A Learned Approach to Index Algorithm Selection

***Abstract*—The recent surge in learned index algorithms, alongside traditional indexes, has greatly diversified indexing options to support query processing in databases. Despite the rapid expansion of learned indexes, there remains a significant gap in tools for index algorithm selection. Traditional research on index selection has largely focused on recommending which columns to index, as the choice between algorithms like B+tree or hash index was once straightforward. This was managed through basic rules or experiential judgment, given the historically limited options. However, this approach is inadequate today, due to the growing diversity and complexity of index algorithms. In this paper, we introduce a Learned INDex Algorithm Selector, LINDAS. Taking a learned approach, LINDAS uniquely focuses on automatically selecting the most suitable *index algorithm* for a specific column, that satisfies diverse performance objectives in a wide range of applications. We explore the design space of LINDAS, employing a carefully designed featurization approach to capture both data- and workload-specific characteristics with attention mechanisms, as well as the meta-features of index algorithms. Two variants of LINDAS are designed to cater to diverse scenarios and adapt readily to new datasets, workloads, and emerging index algorithms. Comprehensive evaluations of LINDAS across various datasets and workloads demonstrate its effectiveness and superiority compared to applicable baselines.**

***Index Terms*—Algorithm selection, Featurization, Adaptation**

I. Introduction

Background. Indexing is one of the primary approaches to accelerate query processing and improve the performance of Database Management Systems (DBMS) [1], [2]. For decades, B+trees have been a staple for indexing in almost every DBMS [3]–[5], and also the best choice for range requests [1], [6]. Recent innovations in index algorithms, known as learned indexes [6]–[8], have exhibited improved query performance and less storage overhead by exploiting the benefit of data distributions, and they hold the promise to replace or enhance traditional index structures like B+trees [2], [9]–[11]. The landscape of learned indexes is continuously evolving, giving rise to a broad spectrum of index algorithms each optimized for specific usage scenarios.

Motivation. While these developments provide significant advantages, the proliferation of index algorithms introduces substantial challenges to the field. Firstly, comprehensive experimental evaluations in the literature have shown that no single index algorithm is universally optimal for all scenarios [2], [9]–[11]. Furthermore, many recent index algorithms are data-dependent, meaning their performance varies across different datasets. Additionally, there is a notable lack of tools that can accurately predict the performance of each index for specific datasets and workloads ahead of time. As a result, practitioners often struggle to select the most appropriate index algorithm for various application scenarios.

Driven by the need for a reliable method to choose the most appropriate index algorithm across diverse environments, we initiate the study on the problem of Index Algorithm Selection (IAS) which aims to automate the process of selecting index algorithms, taking into account the characteristics of datasets, workloads, and constraints (such as a limited storage budget), as well as the spectrum of index algorithms, including both traditional and learned options.

This work differs significantly from the classical Index Selection (IS) problem, which focuses on recommending the optimal index configuration. IS primarily addresses which columns to index, typically with a predetermined algorithm such as the B+tree [1], [5], thus answering the question of "Where to build indexes" [12]–[14].

In contrast, our study seeks to determine "Which index algorithm to use" for a given column. As a pioneering effort in this area, we concentrate on one-dimensional indexes to facilitate a focused and detailed analysis. This approach aligns with much of the existing research on index algorithms, including both traditional studies and recent advancements in learned indexes [6]–[8], [15]–[23]. It is important to note, however, that the methodologies we propose here have the potential for wider applicability beyond the scope of this initial investigation.

Methodology. To facilitate effective IAS across datasets, workloads, index algorithms, and diverse applications, an effective approach to IAS is expected to demonstrate the following properties:

- *Versatile.* In diverse application scenarios, performance priorities vary significantly. For instance, real-time applications prioritize high throughput, whereas applications constrained by storage capacity emphasize minimizing index size [3], [24]. Therefore, it is essential for IAS methods to offer a flexible mechanism enabling practitioners to designate specific performance objectives.
- *Extensible.* The IAS methods must be designed with high extensibility to easily accommodate emerging index algorithms. This flexibility is crucial, given the rapid pace of advancement in index algorithm research [2], [9]–[11], [25].
- *Adaptive.* To effectively address the complexity and variety inherent in real-world applications, IAS methods must be capable of generalizing to previously unseen datasets and workloads. This adaptability is vital for the practical deployment of these methods in various scenarios.

We investigate the potential of employing Machine Learning (ML) techniques for the task of IAS, addressing the above-mentioned issues. Our proposal includes the development of an end-to-end **L**earned **IND**ex **A**lgorithm **S**elector (**LINDAS**), which is designed to analyze and process features from datasets, workloads, index algorithms, and other related aspects, ulti-

mately recommending an index algorithm expected to deliver optimal performance for the specific dataset and workload in question.

A pivotal aspect of LINDAS is its novel featurization module, which encodes characteristics of datasets, workloads, and index algorithms into a unified representation. This foundational approach underpins the selection of index algorithms, enabling effortless generalization to new datasets and workloads, and facilitating the straightforward integration of new index algorithms, which significantly enhances its extensibility. At the heart of LINDAS lies an ML model tasked with selecting the optimal index algorithm tailored to the specific dataset and workload. To cater to a broad spectrum of application scenarios, we introduce two distinct variants of LINDAS: LINDAS-clf and LINDAS-reg. LINDAS-clf utilizes a classification model to select the best index algorithm based on the encoded features of the dataset and workload alone. In contrast, LINDAS-reg employs a regression model that not only considers the features of the dataset and workload but also integrates the unique characteristics of each index algorithm into its prediction of performance. The algorithm that is expected to yield the best performance is subsequently chosen. This inclusion of index algorithm features in LINDAS-reg allows for a more nuanced prediction of performance, and more importantly, it enables LINDAS-reg to accommodate new index algorithms through their featurization.

To ensure LINDAS's adaptability across diverse environments with distinct requirements and constraints, we design both LINDAS-clf and LINDAS-reg to support customizable performance objectives. This flexibility allows practitioners to tailor the objective of IAS to specific needs, ranging from a single performance measure to a combination of multiple measures weighted according to importance, and even to include specific performance constraints, such as limiting the index size to under 10MB. Recognizing the dynamic and intricate nature of data and workload distributions in real-world applications, it is impractical to cover all potential characteristics of application scenarios during the offline training of LINDAS. To bridge this gap and enhance LINDAS's applicability, we introduce a streamlined method for updating the pre-trained LINDAS model to accommodate new, previously unseen data with unique distributions, particularly when a noticeable decline in performance is detected with new datasets and workloads.

Contributions. Our contributions are summarized as follows:

• We propose **LINDAS**, a learned index algorithm selector. To the best of our knowledge, LINDAS represents the first endeavour specifically dedicated to the selection of index algorithms, accommodating a multitude of objectives and adapting to the continuously expanding array of index algorithms.

• We introduce a novel data featurization method and matrix mechanism designed to encode the distributions of datasets and workloads, enhancing LINDAS's ability to adapt and generalize to new, previously unseen datasets and workloads.

• We propose a highly effective technique for extracting features from index algorithms, markedly enhancing both the training process and the efficiency of index algorithm selection, while also facilitating the incorporation of new index algorithms.

• We devise a streamlined method to dynamically adapt LINDAS to the ever-changing distributions of datasets and workloads, ensuring its robust performance in real-world application scenarios.

• We conduct extensive experiments on 12 real datasets with a variety of workloads to evaluate the performance of LINDAS. Our findings indicate that LINDAS outperforms applicable baselines, achieving a performance improvement of up to 97.9%. Moreover, we highlight LINDAS's strong extensibility, showcasing its capability to integrate new index algorithms that were not available during the training phase.

We release the training data used in our study (detailed in §VII), comprising the execution data collected from experiments conducted across 13 index algorithms, 12 data distributions, and over 30 types of workloads, to support and advance community research in IAS.

The rest of this paper is outlined as follows. Related work is reviewed in §II. We then present the preliminaries and problem statement in §III. We introduce an overview of LINDAS in §IV, followed by §V and §VI that describe details of LINDAS. In §VII, we present the implementation and workflow of LINDAS. §VIII reports on the experimental evaluation of LINDAS. §IX concludes the paper.

II. Related work

This section discusses the related work, including index selection, learned indexes, and techniques for featurization in ML.

Index Selection (IS). The classical IS problem, also known as index tuning or index recommendation, has been actively researched for several decades [14], [26]–[28]. As discussed in §I, IS primarily focuses on recommending the optimal index configuration, i.e., deciding the set of columns to be indexed, typically employing a predetermined index algorithm such as the B+tree, hash index, or R-tree (for multi-dimensional data) [1], [29]. Many tools [30], [31] and ML-based IS methods [12], [13], [32], [33] have emerged for this purpose, which leverage the query optimizer or the actual execution costs to estimate the benefit or regression of an index configuration. However, little attention has been paid to identifying the best index algorithm to use to build the recommended index, primarily because in the past the set of available algorithms at our disposal was very limited. Our work can be considered complementary to the methods for IS, in that we focus on selecting the optimal indexing algorithm (i.e., "what to build") *after* the index configuration (i.e., "where to build") is decided.

Learned Indexes. Learned indexes are proposed to adapt to data distribution to reduce the query latency and storage overhead [6], which has been an active and vibrant area. A learned index usually has a hierarchical structure, and each node in the hierarchy maintains an ML model that predicts the positions of keys [2]. As a side-effect, these algorithms do not have clear, analytically predictable performance (unlike traditional non-learned indexes). Several benchmark studies [2], [9]–[11] have conducted comprehensive experiments to evaluate

the performance of learned indexes and traditional indexes. The results demonstrate that there is no clear winner that can offer higher performance over *all* datasets and workloads. GRE [9] suggests including new features for datasets in the IS tools or query optimizers, but it does not cover how to choose an index algorithm. As a starting point for IAS, TLI [2] provides a set of if-else heuristics as guidance to help practitioners select indexes. However, as will be shown in our experiments, simple heuristics are insufficient and may lead to sub-optimal decisions.

Featurization Methods. Several featurization methods [12], [13], [34] have been proposed in the areas of IS and query optimization to predict query performance. These methods aim to represent the query plan but do not consider the fine-grained characteristics of indexes. For example, the Operator-Table-Predicate representation models multiple operations as a tree-structured vector without index information [34]–[36]. LIB [12] proposes index-optimizable operations to encode the index information into the vector. However, the index information embedded in LIB includes only the type of index (one-dimensional or multi-dimensional) and the order of the indexed columns in the multi-dimensional case, which are insufficient to represent a specific index algorithm.

III. Preliminaries

This section introduces the key concepts and notations used throughout the paper and formally defines the problem studied here.

Dataset. A dataset D to be indexed is a one-dimensional sorted array of *keys*, which could correspond to values in a column of a relational table, keys in a key-value store, etc. In the following discussion, we assume that the keys are unique, but our proposal can naturally generalize to cases with duplicates.

Index algorithms. Let the set of index algorithms under consideration be $\mathcal{A}$. An index algorithm $\mathbb{I} \in \mathcal{A}$ is a data structure or ML model built on top of D and serves as a mapping from key k to p, where p denotes the position of k in D[1], to facilitate efficient processing of operations on D.

Workload. A workload $W_D = \{op_1, op_2, ..., op_n\}$ is a collection of n operations on the dataset D. We simplify the notation W_D as W when there is no ambiguity. We consider operations in the following three categories.

a. Insertion operation $insert(k)$ adds a new key k into D so that $D = D \cup \{k\}$, with dataset D remaining sorted after the insertion;
b. Point query $lookup(k)$ returns the value of the search key $k \in D$ (if any), and NULL otherwise;
c. Range query $range(k_l, k_h)$ returns all those records with keys k falling in range $[k_l, k_h]$, i.e., $result = \{k | k \in D, k \geq k_l \wedge k \leq k_h\}$.

Note that processing deletion operations is similar to processing insertion thus excluded from the workload, as per [2], [10].

In this paper, we refer to the combination of a dataset and a workload as a particular problem *instance*, on which index algorithm selection needs to be conducted.

Performance measure. The literature and practical applications have introduced a diverse array of measures to assess index algorithm performance, among which we primarily consider three measures that are widely used in a variety of scenarios and spread across storage overhead and time cost, namely index size, bulk loading time, and throughput. Note that the analysis and approaches developed herein can naturally be applied to other measures.

Given a particular measure m, we quantify the performance of index algorithm $\mathbb{I}$ on a dataset D and workload W using a function $g(m, \mathbb{I}, D, W)$. We note that in practice a single measure or a combination of measures may be used to indicate the desired performance of the index algorithm, and accordingly we define different **performance requirements**, which form the basis of the following three versions of the optimization problem:

- Single-objective **(S-Obj)**: This setting selects an index algorithm that performs the best on a single m (e.g., select $\mathbb{I}$ with the highest throughput).
- Weighted-objective **(W-Obj)**: This configuration integrates various measures, offering users the flexibility to allocate weights to each according to their specific priorities [37] (e.g., a user with a read-only application may assign a weight of 0.4 to both bulk loading time and throughput, and 0.2 to index size).
- Constrained-Objective **(C-Obj)**: This setting identifies the index algorithm performing the best on a certain m while satisfying one or more specific constraints (e.g., select $\mathbb{I}$ with the highest throughput while requiring the index size less than 500MB).

We next define the research problem studied in this paper.

Definition 1: ***Index Algorithm Selection (IAS).*** Given a dataset D, a workload W, a collection of index algorithms $\mathcal{A}$, and a particular performance requirement, the problem of Index Algorithm Selection *IAS* is to solve the following optimization task [2]:

- S-Obj: For a single objective with measure m:

$$\mathbb{I}^* = \arg\max_{\mathbb{I} \in \mathcal{A}} g(m, \mathbb{I}, D, W) \tag{1}$$

- W-Obj: For a weighted objective with a combination of measures:

$$\mathbb{I}^* = \arg\max_{\mathbb{I} \in \mathcal{A}} g\left(\sum_{i=1}^{u} w_i * m_i, \mathbb{I}, D, W\right) \tag{2}$$

  where w_i is a non-negative weight, $\sum_{i=1}^{u} w_i = 1$, and u is the number of considered measures.
- C-Obj: For a single objective with constraints:

$$\begin{gathered} \mathbb{I}^* = \arg\max_{\mathbb{I} \in \mathcal{A}} g(m_i, \mathbb{I}, D, W) \\ \text{subject to } S = \{g(m_j, \mathbb{I}, D, W)\ \mathrm{p}_j\ \Delta_j\}_{j=1}^{c} \end{gathered} \tag{3}$$

  where m_i and m_j are measures, Δ_j is a threshold value, p_j can be either $\leq$ or $\geq$, and c is the number of constraints.

[1]In cases of duplicate keys, p can either be the first occurrence of k [23], or an arbitrary occurrence of k when a local search range is needed [6], [7], [21].

[2]Without loss of generality, we assume that a higher value of $g(m, \mathbb{I}, D, W)$ indicates better performance.

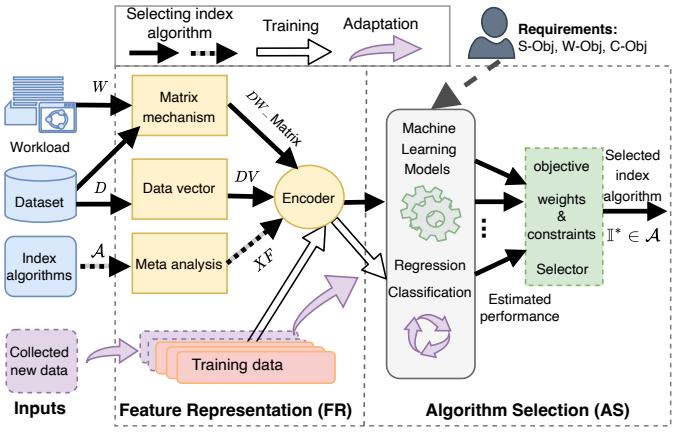


Fig. 1: The overview of LINDAS

IV. LINDAS Framework

In this section, we propose an end-to-end solution for the IAS problem defined in Definition 1, termed Learned Index Algorithm Selector (LINDAS), introducing its framework and the functionality of the primary modules.

To facilitate effective IAS and offer substantial generalizability to new datasets, workloads, index algorithms, and performance requirements, LINDAS consists of two main modules, namely the Feature Representation (FR) and the Algorithm Selection (AS), as illustrated in Fig. 1.

Feature Representation (FR). To enhance the selection of index algorithms for specific problem instances (i.e., datasets and workloads) using ML models, we design a Feature Representation module. This module is tasked with extracting characteristics from the dataset, workload, and index algorithm, and converting these characteristics into vectors suitable for direct input into ML models. We adapt the methodology used in measuring data hardness in [9] and produce a vector representation for the characteristics of a dataset through piecewise linear approximation. Moreover, we devise a matrix mechanism and workload encoding model for the workload representation. To encode the index algorithm, we perform meta-analysis to extract those features of the index algorithm that are relevant to its performance. Then, we aggregate features of the dataset, workload, and index algorithm to produce a representation vector of a specific instance for the IAS problem. Such a representation can then be used as input to the ML models for Algorithm Selection.

Algorithm Selection (AS). The Algorithm Selection module takes the feature vector generated by the FR module and identifies the index algorithm resulting in the best anticipated performance on the given dataset and workload. The AS module utilizes ML models to perform index algorithm selection, which is trained to optimize for performance requirements in the IAS task (defined in Definition 1). As will be introduced in §VI, to better suit the requirements of various objectives and scenarios, we consider two types of models to utilize for AS, namely, classification models and regression models, leading to two variants of LINDAS, referred to as LINDAS-clf and LINDAS-reg respectively. To facilitate the application of LINDAS to practical settings with complex and dynamic data distributions, we design a lightweight adaptive strategy to apply LINDAS to unseen datasets and workloads (detailed in §VI-C).

V. Feature representation

This section illustrates the FR module in LINDAS and discusses the encoding of the dataset, workload, and index algorithm.

A. Dataset Encoding

The first challenge of feature representation is to extract and featurize the dataset, preserving sufficient information regarding the underlying distribution. Recently, GRE [9] proposes to use the number of piecewise linear approximation (PLA) models fitted on dataset D to measure the hardness of D, which has also been used to design learned indexes [21].

More specifically, with PLA (parameterized by ε) the dataset D is partitioned into sub-ranges, and a separate model is trained for each sub-range, achieving ε-approximation [9], [21], as defined below.

Definition 2: ε-approximation [9], [21]. Given a sorted sub-range $\mathscr{a} = [k_1, \cdots, k_{|\mathscr{a}|}]$ and a model M fitted on $\mathscr{a}$, we say M achieves *ε-approximation* if $\forall i \in [1, |\mathscr{a}|], |M(k_i) - i| \leq \varepsilon$, where $M(k_i)$ is the position of k_i predicted by M.

Evidently, different numbers of models are needed to provide *ε-approximation* with different ε values. GRE uses $\varepsilon = 4096$ and $\varepsilon = 32$ to separately characterize the *global* and *local hardness* of the dataset, proposing a metric named data hardness (DH). While DH has been shown to be related to the index performance [9], it is observed in our work that using DH only is insufficient for the problem of IAS (as shown in §VIII-G).

Therefore, we propose a Data Vector representation (denoted by DV) to encode the characteristics of a dataset. The DV also integrates the observation by Ferragina et al. [38] that the robustness and efficiency of learned indexes are closely related to the gaps between consecutive keys and the segment size (i.e., the number of keys in a segment), where each segment is a sub-range as per the partitioning in the PLA.

Particularly, the DV performs *ε-approximation* PLA at a finer granularity and incorporates both segment size distribution and gap distribution. More specifically, with DV, we partition dataset D with $\varepsilon = 2^b$, where $2 \leq b \leq 12, b \in \mathbb{Z}$, and for each ε, we extract five features, namely (1) the number of piecewise linear approximation models PLA_num, (2) the maximal gap between consecutive keys max_gap, (3) the minimal gap between consecutive keys min_gap, (4) the maximal segment size max_size, and (5) the minimal segment size min_size. Furthermore, LINDAS collects the total number of keys (num_keys), the maximal key value (max_key), and the minimal key value (min_key) as global statistics, which are independent of the value of ε. Tab. I presents examples of DV on datasets OSM with 75 million keys and LIBIO with 100 million keys.

By considering both gap distribution and segment size distribution of the underlying dataset D and extracting local as well as global statistics, DV captures more comprehensive information of D, which greatly benefits the IAS process. As will be shown in §VIII-G, DV leads to significant performance gains compared with DH.

Tab. I: Examples of Data Vector representation

dataset	global statistics	PLA_num, max_gap, min_gap, max_size, min_size			
	num_keys, max_key, min_key	$\varepsilon = 4$	$\varepsilon = 8$	...	$\varepsilon = 4096$
OSM	75E+6, 6.09E+18, 3.08E+16	2262741, 2.25E+17, 53146, 335, 3	1083180, 2.25E+17, 686694, 567, 8		2104, 5.76E+17, 7.98E+11, 164235, 8202
LIBIO	10E+7, 5.27E+18, 21335	1203709, 74483, 13, 493, 10	437817, 156348, 41, 1438, 18		296, 14842874, 200351, 3618500, 12173

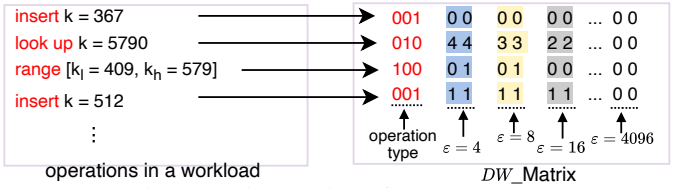


Fig. 2: An illustration of DW_Matrix

B. Workload Encoding

Workload W encapsulates the query patterns that greatly influence the desired properties of the index to be built on D. While existing methods categorize workloads into read-only, write-only, and read-write-mix groups [10], as shown in §VIII-G, they are generally too coarse-grained to be effective for IAS. As such, we propose a novel data distribution-aware approach for obtaining workload representations, called the DW_Matrix mechanism.

1) DW_Matrix Mechanism.: Recall that the DV approach (in §V-A) partitions dataset D into segments, and we label segments in the ascending order of keys with integers $0, \cdots, s$-1. Each operation can then be encoded by the labels of the segments associated with the search key. For example, let Q be a range query with keys k_l and k_h. Assuming k_l and k_h fall into the i-th and the j-th segment respectively, we use ¡i, j¿ to encode Q. For queries with a single search key such as lookup queries and insert queries, we duplicate the search key, i.e., $k_l = k_h$, to ensure consistent vector length. Since DV utilizes 11 different ε values to partition D and thus gives 11 different segment combinations, the search key of each operation can be encoded into a vector of length 22. Additionally, the query type is encoded using a one-hot vector, i.e., "001" for "insert", "010" for "lookup", and "100" for range query, producing an operation representation vector of length 25.

We illustrate the DW_Matrix process in Fig. 2, where operation "insert k=367" is encoded as "001 00 00 00...00", "lookup k= 5790" is encoded as "010 44 33 22...00", and "range query [k_l=409, k_h=579]" is encoded as "100 01 01 00...00".

2) DW_Matrix Transformation.: With the method introduced in §V-B1, we obtain a two-dimensional representation (DW_Matrix) of the workload, with each row corresponding to a particular operation in the workload. We now discuss how to transform the two-dimensional DW_Matrix into a one-dimensional array, so that it can be used together with other feature vectors for IAS. The DW_Matrix transformation consists of the following steps, as illustrated in Fig. 3:

1) **Sampling.** The workload used in our experiments involves 5 million to 60 million operations, resulting in an extremely high feature dimensionality (in the order of millions), which exposes challenges to the ML models utilized in the AS module. We thus conduct a pre-filtering step and only retain the representative operations from the workload, which can effectively reduce

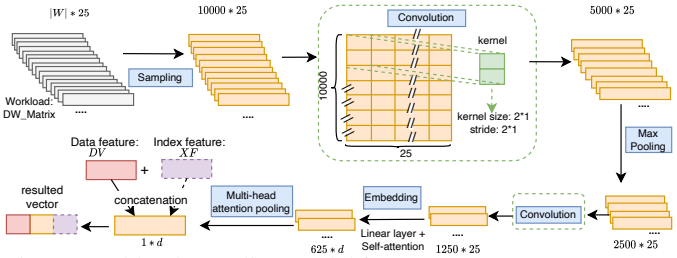


Fig. 3: Workload encoding model for DW_Matrix transformation

the feature dimensionality from millions to thousands. More specifically, we consider three sampling strategies: uniform sampling, systematic sampling, and stratified sampling, with a resolution ranging from 5,000 to 20,000, as will be studied in §VIII-E.

2) **Convolution.** This process utilizes the convolution kernel to extract features from the input DW_Matrix and further reduce the dimensionality of DW_Matrix. The convolution kernel slides along the row axis of the DW_Matrix. After passing through the convolutional layer, the DW_Matirx is abstracted to a feature map. For example, for a DW_Matrix with 10,000 dimensions, a convolution layer with kernel size 2*1 and stride 2*1 generates a feature map with 5,000 dimensions.

3) **Max Pooling.** The max pooling layer reduces the dimensionality of data by retaining the maximum value of each local segment covered by the pooling kernel. The pooling layer helps to reduce the number of parameters in the next layer and improve the efficiency.

4) **Convolution.** We apply another convolution layer after the max pooling layer.

5) **Embedding.** This process first utilizes a linear layer to convert the input vectors $V \in R^{25}$ to $V \in R^{d}$ (d is the embedding size). Then, a self-attention mechanism is applied to extract the relationship among vectors, which corresponds to the interactions of operations within the workload. Self-attention has been proven to efficiently learn high-order interactions among the elements in a set [2].

6) **Multi-head Attention Pooling.** The goal of this process is to further reduce the number of dimensions generated by the embedding layer. Specifically, the multi-head pooling layer aggregates the vectors in the embedded matrix into a single vector, which has been shown to effectively aggregate a set that contains complex interactions among elements [12].

Moreover, LINDAS also maintains the global statistics for a workload, including whether the workload contains range query, the ratio of range queries in the workload, whether the workload contains insertion, the ratio of insertion in the workload, the number of operations executed in parallel, and the total number of operations in a workload. We thus obtain a one-dimensional representation of the workload, which can be combined together with the representations of the dataset and index algorithms, as

will be shown in §VI.

C. Index Algorithms Encoding

We next discuss how to encode index algorithms into vectors. To provide a thorough analysis around the encoding methodology and outcomes, we first list the index algorithms considered in this paper in Tab. II; note that the techniques developed herein can be naturally applied to other index algorithms as well.

Different index algorithms demonstrate varying working mechanisms and may exhibit significantly diverging performance for the same dataset and workload [9]. To facilitate the selection of the most appropriate index algorithm under diverse conditions, it is imperative to characterize and quantify the attributes of each algorithm that significantly influence its performance. We conduct an extensive examination of index algorithms alongside a review of recent benchmarks [2], [9], [10] that assess the effectiveness of learned indexes. Through this analysis, we identify six critical dimensions of index algorithms that exert significant impacts on performance. These dimensions not only play a pivotal role in the performance of index algorithms [2], [10] but also serve as essential considerations in their design. Furthermore, the encoding mechanism is crucial in achieving extensibility to accommodate new index algorithms as detailed in §VI-C. Hereafter, we apply one-hot encoding in featurizing each of these dimensions unless otherwise specified.

1) Distribution-awareness (bits 1-3). First, we categorize index algorithms into two groups, learned indexes and traditional indexes, and allocate one bit to represent this distinction. Next, we recognize the fact that performance of an index is significantly influenced by its distribution-awareness, which includes data-awareness [6], [17] and workload-awareness [7]. We thus allocate two bits to indicate whether a particular index algorithm is data-aware and/or workload-aware respectively. As an example, a learned index that is both data-aware and distribution-aware (such as ALEX [7]) is represented by '111', and a traditional index that is data-agnostic but workload-aware (such as FAST [16]) is encoded as '001'.

2) Insertion strategy (bits 4-16). For representing characteristics of index algorithms related to insertion, we first reserve a bit to indicate whether an index algorithm supports insertion (i.e., dynamic workload) or not. For those index algorithms that can support dynamic workloads, LINDAS further categorizes their insertion strategies into *delta buffer*, *in-place*, and *log structure*, and utilizes the 5-th to the 7-th bits to denote the insertion strategy. Since insertion will cause structure modification operations (SMOs, which are applied to adjust the structure to maintain the desired properties of the index [9]), we integrate the criteria for SMO as features as well, including full-based, threshold-based, cost-based, and conflict-based. The types of SMO are also considered in the encoding process, which consists of buffer-merge, merge-and-train, node-expand-or-split-and-rebuild, subtree-rebuild, and node-split.

3) Lookup operation (bits 17-29). Regarding lookup operations, we consider four factors, including (1) the types of the approximation model used by the learned index, which can be a linear model, a non-linear model, or a hybrid model, and

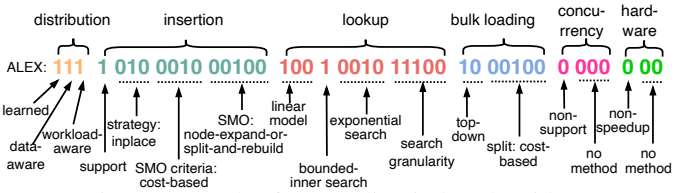


Fig. 4: Examples for encoding index algorithms

thus occupy three bits. For traditional indexes, all three bits are set to '0'; (2) the methods used for inner node search, including bounded-search or exact-location (bit 20); (3) the methods for leaf node search, including linear search, binary search, exponential search, or interpolation search (bits 21-24); (4) search granularity in node (i.e., the number of keys to be searched in a node), which is encoded by a binary sequence of length 5 (bits 25-29).

4) Bulk loading (bits 30-36). We characterize the indexes into three categories depending on how bulk loading is performed in index initialization: *top-down*, *bottom-up*, and *two-phase*. The top-down method first puts the entire dataset in the root node, and then recursively splits the data to child nodes based on some split criterion until no further splits are possible at the leaf level. In contrast, the bottom-up approach first splits data into multiple leaf nodes and then builds the parent nodes level by level until only one node remains which becomes the root. The two-phase strategy is used by DILI [8], which involves building a bottom-up tree and redistributing keys among adjacent nodes. The split strategies used in bulk loading can be further categorized into *greedy-based*, *even-based*, *cost-based*, *conflict-based*, and *trie-based*. We thus allocate bits 26 to 28 to indicate the bulk loading strategy and bits 29-33 to denote different split strategies.

5) Concurrency (bits 37-40). Some indexes can process multiple operations (e.g., lookup, insert) in parallel, while others cannot. To model this distinction, LINDAS uses one bit to encode whether the index algorithm can support concurrent operations or not, and allocates three bits to represent three concurrency control methods, namely optimistic locking [22], [39], RCU-based [22], and ROWEX-based [9], resulting a vector of length 4.

6) Hardware-acceleration (bits 41-43). Both traditional and learned index algorithms aim to take advantage of modern hardware to enhance performance. LINDAS allocates two bits for two types of hardware-acceleration strategies, SIMD instructions and cache-optimized. For example, FINEdex [39] and ART [18] leverage SIMD instructions to boost search performance, while Wormhole [19] leverages cache line prefetching.

Example. We summarize and illustrate the encoding of index algorithm ALEX [7] in Fig. 4.

VI. Algorithm Selection

This section introduces the Algorithm Selection module of LINDAS. To better serve IAS in different scenarios, we propose two variants of LINDAS: LINDAS-clf, LINDAS-reg, which take a classification and a regression approach respectively.

Tab. II: Index algorithms considered in LINDAS

Index algorithms		data-aware	workload-aware	operations	search in node	bulk load	hardware-acceleration	reference
Traditional indexes	FAST	–	✓	read-only	bounded search	bottom-up even-based	SIMD	SIGMOD 2010
	B+tree	–	–	read-write	bounded search	bottom-up even-based	Cache-line*	ACM Compu Surv. 1979 [1
	ART	–	–	read-write, concurrent	bounded search	top-down trie-based	SIMD	ICDE 2013 [1
	HOT	✓	–	read-write, concurrent	bounded search	top-down trie-based	SIMD	SIGMOD 2018
	Wormhole	–	–	read-write, concurrent	bounded search	bottom-up trie-based	Cache-line	Eurosys 2019 [
Learned indexes	RMI	✓	✓	read-only	bounded search	top-down even-based	–	SIGMOD 2018
	MABTREE	✓	–	read-only	bounded search	top-down even-based	–	AIDB@VLDB 20
	ALEX	✓	✓	read-write	unbounded search	top-down cost-based	–	SIGMOD 2020
	LIPP	✓	–	read-write	no search	top-down conflict-based	–	VLDB 2020 [2
	PGM	✓	✓	read-write	bounded search	bottom-up greedy-based	–	VLDB 2020 [2
	DILI	✓	–	read-write	no search	two-phase greedy-based	–	VLDB 2023 [
	XIndex	✓	–	read-write, concurrent	bounded search	top-down even-based	–	PPoPP 2020 [2
	FINEdex	✓	✓	read-write, concurrent	bounded search	bottom-up greedy-based	SIMD	VLDB 2021 [3

* We utilize the widely adopted stx::btree [40], a production quality B+tree implementation that has been optimized for Cache-line efficiency.

A. LINDAS-clf

Choosing the best index algorithm among multiple candidates can naturally be modeled as a classification task, where the candidate index algorithms form the possible model outputs and the model learns a mapping from a particular problem instance (i.e., dataset and workload) to the most suitable index algorithm. We illustrate how LINDAS-clf builds and utilizes classification models for the IAS task in Fig. 5(a).

As demonstrated in Fig. 5(a), we concatenate the dataset representation and the workload representation produced with the methods introduced in §V-A and §V-B respectively, and the resulting vector serves as the input to the LINDAS-clf model, which contains information regarding the problem instance to make index algorithm selection. The possible outputs of LINDAS-clf correspond to the candidate index algorithms.

As highlighted in §III, it is typical for real-world applications to be measured by multiple performance measures, each assigned different levels of importance (i.e., weights). However, when training a classification model to select the optimal index algorithm, it becomes necessary to choose a specific objective (i.e., S-Obj, W-Obj, and C-Obj), which precludes the consideration of alternative objectives during the selection process. Moreover, it is infeasible to develop a distinct classification model for every conceivable combination of measure weights in W-Obj, given the limitless array of potential combinations.

To streamline the process of selecting index algorithms under different measure weighting and reduce the need to train an extensive array of models, we choose a set of representative weight combinations and train a set of base classifiers, each associated with a particular weight combination in this set. When faced with a new IAS task that involves W-Obj, LINDAS-clf first determines the base classifier that most closely aligns with the specified weight combination (e.g., by computing the Euclidean distance between their respective weight arrays). Following this identification, the selected base classifier is employed to make predictions.

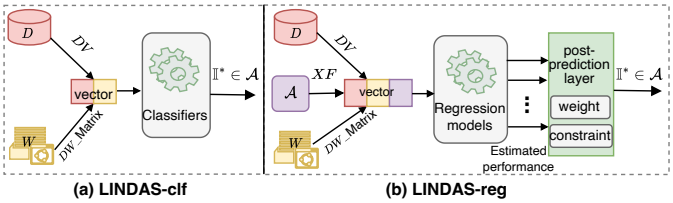


Fig. 5: The structure of LINDAS-clf and LINDAS-reg

B. LINDAS-reg

While LINDAS-clf presents a straightforward approach to selecting index algorithms, it is constrained by two primary limitations. Firstly, its capacity to adapt to new index algorithms is restricted, as it can only recommend algorithms that were included during its training phase. Secondly, the base classifier strategy outlined in §VI-A, despite its ability to manage the selection task for W-Obj to a certain degree, is hindered by its broad categorization. This approach, which relies on a set of predefined weight combinations for training base classifiers, may only be suitable for a narrow range of all conceivable combinations. Moreover, LINDAS-clf cannot explicitly incorporate constraints for C-Obj-oriented selection. Consequently, this could lead to the selection of a less-than-ideal index algorithm, particularly when the selection process involves a wide array of measures or when the number of base classifiers is limited.

To this end, we propose LINDAS-reg, a regression model-based LINDAS variant. Below, we first present the structure of LINDAS-reg and then discuss how it handles IAS.

As shown in Fig. 5(b), LINDAS-reg aims to estimate the

Tab. III: Workload characteristics

Workload	Insert	Query	
		lookup	range query
Pattern	uniform, hotspot, delta	uniform, zipf, hotspot	
Ratio (%)	0, 20, 50, 80	20, 50, 80, 100	
		25, 50, 75	0, 25, 50
Size (M)	5, 10, 20, 40, 60		

performance of an index algorithm on a given dataset and workload, under a particular performance measure. Therefore, in LINDAS-reg, the input to the model is the concatenated vector of the dataset, workload, and index algorithm representations, and the output is the anticipated performance of the corresponding index algorithm.

To support IAS under different performance requirements, LINDAS-reg maintains several regression models, one for each measure supported by LINDAS, and the output of a particular regression model indicates the anticipated performance of the index algorithm in terms of the corresponding measure. LINDAS-reg then utilizes a *post-prediction layer* to identify the optimal index algorithm based on the outputs of one (for S-Obj) or more (for W-Obj and C-Obj) regression models with respect to the performance requirements. For S-Obj, this can be done by simply selecting $\mathbb{I}^*$ with the best performance according to the considered measure. For W-Obj, consider the example where weights 0.2, 0.2, and 0.6 have been assigned to bulk loading time, index size, and throughput respectively, and the post-prediction layer will normalize the output of each regression model into the range [0, 1], calculate the weighted sum of measures for each index algorithm and select the best one. Similarly, for C-Obj, if we assume that the goal is to identify the index algorithm with the highest throughput while ensuring that the index size does not exceed 500 MB, the post-prediction layer prunes index algorithms with size larger than 500 MB and selects from the remaining algorithms the one with the highest estimated throughput.

We note that both LINDAS-clf and LINDAS-reg integrate off-the-shelf ML models for index algorithm selection. We study the performance of different model options in §VIII-C.

C. Model Adaptation

In this section, we discuss how to adapt LINDAS in practical scenarios when the distribution of the dataset or workload changes, or when new index algorithms need to be integrated into LINDAS.

The feature representation module of LINDAS is designed in a way that abstracts the dataset, workload, and index algorithm, and enables rapid adaptation. Specifically, the dataset and index encoding are designed to extract the intrinsic data statistics and meta features of index algorithms; new datasets and index algorithms can thus be seamlessly integrated. The workload encoding model is trained in a self-supervised manner and can be adapted to new distributions via transfer learning [41] to leverage knowledge learned from historical workloads (and from other applications), which also accelerates the training process and enhances the model performance, especially in applications with limited available data. The above properties jointly enable the featurization methods to smoothly adapt to new datasets, workloads, and index algorithms.

When adapting the algorithm selection module, i,e., the classification model in LINDAS-clf and the regression model in LINDAS-reg, to new datasets or workloads, we can use off-the-shelf model adaptation techniques provided by the model implementation. For example, the implementation of tree-based models utilized in our approach offers methods to load a previously built model for continuous training [42], and DNN models can be initialized with pre-trained weights and the parameters (of the whole model or the output layer) can then be refined using the newly collected data [13].

In addition, LINDAS-reg also naturally adapts to new index algorithms. Recall that an index algorithm is encoded into a vector which contains the primary index characteristics such as distribution-awareness and insertion strategy. Given an arbitrary new index $\mathbb{I}_x$, we can follow the same encoding strategy to convert $\mathbb{I}_x$ into a vector and feed it into the LINDAS-reg model together with the representation vectors of the dataset and workload. The anticipated performances of $\mathbb{I}_x$ are then outputted and used by LINDAS-reg.

Another important aspect is identifying when LINDAS's performance begins to decline, necessitating model adaptation to maintain consistent performance. We propose to use *uncertainty score* (such as the R^2 score for regression models and the prediction probability from classification models [43]) as an indicator of distribution drift, which would trigger the model adaptation process. Specifically, model adaptation is initiated when the R^2 score falls below a predetermined threshold or when the prediction probability drops below a certain level. The empirical study in §VIII-F validates the efficacy of employing uncertainty score for automatic model adaptation.

VII. LINDAS Implementation and Workflow

In this section, we discuss in detail the process of building LINDAS as well as utilizing LINDAS for index algorithm selection.

A. Training Data Collection

We next detail the training data collection steps, introducing the datasets, workloads, and index algorithm used in the process.

Datasets. We adopt 12 real-world dataset [2], [9], [11], [25] with varying distributions to evaluate the performance of different index algorithms (as listed in Tab. II). For evaluation purposes, we observe that recent evaluations [2], [9], [25] fix the data size (i.e., the number of keys) as 200 M. In our experiments, to increase the diversity of datasets, and enrich the data distributions in training data, we vary the data size from 50M to 200M.

To quantitatively show the data distributions of different datasets used in LINDAS and the shifts of data distributions, Fig. 6 presents the global data hardness and local hardness [9] of the 12

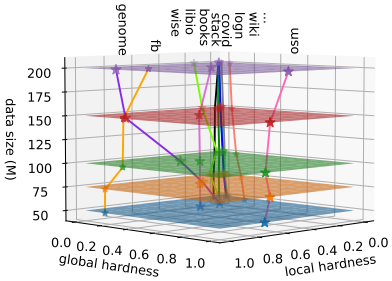


Fig. 6: Data hardness

datasets with different data sizes. Note that we normalize the hardness values using min-max normalization for comparison purposes. As depicted in Fig. 6, different datasets show varying data hardness, and some datasets demonstrate changing data hardness as the data size increases. For example, the global hardness of dataset FB changes at size 100 M, and the local hardness of dataset GENOME grows as the data size increases. Detailed descriptions of the datasets can be found in [44].

Workloads. LINDAS employs synthetic workloads with various patterns. The patterns and ratios of insertion operations, lookup queries, and range queries used in workload generation are shown in Tab. III, from which we can see that the ratio of insert operations contained in a workload ranges from 0% to 80%, and the ratios of lookup and range queries combined are within the range 20% to 100%. The inserted or queried keys are selected from the domain of each above-mentioned dataset based on different patterns (e.g., uniform, zipf). Further, the queries are partitioned into lookups and range queries with different ratios. We vary the range sizes in range queries from 10 to 10^5 as per GRE [9], and categorize sizes below 100 as short range [7] and sizes above 100 as long range, to more thoroughly study the performance of the proposed methods. The insertion patterns include *uniform* [2], [9], *hotspot* [2], and *delta* [2]. The query distributions include *uniform* [9], *zipf* [45], and *hotspot* [46]. We also evaluate the performance of index algorithms with different workload sizes (i.e., the number of operations executed). Additionally, we consider distribution drifts by using different patterns to build and evaluate the model, as detailed in §VIII-F.

Existing benchmarks [2], [9], [10] for learned indexes all conduct experiments with varying workloads to evaluate the performance of learned indexes. To make high-quality index algorithm selection, LINDAS attempts to collect comprehensive training data. Tab. IV depicts the diversity of workload from four dimensions compared with three existing benchmarks (GRE [9], TLI [2] and CLIP [10]). As depicted in Tab. IV, the data collection phase of LINDAS is conducted on workloads with higher coverage than existing benchmarks.

Index algorithms. Tab. II lists all the index algorithms that are currently integrated in LINDAS. As stated in §VI-B, LINDAS can easily incorporate other index structures to achieve extensibility.

As shown in Tab. II, we categorize the index algorithms into traditional indexes and learned indexes as per [2], [9]. Traditional indexes have also been referred to as algorithmic indexes in the literature [20], using the data structure to store the mapping information from key k to position p. In contrast, a learned index applies machine learning models to fit the mapping from k to p, and adopts local search to correctly find the exact position of k.

B. Model Training Workflow

Next, we describe the process of featurizing training data and then discuss the training of models.

Collecting training data involves executing workloads $\mathcal{W}$ on datasets $\mathcal{D}$ with different index algorithms in $\mathcal{A}$. For each execution process, we measure the performance of each index algorithm on the given problem instance (i.e., dataset and workload) with respect to different measures m^i (i.e., bulk loading time, index size, and throughput). Therefore, executing each index algorithm on a particular instance generates three measure values, denoted by m^{BLT}, m^{IS}, and m^T. Meanwhile, LINDAS trains the workload encoding model for DW_Matrix transformation in a self-supervised fashion. Next, the feature representation module is triggered to encode the dataset, workload, and index algorithm into a vector. The training data can thus be represented as $\mathcal{T} = \{< v_i, \{m_i^{BLT}, m_i^{IS}, m_i^T\} >\}_{i=1}^n$, where v_i is the vector associated with a particular dataset, workload, and index algorithm combination, and $\{m_i^{BLT}, m_i^{IS}, m_i^T\}$ denote the corresponding measure values. As the last step, we train the classification models in LINDAS-clf and the regression models in LINDAS-reg. Note that for training the classification model, we identify the optimal index algorithm $\mathbb{I}^*$ on a particular measure or weighted measure combinations for each training data instance, which serves as labels for the training process, while the regression model training uses v_i as inputs and $m_i^{BLT}, m_i^{IS}, m_i^T$ as target values.

Tab. IV: Workload coverage compared with benchmarks

coverage	GRE [9]	TLI [2]	CLIP [10]	LINDAS
query types (lookup, short range, long range)	high	medium	medium	high
query patterns (uniform, hotspot, zipf)	low	low	low	high
insert patterns (uniform, hotspot, delta)	low	high	low	high
concurrent operations (lookup, insert, range)	medium	medium	medium	high

C. Using LINDAS for IAS

Once trained, LINDAS can be invoked to recommend an index algorithm for a given dataset and workload, according to the performance requirement issued by the practitioner.

For LINDAS-clf, the dataset and workload are encoded using the approach proposed in §V, and the resulting vector is then used as input to the classification model of LINDAS-clf, whose output is the selected index algorithm. With LINDAS-reg, for each available index algorithm $\mathbb{I}$, we need to obtain the vector corresponding to the dataset, workload, and $\mathbb{I}$, and use the regression model in LINDAS-reg to obtain the anticipated performance of $\mathbb{I}$. After applying the same process to all candidate indexes, the one leading to the highest performance estimate is then selected.

It is noteworthy that LINDAS does not depend on the particular choice of ML models in Algorithm Selection, allowing for the integration of various ML models that provide high-quality prediction.

VIII. Experiments

In this section, we empirically evaluate the performance of LINDAS and its components. In particular, experiments are conducted from the following aspects: 1) Comparison of LINDAS with baselines; 2) The influence of ML model options on the performance of LINDAS-clf and LINDAS-clf; 3) The impact of the number of base classifiers (base_num) on LINDAS-clf; 4) How different sampling strategies influence

the performance; 5) Model adaptation-related evaluation; 6) Ablation study for feature representation.

A. Experimental Setup

1) Baselines for IAS.: We introduce the baseline methods that can be applied or adapted to index algorithm selection.

Random selection. Selecting an arbitrary index algorithm from candidate index algorithms that is applicable to the corresponding dataset and workload.

TLI selection. Selecting an index algorithm based on the set of if-else rules derived in [2].

GRE selection. Selecting an index algorithm based on the set of rules derived from the evaluation in [9].

We compare LINDAS (including LINDAS-reg and LINDAS-clf) with Random, TLI and GRE selection[3] for S-Obj, W-Obj, and C-Obj as defined in §III.

2) Evaluation.: To assess the relative discrepancy in performance between the best index algorithm and the selected index algorithm, we define the regret measure as follows:

$$regret = 1 - min(\frac{g_{best\_x}}{g_{select\_x}}, \frac{g_{select\_x}}{g_{best\_x}}),$$

Apparently, the range of regret is [0, 1), and the smaller the regret, the better. A regret of 0 indicates that the selected index performs equally well as the optimal one.

The collected data (described in §VII-A) are randomly split into training and test sets with a ratio of 8:2, without overlap; the process is repeated ten times to report the average.

3) Environment and implementation.: **Evaluation environment.** All experiments to evaluate the performance of different index algorithms are conducted on a Ubuntu instance with a 3.6GHz Intel CPU with 256KB L1 cache, 128GB memory (4×32GB). The workload encoding model is trained on a machine with 256GB of memory, and 3 48GB Nvidia A40 GPUs. All the index algorithms are implemented in C++ and the codes are from the open-source publication of original papers (as listed in Tab. II) or the benchmark in TLI [2] and GRE [9]. For index algorithms integrated in LINDAS, we utilize the recommended parameters in each index's original implementation which also give the optimal performance in our evaluation. We report their performance, including bulk loading time, throughput, and index size, from the average of three repetitive runs.

LINDAS implementation. LINDAS is implemented in Python. The workload encoding model is trained using Pytorch [47], and the embedding size used in the workload encoding model is set to $d = 32$. We consider Random Forest (RF), eXtreme Gradient Boosting (XGB), and Multilayer Perceptron (MLP) as model options for LINDAS-clf and LINDAS-reg, which spread across tree-based models, ensemble models, and deep learning models. Scikit-learn implementation [48] is adopted for RF. XGB is implemented in the XGBoost package [42], and the MLP models with four hidden layers are implemented using PyTorch [47]. The MLP in LINDAS-reg utilizes Tanh as the activator and Sigmoid function as the output layer, while the MLP in LINDAS-clf employs Relu as the activator and Softmax being the output layer to predict the class labels. The number of base classifiers (base_num) in LINDAS-clf is set to 16 by default to support W-Obj and we vary base_num in §VIII-D. Additionally, the hyperparameters for models are tuned with Hyperopt [49].

[3]Both TLI and GRE [2], [9] design testbeds to evaluate traditional and learned indexes, then summarize the experimental results to guide practitioners to select proper indexes.

Tab. V: LINDAS-reg: multiple constraints

optimize on	avg_regret	constraints on
throughput	0.028	[index size, bulk load time]
index size	0.072	[throughput, bulk load time]
bulk load time	0.068	[throughput, index size]

Tab. VI: Time efficiency (seconds)

dataset	WISE				GENOME			
data size (M)	50	100	150	200	50	100	150	200
LINDAS	13.21	26.07	39.2	52.22	13.94	28.48	44.07	60.0
Exhaustive	765.3	1063	1545	1927	798.0	1118	1589	3606

B. Comparison with Baselines

We compare LINDAS with Random, TLI, and GRE selection, and the results are shown in Fig. 7.

Fig. 7(a) presents the regrets of different IAS methods, when S-Obj is used as the optimization goal. It is clear that both LINDAS-clf and LINDAS-reg demonstrate lower average regret than Random, TLI and GRE selection on all three measures. Specifically, LINDAS-clf achieves at least 94.2%, 93.7%, and 95.1% lower regret than Random, TLI, and GRE selection, and LINDAS-reg leads to at least 93.3%, 93.8%, and 95.2% reduction in regret respectively. Here the ratio of reduction is calculated by $\frac{\text{regret}_{\text{baseline}} - \text{regret}_{\text{LINDAS}}}{\text{regret}_{\text{baseline}}}$. Interestingly, when the objective includes bulk loading time, TLI and GRE may perform even worse than Random, as they pay little attention to this performance measure.

Fig. 7(b) shows the regret for the case of W-Obj, where the measures in W-Obj are normalized to the same scale, and the weight for each measure is randomly chosen from the range of $[0, 1)$; the weights are normalized such that they sum up to 1. LINDAS-clf and LINDAS-reg exhibit high-quality index algorithm selection, achieving up to 92.3% and 97.9% lower regret than baselines respectively.

Fig. 7(c-e) further illustrate the regret in the case of C-Obj. In each figure, we choose one measure as the optimization objective while setting a constraint on one of the remaining measures. For example, Fig. 7(c) presents the case when index algorithms are expected to maximize throughput, with index size (orange bar) or bulk loading time (green bar) as the constraint. The threshold values in constraints are randomly chosen from the domains of the respective performance measures. As the results show, LINDAS-reg leads to the lowest regret in all settings of C-Obj. The performance of LINDAS-clf is not notably better than Random, TLI and GRE selection, as it cannot explicitly incorporate constraints as conditions.

We further demonstrate the ability of LINDAS-reg to handle multiple constraints, optimizing one measure with constraints on the other two measures. As observed from Tab. V, LINDAS-reg can still yield low regrets, proving its robustness to different performance requirements.

Next, we examine the time efficiency of LINDAS in selecting the optimal index algorithm $\mathbb{I}^*$. Though the rule-based baselines

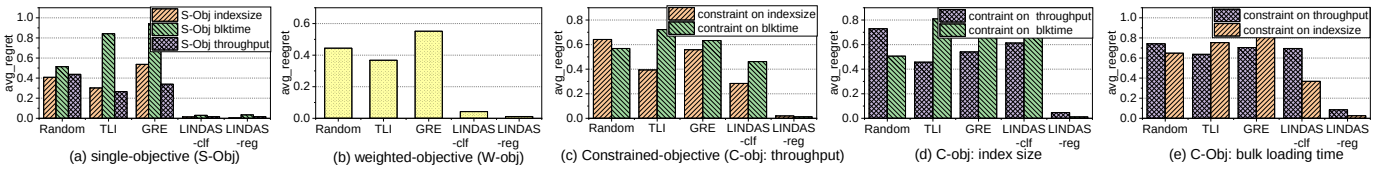


Fig. 7: Comparison with baselines for single-objective, weighted-objective, and constrained optimization

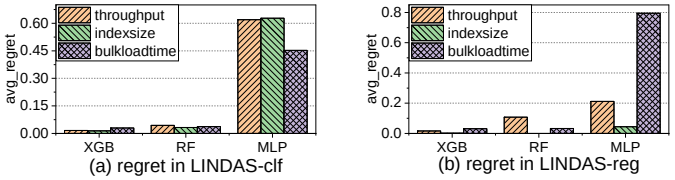


Fig. 8: The regret for different models

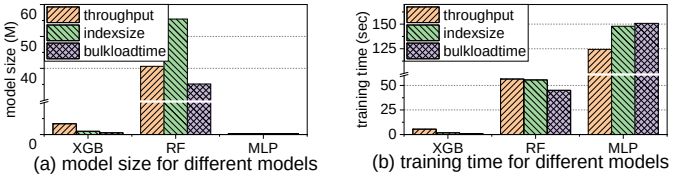


Fig. 9: The model size and training time for different models

in §VIII-A1 are fast, they exhibit high regrets, making them unsuitable for IAS. Therefore, we only show the comparison in selection time (including both featurization and model inference) between LINDAS and exhaustive search, which is guaranteed to find the $\mathbb{I}^*$ with the lowest regret (0) by executing all $\mathbb{I} \in \mathcal{A}$. The results of LINDAS-reg on WISE and GENOME datasets with varying data sizes (i.e., the number of keys) are reported in Tab VI, with similar trends observed on LINDAS-clf and other datasets. LINDAS achieves an average reduction of 98% in time required to identify $\mathbb{I}^*$, an outcome that is impressive given its low regrets (shown in Fig. 7). The approximately linear increase in selection time for LINDAS as data size grows is due to the fact the time to encode the dataset is proportional to its size.

Takeaway. LINDAS is effective in selecting competitive index algorithms that offer high performance on the input instance. While LINDAS-clf is more suited for tasks with S-Obj and W-Obj, LINDAS-reg demonstrates high performance across all settings.

C. Comparing ML Models

This experiment evaluates how different ML model options, including XGBoost, RF, and MLP, influence the performance of LINDAS-clf and LINDAS-reg.

As reported in Fig. 8(a) and (b), the tree-based models have lower regret in all three measures, with XGBoost being the best, for both LINDAS-clf and LINDAS-reg. The MLP shows the highest regret in selecting index algorithms for most of the cases.

Fig. 9(a-b) examine the model size and training time for the models in LINDAS-reg, and LINDAS-clf shows similar trends. While the model size of MLP is the smallest, its regret is higher than that of the other models as shown in Fig. 8(a) and (b). The model size of XGBoost is about 90% less than RF. Fig 9(b) demonstrates that XGBoost also has a much lower training time.

Takeaway. The XGBoost model outperforms others in LINDAS-clf and LINDAS-reg, exhibiting the best prediction performance while maintaining less training time and smaller model size.

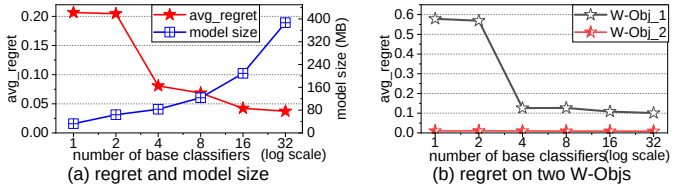


Fig. 10: LINDAS-clf with different number of base classifiers

D. Influence of base_num

Next, we present the empirical study on the influence of the number of base classifiers (i.e., base_num) in LINDAS-clf when serving W-Obj, with 20 randomly chosen weights as described in §VIII-B.

As shown in Fig. 10(a), the average regret decreases rapidly with increasing base_num, reaching diminishing return after base_num=16. Conversely, the model size of LINDAS-clf keeps growing as base_num increases, due to the need to manage a larger number of classification models.

Fig. 10(b) further evaluates the regret on two sets of random weights ([0.637, 0.357, 0.006] and [0.033, 0.153, 0.814] for bulk load time, throughput, and index size respectively), denoted by W-Obj_1 and W-Obj_2 separately. As the figure shows, for W-Obj_1, the average regret is high when a small base_num is used, and decreases rapidly as base_num increases; while for W-Obj_2, the average regret is low even with a small base_num. The reason is that, if LINDAS-clf consists of base classifiers with weights similar to the weights of W-Obj, then the regret is low; otherwise the regret is high and can be reduced by introducing more base classifiers.

Takeaway. We observe that base_num = 16 achieves a good balance between regret and model size. Also, it is beneficial to include the frequently used weights in the base classifiers, when such information is available in prior.

E. Influence of Sampling Options

We next study the impact of sampling strategies and sampling sizes utilized in workload encoding on the performance of LINDAS.

Fig. 11(a) presents the average sampling time when different sampling strategies and sizes are used. Stratified sampling incurs longer time than uniform sampling and systematic sampling, as it requires grouping operations on the operation type in the workload. Another observation is that all the sampling time grows as the sampling size increases but at a slow rate.

Fig. 11(b) displays the training time with systematic sampling when different sampling sizes are used. Similar trends are observed for other sampling methods. Each model is trained

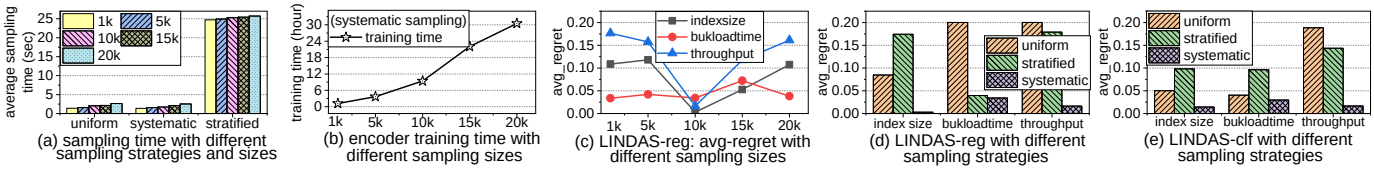


Fig. 11: Impact of sampling strategies and sampling size

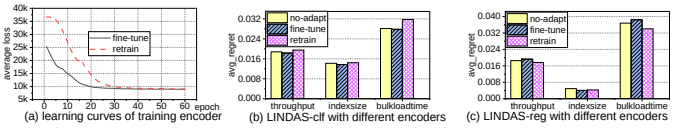


Fig. 12: Evaluation on the adaptation for encoder model

with 60 epochs. The training time increases as the sampling size grows, due to the higher input dimensionality for the workload encoding model.

The average regrets when utilizing systematic sampling with varying sampling sizes are reported in Fig. 11(c). Other sampling strategies show similar trends. The results indicate a rapid drop in regrets for the three measures as the sampling size increases from 1,000 to 10,000. However, the regret begins to increase when the sampling size grows larger than 10,000, due to the increased training difficulty caused by the higher dimensionality.

Fig. 11(d) and (e) further present the regrets of LINDAS-reg and LINDAS-clf when different sampling strategies are used with a sampling size of 10,000. Systematic sampling leads to lower regrets across all measures. This is because systematic sampling preserves the order of operations in the workload during sampling, thereby capturing representative features of the corresponding workload.

Takeaway. A smaller sampling size is usually sufficient to represent the features of workloads, and we recommend setting the sampling size to 10,000 as default. Systematic sampling is preferred for workload encoding, as it incurs lower sampling time and leads to better performance.

F. Model Adaptation

We empirically study the adaptation mechanism of LINDAS and showcase how the workload encoding model as well as the prediction models adapt to new data distributions.

Adaptation of the workload encoding model. As stated in §VI-C, the workload encoding model acts in a self-supervised manner and can be fine-tuned to new workload distributions via transfer learning. To simulate workload distribution changes, in the data collection phase we use training data from half of the workloads for model training, and then either fine-tune or retrain the model using the remaining training data. The training cost and performance of the two adaptation options, as well as the model without adaptation, are reported in Fig. 12. From Fig. 12, we make the following observations: (1) the learning curves of model fine-tuning and model retraining in Fig. 12(a) demonstrate that fine-tuning with transfer learning can converge faster than training the model from scratch; (2) Fig. 12(b) and Fig. 12(c) showcase that model fine-tuning leads to lower regret for LINDAS-clf, while model retraining results in lower regret for LINDAS-reg; (3) the regret of the workload encoding model before adaptation is not significantly higher than that after fine-tuning or retraining, proving its generalization ability to unseen datasets and workloads.

Tab. VII: Training time in LINDAS-reg: adaptation vs. retraining

training time (sec)	zipf	00rq	libio	covid	ALEX	ART
adapt-model	3.78	3.99	3.43	3.49	3.32	3.5
retrain-model	5.41	5.36	5.58	5.58	5.41	5.58

Adaptation of the prediction models. This experiment studies the adaptation of prediction models in LINDAS-clf and LINDAS-reg. To mimic distribution drifts in real scenarios, we reserve one dataset (or workload, index algorithm) for evaluation, and build LINDAS models on the remaining datasets (without overlap). For model adaptation to the reserved dataset, we randomly select 80% samples from the dataset, and follow the steps in §VII-A to generate training data, which is then used to adapt or retrain LINDAS models. The performance of the resulting LINDAS model is then evaluated on the remaining 20% samples of the reserved dataset.

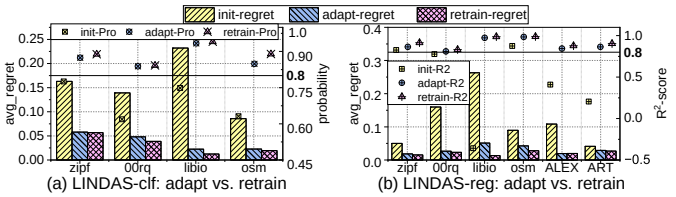
We conduct experiments by reserving each dataset, workload, and index algorithm, and present a subset of results in Fig. 13[4], as other results demonstrate similar trends.

Fig. 13(a) presents the average regret of LINDAS-clf initially trained (init-regret), after adaptation (adapt-regret), and after retraining (retrain-regret). As the results show, model adaptation and retraining can effectively reduce the regret, with retraining leading to higher regret reduction in most cases. The regrets for LINDAS-reg in Fig. 13(b) show similar trends. The scatter plot in Fig. 13(a) showcases the prediction probability of different models. As shown by the plot, the models before adaptation are associated with relatively low prediction probabilities (¡80%), indicating the need for adaptation, and after adaptation or retraining the prediction probability significantly increases. The scatter plot in Fig. 13(b) with R^2-score as the indicator shows similar trends. In our experiments, we observe the threshold of indicator (with probability for LINDAS-clf and R^2-score for LINDAS-reg) set to 0.8 can effectively detect distribution changes and trigger model adaptation when necessary.

The time costs of model adaptation and model retraining in LINDAS-reg on the throughput measure are reported in Tab. VII; similar trends are observed for other measures. The time needed to adapt the model is on average 35% less than that of retraining.

Takeaway. LINDAS supports effective and efficient model adaptation to handle unseen data. In cases when adaptation

[4]Workload '00rq': there are no range queries in the workload, while the ratios of lookups and insertion operations follow the same pattern as in Tab. III.



(a) LINDAS-clf: adapt vs. retrain (b) LINDAS-reg: adapt vs. retrain

Fig. 13: Adaptation for LINDAS-clf and LINDAS-reg

Tab. VIII: LINDAS-clf: ablation on data and workload features

features	(#_features)	regret		
		index size	bulkload time	throughput
DH + w/o DW_Matrix	2	0.24	0.28	0.38
DH + DW_Matrix	41	0.017	0.034	0.018
DV + w/o DW_Matrix	58	0.22	0.22	0.35
DV + DW_Matrix	97	**0.014**	**0.029**	**0.016**

does not significantly reduce the regret, model retraining can be conducted.

G. Ablation Study on Featurization

This experiment conducts ablation studies on the three components in the feature representation module of LINDAS, including the dataset feature DV, workload feature DW_Matrix, and index feature XF. The results are obtained by removing features generated by certain components and measuring the resulting method performance, to evaluate the contribution of each component. We first study the influence of the common features to LINDAS-reg and LINDAS-clf, DV and DW_Matrix, on Tab. VIII and Tab. IX respectively, and then show the influence of XF on LINDAS-reg in Tab. X.

Tab. VIII depicts the regret of LINDAS-clf with different combinations of DH (i.e., the global and local data hardness proposed in [9]), DV and DW_Matrix, w/o DW_Matrx. As shown in the results, DH+w/o DW_Matrix gives the highest regret. The combination of DV and w/o DW_Matrix reduces the regret by a small fraction. The lowest regrets for all the three measures are obtained with feature DV+DW_Matrix, proving the effectiveness of DV and DW_matrix. The results for LINDAS-reg in Tab. IX display similar trends. Note the index encoding (XF) is retained in this experiment.

Tab.X evaluates the impact of XF on LINDAS-reg. The results show all regrets drastically increase when removing the XF[5], proving that XF has a significant influence on the regrets of LINDAS-reg.

Tab. IX: LINDAS-reg: ablation on data and workload features

features	(#_features)	regret		
		index size	bulkload time	throughput
DH + w/o DW_Matrix	45	0.90	0.59	0.87
DH + DW_Matrix	84	0.008	0.034	0.028
DV + w/o DW_Matrix	101	0.76	0.383	0.763
DV + DW_Matrix	140	**0.002**	**0.03**	**0.016**

Tab. X: LINDAS-reg: ablation study on index features

features	(#_features)	regret		
		index size	bulkload time	throughput
w/o XF	98	0.513	0.741	0.189
XF	140	**0.002**	**0.03**	**0.016**

Takeaway. Effective featurization methods play crucial roles in IAS, and the encoding of datasets, workloads, and index features should be given primary consideration as indispensable factors.

IX. Conclusion

In this paper, we propose LINDAS, Learned Index Algorithm Selector, to automate the selection of the most suitable index algorithm for a specific dataset and workload under various performance requirements. LINDAS is the first work to show the benefits of applying ML techniques to the index algorithm selection task. We evaluate LINDAS using a variety of datasets and workloads. The results show LINDAS's superiority in selecting the best-performing index algorithm across different scenarios, demonstrating its adaptability to unseen datasets and workloads, and its extensibility to incorporate the new advanced index algorithms in the research area.

References

[1] D. Shasha and P. Bonnet, *Database tuning: principles, experiments, and troubleshooting techniques*. Elsevier, 2002.

[2] Z. Sun, X. Zhou, and G. Li, "Learned index: A comprehensive experimental evaluation," *VLDB*, vol. 16, no. 8, pp. 1992–2004, 2023.

[3] H. Zhang, D. G. Andersen, A. Pavlo *et al.*, "Reducing the storage overhead of main-memory OLTP databases with hybrid indexes," in *SIGMOD*, 2016, pp. 1567–1581.

[4] A. Silberschatz, H. Korth, and S. Sudarshan, *Database systems concepts*. McGraw-Hill, Inc., 2005.

[5] A. Silberschatz, H. F. Korth, and S. Sudarshan, "Database system concepts," 2011.

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

[7] J. Ding, U. F. Minhas, J. Yu *et al.*, "Alex: an updatable adaptive learned index," in *SIGMOD*, 2020, pp. 969–984.

[8] P. Li, H. Lu, R. Zhu, B. Ding, L. Yang, and G. Pan, "DILI: A distribution-driven learned index," *VLDB*, vol. 16, no. 9, pp. 2212–2224, 2023.

[9] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang, "Are updatable learned indexes ready?" *VLDB*, vol. 15, no. 11, pp. 3004–3017, 2022.

[10] J. Ge, B. Shi *et al.*, "Cutting learned index into pieces: An in-depth inquiry into updatable learned indexes," in *ICDE*, 2023, pp. 315–327.

[11] R. Marcus, A. Kipf, A. van Renen *et al.*, "Benchmarking learned indexes," *VLDB*, vol. 14, no. 1, pp. 1–13, 2020.

[12] J. Shi, G. Cong, and X.-L. Li, "Learned index benefits: Machine learning based index performance estimation," *VLDB*, pp. 3950–3962, 2022.

[13] B. Ding, S. Das *et al.*, "Ai meets ai: Leveraging query executions to improve index recommendations," in *SIGMOD*, 2019, pp. 1241–1258.

[14] J. Kossmann, S. Halfpap *et al.*, "Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms," *VLDB*, vol. 13, no. 12, pp. 2382–2395, 2020.

[15] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[16] C. Kim, J. Chhugani, N. Satish *et al.*, "Fast: fast architecture sensitive tree search on modern cpus and gpus," in *SIGMOD*, 2010, pp. 339–350.

[17] R. Binna, E. Zangerle *et al.*, "Hot: A height optimized trie index for main-memory database systems," in *SIGMOD*, 2018, pp. 521–534.

[18] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *ICDE*. IEEE, 2013, pp. 38–49.

[19] X. Wu, F. Ni, and S. Jiang, "Wormhole: A fast ordered index for in-memory data management," in *EuroSys*, 2019, pp. 1–16.

[20] A. Hadian and T. Heinis, "Madex: Learning-augmented algorithmic index structures." in *AIDB@ VLDB*, 2020.

[5]For option w/o XF, we preserve the bit that represents learned and traditional indexes, as per [2], otherwise the output LINDAS-reg is not associated with an index algorithm.

[21] P. Ferragina and G. Vinciguerra, "The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds," *VLDB*, pp. 1162–1175, 2020.

[22] C. Tang, Y. Wang, Z. Dong, G. Hu, Z. Wang, M. Wang, and H. Chen, "Xindex: a scalable learned index for multicore data storage," in *PPoPP*, 2020, pp. 308–320.

[23] J. Wu, Y. Zhang, S. Chen, Y. Chen, J. Wang, and C. Xing, "Updatable learned index with precise positions," *VLDB*, pp. 1276–1288, 2021.

[24] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte, "The implementation and performance of compressed databases," *SIGMOD*, pp. 55–67, 2000.

[25] A. Kipf, R. Marcus *et al.*, "Sosd: A benchmark for learned indexes," *NeurIPS Workshop MLsys*, 2019.

[26] D. C. Zilio, J. Rao, S. Lightstone *et al.*, "DB2 design advisor: Integrated automatic physical database design," in *VLDB*, 2004, pp. 1087–1097.

[27] D. Dash, N. Polyzotis, and A. Ailamaki, "CoPhy: A scalable, portable, and interactive index advisor for large workloads," *VLDB*, pp. 362–372, 2011.

[28] R. Schlosser, J. Kossmann, and M. Boissier, "Efficient scalable multi-attribute index selection using recursive strategies," in *ICDE*, 2019, pp. 1238–1249.

[29] OceanBase. (2021) Oceanbase. [Online]. Available: https://www.oceanbase.com/docs/oceanbase-database-cn

[30] A. Pavlo, G. Angulo, J. Arulraj *et al.*, "Self-driving database management systems." in *CIDR*, 2017.

[31] S. Chaudhuri and V. Narasayya, "Anytime algorithm of database tuning advisor for microsoft sql server," 2020.

[32] H. Lan, Z. Bao, and Y. Peng, "An index advisor using deep reinforcement learning," in *CIKM*, 2020, pp. 2105–2108.

[33] V. Sharma, C. Dyreson, and N. Flann, "Mantis: Multiple type and attribute index selection using deep reinforcement learning," in *IDEAS*, 2021, pp. 56–64.

[34] R. Marcus, P. Negi, H. Mao *et al.*, "Bao: Making learned query optimization practical," in *SIGMOD*, 2021, p. 1275–1288.

[35] J. Sun and G. Li, "An end-to-end learning-based cost estimator," *VLDB*, vol. 13, no. 3, pp. 307–319, 2019.

[36] R. Marcus, P. Negi *et al.*, "Neo: A learned query optimizer," *VLDB*, vol. 12, no. 11, pp. 1705–1718, 2019.

[37] J. Zhang and Y. Gao, "CARMI: a cache-aware learned index with a cost-based construction algorithm," *VLDB*, pp. 2679–2691, 2022.

[38] P. Ferragina, F. Lillo, and G. Vinciguerra, "Why are learned indexes so effective?" in *ICML*, 2020, pp. 3123–3132.

[39] P. Li, Y. Hua *et al.*, "Finedex: a fine-grained learned index scheme for scalable and concurrent memory systems," *VLDB*, pp. 321–334, 2021.

[40] T. Bingmann. . stx b+ tree. [Online]. Available: https://panthema.net/2007/stx-btree/

[41] F. Zhuang, Z. Qi, K. Duan *et al.*, "A comprehensive survey on transfer learning," *Proceedings of the IEEE*, vol. 109, no. 1, pp. 43–76, 2020.

[42] (2022) Xgboost documentation. [Online]. Available: https://xgboost.readthedocs.io/en/latest/python/python_api.html

[43] Z.-H. Zhou, *Machine learning*. Springer Nature, 2021.

[44] LINDAS. Lindas. [Online]. Available: https://anonymous.4open.science/r/Algorithm-selection-3C00/

[45] J. DeBrabant, A. Pavlo *et al.*, "Anti-caching: A new approach to database management system architecture," *VLDB*, pp. 1942–1953, 2013.

[46] A. Galakatos, M. Markovitch *et al.*, "Fiting-tree: A data-aware index structure," in *SIGMOD*. ACM, 2019, pp. 1189–1206.

[47] P. Team. (2021) pytorch. [Online]. Available: https://pytorch.org/

[48] F. Pedregosa, G. Varoquaux *et al.*, "Scikit-learn: Machine learning in Python," *JMLR*, vol. 12, pp. 2825–2830, 2011.

[49] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *ICML*, 2013, pp. 115–123.