How Good Are Multi-dimensional Learned Indices? An Experimental Study [Experiment, Analysis & Benchmark]

Qiyu Liu
HKUST
lqy@ust.hk

Yanyan Shen
Shanghai Jiao Tong University
shenyy@sjtu.edu.cn

Lei Chen
HKUST
leichen@cse.ust.hk

ABSTRACT

Efficient indexing is fundamental for multi-dimensional data management and analytics. An emerging tendency is to directly learn the storage layout of multi-dimensional data by simple machine learning models, yielding the concept of *Learned Index*. Compared with the conventional indices used for decades (e.g., *k*d-tree and R-tree variants), learned indices are empirically shown to be both space- and time-efficient on modern architectures. However, there lacks a comprehensive evaluation of existing multi-dimensional learned indices under a unified benchmark, which makes it difficult to decide the suitable index for specific data and queries and further prevents the deployment of learned indices in real application scenarios. In this paper, we present the first empirical study to answer the question that *how good are multi-dimensional learned indices*. Six recently published indices are evaluated under a unified experimental configuration including index implementation, datasets, and query workloads. We thoroughly investigate the evaluation results and discuss the findings that may provide insights for future learned index design.

PVLDB Reference Format:
Qiyu Liu, Yanyan Shen, and Lei Chen. How Good Are Multi-dimensional Learned Indices? An Experimental Study [Experiment, Analysis & Benchmark]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/qyliu-hkust/learnedbench.

1 INTRODUCTION

Multi-dimensional data management and analytics play an important role in various domains such as business intelligence [10], smart transportation [64], neural science [51], and climate studies [15]. As the data volume grows at an exponential speed, indices like R-tree [22] and its variants [2, 4, 26, 54] are designed to speedup data access and query processing over big multi-dimensional databases.

Although traditional indices like B+-tree and R-tree have been studied and embedded into practical DBMSs for decades (e.g., Oracle [27] and PostgreSQL [45]), a recent proposal [30] introduced a new index design paradigm called *Learned Index* based on the observation that data indexing can be modeled as a machine learning problem where the input is a search key and the output is its corresponding location onto the storage. As illustrated in Figure 1, supposing that a set of N keys are sorted and stored as consecutive data pages, a B+-tree can be viewed as a mapping from key x to its page ID. Thus, an error-bounded learned CDF model is functionally equivalent to a B+-tree index. Compared with traditional index structures, learned index is supposed to be both space- and time-efficient as a trained model (e.g., piece-wise linear function) is usually compact and simple for inference.

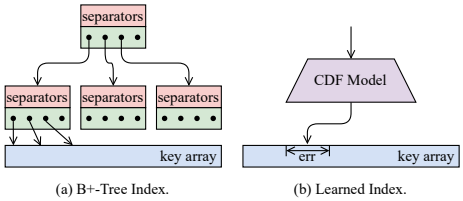


Figure 1: B+-Tree and 1-D learned index.

Inspired by the impressive results obtained from 1-D learned index [17, 30, 35, 58], multi-dimensional learned indices are intensively studied during the past three years such as ZM-Index [57], ML-Index [12], IF-Index [23], RSMI [47], Lisa [33], Flood [38], and Tsunami [14]. These works independently claim that they are empirically more performant than traditional spatial indices like R-tree or *k*d-tree. However, to the best of our knowledge, there lacks a comprehensive evaluation for published multi-dimensional learned indices under a unified experimental configuration, which obscures the impact and future direction of this research field. The limitations of existing experiments are summarized as follows.

First, many newly proposed indices lack enough comparisons with previous studies. Figure 2 visualizes the comparison relationship of existing works where a node refers to an index and an edge $A \rightarrow B$ refers to that index A compares with B in previous literature. From Figure 2, most of the previous works only compare with ZM-Index [57] which combines the space-filling curves and 1-D learned index. However, the original ZM-Index implementation is less optimized (see Section 5.3 for details), meaning that a weak baseline was most frequently compared.

Second, the existing learned indices are not compared under a unified configuration including index implementation, datasets, query workloads, and evaluation metrics. For example, a class of multi-dimensional learned indices [12, 14, 29, 38, 57] utilize 1-D learned index (e.g., RMI [30]) as building blocks, but different implementations are used in different indices, leading to an unfair comparison and unconvincing results on their true performance.

Unlike the experiments in previous studies where only the ZM-Index [57] was compared, in this work, we re-implement and optimize **six** recent multi-dimensional learned indices to perform a

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

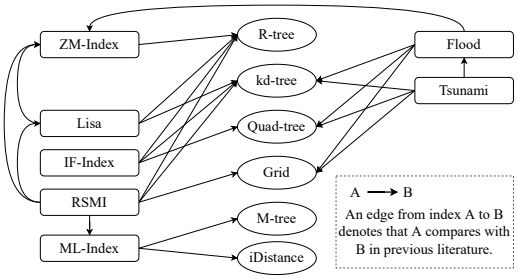


Figure 2: Limited comparison in previous studies.

comprehensive evaluation, which is a nearly complete coverage to the best of our knowledge. To make the comparison fair and make the results more convincing, we standardize the experiment configurations, including the index implementation, datasets, and evaluation query workloads. This also benefits future research as newly proposed multi-dimensional learned indices can be easily evaluated on our benchmark. Though dynamic operations (e.g., insertion and deletion) and IO-efficiency are also important, we focus on the **in-memory** performance of indices over **static** query workloads, which is similar to a recent 1-D learned index benchmark SOSD [35]. Though losing generality to some extent, our empirical study can provide insightful results for in-memory multi-dimensional analytical applications, which are becoming increasingly prominent [53, 60, 63].

The rest of this paper is organized as follows. We review the background of learned index and formulate the multi-dimensional data model and queries of interest in Section 2. We establish a taxonomy and provide an overview of the existing multi-dimensional learned indices in Section 3. Section 4 introduces the index implementation details and experimental setups. The evaluation results and major takeaways are presented in Section 5. Finally, we conclude the paper and discuss future directions in Section 6.

2 PRELIMINARIES AND BACKGROUND

In this section, we first overview the preliminaries and background of multi-dimensional data indexing and learned index structures.

2.1 Multi-dimensional Data Indexing

As adopted by most of the existing works, we consider a collection of N points from d-dimensional Euclidean space, denoted by $O = \{o_1, \cdots, o_N\}$, and we focus on the range query and k nearest neighbour query (kNN) defined as follows.

Definition 2.1 (Range Query). A range query $Range(R)$ takes a d-dimensional hyper-rectangle R as input and returns all the points lie in R, i.e., $\{o | o \in R, o \in O\}$.

Definition 2.2 (k Nearest Neighbour Query). A k nearest neighbour query $kNN(q, k)$ retrieves k objects from O whose distances to q are ranked in ascending order. Formally, for $\forall o \in kNN(q, k)$, $\nexists o' \in O / kNN(q, k)$ such that $||o', q||_2 \leq ||o, q||_2$.

Indexing and query processing over multi-dimensional data have been studied and applied in commercial DBMSs for decades. For low- or medium-dimensional data, traditional indices include R-tree [22] and its variants like R*-tree [4], STR-tree [32] and Hilbert R-tree [26], kd-tree [5], Grid File [39], etc. For high-dimensional space, due to the curse of dimensionality, data is inevitably becoming sparse, and query processing like kNN query based on the aforementioned indices will be no better than a naive linear scan. To this end, pivot-based methods are generally adopted to index high-dimensional data, e.g., iDistance [25], vantage-point tree [62] (VP-tree), MVP-tree [9], etc. A recent proposal [11] surveyed and evaluated the family of pivot-based indices on high-dimensional query processing.

2.2 Learned Data Indexing

A 1-dimensional learned index is intrinsically an error-bounded CDF model (scaled by data size N). In the seminal work [30], Kraska et al. proposed the first learned index RMI that is empirically shown to be Pareto optimal compared with a B+-tree index. Following RMI, Kip et al. proposed a simple learned index called RadixSpline [28] that requires only a single pass of data to construct. Furthermore, PGM-Index [17] adopted the optimal piece-wise linear approximation [41] as the underlying CDF model, leading to strong theoretical results on the space and time complexity. To handle dynamic operations like key insertion and deletion, Ding et al. proposed ALEX [13] by using a gapped array in their structure to handle record updates; LIPP [58] further improved the update efficiency by reducing the last-mile search error in leaf nodes. More discussions and comparisons about 1-D learned indices can be found in a benchmark paper SOSD [35]. Besides data indexing, there are emerging attempts of embedding learned models into conventional data structure and algorithm design, e.g., learned Bloom filters [30, 34, 36], learned sorting [31], learned data compression [6], etc.

Similar to that a R-tree is a multi-dimensional analog to a B+-tree index, it is natural to extend the 1-dimensional learned index to multi-dimensional datasets by learning the mapping from multi-dimensional keys to their storage location, and designing multi-dimensional learned indices has rapidly became a promising research direction during the past three years. Typical works in this area include ZM-Index [57], ML-Index [12], IF-Index [23], RSMI [47], Lisa [33], Flood [38], and Tsunami [14]. Besides, a recent system SageDB [29] also incorporated a learned grid index, which can be viewed as a simplified version of Flood [38]. The details of these indices will be discussed in Section 3. As we will reveal in this work, existing multi-dimensional learned indices mainly target on static read-only workloads, and the query support on the top of these indices are still preliminary (majorly range query and kNN query), remaining a tremendous number of research opportunities (Section 5 and Section 6).

3 MULTI-DIMENSIONAL LEARNED INDICES

This section provides an overview of the major multi-dimensional learned index structures to be evaluated. We first dive into the data layouts adopted by different indices in Section 3.1. Then, we provide an index taxonomy in Section 3.2 and introduce indices in each category in Section 3.3–3.5. Table 1 summarizes the major technical features of existing works.

3.1 Data Layout

We first discuss the multi-dimensional data layout which is the major criterion to establish our taxonomy for learned indices. Data

Table 1: Overview of existing multi-dimensional learned index structures.

Index	ZM-Index	ML-Index	LISA	IF-Index	RSMI	Flood	Tsunami
Reference	[57]	[12]	[33]	[23]	[47]	[38]	[14]
Type	projection	projection	projection	augmentation	augmentation	grid	grid
Data Ordering	Z-curve	proj. function	proj. function	selected Dim	Z-curve	selected Dim	selected Dim
Data Layout	order-based	order-based	order-based	space partition	space partition	grid	grid
Model	RMI	RMI	piece-wise linear	linear interpolation	MLP	RMI	RMI
Model Training	algorithmic	algorithmic	numpy	algorithmic	pytorch	algorithmic	algorithmic
Updatable	×	×	✓	×	✓	×	×
Support Dim.	≥ 2	≥ 2	≥ 2	≥ 2	2	≥ 2	≥ 2
Range Query	✓	✓	✓	✓	✓	✓	✓
kNN Query	×	✓	✓	×	✓	×	×

layout specifies how an index organizes the data points onto the storage, either disk or memory, which plays an important role in multi-dimensional index design. As shown in Table 1, we observe that the existing learned indices usually employ one of the tree types of data layouts, *order-based layout*, *space partition-based layout*, and *grid-based layout*. For order-based layout, as the name implies, the order of points on storage is consistent with a pre-defined sorting order. Different from the 1-D case, there is no intrinsic sorting order for multi-dimensional data, and existing works usually select a sorting dimension or employ the space-filling curves (e.g., Z-order curve [48]) to sort data. The space partition-based layout recursively divides the data space under some strategy (e.g., the middle-point strategy in kd-tree [5]) until a partition threshold is reached. In this case, data points within the same partition are grouped together and sequentially materialized to the storage. The grid-based layout can be regarded as a special case of the partition-based layout as it divides the space into grid cells.

3.2 Taxonomy

Based on the difference in data layouts and how learned CDF models are integrated, we classify the existing multi-dimensional learned indices into three categories: projection-based index, augmentation-based index, and grid-based index.

Projection-based Index adopts a projection function to map k-dimensional keys to 1-D values, and then train a 1-D learned index (e.g., RMI [30] or PGM-Index [17]) over the mapped values. To preserve the spatial locality, space-filling curves like Z-order curve or Hilbert curve are common choices for the projection function. Such idea is not new in conventional spatial index design (e.g., UB-tree [49] and Hilbert R-tree [26]). The projection-based indices include ZM-Index [57], SageDB [29], and ML-Index [12].

Augmentation-based Index extends the traditional index structures that are based on recursive space partitioning but augment the ordinary node search with model-based search. Such kind of indices employ similar space partition and node split strategies from the existing multi-dimensional indices like R-tree and kd-tree, thus inheriting the high generality and wide applications. The augmentation-based indices include IF-Index [23] and RSMI [47].

Grid-based Index employs grid as the data layout. Different from the ordinary grid indices, the learned grid index does not store grid cells; instead, learned CDF functions over a set of selected partition dimensions are trained to locate the correct grid cell. Suppose that the j-th dimension in the grid is partitioned into m buckets, a point o will be placed in the $\lfloor CDF_j(o_j) \cdot m \rfloor$-th bucket. The grid-based indices include Flood [38], Tsunami [14], and Lisa [33]. Lisa [33] can also be classified into the projection-based index as it utilizes a mapping function to order the grid cells.

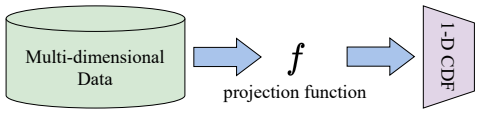


Figure 3: Workflow of projection-based indices.

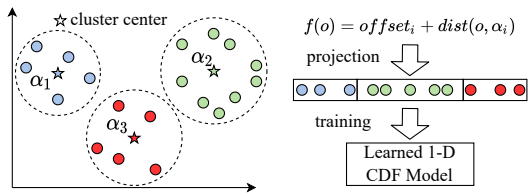
Recent studies like Qd-tree [61] and RLR-tree [21] also leveraged learning techniques, especially deep reinforcement learning (RL), to construct multi-dimensional data index. However, their indexing layer is still the traditional index like R-tree, and most importantly, they use RL for finding better data layout, not for locating data records on storage. Thus, they are out of the scope of learned index and not evaluated in our work.

3.3 Projection-based Index

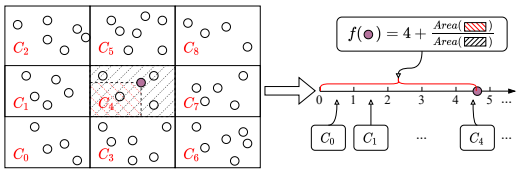
Figure 3 shows the basic workflow of projection-based indices. For a d-dimensional dataset O, a projection function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ is applied to convert O to a set of 1-D values O'. Then, O' is sorted and stored in consecutive data pages, and a 1-D learned index (e.g., RMI) is built to serve the *approximate* mapping to storage location (i.e., block ID or index in a dense array). Similar to the 1-D learned indices, to ensure the correctness of point search, the projection function f should be also a monotonic mapping, i.e., for two points o and o' where o' dominates o on each dimension, $f(o) \leq f(o')$. Then, any range search query can be transformed to 1-D interval search over the mapped values. The major difference of existing works in this class is the choice of projection function f.

ZM-Index [57] is the first multi-dimensional learned index where the Z-order curve is chosen as the projection function. To efficiently compute the Z-addresses, the data space should be partitioned into grids such that the bit interleaving technique can be used [49]. To process a range query, the query box is first decomposed into intervals of Z-addresses using the same technique in UB-tree [49], and then the trained CDF model is queried to efficiently find the corresponding storage location.

ML-Index [12] employs an improved iDistance function to project the multi-dimensional data, which is usually used to index high dimensional data for efficient nearest neighbour search [25]. As shown in Figure 4a, given a set of selected reference points (RP) $\alpha_1, \cdots, \alpha_m$ (e.g., obtained using the k-means algorithm), the input



(a) Illustration of the projection function of ML-Index where k-means centers are used as reference points.



(b) Illustration of the projection function of LISA based on a 3×3 grid partition. Note, the Lebesgue measure in 2-D space is the area of a rectangular region.

Figure 4: Illustration of ML-Index [12] and LISA [33].

data O are partitioned into m partitions based on the distance to each RP. For any point $o \in O$, supposing that α_i is the closest RP to o, the ML-Index adopts the following projection function,

$$f(o) = offset_i + dist(o, \alpha_i), offset_i = \sum_{j<i} \max_{o' \in O_j} dist(o', \alpha_j) \quad (1)$$

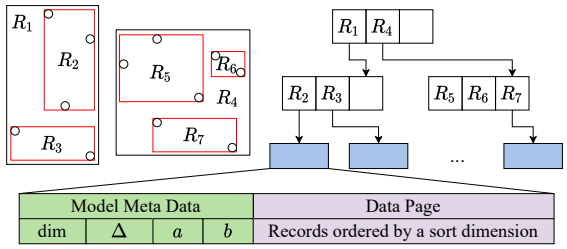
where $O_j = \{o | \alpha_j$ is the closest RP to $o, o \in O\}$.

Compared with the original iDistance method, i.e., $iDist(o) = i \cdot C + dist(o, \alpha_i)$, Eq. (1) eliminates the overlap between different data partitions and reduces the gaps between consecutive partitions, making the learning of a CDF model on the mapped keys much easier. The query processing (range queries or kNN queries) on ML-Index is similar to the iDistance [25] method, where a B+-tree is constructed to maintain the iDistance values. Since the computation of the projection function Eq. (1) only requires a valid distance metric $dist(\cdot, \cdot)$, the ML-Index is also available on data from general metric spaces (e.g., strings and graphs). However, as all the other learned indices do not support metric space indexing, we focus on the performance of ML-Index in Euclidean space and leave extending to general metric space as an interesting future work.

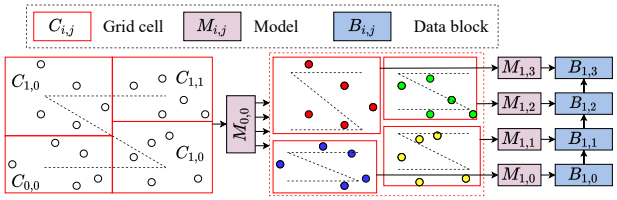
LISA [33] majorly attacked that the space-filling curve-based projection usually accesses data blocks that are irrelevant to the query rectangle. To solve this issue, LISA employs a grid-based projection method. For a d-dimensional dataset, data points are first partitioned into $T_1 \times T_2 \cdots \times T_d$ equal-depth grid cells, and each cell C is associated with a unique ID t, i.e., $C_t = [\theta_l^{(1)}, \theta_h^{(1)}) \times \cdots \times [\theta_l^{(d)}, \theta_h^{(d)})$ and $t = (((i_1 \times T_2 + i_2) \times T_2 + i_3) \times \cdots) \times T_d + i_d$ where $\theta_l^{(i)}, \theta_h^{(i)}$ are the grid cell boundaries on the i-th dimension. Then, for any point o in C_t, the projection function is defined as follows,

$$f(o) = t + \frac{\lambda(H_t)}{\lambda(C_t)}, H_t = [\theta_l^{(1)}, o^{(1)}) \times \cdots \times [\theta_l^{(d)}, o^{(d)}), \quad (2)$$

where $\lambda(\cdot)$ is the Lebesgue measure (area in 2-D space). The projection function of LISA is illustrated in Figure 4b where the areas of red and black dashed regions are the Lebesgue measures of C_4 and H_4 for the point in purple. Intuitively, Eq. (2) is also similar to the iDistance method [25] where grid cells can be viewed as reference points. Clearly, points falling into cell C_t will be mapped to the same interval $[t, t+1)$, which preserves the spatial locality.



(a) Illustration of IF-Index where dim is the selected sort dimension, Δ is the maximum prediction error, and a, b are the slope and interception of the linear model.



(b) Illustration of RSMI where the partition threshold $N' = 5$. Note, that each data block stores a pointer to the next block for fast scan.

Figure 5: Illustration of (a) IF-Index [23] and (b) RSMI [47].

After the grid construction, all the points are mapped to 1-D space by using Eq. (2) and partitioned to shards, and a model called *shard prediction function* (SP), which should be a monotonic function, is trained to map each point to their shard ID. Finally, points belonging to the same shard are stored into data pages, and a local model (i.e., 1-D learned index) is trained to locate the correct data page. As reported in [33], LISA stores all grid boundaries in memory to efficiently compute Eq. (2), which means its original implementation is hard to scale to high dimensional datasets due to the exponential number of grid boundaries to be stored.

3.4 Augmentation-based Index

We then introduce the augmentation-based indices, where learned models are plugged into recursive index structures (e.g., R-tree or kd-tree) to accelerate the search efficiency.

IF-Index [12] replaces the leaf node search procedure in existing tree structures (e.g., R-tree) with 1-D learned index-based search. Figure 5a illustrates the structure of IF-Index based on R-tree, where the non-leaf nodes are constructed by following an ordinary R-tree but the leaf nodes store not only the corresponding data page but also the model metadata. The model metadata contains the dimension dim used to order the points in data page and a linear interpolation model to predict the search key location on data page. To find the best sort dimension, for each leaf node, IF-Index evaluates the interpolation cost on each dimension, taking a time complexity of $O(dN' \log N')$ where d is the number of dimensions and N' is the number of points contained in this leaf node.

RSMI [47] takes a further step by using the model-based search in both leaf and non-leaf nodes. As shown in Figure 5b, to construct the RSMI index, spatial points are first partitioned into $2^{\lfloor \log_4 N'/B \rfloor} \times 2^{\lfloor \log_4 N'/B \rfloor}$ equal-depth grid cells where B is the block size and N' is the partition threshold parameter. In practice, N' is assumed to be the number of points that a learned function can achieve high

accuracy. Then, grid cells are ordered using SFC, and a learned model is trained to map each grid cell to its sorting order (e.g., $M_{0,0}$ in Figure 5b). As each grid cell can still contain many points (i.e., $\gg N'$), the above data partitioning and model training procedures are recursively invoked until each partition has at most N' points. After completing the partitioning, the data points are packed into data blocks based on the order of rank space Z-order curve, and finally, leaf models are trained to predict the corresponding block ID for each point (e.g., $M_{1,0} \sim M_{1,3}$ in Figure 5b).

3.5 Grid-based Index

The grid-based indices Flood [38] and Tsunami [14] target on learning compact multi-dimensional grids to efficiently process orthogonal range predicates.

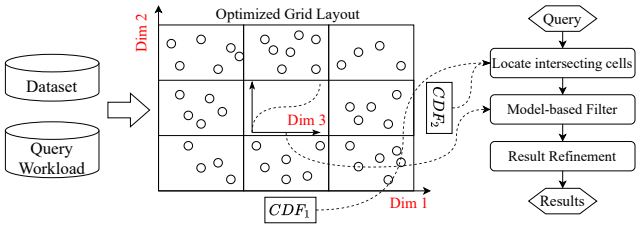
Flood [38] adopts the common grid index as the data layout. To construct Flood on a d-dimensional dataset, a sort dimension is first chosen for ordering the data within each grid cell, and the remaining $d-1$ dimensions are adopted to overlay a grid. Assume, w.l.o.g., that the d-th dimension is the sort dimension. Different from an ordinary grid index, Flood uses learned CDF models (i.e., 1D learned index) to construct grid partitions. Specifically, a Flood grid is typically a multi-dimensional array G of cells, and the corresponding cell of an arbitrary point o in G is

$$G(o) = (\lfloor CDF_1(o_1) \cdot K_1 \rfloor, \cdots, \lfloor CDF_{d-1}(o_{d-1}) \cdot K_{d-1} \rfloor), \quad (3)$$

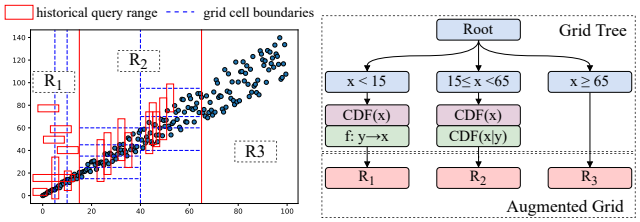
where $CDF_i(\cdot)$ is the CDF trained on the i-th dimensional values of the dataset, and K_i is the partition number for the i-th dimension. To enable faster refinement of a range filter, for each bucket, an auxiliary CDF model is also trained using the sort dimension.

As shown in Figure 6a, Flood establishes a cost model and uses a historical query workload to select the sorting dimension and tune the hyper-parameters (e.g., partition number K_i). To process a range query, cells intersecting the query hyper-rectangle are first retrieved by Eq. (3), and then their intra-cell CDF models are queried to apply efficient filtering based on the range predicates.

Tsunami [14] further improves Flood's performance on correlated data and skewed query workloads. As shown in Figure 6b, Tsunami's structure consists of two parts: a grid tree that adapts to skewed query workloads and an augmented grid index that is optimized to capture data correlations. Similar to a kd-tree, the grid tree is also a space partitioning tree based on a selected subset of dimensions, which divides the whole space into several disjoint regions such that the query skew of a historical workload is reduced within each region (e.g., R_1, R_2, R_3 in Figure 6b). Then, an augmented grid is constructed for each region. Different from Flood, where a grid cell is determined by independent CDF models on each dimension, Tsunami captures correlation patterns using both functional dependency ($f : y \rightarrow x$ in R_1) and conditional CDF model ($CDF(x|y)$ in R_2). To determine the granularity of each augmented grid, historical workloads are considered such that frequently queried regions are intensively partitioned (e.g., region R_2) while less queried regions are mildly partitioned (e.g., region R_1 and R_3). The range query processing on Tsunami is similar to that of Flood, i.e., locating the intersected regions and grids and then refining the results. Both Flood and Tsunami set up a periodical mechanism to monitor the workload shifts. Once the workload distribution significantly differs



(a) The figure shows the Flood index framework on a 3-D dataset where dim1 and dim2 are used to generate a grid layout and dim3 is used to sort the data within each grid cell.



(b) Illustration of a Tsunami index built on a 2-D correlated dataset and tuned by skewed query workloads. The red solid lines refer to the partition boundaries of the grid tree, and the blue dashed lines refer to the bucket boundaries of augmented grids.

Figure 6: Illustration of (a) Flood [38] and (b) Tsunami [14].

from the one used to construct the index, the whole grid layout will be re-tuned using the newly collected workload characteristics.

LISA can be also viewed as a learned grid index. However, different from Flood and Tsunami where the underlying storage layer is still a grid index, LISA encodes all the grid information into a projection function (i.e., Eq. (2)) and ordered data based on this mapping. Thus, we classify LISA into the projection-based index.

3.6 Discussion

In this section, we discuss several omitted details related to the design choices of multi-dimensional learned indices, including index update, model selection, and model training.

Index Update. Most of the learned indices except RSMI [47] and LISA [33] do not support dynamic operations like insertion and deletion. The major bottleneck comes from the hardness of updating outdated models. An interesting finding is that both RSMI and LISA adopt a model-based data layout to handle updates. To insert (or delete) a record, RSMI (or LISA) simply queries the learned models to obtain the ID of block to be inserted and finalize the insertion if the block is not full. In this case, the block ID predicted by the model is always regarded to be *correct* due to the model-based layout. Such strategy to handle dynamic operations is similar to that of updatable 1-D learned indices ALEX [13] and LIPP [58] where new keys are inserted to an array with gaps based on the model's predictions. However, existing indices merely consider the query performance decay problem when the data distribution significantly shifts, and in the worst case, re-constructing the index is inevitable.

Model Selection. A learned index can be conceptually regarded as a combination "data layout + learned model". Though playing an important role, in most cases, the underlying learned models in existing works can be safely replaced by another one as long as it is error-bounded and monotonic to ensure query correctness. Figure 7 roughly depicts the popular learned model choices used in index

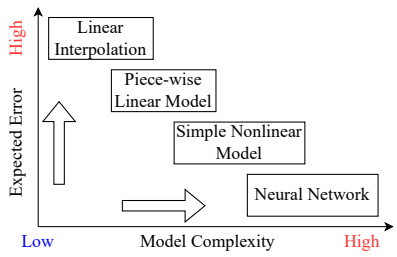


Figure 7: Characteristics of different model choices.

design. Clearly, simple models like linear interpolation or piece-wise linear approximation (PLA) are efficient to learn and query with a sacrifice of limited model capacity, leading to potentially higher error, and vice versa. This increases the freedom of index design and enables a series of trade-offs. For example, a sophisticated model is usually costly to learn and store; however, it can effectively filter out unnecessary points to be examined when processing a range query.

Model Training and Inference. Besides model selection, effective and efficient model training and inference also play an important role in the learned index framework. As shown in Table 1, existing works employ either self-designed algorithms (i.e., algorithmic approach) or utilize mature libraries (usually based on Python) like Pytorch [46], Tensorflow [55], or Numpy [40]. Although these external libraries show much higher flexibility in terms of model design choices, they usually require extra runtime overheads (e.g., `libtorch` for Pytorch) and suffer from longer training time. On the other hand, well-designed algorithmic approaches are usually more efficient for training and inference with a sacrifice of model design flexibility, e.g., the error-bounded piece-wise linear approximation [41] used in PGM-Index [17] and the top-down training used in RMI [30]. From our evaluation results (Section 5), compared with indices internally using PGM-Index, the Pytorch-based solution is not significantly better in terms of query processing efficiency but takes $\sim$ 9000$\times$ longer time to train the model.

4 EXPERIMENT SETUP

This section introduces the index implementation details and experiment setups. We implement the whole benchmark in C++ where the optimization level is set to O3. All experiments are performed on a Ubuntu Linux machine with Intel(R) Core(TM) i7-10700K CPU and 32 GB memory. We disable CPU's Turbo Boost feature and lock the frequency to 4.67 GHz to minimize the influence of dynamic frequency scaling.

4.1 Index Implementation Details

We first introduce the implementation details of all the compared multi-dimensional indices (learned and non-learned). Table 2 summarizes all 12 compared methods where 10 indices support range queries and 7 indices support kNN queries.

`ZMI`: the ZM-Index [57] that combines the Z-order curve and 1-D learned index. To compute the Z-order curve values, for a dataset of N d-dimensional points, we uniformly partition each dimension into $N^{1/d}$ buckets, implying an equal-width grid layout of N buckets. A fine-grained grid can improve the pruning power of unnecessary data access but increase the space and time efficiency

Table 2: Summary of compared indices.

Index	Type	Range	kNN
`ZMI`	learned	✓	✓
`MLI`	learned	✓	✓
`IFI`	learned	✓	✗
`RSMI`	learned	✓	✓
`LISA`	learned	✓	✓
`Flood`	learned	✓	✗
`STRtree`	tree	✓	✓
`R*tree`	tree	✓	✓
`kdtree`	tree	✗	✓
`qdtree`	tree	✓	✗
`ANN`	tree	✗	✓
UG and EDG	grid	✓	✗

of index construction and query processing. We try different grid partition resolutions and find that a $N^{1/d} \times \cdots \times N^{1/d}$ uniform grid can robustly achieve the best performance on different datasets.

`MLI`: the ML-Index [12] that combines the improved iDistance function (Eq. (1)) and 1-D learned index. To appropriately set the reference points, a Lloyd's k-means algorithm is invoked with k-means++ initialization [3]. According to [25], for pivots-based indices, the query efficiency improvement is minor when the number of reference points (P) is larger than 25. In our implementation, the partition number P is set to 20 for datasets of size <20M and 40 for datasets of size >20M.

`LISA`: the LISA index [33] that employs the grid-based projection function (Eq. (2)). The original implementation of `LISA` is in Python and highly depends on Numpy, and to make the comparison fair, we re-implement its index structure at our best. To compute Eq. (2), we construct an equal-depth grid where each grid cell contains roughly $B = 2000$ points. By such setting, the range of Eq. (2) is in $[0, N/B + 1]$.

`IFI`: the IF-Index [23] based on R-tree. The original IF-Index employs linear interpolation to estimate the key location, whose prediction error is large, especially for non-uniform data. To reduce the error and thus improve the query efficiency, we train linear models using the least square method, yielding a 2%–15% performance improvement with a sacrifice of 10%–20% more training time. Besides, through our practice, we find that it is unnecessary to select different sort dimensions for each leaf node as we do not know the distribution of query rectangles when building the index. The capacities of leaf and non-leaf nodes are set to 1000 and 64, which can achieve the best trade-off between index size and performance on different datasets.

`RSMI`: the recursive spatial model index [47]. We choose their original implementation and follow the same index tuning strategies as discussed in their paper (e.g., selection of the partition threshold parameter N'). Note that, `RSMI` only supports 2-D datasets as it mainly targets spatial applications. The model training and inference of `RSMI` is based on Pytorch [46]. Thus, to make it fair when comparing with other indices, we choose the CPU-only version of Pytorch but enable the multi-threading to accelerate the model training (otherwise it fails to terminate in 5 hours for a 20M dataset).

`Flood`: the learned grid index [38]. As `Flood` is not open-sourced at current stage, we implement `Flood` at our best. The original `Flood` index requires a query workload to tune its hyper-parameters (e.g., selection of the sort dimension). However, since other indices are

Table 3: Summary of datasets.

Dataset	Type	Size	#Points	#Dim.	Skew
Uniform	synthetic	N.A.	5–100M	2–8	low
Normal	synthetic	N.A.	5–100M	2–8	mid
Lognormal	synthetic	N.A.	5–100M	2–8	high
FourSquare	real	181 MB	3.7M	2	mid
Toronto3d	real	1.52 GB	21M	3	high
OSM	real	3.1 GB	63 Million	2	mid

not optimized using query histories, we do not implement this part to make the comparison fair. Similarly, `Tsunami` [14] is a fully workload-driven index and thus is not compared in this benchmark due to the same reason. To avoid the exponentially growing number of grid cells w.r.t. data dimension, the partition number on each dimension except the sort dimension is set to $(N/B)^{1/(d-1)}$ where $B = 2000$ is the (rough) number of points in each cell.

Except `RSMI` and `IFI` that employ deep learning and linear regression, we choose the PGMIndex [17] as the underlying 1-D learned index of the multi-dimensional learned indices (i.e., `ZMI`, `MLI`, `LISA` and `Flood`) to unify their implementations. According to a recent benchmark [35], a well-optimized RMI index [30] can slightly outperform the PGMIndex. The reasons that we do not choose RMI are twofold. First, the state-of-the-art RMI implementation is an "index compiler" that takes a dataset as input and generates index data and header files, which is less flexible to be embedded into another index. Second, the parameter tuning of PGMIndex is much easier than RMI (only an error parameter is required). The default error threshold ϵ for PGMIndex is set to 64, which is an empirically robust and optimized value for datasets of different configurations. We perform extensive studies by varying ϵ where the results and discussions can be found in Section 5.6.

Besides learned indices, we also compare 8 non-learned baselines that are commonly used in practice.

`FullScan`: the simple sequential scan baseline.

`R*tree`: the R*-tree index [4] that optimizes the R-tree structure by minimizing the leaf node overlap.

`STRTree`: the R-tree index with STR bulk-loading strategy [32], which is a simple but effective R-tree variant. As the indexed geometries are multi-dimensional points, there is no overlap for the MBRs of leaf nodes. For both `R*tree` and `STRTree`, we choose the implementations from the Boost geometry library [7], and the node capacity (i.e., fanout) is set to 128 based on a benchmark on popular spatial libraries [44].

`kdtree`: the kd-tree index [5] that recursively partition the space by the median of each dimension in a round-robin fashion. We adopt the implementation in the nanoflann project [37].

`ANN`: another kd-tree variant provided in `ANN` [1], a library for efficient exact and approximate kNN search. For both `ANN` and `kdtree`, the leaf node size (i.e., the threshold of a kd-tree turns to use brute-force search) is set to 16.

`qdtree`: the quad-tree index which can be viewed as an adaptive grid index. The implementation from GEOS library [19] is adopted. Note that `quadtree` only supports range queries on 2-D datasets.

`UG`: a uniform grid index (a.k.a., equal-width grid). The partition number of each dimension is set to $(N/2000)^{1/d}$.

`EDG`: a (roughly) equal-depth grid index where each dimension is sorted and partitioned into $(N/2000)^{1/d}$ parts of equal size. `EDG` can be regarded as the non-learned version of the `Flood` index. Compared with the uniform grid, `EDG` can better deal with skewed data distribution with a sacrifice of higher construction cost.

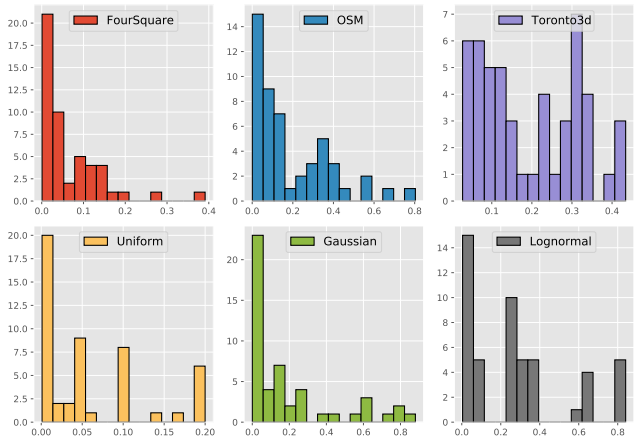


Figure 8: Distribution of range queries. The x-axis and y-axis refer to the query selectivity and number of queries.

4.2 Datasets and Query Generation

We evaluate all the compared methods in Section 4.1 on both real and synthetic datasets. Table 3 summarizes the datasets statistics.

Real Datasets. We adopt three generally used real datasets.

- **FourSquare** is a location dataset extracted from a geo-social network. Each data point represents the coordinates of a co-location event of two users in the social network [18].
- **Toronto3d** is a public dataset of LiDAR images of urban roadways [56]. We extract the spatial coordinates (i.e., X, Y, Z) from the raw data to form a 3-D point cloud.
- **OSM** is a recent dump of OpenStreatMap [42]. We only use the point objects (i.e., pair of latitude and longitude), and other geometries such as polygons and line-strings are excluded.

Synthetic Datasets. To test the scalability in terms of data size and dimension, we sample random points from the uniform and log-normal distributions where the dimension correlation is ignored. We vary the scale factors of each distribution (e.g., the standard deviation of normal distribution) to make sure that the density of generated data is similar. For all synthetic datasets, the default #Points is 20 million, and #Dim. is 2 as some compared methods like `RSMI` and `qdtree` only support 2-D datasets.

Range Query Generation. To generate range queries for evaluation, we first randomly sample $S = 100$ points from a given dataset as the lower corners of query rectangles; then, for each dimension $i = 1, \cdots, d$, a random width δ_i is added to the lower corner to form the upper corner. For a generated query box R, the selectivity of R is the percentage of points that R is expected to cover, i.e.,

$$\text{Selectivity}(R) = \frac{\#\text{Points covered by } R}{\#\text{Points}}. \tag{4}$$

Figure 8 presents the selectivity distributions of generated range queries on different datasets. The default selectivity in the subsequent evaluations is 1%.

kNN Query Generation. A kNN query is a pair of query point q and result number k. For each $k \in \{1, 10, 100, 1000, 10000\}$, we

Table 4: Index construction time (seconds) on real datasets and synthetic datasets of default configurations.

Index	ZMI	MLI	IFI	RSMI	LISA	Flood	STR	R*tree	kdtree	qdtree	ANN	UG	EDG
Uniform	1.89	93.54	4.88	8978	7.46	5.34	3.33	128.92	28.61	2.15	21.61	0.73	4.17
Normal	1.63	90.49	5.04	13017	7.45	5.44	3.25	128.80	27.96	2.46	20.43	0.51	4.16
Lognormal	1.13	83.99	5.08	10311	7.48	5.57	3.22	120.46	29.03	2.77	20.75	0.34	4.15
FourSquare	0.17	14.48	0.71	996.8	1.25	0.99	0.47	18.22	3.87	0.619	3.21	0.07	0.63
Toronto3d	1.09	83.13	4.61	N.A.	7.55	9.80	3.27	155.34	14.66	12.587	2.99	0.48	4.53
OSM	3.08	209.22	10.15	N.A.	15.62	18.05	6.93	389.35	30.53	21.20	22.33	1.24	8.442

Table 5: Index memory overhead (MB) on real datasets and synthetic datasets of default configurations.

Index	ZMI	MLI	IFI	RSMI	LISA	Flood	STR	R*tree	kdtree	qdtree	ANN	UG	EDG
Uniform	152.61	152.63	153.71	21.5	153.12	152.92	314.82	448.09	273.1	1668.7	276.4	152.81	152.81
Normal	152.64	152.63	153.71	22.4	153.12	152.96	314.82	445.07	271.2	1695.4	274.8	152.81	152.81
Lognormal	152.62	152.63	153.71	29.8	153.12	152.96	314.82	440.13	268.7	1723.0	271.5	152.81	152.81
FourSquare	28.11	28.13	28.29	21.7	28.17	28.36	57.93	84.08	44.8	294.6	44.2	28.16	28.16
Toronto3d	164.82	164.62	165.93	N.A.	164.84	167.35	507.94	746.16	281.5	285.51	291.2	164.72	164.72
OSM	480.57	478.92	482.13	N.A.	480.60	483.25	987.51	1500.54	678.6	5061.6	683.3	480.06	480.06

randomly pick $S = 20$ points from each dataset to generate a set of kNN queries. The default k in the evaluation is 1000.

In the subsequent evaluations, we report the average query processing time of different selectivities for range queries and different k for kNN queries. Note that, we do not iterate each query many times, which will warm the CPU cache and thus possibly reduce the query processing time. Such a setting can well simulate the index usage scenarios in real-world applications.

5 EXPERIMENT RESULTS

In this section, we report the evaluation results to answer the central question, *how good are multi-dimensional learned indices*. In particular, we are interested in the following factors.

- **Q1: Construction Time**: whether the construction time of learned indices is as fast as the non-learned baselines;
- **Q2: Space Cost**: whether the space cost of learned indices is significantly lower than the non-learned baselines;
- **Q3: Range Query Efficiency**: Whether the learned indices can outperform baselines in terms of range query processing;
- **Q4: kNN Query Efficiency**: whether the learned indices can outperform baselines in terms of kNN query processing;
- **Q5: Scalability**: whether the learned indices can well scale to datasets of larger size and higher dimension.

5.1 Index Construction Time (Q1)

We first report the index construction time on real datasets and synthetic datasets of the default configuration (#Points=20M and #Dim=2) in Table 4. We set a threshold of 5 hours that if the program cannot finish in 5 hours, it will be killed and N.A. will be reported.

As shown in Table 4, the projection-based indices except MLI are comparable with or even faster than the non-learned baselines in terms of index construction time. This is because we unify the underlying 1-D learned index of these indices as the PGMIndex, which requires only one pass of data to obtain the minimized number of error-bounded line segments [17, 35]. MLI is slower because finding reference points via k-means is costly, which occupies $> 95\%$ of the total index building time. Surprisingly, ZMI is $\sim 2\times$ faster than the bulk-loading R-tree index STR and is only slower than the simple uniform grid index UG. This is because in addition, adopting the PGMIndex, ZMI also benefits from the bit manipulation instruction set provided by modern CPUs, which can significantly accelerate the computation of Z-order curve values [65].

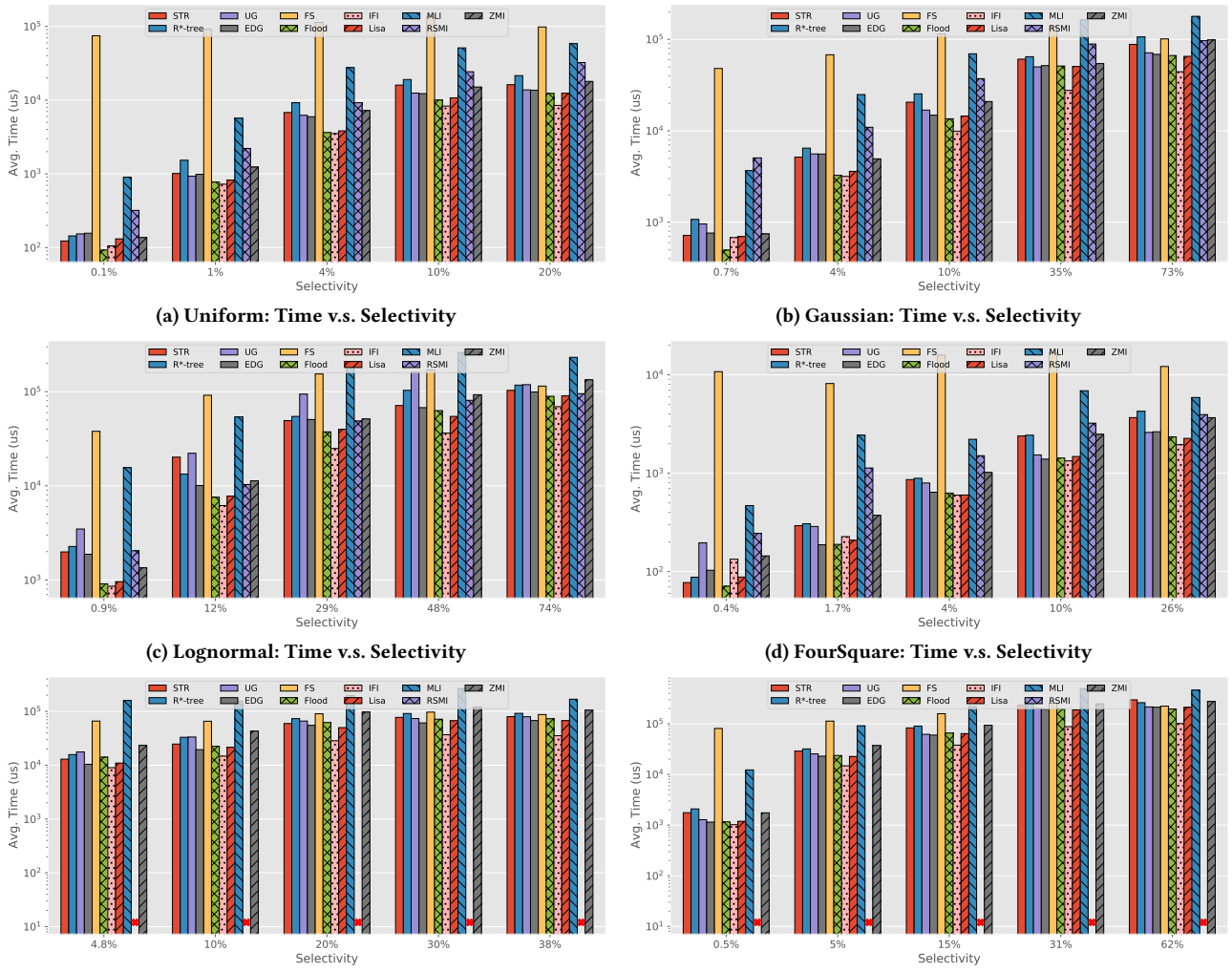
Indices that adopt the grid layouts, i.e., LISA and Flood, take slightly longer construction time due to the generation of internal equal-depth grid cells. The construction time of IFI is 1.5× of STR but much faster than R*tree (about 30× faster). This is because, in our implementation of IFI, we adopt a similar bulking loading strategy to find the leaf nodes. Besides, the leaf node capacity of IFI is generally larger (e.g., 1000) than ordinary R-tree indices (e.g., 128), which also alleviates the extra overhead caused by fitting linear models. Compared with all the other learned indices, RSMI takes a much longer time to build ($10^4\times$ slower than ZMI) and even fails to terminate on the largest dataset OSM. This is because the deep learning models adopted by RSMI are much more complex than the other indices and the CPU-only training is generally inefficient compared with using GPUs. Though accelerating query processing using novel hardware like GPU and TPU is getting popular recently [50], in this paper, we focus on the general application scenarios of multi-dimensional indices where a powerful GPU is usually not available, thus the CPU-based RSMI is chosen for comparison.

Takeaways. The construction time highly depends on the internal model choices of multi-dimensional learned indices. Indices based on PGMIndex are generally as fast as efficient non-learned baselines.

5.2 Index Memory Cost (Q2)

In this section, we report the memory cost of each compared methods. Note that, for indices RSMI, kdtree, qdtree, and ANN, we cannot programmatically retrieve their memory cost at runtime. Thus, for these indices, we perform heap profiling via gperftools [20] starting before building the index and ending after the index is constructed. Such a method for measuring memory cost is also adopted in a benchmark on modern spatial libraries [44].

Table 5 shows the memory cost evaluation results. As we fix the error threshold of PGMIndex to 64, the resulting learned multi-dimensional indices except RSMI share a similar level of memory cost. And all the learned indices achieve significant improvement



(a) Uniform: Time v.s. Selectivity

(b) Gaussian: Time v.s. Selectivity

(c) Lognormal: Time v.s. Selectivity

(d) FourSquare: Time v.s. Selectivity

(e) Toronto3d: Time v.s. Selectivity

(f) OSM: Time v.s. Selectivity

Figure 9: Range query evaluation results on real datasets and synthetic datasets of the default configuration (#Points=20M and #Dim=2). Note that, RSMI is not available on Toronto3d and fails to terminate training on OSM.

of memory overhead compared with popular non-learned indices (e.g., 2× smaller than STR, 3× smaller than R*tree, 5× smaller than kdtree). Note that, grid indices like UG and EDG also have low space cost as they only need to store the partition boundaries on each dimension. However, the grid indices generally perform badly on query processing, especially for skewed and high-dimensional datasets, which will be discussed later.

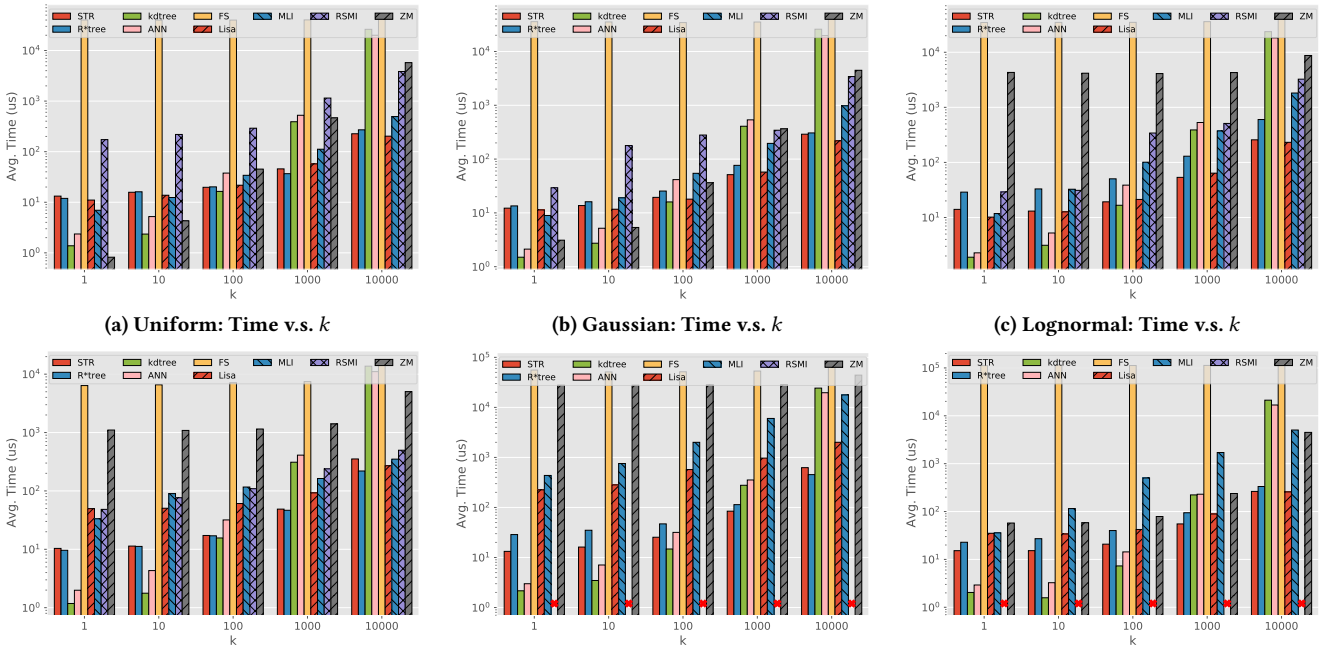
Different from the indices that internally employ PGM-Index, IFI can also achieve remarkable memory cost as its leaf node capacity is much larger (1000) than an ordinary R-tree (64), thus requiring much fewer tree nodes to store. As for RSMI, though its index construction time is significantly higher than other indices (9124× larger than ZMI on dataset lognormal), the trained model is much more compact for storage and not sensitive to the size of various datasets (about 20MB for all datasets). Thus, an ideal use-case of RSMI is to perform offline training using powerful machines and then deploy the trained models for query processing.

Takeaways. All evaluated learned indices can obtain a more compact structure than the commonly used non-learned indices with a sacrifice of affordable index construction cost (except the CPU-version of RSMI).

5.3 Range Query Processing (Q3)

We then report the range query evaluation results using the queries generated in Section 4.1. Figure 9 shows the average query processing time w.r.t. different levels of query selectivities (Eq. (4)).

Among non-learned baselines, STR and EDG are the fastest indices on range query processing. Compared with these two methods, learned indices IFI, Flood, and LISA can always achieve better performance regardless of data skewness and query selectivity. Not surprisingly, learned indices perform much better for queries of lower selectivity. For example, on dataset lognormal, Flood is 2.19× faster than STR when query selectivity is less than 1%; on the other hand, the speedup ratio decreases to 1.31 when the



(a) Uniform: Time v.s. k **(b) Gaussian: Time v.s. k** **(c) Lognormal: Time v.s. k**

(d) FourSquare: Time v.s. k **(e) Toronto3d: Time v.s. k** **(f) OSM: Time v.s. k**

Figure 10: kNN query evaluation results on default settings. Note that `IFI` and `Flood` do not support kNN queries.

selectivity ratio increases to 48%. This is because, the pruning power of learned models becomes more significant for small query ranges, and in the worst case, all the indices (learned and non-learned) are no better than a linear scan. `RSMI` does not show satisfactory performance, especially on non-uniform data. This is because `RSMI` aims to design a disk-based index structure where page access is the major optimized objective. Besides, `RSMI` also supports approximate range query processing, which is about an order of magnitude faster than the exact query processing when query selectivity is lower than 5%. The performance of `MLI` is also not satisfactory as its metric-based projection function (Eq. (1)) cannot well encode locality information and thus is not suitable for orthogonal range query processing.

An interesting point is that, as the first learned multi-dimensional index, `ZMI` is usually regarded as a weak baseline in previous studies [33, 38, 47]. However, from our evaluation, the performance of `ZMI` is not that bad and is comparable with `STR` (recalling that `ZMI` has very low construction time and memory cost). The major reason is that the original implementation of `ZMI` [57] adopts a coarse pruning strategy where the whole range of [Z(min_corner), Z(max_corner)] is searched. In this work, we use the BigMin algorithm [49] to divide a query box into fine-grained candidate Z-value ranges and perform a learned index-based search over the array sorted by Z-values, leading to a significant performance improvement of such a simple baseline.

Takeaways. Learned indices `IFI`, `Flood`, and `LISA` can robustly outperform the non-learned baselines across all levels of query selectivities. The speedup ratio is especially significant for uniformly distributed datasets and low-selectivity queries.

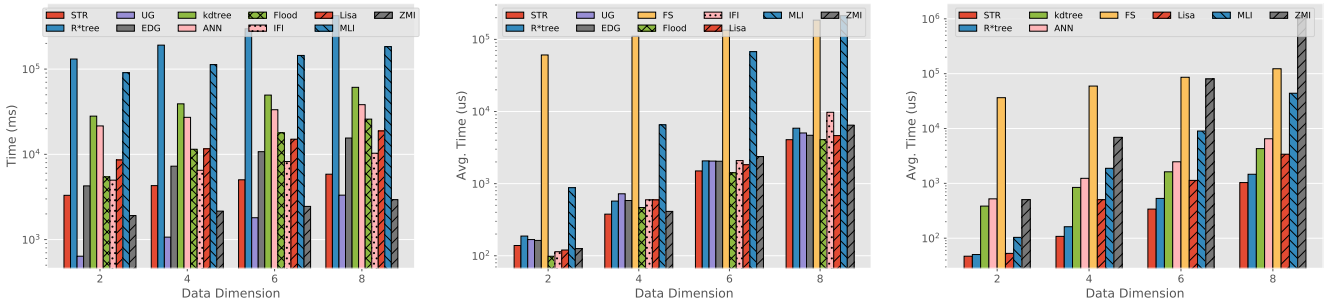
5.4 kNN Query Processing (Q4)

Figure 10 reports the average kNN query processing time w.r.t. different values of $k \in \{1, 10, 100, 1000, 10000\}$.

For non-learned indices, two kd-tree variants `kdtree` and `ANN` perform the best for $k \leq 100$; and R-tree variants `STR` and `R*tree` are efficient for $k > 100$. Different from range queries where learned indices can generally outperform non-learned baselines, only `MLI`, `LISA`, and `ZM` can outperform the highly optimized R-tree and kd-tree libraries on *some* datasets and *some* k. For example, on synthetic datasets, `MLI` is 1.1 $\sim$ 1.3× faster than `STR` when $k \leq 10$; however, `MLI` can be up to 3.6× slower compared with `STR` when k increases to 10000. And unfortunately, no learned index can beat the best non-learned baselines on all three real-world datasets.

The reason is that all of the existing learned indices process kNN queries by repeatedly invoking *range queries* of progressively increased search radius until k results are found. Thus, the kNN processing time highly depends on the setting of the initial search range. In the worst case, such a progressive search method requires invoking range queries from a very small range to the whole data space, where a large portion of range queries are unnecessary to retrieve kNN results. Besides, different from R-tree or kd-tree that is based on recursive space partitioning, it is generally intractable to inject local aggregation information like range count into the learned index structures, also making it hard to perform efficient search range expansion.

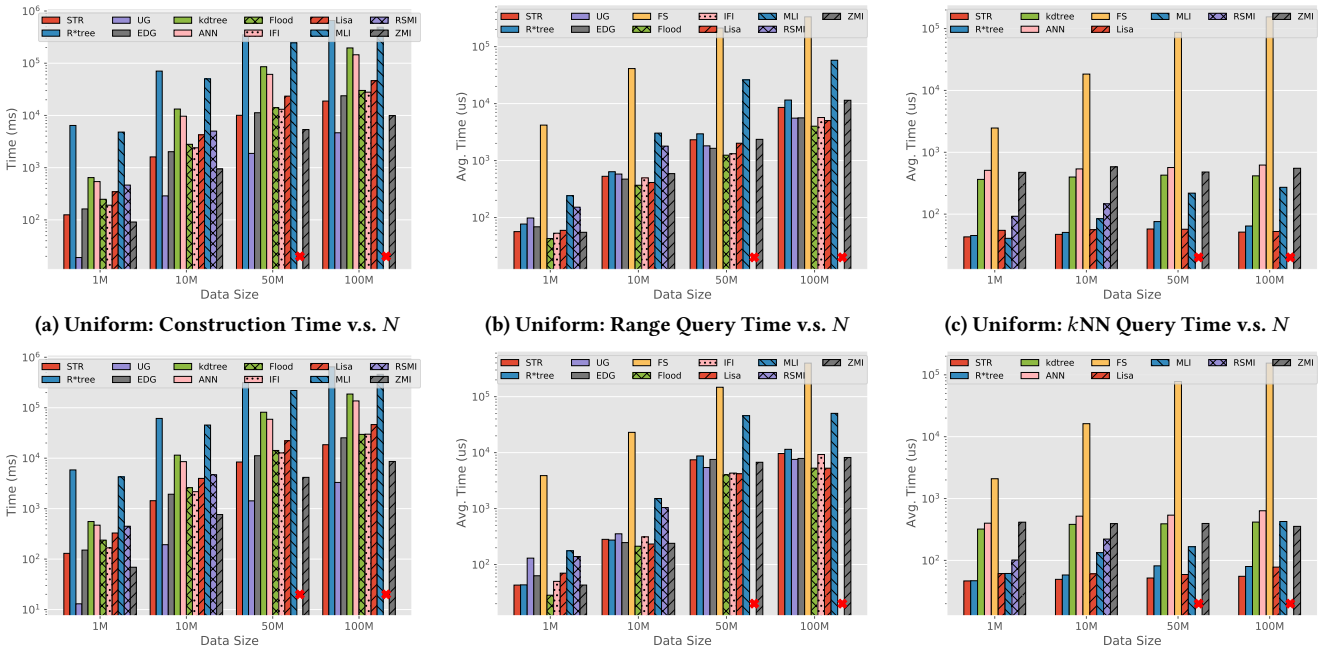
Takeaways. Different from range queries, the kNN performance for learned indices is not significantly better than non-learned baselines. The nature of progressive range search prevents the efficient kNN query processing for learned indices.



(a) Uniform: Construction Time v.s. d (b) Uniform: Range Query Time v.s. d (c) Uniform: kNN Query Time v.s. d

(d) Gaussian: Construction Time v.s. d (e) Gaussian: Range Query Time v.s. d (f) Gaussian: kNN Query Time v.s. d

Figure 11: Scalability evaluation w.r.t. dimension d. The range query selectivity is in $[10^{-5}, 10^{-2}]$, and $k = 1000$ for kNN queries.



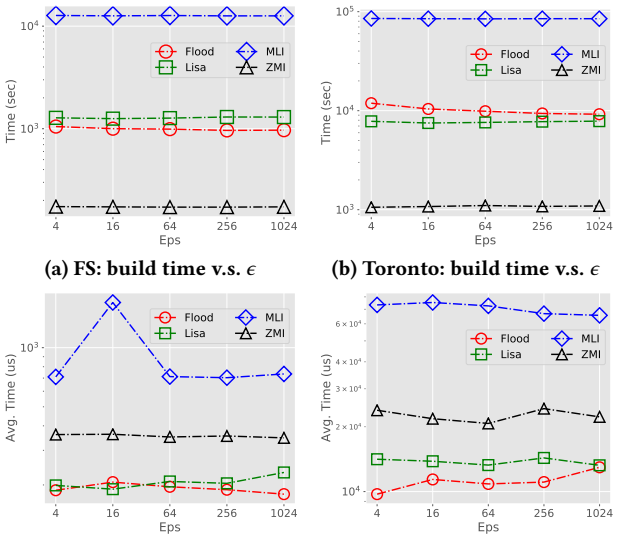
(a) Uniform: Construction Time v.s. N (b) Uniform: Range Query Time v.s. N (c) Uniform: kNN Query Time v.s. N

(d) Normal: Construction Time v.s. N (e) Normal: Range Query Time v.s. N (f) Normal: kNN Query Time v.s. N

Figure 12: Scalability evaluation w.r.t. data size N. The range query selectivity is in $[10^{-4}, 10^{-2}]$, and $k = 1000$ for kNN queries.

5.5 Scalability Evaluation (Q5)

This section studies the index performance when scaling the size and dimension of datasets. Figure 11 shows the scalability evaluation results by varying data dimension $d \in \{2, 4, 6, 8\}$ for two synthetic datasets. The construction time for all compared indices grows as d increases. For all grid-based indices (e.g., `Flood` and `LISA`), we set the partition number of each dimension to $(N/B)^{1/d}$, which avoids the exponentially growing (w.r.t. d) of the grid size and makes each grid cell contains roughly B points. Note that, `ZMI` is insensitive to the increase of d as increasing d slightly slows down the



(a) FS: build time v.s. ϵ (b) Toronto: build time v.s. ϵ

(c) FS: range query v.s. ϵ (d) Toronto3d: range query v.s. ϵ

Figure 13: Evaluation results w.r.t. error parameter ϵ on two real datasets FS and Toronto3d.

computation of Z-curve values which occupies a small portion of the whole index construction. For query processing, the results are similar to that discussed in Section 5.3 and Section 5.4. Specifically, `Flood` and `LISA` can robustly achieve better range query performance. And as the most efficient index on low dimensional datasets (Figure 9), the performance `MLI` is worse than `Flood` and `LISA` due to its simple leaf-node model structure cannot well capture the high-dimensional data distribution features.

Figure 12 also shows the scalability evaluation results by varying the size of synthetic datasets from 1M to 100M. The index construction time for learned indices is basically proportional to the increase of data size N considering that all the learned indices require sorting the data based on some attributes and then training models on the sorted data. `RSMI` fails to terminate training for a data size of 100M (i.e., exceeding 5 hours). For range queries of a fixed selectivity, the processing time of all indices increases as there are more points that need to be reported; however, the relative ratio between learned and non-learned indices is similar to that of the default setting (Figure 9).

Takeaways. Learned indices `Flood` and `LISA` can generally scale to larger datasets (up to 100M) and medium level of dimensions (up to 8). The `IFI` performs much better on low dimensional data.

5.6 Effects of Error Parameter

In this section, we study the effect of the error threshold parameter ϵ for the indices internally using PGM-Index, i.e., `Flood`, `LISA`, `MLI`, and `ZMI`. Figure 13 shows the index construction time and range query processing time when varying ϵ in the range [4, 1024] on two real datasets. The index construction time is generally insensitive to ϵ as the optimal piece-wise linear fitting algorithm used by PGM-Index requires only a simple one pass of data. For range query processing, different from 1-D learned indices where a small ϵ definitely leads to less search time, the relationship between query processing time and ϵ is not monotonic. The reason is that a smaller ϵ can only reduce the overhead of a single point search; however,

Index	ZMI	MLI	IFI	RSMI	LISA	Flood
Space Saving	😎	😎	😎	😎	😎	😎
Range Query	😕	😕	😎	😕	😎	😎
*k*NN Query	😕	😕	N.A.	😕	😕	N.A.
Scalability	😎	☹	😎	☹	😎	😎

Figure 14: Summary of evaluation results. A "cool" face means the learned index can robustly outperform the best non-learned baseline. A "doubtful" face means the learned index can perform better in some cases. A "sad" face means the performance of learned index is unsatisfactory.

the fatal factor to range query processing efficiency is whether the learned models and the designed data layout can well filter unnecessary results. As for space cost, our results are similar to the 1-D case reported in [17] and are omitted due to the space cost. It is clear that decreasing ϵ will enlarge the space overhead as more line segments are used to achieve a smaller ϵ. According to the theoretical analysis of PGM-Index [16], the space cost for a PGM-Index of error bound ϵ is $O(N/\epsilon^2)$. To this end, we suggest $\epsilon = 64$ for all the learned indices that employ the PGM-Index in their structure as such an ϵ can well trade-off the index building time, memory overhead, and query processing efficiency.

6 CONCLUSION AND FUTURE WORKS

Learned data indexing is getting prominent due to its data-driven manner. In this work, we perform intensive experimental studies over six multi-dimensional learned indices. We highlight our evaluation results in Figure 14, and please note that for *k*NN queries, "N.A." indicate that their original papers did not provide any *k*NN algorithms. From Figure 14, the learned indices, especially IF-Index [23], Flood [38], and Lisa [33], can process range queries up to 3× faster than a bulk-loading R-tree with only half of its space. However, the performance of *k*NN query processing is not that significant compared with non-learned baselines. Generally, the efficiency of *k*NN query for learned indices is between R-tree and *k*d-tree, leaving huge optimization space. We further identify the flowing research possibilities for future works.

Efficient Updatable Index. As we discussed in Section 3.6, most of the existing indices focus on the read-only workloads. Though indices like RSMI [47] and Lisa [33] support insertion and deletion via a model-based data layout, it is inevitable to re-train the whole models when the data distribution significantly shifts from the initial one that is used to build the index. A fully updatable learned multi-dimensional index requires efforts on novel data layout design and efficient model update strategy.

Advanced Query Support. In this work, we focus on the in-memory performance of range query and *k*NN query processing as currently learned indices majorly support these two queries. It would be interesting to implement and evaluate the performance of advanced analytical queries based on learned index, e.g., skyline query [8], distance join [24], *k*NN join [59], etc.

Distributed Spatial Analytics. To process and analyze web-scale multi-dimensional data, existing solutions usually adopt distributed engines like Spark [52] to serve the backend, e.g., LocationSpark [53], GeoSpark [63], Simba [60]. As reported in an evaluation study [43], the index memory overhead of existing systems is generally high. It would be promising to design new distributed multi-dimensional analytical systems powered by learned index.

REFERENCES

[1] annproject [n.d.]. ANN Project. http://www.cs.umd.edu/~mount/ANN/. Accessed: 2022-04-15.

[2] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. 2008. The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Trans. Algorithms* 4, 1 (2008), 9:1–9:30.

[3] David Arthur and Sergei Vassilvitskii. 2007. k-means++: the advantages of careful seeding. In *SODA*. SIAM, 1027–1035.

[4] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD Conference*. ACM Press, 322–331.

[5] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

[6] Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. 2021. A "Learned" Approach to Quicken and Compress Rank/Select Dictionaries. In *ALENEX*. SIAM, 46–59.

[7] boostgeometry [n.d.]. Boost Geometry. http://boost.org/libs/geometry. Accessed: 2022-04-15.

[8] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *ICDE*. IEEE Computer Society, 421–430.

[9] Tolga Bozkaya and Meral Ozsoyoglu. 1999. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems (TODS)* 24, 3 (1999), 361–404.

[10] Hsinchun Chen, Roger HL Chiang, and Veda C Storey. 2012. Business intelligence and analytics: From big data to big impact. *MIS quarterly* (2012), 1165–1188.

[11] Lu Chen, Yunjun Gao, Baihua Zheng, Christian S Jensen, Hanyu Yang, and Keyu Yang. 2017. Pivot-based metric indexing. (2017).

[12] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries. In *EDBT*. OpenProceedings.org, 407–410.

[13] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD Conference*. ACM, 969–984.

[14] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (2020), 74–86.

[15] James H Faghmous and Vipin Kumar. 2014. Spatio-temporal data mining for climate data: Advances, challenges, and opportunities. In *Data mining and knowledge discovery for big data*. Springer, 83–116.

[16] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why Are Learned Indexes So Effective?. In *ICML (Proceedings of Machine Learning Research)*, Vol. 119. PMLR, 3123–3132.

[17] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175.

[18] foursquare [n.d.]. FourSquare Data. https://sites.google.com/site/yangdingqi/home/foursquare-dataset. Accessed: 2022-04-15.

[19] geos [n.d.]. GEOS. https://github.com/libgeos/geos. Accessed: 2022-04-15.

[20] gperf [n.d.]. gperftools. https://github.com/gperftools/gperftools. Accessed: 2022-04-15.

[21] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2021. The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. *CoRR* abs/2103.04541 (2021).

[22] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD Conference*. ACM Press, 47–57.

[23] Ali Hadian, Ankit Kumar, and Thomas Heinis. 2020. Hands-off Model Integration in Spatial Index Structures. In *AIDB@VLDB*.

[24] Gísli R. Hjaltason and Hanan Samet. 1998. Incremental Distance Join Algorithms for Spatial Databases. In *SIGMOD Conference*. ACM Press, 237–248.

[25] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. 2005. iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.* 30, 2 (2005), 364–397.

[26] Ibrahim Kamel and Christos Faloutsos. 1994. Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB*. Morgan Kaufmann, 500–509.

[27] Kothuri Venkata Ravi Kanth, Siva Ravada, and Daniel Abugov. 2002. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. In *SIGMOD Conference*. ACM, 546–557.

[28] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *aiDM@SIGMOD*. ACM, 5:1–5:5.

[29] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *CIDR*. www.cidrdb.org.

[30] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD Conference*. ACM, 489–504.

[31] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. 2020. The Case for a Learned Sorting Algorithm. In *SIGMOD Conference*. ACM, 1001–1016.

[32] Scott T. Leutenegger, J. M. Edgington, and Mario Alberto López. 1997. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *ICDE*. IEEE Computer Society, 497–506.

[33] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *SIGMOD Conference*. ACM, 2119–2133.

[34] Qiyu Liu, Libin Zheng, Yanyan Shen, and Lei Chen. 2020. Stable Learned Bloom Filters for Data Streams. *Proc. VLDB Endow.* 13, 11 (2020), 2355–2367.

[35] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.

[36] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *NeurIPS*. 462–471.

[37] nanoflann [n.d.]. nanoflann. https://github.com/jlblancoc/nanoflann. Accessed: 2022-04-15.

[38] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD Conference*. ACM, 985–1000.

[39] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.* 9, 1 (1984), 38–71.

[40] numpy [n.d.]. Numpy. https://numpy.org/.

[41] Joseph O'Rourke. 1981. An on-line algorithm for fitting straight lines between data ranges. *Commun. ACM* 24, 9 (1981), 574–578.

[42] osm-china [n.d.]. OpenStreet Map. https://planet.openstreetmap.org. Accessed: 2022-04-15.

[43] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How Good Are Modern Spatial Analytics Systems? *Proc. VLDB Endow.* 11, 11 (2018), 1661–1673.

[44] Varun Pandey, Alexander van Renen, Andreas Kipf, and Alfons Kemper. 2021. How Good Are Modern Spatial Libraries? *Data Sci. Eng.* 6, 2 (2021), 192–208.

[45] PostgreSQL. 2021. PostgreSQL: The World's Most Advanced Open Source Relational Database. *Web resource: https://www.postgresql.org/* (2021).

[46] pytorch [n.d.]. PyTorch. https://pytorch.org/.

[47] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *Proc. VLDB Endow.* 13, 11 (2020), 2341–2354.

[48] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2020. Packing R-trees with Space-filling Curves: Theoretical Optimality, Empirical Efficiency, and Bulk-loading Parallelizability. *ACM Trans. Database Syst.* 45, 3 (2020), 14:1–14:47.

[49] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. 2000. Integrating the UB-Tree into a Database System Kernel. In *VLDB*. Morgan Kaufmann, 263–272.

[50] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query processing on heterogeneous CPU/GPU systems. *ACM Computing Surveys (CSUR)* 55, 1 (2022), 1–38.

[51] Terrence J Sejnowski, Patricia S Churchland, and J Anthony Movshon. 2014. Putting big data to good use in neuroscience. *Nature neuroscience* 17, 11 (2014), 1440–1441.

[52] spark [n.d.]. Apache Spark. https://spark.apache.org/. Accessed: 2022-04-15.

[53] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. 2016. LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data. *Proc. VLDB Endow.* 9, 13 (2016), 1565–1568.

[54] Yufei Tao, Dimitris Papadias, and Jimeng Sun. 2003. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *VLDB*. Morgan Kaufmann, 790–801.

[55] tensorflow [n.d.]. Tensorflow. https://www.tensorflow.org/.

[56] toronto3d [n.d.]. Toronto3D Data. https://github.com/WeikaiTan/Toronto-3D. Accessed: 2022-04-15.

[57] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *MDM*. IEEE, 569–574.

[58] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proc. VLDB Endow.* 14, 8 (2021), 1276–1288.

[59] Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jin Hu. 2004. Gorder: An Efficient Method for KNN Join Processing. In *VLDB*. Morgan Kaufmann, 756–767.

[60] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In *SIGMOD Conference*. ACM, 1071–1085.

[61] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning Data Layouts for Big Data Analytics. In *SIGMOD Conference*. ACM, 193–208.

[62] Peter N Yianilos. 1993. Data Structures and Algorithms for Nearest Neighbor. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, Vol. 66. SIAM, 311.

[63] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. GeoSpark: a cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL/GIS*. ACM,

70:1–70:4.

[64] Li Zhu, Fei Richard Yu, Yige Wang, Bin Ning, and Tao Tang. 2019. Big Data Analytics in Intelligent Transportation Systems: A Survey. *IEEE Trans. Intell. Transp. Syst.* 20, 1 (2019), 383–398.

[65] zvalue-comp [n.d.]. morton-nd. https://github.com/morton-nd/morton-nd. Accessed: 2022-04-15.