ML-Powered Index Tuning: An Overview of Recent Progress and Open Challenges

Tarique Siddiqui Wentao Wu
Microsoft Research
{tasidd, wentwu}@microsoft.com

ABSTRACT

The increasing scale and complexity of workloads in modern cloud services highlight a crucial challenge in automated index tuning: recommending high-quality indexes while ensuring scalability. This is further complicated by the need for these automated solutions to minimize query performance regressions in production deployments. This paper directs attention to some of these challenges in automated index tuning and explores ways in which machine learning (ML) techniques provide new opportunities in their mitigation. In particular, we reflect on our recent efforts in developing ML techniques for workload selection, candidate index filtering, speeding up index configuration search, reducing the amount of query optimizer calls, and lowering the chances of performance regressions. We highlight the key takeaways from these efforts and underline the gaps that need to be closed for their effective functioning within the traditional index tuning framework. Additionally, we present a preliminary cross-platform design aimed at democratizing index tuning across multiple SQL-like systems—an imperative in today's continuously expanding data system landscape. We believe our findings will help provide context and impetus to the research and development efforts in automated index tuning.

1. INTRODUCTION

Automated index tuning improves the performance of databases by recommending indexes that accelerate query execution. There has been extensive research over the past decades [23,30], and *index tuners* have been developed for both commercial and open-source database systems [14, 15, 29, 65].

Figure 1 presents the typical architecture of such an index tuner [14, 15, 65]. It contains three major components: (1) *workload parsing/analysis*, where an input workload (of SQL queries) is parsed and analyzed; (2) *candidate index generation*, which identifies a set of candidate indexes for each query in the workload; and (3) *configuration enumeration*, which searches for an in-

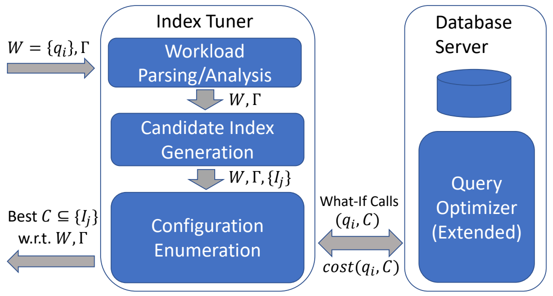

Index Tuner
$W = \{q_i\}, \Gamma$
Workload Parsing/Analysis
W, Γ
Candidate Index Generation
$W, \Gamma, \{I_j\}$
Configuration Enumeration
Best $C \subseteq \{I_j\}$ w.r.t. W, Γ
What-If Calls (q_i, C)
$cost(q_i, C)$
Database Server
Query Optimizer (Extended)

Figure 1: The architecture of an index tuner, where W is the input workload and $q_i \in W$ is a single SQL query, Γ is a set of tuning constraints, $\{I_j\}$ is the set of candidate indexes generated for W, and $C \subseteq \{I_j\}$ represents an index configuration during enumeration.

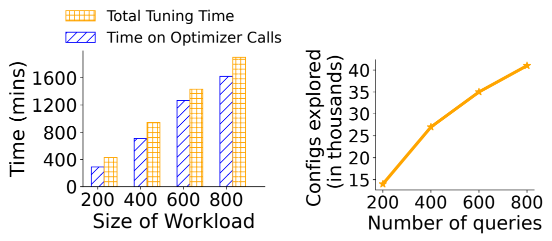
dex configuration from the candidate indexes that meets the user-specified tuning constraints (e.g., the maximum number of indexes allowed or the total amount of storage taken by the indexes) while minimizing the total cost of the workload.[1] For a configuration C considered during enumeration, the index tuner leverages the *what-if API*, an extended functionality of the query optimizer, to estimate the cost of each query on top of C *without* actually building the indexes contained by C [16]. We refer to such query optimizer calls as "what-if (optimizer) calls" in this paper. A what-if call can be time-consuming since it needs to invoke the query optimizer, especially for complex queries.

Despite this success, the recent advances in data management have highlighted the existing challenges and posed new ones. We discuss three key problems.

Problem #1: *The growing scale and complexity of database SQL query workloads in modern cloud environments affect the quality of recommended indexes and contribute to increased time, cost, and resource overheads for index tuning.*

Cloud database services, such as Microsoft's Azure SQL Database [1], host millions of databases with large and complex query workloads. Automatically and ef-

[1] A configuration is defined as a set of indexes.



(a) Tuning time (b) Configurations explored

Figure 2: The growth in tuning time and configuration exploration on increasing workload size.

ficiently tuning indexes at that scale and complexity is a formidable task. In particular, the scalability of index tuning depends on (1) the number of queries in the workload, (2) the number of candidate indexes and resulting configurations that are enumerated, and (3) the number of optimizer invocations or what-if calls. As depicted in Figure 2, we see that the tuning time for a state-of-the-art index advisor [14] grows significantly as we increase the size of the workload. This is primarily because the space of configurations to explore increases (Figure 2b), resulting in a large number of expensive what-if calls (consuming 70% to 80% of the overall tuning time).

Problem #2*: Minimal DBA monitoring and the potential impact on larger workloads in the cloud environments underscores the imperative to mitigate performance regressions stemming due to recommended indexes by index tuners.*

A major impediment to the goal of full automation and scalability is the requirement that index implementations should not cause significant query performance regressions [18]. One important reason for query performance regression (QPR) is that index tuners use query optimizer's cost model (via what-if calls) to measure the improvement in query performance (e.g., execution time) due to recommended indexes [15, 16, 65]. While cost models are much more efficient than directly executing queries, they may not accurately capture the runtime behavior of queries, resulting in a mismatch between the actual and estimated query performance. The issue is further aggravated due to the scale, variety, and complexity of workloads, which make it hard to collect sufficient statistics or incorporate mechanisms for automatically identifying and fixing QPR [18].

Problem #3*: The current approach of building system-specific and tightly-coupled index tuners is less tenable in today's fast-expanding landscape of rapidly growing number and variety of data systems.*

Modern enterprises manage several data systems, each optimized for different use-cases, and frequently add new ones. Data could reside in a variety of locations, e.g., operational stores, data warehouses, or data lakes [3, 46, 47]. Interestingly, only a limited number of database systems, such as Oracle, Microsoft SQL Server, IBM DB2, and PostgreSQL, support index tuning [14, 15, 29, 65]. This is surprising given that the process of index tuning is largely system-independent, with core components such as candidate index generation and configuration search algorithms reusable across systems with minimal changes. Yet, index tuners today are tightly coupled with specific database systems, and developing an index tuner for a new or evolving database system requires massive engineering efforts.

1.1 Paper Overview

In this paper, we reflect on the recent efforts towards addressing the above challenges. While improving the scalability of index tuning and addressing query performance regressions are not new problems, the recent focus has largely been towards leveraging ML-powered techniques that can *efficiently* identify useful configurations without sacrificing the quality of recommendations. Another notable difference compared to prior work is that ML techniques require minimal changes to the underlying query optimizer or to the database system, and can potentially be integrated as "bolt-on" component(s) within existing time-tested and commercially deployed index tuning architectures [14].

Opportunity: *ML-powered techniques have the potential to interoperate with core index tuning components to improve the scalability and reduce query performance regressions, without significant changes to the index tuning architecture, the query optimizer, or the database system.*

Figure 3 outlines an enhanced version of the index tuning architecture depicted in Figure 1 after incorporating ML-based data-driven techniques. It introduces novel software components and functionalities that improve the performance of the end-to-end index tuning workflow: (1) *workload selection* that aims to reduce the size, complexity, and relevance of the input workload; (2) *learned index filter* that aims to prune spurious candidate indexes with little impact on query performance; (3) *MCTS-based enumerator* that aims to improve the effectiveness of index configuration enumeration; (4) *learned cost models* that aim to reduce the number of what-if calls; and (5) *ML-based performance regression predictor* that aims to reduce the chance of query performance regression. We provide an overview of these new functionalities below, and the rest of this paper covers more details of each functionality as well as discussions on the opportunities and open challenges based on lessons learnt from our own experiences.

Workload Selection. We focus on two complementary sub-problems of *workload compression* and *workload*

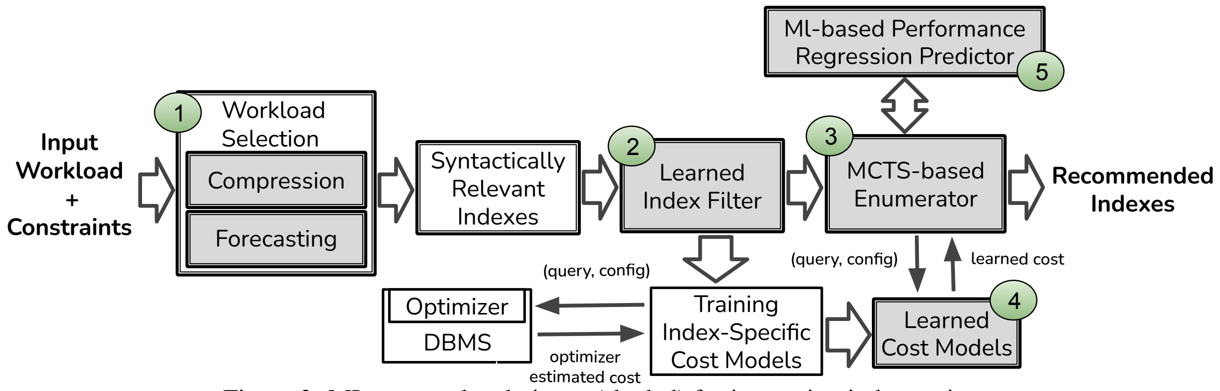


Figure 3: ML-powered techniques (shaded) for improving index tuning.

forecasting. Workload compression selects a small subset of queries from a large input workload, tuning which has the potential to result in as high-quality recommendations as tuning the entire workload. Workload forecasting, on the other hand, predicts arrival rate of queries for *just-in-time* recommendation of indexes, reducing the size of workload that needs to be tuned at given point as well as improving the relevance of recommended indexes for queries in the near future (Section 2).

Learned Index Filter. After selecting SQL queries for tuning, the index tuner parses and analyzes the queries to generate *synthetically relevant indexes* based on *indexable columns* [15] (e.g., columns that appear in filter and join predicates appearing in the *where* clause, as well as columns that appear in the *group-by* and *order-by* clauses). It then tries to identify candidate indexes from the syntactically relevant indexes. Many of such candidate indexes are turned out to be *spurious*, meaning that they have little impact on query performance and can be safely pruned. A learned index filter is developed based on this observation (Section 3.1).

MCTS-based Enumerator. As mentioned, configuration enumeration aims to find the best configuration from the candidate indexes provided. A classic approach to configuration enumeration is *greedy search* [15], which suffers from scalability problems when facing a large search space with many candidate indexes and queries. The MCTS-based enumerator aims to improve the effectiveness of configuration enumeration in large search space by identifying configurations that show promise and potential early on. It leverages reinforcement learning (RL) techniques internally (Section 3.2).

Learned Cost Models. The what-if calls used by index tuner can be expensive, especially when facing large and complex workloads. One important observation we made is that many queries and configurations explored during configuration enumeration are *similar*. This opens up the door of leveraging ML techniques to learn in-situ lightweight cost models for clusters of similar queries and configurations during configuration enumeration, despite the fact that learning a generic cost model is extremely challenging. We can significantly reduce the number of what-if calls by delegating many of them to the cost models learned (Section 3.3).

ML-based Performance Regression Predictor. The what-if calls used by the index tuner rely on the query optimizer's estimated costs, which can be off from the actual query execution time and result in QPR. An ML-based QPR predictor trained on top of query execution data can forecast and therefore avoid QPR firsthand. We highlight the challenges of addressing QPR for production systems, giving an overview of recent efforts and the unsolved challenges that remain open (Section 4).

Cross-platform Index Tuner. Finally, to democratize the ML-powered index tuning techniques over multiple systems, we discuss the problems with the current approach of developing *system-specific* index tuners in today's expanding data system landscape. Towards addressing this, we propose an architecture for a *cross-platform* index tuner, along with abstractions that will allow (the same) index tuning technologies to simultaneously benefit many data systems (Section 5).

1.2 Scope and Limitations

Our primary focus in this paper is on improving the classical *offline* index tuning process as used in commercial tools (e.g., [9, 14, 15, 19, 29, 50, 65, 69]), and the adapted versions of them have also been deployed in modern cloud database services [18]. Notably, there has been significant research efforts on *online* index tuning techniques [6, 10, 11, 41, 43, 44, 48, 49, 52, 53, 54], where the index tuner can create/drop indexes *on the fly* to handle workload and data drifts. However, perhaps due to the inherent complexity and variety that comes with dynamic, ad-hoc, and non-stationary workloads, a consensus has not yet been reached on critical open questions of online index tuning such as the architecture, the optimization problem formulation, the optimality guarantee of the recommended indexes, and the performance evaluation criteria. Consequently, to the best of our understanding, such techniques have yet to find substantial adoption in commercial systems.

Meanwhile, there is a line of recent efforts on using ML for holistic database (knob) tuning (e.g., [5, 34, 63, 66, 68, 76, 77, 80]) that goes beyond the scope of index tuning and therefore this paper. There is also lots of related work on using ML for improving other specific aspects or components of database systems, such as physical data layout (e.g., [26, 74]), buffer pool size (e.g., [62]), and query optimizer (e.g., [38, 39, 64, 75, 78]), which we omit in this paper as well.

Moreover, there are common challenges faced by applying ML techniques to solving data management problems that are not restricted to index tuning per se. There has been recent work on addressing such general challenges, such as reducing the overhead of generating training data [67] and dealing with data updates/drifts [32]. An in-depth discussion on these issues is worthwhile but beyond the scope of this paper.

2. WORKLOAD SELECTION

The focus of workload selection has been in two directions: 1) selecting queries to tune, referred to as *workload compression*, and 2) knowing when a query will arrive, referred to as *workload forecasting*. We discuss representative research efforts in each direction.

2.1 Workload Compression

A key factor affecting the scalability of index tuning is the number of SQL queries in the workload. In a typical cloud database service, a workload can contain hundreds or even thousands of queries. Tuning such a large workload in a reasonable amount of time is challenging. It is therefore natural to ask whether index tuning can be sped up significantly by finding a *substitute* workload of smaller size while qualitatively not degrading the result of the application. It is crucial that this compressed workload can be found *efficiently*; otherwise, the very purpose of compression is negated.

Prior workload compression techniques based on sampling and clustering [13,20] often fail to effectively capture the similarity between queries and miss out less frequent queries that may lead to substantial improvement in performance due to indexes. Furthermore, real workloads have typically more variety in query structures, which makes identifying relevant queries more challenging. To address these issues, we have developed ISUM, an indexing-aware and efficient workload summarization technology [58]. ISUM employs two main techniques to identify relevant queries.

Measuring Potential Improvement: We develop a new technique to efficiently estimate the potential in performance improvement of a query due to indexes *without* requiring optimizer calls, which are key scalability bottlenecks. Our idea is to leverage statistics such as table size, selectivity, and costs of queries while eschewing parts of query optimization unrelated to indexing, to estimate improvement so that it is *highly correlated* with the optimizer estimated improvements.

Capturing Indexing-aware Similarity: On selecting a query, it is also important to quantify the improvement in performance on unselected queries in the workload due to indexes from the selected query. We represent each query as a set of features (derived from *indexable columns* [15]) such that two queries with similar features will likely result in similar sets of indexes. We weigh the features using statistics to capture their relevance to indexes. For instance, features on larger tables are more important, and similarly, the importance of indexable columns can vary depending on whether they occur as part of the filter or join predicates. We can further leverage ML techniques to automatically derive the weights of the features based on table size, selectivity, and position of the columns. Our feature representation also allows us to quantify the similarity between queries with different structures.

Combining both techniques, we measure the improvement due to each query over the entire workload, and develop a *linear-time* algorithm that selects queries in decreasing order of their estimated improvement.

Takeaway #1*: A workload compression technique for scalable index tuning requires efficient estimation of (1) potential performance improvement due to indexes, and (2) indexing-aware similarity between queries, both using minimal optimizer calls (a key scalability bottleneck).*

Figure 4 presents an example of running ISUM on the TPC-DS benchmark workload. Overall, we observe that ISUM can lead to a median of 1.4× and a maximum of 2× performance improvements compared to prior techniques for the same compressed workload sizes. Furthermore, given an input workload consisting of queries along with their costs, the time to select the compressed workload is small (<1%) compared to the tuning time of the compressed workload [58].

Open Challenge #1*: The benefits of shortened tuning time gained from compression is often offset by the overhead involved in parsing queries, gathering statistics, and assessing the improvements brought by recommended indexes across the entire input workload.*

Workload compression techniques, including ISUM, require that statistics such as selectivity, optimizer estimated cost of each query, and other physical plan characteristics are provided as input. We observe that most

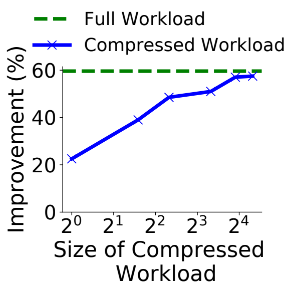


Figure 4: Workload compression on TPC-DS.

database systems expose functionality to collect such information. For database systems where such information is not available, we need to make an optimizer call for each query in the workload, which is expensive for large input workloads.

Open Challenge #2*: Existing workload compression methods focus on specific optimization goals, but there is a need for a more adaptable workload characterization approach that allows for ad-hoc constraints and user-directed query selection.*

Workload compression techniques use pre-defined criteria for selecting queries. However, in practice, one may also want to obtain a representative subset with varying constraints, e.g., 100 most expensive queries while ensuring that every table in the database occurs in at least 5 queries, consuming at least a certain fraction of resources such as CPU and I/O. Thus, the specification for picking a representative subset of a workload depends on the task at hand and requires varying criteria and optimization goals. Additionally, it is crucial to characterize compressed workloads for interpretability. One direction to explore is to report the estimated improvement and drill-downs on how each query in the compressed workload represents queries in the workload that *were not tuned*. Altogether, tighter integration of workload characterization mechanisms into a traditional index tuning engine and their evaluation for a broader set of tasks is an interesting area for future work.

2.2 Workload Forecasting

Workload forecasting allows index tuners to make just-in-time recommendations for the workload expected to arrive in near future. Furthermore, workload forecasting can reduce the number of queries that index tuners need to analyze in each cycle.

As one of the representative works, Ma et al. [37] develop a workload forecasting technique and leverage it to improve index tuning. It uses a two-phase framework. In the first phase, raw queries are pre-processed and clustered based on query templates (i.e., query instances without parameter binding). Clustering is necessary, as it is computationally infeasible to build models to capture and predict the arrival patterns for each template. In the next phase, an ML-based forecasting model is trained for each cluster that predicts how many queries the application will execute in the future (e.g., one hour from now, one day from now, etc.).

Takeaway #2*: Predicting arrival rates of queries in near-future can help reuse traditional offline index tuners for scalable and just-in-time index selection.*

Workload forecasting partially mitigates the inability of offline index tuning in handling dynamic workloads (a core focus of online index tuning [11]) while reusing the offline index tuners. The empirical findings show that when using forecasting, the throughput and latency of MySQL executing real workloads improve by 5× and 78% over the 16-hour period when the indexes are added or removed after every hour. Similarly, over PostgreSQL, the technique achieves 180× better throughput and 99% better latency [37].

Open Challenge #3*: A more holistic forecasting of future workloads, combining both arrival times as well as query instances (e.g., predicate values), is desired to enhance the quality of index recommendations.*

Prior work on index tuning as well as workload forecasting assumes that the query expressions remain unchanged over time. However, the recommended indexes may be sub-optimal when the expressions themselves evolve over time, e.g., a recurring analytical query that looks at last two days of sales data, or a query template that changes bindings based on the day the query runs or the same query template used by different teams with different parameter bindings. Our analysis of enterprise workloads shows that while literal values may change over time, there are high-level patterns that can be learnt to predict the potential bindings in advance. Thus, an interesting direction for future work is to predict entire query instances in addition to the arrival times. There has also been recent work along the line of *robust index selection* [51], with the idea of selecting indexes that are optimal considering the dynamic nature of the workload, which can be combined with workload forecasting to yield even better indexes.

3. SPEEDING UP INDEX TUNING

Searching for the best configuration in a large space with many candidate indexes is inherently challenging. In fact, even a restricted version of the index selection problem is *NP-hard* [17] and/or even *hard to approximate* [12]. State-of-the-art index search algorithms, such as the *greedy* algorithm [14, 15, 30], therefore rely on heuristics to reduce the search space. However, scalability and efficiency remain challenging even in such reduced search spaces. We discuss how we can take a data-driven perspective by leveraging ML techniques to

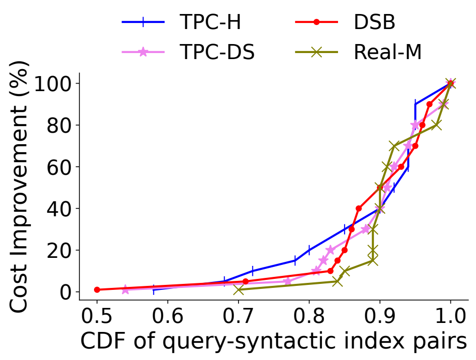


Figure 5: Cost improvement for different fraction of query and syntactically relevant index pairs.

speed up different components of index tuning.

3.1 Filtering Spurious Indexes

Index tuners perform syntactic analysis (e.g., using a set of rules) to select an initial set of indexes for each query, called *syntactically-relevant indexes*, for evaluation [15]. However, as showcased in Figure 5, we observe that 60% to 70% of such indexes are *spurious*—they actually do not result in significant improvement in query performance [59]. Thus, these spurious indexes can be filtered out and the optimizer calls made on these indexes can be avoided.

To prune such indexes early in the search process, we learn a *workload-agnostic* model that uses structure and statistics information in the input (query, index) pair to identify when the index may not lead to a significant improvement in cost [59]. We then use this model to remove a large number of spurious indexes. Our key insight is that we can probe the original physical plan of the query (i.e., the plan generated with existing indexes) to estimate the potential for improvement in the cost of the query due to a given index. For instance, if a join or sort operation is already efficient due to extensive filtering from earlier operations, adding an index that optimizes this operation is less beneficial. Similarly, if a filter column is not selective, we can easily prune an index that uses it as the leading key column. Furthermore, in many cases, we can compare the ordering of physical operators in the original plan with the structure of the index to identify spurious indexes. Altogether, we capture many such signals and train a regression model to automatically learn rules to predict spurious indexes.

Takeaway #3*: Many syntactically-relevant indexes do not lead to improvement in performance. ML models trained on top of domain-specific signals can filter such spurious indexes in orders of magnitude less time compared to making what-if (optimizer) calls.*

As shown in Figure 6, we find that index filtering models can be accurately learnt using (query, index) pairs

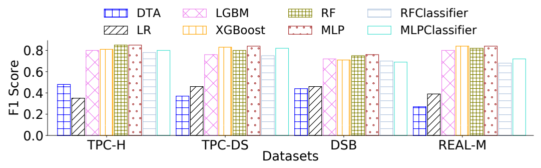


Figure 6: Learned Index Filter.

generated from 3 to 4 databases and workloads and can remove more than 70% spurious indexes with a low rate (typically less than 10%) of false negatives [59].

3.2 Search by Reinforcement Learning

Given the large number of possible index configurations during configuration enumeration for cloud-scale workloads, it is practically impossible to have one what-if optimizer call for *every* configuration and *every* query enumerated. This raises a trade-off between *exploration* (of new configurations) and *exploitation* (of promising configurations that are already known) when determining which configurations are worth what-if calls. We develop a new index search framework based on Monte Carlo tree search (MCTS) [72], a classic reinforcement learning (RL) technology [8, 61], to make better decisions on this exploration/exploitation trade-off. In particular, we adapt the classic greedy search algorithm, typically used during configuration search [14], to handle the trade-off in a data-driven manner as follows:

- **Exploitation:** We can expand configurations that *show promise*, e.g., ones that contain the best configuration found by the greedy algorithm so far as a subset;
- **Exploration:** We can consider configurations that have been overlooked but may have *potential* for improvement, e.g., ones that are not the *winner* configuration found by the greedy algorithm, but have similar costs and can be utilized by more queries.

From this viewpoint, the existing greedy search approach can be viewed as one extreme—it relies on *full* exploitation of what has been found with *no* exploration. Our RL-based approach, on the other hand, encourages more exploration, offering a principled way of tackling the above exploitation/exploration trade-off.

Takeaway #4*: RL-based techniques help navigate exploration and exploitation trade-offs more effectively on deciding which (query, configuration) to evaluate next.*

Figure 7 presents evaluation results on the TPC-DS benchmark and a customer workload (Real-M) with the maximum desired configuration size K set to 20. We compare the MCTS-based approach with both the *vanilla* greedy search algorithm and its variants (shown as *two-phase* greedy and *AutoAdmin* greedy in Figure 7) proposed in [15] and used in the Database Tuning Advisor (*DTA*) developed for Microsoft SQL Server [14], which represents the current state of the art [30]. As depicted in the figure, MCTS outperforms the greedy search algorithms consistently on both workloads w.r.t. the varying number of what-if calls.

***Open Challenge #4**: Integrating MCTS-based search into commercial index tuning tools such as DTA remains an open problem, considering additional requirements such as anytime tuning, incremental handling of input workloads, and supporting reproducibility (difficult due to randomness inherent to MCTS).*

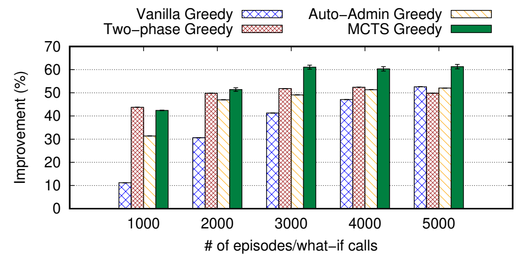
When the input workload is large and/or complex, we may want to run index tuning with a specific time-bound [14], or we may want to stop the tuning after some time without specifying a budget initially. Therefore, the search algorithm is desired to have the *anytime* property, i.e., it should progressively find better configurations over time. This also requires incremental handling of more queries as input to the search algorithm and maintaining and reasoning about the intermediate state to minimize redundant work. Furthermore, the final recommended indexes can vary due to randomness in MCTS/RL, which affects reproducibility. Handling these challenges in a commercial tuning tool like DTA requires non-trivial adaptations to the MCTS algorithm.

We note that there has been other recent work on ML-based configuration search [31, 33, 44, 45, 55], primarily targeting an online index tuning scenario. This line of work may be adaptable to offline index tuning but it shares the same challenges, as highlighted above, when it comes to integration with existing index tuners. Notably, the recent work by Kossmann et al. [31] proposed training an RL agent that can be used for offline index tuning, where test workloads are presumably similar to training workloads observed by the RL agent. Whether this approach can be further extended to tune completely unseen workload remains an open question.

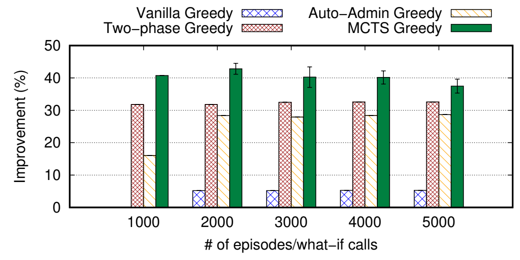
3.3 Reducing What-If Optimizer Calls

To achieve the best possible improvement in performance, the number of optimizer calls made during index configuration search can remain considerable despite pruning of spurious indexes and judicious enumeration of configurations. To further improve the efficiency, we find that a significant number of optimizer calls for costing (query, configuration) pairs can potentially be replaced by more efficient data-driven cost models.

Developing a general cost model that is independent of databases and workloads is hard due to the large varieties in the schema, query structures, and data distributions, despite the intensive efforts in the past decade (e.g., [4,24,35,40,42,57,60,70,71,73,79]). Our key observation to developing a lightweight cost model in the specific context of index tuning is that many queries in large workloads are *self-similar*, e.g., multiple instances of the same stored procedure or query template parameterized differently. Many indexes explored during tuning can also be similar (e.g., sharing the same prefix of key columns, or influencing the same set of operators in the plan), which leads to similar configurations and results in similar cost reductions. As a result, the number of unique cost values is often much smaller than the number of index configurations explored during tuning (e.g., on average only 6 unique costs over 81 configurations explored per query for the TPC-H workload).



(a) TPC-DS ($K = 20$)



(b) Real-M ($K = 20$)

Figure 7: Evaluation of MCTS configuration search.

To leverage these characteristics, we group similar queries with the same query template and then learn a cost model for each group [59]. For efficient in-situ learning during index configuration enumeration, we develop an *iterative* training procedure (with optimality guarantees) and select diverse training instances (e.g., queries with different selectivities, indexes affecting different operators in the query, etc.) that minimize the number of optimizer calls for training each cost model (e.g., less than 50 optimizer calls per model on average across workloads). We show that it is possible to use low-overhead ML models that are significantly more efficient than making what-if optimizer calls. A key characteristic of these models is that they are *agnostic* of the search algorithm (thus can be used by any algorithm), and do not require changes to the query optimizer.

***Takeaway #5**: ML-based cost models can be used as a generalized cache for similar (query, configuration) pairs, thereby avoiding many "similar" what-if calls.*

Figure 8 depicts the effectiveness of different ML algorithms when used to train per-template cost models, with tree-based models achieving Q-error as low as 1.2. Furthermore, we find that combing ML-based cost models with filtering models for pruning spurious indexes

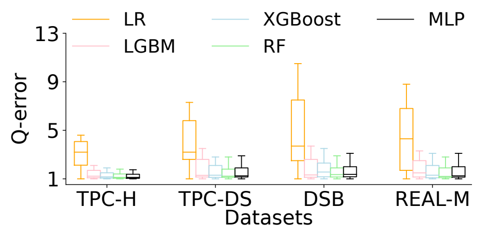


Figure 8: Learned Index Cost Model.

(see Section 3.1) helps scale index tuning to large workloads without sacrificing the quality of the recommended indexes. For instance, for a TPC-DS workload with over 900 queries, combining index filtering and costing models can give index recommendations with similar quality as DTA but in 3× less time.

Open Challenge #5*: Creating ML-based cost models across queries with different templates and across workloads can further improve the scalability by reducing training overhead (i.e., optimizer calls).*

The per-template cost models are less effective for workloads with many templates. There is a significant potential for reducing the number of optimizer calls as well as the number of models if we can generalize cost models across templates. There has been recent work on zero-shot cost models [25]; however, such techniques require a physical query plan (and thus an optimizer call) for featurization. Furthermore, in our current approach, we re-train during every tuning session from scratch due to limited mechanisms for meta-learning or fine-tuning learned models to capture workload and data drifts. This is another area where there are some intersections with online index tuning work [6,44,48].

4. PERFORMANCE REGRESSION

An important requirement of automated index tuning for production systems is creating or dropping indexes should not cause significant query performance regressions (QPR), where a query's execution cost increases dramatically after changing the indexes [21]. Such regressions are a major impediment for fully-automated and scalable index tuning [18]. When an index tuner searches for optimal configurations, it compares estimated improvements of query performance based on the optimizer's estimated costs. Due to well-known limitations in the optimizer's estimates, such as errors in cardinality estimation [27] or cost modeling [71], using the optimizer's estimates can result in significant cost estimation errors. The following trade-off is at the heart of why it is hard to achieve scalability and low rate of QPR in index tuning simultaneously:

Efficiency vs. Accuracy Trade-off*: Optimizer's estimated costs are much faster to compute, but they can be erroneous and result in low-quality recommendations and query performance regressions. On the other hand, actual query execution time is much more accurate but it can only be obtained with significantly higher overhead, affecting the scalability of index tuning.*

One idea to reduce query performance regression is to selectively use execution time during index tuning along with optimizer's estimated cost. Towards this end, Ding et al. [21] proposed a suite of ML techniques that learn over query execution telemetry collected from tens of databases to predict whether or not a new plan due to a selected index configuration has regressed with respect to another plan. Active learning techniques have been used to selectively collect query execution data for ML model training by deploying the same target database on non-production servers [36]. Furthermore, techniques for fixing QPR have also been proposed [21,22].

Takeaway #6*: Leveraging optimizer's estimated costs for index tuning, while* verifying *selected configurations at each step of configuration enumeration through machine learning models trained on query execution statistics, can reduce query performance regressions.*

Unfortunately, from the scalability perspective, the inference process in [21] is expensive since it requires query optimizer calls to obtain the physical plans of the queries. Indeed, the focus of [21] was not scalability, targeting a closed-loop continuous tuning scenario where index tuning time is perhaps trivial considering the workload execution time, especially if there are query performance regressions.

Open Challenge #6*: Detecting configurations that cause query performance regressions with both efficiency and wide coverage, while preserving the scalability of index tuning, remains a significant challenge for large-scale workloads.*

A promising direction, intersecting with challenges discussed in Section 3.3, is to learn pre-trained cost models that bridge the gap between optimizer cost models and the execution behaviour of queries. A challenge that needs to be addressed is that such pre-trained models may not be accurate without requiring plan-level details that need what-if optimizer calls. Toward this end, we can explore techniques similar to the ones used for filtering spurious indexes (Section 3.1) where the original physical plan is probed with properties provided by an index to reason about potential improvement in the cost, as showcased by the very recent work [56]. While learning a generalized model that can work across workloads is challenging (as discussed earlier), we can narrow down the problem by focusing only on indexing-specific improvements. If we can accurately learn such models, it opens up opportunities to eschew both optimizer calls as well as query executions during index tuning, thereby significantly improving the scalability.

5. CROSS-PLATFORM TUNING

The current database management landscape involves many SQL-like systems, with only a few supporting index tuning. While the systems differ in SQL dialects and functionalities (e.g., what-if API), the core ideas for index tuning can often be reused. This is more true for data-driven techniques discussed in this paper, where the ML models have limited dependency on the dialects or unique features of a particular system.

Open Challenge #7: *There are many database systems (where indexes help improve performance) that either have no or low-quality automated index tuning capabilities, forcing users to manually select indexes for their workloads. Adding an index tuner to a new or evolving database system requires substantial engineering overhead, despite that many core ideas in index tuning are cross-platform reusable.*

We hereby call for research efforts on developing a cross-platform index tuner that can work across multiple SQL-like systems, reusing core index tuning techniques (e.g., data-driven ML models as well as the search algorithms currently used in state-of-the-art index tuners). Similar efforts have been made in other areas such as query optimization [7,28]. A cross-platform index tuner needs to adapt to the heterogeneity of features varying across database engines, while reusing the common steps as much as possible. We abstract such a system in Figure 9 with the following main components:

- *Common Data Representation (IR)* consisting of a basic set of elements that need to be captured across systems, e.g., database, tables, columns, logical operators, physical operators, and sub-plans. A cross-language specification such as Subtrait [2] can potentially be leveraged for IR.
- *Common System Interaction APIs* consisting of a common set of APIs that can be used to interact with the database during index tuning. Examples of such APIs include ones for query optimization in the presence of one or more indexes, query execution, creation of hypothetical indexes similar to the what-if API, and building of statistics.
- *Adapters* providing the system-specific implementation of the common system interaction APIs that vary across systems.
- *Index Tuning Planner* enabling cross-platform tuning functionality. It considers system-specific features and user requirements, and outputs an index tuning plan (analogous to query execution plan in database systems). The index tuning task can be represented with a small set of *operators* (e.g., `enumerate`, `combine`, `evaluate`) that can be composed together and configured to perform index tuning. The index tuning plan can be an *acyclic graph* of these operators that is dynamically constructed and optimized by the planner based on system features and user requirements.

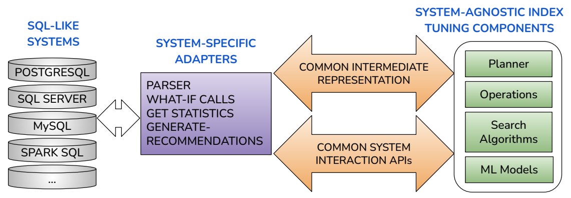


Figure 9: A Cross-Platform Design for Index Tuning.

Overall, a cross-platform index tuner consisting of the components as envisioned above has the potential to democratize index tuning to many more systems than those that are currently supported. In addition, such an index tuner will allow (a) borrowing of the best concepts (implemented as operators) from different index-tuning algorithms, (b) independent improvement and maintenance of the functionality of operators, and (c) extensibility by incorporating new techniques (implemented as new operators) in the future without rewriting the algorithms or changing unrelated operations.

6. CONCLUSION

In this paper, we have highlighted the challenges inherent to automated index tuning, which are further exacerbated within modern cloud environments, and we have discussed recent efforts and opportunities in leveraging ML-powered techniques to address them. We presented an end-to-end analysis of the index tuning workflow, with a focus on the core components such as workload selection and configuration search. We described the issue of query performance regression (QPR) and discussed ML techniques for addressing QPR without affecting index tuning scalability. We also sketched the design of a cross-platform index tuner that extends the current index-tuning software stack to support multiple SQL-like systems. We believe this paper will help create awareness of recent progress and highlight open challenges for future research in index tuning.

Acknowledgement. We thank Surajit Chaudhuri and Vivek Narasayya for their valuable feedback on this work.

7. REFERENCES

[1] Azure sql database. `https://azure.microsoft.com/en-us/products/azure-sql/database/`.
[2] Substrait: Cross-language serialization. `https://substrait.io/`, 2022.
[3] J. Aguilar-Saborit and R. Ramakrishnan. POLARIS: the distributed SQL engine in azure synapse. *Proc. VLDB Endow.*, 13(12):3204–3216, 2020.

[4] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *ICDE*, pages 390–401, 2012.

[5] D. V. Aken, D. Yang, S. Brillard, A. Fiorino, B. Zhang, C. Billian, and A. Pavlo. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proc. VLDB Endow.*, 14(7):1241–1253, 2021.

[6] 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 *DEXA*, pages 253–268, 2015.

[7] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*, pages 221–230, 2018.

[8] C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. P. Liebana, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intell. AI Games*, 4(1):1–43, 2012.

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

[10] N. Bruno and S. Chaudhuri. To tune or not to tune? A lightweight physical design alerter. In *VLDB*, pages 499–510. ACM, 2006.

[11] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, pages 826–835, 2007.

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

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

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

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

[16] S. Chaudhuri and V. R. Narasayya. Autoadmin 'what-if' index analysis utility. In *SIGMOD*, pages 367–378, 1998.

[17] D. Comer. The difficulty of optimum index selection. *ACM Trans. Database Syst.*, 3(4):440–445, 1978.

[18] 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.

[19] D. Dash, N. Polyzotis, and A. Ailamaki. Cophy: A scalable, portable, and interactive index advisor for large workloads. *Proc. VLDB Endow.*, 4(6):362–372, 2011.

[20] S. Deep, A. Gruenheid, P. Koutris, J. F. Naughton, and S. Viglas. Comprehensive and efficient workload compression. *Proc. VLDB Endow.*, 14(3):418–430, 2020.

[21] 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.

[22] B. Ding, S. Das, W. Wu, S. Chaudhuri, and V. R. Narasayya. Plan stitch: Harnessing the best of many plans. *Proc. VLDB Endow.*, 11(10):1123–1136, 2018.

[23] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Trans. Database Syst.*, 13(1), 1988.

[24] A. Ganapathi, H. A. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, and D. A. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, 2009.

[25] B. Hilprecht and C. Binnig. Zero-shot cost models for out-of-the-box learned cost prediction. *Proc. VLDB Endow.*, 15(11):2361–2374, 2022.

[26] B. Hilprecht, C. Binnig, and U. Röhm. Learning a partitioning advisor for cloud databases. In *SIGMOD*, pages 143–157. ACM, 2020.

[27] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, pages 268–277, 1991.

[28] A. Jindal, K. V. Emani, M. Daum, O. Poppe, B. Haynes, A. Pavlenko, A. Gupta, K. Ramachandra, C. Curino, A. Müller, W. Wu, and H. Patel. Magpie: Python at speed and scale using cloud backends. In *CIDR*, 2021.

[29] A. Kane. The automatic indexer for postgres. `https://github.com/ankane/dexter`, June 2017.

[30] 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. *Proc. VLDB Endow.*, 13(11):2382–2395, 2020.

[31] J. Kossmann, A. Kastius, and R. Schlosser. SWIRL: selection of workload-aware indexes using reinforcement learning. In *EDBT*, pages 2:155–2:168, 2022.

[32] M. Kurmanji and P. Triantafillou. Detect, distill and update: Learned DB systems facing out of distribution data. *Proc. ACM Manag. Data*, 1(1):33:1–33:27, 2023.

[33] H. Lan, Z. Bao, and Y. Peng. An index advisor using deep reinforcement learning. In *CIKM*, pages 2105–2108, 2020.

[34] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.*, 12(12):2118–2130, 2019.

[35] J. Li, A. C. König, V. R. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for SQL queries using statistical techniques. *Proc. VLDB Endow.*, 5(11):1555–1566, 2012.

[36] L. Ma, B. Ding, S. Das, and A. Swaminathan. Active learning for ml enhanced database systems. In *SIGMOD*, pages 175–191, 2020.

[37] 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 *Proceedings of the 2018 International Conference on Management of Data*, pages 631–645, 2018.

[38] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making learned query optimization practical. *SIGMOD Rec.*, 51(1):6–13, 2022.

[39] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[40] R. C. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow.*, 12(11):1733–1746, 2019.

[41] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. Cliffguard: A principled framework for finding robust database designs. In *SIGMOD*, pages 1167–1182. ACM, 2015.

[42] D. Paul, J. Cao, F. Li, and V. Srikumar. Database workload characterization with query plan encoders. *Proc. VLDB Endow.*, 15(4):923–935, 2021.

[43] R. M. Perera, B. Oetomo, B. I. Rubinstein, and R. Borovica-Gajic. No dba? no regret! multi-armed bandits for index tuning of analytical and htap workloads with provable guarantees. *IEEE Transactions on Knowledge and Data Engineering*, 2023.

[44] R. M. Perera, B. Oetomo, B. I. P. 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.

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

[46] R. Potharaju, T. Kim, E. Song, W. Wu, L. Novik, A. Dave, P. Pirzadeh, A. Fogarty, G. Dhody, J. Li, V. Acharya, S. Ramanujam, N. Bruno, C. A. Galindo-Legaria, V. R. Narasayya, S. Chaudhuri, A. Nori, T. Talius, and R. Ramakrishnan. Hyperspace: The indexing subsystem of azure synapse. *Proc. VLDB Endow.*, 14(12):3043–3055, 2021.

[47] R. Potharaju, T. Kim, W. Wu, V. Acharya, S. Suh, A. Fogarty, A. Dave, S. Ramanujam, T. Talius, L. Novik, and R. Ramakrishnan. Helios: Hyperscale indexing for the cloud & edge. *Proc. VLDB Endow.*, 13(12):3231–3244, 2020.

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

[49] K. Sattler, I. Geist, and E. Schallehn. QUIET: continuous query-driven index tuning. In *VLDB*, pages 1129–1132. Morgan Kaufmann, 2003.

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

[51] R. Schlosser, M. Weisgut, L. Hübscher, and O. Nordemann. Robust index selection for stochastic dynamic workloads. *SN Comput. Sci.*, 4(1):59, 2023.

[52] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: continuous on-line tuning. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *SIGMOD*, pages 793–795. ACM, 2006.

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

[54] K. Schnaitter and N. Polyzotis. Semi-automatic index tuning: Keeping dbas in the loop. *Proc. VLDB Endow.*, 5(5):478–489, 2012.

[55] A. Sharma, F. M. Schuhknecht, and J. Dittrich. The case for automatic database administration using deep reinforcement learning. *CoRR*, abs/1801.05643, 2018.

[56] J. Shi, G. Cong, and X. Li. Learned index benefits: Machine learning based index performance estimation. *Proc. VLDB Endow.*, 15(13):3950–3962, 2022.

[57] T. Siddiqui, A. Jindal, S. Qiao, H. Patel, and W. Le. Cost models for big data query processing: Learning, retrofitting, and our findings. In *SIGMOD*, pages 99–113. ACM, 2020.

[58] 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.

[59] T. Siddiqui, W. Wu, V. R. Narasayya, and S. Chaudhuri. DISTILL: low-overhead data-driven techniques for filtering and costing indexes for scalable index tuning. *Proc. VLDB Endow.*, 15(10):2019–2031, 2022.

[60] J. Sun and G. Li. An end-to-end learning-based cost estimator. *Proc. VLDB Endow.*, 13(3):307–319, 2019.

[61] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[62] J. Tan, T. Zhang, F. Li, J. Chen, Q. Zheng, P. Zhang, H. Qiao, Y. Shi, W. Cao, and R. Zhang. ibtune: Individualized buffer tuning for large-scale cloud databases. *Proc. VLDB Endow.*, 12(10):1221–1234, 2019.

[63] I. Trummer. DB-BERT: A database tuning tool that "reads the manual". In *SIGMOD*, pages 190–203, 2022.

[64] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In *SIGMOD*, pages 1153–1170, 2019.

[65] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, pages 101–110, 2000.

[66] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017.

[67] F. Ventura, Z. Kaoudi, J. Quiané-Ruiz, and V. Markl. Expand your training limits! generating training data for ml-based data management. In *SIGMOD*, pages 1865–1878. ACM, 2021.

[68] J. Wang, I. Trummer, and D. Basu. UDO: universal database optimization using reinforcement learning. *Proc. VLDB Endow.*, 14(13):3402–3414, 2021.

[69] K. Whang. Index selection in relational databases. In *Foundations of Data Organization*, pages 487–500, 1985.

[70] W. Wu, Y. Chi, H. Hacigümüs, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *Proc. VLDB Endow.*, 6(10):925–936, 2013.

[71] 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.

[72] 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.

[73] W. Wu, X. Wu, H. Hacigümüs, and J. F. Naughton. Uncertainty aware query execution time prediction. *Proc. VLDB Endow.*, 7(14):1857–1868, 2014.

[74] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P. Larson, D. Kossmann, and R. Acharya. Qd-tree: Learning data layouts for big data analytics. In *SIGMOD*, pages 193–208. ACM, 2020.

[75] X. Yu, C. Chai, G. Li, and J. Liu. Cost-based or learning-based? A hybrid query optimizer for query plan selection. *Proc. VLDB Endow.*, 15(13):3924–3936, 2022.

[76] B. Zhang, D. V. Aken, J. Wang, T. Dai, S. Jiang, J. Lao, S. Sheng, A. Pavlo, and G. J. Gordon. A demonstration of the ottertune automatic database management system tuning service. *Proc. VLDB Endow.*, 11(12):1910–1913, 2018.

[77] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*, pages 415–432. ACM, 2019.

[78] W. Zhang, M. Interlandi, P. Mineiro, S. Qiao, N. Ghazanfari, K. Lie, M. T. Friedman, R. Hosn, H. Patel, and A. Jindal. Deploying a steered query optimizer in production at microsoft. In Z. G. Ives, A. Bonifati, and A. E. Abbadi, editors, *SIGMOD*, pages 2299–2311, 2022.

[79] Y. Zhao, G. Cong, J. Shi, and C. Miao. Queryformer: A tree transformer model for query plan representation. *Proc. VLDB Endow.*, 15(8):1658–1670, 2022.

[80] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *SoCC*, pages 338–350, 2017.