Efficiently Learning Spatial Indices

Guanli Liu[1], Jianzhong Qi[1†], Christian S. Jensen[2], James Bailey[1], Lars Kulik[1]
[1]*The University of Melbourne*, [2]*Aalborg University*
{*guanli.liu1, jianzhong.qi, baileyj, lkulik*}*@unimelb.edu.au, csj@cs.aau.dk*

***Abstract*—Learned indices can leverage the high prediction accuracy and efficiency of modern deep learning techniques. They are capable of delivering better query performance than traditional indices over one-dimensional data. Recent studies demonstrate that we can also achieve query-efficient learned indices for spatial data by partitioning and subsequently transforming spatial data to one-dimensional values, after which existing techniques can be applied. While enabling efficient querying, building and rebuilding learned spatial indices efficiently remains largely unaddressed. As the model training needed to learn a spatial index is costly, efficient building and rebuilding of learned spatial indices on large data sets is challenging if performed by means of model training and retraining.**

To advance the practicality of learned spatial indices, we propose a system named ELSI that enables the efficient building and rebuilding of a class of learned spatial indices that follow two simple design principles. The core idea is to reduce the model (re-)building times by engineering reduced training sets that preserve key data distribution patterns. ELSI encompasses a suite of methods for constructing small and distribution-preserving training sets from input data sets. Further, given an input data set, ELSI can adaptively select a method that produces a learned index with high query efficiency. Experiments on real data sets of 100+ million points show that ELSI can reduce the build times of four different learned spatial indices consistently (by up to two orders of magnitude) without jeopardizing query efficiency.

***Index Terms*—learned spatial indices, index building**

I. Introduction

Geo-referenced, or spatial, data underpins a wide and expanding range of location-based services, including digital mapping, geo-fencing, location-based social networking, and check-in applications. For example, OpenStreetMap has 7+ billion nodes and includes 9+ billion user-uploaded GPS points [1]. Applications rely on the results of querying such data, e.g., to find all Points of Interest (PoIs) in the region of space covered by a user's screen (a window query) or to find all check-ins at places of interest to users (a point query). Enabling this querying efficiently is important and challenging.

The objective of spatial indices [2]–[4] is to enable efficient spatial query processing. Due to the query performance of *learned indices* [5]–[7] on one-dimensional data, learned spatial indices [8]–[11] have been developed that offer speedups of up to orders of magnitude over existing spatial indices.

A learned (spatial) index can be conceptualized as a *function* $\mathcal{M}$ from a domain of search keys to a range of storage addresses (or partition IDs) where corresponding data is stored. When given a search key $p.key$ of a data point p, $\mathcal{M}(p.key)$ approximates the storage address $p.addr$, where data related to p is stored. We search for p within range $[\mathcal{M}(p.key) + err_l, \mathcal{M}(p.key) + err_u]$, where err_l and err_u represent *error bounds* of $\mathcal{M}$ that bound the value of $\mathcal{M}(p.key) - p.addr$ for all points indexed. These bounds are derived from the learning process of $\mathcal{M}$. A query is then answered by one or a few function invocations. This approach substantially reduces the query costs compared with using traditional indices such as R-trees [12] that may incur recursive tree traversals.

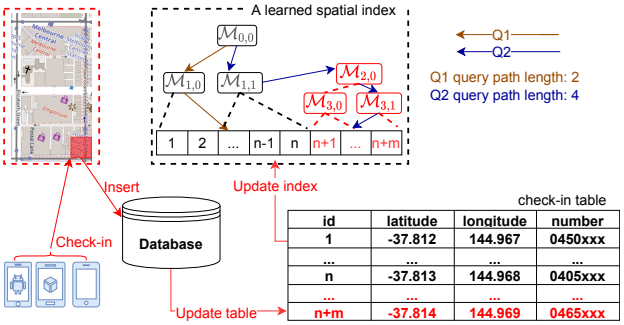


Fig. 1: An unbalanced learned spatial index created by skewed insertions (red objects, best viewed in color).

While learned spatial indices target high query efficiency, their performance can degrade quickly with data updates, especially when the updates cause the data distribution to change. A straightforward solution is to frequently relearn the index function $\mathcal{M}$ but frequent relearning is prohibitively expensive. Learned spatial indices such as *RSMI* [8] and *LISA* [9] use additional models and data pages, respectively, to support data insertions without relearning the initially learned function $\mathcal{M}$, leading to increasingly sub-optimal query processing the more points are inserted.

For example, Figure 1 shows an RSMI index with three models $\mathcal{M}_{0,0}$, $\mathcal{M}_{1,0}$, and $\mathcal{M}_{1,1}$ learned initially on n check-in where $\mathcal{M}_{i,j}$ denotes the jth model at the ith index layer. After m skewed insertions (e.g., check-ins from a small region), three more local models $\mathcal{M}_{2,0}$, $\mathcal{M}_{3,0}$, and $\mathcal{M}_{3,1}$ (in red) are built by RSMI, causing an unbalanced structure and more function invocations for some queries. For example, while query Q1 only needs two model invocations (brown arrows), query Q2 needs four (blue arrows).

To avoid degrading performance, index rebuilds are required, which is also done for one-dimensional indices in database systems such as Oracle and SQL Server [13], [14]. However, (re)building a learned spatial index on a large data set is challenging due to the high cost of learning function $\mathcal{M}$. LISA and RSMI report hours to learn indices on 50 to 100 million points [8], [9].

† Corresponding author.

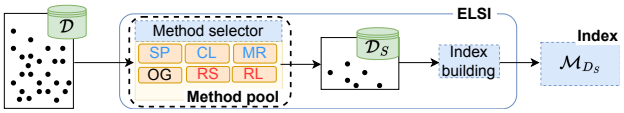


Fig. 2: An overview of ELSI.

To enable efficient index (re)building, we propose a system for efficient learning of spatial indices, name ELSI. As Figure 2 shows, our core idea is that, given a spatial data set $\mathcal{D}$, we compute a much smaller set $\mathcal{D}_\mathcal{S}$ (e.g., $|\mathcal{D}_\mathcal{S}|$ is at the million scale, while $|\mathcal{D}|$ is at billion scale) through ELSI, which preserves the data distribution of $\mathcal{D}$ in $\mathcal{D}_\mathcal{S}$. We then learn an index model $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$ on $\mathcal{D}_\mathcal{S}$ and use it for querying $\mathcal{D}$. Comparing with learning a model $\mathcal{M}_\mathcal{D}$ on $\mathcal{D}$, learning $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$ is much more efficient, since $|\mathcal{D}_\mathcal{S}| \ll |\mathcal{D}|$, and the learned model $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$ is expected to retain high query efficiency on $\mathcal{D}$.

To ensure both build and query efficiency, ELSI offers a method pool of six *index building methods*. Five are for generating (or fetching a pre-generated) $\mathcal{D}_\mathcal{S}$ and the other just uses the original (OG) data set $\mathcal{D}$, which serves as a backup option. ELSI can learn to choose the most suitable method for $\mathcal{D}$ using a method selector, based on predicted index building and query costs. To improve the overall rebuilding efficiency, ELSI also predicts the times when rebuilding is needed.

Three of the six index building methods are adapted from the literature: *sampling* (SP) [15], *clustering* (CL) [11], and *model reuse* (MR) [16]. SP generates $\mathcal{D}_\mathcal{S}$ by *systematic sampling* [17] over $\mathcal{D}$. This is simple and efficient, but queries in sparse regions, where no data points have been sampled for model training, may experience poor performance. CL uses cluster centroids from $\mathcal{D}$ as $\mathcal{D}_\mathcal{S}$. Its limitation is the time needed to build an index, since clustering a large data set is expensive compared to sampling. MR generates synthetic data sets and pre-trains index models on them. Then $\mathcal{D}$ is indexed using a pre-trained model corresponding to the synthetic data set that is the most similar to $\mathcal{D}$ according to their *cumulative distribution functions* (CDFs). Technically, MR does not generate $\mathcal{D}_\mathcal{S}$ after receiving $\mathcal{D}$. Its query efficiency suffers if none of the synthetic data sets are sufficiently similar to $\mathcal{D}$.

Two of the six index building methods are proposed: *representative set* (RS) and *reinforcement learning* (RL), to obtain better approximations $\mathcal{D}_\mathcal{S}$ of $\mathcal{D}$ at reduced costs. RS recursively partitions the data space and selects a point from each final partition to represent the partitions and form $\mathcal{D}_\mathcal{S}$. Each space partition, i.e., a cell, is divided into 2^d (d is the data dimensionality) cells until each cell has at most a preset number of points. For example, when $d = 2$, RS forms a quadtree [18] partitioning. A point from each non-empty cell is selected to form $\mathcal{D}_\mathcal{S}$. This process is highly efficient, and every data point in $\mathcal{D}$ is approximated by a close-by point (i.e., sharing the same cell). RL, on the other hand, partitions the data space through a grid. Instead of selecting from points in each grid cell, it assumes empty grid cells at start and learns to add points to a subset of the cells to form a $\mathcal{D}_\mathcal{S}$ that best approximates (the CDF of) $\mathcal{D}$. We formulate the search process of $\mathcal{D}_\mathcal{S}$ (i.e., a sequence of point adding/removal operations) as a *Markov decision process* and apply reinforcement learning for the search. RL offers a high-quality approximation of $\mathcal{D}$ while the size of $\mathcal{D}_\mathcal{S}$ can be controlled by the grid resolution.

To summarize, the paper makes the following contributions:

(1) We propose ELSI – a system for efficient building and update (in terms of rebuild) of learned spatial indices. ELSI is the first such system that can support any learned spatial indices that follow the map-and-sort index paradigm and the predict-and-scan query paradigm.

(2) To index a data set $\mathcal{D}$, ELSI learns to choose an index building method that generates a small data set $\mathcal{D}_\mathcal{S}$ resembling $\mathcal{D}$ and builds an index with $\mathcal{D}_\mathcal{S}$ for $\mathcal{D}$, based on a preference factor that balances index building cost and the subsequent query processing efficiency. We propose two index building methods RS and RL based on space partitioning and reinforcement learning, both of which build learned indices of high query performance efficiently.

(3) We integrate ELSI into four different learned spatial indices and report on extensive experiments on both synthetic and real data. ELSI improves the index build times by a factor of 70 on average. The resulting indices retain high query efficiency for both point and window query, only the kNN query times differ by just 3%.

II. Related Work

Traditional spatial indices. Traditional spatial indices use *data partitioning*, *space partitioning*, or *data mapping* for index building. Data partitioning-based indices organize the data into partitions of nearby data objects. A typical example is the *R-trees* [4], [12], [19]–[21] that form hierarchical structures of data partitions, each partition being represented by its *minimum bounding rectangles* (MBR). Space partitioning-based indices partition the data space such that data objects in each partition fit into an index unit.The *quadtree* [18] and *grid file* [3] are typical examples. Data mapping-based indices map multidimensional (spatial) objects to one-dimensional values (e.g., using *space-filling curves*, SFC), which are then indexed and queried via one-dimensional indices such as B-trees.

Most traditional indices are tree structured, and queries generally incur at least one tree traversal. Grid-structured indices are a notable exception, but they struggle when choosing a single grid resolution for skewed data. Tree traversals can be expensive, especially for large non-memory resident data sets, which motivates the study of learned spatial indices.

Learned spatial indices. Learned indices view an index structure as an *index function* that maps from a domain of search keys to a range of storage addresses of data objects. Once an index function is learned, a query can be answered by function invocation in constant time. *RMI* [5] implements this idea on one-dimensional data. A number of follow-up studies extend this idea to multidimensional data. Two key questions must be considered when designing a learned spatial index: (i) How are the spatial objects partitioned, such that each partition matches the learning capacity of a machine learning model used for index learning? (ii) How are the spatial objects in each partition sorted for index learning?

The *Z-order model index* (ZM) [10] maps data points to their Z-curve values for sorting and then uses RMI for indexing. *ML-Index* [11] also uses RMI for indexing, but adopts the *iDistance* technique [22] to map the data points to one-dimensional values. *RSMI* [8] creates a hierarchy of space partitions using SFCs. It then maps the data points to their partition IDs (recursively), by which the points are sorted for partitioning and index learning. *LISA* [9] partitions the data space according to a grid and maps the data points to one-dimensional values according to a weighted aggregation of their coordinates. It learns a *shard prediction function* to predict a shard ID for each point given its mapped value. The points are sorted and indexed by their shard IDs in the form of data pages. To process insertions, new points are added to data pages by their predicted shared IDs, and new pages are created as needed. This can lead to a skewed structure that impacts the query efficiency. *SPRIG* [23] also partitions according to a grid. It predicts the cell ID of a point with its coordinates using learned *bi-linear interpolation functions*. *Flood* [24] and *Tsunami* [25] partition a d-dimensional space using a $(d-1)$-dimensional grid. The points in each partition are indexed by their coordinates in the last dimension with RMI.

III. Preliminaries

Given a set $\mathcal{D}$ of n points $p_1, p_2, \ldots, p_n$ in d-dimensional Euclidean space ($d \in \mathbb{N}^+$ and $d \geq 2$), a learned spatial index learns an *index model* $\mathcal{M}$ that predicts the storage address of a point p_i given its coordinates. Our aim is a system that enables efficient (re)building of learned spatial indices. The problem that the system aims to solve is defined as:

Definition 1 (Problem definition)**.** *Given a data set $\mathcal{D}$, we aim to compute a set $\mathcal{D}_\mathcal{S}$ where $|\mathcal{D}_\mathcal{S}| \ll |\mathcal{D}|$, and $\mathcal{D}_\mathcal{S}$ preserves the distribution of $\mathcal{D}$, such that an index model $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$ can be learned on $\mathcal{D}_\mathcal{S}$ efficiently, and $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$ can be used as model $\mathcal{M}$ to query $\mathcal{D}$ with little sacrifice in query efficiency.*

System applicability conditions. We do *not* propose a new learned spatial index but instead propose a versatile system named ELSI that can be integrated into existing learned spatial indices and thus enable efficient (re)building for these. To use with ELSI, a learned spatial index must satisfy:

(1) Map-and-sort index paradigm: Points in $\mathcal{D}$ are mapped to a one-dimensional space and stored based on the sorted order in the mapped space. The index model $\mathcal{M}$ learns this storage order. This condition enables sampling in the mapped space to form $\mathcal{D}_\mathcal{S}$, as well as point ordering in the mapped space after $\mathcal{D}_\mathcal{S}$ is obtained.

(2) Predict-and-scan query paradigm: A point query q (which is the basis for more complex queries such as window queries) on $\mathcal{D}$ is processed by an index model invocation $\mathcal{M}(q)$ to predict the storage address of q, followed by a scan over the addresses in $[\mathcal{M}(q) - err_l, \mathcal{M}(q) + err_u]$. This condition guarantees query correctness (for point queries) when there are prediction errors.

Data set similarity measurement. A core capability of ELSI is to quantify the similarity between $\mathcal{D}_\mathcal{S}$ and $\mathcal{D}$. For this purpose, we use the *cumulative distribution functions* (CDF) of the search keys of the data sets. This is because the data points are sorted and stored in a certain order. *An index model effectively learns a mapping from the key values used for sorting to the sorted ranks of the points. This mapping corresponds to the CDF of the key values.* For example, ZM [10] sort the points by Z-values of the points.

Let $\mathcal{K}(\mathcal{D}_\mathcal{S})$ and $\mathcal{K}(\mathcal{D})$ be the sets of key values of $\mathcal{D}_\mathcal{S}$ and $\mathcal{D}$, and let $cdf_{\mathcal{K}(\mathcal{D}_\mathcal{S})}(\cdot)$ and $cdf_{\mathcal{K}(\mathcal{D})}(\cdot)$ be their CDFs. Following Liu et al. [16], the similarity between the key value distributions of $\mathcal{D}_\mathcal{S}$ and $\mathcal{D}$ is defined as follows.

Definition 2 (Similarity between the key value distributions of two data sets)**.** *Given $\mathcal{D}_\mathcal{S}$ and $\mathcal{D}$, the similarity between the distributions of their key values is 1 minus the maximum distance between the CDFs of $\mathcal{K}(\mathcal{D}_\mathcal{S})$ and $\mathcal{K}(\mathcal{D})$:*

$$sim(\mathcal{D}_\mathcal{S}, \mathcal{D}) = 1 - \sup_{x \in \mathbb{R}} |cdf_{\mathcal{K}(\mathcal{D}_\mathcal{S})}(x) - cdf_{\mathcal{K}(\mathcal{D})}(x)| \quad (1)$$

Here, $\sup_{x \in \mathbb{R}} |cdf_{\mathcal{K}(\mathcal{D}_\mathcal{S})}(x) - cdf_{\mathcal{K}(\mathcal{D})}(x)|$ is the maximum distance between the two CDFs.

For succinctness, we call $sim(\mathcal{D}_\mathcal{S}, \mathcal{D})$ the similarity and $dist(\mathcal{D}_\mathcal{S}, \mathcal{D}) = 1 - sim(\mathcal{D}_\mathcal{S}, \mathcal{D})$ the dissimilarity between $\mathcal{D}_\mathcal{S}$ and $\mathcal{D}$ hereafter as long as the context is clear. This similarity metric is based on the *Kolmogorov–Smirnov* (KS) test [26], which is a non-parametric test that returns the maximum distance between the empirical CDFs of two data sets.

Computing $dist(\mathcal{D}_\mathcal{S}, \mathcal{D})$ (or $sim(\mathcal{D}_\mathcal{S}, \mathcal{D})$) can be done by a scan over $\mathcal{D}_\mathcal{S}$ and $\mathcal{D}$ to compute $cdf_{\mathcal{D}_\mathcal{S}}(x)$ and $cdf_{\mathcal{D}}(x)$ for every $x \in \mathcal{D}_\mathcal{S} \cup \mathcal{D}$. This takes $O(n_\mathcal{S} + n)$ time, where $n_\mathcal{S} = |\mathcal{D}_\mathcal{S}|$ and $n = |\mathcal{D}|$, assuming sorted sets. We use a more efficient algorithm that only scans $\mathcal{D}_\mathcal{S}$. For the i-th value $\mathcal{D}_\mathcal{S}[i]$ in $\mathcal{D}_\mathcal{S}$ (in key value order), we run a binary search to find its rank j in $\mathcal{D}$ (i.e., $\mathcal{D}[j]$ is the first element in $\mathcal{D}$ no smaller than $\mathcal{D}_\mathcal{S}[i]$). We compute the absolute gap $|i/n_\mathcal{S} - j/n|$ and report the maximum gap for all $i \in [1, n_\mathcal{S}]$ as $dist(\mathcal{D}_\mathcal{S}, \mathcal{D})$. This reduces the time complexity to $O(n_\mathcal{S} \log n)$. Since $n_\mathcal{S} \ll n$, $O(n_\mathcal{S} \log n)$ is better than $O(n_\mathcal{S} + n)$ in practice.

The *earth mover's distance* (EMD) [27] is another similarity measure. Computing EMD on $\mathcal{D}$ and $\mathcal{D}_\mathcal{S}$ takes $O(n^3 \log n)$ time (even the state-of-the-art approximation takes $O(dn)$ time [28]), which is too expensive for our system.

IV. Proposed System

This section starts with an overview of ELSI and illustrates how to build an index with it (Section IV-A). We then explain the two core ELSI components, build processor and update processor, in Section IV-B1 and Section IV-B2, respectively.

A. ELSI Overview

As shown in Figure 3, ELSI has a build processor and an update processor that facilitate the building and updates of a learned spatial index, which we call the *base index*.

Building a base index may train a single index model on $\mathcal{D}$, or it may partition $\mathcal{D}$ and train an index model for each partition. ELSI improves the build time of each index model while it does not interfere with the partitioning (cf. Figure 3 where ELSI helps build three models $\mathcal{M}_{0,0}$, $\mathcal{M}_{1,0}$ and $\mathcal{M}_{1,1}$).

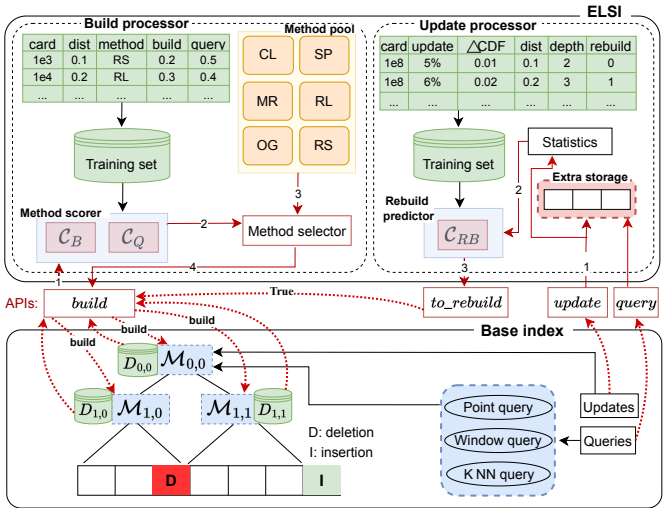


Fig. 3: The ELSI system.

Algorithm 1 summarizes how ELSI builds a single index model for $\mathcal{D}$ (or a partition of it), i.e., the *build* API in Figure 3. Here, we use `typewriter` font to denote functions that come with a base index. The algorithm starts by mapping $\mathcal{D}$ to a one-dimensional space (line 1) and sorting it (line 2). The mapping function $\texttt{map}(\cdot)$ is supplied by the base index (e.g., the Z-curve mapping of ZM). Then, the sorted data set is passed to our build processor (lines 3 and 4, detailed shortly), where we use a small data set $\mathcal{D}_\mathcal{S}$ to approximate $\mathcal{D}$. We train an index model $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$ on $\mathcal{D}_\mathcal{S}$ using the $\texttt{train}(\cdot)$ function of the base index (e.g., FFN training, line 5). Finally, we predict the address of every point in $\mathcal{D}$ using $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$ and a model prediction function $\texttt{predict}(\cdot)$ from the base index (e.g., FFN prediction), and we record the maximum prediction errors err_l and err_u (line 6). Model $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$ is returned as the index model $\mathcal{M}$ for $\mathcal{D}$ together with the error bounds (line 7).

After an index is built, queries are processed using the procedures that come with the base index. Updates go through our update processor (the *update* API in Figure 3, detailed shortly) that predicts the next time to rebuild (the *to_rebuild* API). When a rebuild is triggered, we use the *build* API to rebuild the base index.

Query error bounds. Existing learned spatial indices only offer empirical query error bounds but not theoretical bounds. ELSI enables efficient building of such indices and hence also offers empirical query error bounds. A few learned indices for one-dimensional data, e.g., PGM [6], use piece-wise linear approximation to approximate the CDF of one-dimensional data, which allows a theoretical bound on the query error based on the approximation error. Extending this idea to learned spatial indices is interesting but beyond the scope of our study.

B. ELSI Modules

ELSI has two main modules: the *build processor* and the *update processor*.

1) Build Processor: The build processor uses an *index building method selector* to select a method P from a *method pool* $\mathcal{P}$ for learning an index. The aim is to achieve both

Algorithm 1: build_index

Input: $\mathcal{D}$
Output: Model $\mathcal{M}$, error bounds err_l and err_u
1 $\mathcal{D} \leftarrow \texttt{map}(\mathcal{D})$;
2 $\mathcal{D} \leftarrow \text{sort}(\mathcal{D})$;
3 $P \leftarrow \text{get\_build\_method}(\lambda, w_Q, \mathcal{D})$;
4 $\mathcal{D}_\mathcal{S} \leftarrow \text{compute\_set}(\mathcal{D}, P)$;
5 $\mathcal{M}_{\mathcal{D}_\mathcal{S}} \leftarrow \texttt{train}(\mathcal{D}_\mathcal{S})$;
6 $err_l, err_u \leftarrow \text{get\_error\_bound}(\mathcal{M}_{\mathcal{D}_\mathcal{S}}, \mathcal{D}, \texttt{predict}(\cdot))$;
7 **return** $\mathcal{M}_{\mathcal{D}_\mathcal{S}}, err_l, err_u$;

efficient index building and query processing for $\mathcal{D}$. A *method scorer* is employed, and the method with the maximum score is selected. The key element of method scorer is two FFNs (Component 2 in Figure 4), one that estimates the index building cost of a method P, denoted by $\mathcal{C}_B(\cdot)$, and one that estimates the query cost of the index built by P, denoted by $\mathcal{C}_Q(\cdot)$. We consider point query costs since point queries are building blocks for more complex queries. Costs of other query types, e.g., window queries, can also be considered.

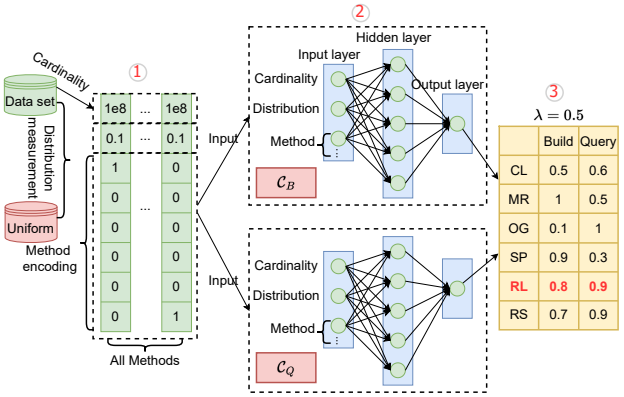


Fig. 4: The ELSI index building method scorer.

The combined score $\mathcal{C}(P, \mathcal{D})$ of method P for $\mathcal{D}$ is then a weighted sum of the two costs, with a balancing parameter $\lambda \in [0, 1]$ and a query frequency parameter $w_Q \in [1, \infty)$ allowing user-defined trade-offs between the costs:

$$\mathcal{C}(P, \mathcal{D}) = \lambda \cdot \mathcal{C}_B(P, \mathcal{D}) + (1 - \lambda) \cdot w_Q \cdot \mathcal{C}_Q(P, \mathcal{D}) \quad (2)$$

Each FFN of the method scorer takes as input the ID (a one-hot embedding) of method P and the cardinality and distribution of $\mathcal{D}$ (Component 1 in Figure 4). We use $dist(\mathcal{D}_\mathcal{U}, \mathcal{D})$ to represent the distribution of $\mathcal{D}$ for fast online computation, where $\mathcal{D}_\mathcal{U}$ is a uniform data set of the same size as $\mathcal{D}$. The method scorer is trained based on every method in the *method pool*, which contains the methods for shrinking the training set. Currently, we have six methods. The output of the two FFNs is a pair of predicted index building and query cost scores of P (Component 3 in Figure 4). They represent the predicted speedups that P can yield, comparing with the original index building method of the base index.

2) Update Processor: The update processor provides default update procedures (a base index can also use its built-in update procedures). It uses a separate list to store newly inserted points and deleted existing points (or just marks the points as deleted if allowed by the base index). This list is scanned when processing a query, and the results are combined

with, or are used to filter, those from the index to form the query result. A binary tree on the IDs of the updated points can be employed to reduce the query time.

As more and more updates are performed on the full data set $\mathcal{D}$, the index models learned for $\mathcal{D}$ may become sub-optimal for queries. We then perform a rebuild, by triggering a full index build with the base index and the build processor. ELSI takes a learning-based approach to predict the time to rebuild, unlike traditional database systems such as Oracle [13], which uses empirical rules. We use a *rebuild predictor* (an FFN, the *to_rebuild* API) $\mathcal{C}_{RB}(\cdot)$ that has a similar structure to that of the FFN in the method scorer (Figure 4) but outputs a binary value, indicating whether or not to rebuild. The rebuild predictor takes as input the cardinality and distribution of $\mathcal{D}$ (i.e., $dist(\mathcal{D}_\mathcal{U}, \mathcal{D})$ as above), the index depth, the update ratio, i.e., $|\mathcal{D}'|/|\mathcal{D}| - 1$ ($\mathcal{D}'$ is the updated data set), and the *changes* to $\mathcal{D}$ caused by updates. We quantify the changes to $\mathcal{D}$ by the difference between the CDFs of $\mathcal{D}'$ and $\mathcal{D}$, i.e., $sim(\mathcal{D}', \mathcal{D})$, since a learned index learns the CDF of a data set. When a full index is built (or rebuilt) on $\mathcal{D}$, we compute and store its CDF (an $O(n)$-sized vector). We maintain a copy of this CDF as the CDF of $\mathcal{D}'$ and compute $sim(\mathcal{D}, \mathcal{D}')$ as updates are processed. We run the rebuild predictor after every f_u, with f_u being a system parameter. Compared with the build processor, the model input has not information about build method because the build processor concerns the build methods, while the rebuild predictor concerns the index itself.

V. Index Building Methods

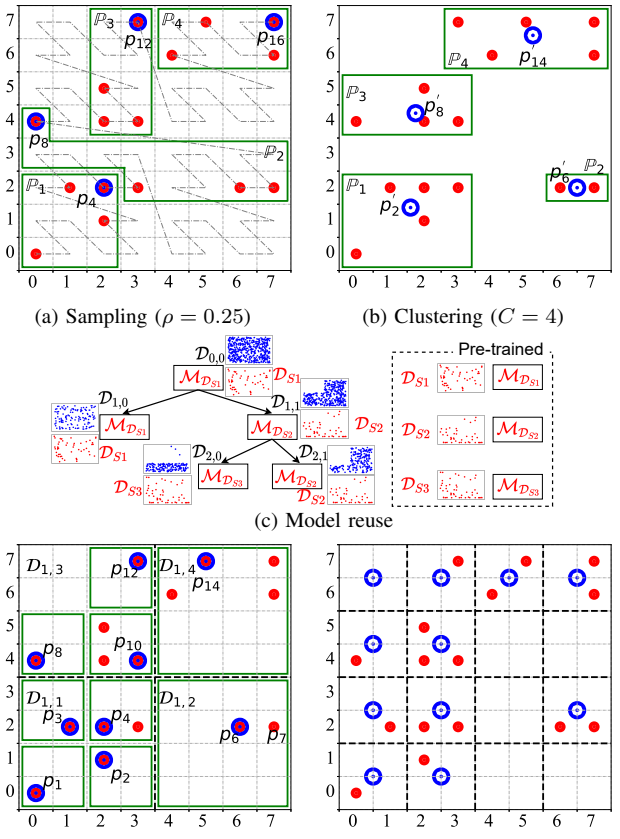
This section details the index building methods in ELSI. As mentioned earlier, these methods do not build new types of indices but rather construct (or find) small data sets $\mathcal{D}_\mathcal{S}$ that resemble the input data set $\mathcal{D}$. We present three methods adapted from the literature in Section V-A and propose two new methods in Section V-B. We cover implementation details in Section VII-B2.

A. Adapted Methods

The first two adapted methods use sampling and clustering to construct $\mathcal{D}_\mathcal{S}$ from $\mathcal{D}$, while the third identifies a pre-generated data set $\mathcal{D}_\mathcal{S}$ that matches $\mathcal{D}$.

1) Sampling: Random sampling was used in a recent learned index [15] to reduce the training set size. The ELSI *sampling* method (SP) uses *systematic sampling* [17] instead. Given a sorted set $\mathcal{D}$, SP constructs $\mathcal{D}_\mathcal{S}$ by selecting a point after every $\lfloor 1/\rho \rfloor - 1$ points, where ρ is the sampling rate. $\mathcal{D}_\mathcal{S}$ then has $\rho \cdot n$ points, where $n = |\mathcal{D}|$. Figure 5(a) shows an example with 16 points (in red) mapped and sorted with a Z-curve (SP also works with other mapped one-dimensional spaces). Given $\rho = 0.25$, $\mathcal{D}_\mathcal{S} = \{p_4, p_8, p_{12}, p_{16}\}$ (points in $\mathcal{D}_\mathcal{S}$ are in blue, also in the figures for subsequent methods).

Let the i-th point in $\mathcal{D}$ be $\mathcal{D}[i]$, and let its nearest sampled point in $\mathcal{D}_\mathcal{S}$ be $\mathcal{D}[j]$ (i.e., the j-th point in $\mathcal{D}$). With systematic sampling, we bound the gap between i and j by $\lfloor 1/\rho \rfloor - 1$, i.e., $|i-j| \leq \lfloor 1/\rho \rfloor - 1$. According to the pigeonhole principle, no other sampling strategy (including random sampling) can achieve a smaller bound on this gap. We thus expect the $\mathcal{D}_\mathcal{S}$ produced by SP to be a strong baseline approximation of $\mathcal{D}$.



(a) Sampling ($\rho = 0.25$) (b) Clustering ($C = 4$)

(c) Model reuse

(d) Representative set ($\beta = 4$) (e) Reinforcement learning ($\eta = 4$)

Fig. 5: Examples of ELSI index building methods.

While SP is efficient, it only considers the point order in the mapped space, not the data point locations. The sampled points may then be far away from the points they represent, especially in sparse regions, cf. p_8 in Figure 5(a).

2) Clustering: To preserve data distribution patterns, the *clustering* method (CL) clusters $\mathcal{D}$ in the original space into C (a system parameter) clusters and uses the set of cluster centroids as $\mathcal{D}_\mathcal{S}$. We use the k-means algorithm due to its simplicity, although other algorithms may also apply. Figure 5(b) shows an example. After clustering the 16 points into $C = 4$ clusters, we compute four centroids p'_2, p'_6, p'_8, and p'_{14} to obtain $\mathcal{D}_\mathcal{S}$. Note that these centroids are closer to the points that they represent than those chosen by SP.

We note that the cluster centroids may not be part of $\mathcal{D}$. However, this does not impact the mapping from a data point's coordinates to its search key, which is either independent of the data set (e.g., ZM [10] using Z-curves), or is computed using $\mathcal{D}$ (e.g., the ML-Index [11] using iDistance [22]). MR and RL, to be presented shortly, also use non-subsets of $\mathcal{D}$.

However, clustering with k-means is expensive when C is large, taking $O(C \cdot |\mathcal{D}| \cdot d \cdot i)$ time for a straightforward implementation with i iterations. Further, it may create unbalanced clusters. For example, in Figure 5(b), cluster $\mathbb{P}_2$ has only two points, while the other clusters have four or five points.

Algorithm 2: get_RS

Input: $\mathcal{D}$, β, $\{(l_1, u_1), (l_2, u_2), \ldots, (l_d, u_d)\}$
Output: Representative set $\mathcal{D}_S$

```
if |D| ≤ β then
    D_S ← {median point in D};
else
    for i = 0; i < 2^d; i++ do
        {(l_1^i, u_1^i), ..., (l_d^i, u_d^i)} ← compute bounds for
          partition i;
        D_i ← get points in the i-th partition;
        D_S ← D_S ∪ get_RS(D_i, β, {(l_1^i, u_1^i), ..., (l_d^i, u_d^i)});
return D_S;
```

3) Model Reuse: Model reuse [16] (MR) was originally proposed for one-dimensional data. It generates synthetic data sets with different distributions (e.g., Gaussian) and pre-trains index models on them. The data set with the CDF that is the most similar to that of $\mathcal{D}$ (by $dist(\mathcal{D}_\mathcal{S}, \mathcal{D})$) is used as $\mathcal{D}_\mathcal{S}$, and its pre-trained model is used to index $\mathcal{D}$. MR first generates CDFs that together (heuristically) cover the CDF space, such that the distance between the CDF of any input data set and that of at least one generated CDF is approximately bounded by a pre-defined threshold $\epsilon \in (0, 1]$. MR then generates synthetic data sets following the CDFs. By following this idea, we first generate CDFs and then corresponding synthetic data sets in the mapped space of a base index.

Figure 5(c) shows an example of a multi-level index (RSMI) built by MR. There are three synthetic data sets $\mathcal{D}_{\mathcal{S}1}$, $\mathcal{D}_{\mathcal{S}2}$, and $\mathcal{D}_{\mathcal{S}3}$ (plotted in the original space for ease of observation), with pre-trained models $\mathcal{M}_{\mathcal{D}_{\mathcal{S}1}}$, $\mathcal{M}_{\mathcal{D}_{\mathcal{S}2}}$, and $\mathcal{M}_{\mathcal{D}_{\mathcal{S}3}}$, respectively. Given a data set $\mathcal{D}_{0,0}$ for indexing, we compare it with all three synthetic data sets and find $\mathcal{D}_{\mathcal{S}1}$ to be the most similar. We thus use $\mathcal{M}_{\mathcal{D}_{\mathcal{S}1}}$ to index $\mathcal{D}_{0,0}$. This model predicts $\mathcal{D}_{0,0}$ into two partitions $\mathcal{D}_{1,0}$ and $\mathcal{D}_{1,1}$, which are the most similar to $\mathcal{D}_{\mathcal{S}1}$ and $\mathcal{D}_{\mathcal{S}2}$ and hence are indexed by $\mathcal{M}_{\mathcal{D}_{\mathcal{S}1}}$ and $\mathcal{M}_{\mathcal{D}_{\mathcal{S}2}}$, respectively. The process continues until no more partitioning is required by the base index.

B. Proposed Methods

To address the limitations related to index building efficiency (CL) or query efficiency (SP and MR) of the adapted methods, we propose two additional methods.

1) Representative Set: The first method, *representative set* (RS), recursively partitions the d-dimensional original data space into 2^d equi-sized partitions (e.g., four quadrants in a 2-dimensional space, as in quadtree partitioning). The process continues until every partition has no more than β points, where β is a system parameter. For each partition, the median point in the mapped one-dimensional space is added to $\mathcal{D}_\mathcal{S}$.

We summarize RS in Algorithm 2, where the lower and the upper bounds of the data space are denoted by $\{(l_1, u_1), (l_2, u_2), \ldots, (l_d, u_d)\}$. The bounds of the 2^d partitions in the recursive function calls are combinations of $l_1, l_2, \ldots, l_d$, $u_1, u_2, \ldots, u_d$, and $\frac{l_1+u_1}{2}, \frac{l_2+u_2}{2}, \ldots, \frac{l_d+u_d}{2}$. Points in each $\mathcal{D}_i$ can be computed with a scan over $\mathcal{D}$.

Figure 5(d) gives an example, where $d = 2$ and $\beta = 4$, and $\mathcal{D}_{i,j}$ denotes partition j at partition level i. At level 1, $\mathcal{D}_{1,2}$ and $\mathcal{D}_{1,4}$ do not exceed 4 points each and do not require further partitioning. Partitions $\mathcal{D}_{1,1}$ and $\mathcal{D}_{1,3}$ are partitioned for one more level. Finally, we have 9 non-empty partitions (solid green rectangles). Their median points form $\mathcal{D}_S = \{p_1, p_2, p_3, p_4, p_6, p_8, p_{10}, p_{12}, p_{14}\}$.

RS uses partitions of the original space and ranks in the mapped space for sampling in order to better approximate the distribution patterns of $\mathcal{D}$ in both the original and mapped spaces. RS is expected to achieve high query performance, to be confirmed experimentally.

2) Reinforcement Learning: The *reinforcement learning* (RL)-based method aims to learn a set $\mathcal{D}_\mathcal{S}$ of up to η^d points that best approximate $\mathcal{D}$, using reinforcement learning. Here, η is a system parameter. RL partitions the data space with an η^d grid, fills every cell with a point initially, and continues to remove points from (or add points back into) the cells (Figure 5(e)). It monitors $dist(\mathcal{D}_\mathcal{S}, \mathcal{D})$ and terminates when $dist(\mathcal{D}_\mathcal{S}, \mathcal{D})$ stops improving.

Reinforcement learning formulation. There are 2^{η^d} point combinations for forming different $\mathcal{D}_\mathcal{S}$. To approach the optimal $\mathcal{D}_\mathcal{S}$, we reduce the search cost via reinforcement learning and formulate the search as a *Markov decision process* (MDP):

(1) State space $\mathcal{S}$, where a state $s_t \in \mathcal{S}$ at time step t is a vector $(s_t[1], s_t[2], \ldots, s_t[\eta^d])$, where $s_t[i]$ is a binary variable that indicates whether there is a point in cell i. The cells are ordered by their ranks in the mapped space of the base index. The initial state s_0 has value 1 in every cell, i.e., $\mathcal{D}_\mathcal{S}$ starts with a uniform distribution.

(2) Action space $\mathcal{A}$, where an action $a \in \mathcal{A}$ is to add (or remove) a point to (from) a cell.

(3) Reward function $\mathcal{R}$: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathcal{R}$, which is calculated as the reduction in $dist(\mathcal{D}_\mathcal{S}, \mathcal{D})$ caused by an action $a_t \in \mathcal{A}$ on state $s_t \in \mathcal{S}$, i.e., $\mathcal{R}(s_t, a_t, s_{t+1}) = dist(\mathcal{D}_{\mathcal{S}t}, \mathcal{D}) - dist(\mathcal{D}_{\mathcal{S}t+1}, \mathcal{D})$.

(4) State transition function $\mathcal{P}$: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, which describes a probability distribution of the next state s_{t+1} given state s_t at time step t and action a_t.

(5) Discount factor $\gamma \in [0, 1]$, which discounts the reward accumulated further into the future ($\gamma = 0.9$ in experiments).

We use a *deep Q-network* [29] (DQN) to learn the optimal policy π: $\mathcal{A} \times \mathcal{S} \to [0, 1]$ that maximizes the reward. At step t, we choose cell i with DQN, update $s_t[i]$ to $s_{t+1}[i] = 1 - s_t[i]$ with probability ζ ($\zeta = 0.8$ in experiments), and compute the reward. The DQN is trained by recent state transition and reward records in memory after every five steps. When the learning stops, the final state yields the $\mathcal{D}_\mathcal{S}$ returned by RL.

VI. Cost Analysis

We analyze the build, query, and update costs when using ELSI and benchmark these against those when not using ELSI.

A. Cost Formulation

For simplicity, we consider building and querying a single index model $\mathcal{M}$ on a data set $\mathcal{D}$. We decompose the build and query costs to facilitate the performance comparison.

Build cost decomposition. The build cost $cost_b$ of a map-and-sort based index can be decomposed into three terms: (i) data preparation cost $cost_{dp}$, (ii) model training cost $cost_{tr}$, and (iii) extra costs $cost_{ex}$ introduced by ELSI where $cost_b = cost_{dp} + cost_{tr} + cost_{ex}$. The build costs are:

(i) $cost_{dp} = O(nd + n \log n)$: All n points in $\mathcal{D}$ are mapped to a one-dimensional space, which typically takes $O(nd)$ time to scan all points and dimensions. The points are then sorted in the mapped space, taking $O(n \log n)$ time.

(ii) $cost_{tr} = \mathbb{T}(n) + \mathbb{M}(n)$: All n points in $\mathcal{D}$ are used to train an index model. The costs depend on the data set size, the model structure (e.g., the numbers of layers), and the number of training iterations. For ease of comparison, we focus on the impact of the data set size and denote the two costs by $\mathbb{T}(n)$ for training and $\mathbb{M}(n)$ for model invocation.

Query cost decomposition. We focus on point queries and consider a predict-and-scan process. The query cost $cost_q$ has two terms: (i) prediction cost $cost_{pr}$ and (ii) scan cost $cost_{sc}$ where $cost_q = cost_{pr} + cost_{sc}$. The query costs are:

(i) $cost_{pr} = \mathbb{M}(1)$: The index model is invoked once with the (mapped) query point. We use $\mathbb{M}(1)$ to denote this cost.

(ii) $cost_{sc} = O(err_l + err_u)$: A scan is run over addresses adjacent to the predicted one, bounded by err_l and err_u.

B. Build Cost

To build an index with ELSI, the data preparation cost remains $cost_{dp} = O(nd + n \log n)$, while the model training cost now is $cost_{tr} = \mathbb{T}(|\mathcal{D}_\mathcal{S}|) + \mathbb{M}(n)$, since training over $\mathcal{D}_\mathcal{S}$. The extra costs $cost_{ex}$ incurred by ELSI include $\mathbb{M}(1) + O(n)$ time to invoke the method scorer to select build method and additional method-specific costs. Next, we consider $\mathbb{T}(|\mathcal{D}_\mathcal{S}|)$ and the extra costs of each method. The overall costs and empirical results of the methods are summarized in Table I.

SP samples $\rho \cdot n$ points. Its model training cost is $cost_{tr} = \mathbb{T}(\rho \cdot n) + \mathbb{M}(n)$, and it incurs $O(\rho \cdot n)$ extra time to sample the points. **CL** uses C cluster centroids for model training. Its model training cost is $cost_{tr} = \mathbb{T}(C) + \mathbb{M}(n)$. Clustering with k-means adds $O(C \cdot n \cdot d \cdot i)$ time to $cost_{ex}$, using a straightforward implementation with i iterations. The centroids also need to be mapped and sorted, but they do not impact $cost_{dp}$ in big-O terms, as $C \ll n$. **MR** uses pre-trained models and does not run online training, i.e., $cost_{tr} = \mathbb{M}(n)$. It computes the similarity between $\mathcal{D}$ and n_{mr} synthetic sets, each of size $n_\mathcal{S}$, taking $O(n_{mr} n_\mathcal{S} \log n)$ extra time. **RS** stops partitioning when a partition reaches β points and uses one point per partition for training. Its training cost is $cost_{tr} = \mathbb{T}(n/\beta) + \mathbb{M}(n)$. The partitioning tree has depth $O(\log_{2^d}(n/\beta))$ on average and takes $O(n \log_{2^d}(n/\beta))$ time to compute. **RL** approximates the original data set with a synthetic set of size η^d, i.e., $cost_{tr} = \mathbb{T}(\eta^d) + \mathbb{M}(n)$. Generating this set with reinforcement learning adds extra costs for state and action training, reward calculation, and cell selection via DQN. Each step invokes the DQN network, which takes $\mathbb{M}(1)$ cost, and calculates the reward (i.e., $dist(\mathcal{D}_\mathcal{S}, \mathcal{D})$), which takes $O(\eta^d \log n)$ time. The DQN is trained once in every five steps, performing learning on α past state transition records and corresponding rewards. With e steps, these combine to contribute $\mathbb{M}(e) + O(e\eta^d \log n) + \mathbb{T}(\alpha)$ extra time.

C. Query Cost

ELSI does not change the structures of index models. Its prediction cost is still $cost_{pr} = \mathbb{M}(1)$. It may incur extra scan costs, since model $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$ that is built on $\mathcal{D}_\mathcal{S}$ may have different error bounds from those of model $\mathcal{M}$ that is built on $\mathcal{D}$. Let err_l^S and err_u^S be the error bounds of $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$, and let $\Delta err = err_l^S + err_u^S - err_l - err_u$. Then, the scan cost of ELSI is $cost_{sc} = O(err_l + err_u + \Delta err)$. Just like there are no non-trivial bounds for err_l and err_u, there is no non-trivial bound for Δerr. Intuitively, Δerr is expected to be inversely proportional to $\rho \cdot n$ for SP, C for CL, $1 - \epsilon$ for MR, $1/\beta$ for RS, and η for RL. For future work, we plan to derive a theoretical bound for Δerr.

D. Update Cost

Update processing with our default procedures takes $O(\log n_u)$ time assuming the list of inserted and deleted points is indexed by the point IDs in a binary tree given n_u points in the tree. The rebuild predictor is triggered to make a prediction after every f_u updates adding $\mathbb{M}(1)/f_u$ time.

VII. Experiments

We run experiments on a computer running 64-bit Ubuntu 20.04 with a 3.60 GHz Intel i9 CPU, an RTX 2080 Ti GPU, 64 GB RAM, and a 500 GB solid-state drive. We use *PyTorch* 1.4 [30] and its C++ APIs to implement the learned indices based on the GPU. The traditional indices are implemented using C/C++ based on the CPU.

A. Experimental Setting Details

We apply ELSI to four learned spatial indices: ZM [10], ML-Index [11], RSMI [8], and LISA [9]. We report the performance of the last three and denote them as "**ML-F**", "**RSMI-F**", and "**LISA-F**", respectively ('-F' indicates that the ELSI framework is used). Since ZM has been shown [8] to be outperformed by RSMI, we only consider it in Section VII-D to investigate the performance of ELSI.

We note that not all index building methods are applicable to all learned spatial indices. For example, CL and RL do not apply to LISA that relies on $\mathcal{D}$ to construct a grid, while CL and RL may generate new points not in $\mathcal{D}$. ELSI offers an API to configure the index building methods used.

Competitors. We benchmark the learned spatial indices and compare them using four traditional indices: (1) **Grid** [3] partitions the data space with a regular grid, assigns data points to the cells they fall into and stores the data points in cell-wise fashion. We use a $\sqrt{n/B} \times \sqrt{n/B}$ grid, i.e., each cell (block) has an average of B points. (2) **KDB** [2] implements a kd-tree [31] with a B-tree structure to support block storage. (3) **HRR** [20] is an R-tree bulk-loaded using a *rank space* technique and a Hilbert-curve for the ordering. This index offers the state-of-the-art window query performance; (4) **RR*** [19] is a version of the R*-tree (i.e., the *revised R*-tree*) with improved query performance.

Data sets. We use four real data sets, **OSM1**, **OSM2**, **TPC-H**, and **NYC**, and two synthetic data sets, **Uniform** and **Skewed**. **OSM1** has about 100 million points (2.2 GB) in North America. **OSM2** has over 180 million points (4.4 GB) in South America. Both sets are extracted from OpenStreetMap [32]. **TPC-H** [33] contains the `quantity` and `shipdate` columns of 120 million records (5.0 GB) from the `lineitem` table of the TPC-H benchmark. **NYC** [34] contains the pickup points of 143 million yellow taxi transactions (5.6 GB) from the New York City.Each synthetic data set has 128 million points (2.5 GB) in a unit square. **Uniform** has a uniform distribution, while **Skewed** is obtained by replacing the y-coordinates of the points in **Uniform** by y^s ($s = 4$), following **HRR** [20].

B. Implementation Details

1) Implementation of the Spatial Indices: For Grid, HRR, KDB, RR*, and RSMI, we follow the implementation by Qi et al. [8]. We implement ML and LISA to enable a consistent comparison with the other indices. We use an FFN for all the prediction models, ReLU activation function for the hidden layer and minimize the L_2 loss, and a learning rate of 0.01 and 500 epochs with the Adam optimizer, respectively. For the method scorer, we set $w_Q = 1.0$ for simplicity, and we study the impact of λ. Note that the use of FFNs instead of the piecewise linear functions used in LISA breaks the monotonicity of its shard prediction functions, which impacts the accuracy of window queries.

We use a block size of $B = 100$ for data storage. We perform all experiments in main memory for ease of comparison (it is straightforward to place the blocks in external memory).

2) Implementation of ELSI: Next, we detail the implementation of the key components of ELSI.

Method scorer training. To train the method scorer, we generate data sets with cardinalities ranging from 10^l to 10^u, where $l = 4$ and $u = 8$. We vary $dist(\mathcal{D}_\mathcal{U}, \mathcal{D})$ from 0.0 to 0.9 with a step size of 0.1 to generate data sets with different distributions. When integrated with a base index, we use every applicable method in the method pool to build an index for each generated data set and run point queries. We record the speedups of index building and querying relative to those of the original methods of the base index. These form the ground truth for training the method scorer.

Rebuild predictor training. To train the rebuild predictor, we build learned indices with data sets of size 10^u and different distributions, as done above. We insert random points (or delete existing points) and run point queries every time $2^i\% \cdot n$ ($i \in \mathbb{N}$) point updates (inserts or deletes), on indices with and without rebuilds. We record the statistics covered in Section IV-B2 to compose model training samples, and we set the corresponding model output heuristically: 1 (to rebuild) when the query time without rebuilds exceeds that with rebuilds by 10%, and 0, otherwise.

System preparation costs. ELSI preparation includes the training of method scorer and rebuild predictor, which require the generation of training sets and the training of new models from scratch. ELSI preparation is an off-line and one-off task, and once learned, the ELSI method selector and rebuild predictor can be reused for different data sets. As u ($u \in \mathbb{N}^+$) decreases from 8 to 4, the ELSI preparation times will drop from 10.5 to 1.9, 0.2, 0.03, and 0.003 hours, respectively.

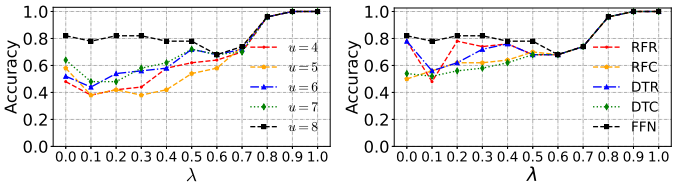
C. Effectiveness of the Method Selector

As mentioned in method scorer training (Section VII-B2), there are 300 generated synthetic data sets, which are composed of the combinations of five different cardinalities, six build methods, and ten different similarities with uniform distribution. We then use accuracy to measure the method selector where the accuracy represents the ratio that the method selector selects the same method as expected.

To test the accuracy of the method selector under different values of u, we vary u from 4 to 8 and report the accuracy in Figure 6(a). As expected, the method selector has the highest accuracy when $u = 8$, which means that the long preparation time (10.5 hours) has paid off. Meanwhile, we see that, if index build time is of priority, even using a small value of $u = 4$ can offer a high accuracy, as it is easier to predict which method to use in this case (e.g., MR).

To further highlight the effectiveness our FFN-based method selector, we compare it with method selectors using random forests (RF) and decision trees (DT), including two variants each, i.e., regression-based (R) and classification-based (C). We compare with four models: RFR, RFC, DTR, and DTC.

Figure 6(b) shows that using an FFN as the method selector has a consistently higher (or the same) accuracy than that using a random forest or a decision tree, especially when $\lambda < 0.6$. This highlights the effectiveness of our FFN-based method selector. When $\lambda \leq 0.6$ (prioritising query times), the selector accuracy is in general lower than that when $\lambda \geq 0.8$ (prioritizing index build times). This is because optimizing the query times of the learned indices is more difficult than optimizing the index build times, since the query times of the indices built by different methods are much closer than the build times of the different methods. FFN manages to achieve an accuracy around 0.8, while the other methods have accuracy at as low as 0.5. Further, when $\lambda \approx 0.6$, the accuracy is the lowest for FFN, because the build times and the query times have similar weights, which makes it difficult to learn and select the optimal index building method. When $\lambda = 0.1$, there is a drop in the accuracy of both RFR and DTR. They have been confused by the high query efficiency of the indices built by OG, and have often selected OG as the building method.



(a) Accuracy vs. u. (b) Accuracy vs. selector model.

Fig. 6: Accuracy of method selector vs. λ

D. Effectiveness of Index Building Methods

We study the relative performance of the index building methods SP, CL, MR, RS, and RL (cf. Section V). We also show the performance of the base indices without ELSI, i.e., *original* (OG), and a random sampling method RSP [15].

Pareto analysis. First, we show the point query times and build times of the methods when varying method-specific parameters following SOSD [35]. Figure 7 presents the results on OSM1 for all four base indices. We make four observations:

(1) The build times increase while the query times decrease for the different methods, with the increase of ρ in SP and RSP, C in CL, $1-\epsilon$ in MR, $1/\beta$ in RS, and η in RL ($\alpha = 10,000, e = 50,000$). RSP has higher query times than SP and few benefits in build times, because RSP has larger CDF distances between $\mathcal{D}_S$ and $\mathcal{D}$ than SP does, impacting query efficiency on $\mathcal{D}$. In what follows, we only consider SP.

(2) The proposed RS and RL methods have low query times, because of their high-quality training sets. RS achieves the lowest query time for ZM, while it is very close to CL for RSMI and SP for LISA. Meanwhile, its build times are much lower than those of CL that employs an expensive clustering process. RL performs close to RS but is inapplicable for LISA.

(3) RS and RL have larger build times than SP and MR, due to their larger training set size and more complex learning process, respectively. SP simply samples ρn points for training, while MR just reuses a pre-trained model. Both methods are thus favored in build-cost sensitive settings.

RS and RL have the advantage that tuning their parameters β and η leads to query times close to, or even below, those of OG, which is more difficult to achieve with SP and MR. The two methods are favored in query-cost sensitive settings.

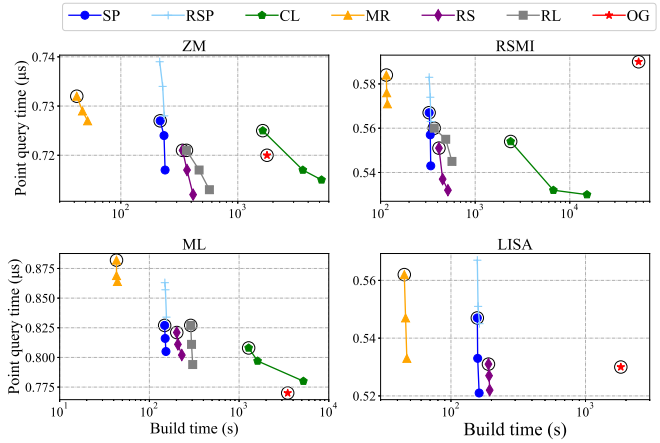


Fig. 7: Comparison of different build methods on OSM1. In each sub-figure, as the query time decreases, ρ in SP and RSP increases from 0.0001 to 0.01, C in CL increases from 100 to 10,000, ϵ in MR decreases from 0.5 to 0.1 (if ϵ is too small, no pre-trained models may be reused), β in RS decreases from 10,000 to 100, and η in RL increases from 8 to 32.

(4) The query times of the indices built by ELSI may be lower than those of OG. This is counter intuitive but can be explained as follows. OG uses the full data sets, which may be too large and may contain noisy patterns (e.g., outliers) that confuse the index model. In comparison, the methods in ELSI learn from representative data samples, which are smaller and contain more consistent patterns, leading to indices with better query performance on average.

Results on the other data sets exhibit similar patterns and are omitted. Since the build times change much more rapidly (note the scales of the axes) and ELSI is proposed to reduce the index build times, we use the values that yield the best build times as default values, i.e., $\rho = 0.0001$, $C = 100$, $\epsilon = 0.5$, $\beta = 10,000$, and $\eta = 8$ (corresponding the points denoted by '$\odot$' in Figure 7).

Cost decomposition. Table I decomposes the index building and query costs on OSM1 with ZM. For index building, as all methods share the same map-and-sort data preparation process and have the same cost (i.e., $O(nd + n \log n)$ cost and 14 s in experiments), we omit it in Table I. SP, CL, RS, and RL train index models using subsets of similar sizes and have similar training times. OG has the largest training time, while MR has the smallest training time, as it simply reuses pre-trained models. After adding in the extra costs, all methods except CL outperform OG by at least an order of magnitude, confirming the efficiency of ELSI in index building. Meanwhile, the model prediction errors $err_l + err_u$ (denoted by "|Error|") of the different methods are all at the same magnitude as OG, confirming the query efficiency of ELSI.

TABLE I: Cost Decomposition on OSM1

	Building cost (seconds)		|Error|
	Training	Extra (+$\mathbb{M}(1) + O(n)$)	($\times 10^5$)
SP	$\mathbb{T}(\rho n) + \mathbb{M}(n)$	$O(\rho n)$	6.4
	1	1	
CL	$\mathbb{T}(C) + \mathbb{M}(n)$	$O(Cndi)$	5.8
	1	213	
MR	$\mathbb{M}(n)$	$O(n_{mr} n_S \log n)$	6.8
	1	0.1	
RS	$\mathbb{T}(n/\beta) + \mathbb{M}(n)$	$O(n \log_{2^d}(n/\beta))$	6.0
	1	20	
RL	$\mathbb{T}(\eta^d) + \mathbb{M}(n)$	$\mathbb{M}(e) + O(e\eta^d \log n + \mathbb{T}(\alpha))$	5.2
	1	18	
OG	$\mathbb{T}(n) + \mathbb{M}(n)$	n/a	6.3
	360	0	

E. Ablation Study

We conduct an ablation study comparing ELSI with a variant of ELSI that uses an index building method selector, denoted by "**Rand**", that selects each index building method with an equal probability.

Performance comparison. Table II shows a comparison of the build and query times when building an index using ELSI, Rand, and each index building method, on OSM1 using $\lambda = 0.8$. As expected, Rand has worse build times than ELSI, as Rand risks selecting a slow method for some index models. Meanwhile, Rand does not offer better query times. This confirms the effectiveness of the ELSI design with a learned method selector.

TABLE II: Comparison of ELSI with a Random Method Selector on OSM1

	Index	ELSI	Rand	SP	CL	MR	RS	RL	OG
Build time (s)	ZM	57	217	277	1,635	42	414	364	1,778
	RSMI	233	557	326	2,372	115	337	369	53,435
	ML	60	212	148	1,278	43	202	289	3,483
	LISA	42	175	157	NA	37	190	NA	1,830
Point query time (μs)	ZM	0.73	0.73	0.73	0.73	0.73	0.72	0.72	0.72
	RSMI	0.57	0.56	0.60	0.55	0.57	0.55	0.56	0.59
	ML	0.88	0.85	0.83	0.81	0.88	0.82	0.83	0.77
	LISA	0.56	0.55	0.55	NA	0.56	0.53	NA	0.53

F. Index Building Performance

We now compare the build times of the learned indices with and without using ELSI and the traditional indices.

Varying data distribution. Figure 8 shows that the traditional indices (i.e., Grid, KDB, HRR, and RR*) are faster to build than the learned indices (i.e., ML, LISA, and RSMI) without ELSI. Using ELSI, the build times of the learned indices (i.e., ML-F, LISA-F, and RSMI-F) are reduced to the level of the traditional indices. The performance gain is at least 25 times (217 s vs. 5,481 s for ML-F and ML on OSM2) and up to 229 times (233 s vs. 53,435 s for RSMI-F and RSMI on OSM1). LISA-F even outperforms all the traditional indices on OSM2, i.e., 74 s vs. 126 s (Grid), on TPC-H, i.e., 43 s vs. 66 s (KDB), and on NYC, i.e., 55 s vs 75 s (KDB), respectively. On average, ELSI speeds up the building of the learned indices by 70 times. We notice that Grid is worse on NYC than on the other data sets. This is because Grid uses a two-level structure where every cell contains an array of MBRs each corresponding to a data block (to help query processing later on). Inserting points into this structure while minimizing the MBRs is more expensive when the data points are skewed (which is the case for NYC), since the data blocks in the denser cells require frequent splits.

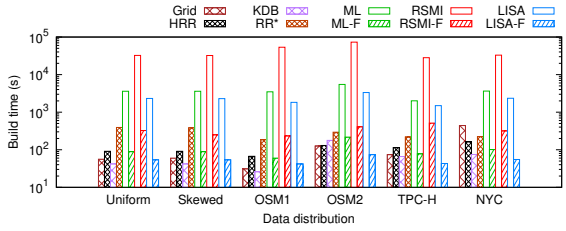


Fig. 8: Build time vs. data distribution.

Varying λ. Figure 9 shows how the build times of the learned indices using ELSI change when varying λ. The traditional indices and the learned indices without ELSI are not impacted by λ. Their performance remain as reported in Figure 8. For ease of comparison, we include the results of the traditional index RR* and the learned index RSMI without using ELSI in Figure 9. As the relative performance of the different indices are similar across the data sets, we only show results on Skewed and OSM1. This also applies to the subsequent studies.

We see that the build times of the ELSI-based indices decrease as λ increases. Guided by Equation 2, the ELSI index building method selector selects more build-time efficient methods given a larger λ. MR is the most frequent choice when $\lambda \geq 0.8$ (as it simply reuses pre-trained models), which brings down the index build times to below those of RR* (except for RSMI-F on OSM1). When λ is small, the query-optimized methods RS, RL and OG are selected. While these incur higher build times, the resultant build times are still much closer to RR* than RSMI without ELSI. This again confirms the effectiveness of ELSI in reducing the index build times.

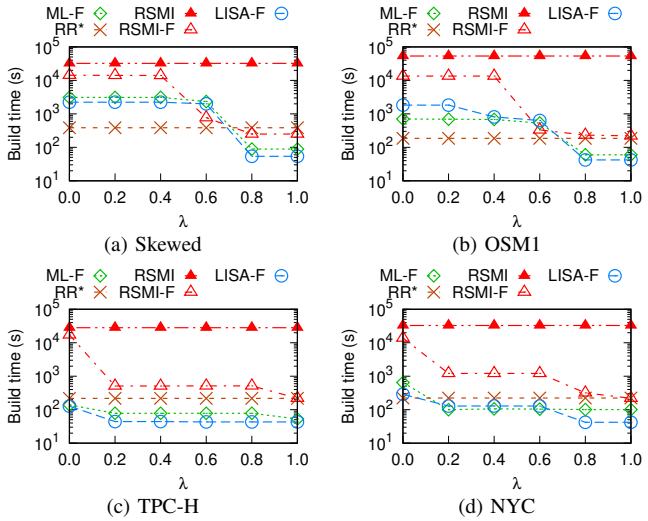


Fig. 9: Build time vs. λ.

G. Query Performance

Next, we report point, window, and k nearest neighbor (kNN) query times of the different indices.

1) Point Queries: We query every point in a data set after indices are built and report the average query time.

Varying the data distribution. As Figure 10 shows, the learned indices outperform the traditional ones except on Uniform where Grid is the fastest. This observation is consistent with those in previous studies on learn spatial indices [8], [9]. Using ELSI yields very similar query times to those of the learned indices when not using ELSI. The point query time increases by at most 14%, i.e., 0.882 μs (ML-F) vs. 0.773 μs (ML) on the OSM1 data set. Recall that our default λ value of 0.8 optimizes towards index build times. MR, which has high query times, is more likely to be chosen by ELSI. RSMI-F and LISA-F still obtain faster point query times than RSMI (on all real data sets) and LISA (on OSM2 and NYC), since real data points are skewed and noisy, and learning an index model on a large set of such data may be impacted by noisy patterns, as discussed earlier. On average, the point query times of the learned indices are not increased by ELSI.

Varying λ. Figure 11 shows that the point query times of the learned indices using ELSI increase slowly with λ. The maximum increase is observed on ML-F, i.e., from 0.741 μs to 0.883 μs on OSM1 for λ equal to 0 vs. 1. RSMI-F and LISA-F outperform both RSMI and RR* on OSM1. RSMI-F, LISA-F, and ML-F all outperform RSMI and RR* on TPC-H. This again confirms the effectiveness of ELSI in reducing index build times while retaining query efficiency, and it justifies using a relatively large λ value to achieve fast index building.

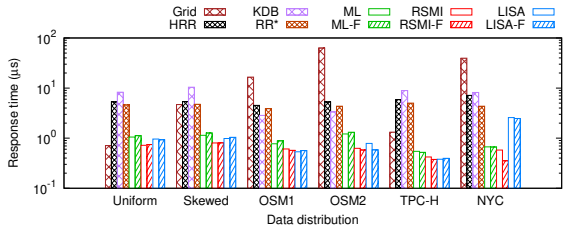
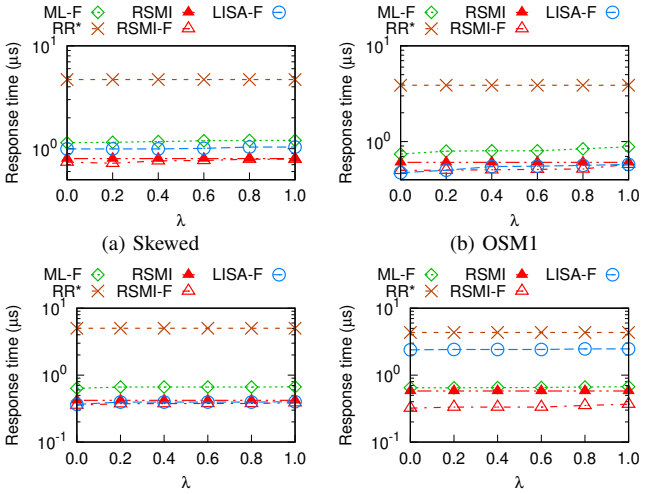


Fig. 10: Point query time vs. data distribution.



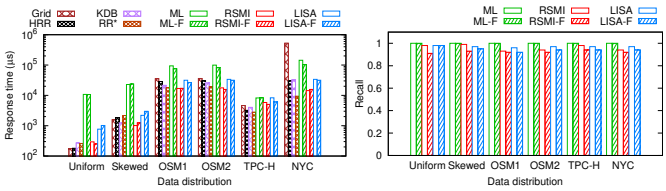
(a) Skewed (b) OSM1

(c) TPC-H (d) NYC

Fig. 11: Point query time vs. λ.

2) Window Queries: We report the average window query performance over 1,000 queries following the data distribution.

Varying data distribution. The learned indices with ELSI have very similar query times to those of the learned indices without ELSI when the query window size is 0.01% of the data space (Figure 12(a)). The worst case is observed on Skewed, where LISA-F is 1.35 times slower than LISA (2,976 μs vs. 2,197 μs), while LISA-F is 1.37 times faster than LISA on TPC-H (6,048 μs vs. 8,329 μs).

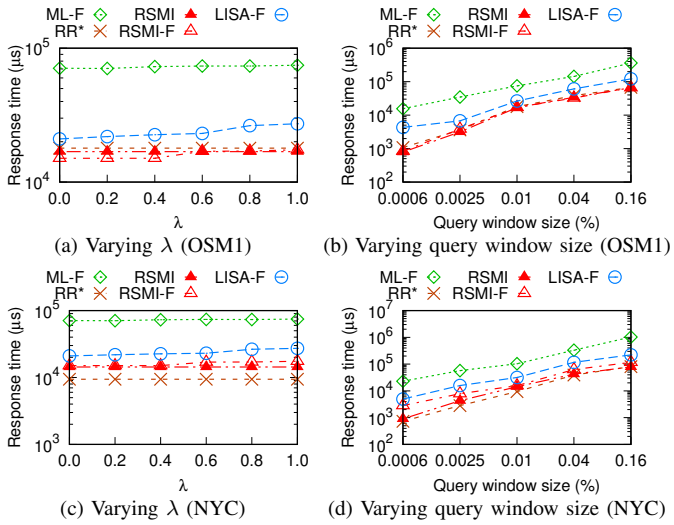


(a) Query time (b) Recall

Fig. 12: Window query time vs. data distribution.

RSMI (by original design) and LISA (when using FFNs) return approximate window query results. Figure 12(b) shows the impact of ELSI on the query *recall* (i.e., the ratio of ground truth points in the returned query results). The recall of both RSMI and LISA drops with ELSI, which is expected. The recall of RSMI-F and LISA-F stays above 91% and 92%, respectively, again confirming the effectiveness of ELSI in retaining the query performance. By design, ML offers accurate results, which are not impacted by ELSI. For brevity, we omit the recall figures for the following window query experiments, since the gaps in the recall remain similar, and the recall of RSMI-F and LISA-F stays over 90%.

Varying λ. Figure 13(a) shows the window query times when varying λ on OSM1. As observed for point queries, the window query times of the ELSI-based indices increase slowly with λ, confirming the robustness of ELSI to variations in λ for window queries. Results on the other data sets exhibit a similar pattern, and their figures are omitted.



(a) Varying λ (OSM1) (b) Varying query window size (OSM1)

(c) Varying λ (NYC) (d) Varying query window size (NYC)

Fig. 13: Window query time vs. λ and window size.

Varying query window size. Figure 13(b) further shows the impact of the query window size. The query times increase with the query window size (from 0.0006% to 0.16% of the data space size) for all indices tested, which is expected. Notably, the query times of the ELSI-based indices do not grow faster than those of RR* and RSMI without ELSI. This indicates the robustness of the ELSI-based indices to variations in the query window size.

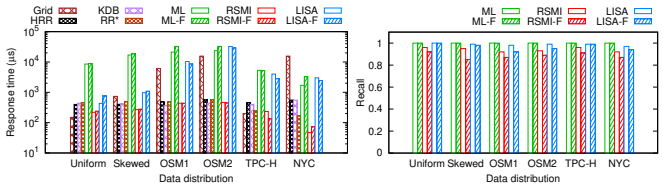
3) KNN Queries: We report average kNN query performance over 1,000 kNN queries that follow the data distribution of the data sets with $k = 25$. We omit results when varying λ and k because they resemble those for window queries.

Varying data distribution. Figure 14(a) shows the kNN query times when $k = 25$. Like the observations on window queries, RSMI and RSMI-F are the fastest, except on Uniform and TPC-H. This is expected because the learned indices use window queries as the basis for kNN queries. Using ELSI retains the query performance, yielding an average increase in the query time of only 3%. In the worst case, ELSI increases the query time from 430 μs (LISA) to 736 μs (LISA-F) on Uniform, which is considered reasonable given the large data set of 128 million points.

In terms of recall, Figure 14(b) shows that ELSI causes a maximum drop of 10% (95% vs. 85% for RSMI and RSMI-F on Skewed). LISA-F drops at most 6% compared with LISA (98% vs. 92% on OSM1), while ML-F stays at 100%.

H. Update Performance

We examine the impact of updates and focus on insertions due to the space limit. We use 10% of the points from OSM1

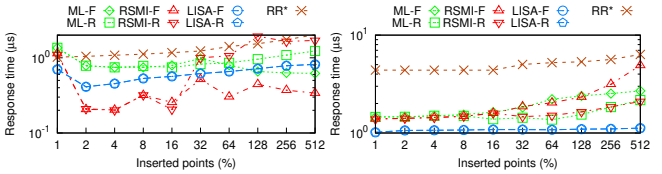


(a) Query time (b) Recall

Fig. 14: KNN query time vs. data distribution.

as the initial set to build an index and use data from Skewed for insertions. We run point queries for all points indexed and 1,000 window queries after the insertions (results of kNN queries are omitted for brevity). We report the average time for each query type.

We show the results of the ELSI-based indices with and without global rebuilding with the update processor. The indices without rebuilds are denoted by ML-F, LISA-F, and RSMI-F; and those with rebuilds are denoted by ML-R, LISA-R, and RSMI-R. We include RR* as it is the traditional index that shows the overall best query performance.



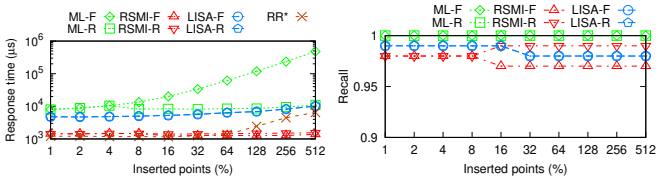
(a) Insertion time vs. insertion ratio (b) Point query time vs. insertion ratio

Fig. 15: Skewed data insertion.

Insertions. Figure 15(a) shows the average insertion time. For RR*, the insertion time grows gradually as more points are inserted due to its self-balancing insertion procedure without rebuilds. For the learned indices, the average insertion times are relatively high for the first 1% of n insertions because most data pages are full after the learned indices are built, meaning that insertions cause the creation of relatively many pages (LISA and RSMI use built-in insertion procedures, and ML uses extra data pages to store points inserted into each index model). Subsequently, LISA-F and LISA-R have stable insertion times that increase gradually as for RR*. The rebuild predictor guides LISA-R not to rebuild as its point query times do not deteriorate (cf. Figure 15(b)). For ML-R and RSMI-R, global rebuilds are triggered after 8% of n insertions (and a rebuild takes up to 133 and 60 seconds, respectively), causing higher insertion times. We will see next that these rebuilds pay off by keeping the query times stable.

Point queries. Figure 15(b) shows the point query times after insertions, which mostly increase as expected. ML-R and RSMI-R are exceptions. Their global rebuilds bring down the query times substantially, e.g., after 512% of n insertions, ML-R and RSMI-R exhibit query times that are 19% and 47% lower than ML-F and RSMI-F, respectively.

Window queries. For window queries after insertions, the query times again increase, as shown in Figure 16(a). Now RR* and RSMI have the lowest query times, which is consistent with observations from the earlier window query experiments. The global rebuilds keep the query times of ML-R lower than those of ML-F. However, RSMI-R is not always faster than RSMI-F. This is because the local index rebuilds of RSMI-F may lead to less accurate structures with fewer points to be scanned for a window query.



(a) Query time vs. insertion ratio (b) Query recall vs. insertion ratio

Fig. 16: Window queries with skewed data insertion.

Figure 16(b) shows the RSMI-R global rebuilds help retain the recall above 97%, while the local rebuilds (RSMI-F) only keep the recall above 90%, further justifying global rebuilds.

I. Summary

We conducted experiments to show (1) the effectiveness of ELSI to select the most appropriate index building method and to determine the right timing for index rebuilds, (2) the efficiency of index building with ELSI, and (3) the efficiency of query processing using the indices built by ELSI. The results show that: (1) ELSI can select the most appropriate index building method with an accuracy over 80%, and it also predicts the right time for index rebuilds, which reduces the query times after data insertions by more than 47%. (2) For index building, using ELSI (i.e., ML-F, RSMI-F, and LISA-F) is one to two orders of magnitude faster than building the learned indices directly (i.e., ML, RSMI, and LISA). This brings down the index building time to the level of those of the traditional indices (Grid, KDB, HRR, and FR*). (3) For query processing, the learned indices built with ELSI share similar times with the respective learned indices built without ELSI, e.g., the kNN query times differ by just 3% on average.

VIII. Conclusions

We propose ELSI, a versatile system that accelerates the building and re-building of a class of learned spatial indices, while retaining the high query efficiency of the indices. ELSI is applicable to learned spatial indices that follow the map-and-sort indexing paradigm and the predict-and-scan query paradigm. More specifically, ELSI encompasses a suite of methods for constructing small and representative training data sets for index learning and rebuilds. Given an input data set, ELSI can adaptively choose a training-set reduction method that produces a learned index with high query efficiency.

In future work, it is of interest to integrate ELSI into open-source systems such as PostgreSQL to make it more broadly available and to gain experience from real applications. We also plan to extend ELSI to support query-aware learned indices such as Flood [24].

Acknowledgment

This work is partially supported by Australian Research Council (ARC) Discovery Project DP230101534.

REFERENCES

[1] OpenStreetMap Stats. (2022) https://www.openstreetmap.org/stats/data_stats.html. Accessed: 2022-07-05.

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

[3] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: An adaptable, symmetric multikey file structure," *ACM Transactions on Database Systems*, vol. 9, no. 1, pp. 38–71, 1984.

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

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

[6] 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.

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

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

[9] 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.

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

[11] 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.

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

[13] Navicat Blog: Oracle Rebuild. (2020) https://www.navicat.com/en/company/aboutus/blog/1303-how-to-tell-when-it-s-time-to-rebuild-indexes-in-oracle. Accessed: 2022-07-05.

[14] P. Antonopoulos, H. Kodavalla, A. Tran, N. Upreti, C. Shah, and M. Sztajno, "Resumable online index rebuild in SQL server," *PVLDB*, vol. 10, no. 12, pp. 1742–1753, 2017.

[15] Y. Li, D. Chen, B. Ding, K. Zeng, and J. Zhou, "A pluggable learned index method via sampling and gap insertion," *CoRR*, vol. abs/2101.00808, 2021.

[16] G. Liu, L. Kulik, X. Ma, and J. Qi, "A lazy approach for efficient index learning," *CoRR*, vol. abs/2102.08081, 2021.

[17] K. C. Claffy, G. C. Polyzos, and H. Braun, "Application of sampling methodologies to network traffic characterization," in *SIGCOMM*, 1993, pp. 194–203.

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

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

[20] 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.

[21] ——, "Packing R-trees with space-filling curves: Theoretical optimality, empirical efficiency, and bulk-loading parallelizability," *ACM Transactions on Database Systems*, vol. 45, no. 3, pp. 14:1–14:47, 2020.

[22] 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," *ACM Transactions on Database Systems*, vol. 30, no. 2, pp. 364–397, 2005.

[23] 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.

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

[25] 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.

[26] Kolmogorov-Smirnov Test. (2022) https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test. Accessed: 2022-07-05.

[27] Earth mover's distance. (2022) https://en.wikipedia.org/wiki/Earth_mover%27s_distance. Accessed: 2022-07-05.

[28] K. Nadjahi, A. Durmus, P. E. Jacob, R. Badeau, and U. Şimşekli, "Fast approximation of the Sliced-Wasserstein distance using concentration of random projections," in *NeurIPS*, 2021, pp. 12 411–12 424.

[29] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing Atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[30] PyTorch. (2016) https://pytorch.org. Accessed: 2022-07-05.

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

[32] OpenStreetMap Data Extracts. (2018) https://download.geofabrik.de/. Accessed: 2022-07-05.

[33] TPC. (2022) TPC-H. http://www.tpc.org/tpch/. Accessed: 2022-09-15.

[34] TLC Trip Record Data. (2022) https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page. Accessed: 2022-09-15.

[35] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "Sosd: A benchmark for learned indexes," *NeurIPS Workshop on Machine Learning for Systems*, 2019.