-size arrays. Within each array, key-value pairs are generated based on the curve value of each point and its position in the array, represented as $\langle key, pos \rangle$. We apply the FMCD algorithm [28] on these key-value pairs to produce optimized learning models for each array. FMCD aims to generate a linear model that minimizes the maximum number of keys assigned to the same slot, i.e., the minimum conflict degree. There are three types of slots (as displayed in Fig. 2 ②-2): DATA, NULL, and NODE. DATA slots hold key-value pairs, NULL slots provide reserved gaps for future insertions, and NODE slots contain pointers to child nodes that manage conflicting key-value pairs. To improve query efficiency, we reorganize the key-value pair layout by placing them in optimal positions according to the model's predictions, thus eliminating the last mile search overhead due to model errors.

Inner layer layout. LASE employs a bottom-up bulk-loading approach to construct the inner layer (as shown in Fig.2 ③). In this process, all leaf nodes in the leaf layer are grouped into fixed-sized inner nodes according to the order of the stored points. This packing continues recursively until a root node (Root in Fig.2) is formed. Each inner node contains the MBR covering all child nodes and pointers to these child nodes. The inner nodes are model-free and utilize R-tree-like traversal operations along with MBRs for searches within the inner layers. To accelerate searches, these inner nodes are designed with a smaller capacity than the leaf nodes.

3 Handling Update

In this section, we first introduce the lightweight SMOs in LASE that support insertion, followed by a description of the LASE insertion process.

3.1 Structural Modification Operations.

As shown in Fig.1(b), The inefficiency of updating learned spatial indexes in dynamic workloads is primarily due to frequent and costly model retraining. To address this, we design lightweight SMOs based on the following strategies:

(1) Limited Model Scope. LASE restricts model usage to the leaf layer, leaving the inner layers model-free, rather than using learned models across all layers from leaf to root as in RSMI [24].

(2) High-Capacity Leaf Nodes. LASE uses larger-capacity leaf nodes to reduce the number of inner and leaf nodes, as well as to decrease the height of the index structure. Additionally, free space is reserved in leaf nodes to accommodate future insertions. During the initial construction of LASE, only half of each leaf node's capacity is filled with point data, while the inner nodes are fully populated. Model retraining is triggered only when a leaf node meets specific conditions, thereby further reducing the frequency of high-cost model retraining.

(3) Simplified Model Design. To lower model training costs, LASE replaces RSMI's complex multilayer perception (MLP) with a linear function, which is more efficient and cost-effective, and further reduces the overhead of structural modification operations.

3.2 Insertion process

The detailed insertion process is shown in Algorithm 1. Given an insertion point p, a tree traversal operation is first performed to locate the leaf node L to which point p belongs, and calculate the Hilbert curve value key of point p (line 1). If the amount of data contained in the leaf node L is less than the predefined threshold B, that is, the leaf node has free space, then point p is inserted into the end of the array within the leaf node L (line 3), and a key-value pair $\langle key, pos \rangle$ is constructed using the Hilbert value of p and the pos in the array. The model within the leaf node L then inserts the key-value pair $\langle key, pos \rangle$ into the predict position (line 4). When querying a key, use the model to locate key-value pair $\langle key, pos \rangle$. Then, the corresponding records from the array are retrieved using the pos, implementing model-based insert and query. If the amount of data contained in the leaf node is equal to B, then a splitting process is triggered (line 11). The value of threshold B is determined by the capacity of the leaf node (as discussed in Section 5 on the impact of leaf node capacity).

Algorithm 1: Insert Processing

```
   Input: A point data p, predefined threshold B
 1 L ← Find(p, Node), key ← get_hilbert_value(p);
 2 if L.capacity < B then
 3     L_array.add(p);
 4     e ← L_model.predict(key, pos);
 5     if e.type == NULL then
 6         e ← insert(key, pos);
 7     else if e.type == DATA then
 8         Create a new node n and set e.type ← NODE ;
 9         insert(key, pos) into n;
10 else
11     left, right ← splitleafnode(L);
12     right_array.add(p);
13     Sort points within left_array, right_array;
14     for p_i in left, i = 0, 1, ..., L_array.length − 1 do
15         train_left_model(⟨ key_i, i⟩, B);
16     for p_j in right, j = 0, 1, ..., L_array.length − i do
17         train_right_model(⟨ key_j, j⟩, B);
```

When using a model to predict the correct position of the key-value pair $\langle key, pos \rangle$ corresponding to p, the insertion process falls into three cases, determined by the slot type: NULL, DATA, or NODE. 1) if we encounter a NULL slot, we directly insert $\langle key, pos \rangle$ into that predicted position (lines 5-6). 2) The encountered slot is a DATA slot, we create a new node n at that position to store the conflicting key-value pairs (lines 7–9). 3) If encountering a NODE slot, LASE will access the child node pointed to by it and repeat the above process, using the child node's model to re-predict the position of $\langle key, pos \rangle$.

If the splitting process is triggered by a full leaf node, we evenly divide them into left and right leaf nodes based on the amount of data they contain (line 11). The new data point p is inserted into the right leaf node (line 12). As a result of the split, the distribution of data within the leaf nodes is modified. Therefore, the FMCD algorithm is used to retrain the linear models for the two new leaf nodes (see Section 2 for the leaf node layout). The algorithm's input includes the sorted key-value pairs corresponding to the data points within each leaf node and the predefined size B of the leaf node (lines 13–17).

The splitting process of leaf nodes propagates upward. If there is free space in the inner node, directly insert the newly created leaf node. If the inner node is full, execute a splitting operation. If the inner node is the root node, increment the height of the index by one level and create a new root node. To control the growth of the index height, LASE assigns a larger capacity to leaf nodes and reserves half of the available space in advance to accommodate new data. After insertion, the MBR information of the leaf nodes and their parent nodes is updated accordingly.

Insertion cost. Given the height of LASE (i.e., the number of layers) H and the leaf capacity B, the overall insertion cost is $O(H + \log^2 B)$, where $\log^2 B$ represents the at most amortized cost for adjusting the leaf node (which is not triggered by each insertion) [28].

4 Query Processing

This section provides the algorithms to process window and KNN queries based on LASE, which are the most typical spatial queries in practice. The process of point query is omitted, as the point is a basic step in window and KNN queries.

4.1 Window Queries

Given a spatial dataset D, a window query of the form $Q(q_{xl}, q_{xh}, q_{yl}, q_{yh})$, with q_{xl}, q_{xh}, and q_{yl}, q_{yh} being as the lower and upper bounds on the x and y dimensions respectively. The window query aims to return a set of points $p_i(x_i, y_i)$ contained in Q, satisfying $q_{xl} \leq x_i \leq q_{xh}$ and $q_{yl} \leq y_i \leq q_{yh}$, where $p_i(x_i, y_i) \in D$, $i = 1, 2, \ldots, m$, and $m \leq |D|$. The window query process is illustrated in Algorithm 2, including three steps: mapping, refining, and scanning.

Mapping. LASE calculates the boundaries $(low, high)$ of the query window Q based on the Hilbert curve, which are one-dimensional values. Due to the

non-monotonic nature of the Hilbert curve [6], we cannot directly determine the one-dimensional boundary values using the lower-left point (q_{xl}, q_{yl}) and the upper-right point (q_{xh}, q_{yh}) of the query window. Instead, we calculate the Hilbert curve values for the four corner points of the query window and select the minimum and maximum values $(low, high)$ as the approximate boundary values in the one-dimensional key space (line 2).

Refining. LASE locates the leaf nodes that intersect with the query window Q through tree traversal operations (lines 3-8). LASE utilizes the boundary values $(low, high)$ of the query window in one-dimensional space as input to refine the scan range of point data within the current leaf node that determines the *start* and *end* positions (line 10). Specifically, the model performs two predictions, returning the first position in the leaf node array that is greater than or equal to the boundary values.

Scanning. Finally, we directly scan all points between *start* and *end* in the ordered array within the leaf nodes, adding the points contained within the query window Q to *result* (lines 11-13).

Algorithm 2: Window Query Processing

```
   Input: A query window Q
   Output: result:the set of points in Q
 1 result ← {}, queue ← {};
 2 low, high ← get_hilbert_value(qxl, qxh, qyl, qyh);
 3 queue ← root.node;
 4 while queue is not empty do
 5 |   node ← queue.pop();
 6 |   if node is not a leaf node then
 7 |   |   if node.mbr intersects with Q.mbr then
 8 |   |   |   queue.add(node);
 9 |   else
10 |   |   (start, end) ← model_predict(low, high);
11 |   |   for p_i in array(start, end) do
12 |   |   |   if Q contains p_i then
13 |   |   |   |   result.add(p_i);
14 return result;
```

Query cost. The cost of window query is $O(H + l + \log B)$, including the cost of mapping, refining, and scanning, where l is the number of scanned points.

4.2 KNN Queries

Given a spatial point $q(x, y)$ and an integer k, the KNN query returns the k points $p_i(x_i, y_i)$ in dataset D that are closest to q, satisfying $\forall o_j(x_j, y_j) \in$

$D \mid \text{dist}(o_j, q) \geq \text{dist}(p_i, q)$ for $i = 1, 2, \ldots, k$ and $j = |D| - i$. LASE handles the KNN query by transforming it into a series of window queries to find the k data points closest to the query point q, which is similar to RSMI [24]. Algorithm 3 describes the details of the KNN query process.

We maintain an array *result* to store the points returned from each window query and a priority queue *queue* to keep the k closest points, ordered by their distance to the query point q (line 1). The algorithm first constructs a square as the initial query window Q_{knn} based on the query point q, the edge length *side* of this initial query window is determined by the value of k and the size of the dataset (line 3). Then, the window query Algorithm 2 is executed to return candidate points, which are added to *result*, and checks if the number of data points in *result* meets or exceeds k. If false, the query window needs to keep expansion (line 7). To accelerate the KNN query, the algorithm clears the candidate points of the previous window query from the *result* after each expansion (line 6), ensuring that the *result* only contains the data points within the current query window. If true, the algorithm stops expanding, retrieves the k points closest to the query point q from the array *result*, organizes them into the *queue* in ascending order based on their distance from q, and finally returns the *queue* (lines 9–12).

Algorithm 3: KNN Query Processing

Input: A query point q, an integer k, dataset size $|D|$
Output: *queue*: contains k nearest points p_i of q

1 $queue \leftarrow \{\}$, $result \leftarrow \{\}$;
2 $side \leftarrow \sqrt{\frac{k}{|D|}}$;
3 Initialize $Q_{knn} \leftarrow \{q.x - \text{side}, q.x + \text{side}, q.y - \text{side}, q.y + \text{side}\}$;
4 $result \leftarrow$ Window query(Q_{knn});
5 **while** *result.size()* $< k$ **do**
6 $\quad result \leftarrow \{\}$;
7 $\quad side \leftarrow side * 2$ and update Q_{knn};
8 $\quad result \leftarrow$ Window query(Q_{knn});
9 **if** *result.size()* $\geq k$ ***and*** *queue.size()* $< k$ **then**
10 $\quad$ **foreach** $p_i \in result$, *sorted by* $dist(p_i, q)$ **do**
11 $\quad\quad queue \leftarrow \{p_i, k\}$;
12 **return** *queue*;

Query Cost. The cost of the KNN query is dominated by the expanded window queries. Assuming the number of expansions is t, the overall cost of the KNN query is $tO(H + l + \log B) + O(m)$, where m represents the number of points returned by the last expanded window query.

5 Experiments

In this section, we empirically evaluate the performance of LASE and its components using real and synthetic datasets with various workloads.

5.1 Experiment Setup

The LASE implementation is in C++. All experiments were performed on a 64-bit Ubuntu 20.04 Linux server with Intel(R) Xeon(R) CPU and 250 GB RAM.

Baselines. We compare LASE against the applicable competitors that support indexing spatial data and dynamic workloads, including the traditional spatial index HRR and learned spatial indexes RSMI[24], ZM[27], and ML[4]. HRR[23] is a more advanced variant of the original R-tree [10] that employs rank space techniques and Hilbert curves, providing a bottom-up bulk-loading method, and supporting efficient queries and updates. The original ZM and ML only support static queries. To make a fair comparison, we adapt ZM and ML to support the dynamic setting using the same approach in [24]. All baselines are implemented using open source code[17] provided in [24], which are all in C++.

The capacity of all nodes in HRR is set to 100, the capacity of leaf nodes in RSMI is set to 100, the partition capacity is set to 10000, the capacity of inner nodes in LASE is also set to 100, and the capacity of leaf nodes is set to 2000 through experiments (which will be explained later in Section 5.2 about the impact of leaf node capacity).

Table 1: Dataset characteristiecs

Dataset	The number of points	Dataset size
Tiger	16,002,334	240.79MB
OSM	69,431,797	1.02GB
Synth	100,000,000	1.49GB

Datasets. We use three distinct datasets including two real-world datasets and one synthetic dataset, which contain no duplicate point data. Two real datasets are Tiger and OSM, respectively, which have complex distributions. The Tiger, representing geographic features of 18 eastern U.S. states, contains 17 million rectangles, which we use as point data by employing the centers of each rectangle [26]. OSM is derived from the OpenStreetMap [9]. For ease of experimentation, we preprocessed the real dataset by normalizing values to the range (0,1) and removing duplicate points. The synthetic dataset (short as Synth), contains 100 million data points and exhibits a skewed distribution. Table 1 lists the characteristics of datasets.

Workloads. In our experiment, we generate workloads using the specified datasets with varying read/write ratios. We considered three types of dynamic workloads: Read-Heavy (RH, 10 queries per insertion), Equal-Heavy (EH, an

equal number of insertions and queries), and Write-Heavy (WH, 10 insertions per query). Additionally, we generate a static query workload, Read-Only (RO), which consists solely of query operations.

We sampled 1 million data from each of the three datasets Tiger, OSM, and Synth to generate all types of workloads with different read/write ratios. The queries consist of window and KNN queries, we conduct further experiments to evaluate the influence of the window ratio and k. For window queries, we apply window ratio (i.e., the area of the queried window / the whole area of the entire data) to generate query windows. The window ratios involve 0.0006%, 0.0025%, 0.01%, 0.04%, and 0.16%, from sparse to dense. For KNN queries, we assess the performance with different values of k: 1, 5, 25, 125, and 625. In other experiments, we use window ratio = 0.01% and $k = 25$ as the defaults.

Below, we first conducted experiments to evaluate the overall performance of LASE and the baseline methods, as well as the index construction time and storage overhead. Next, we investigate the impact of different query window ratios in window queries and the number of k in KNN queries. Finally, we analyze how the leaf node capacity of LASE affects its performance.

5.2 Experimental Results

This section reports the detailed experimental results.

Overall Performance. The experiments compare the average throughput of LASE and baselines under dynamic workloads(RH, EH, and WH), as shown in Fig. 3-4, as well as the average query response time in static query workload RO, as demonstrated in Fig. 5-6.

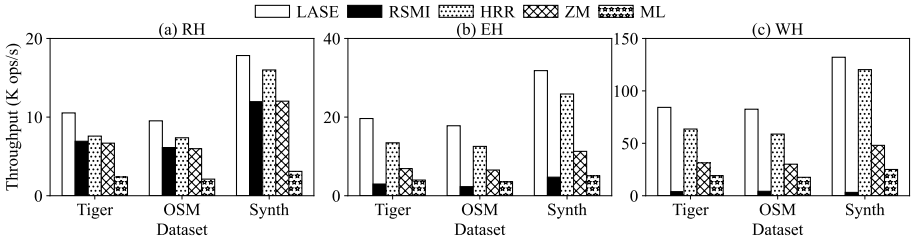


Fig. 3: Throughput in dynamic workloads with window queries

We observed from Fig. 3 and Fig. 4 that: (1) LASE achieves higher throughput than other baselines in both real and synthetic datasets under dynamic workloads, with particularly strong performance evident in Fig. 4. (2)Due to the substantial overhead of model retraining, learned spatial indexes experience significant update delays, with RSMI, ZM, and ML generally exhibiting lower average throughput than HRR. However, LASE utilizes lightweight structural

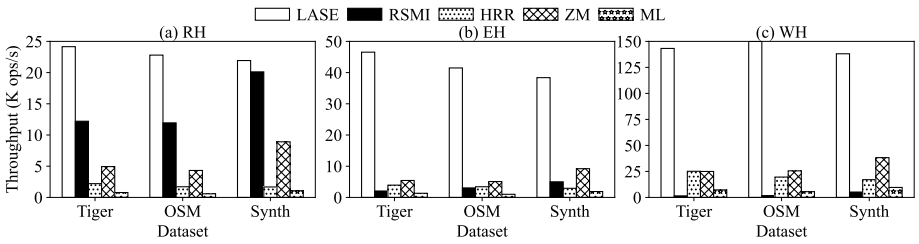


Fig. 4: Throughput in dynamic workloads with KNN queries

modification operations, significantly reducing retraining overhead and delivering exceptional performance. (3) RSMI performs worse than ZM and ML under dynamic workloads, particularly as the insertion ratio increases. In contrast, LASE maintains relatively stable throughput across all three dynamic workloads. The main reason is that RSMI has a larger number of models than ZM and ML, thus affecting and requiring more models to be retrained. LASE, on the other hand, restricts the influence of high-cost model retraining to the leaf level. Furthermore, LASE uses high-capacity leaf nodes and reserved gaps to reduce the frequency of model retraining.

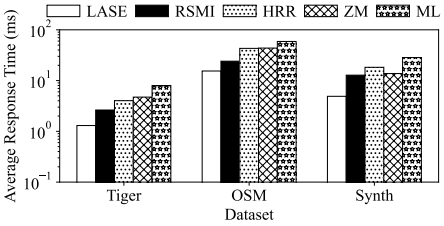


Fig. 5: Static workload (window query)

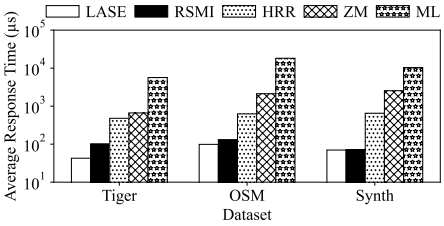


Fig. 6: Static workload (KNN query)

Next, we conduct experiments on the window and KNN queries to analyze the performance of all indexes on static query workloads. As the query time indicated in Fig. 5 and Fig. 6, LASE demonstrates the lowest query latency compared with baselines across all datasets. Specifically, the window query of LASE is 1.83 × faster than that of the suboptimal RSMI index and 3.02 × faster than HRR. For KNN queries, LASE achieves 1.58 × and 8.96 × less query time than RSMI and HRR, respectively, and at least one order of magnitude faster than ZM and ML. The primary advantage of LASE is its model-based insertion approach at the leaf nodes, which effectively avoids the last-mile search overhead typically caused by errors in learned models. Additionally, for KNN queries, LASE only needs to examine candidates within the matched query window derived from the KNN query, reducing access to irrelevant data compared to baseline methods.

Table 2: Index Construction Time and Index Size

Metric	Dataset	**LASE**	**RSMI**	**HRR**	**ZM**	**ML**
Building Time (s)	Tiger	20.80	7480.25	19.08	1512.45	1463.26
	OSM	93.96	35637.66	87.57	6281.42	6533.51
	Synth	136.80	44321.26	130.11	9009.49	9091.56
Index Size (MB)	Tiger	431.87	398.58	863.44	371.18	371.18
	OSM	2147.87	1707.47	3745.95	1611.71	1611.71
	Synth	3698.58	2440.58	5395.28	2319.34	2319.34

Construction time and index size. We next compare the construction time and index size of all methods, with the results presented in Table 2. We observe that LASE achieves a construction time of 1 to 2 orders of magnitude faster than learned indexes (RSMI, ML, ZM) and is comparable to the build speed of HRR. In the index size, the advantages of LASE in index build time and handling dynamic workloads come at the cost of slightly higher memory overhead compared to learned spatial indexes. However, in contrast to HRR, LASE efficiently saves 40% to 50% in memory usage by utilizing high-capacity leaf nodes and learned models at the leaf layer, which reduces the height and number of inner nodes in the inner layer.

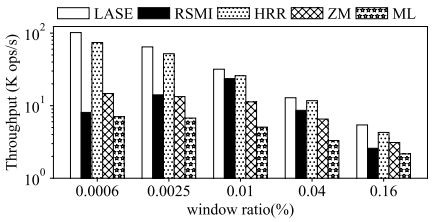


Fig. 7: Window ratio in EH workload

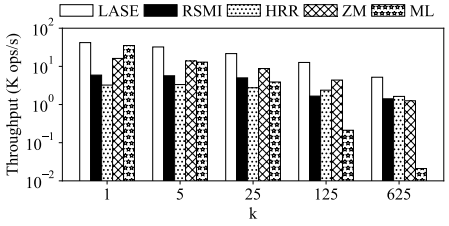


Fig. 8: k in EH workload

Vary window ratio and k. We evaluate the impact of varying query window ratios and parameter k on LASE performance under dynamic workloads. Due to space constraints, we report results only for the EH workload on the Synth dataset. As shown in Fig. 7, the average throughput of all indexes decreases as the window ratio increases. This is primarily due to the increasing number of data points queried as the window ratio grows in window queries. For HRR and LASE, this requires traversing more nodes to locate the target data, while for RSMI, ZM, and ML, it involves scanning and processing a larger data volume. Nonetheless, LASE consistently outperforms all baselines. Fig 8 illustrates the effect of varying k values on indexing performance, revealing trends similar to those observed in Fig 7.

Table 3: Configuration of Leaf Node Capacity

Leaf Node Capacity	1000	**2000**	3000	4000	5000
Construction Time (s)	12.621	12.558	12.924	12.922	12.695
Index Size (MB)	3.638	3.189	2.959	3.054	3.025
Window Query Time (ms)	0.384	0.363	0.416	0.406	0.414
KNN Query Time (ms)	0.029	0.034	0.045	0.053	0.058

Impact of leaf capacity. This experiment studies the effect of leaf node capacity on LASE performance on the Synth dataset with RO workload, and the results are shown in Table 3. The intuition is that a larger leaf capacity can reduce the index height, thus enhancing query efficiency. However, as observed from Table 3, the LASE performance does not improve consistently as the leaf capacity increases. Specifically, in most cases, the performance of KNN and window queries declines, and the index construction time becomes longer. Meanwhile, the capacity of leaf nodes has less influence on index size. Across various metrics, a leaf capacity of 2,000 demonstrates better overall efficiency and is thus chosen as the default setting for LASE.

6 Related Work

This section discusses related work, including traditional and learned approaches to spatial indexes.

6.1 Traditional Spatial Indexes

Spatial indexes are typically tree structures, with leaf nodes storing point data and inner nodes containing pointers to the child nodes and minimum bounding rectangles (MBRs) encompassing their children. They are broadly categorized as space or data partitioning. Space partitioning methods, such as grid file [21], quadtree [7], KD-tree [3] et al, divide space until each partition holds a specified number of points. For data partitioning, the R-tree family is the primary category, which constructs an index structure by partitioning data into nodes. R-tree performance is highly dependent on the organization of the data. There are two organization strategies: 1) Individual insertion method, e.g., R*-tree [2], R+-tree [25], RR*-tree [1], enhances query performance by minimizing node overlap regions during insertion. However, this method often causes a poor R-tree structure with bad query performance. 2) Bulk-loading manner, primarily focuses on a bottom-up strategy, where point data is recursively packed into parent nodes based on a heuristic approach. The work [12] bulk-loading uses Z-order or Hilbert curve sorting to preprocess data, then packs it sequentially into leaf nodes. The STR-tree [15] is sorted by x-coordinate first, then y-coordinate within subsets. HRR [23] transforms data into rank space and employs the Hilbert curve to compactly organize data within leaf nodes, maintaining better spatial proximity and improving query performance.

6.2 Learned Spatial Indexes

Several learned spatial indexes [14,27,4,20,18,24] have been proposed. Unlike one-dimensional data, spatial data lacks natural order, making it challenging to directly learn its cumulative distribution function (CDF). Most existing methods project spatial data into one-dimensional values, and then apply one-dimensional learned indexes. SageDB [14] employs a grid to partition data, selects one dimension to sort the data, and subsequently learns the mapping relationship between the sorting dimension and the sorting order. The ZM index [27] utilizes the Z-order curve to sort data points and learn the CDF. ML index [4] employs iDistance [11] to map point data to a distance value relative to a reference point and learn the CDF. RSMI [24] utilizes rank space technology and Hilbert curves to sort spatial data. LISA [18] is a disk-based learned spatial index that employs grids and Lebesgue measures to partition and sort spatial data, aiming to reduce I/O costs. In contrast, our method prioritizes optimizing CPU time in memory. Flood [20] optimizes grid layout by learning from workloads and. LMSFC [8] and BMTree [16] differ from the above work in that they mainly focus on optimizing projection methods to improve the query performance of learned spatial indexes by optimizing spatial filling curves. The above research mainly focuses on static queries, neglecting dynamic updates. Although LISA [18] and RSMI [24] allow updates, their effectiveness is hampered by costly model retraining. LMSFC [8] uses ALEX [5] for updates but assumes access to historical query workload information to optimize index, which is often unrealistic due to a lack of prior knowledge about query workloads. Our method efficiently supports dynamic workloads without requiring historical query information.

7 Conclusion and future work

We propose LASE, a learned spatial index to address the challenges arising from dynamic workloads. LASE benefits from both traditional and learned techniques to reduce the costly model retraining, caused by dynamic insertions. The experiments demonstrate the superiority of LASE in efficiently dynamic updates and fast query processing. For future work, we aim to explore learned spatial indexes for dynamic workloads with a focus on minimizing I/O costs on disk.

Acknowledgments. This work was supported by the National Key Research and Development Plan (No. 2023YFB4503600) and NSFC (Nos. U23A20299, U24B20144, 62172424, 62276270, 62322214).

References

1. Beckmann, N., Seeger, B.: A revised r*-tree in comparison with related index structures. In: Proceedings of SIGMOD. pp. 799–812 (2009)
2. Beckmann, N., et al.: The r*-tree: An efficient and robust access method for points and rectangles. In: Proceedings of SIGMOD. pp. 322–331 (1990)

3. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Communications of the ACM. **18**(9), 509–517 (1975)
4. Davitkova, A., et al.: The ml-index: A multidimensional, learned index for point, range, and nearest-neighbor queries. In: Proceedings of EDBT. pp. 407–410 (2020)
5. Ding, J., et al.: Alex: an updatable adaptive learned index. In: Proceedings of SIGMOD. pp. 969–984 (2020)
6. Faloutsos, C., Roseman, S.: Fractals for secondary key retrieval. In: Proceedings of PODS. pp. 247–252 (1989)
7. Finkel, R.A., et al.: Quad trees a data structure for retrieval on composite keys. Acta Informatica. pp. 1–9 (1974)
8. Gao, J., et al.: Lmsfc: A novel multidimensional index based on learned monotonic space filling curves. Proc. VLDB Endow. **16**(10), 2605–2617 (2023)
9. Geofabrik: Openstreetmap us northeast data dump (2018), `https://download.geofabrik.de/`, accessed: 2020-06-10
10. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: Proceedings of SIGMOD. pp. 47–57 (1984)
11. Jagadish, H.V., et al.: idistance: An adaptive b^+-tree based indexing method for nearest neighbor search. ACM TODS. **30**(2), 364–397 (2005)
12. Kamel, I., et al.: On packing r-trees. In: Proceedings of CIKM. pp. 490–499 (1993)
13. Kraska, T., et al.: The case for learned index structures. In: Proceedings of SIGMOD. pp. 489–504 (2018)
14. Kraska, T., et al.: Sagedb: A learned database system. In: Proceedings of CIDR (2019)
15. Leutenegger, S.T., et al.: STR: A simple and efficient algorithm for r-tree packing. In: Proceedings of ICDE. pp. 497–506 (1997)
16. Li, J., et al.: Towards designing and learning piecewise space-filling curves. Proc. VLDB Endow. **16**(9), 2158–2171 (2023)
17. Li, L.: Rsmi: Remote sensing mapping index (2021), `https://github.com/Liuguanli/RSMI/tree/rsmi_mr`, accessed: 2020-06-10
18. Li, P., et al.: Lisa: A learned index structure for spatial data. In: Proceedings of SIGMOD. pp. 2119–2133 (2020)
19. Moti, M.H., et al.: Waffle: A workload-aware and query-sensitive framework for disk-based spatial indexing. Proc. VLDB Endow. **16**(4), 670–683 (2022)
20. Nathan, V., et al.: Learning multi-dimensional indexes. In: Proceedings of SIGMOD. pp. 985–1000 (2020)
21. Nievergelt, J., et al.: The grid file: An adaptable, symmetric multikey file structure. TODS. **9**(1), 38–71 (1984)
22. Orenstein, J.A., et al.: A class of data structures for associative searching. In: Proceedings of PODS. p. 181 (1984)
23. Qi, J., et al.: Theoretically optimal and empirically efficient r-trees with strong parallelizability. Proc. VLDB Endow. **11**(5), 621–634 (2018)
24. Qi, J., et al.: Effectively learning spatial indices. Proc. VLDB Endow. **13**(12), 2341–2354 (2020)
25. Sellis, T., et al.: The r+-tree: A dynamic index for multi-dimensional objects. VLDB (1987)
26. U.S. Census Bureau: Tiger/line shapefiles (2006), `https://www.census.gov/geo/maps-data/data/tiger-line.html`, accessed: 2020-06-10
27. Wang, H., et al.: Learned index for spatial queries. In: 20th IEEE International Conference on MDM. pp. 569–574. IEEE (2019)
28. Wu, J., et al.: Updatable learned index with precise positions. Proc. VLDB Endow. **14**(8), 1276–1288 (2021)