A Survey of Multi-dimensional Indexes: Past and Future Trends

Mingxin Li, Hancheng Wang, Haipeng Dai, Meng Li, Chengliang Chai, Rong Gu, Feng Chen, Zhiyuan Chen, Shuaituan Li, Qizhi Liu, Guihai Chen, *Fellow, IEEE*

***Abstract*—Index structures are powerful tools for improving query performance and reducing disk access in database systems. Multi-dimensional indexes, in particular, are used to filter records effectively based on multiple attributes. Classical multi-dimensional index structures, such as KD-Tree, Quadtree, and R-Tree, have been widely used in modern databases. However, advancements in hardware and algorithms have led to the emergence of new types of multi-dimensional index structures. In this paper, we begin by reviewing classical multi-dimensional indexes. Next, we explore the approaches that leverage modern hardware features, such as Solid-State Drive, Non-Volatile Memory, Dynamic Random Access Memory, and Graphics Processing Unit, to improve the performance of multi-dimensional indexes in various aspects. Then, we investigate the novel work of multi-dimensional indexes that apply state-of-the-art machine learning techniques. Finally, we discuss the challenges and future research directions for multi-dimensional indexing methods.**

***Index Terms*—Multi-dimensional Index, Storage Device, Computing Hardware.**

I. Introduction

DATABASE indexing has been extensively studied in academia and industry. Recently, multi-dimensional indexing has become an essential part of indexing research; it involves constructing structures to speed up query processing for filtering based on multiple attributes. Multi-dimensional indexes typically support range queries and k nearest neighbor (kNN) queries. (1) A range query specifies the interval for each dimension and is processed by identifying the boundary and returning data objects within that range. (2) A kNN query seeks to find the k nearest neighbors of a given object. Moreover, updating the structure poses challenges due to the complexity of multi-dimensional data. Although classical multi-dimensional indexes, such as R-Tree and its variants [1]–[3], have been widely studied, advances in hardware and algorithms have led to new possibilities and challenges.

Modern hardware presents both challenges and opportunities. The emergence of Solid-State Drive (SSD), Non-Volatile Memory (NVM), and Dynamic Random Access Memory (DRAM) has made it necessary to optimize indexes for specific storage devices. In addition, Graphics Processing Unit (GPU) and other computing hardware introduce parallelization possibilities. Moreover, algorithms, particularly machine learning algorithms, present significant opportunities. As shown in Fig. 1, this paper aims to explore the multi-dimensional indexes in the context of hardware and algorithms.

A. Hardware

In recent years, hardware has made breakthroughs in storage and computing. Firstly, SSDs are increasingly replacing hard disks as the preferred secondary storage. Secondly, modern computers have relatively large memory capacity. Thirdly, non-volatile storage devices, *e.g.*, NVM, have read/write characteristics similar to DRAM. Finally, a trend in computing is the leveraging of parallel hardware, *e.g.*, GPU, which offers opportunities to design parallelized indexes. Given these advancements, leveraging the features of novel hardware becomes essential to optimize multi-dimensional indexes.

B. Algorithms

Index structures leveraging machine learning methods are called the learned index. The concept of learned indexes, first presented in [4] in 2018, opens up a range of opportunities to improve the performance of index structures. Compared with traditional indexes, learned indexes demonstrate notable improvements in performance metrics, such as query efficiency and storage consumption [4], [5]. As machine learning algorithms evolve, the advantages of the learned index are also reflected in multi-dimensional indexes [6]–[8].

C. Related Survey

The development of multi-dimensional indexes has led to many surveys. Mamun *et al.* [9] focused on learned indexes, introducing both the learned one-dimensional and multi-dimensional indexes. Singh *et al.* [10] investigated traditional and MapReduce-based multi-dimensional indexes. Zhang *et al.* [11] focused on P2P-based multi-dimensional indexes. Ooi *et al.* [12] presented a comprehensive survey on multi-dimensional and spatial indexes. However, this survey lacks an investigation of learned multi-dimensional indexes. Mahmood *et al.* [13] investigated the spatio-temporal access methods. Zhou *et al.* [8] reviewed studies on "AI for DB" and "DB for AI". In their survey, the learned index is covered as a component of the broader discussion. In this paper, we review various multi-dimensional indexes that leverage hardware and novel algorithms, together with a discussion on their advantages and

Mingxin Li, Hancheng Wang, Haipeng Dai, Meng Li, Rong Gu, Feng Chen, Qizhi Liu and Guihai Chen are with Nanjing University. E-mail: {mingxinli, hanchengwang}@smail.nju.edu.cn, haipengdai@nju.edu.cn, menson@smail.nju.edu.cn, gurong@nju.edu.cn, fengchen@smail.nju.edu.cn, {lqz, gchen}@nju.edu.cn.

Chengliang Chai is with Beijing Institute of Technology. E-mail: ccl@bit.edu.cn.

Zhiyuan Chen and Shuaituan Li are with Huawei. E-mail: {hw.chenzhiyuan, hw.lishuaituan}@huawei.com.

(Corresponding authors: Haipeng Dai, Rong Gu and Guihai Chen.)

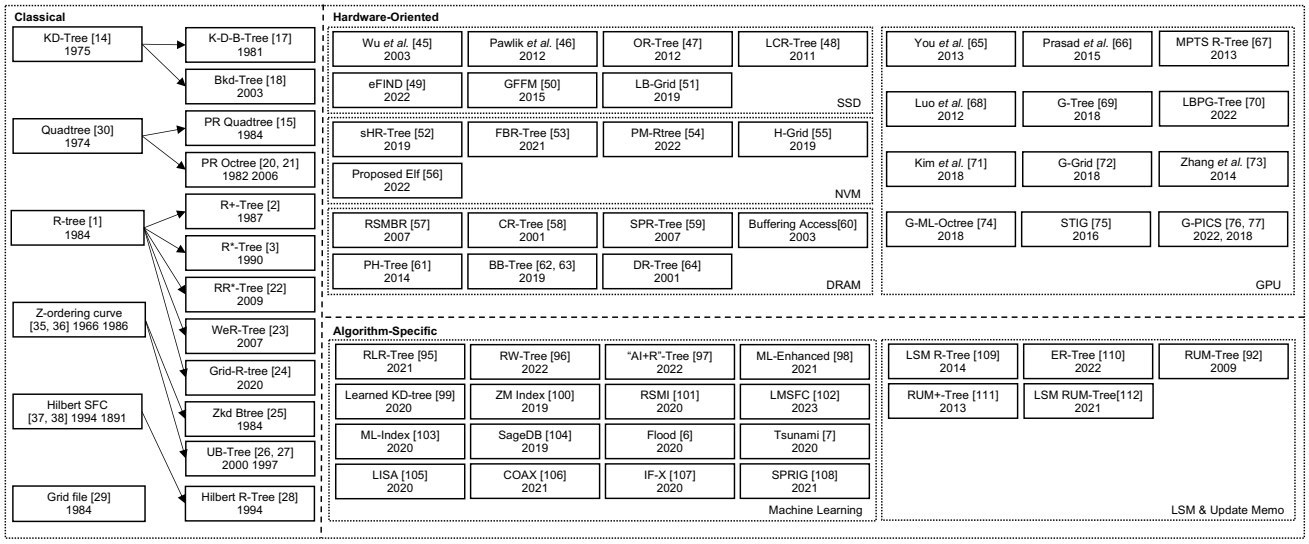


Fig. 1. An overview of classical, hardware-oriented, and algorithm-specific multi-dimensional indexes.

disadvantages. This analysis equips researchers with valuable insights for designing efficient multi-dimensional indexes.

D. Paper Organization

The rest of this paper is organized as follows: Section II introduces classical multi-dimensional indexes. Section III presents studies of optimizing multi-dimensional indexing by leveraging hardware features. Section IV investigates studies on optimizing multi-dimensional index using novel algorithms. Section V summarizes the development trends of multi-dimensional index structures and provides an outlook.

II. Review on Classical Structures

This section reviews classical multi-dimensional index structures, including tree-based structures and space-filling curve-based methods, which are two representative types of the multi-dimensional index [10], [12]. We summarize the function of classical multi-dimensional indexes in Tab. I.

A. Tree-Based Methods

The construction of multi-dimensional indexes using tree structures involves dividing the data into subspaces or subregions, thereby creating a hierarchical structure. In dividing the multi-dimensional data space, there are two primary strategies [12]. The first, exemplified by structures like the KD-Tree [14] and the Point-Region Quadtree [15], along with their various derivatives, involves splitting the space into non-overlapping subspaces. The second, as illustrated by R-Tree [1] and its variants, clusters data into regions that may potentially overlap.

1) KD-Tree and its variants: KD-Tree [14] and its variants are binary tree structures widely used for indexing multi-dimensional point data. These structures support a variety of numerical data types, such as real numbers and integers. For clarity, we focus on the two-dimensional KD-Tree as a representative example. As illustrated in Fig. 2, the KD-tree partitions the data space into regions based on data points in the data space, where each inner node represents a subdivision of its parent node's region [14]. In KD-Trees, the partitioning process alternates between different dimensions. Note that the structure of the tree is affected by the sequences in which

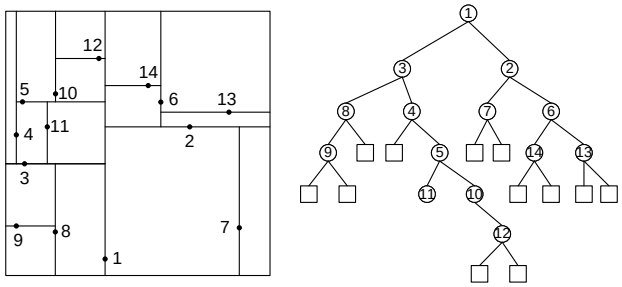


Fig. 2. An illustration of a two-dimensional KD-Tree: The space is alternately split across multiple dimensions, forming a binary tree where circular nodes correspond to data points on the plane [14].

data points are inserted; different insertion sequences can lead to varied tree structures. Friedman *et al.* [16] introduced algorithms specifically designed for KD-Trees aimed at facilitating nearest neighbor searching. They emphasized that the distance function used should demonstrate both symmetry and monotonicity. However, deleting an internal node and subsequently searching for a replacement node in either the left or right subtree can incur considerable search overhead [12]. This aspect underscores the complexity involved in managing KD-Tree structures, especially in dynamic datasets.

K-D-B-Tree [17] is a variant of the KD-Tree and shares a similar strategy for space partitioning. The K-D-B-Tree is comprised of two types of nodes: internal and leaf nodes. In the K-D-B-Tree, the data space is partitioned into subspaces according to the distribution of points, and the information regarding these subspaces is stored in the internal nodes. Each child node represents a disjoint partition of its parent node. A distinctive feature of the K-D-B-Tree is that all data points are stored in leaf nodes, each of which is maintained as a disk block. Similar to the B-Tree, when a single node in the K-D-B-Tree becomes overloaded with data, it necessitates a split. However, such splitting operations in the K-D-B-Tree can potentially lead to decreased space utilization [18]. Additionally, algorithms extending the K-D-B-Tree to support k nearest neighbor searching have been discussed in [19]. Besides, Bkd-Tree [18] is an improved version of K-D-B-Tree.

TABLE I
A SUMMARY OF CLASSICAL MULTI-DIMENSIONAL INDEXES, "R k U" REPRESENTS "RNAGE QUERY, kNN QUERY AND UPDATE", ✔ REPRESENTS "SUPPORTED", ✖ REPRESENTS "NOT SUPPORTED", ❍ REPRESENTS "IN ORIGINAL PAPER, NOT DISCUSSED IN DETAIL"

Name	Method	R k U	Remark
KD-Tree [14]	-	✔ ✔ ✔	The performance of KD-Tree degrades as the dimensionality K increases, especially when the number of data points N is close to or exceeds K.
K-D-B-Tree [17]	KD-Tree-based	✔ ✔ ✔	K-D-B-Tree combines the search efficiency of a balanced KD-Tree with the I/O efficiency of a B-Tree.
Bkd-Tree [18]	KD-Tree-based	✔ ❍ ✔	Bkd-Tree is space-efficient and ensures consistent query performance regardless of the number of insertions. This characteristic makes it particularly suitable for dynamic environments.
PR Quadtree [15]	Quadtree-based	✔ ✔ ✔	PR Quadtree is limited to indexing exclusively two-dimensional data.
PR Octree [20], [21]	Quadtree-based	✔ ✔ ✔	PR Octree is limited to indexing exclusively three-dimensional data.
R-Tree [1]	-	✔ ✔ ✔	R-Tree is effective at indexing both hyper-rectangles and data points. However, the overlaps in its MBRs lead to search overhead.
R+-Tree [2]	R-Tree-based	✔ ❍ ✔	R+-Tree enhances the R-Tree by eliminating overlaps between bounding rectangles, achieving a reduced search cost.
R*-Tree [3]	R-Tree-based	✔ ❍ ✔	R*-Tree enhances numerous aspects of internal bounding rectangles, managing to do so with an insignificant increase in overhead.
RR*-Tree [22]	R*-Tree-based	✔ ✔ ✔	RR*-Tree restricts insertions to a single path and introduces an innovative subtree choosing and node splitting algorithm.
WeR-Tree [23]	R-Tree-based	✔ ✔ ✔	WeR-tree is designed to be insertion-friendly, as it reconstructs subtrees to ensure optimal balance, thereby achieving high node space utilization.
Grid-R-tree [24]	R-Tree-based	✔ ✔ ✔	Grid-R-Tree incorporates grid concepts, which can be applied to R-Tree variants.
Zkd Btree [25]	SFC-based	✔ ❍ ❍	Zkd Btree utilizes the Z-ordering curve to organize multi-dimensional data.
UB-Tree [26], [27]	SFC-based	✔ ❍ ✔	UB-Tree utilizes the Z-ordering curve to sort multi-dimensional data.
Hilbert R-Tree [28]	SFC-based	✔ ✔ ✔	Hilbert R-Tree employs the Hilbert SFC to establish an ordering for rectangles.
Grid file [29]	-	✔ ❍ ✔	Grid File partitions data space into grids, storing points within cells on disk pages.

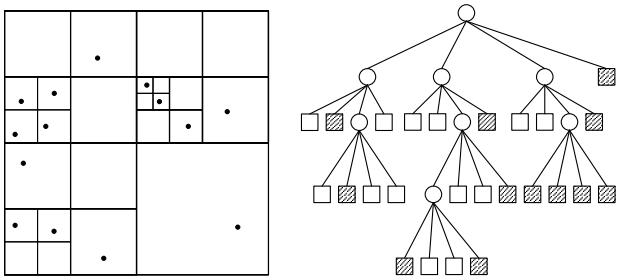


Fig. 3. An illustration of PR Quadtree: The space is recursively quartered until each cell contains at most one data point; shaded squares indicate leaf nodes with data, and empty squares represent vacant regions [15].

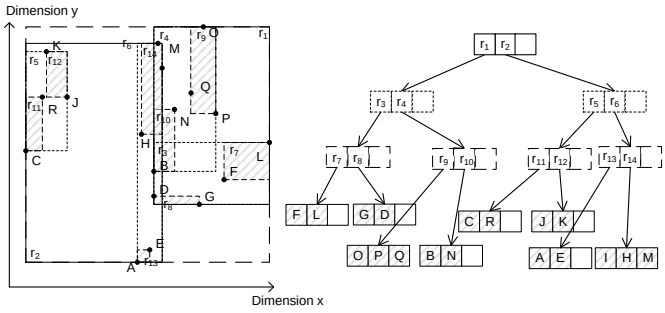
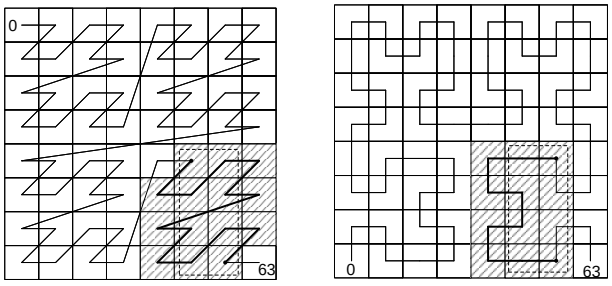


Fig. 4. An illustration of R-Tree: The R-Tree, with a node capacity of three, clusters nearby two-dimensional objects within a single bounding rectangle for indexing [1], [12].

2) Quadtree and its variants: Quadtree [30] is designed to index two-dimensional data. It is capable of representing a variety of data types, including point data, regions, and curves [15]. Among its variants, the Point-Region Quadtree (PR Quadtree) [15] is tailored for indexing points, with each dimension's data type being a real number. Here, we explain the structure of PR Quadtree in detail. Contrasting with the partitioning method of the KD-Tree, the PR Quadtree divides a region into four equally sized smaller regions at each division. In the PR Quadtree, if a point already exists in the target region during data insertion, the square region is further partitioned until each region contains no more than one point. Moreover, as illustrated in Fig. 3, each undivided square region corresponds to a leaf node in the tree structure [15]. Each batch of regions formed by a division in the PR Quadtree corresponds to a layer of nodes, and each undivided region corresponds to a leaf node within the tree structure. Additionally, the algorithms proposed in [19] enable the Quadtree [30] to support kNN searching, with the Euclidean distance function applicable for the searching. The Octree [20] is an encoding and indexing method for three-dimensional objects. Conceptually, the Octree is analogous to the Quadtree.

We observe that both the KD-Tree and the PR Quadtree partition a set of multi-dimensional data into subsets by directly dividing the space, with corresponding nodes in the tree representing these data subsets. However, a notable limitation of quadtree-based indexes is their dimensional constraints.

3) R-Tree and its variants: R-Tree [1] can be considered as an extension of the B-Tree into multi-dimensional space. R-Trees are primarily used to index points or hyper-rectangles, where each dimension typically comprises numerical data. The B-Tree organizes data in a sorted manner, dividing it into segments stored in leaf nodes. R-Tree expands the basic structure of B-Tree to accommodate multi-dimensional data handling. As illustrated in Fig. 4, an R-Tree partitions the multi-dimensional data space into various areas [1]. This is achieved by enclosing nearby data within bounding boxes, and the creation of a Minimum Bounding Box (MBB) serves to enhance the efficiency of an R-Tree. In two-dimensional space, each bounding rectangle corresponds to a node, one or more bounding rectangles can be stored in a node, and a higher-level bounding rectangle is built by grouping nearby lower-level ones. In R-Trees, the bounding rectangle is expanded to enclose a newly inserted object. The subtree that requires the least enlargement is selected to find the target leaf node. If this leaf node is not full, the data is inserted; otherwise, a split operation is performed. In the original approach, due to the overlap between bounding rectangles of the same level, during the search process, more than one rectangle may intersect with the target, which may result in searching over more than one subtree. Additionally, R-Trees support various distance functions, including Euclidean distance and great-circle distance [31]. The MinDist function [32] is utilized to



(a) (b)

Fig. 5. Illustrations of representative space-filling curves [12], [39], [40]: (a) an illustration of two-dimensional Z-ordering curve [35] (b) an illustration of two-dimensional Hilbert space-filling curve [38].

calculate the minimum distance between objects, which may be rectangles or points, within an R-Tree [33].

In the construction of an R-Tree, two primary objectives are typically prioritized. The first goal is to minimize the size of the bounding rectangle, while the second is to reduce the overlapping area between rectangles. R+-Tree [2] improves upon the R-Tree by eliminating overlaps between bounding rectangles. It allows the identifier of the same object to be stored in multiple leaf nodes [12]. In certain conditions, the R+-Tree demonstrates superior search performance compared to the original R-Tree, primarily because searching non-overlapping nodes achieves a lower cost. R*-Tree [3] employs an optimization strategy that considers the area of the directory bounding rectangle, the size of the overlapping area, and the cumulative edge length of each bounding rectangle. Recent developments in this field have led to the emergence of several new variants, including the RR*-Tree [22], WeR-Tree [23], and Grid-R-Tree [24].

Considering that R-Tree [1] and its variants are tailored for indexing hyper-rectangles, specialized algorithms have been developed for calculating distances between objects. These include variations of the Euclidean distance [32], [34]. Depending on the applicable scenario, some present a different distance function [31].

B. Space-Filling Curve-Based Methods

A typical space-filling curve-based (SFC-based) index divides the data space into fine-grained grids, which are subsequently sorted in a specific order. The key concept of this sorting approach is to map multi-dimensional data into a one-dimensional format [12], *i.e.*, employing space-filling curves to establish a deterministic order for the data. Fig. 5 illustrates the two most representative space-filling curves: the Z-ordering curve [35], [36] and Hilbert space-filling curve [37], [38].

1) Z-ordering curve: Fig. 5a illustrates an example of Z-order, also known as the Morton-order [35]. This method recursively divides the space into subspaces. During this division process, the order of each subspace is determined. Subsequently, the Z-ordering curve assigns a unique address to each object within these subspaces.

Orenstein [25] introduced the Zkd Btree, which incorporates the Z-ordering curve for data organization. Ramsak *et al.* [26] further elaborated on applying the Z-ordering curve's sorting method to construct the UB-Tree [27]. To utilize the UB-Tree for indexing, the attributes should be transformed from the original types into a bitstring, necessitating the design of transformation algorithms tailored to each dimension's specific data type. They argue that, compared to other multi-dimensional indexes, the UB-Tree-based index offers easier integration into database systems due to its inherent reliance on the B-Tree structure.

2) Hilbert space-filling curve: Fig. 5b illustrates the Hilbert space-filling curve, which was first proposed by David Hilbert in 1891 [38]. The Hilbert space-filling curve effectively maps data objects from a multi-dimensional space to a one-dimensional space. This curve is notably proficient at preserving the proximity of data points [41]–[43].

Kamel *et al.* [28] proposed the Hilbert R-Tree, utilizing the Hilbert space-filling curve to determine the sorting order of rectangles within the tree. The Hilbert R-Tree supports kNN query, the kNN search algorithm [32] can be effectively applied to the Hilbert R-Tree, the algorithm considers the distance function between a rectangle and a data point.

As discussed previously, classical multi-dimensional indexes exhibit distinct characteristics, which are detailed in Tab. I. Quadtree-based methods are constrained by the number of dimensions they can handle. R-Tree-based methods have the capability to index not only data points but also objects within the space. Classical multi-dimensional indexes lay the basic framework for supporting queries over multi-dimensional data. Significant opportunities exist to enhance the efficiency of classical multi-dimensional indexes and to develop more sophisticated indexing and query processing approaches. These include supporting efficient kNN queries [32], [34], [44], as well as incorporating a broader range of distance functions for specific scenarios [31].

III. Hardware-Oriented Multi-Dimensional Indexes

On the one hand, the advent of advanced storage devices such as Solid-State Drives (SSD), Non-Volatile Memory (NVM), and Dynamic Random Access Memory (DRAM) has led to the development of multi-dimensional index structures optimized for these modern technologies. One notable trend is the replacement of hard disks with flash SSDs, known for their higher performance and decreasing costs [78]. For the convenience of expression, in this paper, SSD specifically refers to the flash SSD unless otherwise stated. Another emerging trend is the integration of NVM-based storage into modern systems, offering data persistence even during power interruptions and potentially outperforming SSDs in terms of read and write performance [79]. Here, NVM is used to denote technologies like 3DXPoint and PCM, distinguishing them from flash SSDs. A third significant development is the increase in main memory capacity, making DRAM a viable option for storing indexes. The design of in-memory multi-dimensional indexes faces new challenges, including reducing memory consumption and optimizing cache usage. On the other hand, the evolution of computing hardware, such as Graphics Processing Unit (GPU), enables algorithms to run on parallel processors, which presents new opportunities.

TABLE II
A SUMMARY OF HARDWARE-ORIENTED MULTI-DIMENSIONAL INDEXES, "R k U" REPRESENTS "RNAGE QUERY, kNN QUERY AND UPDATE", ✔ REPRESENTS "SUPPORTED", ✖ REPRESENTS "NOT SUPPORTED", ❍ REPRESENTS "IN ORIGINAL PAPER, NOT DISCUSSED IN DETAIL"

Name	Method	R k U	Remark
Wu *et al.* [45]	R-Tree-based	❍ ❍ ✔	The proposed R-Tree introduces a fine-grained updating mechanism that mitigates the effects of the block access mechanism in flash storage devices.
Pawlik *et al.* [46]	R-Tree-based	❍ ❍ ✔	The proposed R-Tree is designed to store aggregated values and tree structure separately.
OR-Tree [47]	R-Tree-based	✔ ❍ ✔	OR-Tree inserts data into an overflow node when the target is full, delaying node splitting.
LCR-Tree [48]	R-Tree-based	❍ ❍ ✔	LCR-Tree records updates in Compact Logs, which are flushed to the R-Tree when full.
eFIND [49]	Tree-based	✔ ❍ ✔	eFIND framework facilitates transitioning hard disk-based indexes to SSD environments.
GFFM [50]	Grid file-based	✖ ✖ ✔	GFFM employs a buffering mechanism to enhance its write performance.
LB-Grid [51]	Grid file-based	✔ ✔ ✔	LB-Grid is designed to speed up updates and incorporates multiple query algorithms.
sHR-Tree [52]	R-Tree-based	✔ ❍ ❍	sHR-Tree is a hybrid R-Tree with nodes stored separately in flash and 3DXPoint SSDs.
FBR-Tree [53]	R-Tree-based	✔ ❍ ✔	FBR-Tree uses lock-free algorithms for superior performance over lock-based mechanisms.
PM-Rtree [54]	R-Tree-based	✔ ❍ ✔	PM-Rtree stores non-leaf nodes in DRAM to reduce Persistent Memory latency, boosting update and search performance beyond the FBR-Tree.
H-Grid [55]	Grid file-based	✔ ✔ ✔	H-Grid maintains the Root Directory in RAM for quick access. The hot regions of the Sub-Directory are stored in 3DXPoint SSDs, while the remaining parts are stored in flash SSDs.
Proposed Elf [56]	Elf-based	✔ ❍ ❍	The proposed Elf implements a caching mechanism that stores frequently accessed parts in DRAM, utilizing both dynamic and static strategies.
RSMBR [57]	R-Tree-based	✔ ❍ ✔	RSMBR compresses MBRs to reduce index size; however, it has been observed that this results in a slight decrease in the performance of operations such as insertion and search.
CR-Tree [58]	R-Tree-based	✔ ❍ ✔	CR-Tree compresses the MBR by representing the coordinates of child nodes as relative MBR to the parent node.
SPR-Tree [59]	R-Tree-based	✔ ❍ ✔	SPR-Tree incorporates selective prefetching as a strategy to reduce cache misses, achieving this without compromising the performance of updates.
Buffering Access [60]	R-Tree-based	✔ ❍ ✔	Buffering Access mechanism offers a way for tree-based indexes, such as the R-Trees, to exploit both spatial and temporal locality of cache.
PH-Tree [61]	Quadtree Based	✔ ✖ ✔	PH-Tree introduces a mechanism aimed at reducing node size. This approach yields a more pronounced performance advantage over the KD-Tree, especially when dealing with larger data sets.
BB-Tree [62], [63]	Other Methods	✔ ✖ ✔	BB-Tree is designed to be a main-memory data structure, features a parallel range query operator.
DR-Tree [64]	Other Methods	✔ ❍ ❍	DR-Tree features a two-level structure specifically designed for indexing complex objects and achieves an ideal storage overhead. This structure decomposes these objects and facilitates the use of R*-Tree or other multidimensional indexing methods as upper-level indexes.
You *et al.* [65]	R-Tree-based	✔ ❍ ❍	The study introduces algorithms for GPU-based parallel bulk-loading and batched querying. The proposed index outperforms the classical R-Tree on multi-core CPUs.
Prasad *et al.* [66]	R-Tree-based	✔ ❍ ❍	The study presents a strategy using the Geo-Packing Principle to reduce the R-Tree's branching factor.
MPTS R-Tree [67]	R-Tree-based	✔ ✖ ❍	MPTS introduces parallel range query algorithms, transforming traditional R-Tree traversal into a sequential processing approach.
Luo *et al.* [68]	R-Tree-based	✔ ❍ ❍	The study introduces a method for GPU-based Parallel R-Tree query and construction, storing the R-Tree in two types of arrays. One array represents the index, while the other stores the rectangle coordinates.
G-Tree [69]	R-Tree-based	✔ ❍ ❍	G-Tree processes large-scale, high-dimensional data using optimized GPU memory data structures. It features a BFS-based lookup and selective dimensionality filters for efficient parallel search.
LBPG-Tree [70]	R-Tree-based	✔ ❍ ❍	LBPG-Tree enhances the G-Tree by exploiting the L2 cache utilization more effectively, resulting in performance improvements by several orders of magnitude.
Kim *et al.* [71]	R-Tree-based	✔ ❍ ❍	The study introduces a co-processing scheme in which the CPU traverses internal nodes while the GPU scans leaf nodes. This approach achieves lower query latency compared to GPU-only searches.
G-Grid [72]	Z-ordering SFC	❍ ✔ ✔	G-Grid introduces a mechanism that reduces the frequency of index updates and presents a novel kNN search algorithm, which combines the advantages of both CPU and GPU.
Zhang *et al.* [73]	Quadtree-based	✔ ❍ ❍	The study introduces a parallel design for indexing complex polygons based on parallel primitives.
G-ML-Octree [74]	Quadtree-based	✔ ❍ ✔	G-ML-Octree is designed to index 3-dimensional moving objects within Euclidean space.
STIG [75]	KD-Tree-based	✔ ✖ ❍	STIG features a two-stage indexing mechanism: the upper part consists of a KD-Tree, while the leaf nodes are novel tree-structured elements pointing to leaf blocks.
G-PICS [76], [77]	Other Methods	✔ ✔ ✔	G-PICS is a framework that provides algorithms for tree-based index building, batch query processing, and update handling.

A. SSD

This subsection focuses on multi-dimensional indexes optimized for flash SSDs. These include indexes based on R-Tree and Grid file structures, as well as their variants [78].

1) R-Tree-based methods: As previously discussed, the R-Tree can be regarded as a multi-dimensional variant of the B-Tree, expanding its utility in spatial data management. It uses bounding rectangles to cluster a set of nearby data objects in two-dimensional space, forming a tree-based structure across multiple levels of these rectangles. Many studies have focused on adapting R-Trees for flash SSD storage. Particularly, some studies leverage the Flash Translation Layer (FTL) to improve the performance of multi-dimensional indexes [78].

Wu *et al.* [45] proposed an R-Tree implementation on the FTL, focusing on reducing unnecessary writes. For an R-Tree on hard disks, modifying an R-Tree node is efficient. However, for an R-Tree on flash storage, the problem is that the update of an R-Tree node is not done in-place. An erase operation should be performed first on a flash memory block. These features can result in high energy consumption. To address these challenges, the study outlines three strategies: Reservation Buffer, Index Units, and Node Translation Table. The modifications on the R-Tree are temporarily held in memory by applying the Reservation Buffer mechanism, which aims to reduce unnecessary modifications on the R-Tree structure. Then, an Index Unit that contains the critical information of the correspondence between an object and an R-Tree node is constructed when the objects in a buffer are flushed into an R-Tree structure. An R-Tree node may have several corresponding Index Units. Thus, the Node Translation Table is built to establish a connection between Index Units and R-Tree. It is an array of lists that maps the Index Unit to a corresponding R-Tree node. These strategies enhance the efficiency and performance of the proposed R-Tree.

Pawlik *et al.* [46] proposed an improved approach for constructing R-Trees. This study introduces a more fine-grained update method, enhancing the performance of the aggregated R-Tree [46], [80], which is a variant of R-Tree that stores aggregated values in R-Tree nodes. In this method [46], R-Tree nodes and aggregated values are stored in separate sectors of flash memory. Therefore, updating an aggregated value does not affect the R-Tree structure's sectors, thereby reducing unnecessary modifications to the flash storage. Overall, this method significantly reduces unnecessary modifications to flash storage, capitalizing on the observation that updates to aggregated values occur more frequently than structural updates to the R-Tree itself.

OR-Tree [47] introduces a unique optimization approach for the R-Tree. Similar to R-Tree, OR-Tree can index spatial data objects, such as the MBRs of polylines. Differing from conventional methods, the OR-Tree employs an unbalanced structure, specifically addressing the node-splitting challenge inherent in R-Trees. While the standard R-Tree requires immediate node-splitting when inserting into a full node, the OR-Tree defers this splitting. In the OR-Tree model, modifications are temporarily housed in an overflow node, delaying immediate splitting. Additionally, the OR-Tree introduces an Overflow Node Table to track the relationship between overflow nodes and their corresponding original nodes scheduled for splitting. A novel algorithm is developed to merge the overflow nodes and R-Tree nodes, which can reduce modifications. The OR-Tree incorporates new insertion algorithms that aim to minimize modifications on flash storage, effectively utilizing the main memory buffer. Due to these structural modifications, the search algorithm of OR-Tree should perform on both the R-Tree node and corresponding overflow nodes. The experimental results indicate that OR-Tree's performance exceeds that of other R-Tree implementations on flash storage, including the standard R-Tree and the FAST-Rtree [81].

LCR-Tree [48] stands for Log Compact R-Tree. The LCR-Tree is capable of indexing data points in spatial dimensions or Minimum Bounding Rectangles (MBRs) of data objects within a data space. A prevalent strategy for optimizing data structures on SSDs involves transforming random writes into sequential writes. Within the LCR-Tree, new modifications are amalgamated with the existing logs of an R-Tree node to form compact logs. Once the compact log reaches its full capacity, its contents are flushed to the corresponding R-Tree nodes. By using the logging mechanism, random writes of SSD are transformed into sequential writes. For read operations, both the R-Tree nodes and the compact logs must be searched, given that updates are temporarily stored in these logs. Additionally, the LCR-Tree algorithm is both flexible and robust. Its strategy is compatible with the original R-Tree structure, allowing it to be seamlessly applied to both R-Tree and its various variants.

eFIND framework [49] is designed to transform multi-dimensional indexes, originally based on hard disk storage, for optimization on SSD storage. This transformation is applicable to indexes like the R-Tree [1] and its variants, including the R*-Tree [3] and the Hilbert R-Tree [28]. Upon transformation, these indexes are respectively renamed as eFIND R-Tree, eFIND R*-Tree, and eFIND Hilbert R-Tree. Additionally,

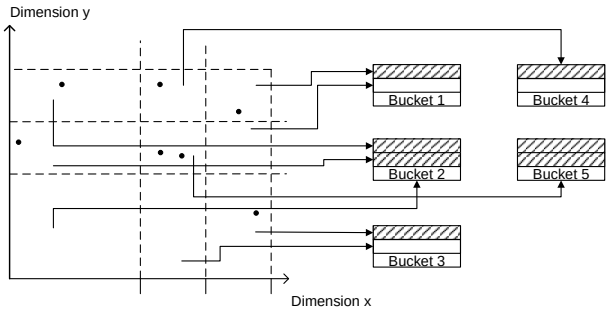


Fig. 6. An illustration of Gird file structure: The dashed lines partition the grid, and the data points are stored in data buckets [29].

eFIND is capable of transforming other index structures, such as the xBR+-tree [82], resulting in the eFIND xBR+-Tree. Indexes transformed by the eFIND framework are adept at indexing real spatial datasets, which consist of data points. The data type of each dimension is numeric, typically represented by floating-point numbers.

Most R-tree-based indexes address the problem of frequent writes on SSD, for the update mechanism of flash SSD causes energy consumption and frequent erasing and writing.

2) Grid file-based methods: As shown in Fig. 6, the Grid file is designed for multi-dimensional data indexing [29]. It divides the data space into disjoint cells. This method comprises two parts: the first consists of k one-dimensional arrays storing the division information for each dimension. The second part, known as the directory, is a k-dimensional array. In the Grid file, each cell points to a specific data page or bucket. Notably, a single data page or bucket can accommodate one or more cells [12], [83].

With the rise of flash storage, it is necessary to evaluate the applicability of the classical Grid file on flash storage. Fevgas *et al.* [50] proposed a Grid file for Flash Memory (GFFM). They incorporated a page buffer method tailored to SSD characteristics, enhancing write performance through a batch writing scheme. In GFFM, pages in the buffer are persisted using a batch-processing approach. Additionally, the study evaluates the performance of the Grid file and implements an R*-Tree [3] on flash devices to compare the performance of these two index structures. The results show that the Grid file on flash devices outperforms the R*-Tree in several aspects. However, GFFM has certain limitations, notably its inability to support kNN queries.

LB-Grid [51] represents a Log Bucket Grid file. LB-Grid is designed to index geographical data points, the datatype of individual dimension is numerical. The structure contains four parts: the root directory stored in memory, the sub-directory, the log-bucket for storing log data, and a bucket translation table maintained in memory. Moreover, LB-Grid introduces specific algorithms for range queries, kNN queries, and group point queries. These algorithms are adaptable to higher-dimensional datasets. Additionally, the study proposes a method to enhance Grid file performance by utilizing the internal parallelization capabilities of SSDs. LB-Grid is evaluated on two types of access interface of SSD, which are Non-Volatile Memory Express (NVMe) and Serial Advanced Technology Attachment (SATA).

B. NVM

There are mainly three categories of multi-dimensional indexes on NVM, *i.e.*, R-Tree-based [1] methods, Grid file-based [29] methods, and Elf-based [84] methods.

1) R-Tree-based methods: sHR-Tree [52] aims to implement an R-Tree on NVM, conducting extensive experiments to assess its performance. It indexes data points extracted from the OSM dataset; therefore, the datatype of individual dimensions can be a real number. Note that the NVM is used as the secondary storage device in this study. There are two approaches to implementing a hybrid R-Tree: (1) Storing hot data region (subtree) in high-speed storage devices and the other parts in low-speed devices; (2) Storing leaf nodes in flash SSDs and inner nodes in 3DXPoint NVM. The study outlines a fetch algorithm designed to retrieve R-Tree nodes from hybrid storage systems. The sHR-Tree focuses on the potential of R-Tree to integrate NVM and SSD in hybrid tree configurations. Additionally, for a more comprehensive evaluation, the study implements two other structures: an R-Tree solely on SSD and another solely on NVM. However, the sHR-Tree does not give a discussion of kNN query support.

FBR-Tree [53] proposes a novel algorithm called the byte-addressable and failure-atomic algorithm to optimize R-Tree on NVM. The FBR-Tree is capable of indexing point data, with each dimension supporting the real number. Due to the different accessing methods on NVM compared to hard disks, the algorithm specifically addresses the challenge of designing efficient insert and delete operations for the FBR-Tree. These operations are executed as a series of 8-byte instructions. Another design challenge is the development of a novel node split algorithm tailored for the FBR-Tree. To address this issue, the FBR-Tree introduces an in-place rebalancing algorithm, supporting efficient node splitting and merging [53].

PM-Rtree [54] is another variant of R-Tree on persistent memory, which is also a hybrid structure. PM-Rtree can index rectangles, points, and other types of objects. In the PM-Rtree, the inner nodes are stored in DRAM, while the leaf nodes are stored in NVM. PM-Rtree addresses the existing problems of FBR-Tree [53], *i.e.*, high update latency, high consumption of mutex locks, large tree height, and the possibility of cache line flushes. The PM-Rtree proposes a lock-free mechanism for insertion and deletion operations, and it also presents a method to alleviate cache line reflushes. Moreover, its hybrid structure significantly reduces write operations on NVM by storing the inner nodes in DRAM.

2) Grid file-based methods: H-Grid [55] is a variant of Grid file, and it is a hybrid multi-dimensional index on flash SSD and 3DXPoint NVM. It can index geographic data points extracted from the OSM dataset. Therefore, it supports point data where each dimension is of numeric type. Note that the 3DXPoint NVM is used as secondary storage in this study. The study introduces a hybrid framework for the Grid file, where the entire index structure is not stored solely in NVM devices. The root directory is stored in Random Access Memory (RAM), which contains the address of the sub-directory. Sub-directories store the address of data buckets. Sub-directories and data buckets are stored in SSD or NVM, based on whether they are in hot or cold data storage regions. H-Grid introduces algorithms for detecting hot regions and then stores these hot regions in 3DXPoint NVM. H-Grid presents algorithms for point query, range query, and kNN query, as well as algorithms to support insert operation. The authors suggested further study of a cooling strategy for buckets in NVM.

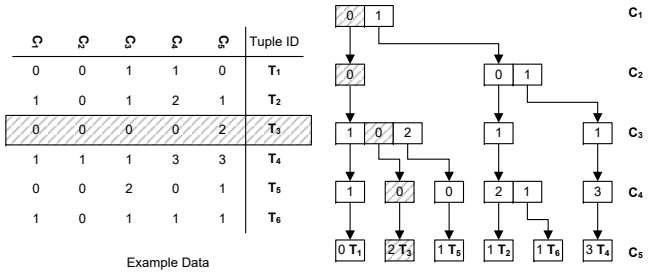


Fig. 7. An illustration of an elf structure: The table represents data to be indexed, with each column corresponding to a layer in a tree structure and each row to a tree path. For example, the row in grey is a path in the tree structure, and the path is also colored grey. The node in the last layer has a unique identifier for the tuple [84].

3) Elf-based methods: Original Elf [84] is a multi-dimensional index designed for DRAM. As illustrated in Fig. 7, Elf constructs a tree structure using a prefix-redundancy approach, where the structure is composed of multiple columns. Each column in the table corresponds to one layer of the tree structure, marked as C_i in Fig. 7, thereby fixing the tree's height to the number of columns. Every node in the tree structure is called a DimensionList; the DimensionList at the last layer of the tree structure stores the unique identifiers of tuples. The Elf tree structure can be mapped into an array. The authors proposed replacing the first layer of Elf with a hash map. This is because the first layer contains only one DimensionList, which includes all distinct values, making scanning this DimensionList costly. Furthermore, to address nodes in deeper layers potentially having only one child, the authors proposed reducing the tree's height. This is achieved by storing subsequent columns adjacent to a DimensionList. For example, some DimensionLists of C_5 could be directly stored after those in C_4.

Jibril *et al.* [56] proposed Pure PM-based, Hybrid, and Cached Elf. The Cached Elf leverages selective caching to adapt Elf [84] to a storage environment containing NVM devices. The proposed Elf supports Exact-Match Query, Range Query, and Partial-Match Query; the data type of each dimension is numerical. The Hybrid Elf involves storing a volatile copy in DRAM while simultaneously constructing an Elf on NVM, aiming to facilitate rapid recovery of the Elf structure. To reduce the overhead, Cached Elf stores critical parts in DRAM, and the study considers strategies of Dynamic Caching and Static Caching. The Dynamic Caching strategy caches each traversed DimensionList, while the Static Caching caches the fixed key parts of elf.

C. DRAM

This section introduces multi-dimensional indexes optimized for DRAM storage, including R-Tree-based methods, Quadtree-based methods, and other new methods.

1) R-Tree-based methods: RSMBR [57] is a Minimum Bounding Rectangle (MBR) compression technique. In R-Trees, Minimum Bounding Boxes (MBB) are used to group

and bound nearby data objects. The tree index is then built on a hierarchy of these MBBs, with the Minimum Bounding Rectangle (MBR) serving as its two-dimensional counterpart. There are traditional methods used to reduce entry size; however, some of these algorithms increase the MBR reconstruction cost and decrease the search efficiency. To implement competitive MBR compression algorithms that improve the performance of MBR reconstruction, RSMBR optimizes the indexes in main memory by setting base points for different distributions to compress MBR. Similarly, CR-Tree [85] also aims to achieve better performance by building a cache-conscious R-Tree. CR-Tree compresses MBR keys to enable more data to fit in a node. Compared with the original R-Tree, the strategy of CR-Tree increases the fanout and decreases index size, which is conducive to better utilization of the cache mechanism. CR-Tree presents the range search methods to find the records within a query rectangle.

SPR-Tree [59] is also a variant of R-Tree, which is short for Selective Prefetching R-Tree. The SPR-Tree is capable of indexing spatial data objects efficiently. The authors noted that some existing methods that reduce the size of an entry may require additional computing consumption. To address the problem of cache misses, SPR-Tree uses a selective prefetching technique by optimizing the size of a node according to cache block size to adapt the structure to the prefetching mechanism. SPR-Tree proposes a search algorithm for finding rectangles that overlap with the query rectangle. The kNN search or other search algorithms are not discussed. Buffering Accesses [60] provides a method for designing a "cache-conscious" index. This method utilizes buffer accesses to enhance index performance and optimizes the cache utilization. Additionally, the study investigates multiple aspects of designing a buffering mechanism. This technique can be applied to R-Tree to reduce the probability of cache miss.

Hwang *et al.* [86] evaluated the performance of main memory R-Trees, *i.e.*, R-Tree [1], R*-Tree [3], Hilbert R-Tree [28], CR-Tree [58], CR*-Tree, and Hilbert CR-Tree.

Analyzing the above studies, the main content is how to build a structure suitable for memory. The design of multiple query algorithms in DRAM, such as the kNN search algorithm, is not well studied.

2) Quadtree-based methods: PH-Tree [61] is a PATRICIA-hypercube-tree based on the Quadtree [15]. Its primary focus is on in-memory storage, facilitating both point and range queries. The PH-Tree indexes multi-dimensional data points; therefore, the data type of each dimension can be numeric, *e.g.*, the real number. It is a variant of the Quadtree, combining hypercubes, prefix-sharing [87], and bit-stream storage [88]. The index is distinguished by three primary features [61]. Firstly, PH-Tree divides a space covered by a tree node across all dimensions instead of one. Secondly, PH-Tree is unbalanced, meaning that extra balancing operations are not necessary; nevertheless, PH-Tree presents an algorithm to limit the imbalance degree of a tree. Finally, PH-Tree's strategy for improving space efficiency focuses on reducing node size rather than increasing the occupancy ratio of nodes.

3) Other methods: BB-Tree [62] is a novel in-memory multi-dimensional index structure. It consists of two essential components: an internal tree search structure and leaf nodes, which are composed of bubble buckets that store data. Its inner search tree adopts a flat and static array format, constituting a cache-friendly design. The capacity of the leaf nodes plays a crucial role in balancing the search efforts between the inner tree and the leaf nodes. BB-Tree is capable of supporting partial-range, complete-match range, and exact-match queries. The experimental results demonstrate that the BB-Tree performs well with data dimensions ranging from 3 to 100.

DR-Tree [64] is a main memory multi-dimensional index structure designed to cooperate with the R*-Tree [3]. To handle complex multi-dimensional data, the study presents a two-step structure combining the DR-Tree and the R*-Tree. The R*-Tree indexes the original data objects, with its leaf nodes storing MBRs and containing pointers to the DR-Tree. DR-Tree is used to organize decomposed components of the original data objects. The proposed approach supports point query, region query, and spatial join query.

The optimization methods for multi-dimensional indexes vary depending on the specific storage device (DRAM, NVM, SSD) and the application scenarios. Index structures on DRAM focus on reducing index size and enhancing cache efficiency, and structures on NVM emphasize utilizing the hierarchy of the DRAM-NVM-SSD storage system to design efficient hybrid structures. Indexes on flash SSD tend to adapt disk-based indexes to the flash SSD.

D. GPU

This section introduces the multi-dimensional indexes optimized for GPU to explore the potential of parallel computing.

1) R-Tree-based methods: You *et al.* [65] proposed a suite of R-Tree algorithms, including a bulk loading algorithm and various query algorithms, to fully harness the capabilities of GPUs. The proposed approach can be used to index rectangles and points and supports range queries. The study introduces a method for representing an R-Tree node as a linear data structure aimed at reducing memory overhead. Moreover, a series of parallel algorithms are designed for R-Tree on GPU. The study demonstrates that the proposed GPU-based R-Tree outperforms the traditional R-Tree implemented on a multi-core CPU. Similarly, Prasad *et al.* [66] proposed an R-Tree variant that is well-suited for GPU parallelization. This R-Tree variant is designed for indexing spatial point data, such as event locations, and supports numerical data types in each dimension. One of its key algorithms is the parallelized R-Tree construction algorithm that employs a technique called Geo-Packing to pre-sort MBRs. This mechanism optimizes GPU memory consumption and enhances search performance by minimizing the dead and overlap areas of MBRs. The study also presents a parallel query mechanism, which leverages the Breadth-First Search (BFS) method and the Depth-First Search (DFS) method. MPTS R-Tree [67] presents a Massively Parallel Three-phase Scanning method, addressing the challenges of traveling an R-Tree for multi-dimensional range searches on GPUs. MPTS R-Tree supports building indexes on point data, and the data type of each dimension can be numeric. It also supports building indexes on two-dimensional MBBs. Handling a query that overlaps with more than one bounding

rectangle is a challenge faced by classical R-Tree. A key strategy of MPTS for enhanced performance involves using multi-threading to cooperatively validate any overlap between a bounding rectangle and the query.

Luo *et al.* [68] proposed a parallel R-Tree implementation, utilizing GPUs to optimize the construction process. The proposed approach builds indexes over data rectangles. The key distinction from classical R-Trees is that this parallel version is stored in GPU global memory, utilizing two types of arrays: the Index Array and the Rectangle Array. Index Array stores the node structures in an R-Tree, and Rectangle Array stores rectangle coordinates, which are the left-bottom and right-top points of a rectangle. G-Tree [69] is designed to construct an index that maintains linear performance, irrespective of increasing data dimensions, by merging the benefits of R-Trees with GPU parallelism. It indexes multi-dimensional image features extracted by SURF algorithm [89] and point data. The G-Tree facilitates range queries on high-dimensional data, supporting up to 256 dimensions. To improve parallelization performance, instead of using a stack or a queue, G-Tree introduces a new structure called the structure of arrays (SoA) that fits the features of GPU memory better, and the structure is used to save traversal results. To maximize GPU parallelism, the G-Tree employs a two-layer framework: the first layer progressively filters a subset of dimensions, while the second layer parallelizes the extraction of exact query results. LBPG-Tree [70] enhances the G-Tree in two major ways: by introducing a level-based pipeline mechanism for handling multiple range queries and by developing a strategy to better utilize the GPU's cache mechanism.

Kim *et al.* [71] proposed a cooperative framework for tree searching tasks, using a CPU-GPU balance where the CPU navigates tree layers for internal nodes while the GPU scans leaf nodes. In the study, R-tree [1] and LBVH [90] can be used as the upper structure. The proposed approach can be used to index multi-dimensional point data. The experimental results demonstrate that this hybrid framework outperforms other GPU-based methods in terms of query response time.

2) Space-filling curve-based methods: As previously discussed, the Z-ordering curve, a type of space-filling curve, establishes a sorting order for multi-dimensional objects. In the construction of multi-dimensional indexes on GPUs, Z-ordering plays a crucial role in the sorting process.

G-Grid [72] is tailored for road networks, and it utilizes GPU capabilities to efficiently manage frequent updates in spatial databases. First, data objects are divided into cells, with each cell representing information about road network vertices and their connecting edges. Subsequently, a Z-ordering curve sorts these cells into a one-dimensional space, assigning a unique Z-value to each cell. The G-Grid index employs a "lazy update" strategy, where updates are cached and only performed when the corresponding cells are queried. Additionally, a GPU-based algorithm determines the necessity of update operations, with parallel processing employed to eliminate unnecessary messages. Furthermore, G-Grid incorporates a CPU-GPU collaborative framework for handling kNN queries. Here, the GPU selects query candidates, and the CPU processes these candidates to derive the final results.

3) Quadtree-based methods: Zhang *et al.* [73] proposed a multi-dimensional index structure based on Quadtree, emphasizing the processing of spatial data using parallel primitives. The study introduces three main modules, *i.e.*, a method for decomposing complex polygons using a top-down approach, an algorithm for constructing a Quadtree, and parallel query algorithms designed to process multiple queries. The adoption of parallel primitives in this study offers advantages not only for GPUs but also for other parallel hardware platforms.

Deng *et al.* [74] introduced the G-ML-Octree, a concept focused on parallelizing the Octree structure. The study first presents an ML-Octree, based on a strategy that combines features of the loose Octree [91] with update-memo strategies [92], aiming to enhance index update efficiency. Then, the study describes the construction of the ML-Octree on a GPU, termed as G-ML-Octree. The structure contains two parts: the first part comprises a cell array and a data array, both maintained in GPU memory; the second part, known as the update memo, is also stored in GPU memory as a two-level hash table. For enhancing the performance of update operations, the G-ML-Octree introduces a method to balance the workloads across multiple GPU threads.

4) KD-Tree-based methods: KD-Tree is a form of binary search tree, and it has been the focus of several studies aiming to integrate it with GPUs. These studies seek either to accelerate GPU computing [93], [94] or to exploit GPUs' parallel processing capabilities to enhance KD-Tree's performance. For instance, the tree node structure of STIG [75] resembles that of the KD-Tree. The index utilizes a series of leaf blocks for data storage, with each leaf node corresponding to a specific leaf block. STIG is designed to index point data, where the individual dimensions may include temporal attributes or location data attributes. The STIG structure offers several advantages in processing multi-dimensional data on GPUs. These include improved efficiency in clustering nearby data, maximized utilization of GPU computing power, and a reduction in the overall size of the tree structure.

5) Other methods: Some studies attempt to address prevalent challenges in processing multi-dimensional data on GPUs by introducing novel frameworks. G-PICS [76], [77] provides a framework specifically designed to support efficient parallel query processing. It encompasses three main components: the construction of a tree, efficient querying over the tree, and updating the tree structure. Specifically, G-PICS is not designed for a specific tree structure. The framework includes construction algorithms for both space-driven and data-driven partitioning trees. G-PICS capitalizes on parallel computing capabilities, processing multiple queries by distributing a batch of queries across multiple threads, with each thread handling a single query.

Designing a GPU-based index requires careful consideration of parallelization algorithms, index structure storage formats, and CPU-GPU cooperative mechanisms. As shown in Tab. II, numerous GPU-based indexes lack support for updating. The G-PICS transforms traditional indexes into GPU-friendly formats and supports various queries, offering insights for designing GPU-based multi-dimensional indexes.

TABLE III
A SUMMARY OF ALGORITHM-SPECIFIC MULTI-DIMENSIONAL INDEXES, "R k U" REPRESENTS "RNAGE QUERY, kNN QUERY AND UPDATE", ✔ REPRESENTS "SUPPORTED", ✖ REPRESENTS "NOT SUPPORTED", ❍ REPRESENTS "IN ORIGINAL PAPER, NOT DISCUSSED IN DETAIL"

Name	Method	R k U	Remark
RLR-Tree [95]	R-Tree based	✔ ✔ ✔	RLR-Tree uses a reinforcement learning-based model for subtree choosing and node splitting. This approach is compatible with the existing algorithms of the R-Tree.
RW-Tree [96]	R-Tree based	✔ ✔ ✔	RW-Tree is designed to learn the distribution of workloads and utilizes a machine learning-enhanced cost model to address the challenge of the subtree choosing and node splitting.
"AI+R"-Tree [97]	R-Tree based	✔ ✖ ❍	"AI+R"-Tree enhances traditional R-Tree node traversal by identifying queries with high overlap, thereby bounding and optimizing the worst-case search performance.
ML-enhanced [98]	Tree based	❍ ✔ ✖	ML-enhanced index is specifically designed for kNN queries in high-dimensional data spaces and is compatible with various tree-based index structures.
Learned KD-tree [99]	KD-Tree based	❍ ✔ ❍	Learned KD-tree, specifically designed for kNN searches in high-dimensional data spaces, it incorporates a neural network model to optimize the nearest neighbor searching.
ZM Index [100]	SFC	✔ ❍ ✖	ZM Index employs a Z-ordering curve to establish an order of data, along with a multi-staged model for data partitioning. Its performance is partially dependent on the data distribution.
RSMI [101]	SFC	✔ ✔ ✔	RSMI transforms the original data into a rank space, which results in a flattened curve. The order of point data is then determined by applying a space-filling curve.
LMSFC [102]	SFC	✔ ❍ ✔	LMSFC is tailored for low-dimensional spatial data and offers a strategy to develop a learnable space-filling curve. This is aimed at enhancing query processing efficiency.
ML-Index [103]	iDistance	✔ ✔ ❍	ML-Index is designed for kNN search and works by aggregating nearby data to build an index. However, the number of query types is limited.
SageDB [104]	Grid partition	✔ ❍ ❍	SageDB proposes a general framework for developing learned multi-dimensional indexes; however, it lacks specific implementation details.
Flood [6]	Grid partition	✔ ❍ ✖	Flood, an in-memory index, is optimized for read operations and adapts to query workloads. However, it falls short in scenarios with data correlation across dimensions and skewed query distributions.
Tsunami [7]	Grid partition	✔ ❍ ✖	Tsunami, an enhanced version of Flood, considers both the data correlation across dimensions and the query skew.
LISA [105]	Grid partition	✔ ✔ ✔	LISA, designed for low-dimensional data, supports various query types, including accurate kNN queries. It also provides functionality for both insertion and deletion.
COAX [106]	Data correlation	✔ ❍ ✖	COAX is designed to learn the correlations between various data attributes, achieving lower memory overhead compared to the R-Tree.
IF-X [107]	Interpolation	✔ ❍ ❍	IF-X is tailored for low-dimensional spatial data, sorting data over one of its multiple dimensions. However, the distribution of queries within this framework should be given further consideration.
SPRIG [108]	Interpolation	✔ ✔ ❍	SPRIG utilizes a spatial interpolation function tailored for two-dimensional spatial data. However, this specific approach inherently limits its capacity to handle data of higher dimensionality.
LSM R-Tree [109]	LSM	✔ ❍ ✔	This study presents a framework for converting indexes including R-Tree into LSM-based indexes.
ER-Tree [110]	LSM	✔ ❍ ✔	ER-Tree embeds an R-Tree in an SSTable and organizes the embedded trees hierarchically.
RUM-Tree [92]	Memo	✔ ❍ ✔	RUM-Tree utilizes the memo mechanism to reduce the update cost by performing lazy deletion.
RUM+-Tree [111]	Memo	❍ ❍ ✔	RUM+-Tree utilizes a hash table to identify the target leaf node for updates and a memo mechanism to transform particular updates into insertions.
LSM RUM-Tree [112]	Memo&LSM	❍ ❍ ✔	LSM RUM-Tree offers methods to reduce the storage overhead associated with Update Memos.

IV. ALGORITHM-SPECIFIC MULTI-DIMENSIONAL INDEXES

In this section, we review the research work on applying algorithms to enhance multi-dimensional index structures. As shown in Tab. III, these algorithms include machine learning methods, the Log Structured Merge Tree (LSM), and innovative update strategies.

A. Machine Learning Methods

Multi-dimensional indexes are typically designed to enhance query efficiency and reduce overall system resource consumption. However, the storage consumption of the index itself also warrants consideration. The application of concepts from one-dimensional learned indexes [4], [5], [113], [114] to multi-dimensional indexes poses greater challenges due to the complex distribution characteristics of multi-dimensional data.

We investigate studies on learned multi-dimensional indexes. Specifically, our analysis of learned multi-dimensional indexes encompasses three key perspectives: (1) Index construction, which details the methods for building an index and the structure of a learned multi-dimensional index; (2) Query processing typically involves approximate methods when dealing with multi-dimensional and spatial data. As such, this presents an opportunity for the application of machine learning techniques; and (3) Update handling, which involves inserting or deleting keys in the index structure while maintaining its stability. We explain how machine learning techniques are applied to various stages of multi-dimensional data indexing, and the summary is shown in Tab. III.

1) Index constructing: Machine learning techniques can be applied to different aspects, such as learning the distribution characteristics of data, learning correlations between attributes of the datasets, or learning query workloads to build indexes. Generally, the use of machine learning in indexing can be categorized into two types [9], [115]. As shown in Fig. 8a, the first category involves integrating machine learning methods into classical indexes, either without any changes or with minor modifications to the original structures. The second category extends the idea of learned one-dimensional indexes, *i.e.*, using a model as an index, leveraging the distribution feature of data, as shown in Fig. 8b.

Tree-based methods. Tree-based learned multi-dimensional indexes employ machine learning techniques to either enhance or completely redesign classical structures like R-Trees and KD-Trees. These studies can be broadly categorized into three groups. The first category concentrates on refining the algorithms of classical indexes, resulting in variants of the original index structures. The second category integrates machine learning models with traditional index structures to form a hybrid index. The third category directly replaces original index structures with machine learning models [9].

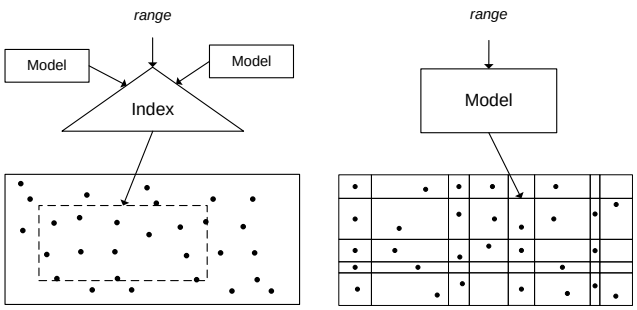


Fig. 8. Two typical approaches to applying machine learning to multi-dimensional indexes: (a) leveraging machine learning model to enhance classical multi-dimensional indexes; (b) training a machine learning model to replace the classical multi-dimensional indexes [9], [115].

i) R-Tree-based learned indexes: In the realm of multi-dimensional index structures, the R-tree is the counterpart to the one-dimensional B-Tree [104]. Several studies have focused on applying machine learning methods to the R-Tree construction process [95], [96]; these approaches make no or minor changes to the classical structures. Therefore, these index structures can be considered as variants of the classical R-Tree, and the strategy of the query algorithm designed for the original R-Tree can also be applied with minor adjustments.

RLR-Tree [95] utilizes reinforcement learning to address problems in index construction. It does not change the structure of the original R-Tree. Therefore, RLR-Tree can also support indexing point data and rectangle objects. As shown in Fig. 9a and Fig. 9b, the two critical operations in the R-Tree construction process are ChooseSubtree and Split, which select a branch for the key to be inserted and provide a strategy for splitting an overflowing node, respectively. The RLR-Tree approaches these operations as a sequential decision-making problem, opting for reinforcement learning methods over traditional heuristic algorithms. Prior to building an R-Tree, the RLR-Tree undergoes offline training with small-sized training datasets. Despite additional computational resources required for offline training, models trained on a small single dataset can be applied to various scenarios. In dynamic update scenarios, the RLR-Tree outperforms various R-Tree variants, demonstrating enhanced performance across datasets with dimensions ranging from 2 to 10.

RW-Tree [96] is another variant of R-Tree, providing a workload-aware construction framework. Similar to R-Tree and RLR-Tree [95], RW-Tree can index both point data and rectangle objects, and the data type of each dimension is numerical. RW-Tree learns query workload features by dividing the space to achieve uniform query distribution, followed by training a learned cost model to estimate these workloads' costs. Note that the cost model is designed to accept only range queries as input, necessitating the conversion of kNN queries into range queries initially. By doing so, the distribution of kNN query workloads can contribute to the cost model. Additionally, RW-Tree employs a strategy that decomposes the cost estimation for a given workload on the R-Tree structure down to each individual node. By leveraging the cost model, RW-Tree can evaluate the performance of queries after an insert operation. Therefore, RW-Tree can select an optimal strategy during the insertion process. The study tests RW-Tree on low-dimensional datasets.

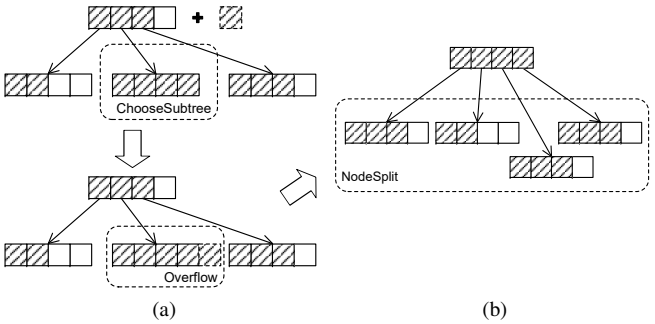


Fig. 9. An illustration of insertion process of R-Tree: The procedure involves two main processes, which are (a) selecting the subtree for insertion and (b) splitting nodes when the insertion node is full, following a specific strategy [95], [96]. For some R-Tree-based learned index, machine learning models can assist these two processes.

As depicted in Fig. 9, both the RLR-Tree [95] and the RW-Tree [96] are specifically designed to tackle decision-making challenges during R-Tree insertion by learning from the distribution of query workloads. These novel indexes are capable of seamlessly adapting to the algorithms utilized in classical R-Trees and their variants, requiring only minor modifications Additionally, there exist other learned indexes based on the R-Tree framework.

"AI+R"-Tree [97] represents a hybrid tree index, integrating three components: an AI-Tree, an R-Tree, and a machine learning model. In the classical R-Tree index, bounding rectangles can overlap with each other, and a query may also overlap with multiple bounding rectangles in one or more nodes, making it challenging to identify inner nodes and leaf nodes that truly contain the objects being searched. To address this, the AI-Tree treats the search process as a multi-label classification task, assigning unique IDs to leaf nodes to serve as classification labels. Moreover, AI-Tree is trained using a multi-model approach, where queries are partitioned into grids, and separate submodels are trained inside each grid cell. Each submodel takes the information of queries that overlap with the corresponding cell as the training data. Additionally, "AI+R"-Tree builds a machine learning model to classify high-overlap queries and low-overlap queries, which is performed as a binary classification task. Following classification, high-overlap queries are processed by the AI-Tree, whereas low-overlap queries are handled by the classical R-Tree algorithm, given the AI-Tree's superior performance with high-overlap queries. Note that the multi-label classification method is not confined to R-Tree indexes alone; it can be effectively applied to other multi-dimensional index structures experiencing overlap issues. "AI+R" indexes spatial data, the type of individual dimension is location data, such as longitude and latitude, with experiments conducted on low-dimensional spatial datasets.

ML-Enhanced High-Dimensional Index [98] introduces a method aimed at enhancing the performance of kNN queries specifically for high-dimensional time series data. This approach employs machine learning methods to boost the efficacy of kNN queries using classical tree indexes like R-Tree [1] and DSTree [116]. The authors asserted that the

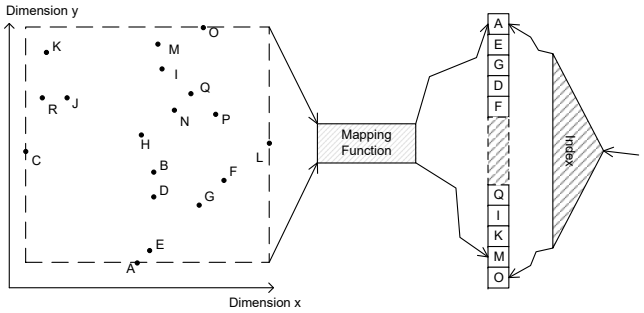


Fig. 10. An illustration of a mapping function: Points on the plane are sorted according to specific rules for index construction [104]. These rules should ensure that points close in multi-dimensional space remain proximate in one-dimensional space. Additionally, the resulting distribution of the sorted data should be simple to approximate.

ML-Enhanced High-Dimensional Index represents a fusion of machine learning techniques with traditional index structures. The approach involves training a Deep Neural Network (DNN) model, which provides a suggested accessing order of leaf nodes to reduce disk access. A key insight of the approach is that the prediction result of whether a kNN target is in the leaf node is fixed after the index is built, allowing the process to be modeled as a multi-class classification task. In addition to a DNN model, the study also presents a quantization-based reordering method. The experimental results demonstrate the effectiveness of this proposed index on high-dimensional datasets with dimensions of 96, 256, and 500.

ii) KD-Tree-based learned indexes: Learned KD-tree [99] treats multi-dimensional indexes as machine learning models by utilizing a multi-stage data processing approach. It can also be used to index word vector data sets, *e.g.*, the GLOVE dataset. This approach conceptualizes nearest neighbor and kNN search as classification problems, introducing a framework that integrates neural networks with the KD-Tree structure. Euclidean distance is used to measure the distance between two data objects. The processing flow of the framework involves several steps: (1) extracting data features to create data vectors, (2) feeding these vectors into a neural network, (3) mapping them to KD-Tree nodes via a fully connected network, and (4) conducting accurate kNN queries in three stages, which are labeling, indexing, and calculation. The Learned KD-Tree is designed for processing high-dimensional data; the experiments are conducted over 50- and 100- dimensional data. However, the efficiency of the Learned KD-tree in processing higher dimensional datasets remains an area for further exploration.

Ordering-based methods. The learned one-dimensional index, RMI [4], models data as a Cumulative Distribution Function (CDF), a process facilitated by sorting one-dimensional datasets. However, sorting multi-dimensional data while preserving its multi-dimensional characteristics is a challenging task, as more than one variable needs to be sorted. To tackle the challenge of constructing learned multi-dimensional indexes, as illustrated in Fig. 10, various studies have concentrated on applying or developing efficient mapping and sorting methods, alongside the creation of more suitable and accurate models for indexing multi-dimensional data [104]. In addition, some multi-dimensional indexes also take the characteristics of workloads into consideration and optimize the mapping or sorting method.

i) Space-filling-based indexes: The space-filling curve provides a method for sorting multi-dimensional data into one-dimensional data, and the Z-ordering curve is one of the most representative curves. However, mapping results using the Z-ordering curve alone is hard to learn [104]. Therefore, some studies try to optimize the space-filling curve for learned multi-dimensional indexes by developing novel frameworks, and there are studies that try to develop new space-filling curves to index multi-dimensional data with machine learning methods.

ZM Index [100] is a multi-dimensional index that employs the Z-ordering curve as its mapping algorithm to transform multi-dimensional data into a one-dimensional format. ZM Index can be used to index point data in Euclidean space; the data type of individual dimension can be numerical. The key idea of the ZM Index lies in its z-address computation algorithm, which assigns a unique z-address to each data point, thereby establishing an exact relative ordering of the multi-dimensional data. The study argues that the accuracy of predicting positions of data is closely related to the model employed, and a CDF model is challenging to learn. To overcome this challenge, the ZM Index implements a Multi-staged Model Index (MMI) designed for efficient approximation of data distribution and minimization of estimation errors. Each model of MMI is an Artificial Neural Network (ANN), which predicts the position of a search key within a range with an error bound. Nevertheless, selecting an appropriate training model for the ZM Index presents several challenges. Therefore, further evaluation of the applicability of Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) is necessary. Additionally, the ZM Index is still a basic version of a learned multi-dimensional index, and there are still issues in terms of supporting updates and other query types. In evaluations conducted with two-dimensional data, the ZM Index demonstrates lower memory overhead and superior query performance compared to the R-Tree.

RSMI [101] presents a multi-stage machine learning methodology for indexing. The index aims to index point data, and the individual dimension is of numerical type. The authors highlighted a challenge with space-filling curves used for sorting data, notably the presence of significant and uneven gaps between adjacent keys. This results in a large number of segments when approximating data by a CDF-based model, which would be inefficient, especially if the scale of data increases. To overcome this challenge, RSMI employs a two-stage approach for learning the distribution of multi-dimensional data. In its first stage, RSMI maps the multi-dimensional data onto an R-Tree rank space [117], forming a grid with the same number of dimensions as the input data. The mapping ensures that there is only one data point for each row and column of the grid. Additionally, in each dimension of a grid, the order of points is the same as the original order of the data. Then, a space-filling curve assigns values to the data in this rank space, and data points are packed into blocks. By this stage, the distribution of gaps between data points becomes more uniform. Therefore, in the following stage, the distribution of data can be modeled more easily. However,

this method creates an $n \times n$ grid, which means the cost of constructing a grid swells significantly as the scale of data grows. To address this problem, RSMI incorporates a recursive partitioning method, ensuring that data within each partition is learned with high accuracy.

To address the challenges of applying space-filling curves (SFC) to learned multi-dimensional indexes, some studies propose the concept of learnable space-filling curve. LMSFC [102] presents constraints for building monotonic SFCs, which can locate a search range faster than non-monotonic ones. This index structure is tailored for indexing point data, with each dimension being of a numerical data type. The construction of LMSFC involves three aspects. Firstly, LMSFC employs Sequential Model-Based Optimization (SMBO) [118] to learn an optimal SFC, aiming to minimize query costs. Secondly, LMSFC provides a cost-based paging mechanism using either a dynamic programming paging method or a heuristic paging method. The heuristic paging method uses a greedy packing algorithm, offering a time-efficient alternative to the dynamic programming approach. Finally, LMSFC leverages query workloads to choose the sort dimension for each page. The dimension that yields the lowest query cost is selected as the sort dimension. Moreover, if a page overlaps with no query, a default order is applied. By the above operations, the data is packed into pages, and then LMSFC applied PGM [5] as the upper structure to index the smallest z-address value of data for a fast search. Additionally, during the online optimization stage, LMSFC implements a query-splitting method to diminish scan overhead and enhance the query efficiency of the index.

ii) Other ordering-based methods: iDistance [119] represents a distinct approach compared to SFC. This method scales multi-dimensional data into a one-dimensional format based on the distance of each point from its nearest reference point.

ML-Index [103] introduces a scaling technique inspired by iDistance. This technique maps nearby multi-dimensional data into a one-dimensional space, placing proximate multi-dimensional points in adjacent positions within a one-dimensional structure. ML-Index utilizes the idea of iDistance by setting reference points in multi-dimensional space and aggregating points near a reference point to form partitions. To address the problem of partition overlap, ML-Index calculates the offset parameter for each region. Then, data points from the multi-dimensional space are mapped and sorted, upon which a recursive learned index is constructed over this sorted array.

Grid-based methods. The basic idea of grid-based indexes involves dividing a spatial area into grid cells of either equal or varying sizes, utilizing machine learning methods for this division. The query process consists of two stages: initially, it involves searching the upper index to locate target cells, followed by a local search within these cells to find the specific data objects. As illustrated in Fig. 11, some learned multi-dimensional indexes [6], [7] create data grids using machine learning methods. Many of these structures develop CDF models across the data space and create adaptive data grids that are tailored to the characteristics of query workloads.

SageDB [104] introduces a concept for projecting k-dimensional data into a one-dimensional format, achieved by segmenting the multi-dimensional data into various data grid layouts. Addressing the challenge of learning data mapped by the Z-ordering curve, SageDB suggests partitioning the data space into uniformly sized cells and devising innovative methods for data projection. Additionally, SageDB introduces three distinct techniques for partitioning multi-dimensional data: employing complex projection methods over static hierarchical partitions, utilizing models for data projection and location, as well as partitioning data based on its distribution and the associated query workloads.

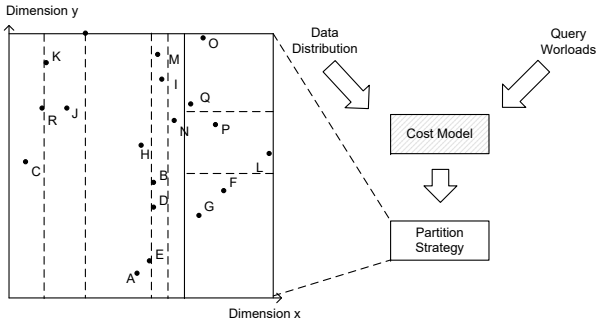


Fig. 11. An overview of the strategy that builds the grid over data: This aspect of the research focuses on data distribution, with the goal of achieving an even distribution of data across each grid cell [6], [7]. Additionally, it accounts for the distribution of query workloads. By tailoring cell division to query distributions, the objective is to minimize the number of cells needed per query, thereby reducing query overhead.

Flood [6] utilizes grid layouts to segment multi-dimensional data. It employs machine learning methods to address several challenges in developing a learned multi-dimensional index. Firstly, Flood models the data distribution by calculating density in one of the multi-dimensions using a CDF, followed by partitioning the data along this target dimension. This process is replicated across other dimensions to create grid cells. Secondly, Flood uses a cost model considering dataset, query workload, and data layout to determine the optimal number of partitions for each dimension. This work tests Flood's performance on synthetic datasets with dimensions less than 18. Flood's query performance is better than that of various indexes, including UB-Tree, KD-Tree, and Z-Order. The experimental results demonstrate Flood's adaptability to shifts in query workloads.

Tsunami [7] addresses drawbacks of Flood [6]. A notable limitation of Flood is its primary focus on uniform workloads. However, in real scenarios, some queries are dense over the specific range of a dimension. To overcome this limitation, Tsunami clusters queries and builds a model to evaluate the degree of skew. It then divides the data space into regions to minimize skew and introduces a "Skew-Tree" for determining partition methods that result in the lowest skew degree. Another limitation of Flood is the lack of consideration for data correlation. To address this issue, Tsunami presents two approaches to balance the number of data within each cell. The first approach performs partition according to one of the dimensions. As data is correlated, the query filter over the target dimension is transformed according to one of the other dimensions. The second approach employs a conditional CDF (X|Y) for partitioning the dimensions. The study conducts experiments to evaluate the applicability of Tsunami over

shifted workloads, and the workload-aware feature of Tsunami is validated. Additionally, the experimental results demonstrate Tsunami's superior performance on datasets with dimensions of 4, 8, 12, 16, and 20.

Both Flood and Tsunami are designed for point data in data space, assuming that the data type of each dimension is a numeric type, for example, timestamps and GPS coordinates.

LISA [105] utilizes grid-based partitioning for numerical data in each dimension and is capable of indexing ImageNet data by converting the ImageNet database contents into vectors. LISA distinguishes itself from other indexes like Flood [6] and Tsunami [7] by storing data on pages, achieved through the construction of shards over grid cells. The first step of constructing LISA is generating cells, and it provides two different methods for different distribution characteristics of data. The partition method in LISA involves dividing the space along each dimension to create a grid. In contrast, for the distribution of data points that have a specific correlation, LISA adopts another partition method. When dividing a dimension, areas obtained by dividing previous dimensions are regarded as a series of independent areas, and the areas are divided separately over the current dimensions to ensure that the number of points in cells is balanced. The second step involves constructing a mapping function to preserve the order of data points. The third step is building shards to cover cells in the grid, where a prediction model is built on the shard ID of a data point. Note that each cell can be part of one or more shards, and similarly, a shard can overlap with multiple cells. The last step is establishing a correspondence between shards and pages. An advantage of this approach is that establishing shards as an intermediate layer delineates a clear relationship between data points and disk pages. Experimental evaluations of LISA were conducted on low-dimensional data with dimensions fewer than 6.

A study [120] evaluates the performance of an index over several spatial partitioning techniques such as Fixed-grid [121], Adaptive-grid [29], KD-tree [14], Quadtree [30], and STR [122]. Data is sorted over one dimension of partitions, and then a learned index, RadixSpline [123], can be applied.

Data correlation-based methods. COAX [106], akin to Tsunami [7], integrates data correlation into its framework. COAX learns from these data correlations. Thus, the complexity of multi-dimensional indexes can be reduced by only indexing a subset of correlated dimensions. Experimental validations of COAX were carried out on low-dimensional datasets with dimensions of 4 and 8.

Interpolation function-based methods. Interpolation is a commonly used method in the field of learned indexes, such as [123], [124]. However, applying the interpolation method to the field of multi-dimensional indexes poses two main challenges. The first challenge involves the selection of an appropriate spatial interpolation method that remains effective as the number of dimensions increases. The second challenge is to ensure that this chosen method can adequately constrain the error bound across various positions.

IF-X [107] is an interpolation function-based index for numerical data dimensions like latitude and longitude. A key challenge in constructing learned multi-dimensional indexes is sorting multi-dimensional data. As shown in Fig. 12, instead of mapping multi-dimensional data into a single dimension through sorting, it selects one dimension's order as the sorting criterion, with the sorting dimension for each leaf node determined independently. Inside a leaf node, an interpolation method is applied to predict the position of data in a sorted array. However, IF-X doesn't consider query workload distribution, which could optimize its sorting dimension selection algorithm. The experimental results demonstrate the performance of IF-X on datasets with dimensions of 2 and 3.

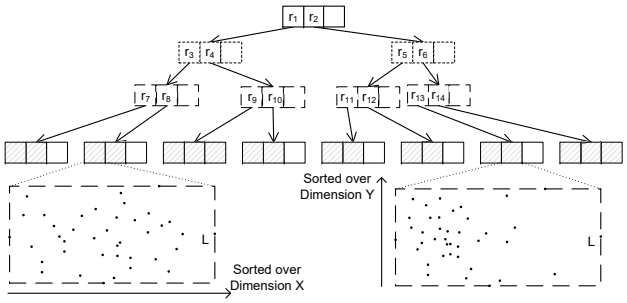


Fig. 12. A brief overview of concepts of the IF-X: IF-X retains the structure of the original index [6], [7]. It can employ the R-Tree as its upper structure. At the leaf node level, data is organized by selecting the best one-dimensional fit from the multi-dimensional distribution. For searches within the leaf node, an interpolation method is employed to predict the data's position.

SPRIG [108] is a spatial interpolation function-based grid index. SPRIG utilizes a spatial interpolation function to approximate the distribution characteristics of multi-dimensional data, locating data points through spatial interpolation. SPRIG can be used to index spatial data, such as the locations of tweets; the data type of each dimension is numerical. To handle range queries and kNN queries, SPRIG constructs an adaptive grid, utilizing a subset of the data as input. Additionally, SPRIG introduces a method to bound the error between the predicted and the actual positions of data points.

While both IF-X [107] and SPRIG [108] utilize interpolation functions, their approaches to constructing indexes are distinct. IF-X can be seen as a compromise in learned multi-dimensional indexing; it builds its model on one dimension within multi-dimensional data, with predictions executed along a single dimension. Note that IF-X's method of selecting a single dimension for sorting remains useful even as the data's dimensionality increases. Besides, SPRIG can be viewed as a multi-dimensional equivalent to a learned one-dimensional index. The approach necessitates suitable interpolation functions as the data's dimensionality expands.

2) Query processing: In this section, we discuss the query algorithms for learned multi-dimensional indexes.

Tree-based query methods. Several multi-dimensional indexes seek to enhance traditional tree-based structures by incorporating machine learning techniques. In query processing, these learned multi-dimensional indexes can benefit from existing algorithms. For example, there are many kNN query algorithms [32], [34], [44], [125], [126] designed for R-Tree. However, the applicability of these algorithms to learned R-Trees warrants discussion, and their performance within this context requires thorough validation.

RLR-Tree [95] maintains the structural integrity of the original R-Tree, implying that query algorithms devised for

R-Trees and their variants can be adapted to RLR-Tree with minimal modifications. The RLR-Tree, which supports range queries, demonstrates superior performance compared to the RR*-Tree [22]. Additionally, RLR-Tree employs the classical kNN query algorithm [32] to process queries, utilizing a variant of Euclidean distance to measure the distance between a point and a rectangle object.

RW-Tree [96] utilizes query workloads to construct and update its R-Tree structure. To leverage the workload of both range query and kNN query, RW-Tree converts kNN queries into range queries. RW-Tree applies the Euclidean distance to partition the space, and based on this, transforms kNN queries into circular area range queries [32].

"AI+R"-Tree [97] presents a hybrid approach for processing range queries. For low-overlap queries, "AI+R"-Tree applies the conventional search algorithms of R-Tree. For high-overlap queries, AI-Tree predicts the potential nodes to be visited and then scans these nodes. However, "AI+R"-Tree does not provide a method for AI-Tree to process kNN queries.

ML-Enhanced Index [98] enhances the efficiency of kNN queries by utilizing a Deep Neural Network (DNN) to predict both the leaf nodes that should be visited and their visiting order. Additionally, to address potential incompleteness in the node set suggested by the DNN, the ML-Enhanced Index implements a strategy that merges this node set with the one derived from the basic search algorithm. Note that the ML-Enhanced Index does not address the applicability of machine learning techniques to enhance range queries.

Grid-based query methods. Grid-based indexes partition the data space into grids and store data points in cells or regions. In query processing, the method involves accessing those cells or regions that contain the relevant data points. The incorporation of machine learning techniques allows for a variety of grid-generating methods, consequently influencing the design of data access algorithms.

While SageDB [104] supports range queries, it does not provide detailed information on its underlying strategy. Flood [6] and Tsunami [7] employ similar query strategies; however, Tsunami enhances Flood's grid-generating approach by introducing the concept of regions. In handling range queries, Flood follows a three-step process: identifying cells intersecting the query filter, determining the physical storage locations of these cells, and then scanning for the target data points. Tsunami adopts the region concept, segmenting the space into multiple areas according to the characteristics of query workloads over the area. Tsunami also employs a three-step approach: locating regions intersecting with the query, identifying their physical locations, and subsequently scanning for data points within these regions. However, neither Flood nor Tsunami adequately considers kNN queries; although Flood is supposed to support kNN queries [6], the detail is not presented.

LISA [105] processes queries using a grid layout but with a distinct approach. It builds shards between cells and pages. In processing range queries within a two-dimensional space, LISA treats the query filter as a rectangle intersecting cell boundaries. Thus, a query rectangle is divided into smaller rectangles that are formed by the overlapping part of the query rectangle and cells, and then the boundary point (bottom-left and upper-right) can be calculated. The prediction model is then used to find the shards of the corresponding rectangles, and the pages of the data points can be accessed. For kNN queries, LISA employs a method that transforms these queries into a sequence of range queries. This process utilizes a lattice regression model [127] to estimate the bounds for each range query. The size of each range query is incrementally expanded until the k nearest neighbors are identified. The distance between the query point x and the k-th nearest key x', denoted as δ, is measured by Euclidean distance.

Space-filling curve-based query methods. Space-filling curve-based query methods are commonly employed in multi-dimensional indexes, wherein data objects are sorted along a space-filling curve. The primary aim of space-filling curves is to ensure that adjacent objects in multi-dimensional space remain close or adjacent even when mapped to one-dimensional space. This proximity is advantageous for processing both range and kNN queries.

ZM Index [100] applies the Z-ordering curve to sort the data within two-dimensional space. Processing range queries in the ZM Index involves a two-step approach. In the first step, the location of the start point and end point of the query range are predicted. Then, cells located between these start and end points are accessed along the Z-ordering curve to retrieve data meeting the query filter. A kNN search algorithm should be further discussed for the ZM Index.

RSMI [101] processes range queries by identifying the bounds of a rectangle, the nature of which depends on the type of space-filling curve used (typically defined by the bottom-left and top-right points). It then scans the data blocks within these bounds to identify and filter the target data points. For kNN queries, RSMI adopts a method akin to the kNN query algorithm [32] of the R-Tree. Here, the MINDIST metric, a variant of Euclidean distance, is utilized to calculate the distance between an object O and a query point P. The size of the range is set according to the distribution of data, and it is gradually increased until the k points are located.

LMSFC [102] handles range queries through a two-step approach: firstly, projection, and secondly, scanning. In the projection step, the search range of a query is determined, defined by a start z-address and an end z-address. Note that a monotonic SFC benefits the fast locating process. In the scanning phase, the pages corresponding to both the start and end z-addresses might include false positive pages. To mitigate unnecessary access, LMSFC employs a technique of recursively splitting queries, effectively dividing a range query into multiple segments, thereby bypassing false positive pages.

Interpolation function-based methods. SPRIG is designed to support both range queries and kNN queries. For range queries, SPRIG identifies the cells that intersect with the query rectangle, which is defined by its lower left and upper right points. In handling kNN queries, SPRIG converts the query into a point and a radius. By progressively expanding this radius, SPRIG locates the k nearest neighbors. For distance measurements in space, SPRIG employs the Euclidean distance to determine the proximity between two points.

3) Update handling: Updating the structure of most learned indexes presents a significant challenge. This difficulty arises

because the underlying machine learning models are trained on specific datasets, and any modifications to the index could potentially compromise the model's accuracy.

As shown in Tab. III, among learned multi-dimensional indexes, classical tree-based types like RW-Tree [96] and RLR-Tree [95] support updates. Grid-based indexes like Flood [6] and Tsunami [7] are predominantly read-only. However, Flood presents possible approaches to support updates in future work, such as maintaining gaps or building a delta index [6], [128]. Similarly, Tsunami plans to incorporate a delta index [128] to facilitate updates in future developments. LISA [105] stands out as a particularly practical learned multi-dimensional index with support for updates. In LISA, insertion involves locating the appropriate shard for the key, followed by inserting it into the target page without sorting if the page is empty. Otherwise, if the page is full, the keys in the page and the inserted key are split into the current page and a new page. Similarly, deletion is performed by finding the target page, removing the key, and then freeing the page if it is empty. RMSI [101] supports the updating of the index structure. Insertion is performed by finding the target block and inserting the data into the block, and then, if the block is full, a new block is created. Deletion also starts with finding the target block by a point query. Then, the target point is swapped with the last point of the block and marked as deleted. IF-X [107] utilizes interpolation in leaf nodes, with the study suggesting that updates could be implemented by amalgamating recent methods for learned one-dimensional indexes. LMSFC [102] handles updates by integrating an updatable upper index like ALEX [113]. An insertion is performed by locating the target page and then inserting the data point into the target page if the page is not full. Otherwise, LMSFC splits the page. To maintain performance, the study presents a strategy to perform a learning process after a series of insertions. The study also proposes a strategy, which is maintaining a delta array of each page to handle the overflow of pages. Yet, LMSFC does not provide a specific strategy for deletion operations.

B. Others

Several studies enhance multi-dimensional indexes based on other algorithms, such as LSM and Update Memo. We summarize these studies in Tab. III.

LSM-based multi-dimensional indexes and their variants have been proposed for indexing spatial data [129]. An LSM-based R-Tree, as proposed in [109], is designed to enhance the performance of the classical R-Tree. ER-Tree [110] presents a novel method to combine R-Tree with LSM architecture. ER-Tree simplifies management by using a single SER-Tree to maintain just one SSTable, thereby reducing the overhead associated with index updates. Moreover, the SER-Trees of levels of LSM are organized as a link list. Kim *et al.* [130] conducted a comprehensive evaluation of five LSM-based multi-dimensional indexes. The evaluated indexes are classified into three types [129]. The first type is based on B-Tree: Dynamic Hilbert B+-tree (DHB-tree) [131], Dynamic Hilbert Value B+-tree (DHVB-tree) [131], and Static Hilbert B+-tree (SHB-tree). The other two types of indexes are LSM R-Tree and Spatial Inverted File (SIF).

The RUM-Tree [92], short for R-Tree with Update Memo, enhances the update efficiency of R-Trees in frequently updated scenarios. RUM+-Tree [111] extends this concept by maintaining a hash table for locating the corresponding leaf node quickly and an Update Memo for organizing the information of multiple object versions. Specifically designed for workloads with frequent updates, the LSM RUM-Tree [112] is a variation of the LSM-based R-Tree that integrates both LSM [132] and Update Memo [92] concepts.

V. Summary and Future Work

Our insights on multi-dimensional indexes are derived from two key perspectives: applicability and versatility of index structures. Classical index structures like B-Trees and R-Trees are typically straightforward and widely utilized in practical systems. Novel indexes that incorporate additional algorithm modules into classical structures often benefit from easier integration. The UB-Tree [26], [27] exemplifies this, merging Space-filling Curve (SFC) concepts with B-Trees.

For algorithm-specific multi-dimensional indexes, RLR-Tree [95] and RW-Tree [96] innovate by replacing traditional R-Tree algorithms with learned models. We believe that this type of improved algorithm based on the original index structure is relatively easy to integrate into actual systems. Merging learned indexes like Flood [6] and Tsunami [7] show superior performance but face challenges in integration due to their novel structures and complex algorithms.

In terms of hardware-oriented multi-dimensional indexes, universal designs offer practical advantages. Given the diverse characteristics of various storage devices, index structures must incorporate complex mechanisms and algorithms to effectively accommodate these differing scenarios. We consider that a framework like eFIND [49] could be ideal, with its capability to adapt multiple indexes for use with SSDs. In terms of computing hardware, index structures and algorithms based on parallelization primitives [65], [73] demonstrate adaptability to various forms of hardware, including GPUs and multi-core CPUs. Moreover, parallelization frameworks such as G-PICS [76] show considerable potential in enhancing the performance and capabilities of multiple indexes.

The second aspect of our analysis concerns the functional support provided by indexes, including supported data types and query types, as detailed in Tab. I, II and III. Note that some original studies lack an explicit discussion on query algorithms; however, there are subsequent studies that have addressed this gap by enhancing query support. The tables in this paper summarize major research works, but it should be noted that the query support information presented is not exhaustive, and the "Support" means originally supported or improved by subsequent studies. Primarily, the index structures explored in this survey are tailored for spatial-temporal data, predominantly assuming numerical data types for each dimension. Concerning supported query types, range queries and kNN queries are the most commonly utilized. Some indexes are specifically designed for particular hardware, wherein the primary focus is not as much on the variety of query types. Consequently, the applicability and performance of classical query algorithms on these indexes warrant further

discussion. For hardware-oriented indexes, the discussion on query type support in LB-Grid [51], H-Grid [55], and G-PICS [76] is particularly comprehensive, demonstrating their practical versatility in handling various query types. Regarding algorithm-specific structures, emerging methods like LISA [105] and RSMI [101] provide in-depth discussions on their functionalities. It can be observed that LISA offers a more comprehensive discussion on functional indicators compared to Flood and Tsunami. Meanwhile, both RLR-Tree [95] and RW-Tree [96] are equipped to build indexes over rectangular areas and support the execution of range queries and kNN queries, making their functionalities quite comprehensive.

A. Hardware-Oriented Multi-dimensional Indexes

The diverse bandwidth and reading latency characteristics of storage devices are crucial considerations in index design [133]. The utilization of multiple storage devices, such as DRAM, NVM, and SSD, presents a dual challenge due to their varying reading performances. Concurrently, this variety offers opportunities for constructing hierarchical structures integrating different storage technologies. Consequently, an important direction for future research lies in the design of hybrid structures that are optimized for use with multiple types of hardware. Additionally, there is significant potential in developing adaptable frameworks for multi-dimensional indexes that can be fine-tuned for various hardware combinations.

B. Algorithm-Specific Multi-dimensional Indexes

Learned multi-dimensional indexes face many challenges, such as practical applications, support for multiple query types [9], [115], and the workload-aware ability.

1) Indexes for practical applications: Several challenges need to be addressed to facilitate the practical application. Firstly, a major challenge is that most learned multi-dimensional indexes lack update support [9], [115]. Secondly, as most learned multi-dimensional index structures are maintained in memory, ensuring their persistence in large-capacity storage devices becomes imperative. This can be achieved by leveraging novel storage devices like NVM or traditional ones such as SSDs and hard disks. Effectively addressing these challenges will not only complete the functional spectrum of an index but also significantly extend its application scenarios.

2) Indexes for multiple query types: Most learned multi-dimensional indexes are optimized for range queries; a notable shortcoming is their lack of support for kNN queries. For some learned multi-dimensional index structures, range queries are executed by scanning from a start-point to an end-point, locating objects that meet the query filter. However, the design of a specialized approach for kNN queries remains a necessity. Future research should prioritize the development of index structures uniquely designed for kNN queries, alongside the creation of more efficient algorithms dedicated to these types of queries [9], [115].

3) Indexes with workload-aware ability: For a multi-dimensional index, learning the distribution of query workload is essential, as skewed queries may affect the efficiency of the index. Recent studies [6], [7], [96] have notably concentrated on developing workload-aware indexes. Constructing and maintaining a workload-aware index involves several aspects. According to previous studies, such indexes typically extract query workloads by clustering historical queries into groups based on their patterns. Subsequently, these indexes adjust their operational parameters by evaluating their performance against the identified characteristics of these workloads.

Several studies leverage workload features to build self-driving databases, also known as "AI for DB" [8], [134]. These studies are related to the construction of workload-aware data structures, including multi-dimensional indexes. Research focused on index selection, tuning, and database configuration [135]–[139], which explore methods for extracting various query workload features, could significantly aid in constructing more efficient learned multi-dimensional indexes. Furthermore, acquiring query workload as a priori knowledge is instrumental in optimizing the performance of indexes [107]. Some studies [140], [141] introduce methods to forecast the characteristics of future query workloads, like query arrival rates, while other studies apply forecasting techniques to index advisors. As such, dedicating efforts to workload forecasting is crucial, yet applying machine learning for accurate workload prediction presents significant challenges [96].

Since the applicable scenarios of the above technology are different from those of multi-dimensional indexing, directly applying the existing techniques to multi-dimensional indexes is not promising, which brings challenges and opportunities to propose a new workload-aware framework.

VI. Conclusion

Multi-dimensional indexes support filtering according to multi-key ranges and have a wide range of applications. With the development of hardware and algorithms, there have been developments in the field of multi-dimensional indexes.

In this paper, we first review classical multi-dimensional indexes. Then, we investigate two major developments. The first is the impact of hardware development, including storage devices such as SSD, NVM, and DRAM, as well as computing hardware such as GPU. The second is applying novel algorithms, such as the state-of-the-art learned index methods.

We summarize the current challenges and future directions in the field of multi-dimensional indexing. In terms of hardware-oriented indexes, multi-dimensional indexes have the potential to be optimized by leveraging combined modern hardware. In terms of algorithm-specific indexes, we focus on learned multi-dimensional indexes. Moreover, we believe query workload is closely related to building efficient learned multi-dimensional indexes, and thus, the prediction and utilization of query workload need to be further studied.

References

[1] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD*, 1984, pp. 47–57.
[2] T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects," in *VLDB*, 1987, pp. 507–518.
[3] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *SIGMOD*, 1990, pp. 322–331.
[4] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *SIGMOD*, 2018, pp. 489–504.

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

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

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

[8] X. Zhou, C. Chai, G. Li, and J. Sun, "Database meets artificial intelligence: A survey," *TKDE*, vol. 34, no. 3, pp. 1096–1116, 2022.

[9] A. Al-Mamun, H. Wu, and W. G. Aref, "A tutorial on learned multi-dimensional indexes," in *SIGSPATIAL*, 2020, pp. 1–4.

[10] H. Singh and S. Bawa, "A survey of traditional and mapreducebased spatial query processing approaches," *SIGMOD Rec.*, vol. 46, no. 2, pp. 18–29, 2017.

[11] C. Zhang, W. Xiao, D. Tang, and J. Tang, "P2p-based multidimensional indexing methods: A survey," *JSS*, vol. 84, no. 12, pp. 2348–2362, 2011.

[12] B. C. Ooi, R. Sacks-Davis, and J. Han, "Indexing in spatial databases," *Unpublished/Technical Papers*, 1993.

[13] A. R. Mahmood, S. Punni, and W. G. Aref, "Spatio-temporal access methods: a survey (2010 - 2017)," *GeoInformatica*, vol. 23, no. 1, pp. 1–36, 2019.

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

[15] H. Samet, "The quadtree and related hierarchical data structures," *CSUR*, vol. 16, no. 2, pp. 187–260, 1984.

[16] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *TOMS*, vol. 3, no. 3, pp. 209–226, 1977.

[17] J. T. Robinson, "The k-d-b-tree: A search structure for large multidimensional dynamic indexes," in *SIGMOD*, 1981, pp. 10–18.

[18] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter, "Bkd-tree: A dznamic scalable kd-tree," in *SSTD*, vol. 2750, 2003, pp. 46–65.

[19] G. R. Hjaltason and H. Samet, "Ranking in spatial databases," in *SSD*, vol. 951, 1995, pp. 83–95.

[20] D. Meagher, "Geometric modeling using octree encoding," *Comput. Graph. Image Process.*, vol. 19, no. 2, pp. 129–147, 1982.

[21] H. Samet, *Foundations of multidimensional and metric data structures*, ser. Morgan Kaufmann series in data management systems. Academic Press, 2006.

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

[23] P. Bozanis and P. Foteinos, "Wer-trees," *DKE*, vol. 63, no. 2, pp. 397–413, 2007.

[24] P. Goyal, J. S. Challa, D. Kumar, A. Bhat, S. Balasubramaniam, and N. Goyal, "Grid-r-tree: a data structure for efficient neighborhood and nearest neighbor queries in data mining," *Int. J. Data Sci. Anal.*, vol. 10, no. 1, pp. 25–47, 2020.

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

[26] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer, "Integrating the ub-tree into a database system kernel," in *VLDB*, 2000, pp. 263–272.

[27] R. Bayer, "The universal b-tree for multidimensional indexing: general concepts," in *WWCA*, vol. 1274, 1997, pp. 198–209.

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

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

[30] R. A. Finkel and J. L. Bentley, "Quad trees: A data structure for retrieval on composite keys," *ACTA*, vol. 4, pp. 1–9, 1974.

[31] E. Schubert, A. Zimek, and H. Kriegel, "Geodetic distance queries on r-trees for indexing geographic data," in *SSTD*, vol. 8098, 2013, pp. 146–164.

[32] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *SIGMOD*, 1995, pp. 71–79.

[33] A. Corral and J. M. Almendros-Jiménez, "A performance comparison of distance-based query algorithms using r-trees in spatial databases," *Inf. Sci.*, vol. 177, no. 11, pp. 2207–2237, 2007.

[34] J. Kuan and L. Paul, "Fast k nearest neighbour search for r-tree family," in *ICICS*, vol. 2, 1997, pp. 924–928.

[35] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," 1966.

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

[37] H. Sagan, *Hilbert's Space-Filling Curve*. Springer New York, 1994, pp. 9–30.

[38] D. Hilbert, "Ueber die stetige Abbildung einer Line auf ein Flächenstück," *Mathematische Annalen*, vol. 38, no. 3, pp. 459–460, 1891.

[39] R. Zhang, J. Qi, M. Stradling, and J. Huang, "Towards a painless index for spatial objects," *TODS*, vol. 39, no. 3, pp. 19:1–19:42, 2014.

[40] D. Lea, "Digital and hilbert k-d trees," *IPL*, vol. 27, no. 1, pp. 35–41, 1988.

[41] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, *Dynamic Versions of R-trees*. London: Springer London, 2006, pp. 15–34.

[42] H. V. Jagadish, "Analysis of the hilbert curve for representing two-dimensional space," *IPL*, vol. 62, no. 1, pp. 17–22, 1997.

[43] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of the hilbert space-filling curve," *TKDE*, vol. 13, no. 1, pp. 124–141, 2001.

[44] K. L. Cheung and A. W. Fu, "Enhanced nearest neighbour search on the r-tree," *SIGMOD Rec.*, vol. 27, no. 3, pp. 16–21, 1998.

[45] C. Wu, L. Chang, and T. Kuo, "An efficient r-tree implementation over flash-memory storage systems," in *GIS*, 2003, pp. 17–24.

[46] M. Pawlik and W. Macyna, "Implementation of the aggregated r-tree over flash memory," in *DASFAA*, vol. 7240, 2012, pp. 65–72.

[47] N. Wang, P. Jin, S. Wan, Y. Zhang, and L. Yue, "Or-tree: An optimized spatial tree index for flash-memory storage systems," in *ICDKE*, vol. 7696, 2012, pp. 1–14.

[48] Y. Lv, J. Li, B. Cui, and X. Chen, "Log-compact r-tree: An efficient spatial index for SSD," in *DASFAA*, vol. 6637, 2011, pp. 202–213.

[49] A. C. Carniel, G. Roumelis, R. R. Ciferri, M. Vassilakopoulos, A. Corral, and C. D. de Aguiar Ciferri, "Porting disk-based spatial index structures to flash-based solid state drives," *GeoInformatica*, vol. 26, no. 1, pp. 253–298, 2022.

[50] A. Fevgas and P. Bozanis, "Grid-file: Towards to a flash efficient multi-dimensional index," in *DEXA*, vol. 9262, 2015, pp. 285–294.

[51] Athanasios Fevgas and Panayiotis Bozanis, "LB-Grid: An SSD efficient Grid File," *DKE*, vol. 121, pp. 18–41, 2019.

[52] A. Fevgas, L. Akritidis, M. Alamaniotis, P. E. Tsompanopoulou, and P. Bozanis, "A study of r-tree performance in hybrid flash/3dxpoint storage," in *IISA*, 2019, pp. 1–6.

[53] S. Cho, W. Kim, S. Oh, C. Kim, K. Koh, and B. Nam, "Failure-atomic byte-addressable r-tree for persistent memory," *TPDS*, vol. 32, no. 3, pp. 601–614, 2021.

[54] B. Lavinsky and X. Zhang, "Pm-rtree: A highly-efficient crash-consistent r-tree for persistent memory," in *SSDBM*, 2022, pp. 4:1–4:11.

[55] A. Fevgas and P. Bozanis, "A spatial index for hybrid storage," in *IDEAS*, 2019, pp. 22:1–22:8.

[56] M. A. Jibril, P. Götze, D. Broneske, and K. Sattler, "Selective caching: a persistent memory approach for multi-dimensional index structures," *DPD*, vol. 40, no. 1, pp. 47–66, 2022.

[57] J. Kim, H. Kang, D. Hong, and K. Han, "An efficient compression technique for a multi-dimensional index in main memory," in *VISUAL*, 2007, vol. 4781, pp. 333–343.

[58] K. Kim, S. K. Cha, and K. Kwon, "Optimizing multidimensional index trees for main memory access," in *SIGMOD*, 2001, pp. 139–150.

[59] H. Kang, J. Kim, D. Kim, and K. Han, "An extended r-tree indexing method using selective prefetching in main memory," in *ICCS*, vol. 4487, 2007, pp. 692–699.

[60] J. Zhou and K. A. Ross, "Buffering accesses to memory-resident index structures," in *VLDB*, 2003, pp. 405–416.

[61] T. Zäschke, C. Zimmerli, and M. C. Norrie, "The ph-tree: a space-efficient storage structure and multi-dimensional index," in *SIGMOD*, 2014, pp. 397–408.

[62] S. Sprenger, P. Schäfer, and U. Leser, "Bb-tree: A main-memory index structure for multidimensional range queries," in *ICDE*, 2019, pp. 1566–1569.

[63] S. Sprenger, P. Schäfer, and U. Leser, "Bb-tree: A practical and efficient main-memory index structure for multidimensional workloads," in *EDBT*, 2019, pp. 169–180.

[64] Y. Lee and C. Chung, "The dr-tree: A main memory data structure for complex multi-dimensional objects," *GeoInformatica*, vol. 5, no. 2, pp. 181–207, 2001.

[65] S. You, J. Zhang, and L. Gruenwald, "Parallel spatial query processing on GPUs using R-trees," in *BigSpatial@SIGSPATIAL*, 2013, pp. 23–31.

[66] S. K. Prasad, M. McDermott, X. He, and S. Puri, "GPU-based Parallel R-tree Construction and Querying," in *IPDPS Workshops*, 2015, pp. 618–627.

[67] J. Kim, S. Kim, and B. Nam, "Parallel multi-dimensional range query processing with r-trees on GPU," *JPDC*, vol. 73, no. 8, pp. 1195–1207, 2013.

[68] L. Luo, M. D. F. Wong, and L. Leong, "Parallel implementation of r-trees on the GPU," in *ASPDAC*, 2012, pp. 353–358.

[69] M. Kim, L. Liu, and W. Choi, "A gpu-aware parallel index for processing high-dimensional big data," *TC*, vol. 67, no. 10, pp. 1388–1402, 2018.

[70] M. Kim, L. Liu, and W. Choi, "Multi-GPU efficient indexing for maximizing parallelism of high dimensional range query services," *TSC*, vol. 15, no. 5, pp. 2910–2924, 2022.

[71] J. Kim and B. Nam, "Co-processing heterogeneous parallel index for multi-dimensional datasets," *JPDC*, vol. 113, pp. 195–203, 2018.

[72] C. Li, Y. Gu, J. Qi, J. He, Q. Deng, and G. Yu, "A GPU Accelerated Update Efficient Index for kNN Queries in Road Networks," in *ICDE*, 2018, pp. 881–892.

[73] J. Zhang, S. You, and L. Gruenwald, "Data parallel quadtree indexing and spatial query processing of complex polygon data on gpus," in *ADMS@VLDB*, 2014, pp. 13–24.

[74] Z. Deng, L. Wang, W. Han, R. Ranjan, and A. Y. Zomaya, "G-ML-Octree: An Update-Efficient Index Structure for Simulating 3D Moving Objects Across GPUs," *TPDS*, vol. 29, no. 5, pp. 1075–1088, 2018.

[75] H. Doraiswamy, H. T. Vo, C. T. Silva, and J. Freire, "A GPU-based index to support interactive spatio-temporal queries over historical data," in *ICDE*, 2016, pp. 1086–1097.

[76] Z. N. Lewis and Y. Tu, "G-PICS: A Framework for GPU-Based Spatial Indexing and Query Processing," *TKDE*, vol. 34, no. 3, pp. 1243–1257, 2022.

[77] Z. Nouri and Y. Tu, "GPU-based parallel indexing for concurrent spatial query processing," in *SSDBM*, 2018, pp. 1–12.

[78] A. Fevgas, L. Akritidis, P. Bozanis, and Y. Manolopoulos, "Indexing in flash storage devices: a survey on challenges, current approaches, and future trends," *VLDB J.*, vol. 29, no. 1, pp. 273–311, 2020.

[79] S. Kargar and F. Nawab, "Extending the lifetime of NVM: challenges and opportunities," *VLDB*, vol. 14, no. 12, pp. 3194–3197, 2021.

[80] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao, "Efficient olap operations in spatial data warehouses," in *SSTD*. Springer, 2001, pp. 443–459.

[81] M. Sarwat, M. F. Mokbel, X. Zhou, and S. Nath, "FAST: A generic framework for flash-aware spatial trees," in *SSTD*, vol. 6849, 2011, pp. 149–167.

[82] G. Roumelis, M. Vassilakopoulos, T. Loukopoulos, A. Corral, and Y. Manolopoulos, "The xbr ˆ+ -tree: An efficient access method for points," in *DEXA*, vol. 9261, 2015, pp. 43–58.

[83] A. N. Papadopoulos, Y. Manolopoulos, Y. Theodoridis, and V. Tsotras, *Grid File (and Family)*. Boston, MA: Springer US, 2009, pp. 1279–1282.

[84] D. Broneske, V. Köppen, G. Saake, and M. Schäler, "Accelerating multi-column selection predicates in main-memory - the elf approach," in *ICDE*, 2017, pp. 647–658.

[85] K. Kim, S. K. Cha, and K. Kwon, "Optimizing multidimensional index trees for main memory access," in *SIGMOD*, 2001, pp. 139–150.

[86] S. Hwang, K. Kwon, S. K. Cha, and B. S. Lee, "Performance evaluation of main-memory r-tree variants," in *SSTD*, vol. 2750, 2003, pp. 10–27.

[87] D. R. Morrison, "Patricia—practical algorithm to retrieve information coded in alphanumeric," *JACM*, vol. 15, no. 4, pp. 514–534, 1968.

[88] U. Germann, E. Joanis, and S. Larkin, "Tightly packed tries: How to fit large models into memory, and make them load fast, too," in *SETQA-NLP*, 2009, pp. 31–39.

[89] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (surf)," *CVIU*, vol. 110, no. 3, pp. 346–359, 2008.

[90] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast bvh construction on gpus," in *CGF*, vol. 28, no. 2, 2009, pp. 375–384.

[91] T. Ulrich, "Loose octrees," *Game programming gems*, vol. 1, pp. 434–442, 2000.

[92] Y. N. Silva, X. Xiong, and W. G. Aref, "The rum-tree: supporting frequent updates in r-trees using memos," *VLDB J.*, vol. 18, no. 3, pp. 719–738, 2009.

[93] T. Foley and J. Sugerman, "Kd-tree acceleration structures for a GPU raytracer," in *Graphics Hardware*, 2005, pp. 15–22.

[94] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree GPU raytracing," in *SI3D*, 2007, pp. 167–174.

[95] T. Gu, K. Feng, G. Cong, C. Long, Z. Wang, and S. Wang, "The rlr-tree: A reinforcement learning based r-tree for spatial data," *CoRR*, vol. abs/2103.04541, 2021.

[96] H. Dong, C. Chai, Y. Luo, J. Liu, J. Feng, and C. Zhan, "Rw-tree: A learned workload-aware framework for r-tree construction," in *ICDE*, 2022, pp. 2073–2085.

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

[98] R. Kang, W. Wu, C. Wang, C. Zhang, and J. Wang, "The case for ml-enhanced high-dimensional indexes," in *AIDB@VLDB*, 2021.

[99] Y. Peng, W. Zhou, L. Zhang, and H. Du, "A study of learned KD tree based on learned index," in *NaNA*, 2020, pp. 355–360.

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

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

[102] J. Gao, X. Cao, X. Yao, G. Zhang, and W. Wang, "Lmsfc: A novel multidimensional index based on learned monotonic space filling curves," *CoRR*, vol. abs/2304.12635, 2023.

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

[104] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan, "Sagedb: A learned database system," in *CIDR*, 2019.

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

[106] A. Hadian, B. Ghaffari, T. Wang, and T. Heinis, "Coax: Correlation-aware indexing on multidimensional data with soft functional dependencies," *CoRR*, vol. abs/2006.16393, 2021.

[107] A. Hadian, A. Kumar, and T. Heinis, "Hands-off model integration in spatial index structures," in *AIDB@VLDB*, 2020.

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

[109] S. Alsubaiee, A. Behm, V. R. Borkar, Z. Heilbron, Y. Kim, M. J. Carey, M. Dreseler, and C. Li, "Storage management in asterixdb," in *VLDB*, vol. 7, no. 10, 2014, pp. 841–852.

[110] J. He and H. Chen, "An lsm-tree index for spatial data," *Algorithms*, vol. 15, no. 4, p. 113, 2022.

[111] Y. Zhu, S. Wang, X. Zhou, and Y. Zhang, "Rum+-tree: A new multidimensional index supporting frequent updates," in *WAIM*, vol. 7923, 2013, pp. 235–240.

[112] J. Shin, J. Wang, and W. G. Aref, "The LSM rum-tree: A log structured merge r-tree for update-intensive spatial workloads," in *ICDE*, 2021, pp. 2285–2290.

[113] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. B. Lomet, and T. Kraska, "ALEX: An Updatable Adaptive Learned Index," in *SIGMOD*, 2020, pp. 969–984.

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

[115] Abdullah-Al-Mamun, H. Wu, and W. G. Aref, "A tutorial on learned multi-dimensional indexes (extended)," 2020, https://www.cs.purdue.edu/homes/aref/.

[116] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang, "A data-adaptive and dynamic segmentation index for whole matching on time series," *VLDB*, vol. 6, no. 10, pp. 793–804, 2013.

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

[118] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *LION*, vol. 6683. Springer, 2011, pp. 507–523.

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

[120] V. Pandey, A. van Renen, A. Kipf, J. Ding, I. Sabek, and A. Kemper, "The case for learned spatial indexes," in *AIDB@VLDB*, 2020.

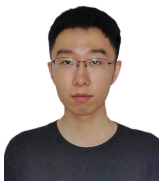
[121] J. L. Bentley and J. H. Friedman, "Data structures for range searching," *CSUR*, vol. 11, no. 4, pp. 397–409, 1979.

[122] S. T. Leutenegger, J. M. Edgington, and M. A. López, "STR: A simple and efficient algorithm for r-tree packing," in *ICDE*, 1997, pp. 497–506.

[123] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "Radixspline: a single-pass learned index," in *aiDM@SIGMOD*, 2020, pp. 5:1–5:5.

[124] A. Hadian and T. Heinis, "Interpolation-friendly b-trees: Bridging the gap between algorithmic and learned indexes," in *EDBT*, 2019, pp. 710–713.

[125] J. S. Challa, P. Goyal, S. Nikhil, S. Balasubramaniam, and N. Goyal, "A concurrent k-nn search algorithm for r-tree," in *COMPUTE*, 2015, pp. 123–128.
[126] A. Papadopoulos and Y. Manolopoulos, "Performance of nearest neighbor queries in r-trees," in *ICDT*, vol. 1186, 1997, pp. 394–408.
[127] E. K. Garcia and M. R. Gupta, "Lattice regression," in *NeurIPS*, 2009, pp. 594–602.
[128] D. G. Severance and G. M. Lohman, "Differential files: Their application to the maintenance of large databases," *TODS*, vol. 1, no. 3, pp. 256–267, 1976.
[129] C. Luo and M. J. Carey, "Lsm-based storage techniques: a survey," *VLDB J.*, vol. 29, no. 1, pp. 393–418, 2020.
[130] Y. Kim, T. Kim, M. J. Carey, and C. Li, "A comparative study of log-structured merge-tree-based spatial indexes for big data," in *ICDE*, 2017, pp. 147–150.
[131] J. K. Lawder, "The application of space-filling curves to the storage and retrieval of multi-dimensional data," Ph.D. dissertation, Birkbeck, University of London, UK, 2000.
[132] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (lsm-tree)," *ACTA*, vol. 33, no. 4, pp. 351–385, 1996.
[133] S. Chockchowwat, W. Liu, and Y. Park, "Automatically finding optimal index structure," *CoRR*, vol. abs/2208.03823, 2022.
[134] G. Li, X. Zhou, J. Sun, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li, "opengauss: An autonomous database system," in *VLDB*, vol. 14, no. 12, 2021, pp. 3028–3041.
[135] V. Sharma, C. E. Dyreson, and N. Flann, "MANTIS: multiple type and attribute index selection using deep reinforcement learning," in *IDEAS*, 2021, pp. 56–64.
[136] G. Li, X. Zhou, S. Li, and B. Gao, "Qtune: A query-aware database tuning system with deep reinforcement learning," in *VLDB*, vol. 12, no. 12, 2019, pp. 2118–2130.
[137] V. Sharma and C. E. Dyreson, "Indexer++: workload-aware online index tuning with transformers and reinforcement learning," in *SAC*, 2022, pp. 372–380.
[138] T. Siddiqui, S. Jo, W. Wu, C. Wang, V. R. Narasayya, and S. Chaudhuri, "ISUM: efficiently compressing large and complex workloads for scalable index tuning," in *SIGMOD*, 2022, pp. 660–673.
[139] X. Zhou, L. Liu, W. Li, L. Jin, S. Li, T. Wang, and J. Feng, "Autoindex: An incremental index management system for dynamic workloads," in *ICDE*, 2022, pp. 2196–2208.
[140] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based workload forecasting for self-driving database management systems," in *SIGMOD*, 2018, pp. 631–645.
[141] Z. Sadri, L. Gruenwald, and E. Leal, "Drlindex: deep reinforcement learning index advisor for a cluster database," in *IDEAS*, 2020, pp. 11:1–11:8.



Mingxin Li received the B.E. degree from Harbin Institute of Technology in 2022. He is currently a Ph.D. student in Nanjing University.

Hancheng Wang received the B.E. degree from Jilin University in 2020. He is currently a Ph.D. student in Nanjing University. His research interests include high-performance query processing and query optimization.

Haipeng Dai (Senior Member, IEEE) received the Ph.D. degree from Nanjing University in 2014. He is an associate professor in Nanjing University. His research interests include data mining and mobile computing. He has published research papers in VLDB, IEEE ICDE, ACM WWW, ACM SIGMETRICS, IEEE INFOCOM, IEEE TKDE, IEEE TPDS, and IEEE TMC. He received Best Paper Award (ICNP'15), Best Paper Award Runner-up (SECON'18), and Best Paper Award Candidate (INFOCOM'17).

Meng Li received the B.S. degree from Nanjing University in 2016. He is currently a Ph.D. student in Nanjing University. His research interests are in the area of high-performance query processing in databases.

Chengliang Chai received the PH.D. degree in computer science and technology from Tsinghua University. He is currently an associate professor in Beijing Institute of Technology, Beijing, China. His research interests include crowdsourcing data management, data mining and database.

Rong Gu (Member, IEEE) received the Ph.D. degree from Nanjing University in 2016. He is a research associate professor in Nanjing University. His research interests include parallel and distributed computing, big data systems. His research papers have been published in many conferences and journals, including IEEE TPDS, IEEE ICDE, IEEE IPDPS, IEEE ICPP, JSA, Parallel Computing, JPDC, and SPE.

Feng chen received the B.S. degree from Hunan University in 2021. He is currently a master student in Nanjing University.

Zhiyuan Chen received the B.S. degree from Xi'an Jiaotong University in 2012. He is currently a senior expert at Huawei Technologies Co., Ltd. He has been working on database engines for nearly ten years.

Shuaituan Li received his M.S. degree from Xidian University in 2012. He is currently affiliated to Huawei Technologies Co., Ltd. His research interests include relational database storage technology.

Qizhi Liu received the Ph.D. degree from Wuhan University in 2001. She is a professor in Nanjing University. Her research interests include information retrieval and data quality services. She has published research papers in ACM SIGIR, DASFAA, and APWeb/WAIM.

Guihai Chen (Fellow, IEEE) received the Ph.D. degree from the University of Hong Kong in 1997. He is a professor and deputy chair of the Department of Computer Science, Nanjing University, China. He had been invited as a visiting professor by many foreign universities, including Kyushu Institute of Technology, University of Queensland, and Wayne State University, USA. His research interests include parallel computing, high-performance computing, and data engineering.