Automatic Index Tuning: A Survey

Yang Wu, Xuanhe Zhou, Yong Zhang, Guoliang Li

***Abstract*—Index tuning plays a crucial role in facilitating the efficiency of data retrieval within database systems, which adjusts index settings to optimize the database performance. Recently, with the growth of data volumes, the complexity of workloads, and the diversification of database applications, various Automatic Index Tuning (AIT) methods have been proposed to address these challenges. In this paper, we provide a comprehensive survey on Automatic Index Tuning. First, we overview the AIT techniques from multiple aspects, including *(i)* problem definition, *(ii)* workflow, *(iii)* framework, *(iv)* index types, *(v)* index interaction, *(vi)* changing factors, *(vii)* automation level, and show the development history. Second, we summarize techniques in the main modules of AIT, including *preprocessing*, *index benefit estimation*, and *index selection*. *Preprocessing* involves workload compression, index candidate generation, feature representation of workloads and databases, and workload reduction. *Index benefit estimation* approaches are categorized into empirical methods and machine learning based methods. *Index selection* involves algorithms of offline AIT and online AIT. Moreover, we summarize the commonly-used datasets in AIT and discuss the applications of index tuning in commercial and opensource database products. Finally, we outline potential future research directions. Our survey aims to enhance both general knowledge and in-depth insights into AIT, and inspire researchers to address the ongoing challenges.**

***Index Terms*—Automatic Index Tuning, Machine Learning, Index Selection, Index Benefit Estimation, Reinforcement Learning**

I. Introduction

INDEXES serve as essential data structures in database systems, which help to accelerate data retrieval by reducing disk I/O operations, at the cost of extra maintenance work and storage space. Traditionally, the task of designing and adjusting indexes has been predominantly carried out by human database administrators (DBAs), relying on empirical theories and their cumulative experience. However, this manual approach is labor-intensive and time-consuming. With the exponential data growth, increasing workload complexities, and widespread adoption of cloud databases, the challenges of manual tuning have become even more significant. Consequently, *Automatic Index Tuning* (AIT) methods have been proposed to alleviate the burden of DBAs [29], [131], [150].

One key advantage of AIT lies in its ability to effectively handle large volumes of data and complex workloads. These methods employ various techniques to analyze database statistics (e.g., data distributions, query selectivities, scan costs) [119], [120] and workload patterns. With this comprehensive analysis, they can make informed decisions regarding the creation or update of indexes to optimize workload execution. Furthermore, in online index tuning, some AIT methods can adapt to changing workloads under limited time threshold [23], [150]. By doing so, AIT minimizes maintenance overhead and ensures the effectiveness of selected indexes.

Yang Wu, Xuanhe Zhou, and Guoliang Li are with the Department of Computer Science, Yong Zhang is with Beijing National Research Center for Information Science and Technology, all at Tsinghua University, Beijing, China. E-mail: {wu-y22, zhouxuan19}@mails.tsinghua.edu.cn, {zhangyong05, liguoliang}@tsinghua.edu.cn
Yang Wu and Xuanhe Zhou are co-first authors and make equal contributions. Corresponding author: Yong Zhang and Guoliang Li.

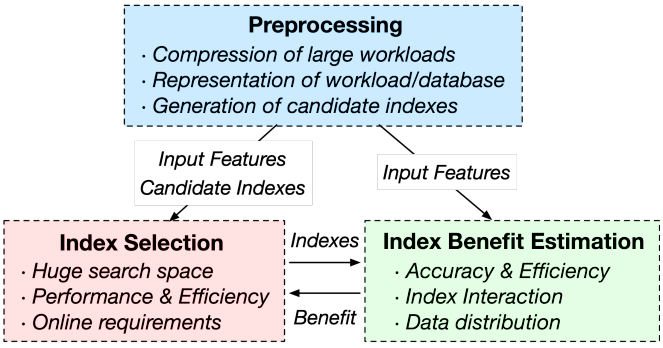

Fig. 1: Challenges of Designing an Automatic Index Advisor

However, it is important to note that there is *no universal solution that guarantees optimal results in all cases* [71]. AIT advisors may not always recommend the desired indexes due to various challenges, e.g. they may derive sub-optimal index configurations due to inaccurate index benefit estimation results. Besides, the choice of different AIT methods depends on the specific scenarios and even the database systems (e.g., some databases do not support hypothetical indexes).

This survey aims to provide a comprehensive overview of AIT techniques, addressing their strengths and suitability for various applications. It is designed for a wide variety of readers. (i) Researchers on AIT can gain insights into the internal mechanisms of index advising tools, and grasp the knowledge of how to design an effective index advisor. (ii) Database administrators (DBA) can learn about various automatic index tuning tools, know how to fix performance problem, and work together with tuning tools. (iii) Cloud service providers can select an automatic index advising algorithm that best fits their business scenario. (iv) Common database users can have a grasp of the features of the automatic index tuning tools for commercial or open-source database products, and choose the database platform they are going to use.

Through extensive literature review, we summarize three main challenges for designing an index advisor (Figure 1).

A. Challenges of Designing an Automatic Index Advisor

The first challenge is *how to judiciously prepare the index tuning features and candidate indexes*. First, for workloads with an extensive number of queries, it is important but tricky to compress the workload, i.e. select a subset of queries that

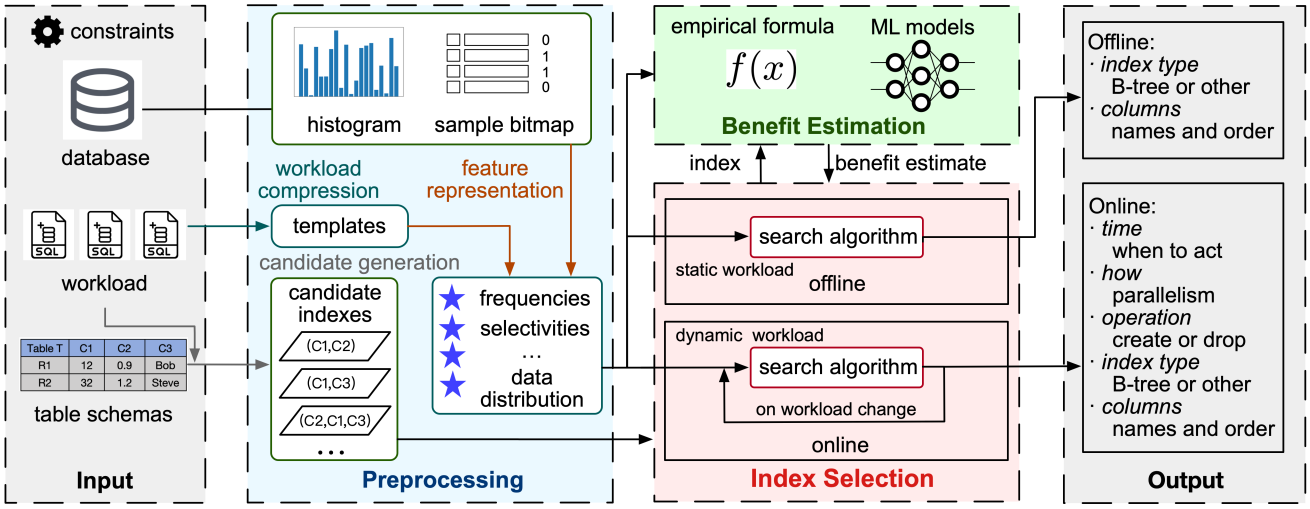

Fig. 2: The Workflow of Automatic Index Tuning

are (i) frequently executed and (ii) most likely to be improved through the implementation of selected indexes. Second, apart from input features, it is common to generate potential index candidates that form the initial index selection space. It is also a challenge to conduct *index candidate generation* with the goal of smaller quantity and better quality. Besides, with the selected queries, there are numerous features of workloads and databases, which need to be selected and elaborately represented to facilitate the index selection procedure. We will discuss relevant techniques in Section III.

The second challenge is *how to accurately estimate the benefit of different index configurations*. First, the precision of benefit estimates is of paramount importance for the final effectiveness of the selected indexes. Besides, the efficiency of benefit estimation also significantly impacts the overall index tuning overhead (e.g., taking over 75% of the total index tuning time [139]). However, achieving precise and efficient index benefit estimates is challenging due to factors like (i) the interaction between indexes [114], (ii) the multitude of candidate combinations, and (iii) different data distributions. We summarize relevant methods in Section IV.

The third challenge is *how to effectively and efficiently select index configurations from the large search space*. Taking a table with 10 columns as an example, there are 10 single-column candidate indexes, 90 double-column candidate indexes, 30240 five-column candidate indexes, which lead to tremendous search space. The hardness and complexity of Offline Index Tuning (NP-hard combinatorial optimization problems [27], [36], [100]) have been analyzed in existing literature, e.g. by establishing connections with *minimum cover problem* and *k-densest subgraph problem*. Naturally, the difficulties lie in exploring diverse combinations of indexes to pinpoint the most effective index configurations that facilitate query execution. Performance and efficiency of the selection algorithm need to be balanced, especially in online settings. We will introduce offline index selection algorithms in Section V and address special concerns for online index tuning in Section VI.

Although there exists summarizing work on index tuning, they have limitations. The survey by Siddiqui et al. [121] only focuses on machine learning-based index advising. The experimental paper by Kossmann et al. [71] only focuses on heuristic algorithms. The survey on database index tuning and defragmentation [130], however, is short in space and lacks extensive review. Our survey makes a comprehensive analysis of both offline and online, both heuristic and machine learning-based index selection algorithms. We present a more structured survey, introducing the background, providing the problem formulation, summarizing the framework and workflow, studying the transferability of automatic index tuning, comparing existing index advisors in detail, and finally presenting potential future directions for index tuning research.

B. Contributions

This survey provides a comprehensive review of existing AIT research and real-world projects. First, we overview the AIT techniques from multiple aspects, including *(i)* problem definition, *(ii)* AIT workflow, *(iii)* AIT framework, *(iv)* index types in AIT, *(v)* index interaction in AIT, *(vi)* changing factors in AIT, *(vii)* automation level of AIT, drawing a figure illustrating the development history of AIT (Section II). Next, we explain each module in the framework, including preprocessing (Section III), index benefit estimation (Section IV), offline index tuning (Section V), online index tuning (Section VI), and real-world deployment (Section VII). Finally, we present potential future directions of AIT research (Section VIII).

II. Index Tuning Overview

In this section, we first formalize the definition of automatic index tuning. Next, we outline the workflow and framework of automatic index tuning. Furthermore, we discuss some important issues that are commonly analyzed in relevant works, such as the index types, index interactions, changing factors, and automation levels.

A. Definition of Automatic Index Tuning

Automatic Index Tuning (AIT) refers to the iterative process of appropriately selecting, creating, updating and dropping indexes in order to optimize the database performance. Database performance refers to the overall efficiency and responsiveness of a database system, affected by various factors, including index design, query optimization, hardware resources, and knob configurations. It is typically measured in terms of workload execution speed and storage occupation.

Depending on the workload characters – whether they are static or dynamically changing, AIT falls into two categories: offline index tuning and online index tuning[1]. Definition 1 is the definition of Offline Index Tuning, where the schema, data, and workload are assumed to be known in advance and remain unchanged. We give the definition of Online Index Tuning in Definition 2.

Definition 1 (Offline Index Tuning): Let D denote a database schema with N_t tables together with the initial data stored, W denote the workload of queries to be executed, C denote N_c constraints (such as storage budget S_{max}[2], selectable index types T_s, maximum index width W_{max}[3], and the maximum number of indexes that can be built N_{max}.), P denote the set of candidate indexes on the table columns, the goal is to find a subset of P under constraints C, such that the execution cost of workload W is minimized.

Definition 2 (Online Index Tuning): Let D denote a database schema with N_t tables together with the initial data stored[4], $W_s = \{w_1, w_2, \ldots, w_T\}$ denote a sequence of T workloads, C denote N_c constraints (such as storage budget S_{max}, selection algorithm running time limit T_{max}, selectable index types T_s, maximum index width W_{max}, and maximum index number N_{max}, etc.), the goal is that at every timepoint $t(1 \leq t \leq T)$, given current index configuration C_t, the new index configuration C_{t+1} should be recommended incrementally for the workload W_{t+1} of next step within time limit T_{max}, and C_{t+1} should not exceed W_{max} or N_{max}.

Example: Assume there are 2 queries: Q_1 and Q_2.

Q_1: select c_custkey, c_name from customer where c_custkey < 10;

Q_2: select c_custkey, c_nationkey from customer wher c_custkey <20;

As for offline tuning, suppose its workload W_1 only contains Q_1 and the memory budget is 500MB. Then the offline index selection algorithm can enumerate different candidate index combinations from scratch via branch-and-cut, heuristic, DQN etc., and decide index configuration as $(c\_custkey, c\_name)$ using 488MB with the highest estimated workload cost reduction.

For online index tuning, given historical workload W_1 (with Q_1) and an incoming workload W_2 (with Q_1 and Q_2), the online index selection algorithm can incrementally search the candidate index space (e.g. rely on MAB, MCTS or some heuristic rules), and choose to (i) add a new index $(c\_custkey, c\_nationkey)$, or (ii) remove the index $(c\_custkey, c\_name)$ and add a index $(c\_custkey, c\_name, c\_nationkey)$ within 10 seconds.

[1]In this paper, *index selection* refers to selecting index type and columns as well as how and when to create them. *Index tuning* includes preprocessing, index selection, and index benefit estimation.

[2]Some researchers make storage budget one of the objective functions [72], [135]. They try to minimize workload execution cost and index storage occupation at the same time.

[3]Index width refers to the number of columns in a composite index.

[4]Only initial data distribution is defined because later data distributions can be determined by the queries executed.

B. Workflow of Automatic Index Tuning

As shown in Figure 2, the general workflow of AIT consists of three parts:

1) **Preprocessing**: Analyze query workloads in database logs or predicted future queries provided by DBAs, analyze query frequencies, identify the most resource-intensive and frequently accessed tables and queries, and extract workload features after workload compression. Furthermore, generate candidate indexes as the initial search space of the following index selection algorithm.
2) **Index Benefit Estimation**: During the index selection process, the benefits for workload execution over the created indexes need to be estimated by the query optimizer or machine learning models, so that the *Index Selection* module can be guided to select the most beneficial index configurations.
3) **Index Selection**: Determine (i) which table and columns require indexing, (ii) index types (e.g., B-tree, hash, GiST, GIN, learned index), and (iii) the column order in composite indexes, as well as (iv) when and how to create or drop the index in online index tuning.

C. Framework of Automatic Index Tuning

Figure 3 summarizes the techniques in index tuning, involving preprocessing, index benefit estimation, index selection, and deployment into databases.

1) Preprocessing: When the workload comprises a large number of queries, compressing the workload by selecting representative queries is essential and it should be ensured that index tuning on the selected queries resembles the tuning on the original workload. A comprehensive representation of the target workloads is required to facilitate optimal index configuration search. We will introduce workload compression, feature representation techniques, and candidate generation rules in Section III.

2) Index Benefit Estimation: During the search for optimal indexes, it is essential to quantify the benefits of building indexes at each step. The benefit of a set of indexes is typically measured by comparing the reduction in workload execution cost with the original cost before the indexes were built. Apart from physical creation of indexes and actual execution of workload, there are two common ways for cost estimation: empirical formulas and machine learning models. We will introduce them in detail in Section IV.

3) Offline Index Selection Algorithms: Offline index selection algorithms can be broadly categorized into the following two classes:

• Exact algorithms: These algorithms guarantee finding the exact optimal solutions. However, they are not scalable and may not be practical for large datasets.

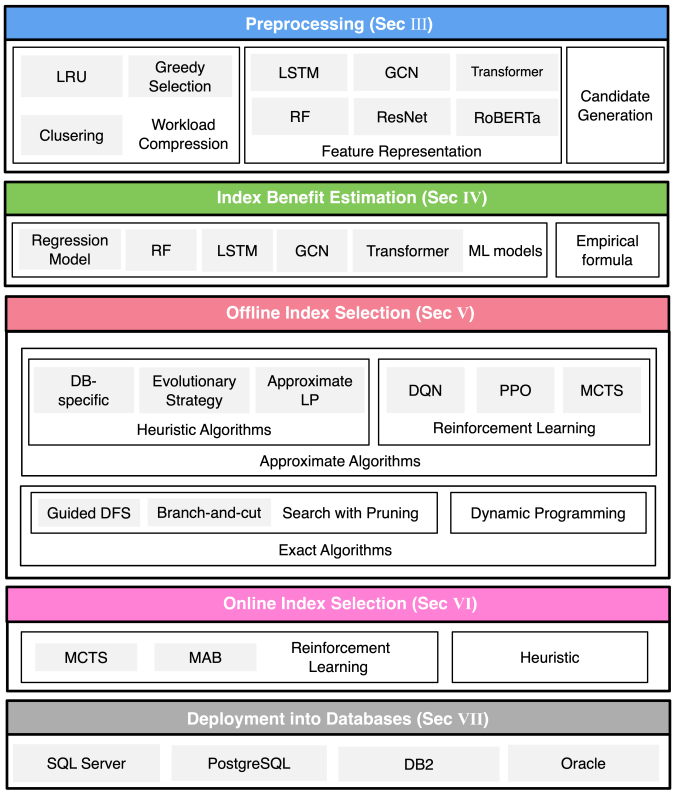

Fig. 3: The Framework of Automatic Index Tuning

• Approximate algorithms: These algorithms are designed to find near-optimal solutions more efficiently.

We will introduce and compare different offline index selection algorithms in Section V.

4) Online Index Selection Algorithms: In online scenarios, the index tuner should detect the workload change timely and promptly re-recommend indexes, create new indexes, and delete obsolete indexes, in adaptation to workload shifts. Currently, there are heuristic approaches and machine learning approaches (mainly Reinforcement Learning) for online index selection. We will introduce online index selection methods in Section VI.

D. Index Types in Automatic Index Tuning

In current research on index tuning, most index advisors recommend B-tree indexes by default. Additionally, there have been notable advancements in recommending different index types based on specific requirements and database characteristics.

MISA [12] recommends spatial, text, and hash indexes for MongoDB. MANTIS [118] also provides recommendations for various index types, including B-tree indexes, block range indexes, hash indexes, and spatial indexes (specifically for Postgres). Block range index is helpful in range queries. Hash index improves the performance of hash joins and point queries. Spatial index helps to quickly retrieve spatial data based on its location.

ALMSS [152] goes even further by considering learning-based index types. It dynamically evaluates the performance

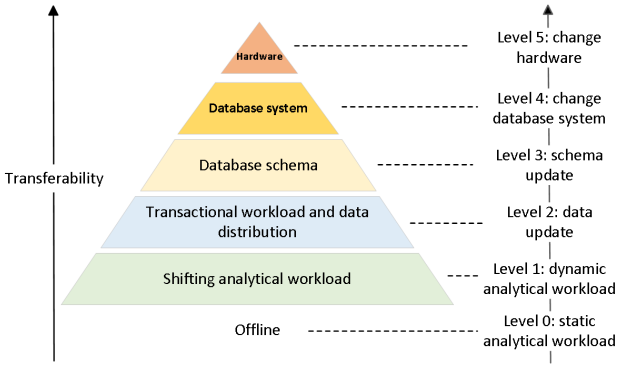

Fig. 4: Level Pyramid of Changing Factors in AIT

of learning-based indexes and, if the error exceeds a certain threshold, it reverts to using a traditional index type (B-tree or hash) for the respective leaf node. A random forest classifier and different regression models are utilized to compute the error.

By incorporating various index types into recommendations, index tuning can provide more tailored and effective index configurations for different database scenarios.

E. Index Interaction in Automatic Index Tuning

Index interaction refers to the influences of one group of indexes on the benefits of another group. Some researchers assume that index interaction has little impact on index selection results and ignore the mutual influence between indexes in their studies [25], [34], [40], [53], [55]. These researchers determine the impact of each index on queries independently and then aggregate them to obtain the net impact.

Nevertheless, it has been observed that the benefit of combined use of multiple indexes (index intersection) often differs from the mere sum of the benefits obtained when using the indexes separately. As a result, the index benefit estimation models need to take mutual influence between indexes into account and re-evaluate different index configurations as a whole [76]. Benefit estimation techniques considering index interaction will be explored in Section IV.

F. Changing Factors in Automatic Index Tuning

In AIT, workload, data, schema [134], database system, operating system, and even hardware [63], [145] may change. These changing factors affect index advisors by affecting benefit estimation model. Wang et al. [153] points out that an open problem is to enhance the index advisor's generalization across various database systems, considering differences in schema, query optimization strategies, and data distributions. Sun and Li [123] also extend their models to handle (i)tuple update, (ii) column update, (iii) table update.

To make a clearer clarification of the transferability of index tuning, which is vital to achieve more efficiency and better generalizability, we define six levels for AIT on the basis of changing factors, such as changing workloads [112], [117], [150], data [92], schema [123], [134], database system [121], and even hardware [63]. as shown in Figure 4.

1) Level 0: Static Analytical Workload: This level corresponds to offline index tuning, which refers to index tuning for a static workload that is known in advance. It constitutes the majority of current research on index tuning, as introduced in Section V. For well-defined and relatively stable database application scenarios, offline index tuning algorithms can be employed before deploying the database. The advantage of offline algorithms lies in their ability to leverage a large amount of historical query data and sufficient time to train the model, resulting in high-quality index recommendations.

2) Level 1: Dynamic Analytical Workload: This level and those above belong to online index tuning. In adaptation to dynamic analytical workloads, online index tuning is more complex than offline index tuning, as it requires continuous adjustments and needs to meet real-world requirements, such as stricter time constraints, as explained in Section VI.

3) Level 2: Data Update: Some AIT methods focus exclusively on analytical workloads, where the goal is to optimize the retrieval of data without considering the impact of data modifications [105], [106], while other researchers take data update operations into considerations [69], [92], [97], [150], which necessitates addressing the index maintenance cost. As data evolves, indexes may become less effective, and the overhead of keeping them up-to-date can be significant. This includes the cost of index creation, deletion, and maintenance during data updates.

4) Level 3: Schema Update: We define schema update as the change of table definition such as table creation, column appending, and removing. It demands careful design to handle schema changes, which might bring about more candidate indexes and different feature representations. Existing approach to column name encoding is typically one-hot, thus not directly transferable to new database schemas. However, column can be encoded in a transferable way for cost estimation [56][5]. Sun and Li [123] also considered column update and table update in their cost estimation model. Learning-based algorithms need to be schema-agnostic, trained on several schemas, and capable of recommending indexes for a new schema.

5) Level 4: Change Database System: Most index advisors are designed for relational databases [33], [131]. As the popularity and usage of NoSQL databases grow, there has been an increasing focus on developing index tuning approaches for non-relational database systems, such as MISA for MongoDB [12], DRLISA for NoSQL database [143], and holistic indexing for main-memory column stores [99]. Recommending indexes for different database types enables better performance optimization in diverse data management environments. Moreover, there have been notable developments in index advisors specifically designed for cluster databases [37], [52], [104]–[106], [127]. The change of database system requires cost estimation model and index advisor to achieve cross-database compatibility, e.g. pretrained models should transfer across different database systems [93].

6) Level 5: Change Machine: Yu et al. [145] observed that the hardware can influence plan node of different types. Wu et al. [138] make modifications to query optimizer's cost model to predict runtime on different hardwares. Data Calculator [63] and Wehrstein et al. [134] also takes hardware (CPU, memory, caches) into account. By utilizing these hardware-transferable cost estimation techniques, or to be hardware-transferable itself, a universal cross-platform automatic index tuner will be able to adapt to the heterogeneity of features varying across databases, OS, and hardware [121], achieving zero-shot or few-shot learning ability and guaranteeing competitive performance.

[5]https://github.com/DataManagementLab/zero-shot-cost-estimation

G. Automation Level of Automatic Index Tuning

Based on the level of human involvement, index tuning can be classified into semi-automatic tuning and fully automatic tuning from the aspect of automation level.

- Semi-automatic tuning: Semi-automatic methods emphasize the interaction between humans and tools. These methods involve the use of algorithms to recommend indexes, but DBAs can stop the algorithm when they are satisfied with the intermediate results [40] and they will confirm the indexes recommended by algorithms. Implemented in Kaizen [66] system, WFIT [113] combines the best of DBAs and automatic tools, providing indirect expert knowledge to the recommendation tool and provides feedback on the recommended results by allowing for iterative adjustments of indexes.
- Fully automatic tuning: Fully automatic methods aim to make index selection tools self-driving without the need for human intervention. These methods employ algorithms, such as machine learning or optimization techniques, to analyze workloads and recommend suitable indexes for various queries and data distributions.

In the following sections, we separately introduce the detailed techniques in different AIT modules (Figure 5) and discuss their advantages and disadvantages.

III. Preprocessing

In order to optimize index configurations for query execution, it is crucial to handle the huge search space by compressing a large workload [28], [120] and generating candidate indexes [77], both of which work ultimately to reduce the *number* of required index benefit estimation calls. Feature representation of queries and databases [32], [43], [72], [117], [150] is also of significance to facilitate other preprocessing steps as well as index selection and index benefit estimation. These approaches enable search algorithms to prioritize and focus on frequently executed and significant queries, leading to improved index recommendations. Additionally, Brucato et al. [20] proposed *workload reduction* recently to reduce the *complexity* of what-if calls, which is a research direction orthogonal to other preprocessing techniques.

A. Workload Compression

When dealing with a large number of queries, workload compression (a.k.a. workload summarization) is often required, because it is hard to take all queries into consideration. This involves extracting key queries, focusing only on dominating

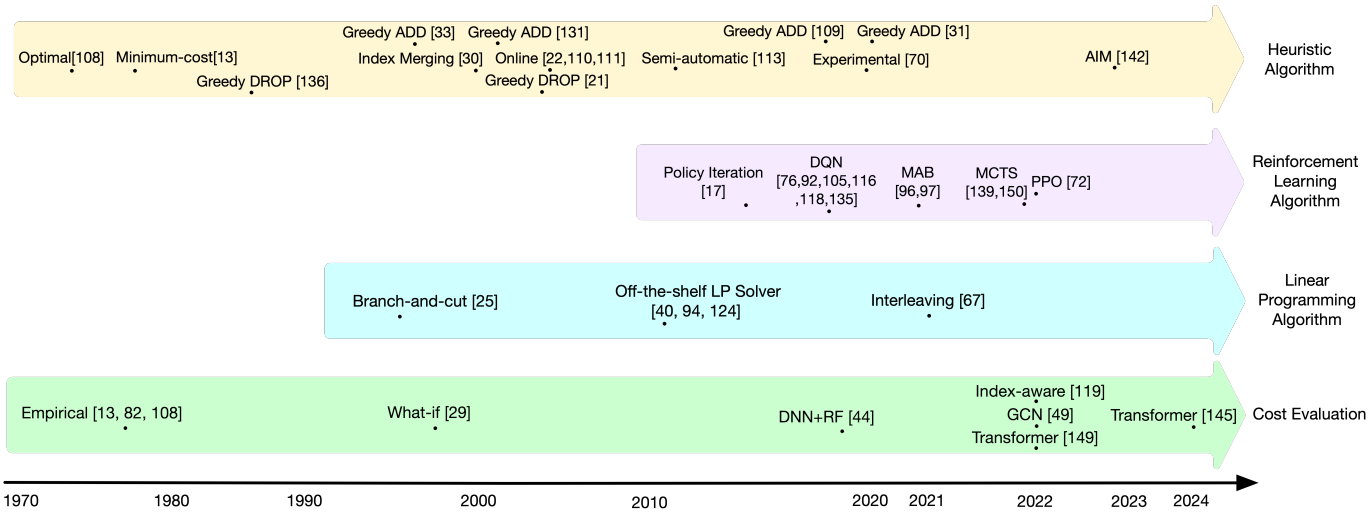

Fig. 5: Development History of Automatic Index Tuning

queries, and merging similar queries into templates. Suitable indexes can then be recommended for these critical queries. The objective of workload compression is to ensure that the index recommendations for the compressed workload closely align with those for the uncompressed workload [120].

We can categorize workload compression methods into indexing-agnostic and index-aware approaches, as Siddiqui et al. [120] do.

1) Indexing-Agnostic Methods: These methods do not take indexes into special consideration when compressing workloads. They often involve uniform sampling and extraction of query templates from a large number of queries. A template represents a group of structurally similar queries. Chaudhuri et al. proposed GSUM [32] to compress workloads, maximizing the coverage of features (e.g., columns) and representativity of the entire workload. Ma et al. [83] and Chaudhuri et al. [28] reduced the number of queries using query clustering algorithms. Deep et al. [43] employed a greedy algorithm to select the most representative subset of queries from the entire workload. Zhou et al. [150] mapped queries to query templates and maintained a certain number of SQL templates using an LRU strategy to identify candidate indexes. While indexing-agnostic workload compression methods can identify frequently occurring queries in the workload, they may overlook queries that have more significant implications for index creation.

2) Index-Aware Methods: One example of index-aware workload compression approaches is ISUM [120]. It decomposes the benefit of an index recommended for a query into *utility* (benefit for the query) and *influence* for other queries. The *influence* of q_i on q_j is the multiplication of the *similarity* between the two queries and the *utility* of q_j, representing the reduction in the utility of q_j when q_i is selected for index tuning. The similarity between queries is determined based on their indexable columns, each of which is assigned an importance value and a weight. To avoid pairwise similarity computation, a summary feature is computed for a workload and each query's similarity to the workload is computed against the summary feature. The algorithm selects the maximum benefit greedily at every step and updates query features and utilities of queries correspondingly. After queries are picked out, they are clustered in templates and their weights are re-calibrated.

B. Candidate Generation

To narrow the search space, candidate generation is often performed based on heuristic rules summarized by DBAs from their tuning experience. Indexable columns are combined to form candidate indexes, among which the optimal index configurations will be searched for in the index selection process. This greatly speeds up index selection, compared with searching among all permutations of indexable columns.

Candidate generation can be regarded as a rule-based approximate index selection algorithm without storage constraints[6].

Lan et al. [77] proposed five rules to guide the generation of index candidates, e.g. first attributes in predicates, then GROUP BY and ORDER by etc.

Yadav et al. [142] analyzed PROJECTION, SELECTION, JOIN, GROUP BY, and ORDER BY clauses to obtain partial orders of indexable columns, e.g. factorizing complex AND-OR predicates into disjunctive normal form and obtaining a partial order for each factor, with the more selective predicate columns put first. Then these partial orders are merged according to some rules, e.g. index-prefix-predicate columns take higher priority than columns that feature in GROUP BY and ORDER by clauses.

Lahdenmaki and Leach [75] proposed the Three-Star Index criteria for a query, introducing concepts of matching columns, screening columns, fat index and so on[7]. Their proposed

[6]However, it is hard to recommend optimal indexes for a workload of queries under storage constraints using simple rules, because this is an optimization problem. A cost-based search is indispensable.

[7]These concepts help us to understand how to devise indexes to minimize I/O operations.

QUBE analysis is helpful for understanding ideal index selection and candidate index generation, e.g. about positioning EQUAL predicate columns, RANGE predicate columns, ORDER BY and GROUP BY columns in a composite index[8].

Yadav et al. [142] proposed an automated index management system AIM, which identifies impactful secondary indexes for SQL databases in order to efficiently use available resources such as CPU, I/O, and storage. The paper mainly introduces heuristic algorithms to generate candidates for SELECT, GROUP BY, and ORDER BY clauses, followed by a partial order merge process. AIM treats complex join queries systematically, considering coordinated exploration of candidate indexes across multiple tables. AIM is featured by its solution quality better or at par with the state-of-the-art algorithms and relatively cheap and stable runtime.

C. Feature Representation

An effective representation of workload and database features plays a critical role in enhancing the selection of optimal indexes. The process faces two main steps:

- **Feature selection and extraction**. This entails identifying pertinent features to extract from query plan trees, such as operator type, predicate selectivity, data statistics, and index information.
- **Feature representation**. This involves selecting an appropriate model to embed the concatenated features into a low-dimensional representation, such as LSTM (Long Short-Term Memory) [58], GNN (Graph Neural Network [107]), and Transformer models.

The difficulties of representation primarily stem from the inherent high dimensionality of features and the intricacies of query interactions. Notably, queries may introduce interference that is not easily discernible based solely on plan features, such as resource contention between concurrent queries. Taking index features as one of the inputs, as Shi et al. [119] did, is particularly vital, especially in the context of index tuning.

Various types of Machine Learning models have been utilized to represent queries, e.g. Ding et al. [44]'s DNN with random forest, Yuan et al. [146]'s LSTM, Gao et al. [49]'s combined use of LSTM, GCN, and ResNet, Zhao et al. [149]'s tree-structured Transformer, Kossmann et al. [72]'s "bag-of-operator", and Sharma and Dyreson et al. [117]'s using pretrained RoBERTa [80].

Features and models in these methods are compared in detail in Table I in Section IV.

D. Workload Reducation

Brucato et al. [20] proposed a new idea to decrease what-if call time. Their method, named WRED, rewrites workload queries to be simpler, by eliminating column and table expressions unlikely to benefit from indexes from the query plans, such that it will be quicker for the optimizer to do what-if estimates while guaranteeing that the indexes recommended on the rewritten queries are similar to those recommended on original queries. This *workload reduction* technique is complementary and orthogonal to other index tuning techniques such as *workload compression*. Experiments show that combining WRED and ISUM results in higher speedups than either of the two techniques alone and maintains the quality of index tuning.

[8]One execution strategy of query Q might involve sorting rows in group order before selection of rows, whereas another strategy might prefer selection before sorting in group order. If selectivity is high, selection before sorting is preferable. If there is 'LIMIT N' in the query, maybe sorting first is better. This decision influences index selection and depends on specific scenarios.

IV. Index Benefit Estimation

During the search for optimal indexes, estimating the benefit of each index configuration is crucial so that the most beneficial one can be selected. The benefit of an index configuration is determined by how much it reduces the query execution time and minimizes the index maintenance cost after its creation. The lower the query execution time and the index maintenance cost, the higher the index benefit.

Ideally, the most accurate way to evaluate index benefit is to actually build each index and execute the workload to observe the performance improvement. However, this approach is computationally expensive and impractical for large databases and complex workloads.

To overcome this challenge, benefits are typically estimated based on the reduction in estimated costs [76], [77], [116]. Fortunately, it is sometimes sufficient to provide relative cost estimates for different index configurations to select the better ones, not necessary to obtain the exact execution time of queries.

Typically, there are two ways to estimate query cost:

1) Utilize empirical formulas.
2) Train machine learning models such as neural networks.

A. Cost Estimation Based on Empirical Formula

In the early stages of database optimization, optimizers did not provide externalized cost estimates, researchers had to resort to developing their own simple cost models to mathematically emulate the cost estimates given by query optimizers [13], [82], [108], [136], [144]. Modern optimizers, however, use more refined cost models and DBAs rely on query optimizers to estimate the costs of query execution. This technique is called hypothetical index, also known as "what-if cost estimation" [29].

Hypothetical indexes work to simulate how query execution plans would change if the hypothetical indexes were actually created in the database and they won't be used in the actual execution path of any query. One advantage of using optimizer-estimated cost as the cost metric of a query is that the "consumer" of a configuration is the optimizer, resulting in better synchronization. An optimizer's decision on whether or not to use an index is solely based on the statistical information on the column(s) in the index, such as histograms of the column values obtained via sampling instead of scanning all rows [29].

Lum and Ling [82], Anderson and Berra [13], Schkolnick [108], and Whang et al. [136] all proposed their own empirical formulas to estimate costs.

Presently, many mainstream database products rely on query optimizers to evaluate costs, such as AutoAdmin [29] for

Microsoft SQL Server, Oracle's index advisor, and DB2's advisor [131].

Taking AutoAdmin as an example, it creates hypothetical indexes by sampling and allows users to specify the fraction of the table to be scanned when gathering sample data on columns of the index. It adopts an adaptive page-level sampling algorithm and starts with m pages (e.g. the square root of the number of pages), the *Sample-Table* contains statistical measures such as the data density and histograms. Then it samples new pages as cross-validation and stops the sampling if the density measure converges.

AIM [142] is a recent practical heuristic online index tuning approach that ranks and selects candidate indexes based on heuristic formulas. The approach separately estimates the gain of index set for read queries and the overhead of index maintenance.

Idreos et al. [63] proposed Data Calculator that can answer what-if data structure design questions to understand how the introduction of new design choices, workloads, and hardware affect the performance (latency) of an existing design. Therefore, it potentially increases the transferability of index advisors, and is very useful to estimate index benefit and help index advisors to choose the best index design.

Although there may be discrepancies between estimated costs and actual execution costs [95], these estimations can be used to compare different plans and choose the best. The advantage of empirical cost estimation is their light weight, thus widely applied in real-world database systems.

B. Cost Estimation Based on Machine Learning

When relying on the optimizer, biases in index benefit estimates can come from the limitations of the optimizer, such as erroneous selectivity estimates and the inaccurate cost model [79]. These traditional cost models are typically based on the weighted sum of cost features, where the weights are static [150] and may cause errors [150]. The biases in index benefit estimation can affect the accuracy of the search algorithms. In recent advancements in academia, machine learning algorithms are employed to estimate query cost, some achieving better accuracies than traditional models. These algorithms can be divided into *general* methods and *index-specific* methods, based on whether they contain information of existing indexes in their feature representation and therefore whether they rely on query optimizer's what-if calls.

General cost estimation methods such as QueryFormer [149], MB2 [84], ZeroShot [56], AImeetsAI [44], E2ECost [123], QPPNet [85] do not contain the information of existing indexes in their features, so they rely on what-if calls to get query execution plans and they learn a mapping from the featurized query execution plan to query execution cost[9].

Index-specific methods such as DISTILL [122] and LIB [119], however, contain the information of existing indexes in their feature representation. Therefore, they need not rely on what-if calls. They can learn a mapping from the tuple <query SQL statement, indexes> to the execution cost of the query or the benefit of the indexes[10]

1) General Methods: Ding et al. [44] presented a technique to featurize query plans into vectors and trained classifier models to compare the costs of two plans. In adaptation to other databases, the authors combined offline models and local models to improve prediction accuracy. The authors tried logistic regression, bagging and boosting ensemble of trees as the offline model per database or globally. The local model is lightweight, only trained on a small amount of execution data from one database. Experiments show the proposed classifier results in 2 to 5 times of reduction in errors than regression models that first predict costs and then compare.

Zhao et al. [149] replaced the representation model of E2ECost [123] with a learning-based query plan representation model called QueryFormer. The model has a tree-structured Transformer architecture that can model long paths of information flow in query plans and capture parent-children dependency. It takes data statistics into account, such as histograms, random samples, and physical query plans. QueryFormer can better estimate query costs than AIMeetsAI and help downstream index selection tasks perform better.

Hilprecht and Binnig [56] introduced zero-shot cost models to generalize across databases[11]. The database-dependent module captures database-specific data characteristics(e.g., histograms, estimated cardinalities) while query plans are represented as database-agnostic features and these transferable features serve as input to the general database-agnostic GNN module, which predicts query costs. The sophisticated feature engineering and the powerful graph neural network make the proposed cost model more accurate than the state-of-the-art models for a wide range of real-world databases with few query executions on unseen databases. Providing cost estimates, the zero-shot cost model is promising to enhance the transferability of index advisors across databases.

2) Index-specific Methods: Siddiqui et al. [122] trained a cost estimation model for each group of queries that share similar patterns concurrently during index tuning. Among linear regression, decision tree, tree-based ensemble model, and multilayer neural network, the tree-based ensemble model achieves the best result in consideration of both estimation time and accuracy.

Shi et al. [119] introduced the Learned Index Benefit (LIB) model that contains the Set Transformer and represents interactions between indexes and takes operator types, column statistics, and index information as input. Its parallel execution and dedicated design enable it to accurately estimate costs, generalize well to unseen workloads and datasets, and make index tuning faster.

Gao et al. introduced SmartIndex [49], which extracts features from query plan trees and relevant indexes using LSTM,

[9]General methods can also handle INDEX SCAN operator type or node, but the information of the use of INDEX SCAN is provided by the query optimizer.

[10]Containing index features does not mean they can't use query plans. Index-specific RIBE [145] encodes indexes but it also utilizes query plans, because they obtain query plans under no indexes ahead of time. RIBE uses ChangeFormer to predict whether query plan structure will change after new indexes are built. Therefore, RIBE is a little bit different from previous index-specific methods like DISTILL and LIB.

[11]https://github.com/DataManagementLab/zero-shot-cost-estimation

TABLE I: Comparison among ML-based Cost Estimation Methods

Type	Method	Features	Models	Target of estimation	Advantage	Disadvantage	Performance
general	AIMeetsAI [44]	measure of work done, structural information, and physical operator details	RF and DNN	plan cost comparison	utilize estimated information of query execution	require many comparisons between query plans	render better index selection quality
	QueryFormer [149]	operator type, predicate, and data statistics	Tree-structured Transformer	query execution cost/time	consider parent-children dependency and long information flow in query plans	slower than AIMeetsAI [44] due to complex encoding schema	better than E2ECost [123] and AIMeetsAI [44]
	MB2 [84]	amount of work, parallel invocation status, and DB knobs	OU-specific models and one interference model	query execution cost/time	less training time and data, better accuracy and robustness	no hardware-across generalization	more accurate than QPPNet [85] and E2ECost [123]
	ZeroShot [56]	parallelism, operators, data statistics, etc.	GNN	query execution cost/time	generalization across databases	computationally expensive	better or equal to workload-driven cost models
	QPPNet [85]	operator types, cardinalities, I/O number, etc.	Tree-structured neural network	query execution cost/time	capturing the structure of query plans	large training overhead	outperform TAM and SVM.
	E2ECost [123]	operator, predicate, metadata, sample bitmap	Tree-structured neural network	query execution cost/time	training cost and cardinality estimation simultaneously and handles string embeddings	large training overhead	outperform traditional optimizer's cost estimation.
index-specific	DISTILL [122]	operator type, estimated computation, and index features	LR, tree-based ensemble model, and MLP	improvement	clustering and grouping of queries reduce computation	time-consuming query plan generation	better than DTA [31]
	LIB [119]	operator type, data statistics, and indexes	Set Transformer	improvement	consider index interaction and run in parallel	ignore the relation between operators	more accurate and faster than PostgreSQL cost estimator
	SmartIndex [49]	operator type, estimated computation, execution order, and index features	LSTM, GCN, and Attention Model	query execution cost/time	represent execution plans and indexes better	require a large amount of computation	faster than DB2 Advisor [131] and Extend [109]
	AutoIndex [150]	query type and index features	deep regression model and empirical formula	query execution cost/time	dynamically balance IO cost and CPU cost	model not complex enough	higher estimation accuracy
	ChangeFormer [145]	operator type, table, statistics of input and output, predicate, join schema, index	Transformer	change	prune the number of what-if calls	incorrect prediction of change harms accuracy	achieve cost estimation accuracy comparable to that of the optimizer

GCN, and ResNet. They used Mean Square Error (MSE) as the loss function and the actual execution time of query as labels. Integrated with the proposed cost estimation model, their greedy search algorithm outperforms DB2 Advisor [131] and Extend [109] on the Job dataset.

Zhou et al. [150] trained a one-layer deep regression model that considers index update cost, using empirical formulas with hyperparameters to estimate CPU and IO costs and summing them up with dynamically learned weights. Focusing on index updates that involve disk IO and ignoring in-placement updates where the new and old index tuples are recorded in the same heap page, the authors designed meticulous cost models that result in improved estimation results.

Sun and Yan [145] proposed RIBE, they trained ChangeFormer to predict whether the structure of query plan will change after new indexes are built. If they predict positive,

what-if has to be called again. However, if the structure of query plan is predicted not to change, RIBE utilized a much simpler method to estimate the benefit of the indexes than calling the complex what-if call again. For example, simply subtract INDEX SCAN cost from SEQ SCAN cost. Experiments show that RIBE can achieve cost estimation accuracy comparable to that of the optimizer.

The disadvantage of ML-based methods is that they often require a substantial amount of training data and may not be suitable for general business environments. To mitigate the requirement of huge training data, Ma et al. [84] proposed a decomposed behavior model MB2 and trained a separate model for each Operating Unit (OU). Table I shows a comparison among ML-based cost estimation methods.

C. Challenges and Solutions

Index benefit estimation faces two challenges:

1) **C1**: Accuracy (index interaction). Factors accounted for the costs are complex and interrelated, and include the frequency of read and write queries, distribution of data, index size, and maintenance of candidate keys and index structures [13]. Moreover, there exist interactions between indexes that affect their benefits mutually. Creating or deleting one index might affect the performance of other indexes [16], [64], [137]. Thus, the benefit of a set of indexes can not be calculated as the sum of the benefits of each individual index, which makes index selection harder than the classical Knapsack Problem.
2) **C2**: Efficiency (slow evaluation). The total number of candidate indexes is huge and evaluating each takes significant time. Papadomanolakis et al. [95] showed that cost estimation accounts for 90% of the total time of index tuning. Even if virtual indexes [29] are used, the majority of index tuning time is still spent on what-if calls [139]. Neural network models trained on historical data of query execution can lead to more accurate cost estimation but they take even more time to estimate execution cost, not to mention its prolonged training time and big size.

To address **C1**, index interaction should be properly handled to make index benefit estimation more accurate. Schnaitter et al. [114] proposed to compute the degree of interaction (DOI) between pairs of indexes in offline index selection setting. DOI is calculated as the difference in the benefit of one index before and after another index is created divided by the query cost. The authors integrated index interaction evaluation into index tuning and presented heuristics to schedule index materializations. Bruno and Chaudhuri [22], [23] also considered index interaction by analyzing the usefulness of one index given another. Their approach classifies usefulness into four levels, adjusting the benefits of other indexes accordingly, e.g. after one index is created, it reduces the benefits of relevant indexes, and increases benefits for index deletion.

Approaches to address **C2** can be classified as below:

- **Caching mechanism**. Store previous what-if cost estimation results of one query under one index configuration for later use to avoid repeated identical calls [24], [71], [95]. The space of the cache is limited and the Least Recently Use (LRU) strategy can be used to replace outdated evaluation results with new ones.
- **Cost derivation**. Derive costs from previous what-if calls to reduce the number of optimizer calls. For example, C-PQO adopts the MEMO data structure (a configuration-parametric physical operator tree) that enables fast cost estimation based on previously optimized queries with similar structures. BAIT [139] limits the budget of what-if calls and derives the cost of an index set as the minimum cost over all subset configurations with known what-if calls for offline index selection scenarios, where database and workload are assumed to be fixed.
- **Independent partitioning**. WFIT [113] divides indexes into independent sets, ensuring that indexes in different sets do not interact with each other. In this way, benefits of indexes from different sets satisfy linearity and can be added up, thus reducing the number of index configurations that need to be evaluated.
- **Ranking strategy**. Proposed by Schnaitter et al. [110], COLT first ranks index candidates with easy-to-compute yet crude performance statistics and identify a small set of *hot indexes*. The benefits of *hot indexes* and *materialized indexes* are estimated with accurate and expensive methods. *Materialized indexes* are given precedence in spending what-if budgets.
- **Query rewrite for reduced complexity**. Brucato et al. [20] proposed WRED that rewrites query (e.g. eliminating some filters, joins, etc.) to reduce optimization complexity. Their experiment shows that the average what-if call time grows proportionally to the average number of column references and table reference.

V. Offline Index Selection Algorithms

Offline index selection algorithms can be classified into exact algorithms and approximate algorithms. Exact algorithms such as *brand-and-cut* can enumerate and find the optimal index configuration, but they are not efficient in tuning large-scale databases. On the other hand, approximate algorithms offer more scalable solutions, including heuristic algorithms and learning-based ones, especially Reinforcement Learning (RL).

A. Exact Algorithms

Exact algorithms for offline index selection include searches with pruning and Dynamic Programming approaches. They are guaranteed to find the optimal solution for the optimization problem.

1) Search with Pruning: To tackle the offline index selection Problem, researchers often take Knapsack Problem (KP) [68] and Generalized Uncapacitated Facility Location Problem (GUFLP) [25], [26] as reference. Formulated as Linear Programming (LP) problems, index selection problem can be solved by searching the candidate space. Appropriate pruning is applied in efficient searches for optimal solutions to these problems.

For example, Caprara et al. [25] assumed that a query can only utilize at most one index in their early methods,

and Dash et al. [40] assumed the cost model is linearly separable [24], [95].Moreover, the assumption that there is no interaction between indexes is made by many researchers such as Cophy [40] so that the benefit of each index can be evaluated independently.

Caprara et al. [25], [26] employed a branch-and-cut algorithm to search, and it could provide the distance between a solution and the optimum. Schkolnick [108] used a guided depth-first search to find the optimal set of indexes if the cost function is *regular*. This method constructs chains of sets, each being a superset of the previous one by adding one more candidate index. The process ensures a non-increasing sequence of workload costs. Points with smaller costs are included in the current set, and the precedence between two points in an independent set is discovered during the partial search.

Exact algorithms are only suitable for small-scale problems. Moreover, existing research on exact algorithms often makes certain assumptions to simplify the mathematical model, but the assumptions are sometimes unrealistic in practical cases. For instance, the *regular* property [108] does not necessarily hold due to interaction among indexes (e.g., the cost function value of the index set (A, B, C) is the smallest, but the cost of (A, B) is larger than that of (A) and (B)).

2) Dynamic Programming Methods: Dynamic programming is commonly used to calculate the optimal index configuration under different space constraints and recursively compute the results by modeling offline index selection as KP.

Let $Benefit(X, S_{max})$ be the maximum benefit we can get from building a subset of index set X under the storage constraint of S_{max}. The recursive formula would be

$$\begin{aligned} &Benefit(X, S_{max}) \\ &\quad = \max\{Benefit(X - I, S_{max} - Storage(I)) \\ &\qquad + Benefit(I, Storage(I)\}, \\ &\qquad\quad \text{for } I \in X \text{ and } Storage(I) \leq S_{max} \end{aligned} \quad (1)$$

Qiu et al. [102][12] employed Dynamic Programming techniques to recommend the optimal indexes and stored results of subproblems after hashing to avoid duplicate computing. Yang et al. [144][13] proposed CedarAdvisor based on Xiaomi SQL Optimization and Rewriting Tool (SOAR) [141]. CedarAdvisor automatically collects workloads from logs, gathers query frequencies, generates candidate indexes for individual queries, and evaluates index benefits and costs. It also employs Dynamic Programming to find the optimal index configuration for the entire workload, or near-optimal one if time is constrained. Tested on the distributed database Cedar [42], experiments demonstrate the effectiveness of CedarAdvisor.

Dynamic Programming is not scalable to large data volumes because it involves many times of calculating benefit(X, storage) which is NP-hard itself [27], [36], [100], and thus not widely used.

B. Approximate Algorithms

In practical applications, exact optimal solutions for index selection are often unattainable and unnecessary. On one hand, the mathematical modeling of index selection itself is an approximation of the real-world scenario. The assumptions and simplifications made in the modeling process introduce inherent imprecision. On the other hand, it is often adequate to find an acceptable solution, as long as the solution provides significant performance improvements and meets desired goals. Most importantly, exploring the entire solution space with limited computational resources and time is unfeasible.

To address these challenges, researchers have explored various approaches to select indexes more efficiently. These approaches include DB-Specific heuristics, approximate LP approaches, evolutionary strategies, and RL algorithms.

Apart from generating candidate indexes based on empirical rules, integrating heuristics into the search process can also lead to improved efficiency in index selection. One common type of heuristics is the greedy heuristic, which aims to get a near-optimal result at every step greedily (e.g. selecting the index with the best benefit-to-size ratio). There are two directions to search: ADD heuristics and DROP heuristics:

- **Greedy ADD Heuristics**. AutoAdmin [33], DB2 Advisor [131], and Extend [109] apply ADD heuristics. Their search starts from an empty set and gradually adds useful indexes until constraints are violated. Ip et al. [64] and Chaudhuri et al. [27] also employed this heuristic. ADD heuristics are more widely applied in index selection approaches, but they might choose an insignificant index that looks as if it were important at the initial stage of the design but deviates from the optimal.
- **Greedy DROP Heuristics**. Drop [136] and Relaxation [21] apply DROP heuristics. Their search starts from a full set of all possible candidate indexes (e.g., the combination of the optimal indexes for each query) and gradually drops useless indexes until the space constraint is satisfied. The advantage of DROP heuristics over ADD heuristics is that Drop heuristics can take the influence of an index on others into account from the beginning because all indexes are initially present [136].

1) DB-Specific Heuristics: AutoAdmin by Chaudhuri et al. [33] adopts a greedy iterative approach that first recommends single-column indexes and then multi-column indexes of increasing width. At each step, they added the index that resulted in the highest cost reduction among all possible choices.

As for DB2 Advisor [131], after the completion of greedy selection, it introduces one more variation step, where parts of the existing indexes are randomly replaced with indexes not recommended but more beneficial for workload execution.

Extend [109] recursively adds single-column indexes with the highest cost-space reduction ratio or extends an existing index by appending a column at its right, until the storage budget is reached or no further cost improvement can be made.

[12]PDF version of this paper can be found at https://jos.org.cn/josen/article/abstract/5906

[13]PDF version of paper can be found at https://cloud.tsinghua.edu.cn/f/a7ffed1ca8234d4eb75f/

It takes index interaction into account, efficiently recommends multi-attribute indexes, and can extend to large-scale instances.

Drop [136] iteratively removes indexes that lead to the lowest cost of processing transactions from a full index set until no further cost reduction is possible. Relaxation [21] iteratively transformed the initial set containing all queries' optimal indexes into sets that consume less storage space by merging, splitting, prefixing, clustering, and removing, at the same time trying to keep the benefit as high as possible.

Chaudhuri [30] presented a greedy heuristic that merges multiple indexes into a single index. The goal was to minimize either the query execution cost given a storage budget or the storage requirement given a cost reduction budget. This merging procedure can be utilized after indexes are recommended by other index selection algorithms to reduce storage overhead as well as maintain query performance.

Ameri et al. [12] employed mining algorithms [45] to generate column combinations. Schnaitter and Polyzotis [113] proposed WFIT that independently searches for candidate indexes that do not affect each other, and the optimal indexes are the ones that minimize the total function value by employing divide-and-conquer.

DB-specific heuristic algorithms of index selection are most widely applied due to their simplicity and effectiveness.

2) Approximate Linear Programming: Papadomanolakis and Ailamaki [94], Dash et al. [40] and Talebi et al. [125] all employed off-the-shelf LP solvers to find approximate solutions after formulating the offline index selection as LP problems.

Papadomanolakis et al. [95] generated several index configurations. Each index configuration is a set of candidate indexes. They represented each index configuration with one variable, pruned the search space by fixing one variable to 0 or 1, and obtained two sub-problems, which were solved by depth-first or breadth-first search, guided by heuristic rules.

CoPhy [40] applies soft constraints and Chord algorithm [41] to search for Pareto Optimal, balancing the index space and the algorithm runtime, but one variable represents one index in CoPhy, leading to fewer variables and faster speed than Papadomanolakis's. Talebi et al. [125] and Kllapi et al. [68] also employed LP techniques to recommend indexes for OLAP databases and data stream processing engines separately.

Despite their efficiency, approximate algorithms to LP formulations are not flexible to adapt to workload shifts.

3) Evolutionary Strategy: Many researchers [48], [67], [69], [70], [74], [89], [96] applied evolutionary strategies to offline index selection, where each candidate index (individual) is represented as binary strings (genome) and index benefit acts as the fitness function, quantifying how well the index contributes to query execution.

Unlike traditional optimization methods, GAs start the search from a group of feasible solutions instead of a single one and update through selection, crossover, and mutation until reaching the optimal criteria or the maximum number of iterations. Transitions in GAs are probabilistic rather than deterministic, enabling the search to jump out of local optima. Another advantage of GA is that it only requires the evaluation of solutions to determine their fitness, without the need to deduce knowledge from the original problem.

However, GA approaches haven't been applied in any real system as far as we know and not the focus of research on AIT because the evolving process takes a long computational time.

4) Reinforcement Learning Algorithms: Since 2015, a significant number of RL approaches have been explored for AIT. RL is a prominent paradigm in Machine Learning, used to describe and solve problems where an intelligent agent interacts with an environment to learn strategies that maximize rewards so as to achieve specific goals [103], [124].

To apply RL algorithms, index selection is commonly modeled as a Markov Decision Process (MDP) where indexes are recommended one by one at each step until constraints are reached.

We make a detailed comparison among different index selection methods based on RL. Table II presents a summarization of the algorithms as well as the designs of state, action, and reward for each RL method.

These methods can be categorized into offline and online approaches, depending on their ability to adapt to changing workloads. In this section, our primary focus is on RL-based offline index selection approaches. We will discuss online tuning methods in Section VI. For brevity, we include RL-based online index selection approaches in the table, because they share similar characters.

Basu et al. [17] were the first to apply RL to index selection[14] and their approach surpassed WFIT [113].

Sharma et al. [116][15], Lan et al. [77][16], Sharma et al. [118], Yan et al. [143], Wu et al. [140] all applied Deep Q-Networks (DQN) [86] for recommending indexes, from single-column B-tree index [116], to multi-column [77]. MANTIS [118] and DRLISA [143] can even recommend multiple types of indexes. Taking [77] as an example, the current state, represented as a one-dimensional array, is the input to the neural network, and the output is an array of the same dimension, where each value represents the Q-value for selecting the corresponding candidate index at the corresponding position in the array. The agent selects the index at the position with the maximum Q-value in a greedy manner, receives feedback from the DBMS, and updates its value function.

Welborn et al. [135] designed a structured action space and applied permutation learning with Sinkhorn Policy Gradient Algorithm [46] to encode the inductive bias of index selection task, addressing the instability of training and inefficiency of samples when applying Deep RL.

Instead of using neural networks to approximate reward values, Paludo et al. [92] designed a Q-learning algorithm with a linear function approximator and it is a rare research that measured benefits with actual query execution time[17].

Sadri et al. [105], [106] proposed DRLIndex, an index advisor for cluster databases utilizing RL. In this context,

[14]https://github.com/Debabrota-Basu/rCOREIL-Learning-to-Tune-Databases
[15]https://github.com/shankur/autoindex
[16]https://github.com/rmitbggroup/IndexAdvisor
[17]https://github.com/mir-pucrs/smartix-rl

TABLE II: Comparison among Index Selection Methods Based on RL

Algorithm	Method	Multi-column	Data update	Workload shifts	State	Action	Reward function	Terminal condition
Policy iteration	COREIL [17]	support	support	not support	index configurations built	changes of indexes	the negative of action cost	when parameters converge
Linear Q-learning	SmartIX [92]	not support	support	support	index configurations built	create or delete a one-column index	QPHH@SIZE	N/A
DQN	NoDBA [116]	not support	not support	not support	index configurations built and their selectivities	create a single-column index	reduction of query execution cost	limit on the index number
	Lan's DQN [77]	support	not support	not support	index configurations built	choose one candidate index and create it	relative reduction of workload execution cost	limit on index number
	DRLindex [105]	not support	not support	not support	index configurations built, workloads, access vectors, and index selectivities	create a single-column index on one replica	weighted sum of cost reduction and reciprocal of workload shift	limit on the index number
	MANTIS [118]	support	not support	not support	index configurations built	build an index of one type	reduction of cost and index space	limit on index space
	DRLISA [143]	support	support	not support	index configurations built and workloads	N/A	throughput increase minus index update cost	when no more performance improvement
Sinkhorn Policy Gradient	Welborn's index advisor [135]	support	not support	not support	current queries and history of building indexes	choose one candidate index and create it	ratio of cost reduction and space increase	N/A
PPO	SWIRL [72]	support	not support	support	workload features and meta-data	choose a candidate index	ratio of cost reduction and space increase	limit on index space
MCTS	BAIT [139]	support	not support	not support	index configurations built	choose a candidate index	percentage of performance improvement	limit on the number of what-if calls
	AutoIndex [150]	support	support	support	index configurations	choose an un-built index	cost reduction of read queries and cost increase of write queries	limit on index space
PPO-MC	Lai's PPO-MC [76]	support	not support	not support	index configurations built and workload selectivity matrix	choose one column to build index	reduction of cost	N/A
MAB	DBABandit [97]	support	not support	support	index configurations built and workloads	choose a super-arm	reduction of workload execution time	N/A
	HMAB [98]	support	not support	support	index configurations built and workloads	choose a super-arm	reduction of query execution time and time of index recommending	N/A

the advisor not only recommends indexes for replicas but also considers workload balance and generates route tables. Kossmann et al. reproduced the algorithm and compared it with their own proposed method[18] [72].

Lai et al. [76] trained an RL agent with PPO-MC (Proximal Policy Optimization - Monte Carlo) method [65], which has the advantages of fast convergence and reliable performance. Implemented on Kossmann et al.'s index selection evaluation platform, Lai's PPO-MC index advisor has a shorter training time and achieves some improvement in the effectiveness of index selection in comparison with Autoadmin [29], DB2 Advisor [131], and Relaxation [21].

Kossmann et al. proposed index advisor SWIRL [72] that utilizes Proximal Policy Optimization (PPO) [115] with Invalid Action Masking [60][19]. They maintained a vector representing the indexing of all columns but in a compressive way. Implemented on the platform [71], SWIRL outperforms

[18]https://github.com/hyrise/rl_index_selection/tree/main/experiments/drlinda_multi_attribute

[19]https://github.com/hyrise/rl_index_selection

Extend [109] in experiments, showing a significant reduction in execution time.

To accelerate index selection, Wu et al. [139] set limits on the number of what-if calls and the number of indexes. For a given index configuration and query pair, they derived cost from the minimum cost over all subset configurations with known what-if costs, assuming that including more indexes into a configuration does not increase the what-if cost[20]. In this case, the derived cost is essentially an upper bound on the what-if cost, and the index benefit function is a monotone submodular set function, which provides a theoretical foundation for their formulation of offline index selection as well as a lower bound on the benefit of indexes selected by their greedy algorithm on Monte Carlo Tree [19]. Experiments showcase the superiority of their method compared to DBA bandits [97] and NoDBA [116].

RL-based index selection methods optimize database index selection by capturing complex workload patterns and making intelligent recommendations through learned policies, often using deep neural networks. However, they rely on quality historical data, require significant computational resources and training time, and lack interpretability. In contrast, rule-based methods are interpretable but cannot handle all situations with a universal set of rules.

VI. Online Index Selection Algorithms

Compared to offline index selection, online index selection targets varying workloads, requiring a workload change detector or forecaster. The change detector identifies significant fluctuations in workloads or data patterns, while the forecaster predicts future workloads [9], [59] to preemptively select and build indexes, aiding with periodic queries . When significant changes are observed, the index configuration updater is triggered, working incrementally to add beneficial indexes and remove outdated ones, eliminating the need for a complete search from scratch each time.

Schnaitter and Polyzotis [112] proposed a benchmark for evaluating the performance of an online tuning algorithm in a principled fashion. Their workload suites are described in Section VII. In this section, we will introduce online index selection algorithms based on heuristics and RL.

A. Heuristic Approaches

Bruno and Chaudhuri [23] proposed an online predictive index selection approach based on a retrospective approach that finds optimal configurations offline. Knowing the entire sequence of queries in the workload, the optimal algorithm determines the optimal schedule by analyzing the cumulative benefits of candidate indexes for sub-sequences of queries. It creates an index if the future benefit of the index is promising and drops an index otherwise, based on whether the reduction in query execution cost outweighs the overhead of index update.

In Bruno and Chaudhuri's approach [23], the online algorithm uses past information to decide whether to create or drop an index based on inferred optimal strategies for previous queries. It tracks and compares the maximum and minimum benefits of indexes. If the optimal strategy would have taken a certain action, the online algorithm follows suit shortly after. When storage limits prevent building promising indexes, it replaces less promising ones with better options. Experiments show that this online algorithm performs competitively with the optimal strategy.

COLT [111] models the current workload based on incoming queries, divides the online workload into epochs, estimates index gains, and selects those providing the best performance within space constraints. To control overhead, COLT adjusts its budget for what-if calls at each epoch's end, increasing it when detecting workload shifts and decreasing it when the workload is stable. Experiments show COLT matches offline selection algorithms for stable workloads and outperforms them for evolving workloads. Running parallel to query execution, COLT is resilient to noise but only selects single-column indexes for simplicity.

AIM [142] is one of the few industrial strength index recommendation engines that is deployed on production databases at a large scale. It mainly focuses on index recommendation for static workload, but by running periodically it is also suitable for online index tuning.

Instead of modeling workload as a set of queries, Agrawal et al. [11] treat workload as a sequence of sets, where each set is a group of queries. The aim is to find a configuration sequence that minimize the workload sequence execution cost. They showed that finding the minimum sequence cost is equivalent to finding the shortest path in the possible configuration transition graph. Cost-based pruning and split-and-merge are applied to improve the efficiency of the algorithm.

Adaptive indexing [50], [51], database cracking [54], [61], [62], holistic indexing [99], and predictive indexing [15] are a line of recent research that build indexes online, partially and incrementally, during CPU idle time or query processing. They are mainly designed for column-store databases and fall under the category of new index structure design (specifically workload-adaptive, self-involving index structures), thus out of the scope of our survey. We mainly focus on tuning existing index structures to optimize workload performance.

B. Reinforcement Learning Approaches

Reinfocement Learning endows index advisors with the ability to intelligently adapt to changing workloads.

Perera et al. [97] introduced a self-driving approach for online index selection, referred to as DBABandit, which addresses the index selection problem as a sequential decision-making task and employs Multi-Armed Bandit (MBA) algorithm [132]. At each timestep t, the algorithm selects an arm (chooses new index configuration C_t) for new workload W_t based on the C^2UCB algorithm [101], in maximizing the cumulative reward. In balancing the exploration of unknown actions and exploitation of known optimal actions, DBABandit can provide regret bounds to ensure the effectiveness of indexes recommended online. In terms of convergence speed

[20]This assumption targets offline index selection and assumes no data update.

and performance stability, DBABandit outperforms other DRL approaches [21].

Perera et al. [98] proposed Hierarchical MAB (HMAB) to improve their previous work. HMAB uses a two-level bandit structure to handle large action spaces and enable parallel execution of the L1 bandit. The L1 bandits select candidate indexes, which serve as arms for the L2 bandit, with one group recommending database-wide views and another recommending table-specific indexes. Different contexts for each bandit group and the L2 bandit enhance decision-making. This hierarchical approach achieves a 96% improvement in index and view tuning for dynamic workloads compared to commercial tools[22].

Sharma et al. developed Indexer++ [117], an online index advisor using DQN and a pre-trained Transformer model to detect workload trend changes. Index++ treats similar workloads as unchanged if their trend centers overlap; otherwise, it triggers an index configuration update using DQN [77]. The approach clears the replay buffer, updates the DQN model with new data, merges workloads, and re-recommends indexes incrementally, avoiding retraining from scratch. Experiments on TPC-H and IMDB datasets showed Indexer++ dynamically adapts to workload changes and recommends appropriate indexes. The weakness of Indexer++ is that it ignores index maintenance costs.

Zhou et al. proposed AutoIndex [150] [23], an incremental index management system for openGauss [79]. AutoIndex matches incoming workloads with templates, extracts promising candidate indexes, and uses *Monte Carlo Tree Search* [19] to select high-benefit indexes. The input to the model includes information on the current workload and historical index statistics. When workload changes lead to decreased efficiency, the system decides whether it's necessary to update indexes. The policy tree is used to evaluate the current indexes and explore potential indexes that yield greater benefits, thus newly beneficial indexes will be added and obsolete indexes will be deleted at each step. In experiments conducted on the TPC-C and TPC-DS, indexes recommended by AutoIndex achieve better efficiency and throughput than baselines.

Currently, RL algorithms fail to be practical to be deployed in real system due to their expensive computation and long runtime, but they remain an interesting and promising direction for online index tuning.

C. Challenges of Online Index Selection

Online index selection faces two primary challenges:

1) Promptly determining what indexes to update as well as when and how in adaptation to workload changes.
2) Considering overheads of index construction and maintenance in addition to the benefit indexes bring to query execution.

For the first challenge, workload changes should be detected timely and the index selection algorithm should finish in an acceptable duration. Additionally, it needs to avoid oscillation (i.e., the same indexes are continuously created and dropped) if the selection algorithm reacts too quickly. For the second challenge, COLT [111], OnlinePT [23], WFIT [113], SOFT [81] involves calculating index overheads such as creation cost.

[21]https://github.com/malingaperera/DBABandits
[22]https://github.com/malingaperera/HMAB
[23]https://github.com/zhouxh19/AutoIndex

VII. Application of Index Tuning Algorithms

In this section, we first summarize commonly-used datasets in research on index tuning, and then take six databases as examples to introduce how index advisors are applied in real-world database systems.

A. Commonly-Used Datasets for Index Tuning

In papers on open-source systems or academic research of index tuning, common-used datasets include TPC-H, TPC-C, TPC-DS, JOB, SSB, and YCSB, as shown in Table III. To test index advisors' ability to handle skewed data distribution, skewed versions of benchmarks are also widely used [104], [106].

Some researchers use self-generated datasets to test performance on more complex workloads [139].

Schnaitter and Polyzotis [112] described how they design workload suites to test specific features of online index tuning algorithms. They varied the time period of phases, the complexity of workloads, the inclusion of data updates, and the stability of workloads, so as to test the agility at adapting to changes, the performance, the consideration of maintenance overheads, and the convergence of online index tuning algorithms.

B. Applications of Index Advisors in Major Databases

We list several major index tuning tools, comparing their differences and similarities.

1) Index Advisor for Microsoft SQL Server: It narrows down the candidate index space based on workload query features, recommends indexes for each query independently, and then iteratively merges single-column indexes into composite indexes with a given maximum index number as the constraint.

AutoAdmin [33], later evolved into Anytime [10], [31], [33], a.k.a. DTA [71]. In DTA, indexes can be enumerated with arbitrary width, no need to search narrower indexes before wider indexes as AutoAdmin does. The feature of Anytime is that the tuning can be interrupted at any time and provide temporarily optimal indexes, with the option to continue searching for better indexes if time permits.

There is also a practice report [39], which provides a detailed analysis of incorporating AIT in Microsoft Azure SQL Database. They utilized machine learning techniques to horizontally learn from all databases in Azure SQL Database and dynamically enhance the tuning actions. Employing DTA [31] as the selection algorithm, the index advisor results in about 82% CPU time improvement over DBAs on average in A/B testing experiments on Azure.

TABLE III: Commonly-Used Datasets for Index Tuning

Name	Scale	Data types	Database Property	Using Scenario
TPC-H [1]	8 tables, 61 columns, 22 query templates	structured data, primarily numeric, text, and date	a decision support benchmark, focusing on ad-hoc queries and concurrent data modifications.	used to evaluate the performance of decision support systems that analyze large volumes of data and execute complex queries.
TPC-DS [2]	25 tables, 429 columns, 99 query templates	structured data, including numeric, text, and dates.	a decision support benchmark, models several generally applicable aspects of a decision support system, including queries and maintenance.	covering various scenarios ranging from simple report generation, data mining, to complex OLAP, it provides relevant , objective performance data to industry users.
TPC-C [3]	nine types of tables with a wide range of record and population sizes	Structured data, including numeric, text, and dates.	a complex OLTP system benchmark, involving a mix of five concurrent transactions of different types and complexity either executed on-line or queued for deferred execution	portraying the activity of a wholesale supplier, representing any industry that must manage, sell, or distribute a product or service.
JOB [4][24]	113 query instances, 33 query templates, 21 tables, and 108 columns, based on IMDB[25]	structured data, primarily numeric and text.	focus on the performance of join operations, especially evaluating different join order strategies.	used to evaluate the efficiency of database systems in executing complex join queries.
SSB [91]	6 tables, 96 columns, 13 query templates	structured data, primarily numeric, text, and date	a simplified star schema data based on TPC-H	used to test the performance of multi-table JOIN query under star schema or to test the performance of the query engine
YCSB [38]	flexible schema, designed to work with large, distributed databases; data sizes can range from a few GB to several TB	key-value pairs, semi-structured data.	measures the performance of cloud-serving systems, focusing on CRUD operations (Create, Read, Update, Delete)	used to evaluate the performance of NoSQL databases and cloud data services, particularly for web applications.

2) Dexter for PostgreSQL: Dexter [14] is an open-source index advisor developed by Andrew Kane to automatically tune indexes for PostgreSQL [14]. It relies on HypoPG to create hypothetical indexes and pg_query to parse queries and extract workload features. By creating hypothetical indexes on unindexed columns, Dexter chooses the indexes with the most significant cost-saving, after comparing the query cost without index and with index. The drawback of Dexter is that the index maintenance costs are not considered and write-heavy tables need to be identified manually.

3) DB2 Advisor: DB2 Advisor [131] is used for index selection in DB2 database. As introduced in Section V, it models the offline index selection problem as a variant of the Knapsack Problem and contains two steps – candidate generation and greedy selection. One key advantage of DB2 Advisor is that it places the enumeration algorithm inside the optimizer, thus greatly reducing the number of optimizer calls.

4) Oracle Access Advisor: Oracle Access Advisor [8] is a SQL tuning tool for Oracle databases that recommends not only indexes but also materialized views and data partitioning. It selects candidate indexes based on column usage patterns in queries, identifies candidate indexes that effectively reduce workload execution time by executing test queries, and automatically creates or deletes indexes based on changes in the application workload. Users can specify the size of additional space for recommendations or filter out queries that satisfy certain conditions. Structural statistics about tables and indexes should be collected to improve the advisor's recommendations.

5) Index Advisor for OpenGauss: The embedded index advisor [78] of openGauss employs rule-based analysis methods to recommend indexes for a single query. As for a workload of queries, it will first compress the workload by templating and sampling to reduce the number of required functional calls, recommend indexes for each query template, estimate the benefit of each candidate index for the entire workload, and select candidate indexes greedily, e.g. in the decreasing order of their benefits.

The primary similarity among all these tools is that they all use the optimizer's hypothetical index utility and they greedily select candidate indexes according to their benefits.

These index advisor modules play a crucial role in automating index tuning processes in large-scale database management systems. It is noticeable that the complexity, big size, and inefficiency of neural network models are still hindering ML-based index selection algorithms from being deployed into real systems.

VIII. Future Research Directions

In this section, we present possible future research directions.

A. Preprocessing for Index Tuning

Preprocessing can be extremely important to improve the efficiency or quality of the following index selection. However, most of existing preprocessing methods only rely on manually-crafted rules so their benefits can be limited. Thus, we can adopt more advanced preprocessing methods to enhance the

semantic understanding capabilities for different tuning scenarios. First, generative models such as Large Language Models (LLMs) [47] have demonstrated excellent context understanding and code generating ability [47], [128], [151]. By employing these models to learn from database code, documents, and even historical queries and data, we can further automate the process of generating rules that guide candidate index generation, which currently rely on the specialized knowledge of human engineers. Second, we can leverage LLMs to understand relationships and extract features from query SQL statements and database schema information. These features, combined with the current index configuration, can serve as input to a neural network, generating index recommendations. Now that Trummer [129] can predict data correlations from column names with LLMs, it might be possible for LLMs to predict potential indexes.

B. Index Benefit Estimation

It is vital to improve the transferability of existing learned benefit estimation model. Hilprecht and Binning [56] have shown the feasibility of a zero-shot cost model that can generalize to different unseen database schemas. Therefore, it is a promising direction to devise an index estimation model that can provide high-quality estimation for unseen workloads and databases. Siddiqui et al. [121] call for researchers to design system-agnostic estimation and selection components for index tuning. The representation component should adapt to the heterogeneity of features varying across database engines. Only system-specific interaction APIs need to be implemented for the index tuner to work for new systems[26].

C. Index Selection

There are still several challenges in index selection, especially considering new database characteristics and complex correlated database mechanisms. First, the range of index types is extensive, such as LSM-trees [90], hashing. Current index selection algorithms primarily focus on B-tree indexes. Future research should intelligently recommend more suitable index types based on specific application scenarios, and various index structures (both learned indexes [73] and adaptive indexes [62]). Second, integrating index selection models with recommendations for other physical designs, such as buffer management [126], automatic compression [18], knob tuning [148] and shard selection [57], which is not explored enough but remains necessary and significant because the performance of physical designs, such as MVs and indexes, correlates with other system knobs and designs. For example, UDO has unified transaction code variants picking, index selection, and database system parameter tuning [133]. UniTune [147] has tuned index, knobs, and SQL query advisors together. However, further analysis of more unified framework requires exploring.

[26]Examples of system-specific APIs are parser, what-if calls, statistics getting etc. and system-agnostic index tuning components include planner, operations, search algorithms, and ML models [121].

D. Index Tuning from an Extensive Viewpoint

From an extensive viewpoint, index tuning should contain index selection, index materialization, index deletion, index suspension, invisible indexes [5], [110], and index defragmentation [35], [88], [130]. Through literature review, we find that current researches focus on directly adding or removing indexes, but lacks enough exploration in the three areas of improvement: (i) index materialization (when and how); (ii) the decision-making among index deletion[27], disabling[28] and invisibility[29]; and (iii) the timing for index defragmentation, including whether to rebuild[30] or reorganize the existing indexes. The key point of defragmentation decision-making is determining when index fragmentation level is too high: at this point, the benefits of defragmenting the index outweigh the costs associated with using an index that leads to high random read frequencies.

IX. Conclusions

This paper provides a comprehensive review of current research in Automatic Index Tuning. We give the definition and workflow of AIT, and summarize its framework of 3 modules: preprocessing, index benefit estimation, and index selection. We also discuss index types, index interaction, changing factors, and automation level of AIT, summarizing the development history of AIT in a figure. For each module of AIT, we categorize existing solutions and introduce typical approaches together with their advantages and limitations. Additionally, we discuss commonly-used datasets in AIT and applications of index advisors in major databases. We finally propose potential future directions in AIT research. We hope the survey provides a comprehensive summary of current research on AIT, inspirations for better approaches to AIT designs, and solutions for deploying AIT into database systems.

Acknowledgments

This paper was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (61925205, 62232009, 62102215), Zhongguancun Lab, Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist).

References

[1] https://www.tpc.org/tpch/.
[2] https://www.tpc.org/tpcds/.
[3] https://www.tpc.org/tpcc/.
[4] https://github.com/RyanMarcus/imdbpy.
[5] https://dev.mysql.com/doc/refman/8.0/en/invisible-indexes.html.

[27]If an index is deleted, it will be physically deleted.

[28]Disabling a clustered index on a table prevents access to the data. The data still remains in the table, but is unavailable for data manipulation language (DML) operations until the index is dropped or rebuilt. The index isn't maintained while it's disabled [6].

[29]If an index is made invisible, it won't be used by the planner, but it will still be maintained in case of data update. If the index should be used again, it can be made invisible and ready for immediate use.

[30]Index rebuilding, a.k.a. re-indexing [7], [87]. It deletes the index completely and build it from scratch, which is different from *index reorganizing*.

[6] https://learn.microsoft.com/en-us/sql/relational-databases/indexes/disable-indexes-and-constraints?view=sql-server-ver16.

[7] https://learn.microsoft.com/en-us/sql/relational-databases/indexes/reorganize-and-rebuild-indexes?view=sql-server-ver16#reorganize-an-index.

[8] Oracle base - sql access advisor in oracle database 10g. https://oracle-base.com/articles/10g/sql-access-advisor-10g.

[9] M. Abebe, H. Lazu, and K. Daudjee. Tiresias: enabling predictive autonomous storage and indexing. *VLDB Endowment*, 15(11):3126–3136, 2022.

[10] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005. In *SIGMOD*, pages 930–932, 2005.

[11] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD*, SIGMOD '06, page 683694. Association for Computing Machinery, 2006.

[12] P. Ameri, J. Meyer, and A. Streit. On a new approach to the index selection problem using mining algorithms. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2801–2810. IEEE, 2015.

[13] H. D. Anderson and P. B. Berra. Minimum cost selection of secondary indexes for formatted files. *ACM Trans. Database Syst.*, 2(1):6890, mar 1977.

[14] Ankane. The automatic indexer for postgres. https://github.com/ankane/dexter.

[15] J. Arulraj, R. Xian, L. Ma, and A. Pavlo. Predictive indexing. *arXiv preprint arXiv:1901.07064*, 2019.

[16] E. Barcucci and R. Pinzani. Optimal selection of secondary indexes. *IEEE Transactions on Software Engineering*, 16(1):32–38, 1990.

[17] D. Basu, Q. Lin, W. Chen, H. T. Vo, Z. Yuan, P. Senellart, and S. Bressan. Cost-model oblivious database tuning with reinforcement learning. In *Database and Expert Systems Applications*, pages 253–268. Springer, 2015.

[18] M. Boissier. Robust and budget-constrained encoding configurations for in-memory database systems. *VLDB*, 15(4):780–793, 2021.

[19] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[20] M. Brucato, T. Siddiqui, W. Wu, V. Narasayya, and S. Chaudhuri. Wred: Workload reduction for scalable index tuning. *Proc. ACM Manag. Data*, 2(1), mar 2024.

[21] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD*, pages 227–238, 2005.

[22] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, pages 826–835. IEEE, 2006.

[23] N. Bruno and S. Chaudhuri. Online autoadmin: (physical design tuning). In *SIGMOD*, pages 1067–1069, 2007.

[24] N. Bruno and R. V. Nehme. Configuration-parametric query optimization for physical design tuning. In *SIGMOD*, pages 941–952, 2008.

[25] A. Caprara, M. Fischetti, and D. Maio. Exact and approximate algorithms for the index selection problem in physical database design. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):955–967, 1995.

[26] A. Caprara and J. González. A branch-and-cut algorithm for a generalization of the uncapacitated facility location problem. *Top*, 4(1):135–163, 1996.

[27] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE TKDE*, 16(11):1313–1323, 2004.

[28] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing sql workloads. In *SIGMOD*, pages 488–499, 2002.

[29] S. Chaudhuri and V. Narasayya. Autoadmin "what-if" index analysis utility. *ACM SIGMOD Record*, 27(2):367–378, 1998.

[30] S. Chaudhuri and V. Narasayya. Index merging. In *ICDE*, pages 296–303, 1999.

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

[32] S. Chaudhuri, V. Narasayya, and P. Ganesan. Primitives for workload summarization and implications for sql. In *Proceedings 2003 VLDB Conference*, pages 730–741. Elsevier, 2003.

[33] S. Chaudhuri and V. R. Narasayya. An efficient, cost-driven index selection tool for microsoft sql server. In *VLDB*, volume 97, pages 146–155. Citeseer, 1997.

[34] S. Choenni, H. Blanken, and T. Chang. Index selection in relational databases. In *Proceedings of ICCI'93: 5th International Conference on Computing and Information*, pages 491–496. IEEE, 1993.

[35] C. Cioloca and M. Georgescu. Increasing database performance using indexes. *Database Systems Journal*, 2:13–22, 2011.

[36] D. Comer. The difficulty of optimum index selection. *ACM Transactions on Database Systems (TODS)*, 3(4):440–445, 1978.

[37] M. P. Consens, K. Ioannidou, J. LeFevre, and N. Polyzotis. Divergent physical design tuning for replicated databases. In *SIGMOD*, pages 49–60, 2012.

[38] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[39] S. Das, M. Grbic, I. Ilic, I. Jovandic, A. Jovanovic, V. R. Narasayya, M. Radulovic, M. Stikic, G. Xu, and S. Chaudhuri. Automatically indexing millions of databases in microsoft azure sql database. In *SIGMOD*, pages 666–679, 2019.

[40] D. Dash, N. Polyzotis, and A. Ailamaki. Cophy: a scalable, portable, and interactive index advisor for large workloads. 4(6):362372, mar 2011.

[41] C. Daskalakis, I. Diakonikolas, and M. Yannakakis. How good is the chord algorithm? *SIAM Journal on Computing*, 45(3):811–858, 2016.

[42] E. C. N. U. Data Science Engineering. Cedar. https://github.com/daseECNU/Cedar/blob/master/CEDAR_0.2/README-English.md.

[43] S. Deep, A. Gruenheid, P. Koutris, J. Naughton, and S. Viglas. Comprehensive and efficient workload compression. *Proc. VLDB Endow.*, 14(3):418430, nov 2020.

[44] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya. Ai meets ai: Leveraging query executions to improve index recommendations. In *SIGMOD*, pages 1241–1258, 2019.

[45] J. Dong and M. Han. Bittablefi: an efficient mining frequent itemsets algorithm. *Knowledge-Based Systems*, 20(4):329–335, 2007.

[46] P. Emami and S. Ranka. Learning permutations with sinkhorn policy gradient. *arXiv preprint arXiv:1805.07010*, 2018.

[47] L. Floridi and M. Chiriatti. GPT-3: its nature, scope, limits, and consequences. *Minds Mach.*, 30(4):681–694, 2020.

[48] F. Fotouhi and C. E. Galarce. Genetic algorithms and the search for optimal database index selection. In *Great Lakes CS Conference on New Research Results in Computer Science*, pages 249–255. Springer, 1989.

[49] J. Gao, N. Zhao, N. Wang, and S. Hao. Smartindex: An index advisor with learned cost estimator. In *CIKM*, CIKM '22, page 48534856. Association for Computing Machinery, 2022.

[50] G. Graefe, F. Halim, S. Idreos, H. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *Proc. VLDB Endow.*, 5(7):656667, mar 2012.

[51] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, page 371381. Association for Computing Machinery, 2010.

[52] L. Gruenwald, T. Winker, U. Çalikyilmaz, J. Groppe, and S. Groppe. Index tuning with machine learning on quantum computers for large-scale database applications. In *VLDB Workshops*, 2023.

[53] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for olap. In *ICDE*, pages 208–219. IEEE, 1997.

[54] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *Proc. VLDB Endow.*, 5(6):502513, feb 2012.

[55] C. Heeren, H. Jagadish, and L. Pitt. Optimal indexing using near-minimal space. In *SIGMOD*, pages 244–251, 2003.

[56] B. Hilprecht and C. Binnig. Zero-shot cost models for out-of-the-box learned cost prediction. *arXiv preprint arXiv:2201.00561*, 2022.

[57] B. Hilprecht, C. Binnig, and U. Röhm. Learning a partitioning advisor for cloud databases. In *SIGMOD*, SIGMOD '20, page 143157. Association for Computing Machinery, 2020.

[58] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):17351780, nov 1997.

[59] M. Holze, C. Gaidies, and N. Ritter. Consistent on-line classification of dbs workload events. In *CIKM*, CIKM '09, page 16411644. Association for Computing Machinery, 2009.

[60] S. Huang and S. Ontañón. A closer look at invalid action masking in policy gradient algorithms. *arXiv preprint arXiv:2006.14171*, 2020.

[61] S. Idreos, M. L. Kersten, S. Manegold, et al. Database cracking. In *CIDR*, volume 7, pages 68–78, 2007.

[62] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *VLDB Endowment*, 4(9):586–597, 2011.

[63] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. The data calculator: Data structure design and cost synthesis from first

principles and learned cost models. In *SIGMOD*, SIGMOD '18, page 535550. Association for Computing Machinery, 2018.

[64] M. Y. L. Ip, L. V. Saxton, and V. V. Raghavan. On the selection of an optimal set of indexes. *IEEE Transactions on Software Engineering*, (2):135–143, 1983.

[65] F. James. Monte carlo theory and practice. *Reports on progress in Physics*, 43(9):1145, 1980.

[66] I. Jimenez, H. Sanchez, Q. T. Tran, and N. Polyzotis. Kaizen: A semi-automatic index advisor. In *SIGMOD*, pages 685–688, 2012.

[67] R. Kain, D. Manerba, and R. Tadei. The index selection problem with configurations and memory limitation: A scatter search approach. *Computers & Operations Research*, 133:105385, 2021.

[68] H. Kllapi, I. Pietri, V. Kantere, and Y. E. Ioannidis. Automated management of indexes for dataflow processing engines in iaas clouds. In *EDBT*, pages 169–180, 2020.

[69] P. Kołaczkowski and H. Rybiński. Automatic index selection in rdbms by exploring query execution plan space. In *Advances in Data Management*, pages 3–24. Springer, 2009.

[70] M. Korytkowski, M. Gabryel, R. Nowicki, and R. Scherer. Genetic algorithm for database indexing. In *International Conference on Artificial Intelligence and Soft Computing*, pages 1142–1147. Springer, 2004.

[71] J. Kossmann, S. Halfpap, M. Jankrift, and R. Schlosser. Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms. *VLDB Endowment*, 13(12):2382–2395, 2020.

[72] J. Kossmann, A. Kastius, and R. Schlosser. Swirl: Selection of workload-aware indexes using reinforcement learning. In *EDBT*, pages 2–155, 2022.

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

[74] J. Kratica, I. Ljubić, and D. Tošić. A genetic algorithm for the index selection problem. In *Workshops on Applications of Evolutionary Computation*, pages 280–290. Springer, 2003.

[75] T. Lahdenmaki and M. Leach. *Relational Database Index Design and the Optimizers: DB2, Oracle, SQL Server, et al.* John Wiley & Sons, 2005.

[76] S. Lai, X. Wu, S. Wang, Y. Peng, and Z. Peng. Learning an index advisor with deep reinforcement learning. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data*, pages 178–185. Springer, 2021.

[77] H. Lan, Z. Bao, and Y. Peng. An index advisor using deep reinforcement learning. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 2105–2108, 2020.

[78] G. Li. *Source code analysis of openGauss Database*. Tsinghua University Press, 2021.

[79] G. Li, X. Zhou, J. Sun, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li. Opengauss: An autonomous database system. *Proc. VLDB Endow.*, 14(12):30283042, jul 2021.

[80] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[81] M. Luhring, K.-U. Sattler, K. Schmidt, and E. Schallehn. Autonomous management of soft indexes. In *ICDEW*, pages 450–458. IEEE, 2007.

[82] V. Y. Lum and H. Ling. An optimization problem on the selection of secondary keys. In *Proceedings of the 1971 26th annual conference*, pages 349–356, 1971.

[83] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In *SIGMOD*, pages 631–645, 2018.

[84] L. Ma, W. Zhang, J. Jiao, W. Wang, M. Butrovich, W. S. Lim, P. Menon, and A. Pavlo. Mb2: Decomposed behavior modeling for self-driving database management systems. In *SIGMOD*, SIGMOD '21, page 12481261. Association for Computing Machinery, 2021.

[85] R. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow.*, 12(11):17331746, jul 2019.

[86] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[87] E. Morelli, A. Almeida, S. Lifschitz, J. M. Monteiro, and J. Machado. Autonomous re-indexing. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, page 893897. Association for Computing Machinery, 2012.

[88] V. Narasayya and M. Syamala. Workload driven index defragmentation. In *ICDE*. IEEE, March 2010.

[89] P. Neuhaus, J. Couto, J. Wehrmann, D. D. A. Ruiz, and F. R. Meneguzzi. Gadis: A genetic algorithm for database index selection. In *The 31st International Conference on Software Engineering & Knowledge Engineering, 2019, Portugal.*, 2019.

[90] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[91] P. E. ONeil, E. J. ONeil, and X. Chen. The star schema benchmark (ssb). *Pat*, 200(0):50, 2007.

[92] G. Paludo Licks, J. Colleoni Couto, P. de Fátima Miehe, R. De Paris, D. Dubugras Ruiz, and F. Meneguzzi. Smartix: A database indexing agent based on reinforcement learning. *Applied Intelligence*, 50(8):2575–2588, 2020.

[93] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[94] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *ICDEW*, pages 442–449. IEEE, 2007.

[95] S. Papadomanolakis, D. Dash, and A. Ailamaki. Efficient use of the query optimizer for automated physical design. In *VLDB Endowment*, pages 1093–1104, 2007.

[96] W. G. Pedrozo, J. C. Nievola, and D. C. Ribeiro. An adaptive approach for index tuning with learning classifier systems on hybrid storage environments. In *International conference on hybrid artificial intelligence systems*, pages 716–729. Springer, 2018.

[97] R. M. Perera, B. Oetomo, B. I. Rubinstein, and R. Borovica-Gajic. Dba bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In *ICDE*, pages 600–611. IEEE, 2021.

[98] R. M. Perera, B. Oetomo, B. I. Rubinstein, and R. Borovica-Gajic. Hmab: self-driving hierarchy of bandits for integrated physical database design tuning. *VLDB Endowment*, 16(2):216–229, 2022.

[99] E. Petraki, S. Idreos, and S. Manegold. Holistic indexing in main-memory column-stores. In *SIGMOD*, SIGMOD '15, page 11531166. Association for Computing Machinery, 2015.

[100] G. Piatetsky-Shapiro. The optimal selection of secondary indices is np-complete. *ACM SIGMOD Record*, 13(2):72–75, 1983.

[101] L. Qin, S. Chen, and X. Zhu. Contextual combinatorial bandit and its application on diversified online recommendation. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, pages 461–469. SIAM, 2014.

[102] T. Qiu, B. Wang, Z. Shu, Z. Zhao, Z. Song, and Y. Zhong. Intelligent index tuning approach for relational databases. *Journal of Software*, 31(3):634–647, 2020.

[103] X. Qiu. *Neural Networks and Deep Learning*. CHina Machine Press, 2019.

[104] Z. Sadri and L. Gruenwald. A divergent index advisor using deep reinforcement learning. In *International Conference on Database and Expert Systems Applications*, pages 139–152. Springer, 2022.

[105] Z. Sadri, L. Gruenwald, and E. Lead. Drlindex: deep reinforcement learning index advisor for a cluster database. In *Proceedings of the 24th Symposium on International Database Engineering & Applications*, pages 1–8, 2020.

[106] Z. Sadri, L. Gruenwald, and E. Leal. Online index selection using deep reinforcement learning for a cluster database. In *ICDEW*, pages 158–161. IEEE, 2020.

[107] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[108] M. Schkolnick. The optimal selection of secondary indices for files. *Information Systems*, 1(4):141–146, 1975.

[109] R. Schlosser, J. Kossmann, and M. Boissier. Efficient scalable multi-attribute index selection using recursive strategies. In *ICDE*, pages 1238–1249. IEEE, 2019.

[110] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. Colt: Continuous on-line tuning. In *SIGMOD*, SIGMOD '06, page 793795. Association for Computing Machinery, 2006.

[111] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. In *ICDEW*, pages 459–468. IEEE, 2007.

[112] K. Schnaitter and N. Polyzotis. A benchmark for online index selection. In *ICDE*, pages 1701–1708. IEEE, 2009.

[113] K. Schnaitter and N. Polyzotis. Semi-automatic index tuning: keeping dbas in the loop. *Proc. VLDB Endow.*, 5(5):478489, jan 2012.

[114] K. Schnaitter, N. Polyzotis, and L. Getoor. Index interactions in physical design tuning: modeling, analysis, and applications. *VLDB Endowment*, 2(1):1234–1245, 2009.

[115] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
[116] A. Sharma, F. M. Schuhknecht, and J. Dittrich. The case for automatic database administration using deep reinforcement learning. *arXiv preprint arXiv:1801.05643*, 2018.
[117] V. Sharma and C. Dyreson. Indexer++ workload-aware online index tuning with transformers and reinforcement learning. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 372–380, 2022.
[118] V. Sharma, C. Dyreson, and N. Flann. Mantis: Multiple type and attribute index selection using deep reinforcement learning. In *25th International Database Engineering & Applications Symposium*, pages 56–64, 2021.
[119] J. Shi, G. Cong, and X.-L. Li. Learned index benefits: Machine learning based index performance estimation. *Proc. VLDB Endow.*, 15(13):39503962, sep 2022.
[120] T. Siddiqui, S. Jo, W. Wu, C. Wang, V. Narasayya, and S. Chaudhuri. Isum: Efficiently compressing large and complex workloads for scalable index tuning. In *SIGMOD*, pages 660–673, 2022.
[121] T. Siddiqui and W. Wu. Ml-powered index tuning: An overview of recent progress and open challenges. *SIGMOD Rec.*, 52(4):1930, jan 2024.
[122] T. Siddiqui, W. Wu, V. Narasayya, and S. Chaudhuri. Distill: low-overhead data-driven techniques for filtering and costing indexes for scalable index tuning. *VLDB Endowment*, 15(10):2019–2031, 2022.
[123] J. Sun and G. Li. An end-to-end learning-based cost estimator. *Proc. VLDB Endow.*, 13(3):307319, nov 2019.
[124] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
[125] Z. A. Talebi, R. Chirkova, and Y. Fathi. An integer programming approach for the view and index selection problem. *Data & Knowledge Engineering*, 83:111–125, 2013.
[126] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. Tung. A new approach to dynamic self-tuning of database buffers. *ACM Transactions on Storage (TOS)*, 4(1):1–25, 2008.
[127] Q. T. Tran, I. Jimenez, R. Wang, N. Polyzotis, and A. Ailamaki. Rita: An index-tuning advisor for replicated databases. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2015.
[128] I. Trummer. DB-BERT: A database tuning tool that "reads the manual". In Z. G. Ives, A. Bonifati, and A. E. Abbadi, editors, *SIGMOD*, pages 190–203. ACM, 2022.
[129] I. Trummer. Can large language models predict data correlations from column names? 16(13):43104323, sep 2023.
[130] M. Valavala and D. W. Alhamdani. A survey on database index tuning and defragmentation. 2020.
[131] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, pages 101–110. IEEE, 2000.
[132] J. Vermorel and M. Mohri. Multi-armed bandit algorithms and empirical evaluation. In *European conference on machine learning*, pages 437–448. Springer, 2005.
[133] J. Wang, I. Trummer, and D. Basu. Demonstrating udo: A unified approach for optimizing transaction code, physical design, and system parameters via reinforcement learning. In *SIGMOD*, SIGMOD '21, page 27942797. Association for Computing Machinery, 2021.
[134] J. Wehrstein, B. Hilprecht, B. Olt, M. Luthra, and C. Binnig. The case for multi-task zero-shot learning for databases. In *4th International Workshop on Applied AI for Database Systems and Applications (AIDB) colocated with VLDB 2022*, pages 1–8, 2022.
[135] J. Welborn, M. Schaarschmidt, and E. Yoneki. Learning index selection with structured action spaces. *arXiv preprint arXiv:1909.07440*, 2019.
[136] K.-Y. Whang. Index selection in relational databases. In *Foundations of Data Organization*, pages 487–500. Springer, 1987.
[137] K.-Y. Whang et al. Separabilityan approach to physical database design. *IEEE transactions on computers*, 100(3):209–222, 1984.
[138] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, pages 1081–1092, 2013.
[139] W. Wu, C. Wang, T. Siddiqui, J. Wang, V. Narasayya, S. Chaudhuri, and P. A. Bernstein. Budget-aware index tuning with reinforcement learning. In *SIGMOD*, pages 1528–1541, 2022.
[140] Y. Wu, Y. Zhang, and N. Li. Index advisor via dqn with invalid action mask in tree-structured action space. In *ADMA 2022*, pages 434–445. Springer, 2022.
[141] XIAOMI. Soar. https://github.com/XiaoMi/soar.
[142] R. Yadav, S. R. Valluri, and M. Zaït. Aim: A practical approach to automated index management for sql databases. In *ICDE*, pages 3349–3362, 2023.
[143] Y. Yan, S. Yao, H. Wang, and M. Gao. Index selection for nosql database with deep reinforcement learning. *Information Sciences*, 561:20–30, 2021.
[144] W. Yang, H. Hu, H. Duan, Y. Hu, and W. Qian. Cedaradvisor: A load-adaptive automatic indexing recommendation tool. *Journal of East China Normal University (Natural Science)*, 2020(6):52, 2020.
[145] T. Yu, Z. Zou, W. Sun, and Y. Yan. Refactoring index tuning process with benefit estimation. *Proc. VLDB Endow.*, 17(7):1528–1541, 2024.
[146] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han. Automatic view generation with deep learning and reinforcement learning. In *ICDE*, pages 1501–1512. IEEE, 2020.
[147] X. Zhang, Z. Chang, H. Wu, Y. Li, J. Chen, J. Tan, F. Li, and B. Cui. A unified and efficient coordinating framework for autonomous dbms tuning. 1(2), jun 2023.
[148] X. Zhao, X. Zhou, and G. Li. Automatic database knob tuning: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 35(12):1247012490, apr 2023.
[149] Y. Zhao, G. Cong, J. Shi, and C. Miao. Queryformer: a tree transformer model for query plan representation. *VLDB Endowment*, 15(8):1658–1670, 2022.
[150] 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*, pages 2196–2208. IEEE, 2022.
[151] X. Zhou, Z. Sun, and G. Li. Db-gpt: Large language model meets database. *Data Science and Engineering*, pages 1–10, 2024.
[152] R. Zhu, H. Wang, Y. Tang, and B. Xu. Almss: Automatic learned index model selection system. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data*, pages 440–445. Springer, 2021.
[153] C. L. Z. B. G. L. T. W. Zijia Wang, Haoran Liu. Leveraging dynamic and heterogeneous workload knowledge to boost the performance of index advisors. *Proc. VLDB Endow.*, 17(7):1642–1654, 2024.

Yang Wu received his bachelor's degree in Computer Science from Northwestern Polytechnical University in 2022. He is currently working towards the PhD degree in the Depertment of Computer Science, Tsinghua University, Beijing, China. His research interests lie in database optimization, especially in the area of AI for databases (AI4DB).

Xuanhe Zhou received his PhD degree in the Department of Computer Science from Tsinghua University, Beijing, China in 2024. His research interests lie in intelligent database optimization and data governance for AI.

Yong Zhang is currently working as an associate research fellow in Tsinghua University, Beijing, China. He received both his bachelor's degree and his PhD degree in the Department of Computer Science, Tsinghua University, in 1997 and in 2002. His research interests include database, management and analysis of big data, and intelligent healthcare.

Guoliang Li is currently working as a professor in the Department of Computer Science, Tsinghua University, Beijing, China. He received his PhD degree in Computer Science from Tsinghua University, Beijing, China in 2009. His research interests mainly include data cleaning and integration, spatial databases, crowdsourcing, and AI &DB co-optimization.