Published: 25 October 2025
Online AM: 17 September 2025
Accepted: 27 August 2025
Revised: 08 November 2024
Received: 11 March 2024

SURVEY

A Survey of Learned Indexes for the Multi-dimensional Space

ABDULLAH AL-MAMUN, Purdue University, West Lafayette, IN, United States

HAO WU, Purdue University, West Lafayette, IN, United States

QIYANG HE, Purdue University, West Lafayette, IN, United States

JIANGUO WANG, Purdue University, West Lafayette, IN, United States

WALID G. AREF, Purdue University, West Lafayette, IN, United States

A Survey of Learned Indexes for the Multi-dimensional Space

ABDULLAH AL-MAMUN, Computer Science, Purdue University, West Lafayette, United States
HAO WU, Computer Science, Purdue University, West Lafayette, United States and Keystone Strategy, San Francisco, United States
QIYANG HE, Computer Science, Purdue University, West Lafayette, United States and Snowflake, Bellevue, United States
JIANGUO WANG, Computer Science, Purdue University, West Lafayette, United States
WALID G. AREF, Computer Science, Purdue University, West Lafayette, United States

A recent research trend involves treating database index structures as Machine Learning (ML) models. In this domain, single or multiple ML models are trained to learn the mapping from keys to positions inside a dataset. This class of indexes is known as "Learned Indexes." Learned indexes have demonstrated improved search performance and reduced space requirements for one-dimensional data. The concept of one-dimensional learned indexes has naturally been extended to multi-dimensional (e.g., spatial) data, leading to the development of "Learned Multi-dimensional Indexes." This survey presents a taxonomy that classifies and categorizes both learned one- and multi-dimensional indexes, and surveys the existing literature on learned indexes according to this taxonomy with an emphasis on learned multi-dimensional index structures. Specifically, it reviews the current state of this research area, explains the core concepts behind each proposed method, and classifies these methods based on several well-defined criteria. Additionally, we present a timeline to illustrate the evolution of research on learned indexes. Finally, we highlight several open challenges and future research directions in this emerging and highly active field.

CCS Concepts: • **Database Systems** → **Indexing**; • **Machine Learning** → *ML for Systems;*

Additional Key Words and Phrases: Learned index structures, multi-dimensional indexes, spatial indexes, learned multi-dimensional indexes, learned spatial indexes

ACM Reference Format:
Abdullah Al-Mamun, Hao Wu, Qiyang He, Jianguo Wang, and Walid G. Aref. 2025. A Survey of Learned Indexes for the Multi-dimensional Space. *ACM Comput. Surv.* 58, 4, Article 96 (October 2025), 37 pages. https://doi.org/10.1145/3768575

Walid Aref and Jianguo Wang acknowledge the support of the NSF under Grant Numbers IIS-1910216 and IIS-2337806.
Authors' Contact Information: Abdullah Al-Mamun, Computer Science, Purdue University, West Lafayette, Indiana, United States; e-mail: mamuna@purdue.edu; Hao Wu, Computer Science, Purdue University, West Lafayette, Indiana, United States and Keystone Strategy, San Francisco, California, United States; e-mail: haowu3@alumni.cmu.edu; Qiyang He, Computer Science, Purdue University, West Lafayette, Indiana, United States and Snowflake, Bellevue, Washington, United States; e-mail: qiyang.he@snowflake.com; Jianguo Wang, Computer Science, Purdue University, West Lafayette, Indiana, United States; e-mail: csjgwang@purdue.edu; Walid G. Aref, Computer Science, Purdue University, West Lafayette, Indiana, United States; e-mail: aref@purdue.edu.

ACM 0360-0300/2025/10-ART96
https://doi.org/10.1145/3768575

1 Introduction

Recently, due to tremendous progress in the field of machine learning, two research trends have emerged in the area of **Database Systems** (**DB**): ML for DB, and DB for ML. In ML for DB, the goal is to develop ML-enhanced core database systems components, e.g., learned indexes [102], learned query optimizers [139], and self-driving databases [156]. These initial learned systems components have demonstrated superior performance compared to their traditional counterparts. On the other hand, in DB for ML, the objective is to extend the traditional DB architecture, components, and query languages to support efficient in-database ML workloads [32, 90, 118].

This survey article focuses on ML for DB, particularly on the idea of replacing traditional database index structures (e.g., B-tree [18, 39]) with ML models, first proposed in [102]. The proposed **Recursive Model Index** (**RMI**) can be considered as the first instance of a "Learned Index". Note that traditional indexes provide theoretical guarantees on performance, are well-studied, and have been successfully integrated into practical data systems. In contrast, *pure* learned indexes learn the key-to-position mapping with some error-correction mechanisms to achieve better search performance, and reduce space requirements. On the other hand, there are *hybrid* learned indexes that optimize traditional indexes with helper ML models. The spectrum of learned indexes is given in Figure 1.

Indexing the Learned Models vs. Learning the Index. Although the term "Learned Indexes" has been coined very recently [102], the concept of using a learning mechanism in the context of database indexing has been studied previously. An example of an earlier index structure that combines ML techniques in the context of database indexing is the handwritten trie [14]. The handwritten trie employs **Hidden Markov Models** (**HMMs**) [164] on a trie[63] structure to index the learned models. Notice that this handwritten trie focuses on the idea of *indexing the learned models* instead of *learning the index*. On the other hand, one of the earliest articles on a distribution-aware index structure for spatial (i.e., multi-dimensional) data can be found in [17]. In [17], the proposed technique combines an R-tree [74] with a self-organizing map [98]. Moreover, the initial promising results of one-dimensional learned indexes [102] have inspired researchers to extend the concept in the context of multi-dimensional data. As a result, various methods for learned multi-dimensional indexes have been introduced in recent years. One of the key assumptions in one-dimensional learned indexes is that the data can be sorted (i.e., totally ordered). However, there is no obvious total sort order for multi-dimensional data. As a result, it is challenging to define an error-correction mechanism in case of mis-predictions. Moreover, the layout of the multi-dimensional data might need to be re-arranged based on a pre-defined mechanism so that it can be easily learned by ML models. The choice of ML model might also vary from learned one-dimensional indexes due to the impact of dimensionality. In summary, learned multi-dimensional indexes need to address additional research challenges that are not associated with learned one-dimensional indexes. In this survey, we distinguish between multi- and high-dimensional data. In the context of multi-dimensional data, we assume that the dimension of the data is typically between 2 to 10 [148]. On the other hand, the dimension of the data can be very high (e.g., 100) [53] in the context of high-dimensional data. Although there are studies related to ML-enhanced high-dimensional indexes, we have not included them in this study unless some of the techniques in this domain have influenced the research of learned multi-dimensional indexes.

In this article, we present a comprehensive survey of recent advances in the area of learned multi-dimensional indexes using that taxonomy given in Figure 2. In the taxonomy, by *learning the index*, we refer to the problem of replacing a traditional database index structure with an ML model. For example, instead of searching a B^+-tree to locate the leaf page that contains an input search key, one uses an ML model that predicts the location that contains the search key. In the taxonomy, in the context of learning the index, we further distinguish among learned indexes along

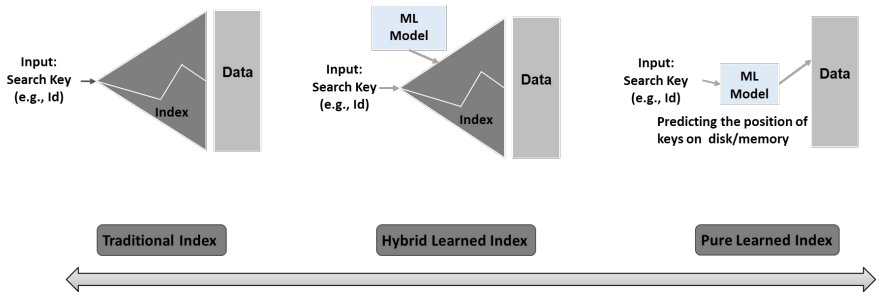


Fig. 1. Spectrum of learned indexes.

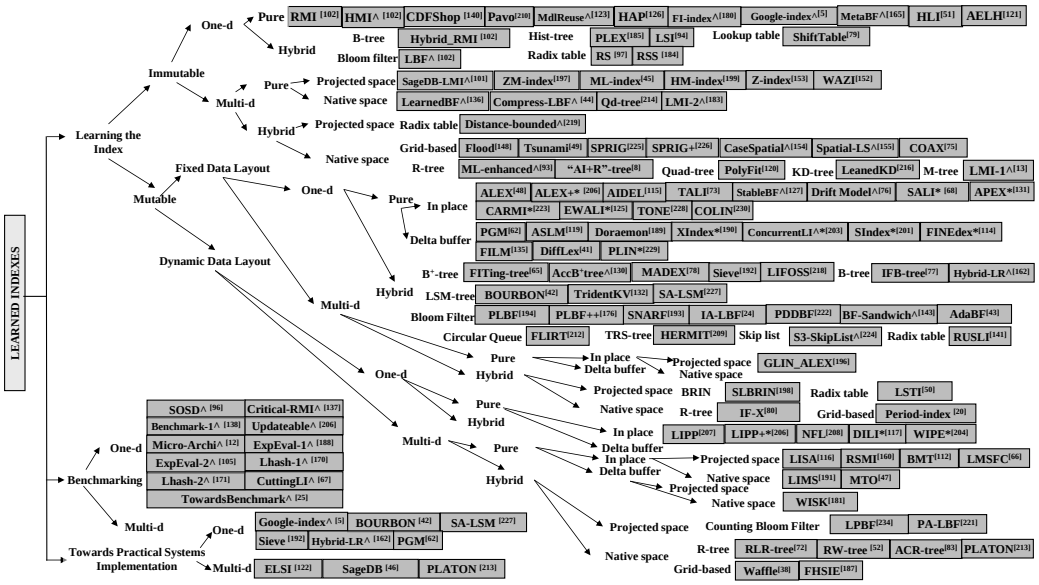
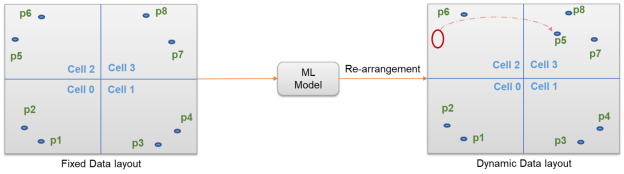


Fig. 2. Taxonomy of learned indexes. The end of a branch indicates that there are no articles in that category as of the time this article has been written. An asterisk (*) symbol is used if the index natively supports concurrency. A wedge (∧) symbol indicates that we have assigned a name to refer a particular article or an index whenever a fixed name is not given by the original authors of that article. The hybrid learned indexes are categorized based on their underlying traditional data structures.

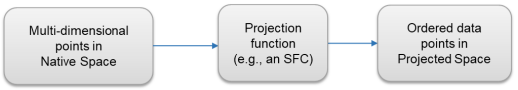
the following dimension: **learned indexes that support static datasets (i.e., Immutable) vs. learned indexes that support inserts/updates (i.e., Mutable)**. The issue of supporting static vs. dynamic datasets is crucial because learning an index requires offline training that is relatively slow in nature. Thus, learned indexes that support inserts/updates need to accommodate this fact, yet still offer online responses. In our taxonomy, mutable learned indexes are further divided into **fixed vs. dynamic data layout**. A fixed data layout refers to the class of indexes where the layout of the data and the structure of the index are fixed before the index-building phase. On the other hand, if the layout of the data is arranged/re-arranged by the ML models while building the learned index, we refer to them as having a dynamic data layout. For example, in Figure 3(a), the initial fixed data layout is re-arranged using an ML model.

Next, we distinguish between learned indexes along the dimensionality of the data: **learned indexes for one-dimensional data vs. learned indexes for multi-dimensional data**.



(a) Fixed vs. Dynamic data layout

(b) In-place vs. Delta buffer-based insertion



(c) Native vs. Projected space

Fig. 3. Illustration of the taxonomy criteria.

Under each category, we further classify the indexes into **pure vs. hybrid learned indexes**. Pure learned indexes are designed without leveraging a traditional index structure (e.g., a B-tree or an R-tree). Moreover, pure learned indexes are designed to replace a traditional index structure. In contrast, a hybrid learned index combines a traditional index structure with ML models to build an ML-enhanced index structure. As a result, hybrid learned indexes are categorized further based on their **underlying traditional data structure**. For example, in the lower right section of Figure 2, RW-tree [52], RLR-tree [72], ACR-tree [83], and PLATON [213] are grouped together because the underlying traditional data structure for all the proposed hybrid learned indexes is the R-tree. Similarly, other hybrid learned indexes are grouped based on their underlying traditional structures (e.g., Grid-based).

On the other hand, mutable pure learned indexes are classified based on their policy for supporting new data insertions: **in-place vs. delta buffer insertion policies**. The in-place insertion policy uses ML models to find the position where the new data item should be inserted. Additionally, an in-place insertion policy might reserve some gaps in the index so that insert operations can be accommodated immediately. Conversely, the delta buffer insertion policy employs fixed-size buffers to hold the new data items for a short period of time. These temporary buffers are synchronized with the index structure periodically (e.g., during ML model re-training). However, we do not classify mutable hybrid indexes based on insertion strategies due to the presence of a traditional index, where that might influence the choice of the insertion strategy. The difference between the two insertion policies are illustrated with the example given in Figure 3(b).

We further classify the learned multi-dimensional indexes into two classes: **indexes operating in the native space of the data vs. a projected space of the data**. Here, if the ML models are trained on the original representation of the multi-dimensional data, we refer to them as indexes built in native space. Conversely, if the multi-dimensional data is projected or linearized into a projected space using a projection function or a **Space Filling Curve** (**SFC**) [145, 172], we refer to them as indexes operating in a projected space (Figure 3(c)).

Additionally, in the taxonomy, we highlight studies related to benchmarking learned one-dimensional indexes. As of the time of writing this there are no comprehensive benchmarking

studies for learned multi-dimensional indexes. We highlight and discuss studies related to the integration of learned indexes into practical systems.

Contributions. The main contributions of this survey are as follows. We provide an up-to-date coverage of learned multi-dimensional indexes (until the end of 2023). We present a taxonomy of existing learned indexes in both the one- and multi-dimensional spaces. We expect that upcoming learned index structures can be categorized within the taxonomy based on the proposed criteria.

Article Organization. The rest of this article is organized as follows. Section 2 presents existing work in the area of indexing the learned models. Section 3 provides an overview of related surveys and tutorials. Section 4 presents a timeline of the evolution of learned indexes.

Sections 5–7 present both one-dimensional and multi-dimensional indexes according to the taxonomy in Figure 2. The properties of immutable and mutable learned multi-dimensional indexes are summarized in Tables 1–3. Table 4 provides a summary of ML techniques used in existing learned multi-dimensional indexes. Section 8 discusses studies related to benchmarking learned indexes. Section 9 presents steps taken towards the integration of learned indexes in practical systems. Section 10 provides an overview of open challenges and discusses future research directions. Finally, Section 11 concludes the article.

2 Indexing the Learned Models

Assume that we have a collection of learned models, where each model represents a certain class of objects, e.g., the class cat, dog, and so on. Given an input instance, e.g., an image of an object, we need to execute each of the learned models to identify which model this input is likely to belong to, and hence identify the class of the input instance, e.g., being a cat with high probability. To avoid running each of the models on the instance, it would be good if we can *index* these learned models to distinguish between these models in a scalable way. We present this idea in the context of handwritten words and the Handwritten Trie [14] that highlight an example of the body of work for the class of Indexing the Learned Models. Other similar indexes in the same class can be found in [15].

Handwritten Trie [14]. Given a collection of learned models, where each model corresponds to one handwritten word, e.g., a person's signature, handwritten indexes are used to speed up the search for a matching model given an input handwritten text, e.g., to help identify whose person this signature belongs to. The core idea of the handwritten trie involves a trie structure as well as a collection of HMMs that serve as the alphabet symbols in the trie nodes. The input to the handwritten trie is handwritten text. This input is segmented into alphabet symbols, and the index is traversed by executing the HMM models associated with each of the trie nodes. Each HMM is trained to represent a pictogram class so that each HMM accepts a specific pictogram with high probability. By traversing nodes at each level, the model chooses nodes with the highest probability for a particular letter (pictogram) and eventually obtains a set of nodes with the highest combined probability. Since the handwritten trie does not actually learn the distribution of key inputs, but rather indexes the HMM models, it is categorized as an index for the Learned Models rather than a learned index by itself.

Other indexes exist in this category. In the context of music retrieval [91], an R*-tree [19] is used to index the HMM models. In the BayesTree [178], an R-tree is employed to index a hierarchy of mixture density models. More indexes, e.g., as in [15], fall under this category, but will not be discussed further in this survey.

3 Related Surveys and Tutorials

The topic of learned indexes is relatively new, and hence there are only a few related surveys in the literature. Most of the existing studies focus on learned one-dimensional indexes [61, 128, 232].

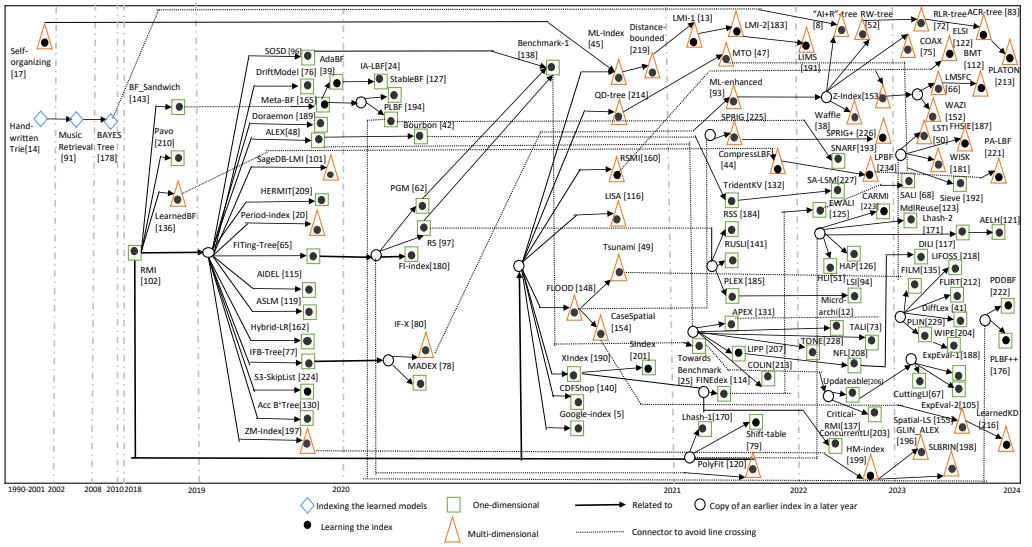


Fig. 4. Timeline of the evolution of learned indexes. Lines connecting between the various learned multi-dimensional indexes reflect dependence of these indexes on earlier work.

Related conference-based tutorials on the subject of learned one-dimensional indexes can be found in [85, 109–111, 200]. Moreover, progress reports on ML in databases, and instance optimized data systems have been presented in [100, 103]. A recent book on the topic of designing Data Structures with discussion on learned indexes for one-dimensional data can be found in [16].

There is a 4-page conference tutorial on the subject of learned multi-dimensional indexes [9] that is an earlier version of this survey article by the same authors. That tutorial covers methods only until Year 2020. Moreover, the tutorial does not include descriptions of the core concepts of the learned index structures listed in that short tutorial. This survey serves as an extension to that earlier conference tutorial and adds new criteria for classifying new indexes into a more sophisticated and comprehensive taxonomy as well as covers many more multi-dimensional learned indexes that have been developed until 2024.

Another related survey on multi-dimensional indexes can be found in [113]. However, in [113], the survey focuses on the impact of modern hardware platforms (e.g., non-volatile memory, GPU), and ML algorithms in the context of multi-dimensional indexes. There is also a series of related tutorials on big spatial data [167–169]. However, the focus of this series of tutorials is not on learned multi-dimensional index structures. There is also a related tutorial that covers ML-enhanced indexes particularly designed for high-dimensional data [53]. However, only a few learned multi-dimensional indexes are included in [53]. On the other hand, there are several comprehensive surveys for traditional multi-dimensional indexes [64, 173, 174]. As the topic of learned indexes is relatively new, to the best of our knowledge, this survey article is the first to focus particularly on learned multi-dimensional index structures.

4 Evolution of Learned Indexes

Figure 4, presents a chronological diagram to depict the evolution of both one- and multi-dimensional learned indexes. Here, we have grouped the articles on learned one- and multi-dimensional indexes based on their publication years. Moreover, we have used the → symbol to indicate if a later article is related to an earlier one. The ⇢ is used when there is a line crossing.

Additionally, a ∘ symbol is used to denote a copy of an earlier article in a later year. One- and multi-dimensional indexes are differentiated using the □ and △ symbols, respectively.

5 Learned Immutable Indexes

Immutable learned indexes are built on static datasets and do not support dynamic inserts/updates. In this section, we present the Learned Immutable Indexes in both the one- and multi-dimensional spaces.

5.1 The One-dimensional Case

While one-dimensional learned indexes are not the focus of this survey, we present a few foundational indexes in the category as many indexes in the case of multi-dimensional learned indexes built on their one-dimensional counterparts.

5.1.1 Pure Learned Indexes. Here, we present the core concepts of RMI [102] to highlight the class of Immutable One-dimensional Pure Learned Indexes.

***RMI** [102].* The key idea in [102] is: "Indexes are models." For example, given a key k, an index simply predicts the position of k in a sorted array. Also, it has also been observed that "a model predicting the position of a key within a sorted array effectively approximates the **Cumulative Distribution Function** (**CDF**)." As a result, indexes can be learned. In [102], it has been demonstrated how learned index structures can be implemented for three types of indexes: B-tree [18, 39], Hash map [35, 146], and Bloom filter [26]. Similar to a B-tree, for range queries, the RMI structure is introduced that recursively chooses a model at multiple levels. With an input key, the root model chooses a child model based on its output. This continues until it reaches the leaf model that predicts the actual position in the underlying sorted array.

The prediction by the leaf-level ML model might not be accurate. As a result, an error correction mechanism is employed to correct the misprediction within a predefined error bound (i.e., min and max error). Because data in the one-dimensional case is sorted, and hence a total order exists among the elements of the indexed array, any misprediction can be corrected by searching either to the left or to the right of the mispredicted value. A binary or exponential search operation can be performed to search for the correct value from the mispredicted location. The proposed RMI structure can be built using a mixture of ML models (e.g., neural networks [7] or linear regression [89]). Moreover, at the bottom level of RMI, a traditional B-tree can also be used as a model, introducing the idea of Hybrid Learned Indexes (i.e., Hybrid RMI). For point queries, the idea of **Hash-Model Index** (**HMI**) has been proposed in the same article. For minimizing collisions, HMI leverages the CDF of the dataset by employing learned hash functions. For existence indexes, the **Learned Bloom Filter** (**LBF**) has been introduced [102]. LBF is formulated as a classification problem. It proposes adopting a hybrid approach by integrating a traditional bloom filter with an ML model. During query processing, the key is first passed into a learned model to decide on existence, and a traditional bloom filter is used to catch any false negative cases from the learned model.

Other example indexes in this category are as follows. CDFShop [140] is a tool that guides users on tuning the parameters of the RMI index structure. Pavo [210] leverages **Recurrent Neural Networks** (**RNNs**) [36] to replace the hash function in a traditional inverted index structure. In MdlReuse [123], pre-trained ML models are used instead of training a model during the construction of the learned index. HAP [126] is a Hamming space indexing framework that leverages ML models for cost estimation, query processing, and index compression. The FI-index [180] investigates the use of function interpolation models as an alternative choice of ML model in the context of learned indexes. In Google-index [5], a learned index is introduced for a disk-based distributed system. In MetaBF [165], a learned bloom filter is proposed by leveraging meta-learning [175]. HLI [51] uses

ML models with error bounds for leaf nodes and models without error bounds for inner nodes to achieve the benefits of both approaches. AELH [121] is a learned hash index structure that leverages Autoencoders [195] for binary hash codes.

5.1.2 Hybrid Learned Indexes. Hybrid learned indexes combine traditional index structures with ML models to build ML-enhanced index structures. In this section, we present the core concepts of RS [97] as an example to highlight the class of Immutable One-dimensional Hybrid Learned Indexes. Then, in the section to follow, we survey in more detail, the multi-dimensional case.

RadixSpline (RS) *[97].* RS is a hybrid learned index structure that combines the linear spline model with a traditional radix table [107]. Given a search key, the traditional radix table is used to locate the two spline points bounding the search key. RS can be constructed in a single pass if the data is sorted. The steps to construct the RS index are as follows: (i) Fit a linear spline to the CDF of the data to ensure a certain error-bound, and collect a set of spline points, and (ii) Construct an approximate index of the spline points using a radix table. The proposed index supports both equality and range predicates. However, performance of the RS index can be negatively impacted by skewed data distributions. To address this, a tree-structured radix table can be utilized. Note that the RS structure can be tuned using only two hyperparameters for a given data and memory budget.

The other indexes in this category are as follows. Hybrid_RMI [102] combines a traditional B-tree with an RMI structure to create a hybrid index structure. PLEX [185] leverages a traditional Hist-tree [40] to build a learned index with a single tunable parameter. LSI [94] also leverages a traditional compact Hist-tree to construct a learned index for unsorted data. ShiftTable [79] is a helper error correction layer that can be combined with the ML models of a learned index. LBF [102] introduces the learned bloom filter structure by combining ML models with traditional bloom filters. RSS [184] extends the methods of RS [97] for indexing strings.

5.2 The Multi-dimensional Case

In this section, we explain the learned multi-dimensional indexes along the various dimensions of our taxonomy. The properties of the class of immutable learned multi-dimensional indexes are presented in Table 1. We highlight the types of queries supported by each of the learned multi-dimensional indexes. For point queries, some indexes do not explicitly provide a query processing algorithm or experimental results. However, if an index can be easily extended to support point queries, we highlight that by listing that the index supports point query processing. On the other hand, due to the nature of the application domain, an index might be designed emphasizing efficiency over accuracy. As a result, an index can support either exact or approximate query processing. Here, approximate query processing refers to the event of missing some results from the set of exact answers to the query. We indicate whether a supported query type returns an exact or an approximate answer. Notice that we have excluded the learned multi-dimensional bloom filters (e.g., LearnedBF [136], CompressLBF [44]) in the context of supported query types because they are considered as probabilistic existence index structures.

5.2.1 Pure Learned Indexes in the Projected Space.

SageDB includes a learned multi-dimensional index (SageDB-LMI) *[101].* SageDB is an ML-enhanced database system that has been envisioned in [101]. SageDB-LMI, that is one of the earliest attempts to extend techniques similar to RMI [102] in the context of multi-dimensional data. SageDB-LMI projects the multi-dimensional data points into one-dimensional space by consecutively sorting and partitioning the points along a sequence of dimensions (e.g., first x-dimension then y-dimension) into uniformly-sized cells. This produces a layout that is easily learnable compared to the more

Table 1. Properties of the Immutable Learned Multi-dimensional Indexes

Index	Pure Learned	Hybrid Learned	Data Space	Point Query	Range Query	kNN Query	Join Query
SageDB-LMI [101]	✓	×	Projected	Exact	Exact	×	×
ZM-index [197]	✓	×	Projected	Exact	Exact	×	×
ML-index [45]	✓	×	Projected	Exact	Exact	Exact	×
HM-index [199]	✓	×	Projected	Exact	Exact	×	×
Z-index [153]	✓	×	Projected	Exact	Exact	×	×
WAZI [152]	✓	×	Projected	Exact	Exact	×	×
LearnedBF [136]	✓	×	Native	—	—	—	—
CompressLBF [44]	✓	×	Native	—	—	—	—
Qd-tree [214]	✓	×	Native	Exact	Exact	×	×
LMI-2 [183]	✓	×	Native	Approx.	Approx.	Approx.	×
Distance-bounded [219]	×	Radix table	Projected	Approx.	Approx.	×	Approx.
Flood [148]	×	Grid	Native	Exact	Exact	×	×
Tsunami [49]	×	Grid	Native	Exact	Exact	×	×
SPRIG [225]	×	Grid	Native	Exact	Exact	Exact	×
SPRIG+ [226]	×	Grid	Native	Exact	Exact	Exact	×
ML-enhanced [93]	×	R-tree	Native	×	×	Approx.	×
"AI + R"-tree [8]	×	R-tree	Native	Exact	Exact	×	×
CaseSpatial [154]	×	Grid	Native	Exact	Exact	×	×
Spatial-LS [155]	×	Grid	Native	Exact	Exact	×	Exact
COAX [75]	×	Grid	Native	Exact	Exact	×	×
PolyFit [120]	×	Quad-tree	Native	Approx.	Approx.	×	×
LearnedKD [216]	×	KD-tree	Native	×	×	Approx.	×
LMI-1 [13]	×	M-tree	Native	Approx.	Approx.	Approx.	×

complex space-filling curves, e.g., the Z-order SFC [145]. Notably, any one-dimensional learned indexing method can be applied to the projected space. Consequently, in the second step, SageDB-LMI uses a trained CDF model (e.g., RMI) to predict the physical location of the point. In an in-memory, read-only experimental setup, SageDB-LMI outperforms a traditional R-tree in terms of average query time and index size.

***ZM-index** [197].* Similar to SageDB-LMI [101], the ZM-index projects multi-dimensional data into a one-dimensional projected space. Particularly, the ZM-index involves two essential components: 1. A Z-order [147, 151, 157] curve to linearize the multi-dimensional space, and 2. A multi-staged model index to support search. The Z-order curve sorts the multi-dimensional data according to the corresponding Z-values computed by bit interleaving. The multi-staged model index consists of neural network models at multiple stages that recursively partition the data space into sub-regions. As in RMI [102], given a query key, the model at Stage i predicts a position based on the CDF and the number of all keys. After that, it chooses a model at Stage $i + 1$ according to the predicted position or directly outputs the predicted position if already at the leaf level. The steps for range query processing using ZM-index are as follows: (i) Computing Z-addresses of the start and end points of the range query. (ii) Predicting the positions of the start and end points using the multi-staged model index structure. (iii) Finding the exact positions of the start and end points using model-based search. (iv) Scanning through the points within the range. Although the ZM-index can achieve query processing time similar to an R-tree, it can significantly reduce the index size compared to its traditional counterpart.

Multi-dimensional Learned index (ML-index) *[45].* In the case of learned multi-dimensional indexes in the projected space, it is desirable for the multi-dimensional data to be projected in an order that can be easily learned by ML models. To achieve this goal, the proposed ML-index generalizes the idea of the previously proposed iDistance [87]. The ML-index partitions and transforms the data into one-dimensional values based on distribution-aware reference points. It proposes an efficient scaling method so that, after projecting the multi-dimensional points into the one-dimensional space, the spatial proximity in the native space is well-preserved in the lower dimension. Similar to the learned B-tree index in [102], an RMI is applied to the projected space. The ML-index can support point, range, and kNN queries.

Hilbert Model index (HM-index) *[199].* The HM-index follows the similar principle as that of the ZM-index [197]. Here, the core idea is to project the multi-dimensional data into the one-dimensional space using the Hilbert SFC [82, 172] and applying one-dimensional learned indexing techniques on the projected space. Particularly, a two-stage model has been used for learning the underlying data distribution. During the prediction step, given a query point, the model either predicts the position of the query point or a range in which the position of the query points can be searched. Moreover, due to the application of the Hilbert linear ordering, an error correction mechanism can be used in case of an incorrect prediction with an error bound. Notice that the nearby data points in the projected Hilbert space may be far apart in the native space. As a result, it can impact the performance of range query processing. Thus, a query partitioning technique based on n-order Hilbert regions has been adopted to minimize the impact on performance.

Z-index *[153].* Many spatial indexes use a pre-defined SFC, e.g., the Z-curve or Hilbert-curve, to order multi-dimensional data. However, existing SFCs are not designed to be instance-optimized [100] for given data and query workloads. The idea of an instance-optimized SFC-based index has been investigated [153]. Particularly, for a given data and query workloads, an instance-optimized variant of the Z-index has been proposed by varying the partitioning and the ordering [166]. The goal is to reduce the number of false positives during range query processing. In the proposed algorithm, two heuristic-based approaches are explored: the Independence-based Heuristic, and the Sampling-based Heuristic. For the sampling-based heuristic, after the partitioning step, a learned density model is applied to approximately calculate the number of data points falling into each of the children-cells. Notice that an instance-optimized SFC is directly applicable to the class of learned multi-dimensional indexes operating on a projected space.

Workload-aware Z-Index (WAZI) *[152].* The WAZI is an extension of the previously proposed instance-optimized Z-index [153]. Given a spatial dataset and range query workloads, the proposed index optimizes the partitioning and Z-ordering. WAZI's partitioning technique produces cells that are accessed by a similar set of range queries. This enables the proposed index to reduce extraneous cell accesses during query processing. The benefit will persist with the proposed partitioning and ordering as long as the distribution of the query workload remains unchanged. During the index construction phase, a greedy algorithm is proposed for the partitioning and ordering of the cells. After the index construction phase, a particular partitioning and ordering are chosen by minimizing a pre-defined objective function based on the number of accessed data points. Here, **Random Forest Density Estimation** (**RFDE**) [205] models are used to approximate the exact data and query distributions.

5.2.2 Pure Learned Indexes in Native Space.

LearnedBF *[136].* LearnedBF extends the concept of the one-dimensional LBF [102] to design a learned multi-dimensional bloom filter. When constructing a LearnedBF, the strings with k-tuples are converted into an embedding vector. RNN with gated recurrent units [36] are applied for the

embedding of high-cardinality attributes. On the other hand, a direct embedding technique is used for low-cardinality attributes. Moreover, it has been observed that a learned bloom-filter can handle multidimensional data (e.g., k-tuple) effectively if there is a "co-occurrence structure" between in-index k-tuples. Based on the observation, the embedding vectors are concatenated and are sent as input to a densely connected neural-network layer. The output of this layer is used as input to a sigmoid function. After that, a sandwich structure [143] is employed to get the final output of LearnedBF.

***CompressLBF** [44].* CompressLBF has been introduced to reduce space consumption of the previously proposed LearnedBF [136]. It is observed that the size of the trained ML model is significantly impacted by the number of distinct values in the input. Specifically, for a column with a high number of unique values, the size of the corresponding embedding matrix grows linearly. Thus, it has been proposed to compress the input embedding by splitting a column into several sub-columns where the number of sub-columns is pre-defined. This split operation creates sub-columns with fewer dimensions. As a result, the embedding of the input will be smaller in size, and hence the proposed filter requires less space. Moreover, the compressed filter achieves high accuracy and saves model training time.

***Query-data Routing Tree (Qd-tree)** [214].* It has been observed that modern big data analytical systems partition data mainly by two approaches: (i) Hash/time-based, and (ii) Clustering-based [88]. These existing methods do not consider the distribution of the query workload while partitioning the data. Given a particular data and query workloads, the Qd-tree leverages **Reinforcement Learning (RL)** [92] to partition the data so that the number of blocks accessed by a particular query workload is minimized. The Qd-tree can be considered as a workload-aware multi-dimensional index structure where each non-leaf node partitions the data using a particular query predicate. Moreover, data in a leaf node is routed to the same disk block. For the formulation of the Qd-tree construction process as a **Markov Decision Process** (**MDP**) [159], the set of nodes are represented as states, the action space is represented as the set of query predicates (i.e., allowed cuts), and the reward is calculated using the number of skipped blocks over all queries. Notice that calculating the actual reward (i.e., number of skipped blocks) by executing queries is a costly process. As a result, a sampling method has been leveraged to avoid the costly query execution. On the other hand, the proposed RL agent, namely, "Woodblock", consists mainly of two learnable networks: (i) Policy network, and (ii) Value network. Moreover, **Proximal Policy Optimization (PPO)** [177] is used as the underlying learning algorithm.

***LMI-2** [183].* LMI-1 [13] is a Learned Metric Index that leverages predictions from a hierarchical structure with ML models for query processing. LMI-1 is a hybrid structure that requires a pre-existing index structure to partition the data. LMI-2 extends LMI-1 by eliminating the requirement of a pre-existing index structure to partition the data objects. Moreover, LMI-2 adopts an unsupervised approach for ML model training by leveraging clustering [88] techniques. Thus, LMI-2 requires less index construction time than LMI-1 and outperforms LMI-1 in terms of query processing time.

5.2.3 Hybrid Learned Indexes in the Projected Space.

***Distance-bounded** [219].* Most traditional spatial data processing methods follow a pipeline of coarse-grained filtering followed by exact geometric tests. In contrast, the method proposed in [219] advocates for fine-grained raster approximations to eliminate the need for exact geometric tests, prioritizing efficiency over accuracy. As a result, the proposed method leverages a raster-based approximation of spatial objects for approximate query processing. Moreover, a user-defined distance bound (i.e., error bound) is used to control the accuracy of the spatial approximation. Based on the user defined distance bound, the space is divided into uniform two-dimensional raster cells.

Subsequently, the two-dimensional raster cells are projected into a one-dimensional array using a SFC. After that a one-dimensional learned index (e.g., RS [97]) is constructed to learn the position of the cells in the one-dimensional array. Notice that the learned index is used for approximate point query processing. On the other hand, to enhance query performance for polygons, an **Adaptive Cell Trie** (**ACT**) structure [95] is used.

5.2.4 *Hybrid Learned Indexes in Native Space.*

Flood *[148].* Flood is a clustered in-memory read-only learned multi-dimensional index (for column stores) optimized for specific datasets and query workloads. During the index construction phase, Flood takes query workloads as input and learns to automatically create a data layout optimized for the given query distribution. Given a dataset with d dimensions, Flood models the empirical CDF of each dimension using RMI [102]. Then, Flood uses these models to create partitions of equal size, ensuring each partition contains an equal number of data points. These partitions are grouped to form a grid structure. Flood optimizes the data layout for particular workloads by adjusting the number of partitions in each dimension. It uses d-1 dimensions for partitioning, designating the last dimension as the "sort dimension". Flood fine-tunes the parameters of its grid structure using a cost model, and applies a gradient-descent algorithm to the cost model to optimize its parameters jointly. During query processing, for each query, Flood identifies all grid cells intersected by the query, and follows a projection-refinement-scan approach.

Tsunami *[49].* Traditional multi-dimensional indexes, e.g., the *k*-d tree [22], partition the underlying space based on the data distribution alone. In contrast, learned indexes, e.g., Flood [148], optimize for both data and query workloads. However, Flood's performance suffers under skewed query workloads and correlated data. To overcome Flood's limitations, Tsunami is proposed. Tsunami introduces a Grid Tree structure and an Augmented Grid to address the issues of skewed query workloads and correlated data, respectively. The proposed grid tree is a lightweight Decision Tree [163], and the augmented grid uses conditional CDFs and functional mappings. For the grid tree, Tsunami clusters queries by selectivity and builds a Grid Tree for each query type. Each node in the Grid Tree is split into several ranges along one dimension (except leaf nodes) until it reaches a minimum threshold or exhibits low query skew. For the augmented grid structure, Tsunami employs three strategies: (i) Partitioning Dimension X independently by its CDF (similar to Flood), (ii) Eliminating Dimension X by constructing a mapping from X to Y (if they are monotonically correlated), and (iii) Partitioning X dependent on Dimension Y by CDF(X|Y). To find the best augmented grid structure, Tsunami defines the number of partitions and a search strategy including the above three approaches. Then, it uses adaptive gradient descent to iteratively search for an optimal strategy. Tsunami outperforms the previously proposed Flood in scenarios with skewed query workloads and correlated data.

Spatial Interpolation Function-based Grid (SPRIG) *[225].* SPRIG is a grid-based learned multi-dimensional index designed for read-only workloads. SPRIG applies a sampling technique to the dataset, and uses the sampled data to build an adaptive grid structure. Then, it leverages the sampled data to fit a spatial interpolation function [142], specifically using the bilinear interpolation function. During query processing, given a search key, SPRIG predicts the approximate position of the key using the learned interpolation function. Notice that SPRIG requires a local search as an error correction mechanism following the prediction step. Moreover, to provide an error bound, SPRIG introduces a maximum estimation error calculated using the query workload. SPRIG can process both range and kNN queries.

SPRIG+ *[226].* Observing the imprecision of a single spatial interpolation function and the large prediction error in SPRIG [225], SPRIG+ augments space-partitioning trees with a prediction

mechanism. Firstly, SPRIG+ divides the two-dimensional grid into four sub-regions recursively. Then, a spatial interpolation function is learned for each region. To optimize storage space, SPRIG+ stores only the integer coordinates (used for calculating the interpolation function coefficients) instead of keeping the double coefficients (for lazy calculation). Additionally, SPRIG+ compresses all data into a bit vector, ensuring that the number of bits occupied by each integer remains as small as possible.

***CaseSpatial** [154].* The techniques proposed in Flood [148] have been applied to five in-memory traditional spatial index structures in CaseSpatial [154]: Fixed-grid [23], Adaptive-grid [149], k-d Tree [22], Quad-tree [173], and STR-tree [108]. Moreover, the query processing in all these techniques is performed in three steps: Index lookup, Refinement, and Scan. Similar to Flood, data in each partition are sorted using one dimension. During range query processing, while searching within a partition, it is proposed to use a one-dimensional learned index (e.g., RS [97]) on the sorted dimension instead of a binary search. As a result, performance has improved for range queries with low selectivity. However, there is less improvement in the case of range queries with high selectivity. This has led to the conclusion that the performance of a range query with low selectivity can be significantly improved by employing learned indexing techniques. It has also been observed that filtering in one dimension and refining in the other dimension using a one-dimensional learned index can outperform methods that filter on two dimensions.

***Spatial-LS** [155].* Spatial-LS is an extension of the techniques proposed in CaseSpatial [154]. Here, six spatial partitioning techniques have been considered for experiments, and these techniques are applied to achieve instance-optimization. Notice that CaseSpatial [154] supports only range queries. In contrast, Spatial-LS supports point, range, distance, and spatial join queries. Moreover, their extensive experimental studies show that: (i) Properly tuned grid-based structures can outperform tree-based structures due to fewer random accesses and the benefit of learned search within larger partitions; (ii) Within a large partition, a learned model performs better than binary search; and (iii) The impact of ML-enhanced grid-based index structures is less in the case of queries with high selectivity. Notice that all these techniques are developed considering an in-memory setup. As a result, the benefits of the proposed techniques might not be directly applicable in a disk-based setup due to the diminished importance of searching within each partition.

***Correlation-Aware Indexing (COAX)** [75].* It has been observed that two or more attributes are correlated in many real-world multi-dimensional datasets. As a result, these correlations can be leveraged for dimensionality reduction. COAX learns the correlations among the attributes. The main idea is to construct a multi-dimensional index by excluding the highly-correlated attributes. As a result, the constructed index will be significantly reduced in size, and that can enhance query processing performance. Moreover, if a query involves a non-indexed attribute, say $a1$, that is correlated with another indexed attribute $a2$, COAX uses only the correlated indexed attribute to process the query. COAX learns the correlation by learning **Soft Functional Dependency** (**softFD**) [86]. Furthermore, COAX is implemented with Grid Files [149] by adopting a hybrid approach and supports point and range queries.

***ML-enhanced** [93].* The performance of existing pure learned multi-dimensional indexes degrades for high-dimensional data due to the "curse of dimensionality". ML models have been incorporated into different traditional high-dimensional indexes (e.g., VA+Index [56], DS-tree [202], iSAX [182]) to build the ML-enhanced variants of these traditional index structures. Notice that an R-tree has been used for implementing the VA+Index structure. Moreover, the proposed ML-enhanced indexes focus on improving recall for approximate kNN query processing in high-dimensional data. The underlying intuition behind all the ML-enhanced indexes is "once an index

is built, the distribution of the nearest neighbors given a query object has been fixed, and therefore is learnable". As a result, the proposed ML-enhanced indexes use deep neural networks to guide **k-nearest-neighbor** (**kNN**) search on traditional tree-based indexes. Particularly, the problem of kNN query processing has been formulated as a multi-class classification problem [6], where the goal is to predict the leaf nodes that contain the nearest neighbors given an input query. Hence, by leveraging the predictions of the ML models, the ML-enhanced indexes improve the leaf node access order of their traditional counterparts.

***AI+R-tree** [8].* Similar to the ML-enhanced index [93], the AI+R-tree incorporates ML models to enhance the performance of a traditional R-tree. As areas covered by R-tree nodes overlap in space, searching for a single object may require exploring multiple paths from root to leaf. This overlapping issue negatively impacts R-tree query processing performance. In the AI+R-tree, for a range query, an overlap ratio is proposed to quantify the degree of unnecessary leaf node accesses. Moreover, range query processing in an R-tree has been formulated as a multi-label classification problem [81]. Specifically, a new AI-tree is designed that trains multiple ML models to directly predict the true leaf node IDs for an input range query. To improve the query processing time of a traditional R-tree in the case of high-overlap queries, a hybrid AI+R-tree is proposed that processes high- and low-overlap queries using the AI-tree and the regular R-tree, respectively. The AI+R-tree uses the overlap ratio to train an ML model, enabling it to classify an input query as high- or low-overlap. It assumes that the data and query workloads are fixed. For efficient query processing, it also leverages a traditional grid structure to index the learned ML models [14].

***PolyFit** [120].* PolyFit has been proposed to process approximate range aggregate queries (e.g., SUM, COUNT). It leverages piecewise polynomial functions for query processing, and constructs an index based on polynomial fitting for intervals of data. The choice of polynomial functions is due to their observed benefits over linear regression. Multiple polynomial functions are used to minimize the fitting error, as a single polynomial function might not fit the entire dataset accurately. Additionally, a greedy segmentation method has been proposed to reduce the number of polynomial functions. PolyFit is extended to support queries over multi-dimensional data by estimating cumulative functions for multiple keys and employing greedy segmentation techniques. However, due to the quadratic time complexity of the greedy approach in the multi-dimensional case, a Quad-tree [173] has been incorporated to identify segments.

***LearnedKD** [216].* LearnedKD combines a traditional k-d tree [22] with ML models to enhance the kNN query processing performance. The ML model training of LearnedKD is a two-step process: Firstly, kNN queries are processed using the traditional k-d tree, and the positions of the kNNs are recorded as labels. Secondly, a deep neural network model is trained on the prepared training dataset. During query processing, given a kNN query, the trained ML model is invoked to predict the data objects that are nearest neighbors. This supervised learning process enables the model to predict whether an object is a nearest neighbor for an input query. Notice that the ML model's output does not always directly predict the true kNNs. As a result, in a post-processing step, the true kNNs are calculated from the set of predicted (i.e., potential) kNNs.

***Learned Metric Index (LMI-1)** [13].* Inspired by the benefits of learned multi-dimensional indexes (e.g., Flood [148]), an LMI-1 has been proposed [13]. LMI-1 is one of the earliest works to extend learned multi-dimensional indexes into the metric space. LMI-1 indexes data in metric space, where similar data objects are clustered together using a distance metric. Notice that LMI-1 is a hybrid structure that requires a pre-existing traditional index structure to partition the data. LMI-1 proposes to replace the internal nodes of the traditional index structure with ML models. Particularly, it uses the positions of data objects in the traditional index as labels to train a hierarchy

of supervised ML models. As a result, LMI-1 can avoid the costly distance calculations entirely during the query processing phase.

6 Learned Mutable Fixed Data Layout Indexes

We present the Mutable Fixed Data Layout Learned Indexes in both the one- and multi-dimensional spaces.

6.1 The One-dimensional Case

6.1.1 Pure Learned Indexes with In-Place Insertion. We present the core concepts of ALEX [48], a one-dimensional index that highlights the class of Mutable Pure Learned Indexes with Fixed Data Layout and an In-place Insertion.

***ALEX** [48].* ALEX is an in-memory updatable learned index structure. Its core elements include: (i) RMI as the ML model hierarchical structure, and (ii) A **Gapped Array** (**GA**) and A **Packed Memory Array** (**PMA**) [21] as leaf node layouts in RMI. ALEX adopts an adaptive RMI structure capable of handling the insertion of new keys. During initialization, the root model is executed first, partitioning the key space among its child nodes, that in turn are recursively initialized. Each non-root node is assigned a fixed number of partitions. When a partition size is appropriate, leaf nodes are created in the form of either GA or PMA to support inserts. The main idea behind the proposed GA is to maintain space/gaps at the leaf level so the structure can accommodate in-place ML model-based insertions. Moreover, if the partition size exceeds the maximum number of keys, a new inner node will be created. On the other hand, if the partition size is too small, adjacent partitions will be merged to avoid wasting leaf nodes. An extended version of ALEX, termed ALEX+, with concurrency support has been introduced in [206].

Other indexes in this category are as follows. AIDEL [115] leverages independence among ML models to build a scalable index. TALI [73] uses the update distribution of data for efficient lookup and insertion operations. StableBF [127] designs a learned bloom filter for handling data streams. DriftModel [76] addresses the issue of drift correction (i.e., model error due to updates) in updatable learned indexes. SALI [68] employs probability models to construct a scalable index structure. APEX [131] extends ALEX to develop an index optimized for persistent memory. CARMI [223] offers a cache-friendly extension of the RMI index. EWALI [125] improves the write performance of a learned index by using dual buffers. TONE [228] reduces the tail latency of a learned index through a two-level leaf design. COLIN [230] is a cache-friendly learned index structure that leverages a mixture of learned and simple nodes (i.e., a heterogeneous node structure). A recently proposed AirIndex [37] can optimize the one-dimensional learned indexes (e.g., ALEX) by performing a graph-based search to find the optimal index parameters. Notice that the AirIndex [37] does not introduce a new type of index but rather combines existing learned and traditional indexes within a single framework.

6.1.2 Pure Learned Indexes with Delta-Buffer Insertion. We present the core concepts of PGM [62], a 1D-index that highlights the class of Mutable Pure Learned Indexes with Fixed Data Layout and Delta-Buffer Insertion.

***Piecewise Geometric Model (PGM)** [62].* It has been observed that the initially proposed one-dimensional learned indexes (e.g., RMI) provide an empirical error bound without any formal worst-case bound. As a result, the PGM is proposed with a guaranteed worst-case bound. Although PGM is designed as a mutable learned index structure, its performance gain is much higher in the static setting. The steps for constructing the PGM index are as follows: (i) Computing optimal piecewise linear segments with a pre-fixed model error bound ϵ, (ii) Storing the segments as a triplet of key, slope, and intercept, (iii) Repeat the previous steps recursively over the subset of

keys. PGM uses a delta buffer insertion strategy (similar to the insertion strategy of LSM-tree-like structures) to support dynamic inserts. Moreover, three variants of PGM have been proposed: (i) Compressed PGM for space efficiency, (ii) Self-adaptive PGM for a particular query distribution, and (iii) Multi-criteria PGM that optimizes for a user-given requirement, e.g., query time.

Other indexes in this category are as follows. ASLM [119] employs simple single-layer ML models for efficient update handling. Doraemon [189] re-uses pre-trained ML models for similar data distributions to reduce the model re-training cost. XIndex [190] supports concurrency natively by leveraging a two-phase compaction scheme. In ConcurrentLI [203], XIndex has been extended to XIndex-R and XIndex-H as range and hash indexes, respectively. Similar to XIndex, SIndex [201] is a concurrent learned index specifically designed for indexing strings. FINEdex [114] avoids using a large delta buffer and employs instead a delta per training record to support data insertion. FILM [135] uses a cold data identification mechanism that enables efficient data swapping between disk and memory. DiffLex [41] is a NUMA-aware learned index that leverages sparse and dense arrays based on the hotness (i.e., cold vs. hot) of the keys. PLIN [229] utilizes Optimal Piecewise Linear Representation [211] to build an index optimized for non-volatile memory.

6.1.3 Hybrid Learned Indexes. In this section, we present the core concepts of Bourbon [42], a One-dimensional that highlights the class of Hybrid Learned Indexes with Mutable Fixed Data Layout.

***Bourbon** [42].* Bourbon incorporates ML models to enhance the performance of an LSM-tree [134]. Moreover, Bourbon is an updatable ML-enhanced LSM-tree that has been integrated inside WiscKey, a commercial key-value store [133]. A key observation in Bourbon is that immutable SSTables are suitable for learning because there are no in-place updates. Particularly, the search index block component of the LSM-tree has been replaced with ML models. However, the learning of the SSTables has been categorized based on whether the SSTables are short- or long-lived. As a result, at runtime, a simple cost-benefit analyzer has been used to decide whether learning is beneficial.

The other indexes in this category are as follows. TridentKV [132] adopts an optimized learned index block structure to build an ML-enhanced LSM-tree. SA-LSM [227] exploits a survival analysis technique in an LSM-tree based key-value store. FITing-tree [65] combines a traditional B^+-tree with piecewise linear functions. AccB^+tree [130] employs linear models to enhance the search operation of a traditional B^+-tree. MADEX [78] accelerates the intra-page lookup performance of a traditional B^+-tree by leveraging ML models. Sieve [192] incorporates piecewise linear models with a traditional B^+-tree to design an index for efficient block-skipping. LIFOSS [218] exploits a two-layer learning model to design an index targeted for data streams. IFB-tree [77] designs interpolation-friendly nodes to enhance the performance of a traditional B-tree. Hybrid-LR [162] exploits the benefit of linear regression models and B-trees to design a hybrid structure. PLBF [194], PLBF++ [176], SNARF [193], IA-LBF [24], PDDBF [222], BF-Sandwich [143], and AdaBF [43] combine ML models with traditional bloom filters to build learned bloom filters. FLIRT [212] exploits a circular queue structure to design a parameter-free index for data streams. HERMIT [209] uses an ML-enhanced **Tiered Regression Search Tree** (**TRS-Tree**) to detect column correlations. S3-SkipList [224] employs ML models to select guard entries in a traditional skip-list structure [158]. RUSLI [141] extends RS [97] to support data updates in real time.

6.2 The Multi-dimensional Case

6.2.1 A Summary of the Properties of the Mutable Fixed Data Layout Indexes. The properties of the class of mutable fixed data layout learned multi-dimensional indexes are presented in Table 2. We highlight the types of queries supported by each of the indexes. For point queries, some indexes

Table 2. Properties of the Mutable Fixed Data Layout Learned Multi-dimensional Indexes

Index	Data Layout	Pure Learned	Hybrid Learned	Data Space	Point Query	Range Query	kNN Query	Join Query
GLIN-ALEX [196]	Fixed	In-place	×	Projected	Exact	Exact	×	×
SLBRIN [198]	Fixed	×	BRIN	Projected	Exact	Exact	Exact	×
LSTI [50]	Fixed	×	Radix Table	Projected	Exact	Exact	Exact	×
IF-X [80]	Fixed	×	R-tree	Native	Exact	Exact	×	×
Period Index [20]	Fixed	×	Grid	Native	Exact	Exact	×	×

do not explicitly provide a query processing algorithm or experimental results. However, if an index can easily be extended to support point queries, we have considered that the index supports point query processing. On the other hand, due to the nature of the application domain, an index might be designed emphasizing efficiency over accuracy. As a result, an index can support either exact/approximate query processing. Here, approximate query processing refers to the event of missing some results from the set of exact answers. Here, we also indicate whether a supported query type returns an exact or an approximate answer.

6.2.2 *Pure Learned Indexes with In-place Insertion in the Projected Space.*

***GLIN-ALEX** [196].* **Generic Learned Indexing** (**GLIN**) is a lightweight structure to index complex geometric objects (e.g., polygons). GLIN projects the multi-dimensional geometric objects into one-dimensional intervals of Z-values. After that, the objects are sorted based on their minimum Z-value. As a result, any existing order-preserving one-dimensional traditional or learned indexing technique can be applied to the sorted values. Here, GLIN uses both B-tree and ALEX [48] indexes to produce GLIN-BTREE and GLIN-ALEX, respectively. Moreover, for a particular leaf node, GLIN maintains the **Minimum Bounding Rectangles** (**MBRs**) of all objects of that node. As a result, during the ML model-based refinement step, a leaf node can be skipped if its MBR does not overlap with the query rectangle's MBR. GLIN can support *containment* and *intersect* queries for complex geometric objects. However, GLIN can have true negative results for *intersect* queries. As a result, GLIN adopts a query augmentation technique to widen the Z-value interval. This query augmentation technique introduces a tradeoff between correctness and pruning time. As ALEX adopts an in-place insertion strategy, GLIN-ALEX also uses the same strategy to support inserts.

6.2.3 *Hybrid Learned Indexes in the Projected Space.*

***Spatial Learned Index Based on Block Range Index (SLBRIN)** [198].* SLBRIN extends the **Block Range Index** (**BRIN**) [217] in the context of learned indexing. It aims to achieve high performance query processing while maintaining efficient update operation. The main idea is to divide the index objects into two components: **History Range** (**HR**) and **Current Range** (**CR**). HR and CR are beneficial for query processing and update handling, respectively. For creating HR and index entries, SLBRIN first projects the multi-dimensional data into the one-dimensional space using Geohash [124]. This enables imposing a linear ordering using the Geohash values. After that, HR with index entries are partitioned recursively into ranges so that a one-dimensional learned index can be built for each range. SLBRIN employs **Multi-layer Perceptron** (**MLP**) as the ML model for training. Notice that CR remains empty during this index construction phase because there are no update operations at this step. During an update operation, the data will be encoded using Geohash to create the index entries. After that, the new index entries will be stored in CR for a short period of time. The new entries will be moved to HR during the periodic merge operation of CR. After a merge CR operation, a model re-training process is invoked to correct the error bound. SLBRIN can support point, range, and k-NN queries.

***LSTI** [50].* LSTI is a learned multi-dimensional index for processing spatio-textual data. Here, the multi-dimensional data is first projected into the one-dimensional space using Z/Morton-order [147, 151]. After that, techniques similar to the one-dimensional learned index RS [97] are applied to the projected data. The proposed index can support data updates using techniques similar to the one-dimensional updateable index ALEX [48]. Moreover, it can process four types of queries: boolean point, boolean range, boolean kNN, and top-k text similarity. Given a specific text description. The boolean point, range, kNN, and top-k queries retrieve a specific location, the objects in a specific region, the objects closest to a specific location, and multiple objects (with approximate text descriptions) in a specific spatial region, respectively. Notice that two parameters: max_err and the number of radix bits significantly impact the index performance. As a result, a Random Forest [30] regression model has been used to learn the optimal index parameters for specific data and query workloads.

6.2.4 Hybrid Learned Indexes in Native Space. Hybrid learned indexes combine traditional index structures with ML models to build ML-enhanced index structures. Here, we present the hybrid learned indexes for the multi-dimensional space where these indexes operate in the native data space.

***Interpolation Friendly Indexes (IF-X)** [80].* IF-X have been introduced in [80]. IF-X leverages the idea of augmenting traditional index structures (e.g., the R-tree) with ML models. Particularly, a lightweight technique, e.g., linear interpolation [70], has been used for one of the dimensions. The use of a simple linear interpolation method offers the following benefits: faster computation time and fewer parameters. Moreover, to minimize interpolation error, the proposed index sorts the data entries on the most suitable dimension that is selected based on the least model prediction error. Furthermore, for performing interpolation, the structure of the leaf nodes of the index has also been modified so that the nodes can accommodate the additional information needed for the ML models. Notice that the proposed technique does not consider the query workload when selecting the sort dimension.

***Period Index** [20].* Period Index has been proposed to index intervals (i.e., temporal period data) by position and duration. A grid-based data structure with constant time lookup is utilized. The core idea of the proposed index can be divided into two parts: First, splitting the timeline into buckets of fixed size, where each bucket is further partitioned into cells (referring to the position of the interval); Second, arranging the cells in levels (referring to the duration of the interval). Moreover, it proposes finding the length of the buckets by adapting to the underlying distribution using a cumulative histogram. Hence, an improved version of the Period index, termed Period Index*, has been proposed with this adaptive bucket length. Period Index* can be viewed as a learning-enhanced version of the Period Index. The proposed index can process range, duration, and range-duration queries.

6.2.5 Open Branches in the Taxonomy. As of the time of writing this article, there are no mutable learned multi-dimensional indexes in any of the following categories: (i) Mutable Fixed Data Layout Multi-dimensional Pure Learned Indexes with In-place Insertion Strategy in Native Space, and (ii) Mutable Fixed Data Layout Multi-dimensional Pure Learned Indexes with Delta-Buffer Insertion Strategy. We envision that the above mentioned open branches will be useful for classifying learned multi-dimensional indexes in the future.

7 Learned Mutable Indexes with Dynamic Data Layout

For the class of mutable learned indexes, if the layout of the data is arranged/re-arranged by the ML models while building the learned index, we refer to them as having a dynamic data layout. In

this section, we present the class of Mutable Learned Indexes with Dynamic Data Layout in both the one- and the multi-dimensional spaces.

7.1 The One-dimensional Case

7.1.1 Pure Learned Indexes with In-place Insertion. We study the core concepts of **Learned Index with Precise Position** (**LIPP**) [207] to highlight the class of Mutable Pure Learned Indexes with Dynamic Data Layout and In-place Insertion for the 1d-case.

***LIPP** [207].* It has been observed that the performance of an updatable one-dimensional learned index, e.g., ALEX [48], degrades in the presence of the ML model's misprediction. This degradation occurs because searching after an imprecise prediction always incurs overhead. To address this issue, the LIPPs has been proposed. LIPP avoids misprediction by ensuring that the key-to-position mapping is precise and eliminating the need for localized search following each misprediction. Thus, the lookup cost is bounded by the tree height. Also, each LIPP node contains an ML model, a bit vector indicating entry type, and an array of entries. Each entry can be of Type NULL, DATA, or NODE. Type NULL indicates an unused slot, DATA represents a single entry, and NODE denotes a child node at the next level. Notably, LIPP does not distinguish between non-leaf and leaf nodes. During LIPP construction, if multiple keys map to the same position, a new child node is created. For even distribution of mapping, kernelized linear models are utilized. Since LIPP is a sorted index, the selected kernel function must be monotonically increasing. The **Fastest Minimum Conflict Degree** (**FMCD**) algorithm is proposed for computing the model of each LIPP node. LIPP employs in-place insertion akin to ALEX. LIPP+, has been introduced [206] to support concurrency.

Other indexes in this category are as follows. NFL [208] exploits a key distribution-transformation method to build the learned index in the easy-to-learn transformed space. DILI [117] uses linear regression models to realize a distribution-driven index. WIPE [204] is designed to reduce write-amplification in non-volatile memory.

7.1.2 Open Branches in the Taxonomy. As of now, there are no mutable learned one-dimensional indexes in any of the following categories in the one-dimensional case: (i) Mutable Pure Learned Indexes with Dynamic Data Layout and Delta-Buffer Insertion Strategy, and (ii) Mutable Hybrid Learned Indexes with Dynamic Data Layout. We envision that the above mentioned open branches will be useful for classifying learned one-dimensional indexes in the future.

7.2 The Multi-dimensional Case

7.2.1 A Summary of the Properties of the Mutable Indexes with Dynamic Data Layout. The properties of the class of mutable learned multi-dimensional indexes with dynamic data layout are presented in Table 3. The table gives the types of queries supported by each learned multi-dimensional index. For point queries, some indexes do not explicitly provide a query processing algorithm or experimental results. However, if an index can easily be extended to support point queries, we have considered that the index supports point query processing. Also, we indicate if a supported query type returns an exact or approximate answer. We exclude learned multi-dimensional bloom filters (e.g., LPBF [234], PA-LBF [221]) because they are considered probabilistic existence index structures.

7.2.2 Pure Indexes with In-place Insertion in the Projected Space.

***LISA** [116].* Considering the issues of storage consumption and high I/O cost of the R-Tree, LISA, a disk-based Learned Index for Spatial Data, has been introduced. The main idea of LISA is to generate a searchable data layout in disk pages for an arbitrary spatial dataset using a machine learning model. LISA consists of four parts: (i) Space partitioning into a series of grid cells; (ii) A

Table 3. Properties of the Mutable Learned Indexes with Dynamic Data Layout in the Multi-dimensional Space

Index	Data Layout	Pure Learned	Hybrid Learned	Data Space	Point Query	Range Query	kNN Query	Join Query
LISA [116]	Dynamic	In-place	×	Projected	Exact	Exact	Exact	×
RSMI [160]	Dynamic	In-place	×	Projected	Exact	Approx.	Approx.	×
BMT [112]	Dynamic	In-place	×	Projected	Exact	Exact	Exact	×
LMSFC [66]	Dynamic	In-place	×	Projected	Exact	Exact	×	×
LIMS [191]	Dynamic	In-place	×	Native	Exact	Exact	Exact	×
MTO [47]	Dynamic	In-place	×	Native	Exact	Exact	×	Exact
WISK [181]	Dynamic	Delta Buffer	×	Native	Exact	Exact	Exact	×
LPBF [234]	Dynamic	×	CBF	Projected	—	—	—	—
PA-LBF [221]	Dynamic	×	CBF	Projected	—	—	—	—
RLR-tree [72]	Dynamic	×	R-tree	Native	Exact	Exact	Exact	Exact
RW-tree [52]	Dynamic	×	R-tree	Native	Exact	Exact	Exact	×
ACR-tree [83]	Dynamic	×	R-tree	Native	Exact	Exact	Exact	×
PLATON [213]	Dynamic	×	R-tree	Native	Exact	Exact	Exact	×
Waffle [38]	Dynamic	×	Grid	Native	Exact	Exact	Exact	×
FHSIE [187]	Dynamic	×	Grid	Native	Exact	Exact	Exact	×

mapping function M for mapping search keys to 1D values; (iii) A monotonic prediction function SP that takes the output of M as input and predicts the shard ID; (iv) Local models for each shard for allocating, splitting, and merging pages. During query processing, given a query rectangle *qr*, it is decomposed into smaller rectangles. Then, the shard prediction function *SP* is used to select the shard in each cell (overlapping with *qr*). Subsequently, local models are utilized to find the address of pages overlapping with *qr*. LISA supports range queries, KNN queries, insertions, and deletions. For KNN query processing, each KNN query is converted into a series of range queries. LISA supports dynamic insertions by employing a model-based in-place insertion policy.

Recursive Spatial Model Index (RSMI) *[160].* RSMI is a disk-based updateable learned spatial index. Due to uneven gaps in the empirical CDF of the Z-order space-filling curve, RSMI leverages rank space-based ordering [161] to project multi-dimensional points into the one-dimensional space. Data points are sorted in ascending order based on their one-dimensional projection values. Points are packed in blocks based on a predefined block size. To scale with larger datasets, RSMI recursively partitions the data and trains models for each partition. Moreover, multiple ML models are trained to learn the mapping (i.e., from search key to disk block ID) in the projected space. RSMI can support point, range, and kNN queries on point data. The proposed RSMI produces approximate answers with high accuracy. For exact query processing, RSMI advocates for a search operation similar to an R-tree. Note that all the proposed techniques are primarily focused on point data. Query processing performance may be negatively impacted if the proposed method is applied to data objects with extension. RSMI supports dynamic inserts by employing an in-place insertion policy.

Bit Merging Tree (BMT) *[112].* It has been observed that each existing SFC has a pre-fixed projection function that neither considers the underlying data distribution nor the query workloads. As a result, a specific SFC cannot perform well across a variety of datasets and query workloads because a fixed **Bit Merging Pattern** (**BMP**) is applied to the entire dataset to build a particular SFC (e.g., bit interleaving for Z-order). Thus, for a given data and query workloads, the core idea is to apply different BMPs to different subspaces to construct a Piecewise SFC. Hence, a tree-based structure has been designed where each leaf node refers to a subspace. Moreover, for each subspace

(i.e., leaf node), the path from root to leaf creates the BMP by representing each internal node as a bit value for different dimensions. The proposed BMT preserves two important properties of an SFC: Injection (i.e., generating a unique value for a given input), and Monotonicity [106]. The BMT construction process has been formulated as a sequential decision-making process so that RL-based techniques can be leveraged to learn an effective BMT construction policy. Particularly, a greedy policy has been incorporated into a **Monte Carlo Tree Search** (**MCTS**) [31]. The main idea of the greedy policy is to choose an action (i.e., choose a bit for each node from a pool of candidate bits) so that the RL agent can maximize its reward (e.g., query performance). Notice that executing queries on BMT to calculate the reward is a costly process. As a result, a new metric called ScanRange has been proposed to calculate the reward efficiently. The proposed BMT has been integrated into two previously proposed learned indexes as a replacement of a traditional SFC: BMT+RSMI [160] and BMT+ZM-index [197].

Learned Monotonic Space Filling Curve (LMSFC) *[66].* LMSFC has been proposed to learn a parameterized monotonic Z-order for given data and query workloads. The outcome of learning a parameterized Z-order is to find parameters for a particular instance to minimize query processing cost. To minimize the cost of query processing for a given instance, the problem of learning an optimal parameterized SFC is formulated as an optimization problem. A Bayesian optimization method termed SMBO [84] is used to approximately solve the optimization problem. Then, an offline dynamic programming-based (and a heuristic-based approach) is proposed to pack data points into disk pages based on a density-based cost function. The goal is to reduce the dead space of the MBRs of the disk pages by packing the data points as tightly as possible. After packing the points into the pages, the minimum z-value from each page is used to create a sorted array. Then, a one-dimensional learned index (e.g., PGM [62]) is used to find the page (given a z-value as input). However, even with an instance-optimized Z-order SFC, the issue of false positives remains. As a result, recursive splitting of the query rectangle is proposed to further reduce the number of false positives during the search operation. The proposed LMSFC supports inserts by adopting an in-place insertion strategy.

7.2.3 Pure Indexes with In-place Insertion in Native Space.

Learned Index for Metric Space (LIMS) *[191].* It has been observed that most existing techniques for learned multi-dimensional indexes (in vector space) may not be directly applicable for indexing in metric space due to the lack of coordinate structure and dimension information. Moreover, only the property of triangular inequality can be leveraged for pruning in a metric space. As a result, an LIMS has been proposed to efficiently support point, range, and kNN queries in the metric space. To achieve these goals, LIMS clusters the data objects into multiple groups, and builds a learned index for each cluster. Additionally, it selects a set of data objects as pivots for each cluster so that the distance between the pivot and the data objects can be pre-computed. Notice that the pre-computation of distances in each cluster enables LIMS to build learned indexes on these distance values. It has been observed that the selection of pivots can significantly impact the performance during query processing.

Multi-Table Optimizer (MTO) *[47].* MTO is designed to reduce I/O cost for analytics over multi-table datasets. Similar to the Qd-tree [214], MTO leverages RL to route each record to the corresponding data block, ensuring that all elements in one block are from the same table. Besides using simple predicates (e.g., A.X $<$ 100), MTO can use join-induced predicates to optimize the layout for all tables. For join-induced predicates, it evaluates subqueries in these predicates to obtain indexes that are candidates for join operations. Then, these predicates and the table are sent to the Qd-tree as input to train the model. During query processing, MTO uses the Qd-tree to guide

block access. To support dynamic workloads, MTO adopts an in-place insertion policy while taking into account block access reduction and sub-tree reorganization cost in the reward function.

7.2.4 Pure Indexes with Delta Buffer Insertion in Native Space.

Workload-aware Learned Index for Spatial Keyword (WISK) *[181].* (WISK is a learned spatial index for keyword queries. Given data and query workloads, WISK employs ML models that can learn using both spatial and textual information. This is achieved in two phases: firstly, it learns an optimal data layout for a particular query workload, and builds the index based on the learned layout. The index construction steps are as follows: (i) ML models are trained to approximate the CDF of the spatio-textual objects, (ii) A cost estimation function is defined based on the learned CDF, (iii) SGD is applied to learn the optimal partition by minimizing the cost, (iv) The hierarchy of WISK is constructed using a bottom-up packing approach, (v) The packing process has been transformed into a sequential decision-making process so that it can be solved using an RL technique. As the query processing cost of WISK largely depends on how the data objects are partitioned during the construction of the index, an optimal partitioning problem has been formulated to construct the leaf nodes of the WISK. It has been shown that the problem of partitioning with optimal cost is NP-hard. As a result, a heuristic algorithm using **Stochastic Gradient Descent** (**SGD**) [29] has been proposed. WISK supports dynamic inserts by employing a delta-buffer insertion strategy.

7.2.5 Hybrid Learned Indexes in the Projected Space.

Learned Prefix Bloom Filter (LPBF) *[234].* LPBF is a learned bloom filter particularly designed for spatial data. The core idea of LPBF is to project multi-dimensional data into one-dimensional binary code using a space-filling curve (e.g., Z-order). After that, the prefixes of binary z-values are grouped into several clusters based on a pre-defined parameter. Moreover, the suffixes of the same prefix are learned using **sub-Learned Bloom Filters** (**sub-LBFs**) [136]. LPBF leverages a traditional **Counting Bloom Filter** (**CBF**) [55] as a backup filter to support data updates. The use of CBF and sub-LBF helps minimize the false positive rate and the model training time for the proposed method.

Prefix-Based and Adaptive Learned Bloom Filter (PA-LBF) *[221].* PA-LBF is an extended version of the previously proposed LPBF [234]. Notice that the concept of adaptation has been introduced during the process of sub-LBF model training. For each sub-LBF, the main extension is the use of an adaptive learning process to calculate the number of filter layers based on a pre-defined threshold. This adaptive learning approach enables PA-LBF to further minimize the false positive rate compared to the previously proposed LPBF.

7.2.6 Hybrid Learned Indexes in Native Space.

7.2.6.1 Reinforcement Learning-based R-Tree (RLR-tree) [72] In traditional R-tree construction, there are several existing node splitting strategies (e.g., linear, quadratic) [74]. For a chosen node splitting strategy, the query processing performance will vary for different data and query workloads. RLR-tree formulates the ChooseSubtree and Split operations of a traditional R-tree as an MDP [159] (i.e., sequential decision-making process). It uses RL to learn models for the ChooseSubtree and Split operations. Notice that the RLR-tree does modify existing query processing algorithms; instead, it aims to construct a better R-tree by optimizing the ChooseSubtree and Split operation. Thus, it can support existing R-tree-based query processing algorithms. Also, the RLR-tree gives the design of state space, action space, transition, and reward for MDP formulation For example, while inserting a new object, a state is a tree node with the following features: area increase, perimeter increase, overlap increase, and occupancy rate. For the action space, given a current state, the RL agent picks a child node into which the new entry will be inserted. For the transition, given a state and

an action, the agent moves to a child node. If the child node is a leaf node, the agent reaches a terminal state. To realize the reward function, the RLR-tree uses a reference tree with a pre-defined ChooseSubtree and Split strategy. The reward is calculated as the difference between the query processing cost of the RLR-tree and the reference R-tree. The RLR-tree adopts a Deep Q-Network learning method [144] for training the agent.

RW-tree *[52].* A traditional R-tree is constructed using one of the following methods: bulk loading vs. one-by-one insertion. In the context of one-by-one insertion, most existing techniques do not take the query workload into account while constructing the R-tree. As a result, a learning-based framework has been proposed for query workload-aware R-tree construction. The goal is to achieve better query processing performance for future queries generated from a similar query distribution. The proposed RW-tree is built on the following core concepts: learning the query workload distribution and leveraging a cost model for the accurate approximation of query execution time. For learning the workload distribution, in the pre-processing step, the entire space is partitioned into grid cells, and all the queries are mapped into their corresponding grid cell. After that, the statistics of the queries are collected per cell. Next, the grid cells are clustered based on queries with similar statistics. Given the learned workload distribution and a new data insertion choice, a cost model is proposed to approximately measure the query execution time. This enables the RW-tree to select the insertion strategy with the lowest cost. The RW-tree can process both range and k-NN queries.

Actor-Critic R-tree (ACR-tree) *[83].* It has been observed that existing methods for R-tree node packing can be categorized into the following types: (i) Heuristics to pack objects into parents in a bottom-up manner, or (ii) A greedy approach to partition nodes into child nodes in a top-down fashion. The goal of the existing methods is to optimize the R-tree construction for the short-term (i.e., without considering the long-term tree construction cost). Moreover, these heuristic-based or greedy methods also try to pack the nodes as full as possible. As the optimal R-tree node packing problem is NP-hard, an RL-based method has been proposed to construct the R-tree for optimizing the long-term tree cost. Particularly, the top-down R-tree construction methods have been formulated as an MDP [159]. Then, an Actor-Critic [71] based RL method has been applied to construct the ACR-tree. The Critic part is designated to estimate the long-term tree cost, and the Actor part is designated to make decisions (e.g., split or pack). The ACR-tree uses a grid-based method to better encode the spatial information. Moreover, based on the estimation of the long-term cost, it either splits a node or packs an object onto a single child node. Notice that the Actor-Critic training process is time-consuming. As a result, a bottom-up model training process with training sharing has been proposed to speed up the training process. The proposed ACR-tree supports both range and kNN queries.

PLATON *[213].* Similar to the ACR-tree [83], it has been observed that both top-down and bottom-up R-tree packing methods cannot adapt to particular data and query workloads due to the use of fixed heuristics. Also, the top-down packing method adheres to a sub-optimal node partitioning policy by ignoring the dependencies among different R-tree nodes. Thus, for a given data and query workloads, a top-down Packing with Learned Partition policy (PLATON, for short) is introduced. Notice that optimal R-tree packing has been proved to be NP-hard. The proposed node partitioning policy is learned by leveraging a RL-based technique (MCTS) [31]. However, in the presence of a large state space and long action sequence, the existing MCTS algorithm suffers from slow convergence. Hence, a divide-and-conquer strategy has been proposed to reduce the size of the state space. Moreover, optimization techniques, e.g., early termination and level-wise sampling, have been applied for faster convergence of the MCTS algorithm.

Waffle *[38].* Waffle is an in-memory grid-based indexing system for moving objects. Waffle is built on the following major components: grid index manager, lock manager, transaction manager, re-grid manager, and an online configuration tuning component. Notice that the proposed indexing system contains many different configuration knobs. As a result, the performance of Waffle heavily depends on the configuration of the knobs. Moreover, due to the dynamic nature of moving objects, the settings of different knobs require online adjustments. To address this issue, an online configuration tuning component, termed Wafflemaker, has been proposed. Wafflemaker is an RL-based component that suggests an optimized knob configuration for Waffle in an online fashion.

Fast Hybrid Spatial Index with External Memory Support (FHSIE) *[187].* The FHSIE has been proposed to efficiently extend the concept of learned spatial indexing to secondary storage. FHSIE clusters spatial objects using unsupervised ML techniques based on a few parameters. During the index construction phase, the K-means clustering method is applied recursively to the spatial data to build a height-balanced hierarchical structure. Then, a grid structure is incorporated at the bottom level of the previously created hierarchical structure. The purpose of the grid structure is to accurately find the intersections of bottom-level clusters with an input query. For extending FHSIE to external memory, the main idea is to allocate disk blocks for each of the following components separately: (i) Internal models, (ii) Bottom-level models, and (iii) Cells of the grid. FHSIE supports inserts using an in-place insertion strategy. Although the proposed index can efficiently support point, range, and KNN queries in secondary storage, the index construction time can be significantly higher than that of traditional spatial indexes.

7.3 Open Branches in the Taxonomy

As of the time of writing this survey article, there are no learned multi-dimensional indexes in the following category: Mutable Dynamic Data Layout Pure Learned Indexes with Delta-Buffer Insertion Strategy in Projected Space. However, this open branch will be useful for classifying learned multi-dimensional indexes in the future.

8 Benchmarking Learned Indexes

We present the studies conducted for benchmarking the one-dimensional and multi-dimensional learned indexes.

8.1 The One-dimensional Case

Several studies on benchmarking learned one-dimensional indexes are available in the literature. In [25], several ideas have been proposed for designing new benchmarks that are better suited to evaluate learned systems. The **Search On Sorted Data (SOSD)** Benchmark has been presented in [96, 138], and However, SOSD only considers in-memory read-only workloads. Moreover, SOSD covers three learned indexes: RMI [102], RS [97], and PGM [62]. A critical analysis of RMI has been presented in [137]. This analysis investigates the impact of each hyperparameter of RMI on prediction accuracy, lookup time, and build time. Moreover, it provides guidelines to configure RMI. In [12], a micro-architectural analysis of ALEX has been presented. This study goes beyond high-level metrics (e.g., latency and index size) towards lower-level metrics (e.g., cache misses per cache level, average number of execution cycles). The experiments are based on Intel's **Top-Down Micro-architecture Analysis Method (TMAM)** [215]. Wongkham, et al. [206], evaluate updateable learned indexes. Five one-dimensional learned indexes have been included in this study: ALEX, PGM, LIPP [207], XIndex [190], and FINEdex [114]. Moreover, this study proposes to extend ALEX and LIPP to support concurrency as ALEX+ and LIPP+. In [188], a testbed has been developed to facilitate the design and testing of existing and upcoming learned index structures. It also presents the design choices of key components in learned indexes (e.g., insertion, concurrency, bulk loading). The experiments cover

eight one-dimensional learned indexes. In [105], it has been shown how to extend four in-memory updatable one-dimensional learned indexes (e.g., Fitting-tree [65], ALEX, PGM, and LIPP) in a disk-based setting. An extensive experimental evaluation is presented for the proposed extensions. In [67], several one-dimensional updateable learned indexes (e.g., RMI, RS, ALEX, XIndex, FITing-tree, and PGM) have been evaluated along four criteria: approximation algorithm, index structure, insertion strategy, and model re-training strategy. Moreover, it also provides several guidelines for designing learned indexes. The concept of using ML models as a replacement for a hash function has been introduced initially as an HMI [102]. In [170] and [171], extensive experimental studies have been conducted to analyze the performance of learned hash index structures. The contribution in these studies is in providing insights for choosing a learned hash index versus a traditional hash index.

8.2 The Multi-dimensional Case

There are no comprehensive benchmarks for learned multi-dimensional indexes. Thus, it is still an important and open area of research.

9 Towards the Integration of Learned Indexes into Practical Systems

A few studies have been conducted on integrating learned index structures into practical systems. Although a few one-dimensional learned indexes have been successfully integrated into practical database/storage engines, similar efforts are still in progress in the context of learned multi-dimensional indexes. In this section, we discuss the studies that take a step forward towards the integration of learned indexes into practical systems.

9.1 The One-dimensional Case

In Google-index [5], a learned index has been integrated into a distributed, disk-based database system: Google's Bigtable [33]. The integrated learned index has demonstrated significant improvements in read latency and memory footprint. In BOURBON [42], the proposed ML-enhanced LSM-tree has been incorporated into a production-quality system, and has achieved high performance compared with its traditional counterpart. In SA-LSM [227], the proposed ML-enhanced LSM-tree has been incorporated into a commercial-strength LSM-tree storage engine, namely, X-Engine. Sieve [192] has been integrated in a distributed SQL query engine, namely Presto [179]. Hybrid-LR [162] has been implemented in PostgreSQL [186]. PGM [62] has been adopted in the column-oriented storage system Manticore [4].

9.2 The Multi-dimensional Case

In ELSI [122], a system for efficiently building and rebuilding learned multi-dimensional indexes has been proposed. ELSI supports any learned spatial index that projects the multi-dimensional points into one dimension (i.e., termed as projected space in this survey) to impose an ordering and process queries using the predict-and-scan method. Two index-building methods have been proposed based on space partitioning and RL. Moreover, ELSI has been combined with four multi-dimensional indexes: ZM-index [197], ML-index [45], RSMI [160], and LISA [116]. However, the integration of ELSI into a commercial database system (e.g., PostgreSQL) has been mentioned as future work. In SageDB [46], the ideas of the Qd-tree [214] have been implemented for creating the data layout inside a practical system. In PLATON [213], the proposed ML-enhanced R-tree has been implemented on top of a real-world spatial library: libspatialindex [1], and a practical database system: PostgreSQL 14.3 [3], with the PostGIS[2] extension.

10 Open Challenges and Future Research Directions

We highlight open research challenges and future directions in the context of learned multi-dimensional indexes.

Table 4. A Summary of ML Techniques for Learned Multi-dimensional Indexes

ML Techniques	Index
RS, Random Forest Regression	LSTI [50]
Lattice Regression Model	LISA [116]
Ploynomial Regression	LIMS [191]
Linear Model	SageDB-LMI [101], Tsunami [49], COAX [75], GLIN_ALEX [196]
Density Model	Z-index [153]
Piecewise Linear Model, RMI	FLOOD [148]
Polynomial Function	PolyFit [120]
Linear Interpolation	IF-X [80]
Spatial Interpolation Function	SPRIG [225], SPRIG+ [226]
Cumulative Histogram	Period-index* [20]
K-means Clustering	LMI-2 [183], FHSIE [187]
Density based Clustering	RW-tree [52]
RFDE Model	WAZI [152]
Random Forest, Decision Tree	LMSFC [66], "AI+R"-tree [8]
RS	CaseSpatial [154], Spatial-LS [155], Distance-bounded [219]
Gradient Boosting	LPBF [234], PA-LBF [221]
Neural Network, Linear Model	ZM-index [197]
Neural Network	LearnedKD [216], LMI-1 [13], ML-enhanced [93], CompressedBF [44], LearnedBF [136], ML-index [45], HM-index [199], RSMI [160], SLBRIN [198]
RL	QD-tree [214], MTO [47], RLR-tree [72], Waffle [38], BMT [112], WISK [181], ACR-tree [83], PLATON [213]

10.1 Total Ordering and Error Bound

In the context of one-dimensional learned indexes, in most cases, the ML models are trained with the assumption that the underlying data is sorted with a total order. However, prediction-based ML models inherently do not provide any guarantee of accuracy when applied to unseen test data. As a result, if the ML models make an error during prediction, it needs to be corrected by a local search based on a predefined error bound. Notice that this error bound can be easily achieved in the case of one-dimensional data because we can sort the data (i.e., total ordering). However, this is not the case for multi-dimensional data because there is no total ordering in the multi-dimensional space. As a result, a class of learned multi-dimensional indexes projects the data into one-dimensional space to impose an ordering. Various space-filling curves have been used in this context, each with different advantages and limitations. On the other hand, the class of learned multi-dimensional indexes in native space does not employ a projection function. However, some learned multi-dimensional indexes might select one of the dimensions to impose an order in native space. Each of the learned multi-dimensional indexes in native space, for exact query processing, has proposed a different mechanism for error correction. However, providing an error bound is challenging for the latter class of indexes, and further research addressing this aspect is needed. Notice that learned one-dimensional indexes have been designed particularly for string keys [184, 201]. As a result, it is an interesting problem to investigate the challenges related to error bounds in the context of strings in the multi-dimensional space.

10.2 Choice of ML Models

A summary of the ML techniques used for learned multi-dimensional indexes is presented in Table 4. Due to recent advances in the area of machine learning, there are numerous types of ML models available with various levels of model complexity. In general, a complex model is expected

to learn the underlying data distribution better than a simpler model. However, in the context of learned indexes, it is normally suggested to adopt a simple ML model (e.g., linear regression, decision tree) whenever possible. The main reason behind this suggestion is performance (e.g., less training time, smaller model size, low model prediction latency). Notice that traditional database indexes (e.g., B^+ tree, R-tree) are highly optimized data structures. As a result, a learned index should avoid using complex ML models so that the model building time and prediction latency do not become bottlenecks to achieving low index construction time and high query processing performance, respectively. However, it has also been observed that a customized ML model yields better performance than basic ML models. For example, in [54], a tailored regression model has been proposed to achieve better performance. Moreover, there is a growing interest in designing and using deep neural network-based ML models [10, 11, 57] in the context of learned indexes. As a result, the choice of ML models needs to be further investigated for designing learned multi-dimensional indexes.

10.3 Model Training and Re-training

Training ML models is one of the most important parts of constructing learned indexes. Most of the proposed learned indexes try to minimize the model training time as much as possible. Moreover, changes in the underlying input data/query distribution should be detected as soon as possible, and a model re-training process should be triggered when necessary. Although these challenges are addressed in many of the proposed indexes, there is still ample room for future research in this area. For example, exploring the concept of Machine Unlearning [104] in the context of mutable learned indexes is an interesting research direction.

10.4 Supporting Dynamic Inserts/Updates

The initial learned indexes only considered read-only workloads in a static scenario. After that, new methods have been proposed to support insert/update operations. However, supporting inserts/updates comes with the cost of periodic re-organization (e.g., re-training) of the learned index structures or incorporating mechanisms that require additional space (e.g., a GA). In this survey, we have mainly observed two main strategies to support dynamic inserts: In-place and Delta buffer. Although these challenges are addressed in the class of mutable multi-dimensional learned indexes, further research is needed to investigate the advantages and disadvantages of each insertion strategy.

10.5 Concurrency

In the context of designing learned indexes, for most cases, supporting concurrency has come as an afterthought. Only a few proposed methods (marked with an * symbol in the taxonomy 2) discuss the issue of concurrency and propose techniques to support concurrency natively. As a result, future research in this area should treat the issue of concurrency as a first-class citizen while designing or implementing learned indexes.

10.6 Index Compression

Both one- and multi-dimensional indexes have demonstrated significant benefits in terms of reduced storage requirements. Particularly, the algorithmic principles of learned indexes have been adopted in the context of data compression on integers [28, 60], strings [27, 57], and time-series [129]. Moreover, in the context of inverted indexes [233], the benefits of index compression by incorporating a learned one-dimensional index into an inverted index structure have been studied [150]. Additionally, the advantages of a learned multi-dimensional bloom filter for index compression are studied in CompressLBF [44]. However, exploring the potential benefits of index

compression using learned multi-dimensional indexes in the context of other index types is an interesting direction for future research.

10.7 Security

The issue of security in the context of learned indexes is also little explored. The impact of poisoning attacks has been discussed in a recent study [99]. It has been shown that "every injection into the training dataset has a cascading impact on multiple data values." This attack can significantly reduce the performance of learned indexes. As a result, robust learned indexing methods should be designed to tackle issues related to security. Particularly, indexes, e.g., PGM [62], that are designed with a worst-case guarantee are expected to perform well in the presence of adversarial queries. Exploring this direction in the context of multi-dimensional learned indexes is an open research topic.

10.8 Benchmarking

Several studies have been performed to benchmark the performance of one-dimensional learned indexes. However, a comprehensive benchmark on learned multi-dimensional indexes is still missing. There are many open-sourced multi-dimensional datasets available in various repositories (e.g., UCR-STAR [69]). However, collecting real query workloads for multi-dimensional and spatial data is challenging. Thus, many studies synthetic query workloads. A benchmarking study on multi-dimensional learned indexes with real data and query workloads will likely fill this gap.

10.9 Theoretical Analysis

There are a few theoretical studies [34, 58, 59, 61, 220] that mathematically analyze the reasons behind the performance gain of learned one-dimensional indexes over traditional indexes. In [58, 59], it has been shown that learned one-dimensional indexes are provably better than traditional indexes. Notice that similar results have been achieved in both [34, 61]. Moreover, it has been proven that, under certain assumptions, learned one-dimensional indexes can answer queries in $O(\log \log n)$ expected query time [220]. Furthermore, for the class of learned one-dimensional indexes adopting piece-wise linear segments, the theoretical analysis on the choice of the parameter ϵ has been presented in [34]. As a result, theoretical analysis of the various components of learned multi-dimensional indexes is needed to better understand the benefits and limitations of existing techniques.

10.10 Leveraging GPUs for Learned Indexes

The benefit of natively implementing a learned one-dimensional index on a GPU has been presented in [231]. Particularly, the PGM index has been implemented on a GPU so that the index can exploit the high concurrency of the GPU. During query processing, GPU-PGM achieves an order of magnitude performance gain over the CPU-PGM. Similar investigation in the context of learned multi-dimensional indexes is an interesting direction for future research.

11 Conclusion

In this survey, we provide an up-to-date coverage of the state-of-the-art in learned multi-dimensional indexes. We have introduced a taxonomy to categorize both one- and multi-dimensional indexes using the following criteria: (i) Immutable vs. mutable learned indexes, (ii) Mutable fixed data layout vs. mutable dynamic data layout learned indexes, (iii) One-dimensional vs. multi-dimensional learned indexes, (iv) Pure learned indexes vs. hybrid learned indexes, (v) In-place vs. delta-buffer insertion strategy for learned indexes, and (vi) Projected vs. native space learned multi-dimensional indexes. We summarize the core ideas of 43 learned multi-dimensional indexes, and highlight the similarities/differences of more than 60 learned one-dimensional indexes. Furthermore, we

have listed the supported query types and the underlying ML techniques for each of the learned multi-dimensional indexes. We include a timeline diagram that shows the evolution of both one- and multi-dimensional learned index structures. Finally, we have discussed several open challenges and future research opportunities in the area of learned multi-dimensional indexes.

References

[1] Accessed in 2023. libspatialindex 1.9.3. Retrieved 22 Dec. 2023 from https://libspatialindex.org/en/latest/

[2] Accessed in 2023. PostGIS. Retrieved 22 Dec. 2023 from https://postgis.net/

[3] Accessed in 2023. PostgreSQL 14.3. Retrieved 22 Dec. 2023 from https://www.postgresql.org/

[4] Accessed in 2024. Manticore. Retrieved 1 Nov. 2024 from https://manticoresearch.com/

[5] Hussam Abu-Libdeh, Deniz Altınbüken, Alex Beutel, Ed H. Chi, Lyric Doshi, Tim Kraska, Andy Ly, Christopher Olston, et al. 2020. Learned indexes for a google-scale disk-based database. arXiv:2012.12501. Retrieved from https://arxiv.org/abs/2012.12501

[6] Charu C. Aggarwal. 2015. Data classification. In *Proceedings of the Data Mining*. Springer, 285–344.

[7] Charu C. Aggarwal. 2018. Neural networks and deep learning. *Springer* 10, 978 (2018), 3.

[8] Abdullah Al-Mamun, Ch Md Haider, Jianguo Wang, and Walid Aref. 2022. The "AI+ R"-tree: An instance-optimized r-tree. In *Proceedings of the IEEE MDM*. 9–18.

[9] Abdullah Al-Mamun, Hao Wu, and Walid G. Aref. 2020. A tutorial on learned multi-dimensional indexes. In *Proceedings of the ACM SIGSPATIAL GIS*. 1–4.

[10] Domenico Amato, Giosué Lo Bosco, and Raffaele Giancarlo. 2022. On the suitability of neural networks as building blocks for the design of efficient learned indexes. In *Proceedings of the International Conference on Engineering Applications of Neural Networks*. Springer, 115–127.

[11] Domenico Amato, Giosué Lo Bosco, and Raffaele Giancarlo. 2023. Neural networks as building blocks for the design of efficient learned indexes. *Neural Computing and Applications* 35, 29 (2023), 21399–21414.

[12] Mikkel Andersen and Pinar Tözün. 2022. Micro-architectural analysis of a learned index. In *Proceedings of the aiDB*. 1–12.

[13] Matej Antol, Jaroslav Ol'ha, Terézia Slanináková, and Vlastislav Dohnal. 2021. Learned metric index-proposition of learned indexing for unstructured data. *Information Systems* 100 (2021), 101774.

[14] Walid Aref, Daniel Barbará, and Padma Vallabhaneni. 1995. The handwritten trie: Indexing electronic ink. *SIGMOD* 24, 2 (1995), 151–162.

[15] Walid G. Aref. 2009. *Electronic Ink Indexing*. Springer US, Boston, MA. DOI: https://doi.org/10.1007/978-0-387-39940-9_143

[16] Manos Athanassoulis, Stratos Idreos, and Dennis Shasha. 2023. Data structures for data-intensive applications: Tradeoffs and design guidelines. *Foundations and Trends® in Databases* 13, 1-2 (2023), 1–168.

[17] G Phanendra Babu. 1997. Self-organizing neural networks for spatial data. *Pattern Recognition Letters* 18, 2 (1997), 133–142.

[18] Rudolf Bayer and Edward McCreight. 1970. Organization and maintenance of large ordered indices. In *Proceedings of the ACM SIGFIDET (Now SIGMOD)*. 107–141.

[19] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. 322–331.

[20] Andreas Behrend, Anton Dignös, Johann Gamper, Philip Schmiegelt, Hannes Voigt, Matthias Rottmann, and Karsten Kahl. 2019. Period index: A learned 2D hash index for range and duration queries. In *Proceedings of the SSTD*. 100–109.

[21] Michael A. Bender and Haodong Hu. 2007. An adaptive packed-memory array. *ACM Transactions on Database Systems* 32, 4 (2007), 26–es.

[22] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM* 18, 9 (1975), 509–517.

[23] Jon Louis Bentley and Jerome H. Friedman. 1979. Data structures for range searching. *ACM Computing Surveys* 11, 4 (1979), 397–409.

[24] Arindam Bhattacharya, Srikanta Bedathur, and Amitabha Bagchi. 2020. Adaptive learned bloom filters under incremental workloads. In *Proceedings of the 7th ACM IKDD CoDS and 25th COMAD*. 107–115.

[25] Laurent Bindschaedler, Andreas Kipf, Tim Kraska, Ryan Marcus, and Umar Farooq Minhas. 2021. Towards a benchmark for learned systems. In *Proceedings of the 2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 127–133.

[26] Burton H. Bloom. 1970. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM* 13, 7 (1970), 422–426.

[27] Antonio Boffa, Paolo Ferragina, Francesco Tosoni, and Giorgio Vinciguerra. 2024. CoCo-trie: Data-aware compression and indexing of strings. *Information Systems* 120 (2024), 102316.
[28] Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. 2022. A learned approach to design compressed rank/select data structures. *ACM Transactions on Algorithms* 18, 3 (2022), 1–28.
[29] Léon Bottou. 1998. Online algorithms and stochastic approximations. *Online Learning in Neural Networks* (1998).
[30] Leo Breiman. 2001. Random forests. *Machine Learning* 45 (2001), 5–32.
[31] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (2012), 1–43.
[32] Chengliang Chai, Jiayi Wang, Yuyu Luo, Zeping Niu, and Guoliang Li. 2022. Data management for machine learning: A survey. *IEEE Transactions on Knowledge and Data Engineering* 35, 5 (2022), 4646–4667.
[33] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems* 26, 2 (2008), 1–26.
[34] Daoyuan Chen, Wuchao Li, Yaliang Li, Bolin Ding, Kai Zeng, Defu Lian, and Jingren Zhou. 2022. Learned index with dynamic ϵ. In *Proceedings of the ICLR*.
[35] Lianhua Chi and Xingquan Zhu. 2017. Hashing techniques: A survey and taxonomy. *ACM Computing Surveys* 50, 1 (2017), 1–36.
[36] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv:1406.1078. Retrieved from https://arxiv.org/abs/1406.1078
[37] Supawit Chockchowwat, Wenjie Liu, and Yongjoo Park. 2023. Airindex: Versatile index tuning through data and storage. *SIGMOD* 1, 3 (2023), 1–26.
[38] Dalsu Choi, Hyunsik Yoon, Hyubjin Lee, and Yon Dohn Chung. 2022. Waffle: In-memory grid index for moving objects with reinforcement learning-based configuration tuning system. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2375–2388.
[39] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys* 11, 2 (1979), 121–137.
[40] Andrew Crotty. 2021. Hist-tree: Those who ignore it are doomed to learn. In *Proceedings of the CIDR*.
[41] Lixiao Cui, Kedi Yang, Yusen Li, Gang Wang, and Xiaoguang Liu. 2023. DiffLex: A high-performance, memory-efficient and NUMA-aware learned index using differentiated management. In *Proceedings of the 52nd International Conference on Parallel Processing*. 62–71.
[42] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From {WiscKey} to bourbon: A learned index for {Log-Structured} merge trees. In *Proceedings of the OSDI*. 155–171.
[43] Zhenwei Dai and Anshumali Shrivastava. 2019. Adaptive learned bloom filter (Ada-BF): Efficient utilization of the classifier. arXiv:1910.09131. Retrieved from https://arxiv.org/abs/1910.09131
[44] Angjela Davitkova, Damjan Gjurovski, and Sebastian Michel. 2021. Compressing (multidimensional) learned bloom filters. In *Proceedings of the Workshop on Databases and AI*.
[45] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A multidimensional, learned index for point, range, and nearest-neighbor queries. In *Proceedings of the EDBT*. 407–410.
[46] Jialin Ding, Ryan Marcus, Andreas Kipf, Vikram Nathan, Aniruddha Nrusimha, Kapil Vaidya, Alexander van Renen, and Tim Kraska. 2022. SageDB: An instance-optimized data analytics system. *PVLDB* 15, 13 (2022), 4062–4078.
[47] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-optimized data layouts for cloud analytics workloads. In *Proceedings of the SIGMOD Conference*. ACM, 418–431.
[48] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: An updatable adaptive learned index. In *Proceedings of the SIGMOD*. 969–984.
[49] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *PVLDB* 14, 2 (2020), 74–86.
[50] Xiaofeng Ding, Yinting Zheng, Zuan Wang, Kim-Kwang Raymond Choo, and Hai Jin. 2023. A learned spatial textual index for efficient keyword queries. *Journal of Intelligent Information Systems* 60, 3 (2023), 803–827.
[51] Yuquan Ding, Xujian Zhao, and Peiquan Jin. 2022. An error-bounded space-efficient hybrid learned index with high lookup performance. In *Proceedings of the International Conference on Database and Expert Systems Applications*. Springer, 216–228.
[52] Haowen Dong, Chengliang Chai, Yuyu Luo, Jiabin Liu, Jianhua Feng, and Chaoqun Zhan. 2022. RW-Tree: A learned workload-aware framework for r-tree construction. In *Proceedings of the 2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2073–2085.

[53] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2021. New trends in high-d vector similarity search: Al-driven, progressive, and distributed. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3198–3201.
[54] Martin Eppert, Philipp Fent, and Thomas Neumann. 2021. A tailored regression for learned indexes: Logarithmic error regression. In *Proceedings of the 4th Workshop in Exploiting AI Techniques for Data Management*. 9–15.
[55] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking* 8, 3 (2000), 281–293.
[56] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, and Amr El Abbadi. 2000. Vector approximation based indexing for non-uniform high dimensional datasets. In *Proceedings of the CIKM*. ACM, 202–209.
[57] Paolo Ferragina, Marco Frasca, Giosuè Cataldo Marinò, and Giorgio Vinciguerra. 2023. On nonlinear learned string indexing. *IEEE Access* 11 (2023), 74021–74034.
[58] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why are learned indexes so effective?. In *Proceedings of the ICML*. PMLR, 3123–3132.
[59] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2021. On the performance of learned data structures. *Theoretical Computer Science* 871 (2021), 107–120.
[60] Paolo Ferragina, Giovanni Manzini, and Giorgio Vinciguerra. 2022. Compressing and querying integer dictionaries under linearities and repetitions. *IEEE Access* 10 (2022), 118831–118848.
[61] Paolo Ferragina and Giorgio Vinciguerra. 2020. Learned data structures. In *Proceedings of the Recent Trends in Learning From Data*. Springer, 5–41.
[62] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175.
[63] Edward Fredkin. 1960. Trie memory. *Communications of the ACM* 3, 9 (1960), 490–499.
[64] Volker Gaede and Oliver Günther. 1998. Multidimensional access methods. *ACM Computing Surveys* 30, 2 (1998), 170–231.
[65] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A data-aware index structure. In *Proceedings of the SIGMOD Conference*. ACM, 1189–1206.
[66] Jian Gao, Xin Cao, Xin Yao, Gong Zhang, and Wei Wang. 2023. LMSFC: A novel multidimensional index based on learned monotonic space filling curves. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2605–2617.
[67] Jiake Ge, Boyu Shi, Yanfeng Chai, Yuanhui Luo, Yunda Guo, Yinxuan He, and Yunpeng Chai. 2023. Cutting learned index into pieces: An in-depth inquiry into updatable learned indexes. In *Proceedings of the 2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 315–327.
[68] Jiake Ge, Huanchen Zhang, Boyu Shi, Yuanhui Luo, Yunda Guo, Yunpeng Chai, Yuxing Chen, and Anqun Pan. 2023. SALI: A scalable adaptive learned index framework based on probability models. arXiv:2308.15012. Retrieved from https://arxiv.org/abs/2308.15012
[69] Saheli Ghosh, Tin Vu, Mehrad Amin Eskandari, and Ahmed Eldawy. 2019. UCR-STAR: The UCR spatio-temporal active repository. *SIGSPATIAL Special* 11, 2 (2019), 34–40.
[70] Goetz Graefe. 2006. B-tree indexes, interpolation search, and skew. In *Proceedings of the Workshop on Data Management on New Hardware*. 5–es.
[71] Ivo Grondman, Lucian Busoniu, Gabriel A. D. Lopes, and Robert Babuska. 2012. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 6 (2012), 1291–1307.
[72] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2023. The RLR-tree: A reinforcement learning based r-tree for spatial data. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[73] Na Guo, Yaqi Wang, Haonan Jiang, Xiufeng Xia, and Yu Gu. 2022. TALI: An update-distribution-aware learned index for social media data. *Mathematics* 10, 23 (2022), 4507.
[74] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the SIGMOD*. 47–57.
[75] Ali Hadian, Behzad Ghaffari, Taiyi Wang, and Thomas Heinis. 2023. COAX: Correlation-aware indexing. In *Proceedings of the 2023 IEEE ICDEW*. 55–59.
[76] Ali Hadian and Thomas Heinis. 2019. Considerations for handling updates in learned index structures. In *Proceedings of the AIDM*. ACM.
[77] Ali Hadian and Thomas Heinis. 2019. Interpolation-friendly B-trees: Bridging the gap between algorithmic and learnedindexes. (2019).
[78] Ali Hadian and Thomas Heinis. 2020. Madex: Learning-augmented algorithmic index structures. In *Proceedings of the aiDB*.
[79] Ali Hadian and Thomas Heinis. 2021. Shift-Table: A low-latency learned index for range queries using model correction. arXiv:2101.10457. Retrieved from https://arxiv.org/abs/2101.10457

[80] Ali Hadian, Ankit Kumar, and Thomas Heinis. 2020. Hands-off model integration in spatial index structures. In *Proceedings of the aiDB*.
[81] Francisco Herrera, Francisco Charte, Antonio Rivera, and María Del Jesus. 2016. Multilabel classification. In *Proceedings of the Multilabel Classification*. Springer, 17–31.
[82] David Hilbert. 1891. Ueber die stetige abbildung einer linie auf ein Flächenstück. *Mathematische Annalen* 38 (1891), 459–460.
[83] Shuai Huang, Yong Wang, and Guoliang Li. 2023. ACR-Tree: Constructing r-trees using deep reinforcement learning. In *Proceedings of the DASFAA*. 80–96.
[84] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5*. Springer, 507–523.
[85] Stratos Idreos and Tim Kraska. 2019. From auto-tuning one size fits all to self-designed and learned data-intensive systems. In *Proceedings of the SIGMOD*. 2054–2059.
[86] Ihab F Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: Automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. 647–658.
[87] Hosagrahar 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 Transactions on Database Systems* 30, 2 (2005), 364–397.
[88] Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. 1999. Data clustering: A review. *ACM Computing Surveys* 31, 3 (1999), 264–323.
[89] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, and Jonathan Taylor. 2023. Linear regression. In *Proceedings of An Introduction to Statistical Learning: With Applications in Python*. Springer, 69–134.
[90] Matthias Jasny, Tobias Ziegler, Tim Kraska, Uwe Roehm, and Carsten Binnig. 2020. DB4ML-an in-memory database kernel with machine learning support. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 159–173.
[91] Hui Jin and H. V. Jagadish. 2002. Indexing hidden markov models for music retrieval. In *Proceedings of the ISMIR*.
[92] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4 (1996), 237–285.
[93] Rong Kang, Wentao Wu, Chen Wang, Ce Zhang, and Jianmin Wang. 2021. The case for ML-enhanced high-dimensional indexes. In *Proceedings of the 3rd International Workshop on Applied AI for Database Systems and Applications*.
[94] Andreas Kipf, Dominik Horn, Pascal Pfeil, Ryan Marcus, and Tim Kraska. 2022. Lsi: A learned secondary index structure. In *Proceedings of the aiDM*. 1–5.
[95] Andreas Kipf, Harald Lang, V. N. Pandey, Raul Alexandru Persa, Christoph Anneser, Eleni Tzirita Zacharatou, Harish Doraiswamy, Peter A Boncz, Thomas Neumann, and Alfons Kemper. 2020. Adaptive main-memory indexing for high-performance point-polygon joins. (2020).
[96] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A benchmark for learned indexes. arXiv:1911.13014. Retrieved from https://arxiv.org/abs/1911.13014
[97] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A single-pass learned index. In *Proceedings of the AIDB*. 1–5.
[98] Teuvo Kohonen. 1990. The self-organizing map. *Proceedings of the IEEE* 78, 9 (1990), 1464–1480.
[99] Evgenios M. Kornaropoulos, Silei Ren, and Roberto Tamassia. 2022. The price of tailoring the index to your data: Poisoning attacks on learned index structures. In *Proceedings of the 2022 International Conference on Management of Data*. 1331–1344.
[100] Tim Kraska. 2021. Towards instance-optimized data systems. *Proceedings of the VLDB Endowment* 14, 12 (2021).
[101] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. Sagedb: A learned database system. (2019).
[102] Tim Kraska, Alex Beutel, Ed Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the ACM SIGMOD*. 489–504.
[103] Tim Kraska, Umar Farooq Minhas, Thomas Neumann, Olga Papaemmanouil, Jignesh M Patel, Christopher Ré, and Michael Stonebraker. 2021. ML-in-databases: Assessment and prognosis. *IEEE Data Engineering Bulletin* 44, 1 (2021), 3–10.
[104] Meghdad Kurmanji, Eleni Triantafillou, and Peter Triantafillou. 2024. Machine unlearning in learned databases: An experimental analysis. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.
[105] Hai Lan, Zhifeng Bao, J. Shane Culpepper, and Renata Borovica-Gajic. 2023. Updatable learned indexes meet disk-resident DBMS-from evaluations to design choices. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–22.

[106] Ken CK Lee, Baihua Zheng, Huajing Li, and Wang-Chien Lee. 2007. Approaching the skyline in Z order. In *Proceedings of the VLDB*. 279–290.
[107] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the ICDE*. 38–49.
[108] Scott Leutenegger, Mario Lopez, and Jeffrey Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of the ICDE*. 497–506.
[109] Guoliang Li and Xuanhe Zhou. 2022. Machine learning for data management: A system view. In *Proceedings of the ICDE*. 3198–3201.
[110] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. AI meets database: AI4DB and DB4AI. In *Proceedings of the SIGMOD*. 2859–2866.
[111] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. Machine learning for databases. In *Proceedings of the 1st International Conference on AI-ML Systems*. 1–2.
[112] Jiangneng Li, Zheng Wang, Gao Cong, Cheng Long, Han Mao Kiah, and Bin Cui. 2023. Towards designing and learning piecewise space-filling curves. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2158–2171.
[113] Mingxin Li, Hancheng Wang, Haipeng Dai, Meng Li, Rong Gu, Feng Chen, Zhiyuan Chen, Shuaituan Li, Qizhi Liu, and Guihai Chen. 2024. A survey of multi-dimensional indexes: Past and future trends. *IEEE Transactions on Knowledge and Data Engineering* 36, 8 (2024), 3635–3655.
[114] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A fine-grained learned index scheme for scalable and concurrent memory systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.
[115] Pengfei Li, Yu Hua, Pengfei Zuo, and Jingnan Jia. 2019. A scalable learned index scheme in storage systems. arXiv:1905.06256. Retrieved from https://arxiv.org/abs/1905.06256
[116] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A learned index structure for spatial data. *SIGMOD* (2020), 2119–2133.
[117] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A distribution-driven learned index. In *Proceedings of the VLDB Endowment* 16, 9 (2023), 2212–2224.
[118] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. Mlog: Towards declarative in-database machine learning. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1933–1936.
[119] Xin Li, Jingdong Li, and Xiaoling Wang. 2019. ASLM: Adaptive single layer model for learned index. In *Proceedings of the DASFAA*. Springer, 80–95.
[120] Zhe Li, Tsz Nam Chan, Man Lung Yiu, and Christian Jensen. 2021. PolyFit: Polynomial-based indexing approach for fast approximate range aggregate queries. In *Proceedings of the EDBT*. OpenProceedings. org, 241–252.
[121] Yuming Lin, Zhengguo Huang, and You Li. 2023. Learning hash index based on a shallow autoencoder. *Applied Intelligence* 53, 12 (2023), 14999–15010.
[122] Guanli Liu, Jianzhong Qi, Christian S. Jensen, James Bailey, and Lars Kulik. 2023. Efficiently learning spatial indices. In *Proceedings of the ICDE*. 1572–1584.
[123] Guanli Liu, Jianzhong Qi, Lars Kulik, Kazuya Soga, Renata Borovica-Gajic, and Benjamin I. P. Rubinstein. 2023. Efficient index learning via model reuse and fine-tuning. In *Proceedings of the IEEE ICDEW*. 60–66.
[124] Jiajun Liu, Haoran Li, Yong Gao, Hao Yu, and Dan Jiang. 2014. A geohash-based index for spatial data management in distributed memory. In *Proceedings of the 2014 22Nd International Conference on Geoinformatics*. IEEE, 1–4.
[125] Li Liu, Chunhua Li, Zhou Zhang, Yuhan Liu, Ke Zhou, and Ji Zhang. 2022. A data-aware learned index scheme for efficient writes. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.
[126] Qiyu Liu, Yanyan Shen, and Lei Chen. 2022. HAP: An efficient hamming space index based on augmented pigeonhole principle. In *Proceedings of the SIGMOD*. 917–930.
[127] Qiyu Liu, Libin Zheng, Yanyan Shen, and Lei Chen. 2020. Stable learned bloom filters for data streams. *PVLDB* 13, 12 (2020), 2355–2367.
[128] Yu Liu, Hua Wang, Ke Zhou, ChunHua Li, and Rengeng Wu. 2022. A survey on AI for storage. *CCF THPC* 4, 3 (2022), 233–264.
[129] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight compression via learning serial correlations. *SIGMOD* 2, 1 (2024), 1–28.
[130] A. Llavesh, Utku Sirin, R. West, and A. Ailamaki. 2019. Accelerating b+ tree search by using simple machine learning techniques. In *Proceedings of the 1st International Workshop on Applied AI for Database Systems and Applications*.
[131] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A high-performance learned index on persistent memory. *Proceedings of the VLDB Endowment* 15, 3 (2021), 597–610.
[132] Kai Lu, Nannan Zhao, Jiguang Wan, Changhong Fei, Wei Zhao, and Tongliang Deng. 2021. TridentKV: A read-optimized LSM-tree based KV store via adaptive indexing and space-efficient partitioning. *IEEE Transactions on Parallel and Distributed Systems* 33, 8 (2021), 1953–1966.

[133] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage* 13, 1 (2017), 1–28.
[134] Chen Luo and Michael J. Carey. 2020. LSM-based storage techniques: A survey. *The VLDB Journal* 29, 1 (2020), 393–418.
[135] Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maoliniyazi. 2022. Film: A fully learned index for larger-than-memory databases. *Proceedings of the VLDB Endowment* 16, 3 (2022), 561–573.
[136] Stephen Macke, Alex Beutel, Tim Kraska, Maheswaran Sathiamoorthy, Derek Zhiyuan Cheng, and Ed H. Chi. 2018. Lifting the curse of multidimensional data with learned existence indexes. In *Proceedings of the Workshop on ML for Systems at NeurIPS*. 1–6.
[137] Marcel Maltry and Jens Dittrich. 2022. A critical analysis of recursive model indexes. *Proceedings of the VLDB Endowment* 15, 5 (2022), 1079–1091.
[138] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *PVLDB* 14, 1 (2020), 1–13.
[139] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1705–1718.
[140] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and optimizing learned index structures. *SIGMOD* (2020), 2789–2792.
[141] Mayank Mishra and Rekha Singhal. 2021. RUSLI: Real-time updatable spline learned index. In *Proceedings of the AIDM*.
[142] Lubos Mitas and Helena Mitasova. 1999. Spatial interpolation. *Geographical Information Systems: Principles, Techniques, Mgmt and Appl.* 1, 2 (1999).
[143] Michael Mitzenmacher. 2018. A model for learned bloom filters, and optimizing by sandwiching. In *Proceedings of the NIPS'18*. 462–471.
[144] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.
[145] Mohamed F. Mokbel, Walid G. Aref, and Ibrahim Kamel. 2003. Analysis of multi-dimensional space-filling curves. *GeoInformatica* 7 (2003), 179–209.
[146] Robert Morris. 1968. Scatter storage techniques. *Communications of the ACM* 11, 1 (1968), 38–44.
[147] Guy M. Morton. 1966. A computer oriented geodetic data base and a new technique in file sequencing. (1966).
[148] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning multi-dimensional indexes. In *Proceedings of the ACM SIGMOD*. 985–1000.
[149] Jürg Nievergelt, Hans Hinterberger, and Kenneth Sevcik. 1984. The grid file: An adaptable, symmetric multikey file structure. *ACM TODS* 9, 1 (1984), 38–71.
[150] Harrie Oosterhuis, J. Shane Culpepper, and Maarten de Rijke. 2018. The potential of learned index structures for index compression. In *Proceedings of the 23rd Australasian Document Computing Symposium*. 1–4.
[151] Jack A. Orenstein and Tim H. Merrett. 1984. A class of data structures for associative searching. In *Proceedings of the ACM PODS*. 181–190.
[152] Sachith Pai, Michael Mathioudakis, and Yanhao Wang. 2023. Workload-aware and learned z-indexes. arXiv:2310.04268. Retrieved from https://arxiv.org/abs/2310.04268
[153] Sachith Gopalakrishna Pai, Michael Mathioudakis, and Yanhao Wang. 2022. Towards an instance-optimal z-index. *AIDB@VLDB* (2022).
[154] Varun Pandey, Alexander van Renen, Andreas Kipf, Ibrahim Sabek, Jialin Ding, and Alfons Kemper. 2020. The case for learned spatial indexes. arXiv:2008.10349. Retrieved from https://arxiv.org/abs/2008.10349
[155] Varun Pandey, Alexander van Renen, Eleni Tzirita Zacharatou, Andreas Kipf, Ibrahim Sabek, Jialin Ding, Volker Markl, and Alfons Kemper. 2023. Enhancing in-memory spatial indexing with learned search. arXiv:2309.06354. Retrieved from https://arxiv.org/abs/2309.06354
[156] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-driving database management systems. In *Proceedings of the CIDR*. 1.
[157] Peano. 1890. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen* 36 (1890), 157–160.
[158] William Pugh. 1990. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM* 33, 6 (1990), 668–676.
[159] Martin L Puterman. 1990. Markov decision processes. *Handbooks in Operations Research and Management Science* 2 (1990), 331–434.
[160] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively learning spatial indices. *PVLDB* 13, 12 (2020), 2341–2354.

[161] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2018. Theoretically optimal and empirically efficient r-trees with strong parallelizability. *PVLDB* 11, 5 (2018), 621–634.
[162] Wenwen Qu, Xiaoling Wang, Jingdong Li, and Xin Li. 2019. Hybrid indexes by exploring traditional b-tree and linear regression. In *Proceedings of the International Conference on Web Information Systems and Applications*. Springer, 601–613.
[163] J. Ross Quinlan. 1986. Induction of decision trees. *Machine Learning* 1, 1 (1986), 81–106.
[164] Lawrence Rabiner and Biinghwang Juang. 1986. An introduction to hidden markov models. *IEEE ASSP Magazine* 3, 1 (1986), 4–16.
[165] Jack Rae, Sergey Bartunov, and Timothy Lillicrap. 2019. Meta-learning neural bloom filters. In *Proceedings of the ICLR*. PMLR, 5271–5280.
[166] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. 2003. *Database Management Systems*. McGraw-Hill New York.
[167] Ibrahim Sabek and Mohamed F. Mokbel. 2019. Machine learning meets big spatial data. *PVLDB* 12, 12 (2019), 1982–1985.
[168] Ibrahim Sabek and Mohamed F. Mokbel. 2020. Machine learning meets big spatial data. In *Proceedings of the 2020 IEEE ICDE*. 1782–1785.
[169] Ibrahim Sabek and Mohamed F Mokbel. 2021. Machine learning meets big spatial data (revised). In *Proceedings of the IEEE MDM*. 5–8.
[170] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, and Tim Kraska. 2021. When are learned models better than hash functions? arXiv:2107.01464. Retrieved from https://arxiv.org/abs/2107.01464
[171] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. 2022. Can learned models replace hash functions? *PVLDB* 16, 3 (2022), 532–545.
[172] Hans Sagan. 2012. *Space-filling Curves*. Springer Science and Business Media.
[173] Hanan Samet. 1984. The quadtree and related hierarchical data structures. *ACM Computing Surveys* 16, 2 (1984), 187–260.
[174] Hanan Samet. 2006. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann.
[175] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. 2016. Meta-learning with memory-augmented neural networks. In *Proceedings of the International Conference on Machine Learning*. PMLR, 1842–1850.
[176] Atsuki Sato and Yusuke Matsui. 2023. Fast partitioned learned bloom filter. arXiv:2306.02846. Retrieved from https://arxiv.org/abs/2306.02846
[177] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. arXiv:1707.06347. Retrieved from https://arxiv.org/abs/1707.06347
[178] Thomas Seidl, Ira Assent, Philipp Kranen, Ralph Krieger, and Jennifer Herrmann. 2009. Indexing density models for incremental learning and anytime classification on data streams. In *Proceedings of the EDBT*. 311–322.
[179] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.
[180] Naufal Fikri Setiawan, Benjamin IP Rubinstein, and Renata Borovica-Gajic. 2020. Function interpolation for learned index structures. In *Proceedings of the Databases Theory and Applications: 31st Australasian Database Conference, ADC*. Springer, 68–80.
[181] Yufan Sheng, Xin Cao, Yixiang Fang, Kaiqi Zhao, Jianzhong Qi, Gao Cong, and Wenjie Zhang. 2023. WISK: A workload-aware learned index for spatial keyword queries. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.
[182] Jin Shieh and Eamonn J. Keogh. 2009. *i*SAX: Disk-aware Mining and Indexing of Massive Time Series Datasets. *Data Mining and Knowledge Discovery* 19, 1 (2009), 24–57.
[183] Terézia Slanináková, Matej Antol, Jaroslav OĬha, Vojtěch Kaňa, and Vlastislav Dohnal. 2021. Data-driven learned metric index: An unsupervised approach. In *Proceedings of the International Conference on Similarity Search and Applications*. Springer, 81–94.
[184] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. 2021. Bounding the last mile: Efficient learned string indexing. arXiv:2111.14905. Retrieved from https://arxiv.org/abs/2111.14905
[185] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. PLEX: Towards practical learned indexing. In *Proceedings of the aiDB*.
[186] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. 1990. The implementation of POSTGRES. *IEEE TKDE* 2, 1 (1990), 125–142.
[187] Xinyu Su, Jianzhong Qi, and Egemen Tanin. 2023. A fast hybrid spatial index with external memory support. In *Proceedings of the 2023 IEEE ICDEW*. 67–73.

[188] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned index: A comprehensive experimental evaluation. *PVLDB* 16, 8 (2023), 1992–2004.
[189] Chuzhe Tang, Zhiyuan Dong, Minjie Wang, Zhaoguo Wang, and Haibo Chen. 2019. Learned indexes for dynamic workloads. arXiv:1902.00655. Retrieved from https://arxiv.org/abs/1902.00655
[190] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: A scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 308–320.
[191] Yao Tian, Tingyun Yan, Xi Zhao, Kai Huang, and Xiaofang Zhou. 2022. A learned index for exact similarity search in metric spaces. *IEEE Transactions on Knowledge and Data Engineering* 35, 8 (2022), 7624–7638.
[192] Yulai Tong, Jiazhen Liu, Hua Wang, Ke Zhou, Rongfeng He, Qin Zhang, and Cheng Wang. 2023. Sieve: A learned data-skipping index for data analytics. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3214–3226.
[193] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: A learning-enhanced range filter. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1632–1644.
[194] Kapil Vaidya, Eric Knorr, Michael Mitzenmacher, and Tim Kraska. 2020. Partitioned learned bloom filters. In *Proceedings of the ICLR*.
[195] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. 2008. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*. 1096–1103.
[196] Congying Wang and Jia Yu. 2022. GLIN: A lightweight learned indexing mechanism for complex geometries. arXiv:2207.07745. Retrieved from https://arxiv.org/abs/2207.07745
[197] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned index for spatial queries. In *Proceedings of the 2019 20th IEEE MDM*. 569–574.
[198] Lijun Wang, Linshu Hu, Chenhua Fu, Yuhan Yu, Peng Tang, Feng Zhang, and Renyi Liu. 2023. SLBRIN: A spatial learned index based on BRIN. *ISPRS International Journal of Geo-Information* 12, 4 (2023), 171.
[199] Ning Wang and Jianqiu Xu. 2021. Spatial queries based on learned index. In *Proceedings of the Spatial Data and Intelligence Conference, SpatialDI*. Springer, 245–257.
[200] Wei Wang, Meihui Zhang, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. 2016. Database meets deep learning: Challenges and opportunities. *ACM Sigmod Record* 45, 2 (2016), 17–22.
[201] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. 2020. SIndex: A scalable learned index for string keys. In *Proceedings of the SIGOPS Asia-Pacific Workshop on Systems*. 17–24.
[202] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. 2013. A data-adaptive and dynamic segmentation index for whole matching on time series. *Proceedings of the VLDB Endowment* 6, 10 (2013), 793–804.
[203] Zhaoguo Wang, Haibo Chen, Youyun Wang, Chuzhe Tang, and Huan Wang. 2022. The concurrent learned indexes for multicore data storage. *ACM Transactions on Storage* 18, 1 (2022), 1–35.
[204] Zhonghua Wang, Chen Ding, Fengguang Song, Kai Lu, Jiguang Wan, Zhihu Tan, Changsheng Xie, and Guokuan Li. 2023. WIPE: A write-optimized learned index for persistent memory. *ACM Transactions on Architecture and Code Optimization* 21, 2 (2023), 1–25.
[205] Hongwei Wen and Hanyuan Hang. 2022. Random forest density estimation. In *Proceedings of the International Conference on Machine Learning*. PMLR, 23701–23722.
[206] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are updatable learned indexes ready? *Proceedings of the VLDB Endowment* 15, 11 (2022), 3004–3017.
[207] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable learned index with precise positions. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1276–1288.
[208] Shangyu Wu, Yufei Cui, Jinghuan Yu, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2022. NFL: Robust learned index via distribution transformation. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2188–2200.
[209] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. 2019. Designing succinct secondary indexing mechanism by exploiting column correlations. arXiv:1903.11203. Retrieved from https://arxiv.org/abs/1903.11203
[210] Wenkun Xiang, Hao Zhang, Rui Cui, Xing Chu, Keqin Li, and Wei Zhou. 2018. Pavo: A RNN-based learned inverted index, supervised or unsupervised? *IEEE Access* 7 (2018), 293–303.
[211] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum error-bounded piecewise linear representation for online stream approximation. *The VLDB Journal* 23 (2014), 915–937.
[212] Guang Yang, Liang Liang, Ali Hadian, and Thomas Heinis. 2023. FLIRT: A fast learned index for rolling time frames. (2023).
[213] Jingyi Yang and Gao Cong. 2023. PLATON: Top-down r-tree packing with learned partition policy. *SIGMOD* 1, 4 (2023), 1–26.
[214] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the SIGMOD Conference*. ACM, 193–208.

[215] Ahmad Yasin. 2014. A top-down method for performance analysis and counters architecture. In *Proceedings of the 2014 IEEE ISPASS*. 35–44.
[216] Peng Yongxin, Zhou Wei, Zhang Lin, and Du Hongle. 2020. A study of learned KD tree based on learned index. In *Proceedings of the NaNA*. IEEE, 355–360.
[217] Jia Yu and Mohamed Sarwat. 2017. Indexing the pickup and drop-off locations of NYC taxi trips in PostgreSQL–lessons from the road. In *Proceedings of the SSTD*. 145–162.
[218] Tong Yu, Guanfeng Liu, An Liu, Zhixu Li, and Lei Zhao. 2023. LIFOSS: A learned index scheme for streaming scenarios. *WWW* 26, 1 (2023), 501–518.
[219] Eleni Tzirita Zacharatou, Andreas Kipf, Ibrahim Sabek, Varun Pandey, Harish Doraiswamy, and Volker Markl. 2021. The case for distance-bounded spatial approximations. In *Proceedings of the CIDR*. www.cidrdb.org.
[220] Sepanta Zeighami and Cyrus Shahabi. 2023. On distribution dependent sub-logarithmic query time of learned indexing. arXiv:2306.10651. Retrieved from https://arxiv.org/abs/2306.10651
[221] Meng Zeng, Beiji Zou, Xiaoyan Kui, Chengzhang Zhu, Ling Xiao, Zhi Chen, Jingyu Du, et al. 2023. PA-LBF: Prefix-based and adaptive learned bloom filter for spatial data. *International Journal of Intelligent Systems* 2023, 1 (2023), 4970776.
[222] Meng Zeng, Beiji Zou, Wensheng Zhang, Xuebing Yang, Guilan Kong, Xiaoyan Kui, and Chengzhang Zhu. 2023. Two-layer partitioned and deletable deep bloom filter for large-scale membership query. *Information Systems* 119 (2023), 102267.
[223] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: A cache-aware learned index with a cost-based construction algorithm. *PVLDB* 15, 11 (2022), 2679–2691.
[224] Jingtian Zhang, Sai Wu, Zeyuan Tan, Gang Chen, Zhushi Cheng, Wei Cao, Yusong Gao, and Xiaojie Feng. 2019. S3: A scalable in-memory skip-list index for key-value store. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2183–2194.
[225] Songnian Zhang, Suprio Ray, Rongxing Lu, and Yandong Zheng. 2021. SPRIG: A learned spatial index for range and kNN queries. In *Proceedings of the SSTD*. 96–105.
[226] Songnian Zhang, Suprio Ray, Rongxing Lu, and Yandong Zheng. 2022. Efficient learned spatial index with interpolation function based learned model. *IEEE Transactions on Big Data* 9, 2 (2022), 733–745.
[227] Teng Zhang, Jian Tan, Xin Cai, Jianying Wang, Feifei Li, and Jianling Sun. 2022. SA-LSM: Optimize data layout for LSM-tree based storage using survival analysis. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2161–2174.
[228] Yong Zhang, Xinran Xiong, and Oana Balmau. 2022. TONE: Cutting tail-latency in learned indexes. In *Proceedings of the CHEOPS*. 16–23.
[229] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, et al. 2022. Plin: A persistent learned index for non-volatile memory with high performance and instant recovery. *PVLDB* 16, 2 (2022), 243–255.
[230] Zhou Zhang, Pei-Quan Jin, Xiao-Liang Wang, Yan-Qi Lv, Shou-Hong Wan, and Xi-Ke Xie. 2021. COLIN: A cache-conscious dynamic learned index with high read/write performance. *Journal of Computer Science and Technology* 36 (2021), 721–740.
[231] Xun Zhong, Yong Zhang, Yu Chen, Chao Li, and Chunxiao Xing. 2022. Learned index on GPU. In *Proceedings of the 2022 IEEE ICDEW*. 117–122.
[232] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2020. Database meets artificial intelligence: A survey. *IEEE TKDE* 34, 3 (2020), 1096–1116.
[233] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM Computing Surveys* 38, 2 (2006), 6–es.
[234] Beiji Zou, Meng Zeng, Chengzhang Zhu, Ling Xiao, and Zhi Chen. 2022. A learned prefix bloom filter for spatial data. In *Proceedings of the DEXA*. Springer, 336–350.

Received 11 March 2024; revised 8 November 2024; accepted 27 August 2025