UTune: Towards Uncertainty-Aware Online Index Tuning

Chenning Wu†, Sifan Chen†, Wentao Wu§, Yinan Jing*†, Zhenying He†, Kai Zhang†, X. Sean Wang†
†Fudan University, Shanghai, China
§Microsoft Research, Washington, USA
{wucn23,sfchen23}@m.fudan.edu.cn, wentao.wu@microsoft.com, {jingyn,zhenying,zhangk,xywangCS}@fudan.edu.cn

***Abstract*—There have been a flurry of recent proposals on learned benefit estimators for index tuning. Although these learned estimators show promising improvement over what-if query optimizer calls in terms of the accuracy of estimated index benefit, they face significant limitations when applied to online index tuning, an arguably more common and more challenging scenario in real-world applications. There are two major challenges for learned index benefit estimators in online tuning: (1) limited amount of query execution feedback that can be used to train the models, and (2) constant coming of new unseen queries due to workload drifts. The combination of the two hinders the generalization capability of existing learned index benefit estimators. To overcome these challenges, we present `UTune`, an uncertainty-aware online index tuning framework that employs operator-level learned models with improved generalization over unseen queries. At the core of `UTune` is an *uncertainty quantification* mechanism that characterizes the inherent uncertainty of the operator-level learned models given limited online execution feedback. We further integrate uncertainty information into index selection and configuration enumeration, the key component of any index tuner, by developing a new variant of the classic epsilon-greedy search strategy with uncertainty-weighted index benefits. Experimental evaluation shows that `UTune` not only significantly improves the workload execution time compared to state-of-the-art online index tuners but also reduces the index exploration overhead, resulting in faster convergence when the workload is relatively stable.**

***Index Terms*—Automated Indexing, Query Optimization, Uncertainty Quantification, Physical Database Design**

I. Introduction

Building appropriate indexes inside a database system is crucial for accelerating query performance. The problem of index tuning/recommendation therefore has been studied intensively and extensively in the past decades (see [1, 2] for recent surveys). A classic scenario is *offline* index tuning, where a fixed workload of queries is given and the goal is to recommend a set (a.k.a. *configuration*) of indexes that can minimize the workload execution cost with respect to certain constraints such as the maximum number of indexes or the maximum amount of storage that can be taken by the indexes [3]. To evaluate the quality of candidate indexes, a common approach is to use the "*what-if*" API provided by the query optimizer, which allows for cost estimation of a query *without* materializing the candidate indexes [4]. The downside of what-if cost estimation is its potential inaccuracy, as it is built on top of query optimizer's cost models that are known to be error-prone [5, 6].

To improve the accuracy of cost estimation in the context of index tuning, recent work has introduced learned index benefit estimators [6–8], which leverage historical execution data to predict the potential benefit of various index configurations. These learned models, typically trained offline using large amounts of query execution telemetry data, have shown promise in improving the quality of recommended indexes. However, they face significant limitations when it comes to *online* index tuning, which is arguably a more common and more challenging scenario in real-world applications [9, 10]. In online scenarios, workload drifts can occur from time to time with incoming new queries that may not be observed in the past workload execution history. Without appropriate and timely adjustments, pre-trained index benefit estimators can hardly generalize well for the unseen queries.

We propose `UTune`, an online index tuning framework equipped with a learned index benefit estimator and an index selector specifically customized for online scenarios. Instead of having a holistic query-level index benefit estimator, as was proposed by recent work [7], in `UTune` we maintain a learned cost model for each individual type of relational operator. Although operator-level learned cost models have been studied in the literature [11–13], their application to online index tuning has not yet been explored. Our motivation of using operator-level instead of query-level model is driven by the observation that the amount of query execution time feedback collected in online index tuning is often limited. Compared to query-level model, with limited training data, the operator-level model often exhibits better generalization capability for unseen queries [11, 12], a fundamental and inevitable challenge faced by online index tuners/advisors.

At the core of `UTune` is an *uncertainty quantification* mechanism that measures the reliability of the operator-level cost models. Considering model uncertainty plays an important role in the effectiveness of `UTune`, as it both improves the cost estimates themselves and reduces the exploration overhead of RL-based online index selection, a paradigm that has been adopted by a stream of recent work on online index tuning [14–18], which `UTune` also follows (see Figure 1). Below, we provide a brief overview of this uncertainty-driven design underlying `UTune`.

* Corresponding author: Yinan Jing (jingyn@fudan.edu.cn).

This work is supported by the National Key R&D Program of China (2024YFC3307800), NSFC (U24A20232, 62072113, and 62272106) and Xiaohongshu–Fudan Collaborative Project (PHT10120241108010).

Uncertainty-aware Cost Estimation Correction: Existing work on index benefit estimation often assumes the availability of abundant historical execution data [7], which is often impractical in real-world online settings where proactive data collection can be prohibitively expensive [6, 13], and future workloads cannot be observed in advance. As a result, in reality, one may have to start online index tuning from scratch. This implies a *cold-start* phase during which the ML-based index benefit estimator is not reliable due to the lack of execution feedback. This issue can be further compounded by frequent workload drifts, which can damage the fidelity of past execution data, and thus the usability of the trained ML models. There has been recent work on modeling index tuning as contextual bandits [19–21] that attempted to address this "cold-start" challenge by incorporating historical knowledge and transfer learning to better initialize the RL models. However, this line of work remains dependent on the simplified assumptions of a linear function between index features and their benefits, which can fail to capture more complex index benefit relationships present in real-world workloads.

Unlike prior methods, UTune employs an uncertainty-aware cost correction mechanism to mitigate the "cold-start" challenge. Specifically, we learn the "cost adjustment multipliers" (CAM) for each relational operator, which is the correction factor applied to the operator's what-if cost estimate. UTune asks for the required *degree of adjustment/correction* rather than the absolute execution time prediction of an operator. In this way, we address the "cold-start" phase by bootstrapping from initial what-if estimates and smoothly transitioning to learned CAM-based adjustments as execution feedback accumulates. To ensure the *reliability* of learned CAMs, we develops a metric to quantify the *uncertainty* of CAM models during this transition, as those with minimal execution feedback are inherently untrustworthy. UTune only updates the operator costs when corresponding models have sufficiently *low uncertainty* (i.e., high confidence).

Uncertainty-aware Index Selection: Most state-of-the-art online index tuners adopt RL-based technologies that essentially make trade-offs between *exploration* and *exploitation* [14–18]. This "trial-and-error" behavior often incurs significant computational overhead by creating and dropping indexes only for exploration purposes. The uncertainty metric developed for measuring the reliability of an operator-level CAM model offers new opportunities to reduce the exploration overhead of existing RL-based online index tuners. Specifically, we propose a new *index value function* that considers both the estimated index benefit and its uncertainty. Given that RL-based online index tuners aim for an optimal policy that decides the next index to be selected for exploration, this new index value function can be easily integrated to revise existing policies. In particular, in UTune we develop a new variant of the classic ϵ-greedy policy [22] that explores candidate indexes with probabilities proportional to their new uncertainty-weighted benefits. Factoring model uncertainty into index selection reduces the chance of exploring indexes for which models already have high confidence (i.e., low uncertainty),

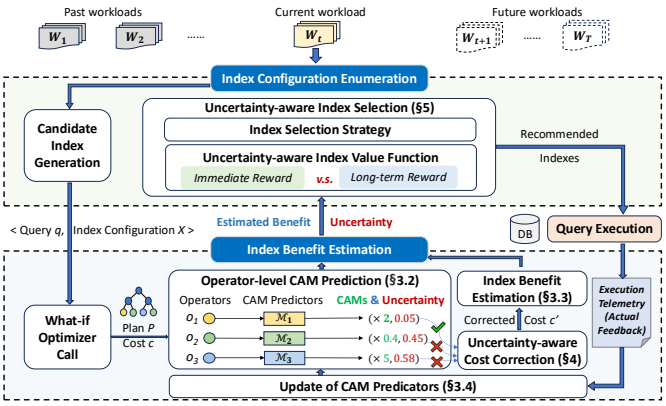


Fig. 1: Overview of UTune.

saving the computational overhead of creating and dropping such indexes that would bring little extra information beyond what the models have already learned. This is in the same spirit of *active learning*.

To summarize, this paper makes the following contributions:

- We propose UTune, an uncertainty-aware online index tuning framework that employs operator-level cost models for more effective utilization of limited query execution feedback and better generalization capability over unseen queries observed during workload drifts (Section II).
- We propose measuring the uncertainty of operator-level cost models and incorporating model uncertainty into both cost estimation (Section IV) and index selection (Section V), which not only improves the reliability of the estimated costs but also reduces the exploration overhead of RL-based online index selection.
- We evaluate UTune on top of three benchmarks **TPC-H**, **TPC-DS**, and **JOB**, with a variety of static and dynamic workloads, and our evaluation results show that UTune can significantly outperform existing state-of-the-art online index tuners in terms of the overall workload execution time improvement (Section VI).

From a broader viewpoint, uncertainty is a ubiquitous concept that can be extended to other components of an index tuner beyond the scope that has been studied in this paper. Future work includes developing an uncertainty-aware configuration enumeration method itself to replace our current reliance on the super-arm selection method from *DBA bandits* [14]. Similarly, uncertainty in ML-based workload forecasting [23] could be leveraged to refine RL-based index selection strategies. Beyond UTune's operator-level cost models, other learned index benefit estimators may also be improved by incorporating model uncertainty. In this regard, we view this work as just the start of a new direction for fertile ground of future work.

II. An Overview of UTune

Online index tuning aims at minimizing the execution time of incoming workload queries by deploying a set of indexes subject to certain constraints, such as the maximum number of indexes allowed or the maximum storage space that can be

TABLE I: Summary of notations

Symbol	Description	Section
X_t	Index configuration at time t	2.1
W_t	Mini-workload at time t	2.1
K	The maximum number of indexes allowed	2.1
$C_{\text{exe}}(W_t, X_t)$	Execution time of W_t under X_t	2.1
$\mathcal{M}$	Operator-level CAM predictor	3.2
$c(q, X)$	Estimated cost of the query q with the index configuration X	3.3
$b_c(X, q)$	Estimated benefit of the index configuration X for the query q	3.3
$t(q, X)$	Actual execution time of the query q with the index configuration X	3.4
$b_t(X, q)$	Actual benefit of the index configuration X for the query q	3.4
$U(o, \mathcal{M})$	Uncertainty of $\mathcal{M}$ for the operator o	4.1
α	Combination weight of data uncertainty and model uncertainty	4.1
ρ	Uncertainty threshold	4.2
$EB(x, W)$	Estimated benefit of the index x for the mini-workload W	5.1
$EV(x, W, \mathcal{M})$	Exploratory value of the index x for the mini-workload W w.r.t. $\mathcal{M}$	5.1
$V(x, W)$	Total value of the index x for the mini-workload W	5.1
λ	Exploration weight in index selection	5.1
γ	Decay factor for index exploration	5.2

taken by the deployed indexes. We start by giving a formal problem definition. We then present an overview of `UTune`.

A. Problem Formulation

Following existing work [14, 24], we model the input workload as a *sequence* of *mini-workloads* $\mathcal{W} = (W_1, W_2, ..., W_T)$ up to some time T, where each mini-workload W_t represents a set of queries and their corresponding frequencies for $1 \leq t \leq T$. We define $\mathcal{X}$ as the set of all possible indexes that could be created. An index *configuration* X is a subset of $\mathcal{X}$. Table I summarizes the notation.

Our goal is to recommend an index configuration X_t for each W_t to minimize the execution time $C_{\text{exe}}(W_t, X_t)$. The recommended X_t is subject to the following constraints: (1) $|X_t| \leq K$, where K is the maximum number of indexes allowed; and (2) X_t is chosen based on only observing the history, i.e., the *execution telemetry* of the mini-workloads $\{W_i\}_{i=1}^{t-1}$ w.r.t. the corresponding index recommendations $\{X_i\}_{i=1}^{t-1}$, and the *characteristics* of the present mini-workload W_t. We can now define the optimization problem associated with online index tuning as to select X_1, ..., X_T to minimize

$$\sum_{t=1}^{T} C_{\text{exe}}(W_t, X_t), \quad s.t.\ |X_t| \leq K \ \ \forall t \in \{1, 2, ..., T\}.$$

B. Workflow of `UTune`

As shown in Figure 1, `UTune` contains two main components: (1) index benefit estimation and (2) index configuration enumeration. Below is an overview of the two components.

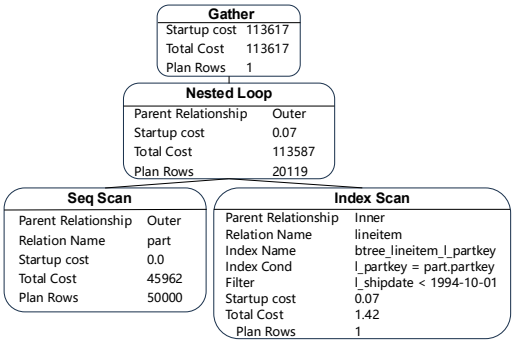
Index Benefit Estimation: `UTune` employs operator-level cost models for better generalization over unseen queries. For a given pair of query and index configuration, we first make a what-if call to obtain the corresponding query plan and its estimated cost. We then traverse the query plan to refine the estimated costs of individual operators, based on the corresponding operator-level learned cost models. Specifically, each operator-level learned cost model will output a "cost adjustment multiplier" (CAM), and we multiply the CAM with the what-if cost of the operator as its *corrected cost*. Given that the operator-level learned cost models are often trained with limited amounts of query execution feedback, we develop an uncertainty quantification mechanism to measure the model uncertainty. We will only correct the what-if cost of an operator if its corresponding model has low uncertainty. We present the details of the operator-level CAM predictors and the uncertainty-aware cost correction mechanism in Sections III and IV, respectively.

Index Configuration Enumeration: `UTune` follows a typical online index tuning workflow that involves candidate index generation, exploration, and selection. For the current mini-workload W_t, `UTune` first generates candidate indexes based on well-known technologies to find *indexable columns* [3], e.g., columns that appear in filter or join predicates. It then selects a subset (i.e., a configuration) from the candidate indexes that aims to minimize the execution time of W_t, subject to the given constraint K on the maximum number of indexes allowed. Unlike existing RL-based online index tuners that only rely on estimated index benefit, `UTune` employs a novel variant of the ϵ-greedy index exploration algorithm that considers both the estimated index benefit *and* model uncertainty when evaluating the value of a candidate index. This improves the effectiveness of exploration by focusing more on indexes that could further reduce model uncertainty, therefore resulting in faster convergence and lower overhead of index selection during the stable period of W_t. `UTune` selects the top-valued indexes as the final configuration, deploys them in the database, and collects query execution telemetry data to update the operator-level CAM predictors.

III. Index Benefit Estimation

Recent work on learned index benefit estimators [6–8] has been done to address the inaccuracy of what-if cost estimates. However, such learned index benefit estimators require a considerable amount of training data that is typically unavailable in online index tuning. Moreover, they were designed for offline index tuning where workload is presumed to be fixed and they do not offer agile model update mechanisms to deal with workload drifts.

To address the above "data starvation" and workload drift challenges in online index tuning, unlike existing learned index benefit estimators that take the entire query plan as input (referred to as query-level models in this paper), we propose having one learned model for each different type of (relational) operator. Although the idea of learning operator-level cost models has been studied in the literature [11–13], it has not yet been applied to online index tuning to the best of our knowledge. Moreover, the specifics of our learned operator-level cost modeling techniques are different from existing work, as we will detail in this section.



(a) An example query plan

Info		Encoding
Node type: index scan		[0,0,1,0,0]
Indexed columns: lineitem(l_partkey)		[0,0,0,0,1,0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0]
Index condition: l_partkey = part.partkey	**left value**: l_partkey	[0,0,0,0,1,0,0,0]
	Right value: part.partkey	[0,1,0,0,0,0,0,0]
	Operator: =	[1,0,0,0,0]
Filter: l_shipdate < 1994-10-01	**Left value**: l_partkey	[0,0,0,0,1,0,0,0]
	Right value: 1994-10-01	Semantic: [0.3,0.68,0.15] Statistical: [0.75]
	Operator: <	[0,0,1,0,0]

(b) Encoding of the `Index Scan` operator

Fig. 2: Feature representation of relational operators for operator-level cost models in `UTune`

Operator-level models are more sample-efficient than query-level models. Query-level models map query plan features to execution time and suffer from the credit assignment problem [25], which makes it hard for the model to discern which part of the plan contributes to the cost/latency estimation errors. Therefore, query-level models require a considerable number of training samples to converge. In contrast, operator-level models decompose estimation errors and execution feedback at the operator level. This allows for fine-grained error correction, enabling operator-level models to learn efficiently from sparse online samples (Section III-B). Moreover, operator-level models generalize better to unseen queries [5, 11, 13], whereas query-level models struggle with workload drifts because they perceive new query templates as out-of-distribution samples.

A. Featurization of Relational Operators

The raw feature representation of a relational operator consists of two parts: *structural information* (SI) and *predicate information* (PI). Structural information includes the operator type and the table columns that the operator processes. For example, the structural information for the `Index Scan` in Figure 2a is represented as

$$\{\textbf{node\_type} : \text{index\_scan}, \textbf{indexed\_column} : \text{l\_partkey}\}.$$

Structural information: We use *one-hot encoding* to encode the node type. For indexed columns, we apply one-hot encoding to each column and concatenate the resulting vectors in sequence to preserve their *positional information*. To ensure a fixed input length, we set a maximum index width (which is 3 in our experimental evaluation) and zero-pad shorter indexes.

Predicate information: This includes both filter and join conditions, represented as $\langle \text{column}, \text{operator}, \text{value/column} \rangle$ triplets. *'Table columns'* and *'comparison operators'* (e.g., $=$, $>$, $<$, $>=$, $<=$, $!=$,`LIKE`, and `SIMILAR TO`) are one-hot encoded. *'Values'* are min-max normalized for numbers, while strings are encoded using both semantic and statistical information. We encode its semantic information with a `word2vec` [26] model pre-trained on strings appearing in the initial workload W_0. For unseen predicates with string value s during workload drifts, we search the vocabulary for the entry v^* that shares the longest prefix with s, and then use the embedding `word2vec`(v^*) to represent s, thereby allowing the model to generalize to previously unseen predicate values based on their lexical similarity to known strings. Statistical information is captured via normalized cardinality rank. Figure 2b illustrates the encoding of the operator `Index Scan` in the query plan of Figure 2a.

B. Cost Adjustment Multipliers

Instead of directly predicting operator execution cost/time [11–13], `UTune` uses operator-level models to predict a cost adjustment multiplier (CAM) that corrects the query optimizer's original cost estimate of the operator. This is motivated by the observation that the query optimizer often exhibits similar degrees of cost estimation errors for similar operators (e.g., `Index Scans` on similar indexes with similar predicates) despite their varying true costs. For instance, an `Index Scan` as the inner child of a `Nested-Loop Join` may be much costlier than standalone execution due to repeated loops, yet the optimizer may underestimates the costs of both by a similar factor. As a result, learning CAMs that captures these systematic estimation improves model's generalization across queries.

We frame the operator-level CAM prediction problem as a multi-class classification task. Specifically, given a relational operator encoding o based on the featurization in Section III-A, the CAM predictor $\mathcal{M}$ selects one CAM from a given set Ω. Formally, $\mathcal{M}$ represents a function $\mathcal{M}(o) = \omega$, for some $\omega \in \Omega$. The distribution of $\omega \in \Omega$ needs to cover both underestimation and overestimation errors up to certain degrees. As a concrete implementation, we consider the following buckets of estimation errors: (1) $\Omega_1 = \{0.01, 0.02, ..., 0.09\}$, (2) $\Omega_2 = \{0.1, 0.2, ..., 0.9\}$, (3) $\Omega_3 = \{1, 2, ..., 9\}$ and (4) $\Omega_4 = \{10, 20, ..., 100\}$. We set $\Omega = \cup_{i=1}^{4} \Omega_i$, and we use a simple multi-layer perceptron (MLP) classifier as $\mathcal{M}$. This design of Ω follows a *non-uniform quantization* strategy. We use buckets with denser incremental steps for small estimation errors (Ω_2 for underestimation and Ω_3 for overestimation) and buckets with wider incremental steps for large estimation errors (Ω_1 for underestimation and Ω_4 for overestimation). This scheme provides both precise adjustments for relatively accurate query optimizer estimates (e.g., for **TPC-H**) and drastic correction of gross estimation errors (e.g., for **JOB**).

Remark: An alternative approach is to model CAM prediction as a *regression* problem instead of a multi-class classification problem. The core reason for the classifier's superiority lies in the simplification of the learning objective. Instead of strictly minimizing error for every individual data point, the classifier focuses solely on learning the boundaries of the decision. This formulation aligns better with index recommendation, where identifying the relative magnitude of index benefit is sufficient for optimal decision-making [13]. By relaxing the requirement from precise numerical estimation to coarse-grained quantization, the classification-based CAM model achieves better learning efficiency and is more robust to noise in online index tuning. We have further implemented a regression-based CAM predictor and compared it with our classification-based approach (Section VI-C).

C. Index Benefit Estimation

To estimate the index benefit, we aggregate the corrected costs of all individual operators in the query plan. For a query q with a proposed (hypothetical) index configuration X, the optimizer's what-if call yields a plan $P(q, X)$ with cost $c(q, X)$. Let $o_1, \ldots, o_p$ be the different types of relational operators in $P(q, X)$, with $\mathcal{M}_j$ as the CAM predictor for o_j ($1 \leq j \leq p$). After applying operator-level cost correction for each o_j based on its corresponding $\mathcal{M}_j$ (details are presented in Section IV), we get the corrected plan cost $c'(q, X)$. The estimated index benefit is then

$$b_c(X, q) = \frac{c(q, \emptyset) - c'(q, X)}{c(q, \emptyset)} = 1 - \frac{c'(q, X)}{c(q, \emptyset)},$$

where $c(q, \emptyset)$ is the estimated cost of q without indexes.

D. Update of CAM Predictors

Unlike offline index benefit estimators that only need one-time training, the operator-level CAM predictors that we proposed for online index tuning require continuous updates as more query execution feedback is available. Although we can obtain operator-level execution telemetry (e.g., by utilizing the `EXPLAIN ANALYZE` command of PostgreSQL), it is difficult to use this execution feedback directly since the what-if cost and execution time are not based on the same *unit of measurement*. One solution could be to perform a calibration of the query optimizer's cost modeling system [5], which would incur non-trivial computation overhead. We instead propose a low-overhead approach to effectively use the query-level execution feedback without extra calibration.

We choose to focus on *leaf* (table access) operators on *indexed* columns, such as `Sequential Scan`, `Index Scan`, `Index Only Scan`, or `Bitmap Index scan` in PostgreSQL, due to their dominant impact on query performance in the context of index tuning [13]. For each X-related operator in the query plan q, we independently correct its cost estimation to make the estimated index benefit $b_c(X, q)$ close to the *actual index benefit*, which is naturally defined as

$$b_t(X, q) = \frac{t(q, \emptyset) - t(q, X)}{t(q, \emptyset)} = 1 - \frac{t(q, X)}{t(q, \emptyset)}.$$

Here, $t(q, X)$ is the actual execution time of q with X.

Algorithm 1 illustrates this execution feedback utilization in detail. It aims to identify an appropriate CAM that aligns estimated and actual index benefits after cost correction (line 7). This is realized through a bottom-up propagation mechanism `UpdateCost` of cost corrections (line 5). The details of `UpdateCost` can be found in Algorithm 3.

Algorithm 1: Execution Telemetry Utilization

Input: P, query plan with execution time feedback; X, index configuration; Ω, candidate CAMs.
Output: $L = [(o_1, \omega_1), (o_2, \omega_2), ..., ((o_{|\mathcal{O}|}, \omega_{|\mathcal{O}|})]$, a list of operator-CAM pairs for model updating.

1 $\mathcal{O} \leftarrow X$-related leaves on P, $L \leftarrow \emptyset$;
2 **foreach** $o \in \mathcal{O}$ **do**
3 $\delta \leftarrow |b_t(X, q) - b_c(X, q)|$, $\omega \leftarrow 1$;
4 **foreach** $\omega' \in \Omega$ **do**
5 `UpdateCost`(P, o, ω');
6 $b_{c'}(X, q) \leftarrow 1 - \frac{c'(q,X)}{c(q,\emptyset)}$;
7 **if** $|b_t(X, q) - b_{c'}(X, q)| < \delta$ **then**
8 $\omega \leftarrow \omega'$;
9 $\delta \leftarrow |b_t(X, q) - b_{c'}(X, q)|$;
10 $L \leftarrow L \cup (o, \omega)$
11 **return** L;

Discussion: Algorithm 1 does not explicitly model *cross-operator* index interactions. Although this could have been done by training a query-level CAM model that captures cross-operator effects, it requires much more training data to cover the interaction space, which would severely delay convergence in "data-starved" online tuning scenarios. Instead, `UTune` captures index interactions *implicitly* based on its *uncertainty-aware* index benefit estimation and index selection strategy. As detailed in the case study from the extended full version of this paper [27], when an index interacts with others (i.e., the index benefit on the same query diverges significantly depending on the presence or absence of another index), it manifests performance volatility in the operator-level feedback, which increases the model-level *aleatoric uncertainty* (see Section IV-A). The uncertainty-aware index selection strategy of `UTune` is then prompted to further explore such indexes in different configurations. Eventually, when the beneficial index combination is sampled, the query performance stabilizes, leading `UTune` to converge.

IV. Uncertainty-aware Cost Correction

We now discuss how to correct the operator-level what-if costs to obtain the adjusted plan cost $c'(q, X)$, by considering the learned operator-level CAMs. To this end, we develop an uncertainty-aware cost correction framework.

A. Uncertainty Quantification

Given the limited amount of query execution feedback and the constant change of workload queries, uncertainty is ubiquitous in online index tuning. To quantify the degree of uncertainty with respect to the operator-level CAM predictors, we consider two major sources of uncertainty:

- *Data uncertainty* (a.k.a. *aleatoric uncertainty* [28]), which stems from the inherent noise and variability in online

workload execution, due to environmental factors such as caching and resource contention. As a result, the same operator may observe different execution times, introducing irreducible variance in the training data.

- *Model uncertainty* (a.k.a. *epistemic uncertainty* [28]), which arises from the limitations of a model's "knowledge." For example, some operators may appear less often than others. As a result, they have fewer training data points and the corresponding operator-level models are less confident when making predictions.

Quantification of Data Uncertainty: We propose using **entropy** to measure data uncertainty, defined as follows:

Definition 1 (Entropy)**.** *Given a* softmax *output of an operator-level CAM predictor $\mathcal{M}$ for the operator o, $\mathcal{M}(o) = [p_1, p_2, ..., p_{|\Omega|}]$, where p_j is* probability *of the CAM $\omega_j \in \Omega$,*

$$\textit{Entropy}(o, \mathcal{M}) = -\sum\nolimits_{\omega_j \in \Omega} p_j \log(p_j).$$

Here, we have abused the notation $\mathcal{M}(o)$ to also represent the softmax output of $\mathcal{M}$. Intuitively, entropy measures the inherent variance observed in the execution time of an operator. When an operator-level CAM predictor sees noisy or ambiguous data during training, its entropy increases. In fact, the entropy is maximized when all CAM outputs are equally likely, implying a highly uncertain situation where the predicted CAM degenerates to a random guess.

Quantification of Model Uncertainty: We propose using **Monte Carlo dropout (MCD)** [29] to measure model uncertainty. It allows us to assess how confident a model is about its predictions across different regions of the input space. Dropout is a standard regularization technique used in the *training* of deep neural networks to prevent overfitting by randomly deactivating neurons. MCD extends this dropout idea to model *inference*, with a very different purpose of measuring the variance of model outputs with random probing of the structure of the neural network.

Specifically, we pass the same encoded relational operator o through the CAM predictor $\mathcal{M}$ for m times. In this way, we obtain slightly different predictions: $\hat{y}_1, \hat{y}_2, ..., \hat{y}_m$, where each $\hat{y}_i$ is a softmax output vector $\hat{y}_i = [p_{i,1}, p_{i,2}, ..., p_{i,|\Omega|}]$. We then compute the variance for each possible CAM output $\omega_j \in \Omega$ as $\text{Var}(\omega_j) = \frac{1}{m}\sum_{i=1}^{m}(p_{i,j} - \bar{p}_j)^2$, where $\bar{p}_j$ represents the mean predicted probability of ω_j. We take the maximum variance across the CAM outputs to obtain the MCD:

$$MCD(o, \mathcal{M}) = \max_{\omega_j \in \Omega} \text{Var}(\omega_j) = \max_{\omega_j \in \Omega} \frac{1}{m} \sum\nolimits_{i=1}^{m} (p_{i,j} - \bar{p}_j)^2.$$

Intuitively, for regions in the input space where the model has not found strong functional mappings to the outputs (e.g., due to lack of training data), MCD introduces randomness in model inference where each "reduced network" with randomly deactivated neurons predicts differently, resulting in higher prediction variability with the same input.

Combined Uncertainty: We combine data and model uncertainty with the following weighing mechanism:

$$U(o, \mathcal{M}) = \alpha \cdot \text{MCD}(o, \mathcal{M}) + (1 - \alpha) \cdot \text{Entropy}(o, \mathcal{M}), \quad (1)$$

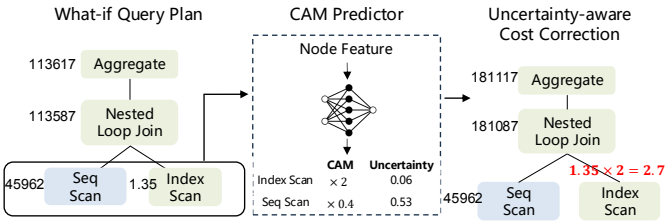


Fig. 3: An example of uncertainty-aware cost correction

where $0 < \alpha < 1$ is a user-specified weight parameter. We set $\alpha = 0.5$ by default, meaning that we treat data and model uncertainty equally. However, one can adjust α based on different situations. For example, one may want to set $\alpha < 0.5$ if query execution exhibits high variance, implying higher data uncertainty on average.

B. Cost Correction and Combination

Algorithm 2: `CostCorrection`(P, $\mathcal{M}$, ρ)

Input: P, query plan; $\mathcal{M}$, operator-level CAM predictors; ρ, uncertainty threshold.
Output: $c'(P)$, corrected cost of P.

for $o \in P.leaves$ **do**
  $U(o, \mathcal{M}) \leftarrow \alpha \cdot \text{MCD}(o, \mathcal{M}) + (1 - \alpha) \cdot \text{Entropy}(o, \mathcal{M})$;
  if $U(o, \mathcal{M}) \leq \rho$ **then** `UpdateCost`$(P, o, \mathcal{M}(o))$;
return $c'(P)$;

Algorithm 3: `UpdateCost`(P, o, ω)

Input: o, the operator whose cost to be corrected; ω, CAM.

if o *is leaf* **then** $c'_e(o) \leftarrow \omega \cdot c_e(o)$;
else Update $c_s(o)$ and $c_e(o)$ via Table II ;
`UpdateCost`$(P, o.\text{parent}, \omega)$;

The uncertainty metric $U(o, \mathcal{M})$ in Equation 1 results in a natural operator-level uncertainty-aware cost correction mechanism as illustrated in Algorithm 2. Specifically, we introduce a threshold parameter ρ to control the acceptable degree of model uncertainty. Cost correction is applied only to *leaf* (e.g., table access and index access) operators with uncertainty below ρ, given their prominent impact on query execution time in the context of index tuning [13]. If so, we invoke the `UpdateCost` function detailed by Algorithm 3 to update its cost as well as its parent's cost recursively.

For each operator o, we update its cost by following an analytic approach [5]. We maintain its *startup cost* $c_s(o)$ and *execution cost* $c_e(o)$, and we only correct the execution cost $c_e(o)$ based on the given CAM. The total cost of o is the sum of $c_s(o)$ and $c_e(o)$. Table II presents the formulas supported by PostgreSQL. Specifically, we can express the *variation* (i.e., change) of the startup cost (resp. execution cost) of an operator o, denoted as $\Delta c_s(o)$ (resp. $\Delta c_e(o)$), in terms of the cost variations of its child operator(s). For example, for PostgreSQL's `Nested-Loop Join` operator, the startup cost variation is the sum of its children's variations, while the execution cost variation also depends on the number of rows from the outer child. After computing the variations in the start and execution costs of an operator o via bottom-up

TABLE II: Cost formulas of relational operators supported by PostgreSQL

Operator o	Startup Cost Variation $\Delta c_s(o)$	Execution Cost Variation $\Delta c_e(o)$
`Nested-Loop Join`	$\sum \Delta c_s(\texttt{child})$	$\text{rows}(\texttt{outer}) \cdot \Delta c_e(\texttt{inner}) + \Delta c_e(\texttt{outer})$
`Limited`	$c_s(\texttt{child})$	$\frac{\text{rows}(\texttt{parent})}{\text{rows}(\texttt{child})} \cdot \Delta c_e(\texttt{child})$
`Hash,Sort,Aggregate,Gather`	$\Delta c_s(\texttt{child}) + \Delta c_e(\texttt{child})$	0
`Hash Join, Gather Merge`	$\sum \Delta c_s(\texttt{child})$	$\Delta c_e(\texttt{child})$

propagation, we can simply update $c_s(o) \leftarrow c_s(o) + \Delta c_s(o)$ and $c_e(o) \leftarrow c_e(o) + \Delta c_e(o)$.

Below, we illustrate how to use the cost formulas for propagating leaf cost corrections (or equivalently, cost variations) with a concrete example based on the query plan in Figure 2a.

Example 1. *Figure 3 illustrates the process of uncertainty-aware cost correction on a simplified query plan shown in Figure 2. The predicted CAMs for leaf operators* `Seq Scan` *(SS) and* `Index Scan` *(IS) are* 0.4 *and* 2*, with uncertainty scores* 0.53 *and* 0.06*. With threshold* 0.1*, only* `IS` *is corrected:*

$$c_e(\texttt{IS}) = 1.35, \quad c'_e(\texttt{IS}) = 1.35 \times 2 = 2.70, \quad \Delta c_e(\texttt{IS}) = 1.35.$$

The corrected cost estimation of `IS` *is then propagated to the parent operator* `Nested-Loop Join` *(NLJ) according to the cost formulas in Table II:*

$$\begin{aligned} c'_e(\texttt{NLJ}) &= c_e(\texttt{NLJ}) + \textit{rows}(\texttt{SS}) \cdot \Delta c_e(\texttt{IS}) + \Delta c_e(\texttt{SS}) \\ &= 113587 + 50000 \times 1.35 + 0 = 181087, \end{aligned}$$

One question is how to specify the uncertainty threshold ρ. In our current implementation of `UTune`, we choose a uniform uncertainty threshold $\rho = 0.1$ based on the observed distribution of data and model uncertainty. It remains an interesting direction for future work to explore more adaptive ways of varying the uncertain threshold ρ as the process of online index tuning proceeds.

V. Uncertainty-Aware Index Selection

Existing online index tuners typically employ RL-based search strategies to balance *exploration* and *exploitation*, which either directly rely on execution feedback or use learned models for reward estimation. While direct usage of observed execution feedback provides accurate estimation, it lacks generalization to unseen queries. Learned models improve generalization, but existing approaches [14, 30, 31] do not consider model uncertainty as far as we know.

In this section, we show how the uncertainty quantification mechanism in Section IV-A can be used to improve RL-based index selection. We first propose a new uncertainty-aware index value function that rebalances exploration with exploitation. We then design a variant of the classic ϵ-greedy action/index selection strategy by integrating the uncertainty-aware index value function.

A. Uncertainty-aware Index Value Function

We propose an online index value function that dynamically balances between the *immediate reward* (i.e., improvement in workload execution time by deploying an index) and the *long-term reward* (i.e., improvement in model prediction accuracy with the new workload execution time feedback).

Immediate Reward: We define the immediate reward of deploying an index x as its *execution benefit* $\text{EB}(x, W)$, i.e., the improvement on workload execution time:

$$\text{EB}(x, W) = \frac{\sum_{q \in W} c(q, \emptyset) - \sum_{q \in W} c(q, x)}{\sum_{q \in W} c(q, \emptyset)} = 1 - \frac{\sum_{q \in W} c(q, x)}{\sum_{q \in W} c(q, \emptyset)}.$$

Long-term Reward: We define the long-term reward of deploying an index x as its *exploratory value* $\text{EV}(x, W, \mathcal{M})$, i.e., the potential improvement in model prediction. This is inspired by *active learning*: the more *uncertain* the model $\mathcal{M}$ is about its predictions on (what-if) query plans containing the index x as access paths, the more *valuable* the workload execution time feedback is if we create the index x. In this spirit, we define $\text{EV}(x, W, \mathcal{M}) = \sum_{o \in \mathcal{O}_x} U(o, \mathcal{M})$, where $U(o, \mathcal{M})$ is the uncertainty metric defined by Equation 1, and $\mathcal{O}_x$ represents the set of all relevant relational operators that access the index x.

Index Value Function: We now define the *total value* $V(x, W)$ of a candidate index x for the workload W by combining $\text{EB}(x, W)$ and $\text{EV}(x, W, \mathcal{M})$ as follows:

$$V(x, W) = \text{EB}(x, W) \cdot \Big(1 + \lambda \cdot \text{EV}(x, W, \mathcal{M})\Big). \tag{2}$$

The $\lambda \in (0, 1)$ here is an *exploration weight* that further controls the degree of exploration as online tuning proceeds. We multiply $\text{EB}(x, W)$ and $\text{EV}(x, W)$ instead of adding them, due to their different semantics. Intuitively, one can think of $\text{EV}(x, W, \mathcal{M})$ as an uncertainty-based weight placed on $\text{EB}(x, W)$, and the total value is simply the uncertainty-adjusted index benefit.

B. Index Selection Strategy

We develop a new variant of the classic ϵ-greedy search strategy [22] for index/action selection by integrating the index value function $V(x, W)$. In standard ϵ-greedy algorithm, the best action is chosen with probability $1 - \epsilon$, and other actions are selected uniformly randomly with probability ϵ, making no distinction between suboptimal choices. *Boltzmann exploration* [32] addresses this by choosing an action based on its softmax probability w.r.t. the estimated action value. However, it requires tuning a temperature parameter τ. To retain the distinction capability of Boltzmann exploration without the need of tuning τ, following [30] we redefine the index selection probability $\Pr(x) = \frac{V(x, W)}{\sum_{x' \in \mathcal{X}} V(x', W)}$. That is, we pick an index with probability proportional to its estimated index value.

Dynamic Adjustment of Exploratory Value: We further adjust the exploratory value using the exploration weight λ. Initially, with little feedback on any index, more exploration is encouraged. After we observe more and more workload execution telemetry as tuning proceeds, exploration needs to be gradually discouraged. Therefore, we set $\lambda_t = \lambda_0 \cdot \gamma^t$

Algorithm 4: Index Configuration Enumeration

Input: W_t, the current mini-workload at time t; $\mathcal{X}_t$, the candidate indexes for W_t; K, the maximum number of indexes to be selected; $\mathcal{M}$, the operator-level CAM predictors; $\lambda_0 = 0.5$, the initial exploration weight; γ, the decay rate.

Output: X_t, the recommend index configuration for W_t.

$\lambda \leftarrow \lambda_0 \cdot \gamma^{\beta t}$;
foreach $x \in \mathcal{X}_t$ **do**
  $V(x, W_t) \leftarrow \text{EB}(x, W_t) \cdot \Big(1 + \lambda \cdot \text{EV}(x, W_t, \mathcal{M})\Big)$;
foreach $x \in \mathcal{X}_t$ **do**
  $\Pr(x) = \frac{V(x,W_t)}{\sum_{z \in \mathcal{X}_t} V(z,W_t)}$;
$X_t \leftarrow \emptyset$;
for $1 \leq k \leq K$ **do**
  Select x_k from $\mathcal{X}_t$ w.r.t. $\Pr(x_k)$;
  if x_k *cannot be pruned* **then** $\mathcal{X}_t \leftarrow \mathcal{X}_t \cup \{x_k\}$;
return X_t;

when tuning the mini-workload W_t, where λ_0 is the initial exploration weight and $0 < \gamma < 1$ is a decay factor dictating how fast exploration diminishes. To respond proactively to workload drift, we further introduce a dynamic reset mechanism controlled by a reset factor $\beta = 1 - \frac{N_{\text{unseen}}}{N}$, which represents the proportion of unseen queries in the present mini-workload W_t. We incorporate this reset mechanism into the exploration weight as $\lambda = \lambda_0 \cdot \gamma^{\beta t}$.

Configuration Enumeration: Algorithm 4 presents the details of the index configuration enumeration algorithm based on the above variant of ϵ-greedy index selection strategy. When a candidate index x_k is selected, we check whether x_k can be pruned according to the following rules: (1) the table on top of which x_k is recommended has reached the maximum number of indexes allowed; (2) x_k can be superseded by a "covering index" that has already been selected; or (3) a similar index with the same prefix of (but more) key columns has already been selected. We include x_k into the final recommended configuration X_t if x_k cannot be pruned.

VI. Experimental Evaluation

We evaluate the performance of UTune and compare it with existing state-of-the-art (SOTA) online index tuning solutions.

A. Experimental Setup

Benchmark Datasets: We use three publicly available benchmark datasets with different scales and characteristics: **TPC-H**, **TPC-DS**, and **JOB** (a.k.a. the "join order benchmark" [33]). Table III summarizes the properties of these datasets. Following previous studies [31, 34], we exclude the queries 4, 6, 10, 11, 35, 41, and 95 from **TPC-DS** because their execution costs are orders of magnitude higher than the other queries. Including these expensive queries can make the index selection problem less challenging, because indexes that can accelerate any of such queries are clearly favorable.

Experiment settings: We conduct all experiments on a workstation with two 24 Core Xeon(R) Gold 5318Y CPU at 2.10GHz and 1TB main memory, running Ubuntu 20.04.6 LTS with PostgreSQL 12.1. We clear the database buffer pool and the file system cache before running each mini-workload.

TABLE III: Properties of the benchmark databases

Benchmark	Dataset	#Tables	#Attributes	#Templates	Size (GB)
TPC-H	Synthetic uniform	8	61	22	10
TPC-DS	Synthetic skew	24	429	99	10
JOB	Real-world	21	108	33	6

TABLE IV: Estimation accuracy of index benefit estimators

Dataset	What-if	LIB		UTune	
		40%	80%	40%	80%
TPC-H	0.189	0.171	0.124	0.137	0.115
TPC-DS	0.062	0.118	0.046	0.056	0.039
JOB	0.757	0.762	0.531	0.544	0.386

Baselines: We compare UTune with three online RL-based index tuners/advisors and one offline index tuner/advisor specifically designed for dynamic workloads:

- **AutoIndex** [15], which is an online index advisor that uses Monte Carlo tree search (MCTS) for index selection and a deep regression model for index benefit estimation;
- **HMAB** [35], which applies hierarchical multi-armed bandits for online index selection and learns index benefit from query execution feedback; **HMAB** further extends and improves over DBA Bandits [14], which pioneered the idea of modeling online index tuning as contextual bandits.
- **Indexer++** [17], which is an online index advisor that uses deep Q-learning (DQN) for index selection and prioritized experience sweeping for adaption to dynamic workloads;
- **SWIRL** [31], which is an index advisor that uses deep reinforcement learning (DRL) to train an index selection policy offline and applies a sophisticated workload embedding model for generalization over unseen queries.

Performance Metrics: We use *improvement of workload execution time* as our primary performance metric, defined as $\frac{\sum_{t=1}^{T}\Big(C_{\text{exe}}(W_t,\emptyset) - C_{\text{exe}}(W_t,X_t)\Big)}{\sum_{t=1}^{T} C_{\text{exe}}(W_t,\emptyset)} \times 100\%$, where $C_{\text{exe}}(W_t, \emptyset)$ is the execution time of the mini-workload W_t without indexes and $C_{\text{exe}}(W_t, X_t)$ is the execution time of W_t with the recommended index configuration X_t.

B. Evaluation of Index Benefit Estimation

We evaluate UTune's *operator-level* index benefit estimation model to assess its accuracy, generalization capability, and impact on index tuning. We compare UTune against two baselines: the traditional what-if cost estimator and **LIB** [7], a state-of-the-art *learned query-level* estimator.

Estimation Accuracy: We use "Mean Absolute Error (MAE)" to measure the accuracy of an index benefit estimator, which is defined as $\frac{1}{n}\sum_{i=1}^{n}\Big(|b_c(x_i,q) - b_t(x_i,q)|\Big)$, where $\{x_i\}_{i=1}^{n}$ represents the set of all candidate indexes. A smaller MAE means a more accurate estimator.

For the CAM predictors employed by UTune and the **LIB** baseline, we randomly select 40% and 80% of all query templates for model training, and we test the trained models using all query templates. This is different from a standard "train-test split" model validation procedure, where trained models are tested using only holdout data. We chose to use this setup instead of the standard one, since it is closer to

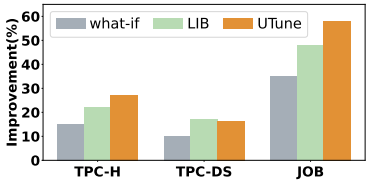
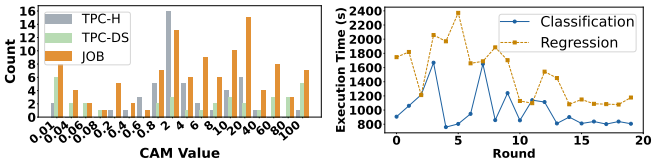


Fig. 4: Improvement of workload execution time when integrating greedy search with index benefit estimators



(a) Distribution of ω (b) Classifier vs. regressor

Fig. 5: Validation of CAM model design choices.

the use case when deploying learned index benefit estimators for real online index tuning. Specifically, given that workload drifts are unpredictable, a more reasonable goal for online index benefit estimators is to generalize over all observed query templates. The setup that we chose therefore emulates the evolving generalization behavior of online index benefit estimators as more and more query templates are observed.

Table IV summarizes the MAE results.

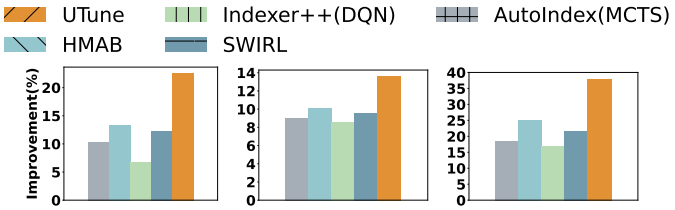
`UTune` consistently outperforms both baselines, especially in the data-scarce scenario (with 40% training data) and for complex workloads such as **JOB**. The performance gap highlights advantages of `UTune`'s operator-level models over the query-level models used by **LIB**, as discussed in Section III.

Impact on Index Tuning: We are interested in the impact of improved index benefit estimates on end-to-end index tuning. To this end, we use the classic greedy index selection strategy [4] and integrate it with the index benefit estimators. We set the maximum number of indexes allowed to 8. Figure 4 presents the index tuning results in terms of the improvement on workload execution time using recommended indexes. Both **LIB** and `UTune` outperform what-if cost in selecting effective indexes with knowledge learned from actual query execution feedback data. `UTune` performs even better than **LIB** on **JOB** because of its success in detecting indexes that cause severe query regression. `UTune` is equally competitive as **LIB** in correcting what-if cost estimation errors but features a simpler structure and better generalization capability. Therefore, it can work more effectively for online index tuning with limited amount of query execution feedback data.

C. Validation of Design Choices of CAM Models

Figure 5a presents the distributions of the CAM values for the workloads used in our evaluation, confirming that the effective range of Ω spans a broad spectrum (Section III-B).

To validate our design choice of formulating CAM prediction as a classification task (Section III-B), we implemented a regression-based variant of `UTune`. To train the regression model, we perform a binary search to find the CAM value (between 0.1 and 100) that minimizes the error between the estimated and actual index benefits, within a specified



(a) TPC-H (b) TPC-DS (c) JOB

Fig. 6: Improvement on the execution time of static workload when tuned by different index advisors

tolerance. Figure 5b compares the execution time of **JOB** with the two models. The classification model converges to a better configuration (with lower execution time) compared to the regression model because it effectively handles the volatility of **JOB** with its non-uniform candidate buckets Ω. It also identifies the best configuration much faster (around round 4) than the regression model (around round 11) with less exploration costs in the initial rounds.

D. Workloads for Online Index Selection

We evaluate the performance of `UTune` and compare it with SOTA index advisors, using static and dynamic workloads. Table V summarizes the workloads used in our evaluation.

Static Workload: The query templates do not vary across mini-workloads, and each mini-workload contains all query templates. Static workload is useful for evaluating the capability of `UTune` to utilize execution feedback in a stable operational environment.

a) Dynamic Workload: We introduce three types of dynamic workload to evaluate `UTune` in more realistic online index tuning scenarios:

- **(Continuous Variation)** A certain proportion (e.g., 20%) of query templates will vary in each consecutive mini-workload, which we refer to as a *workload drift* below. We use continuous variation to assess an index advisor's ability to adapt to constantly evolving workload drifts.
- **(Periodic Variation)** Workload drift occurs every 4 or 5 mini-workloads. We use periodic variation to evaluate how quickly an index advisor can catch up with workload drift.
- **(Cyclic Variation)** A *cycle* comprises a repeating sequence of 15 drifting mini-workloads. We use cyclic variation to evaluate an index advisor's ability of reusing its cumulative knowledge learned from past execution history.

E. Online Evaluation with Static Workload

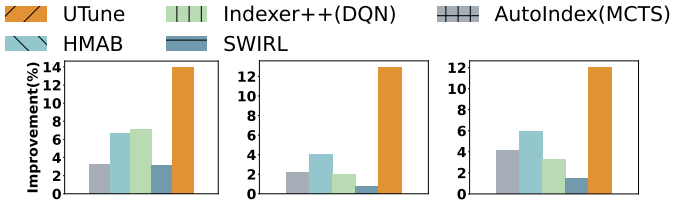
Figure 6 presents the improvement on the execution time of the static workload when tuned by different online index advisors. We replay the same workload 20 times. We observe that `UTune` outperforms the best baseline index advisor by 7%, 4%, and 11% for **TPC-H**, **TPC-DS**, and **JOB**, respectively. A more detailed analysis can be found in the extended full version [27].

F. Online Evaluation with Dynamic Workload

1) Continuous Variation: Figure 7 presents the overall improvement in workload execution time under continuously

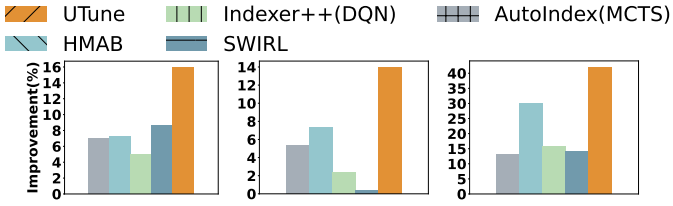
TABLE V: Static and dynamic workloads used in online index tuning evaluation

Workload	Workload Drift	T	Templates per W_t	Queries per template
Static	No	20	all templates	20 (**TPC-H**) 10 (**TPC-DS**, **JOB**)
Continuous Variation	Change of 20% templates / W_t	24	10 (**TPC-H**),20 (**TPC-DS**, **JOB**)	20
Periodic Variation	Change of 20% templates every 4 (**TPC-H**) or 5 (**TPC-DS**, **JOB**) W_t	24 (**TPC-H**) 30 (**TPC-DS**, **JOB**)	10 (**TPC-H**),20 (**TPC-DS**, **JOB**)	20
Cyclic Variation	Change of 20% templates / W_t, recurring every 15 W_t	30	16 (**TPC-H**), 20 (**TPC-DS**, **JOB**)	20



(a) TPC-H (b) TPC-DS (c) JOB

Fig. 7: Improvement of workload execution time when tuned by different index advisors under continuous variation



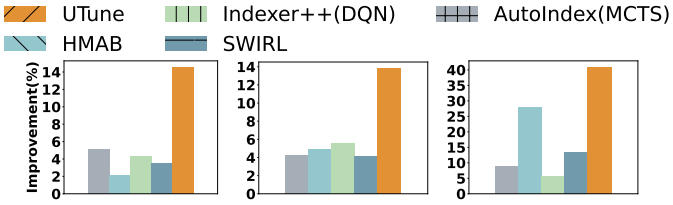
(a) TPC-H (b) TPC-DS (c) JOB

Fig. 8: Improvement of workload execution time when tuned by different index advisors under periodic variation

changing workloads. Compared to the best baseline, UTune offers 7% more (i.e., from 7% to 14%) improvement on **TPC-H**, 7% more (i.e., from 5% to 12%) improvement on **TPC-DS**, and 6% more (i.e., from 6% to 12%) improvement on **JOB**.

Continuous variation presents the most challenging case among the dynamic workloads considered in our evaluation. **SWIRL** performs the worst because it is designed for offline index tuning. Its index selection policy (trained on W_0) fails on unseen queries during workload drifts. Other online baselines such as **HMAB** leverage query execution feedback passively and cannot react as fast as workload drifts that occur constantly. By contrast, UTune exhibits stronger adaptive capability to unseen query templates, attributed to its uncertainty-aware operator-level cost correction framework. More detailed analysis can be found in [27].

2) Periodic Variation: Each mini-workload of **TPC-H** contains 10 query templates, whereas workload drift occurs every 4 mini-workloads. That is, the same mini-workload will repeat 4 times before a workload drift occurs. Similarly, each mini-workload of **TPC-DS** and **JOB** contains 20 query templates, and workload drift occurs every 5 mini-workloads. We chose these settings based on the number of available query templates in each benchmark to allow for sufficient diversity of workload drifts. Figure 8 presents the improvement of overall workload execution time under periodic variation. We observe that UTune outperforms the best baseline index tuner by 7% on **TPC-H** (i.e., 16% vs. 9%), by 6% on **TPC-DS** (i.e., 14% vs. 8%), and by 12% on **JOB** (i.e., 42% vs. 30%).



(a) TPC-H (b) TPC-DS (c) JOB

Fig. 9: Improvement of workload execution time when tuned by different index advisors under cyclic variation

Periodic variation evaluates how quickly online index advisors converge under a relatively stable workload. Faster convergence can result in lower index exploration overhead, which is often nontrivial. Among the online index advisors, we observe that UTune and **HMAB** typically converge faster than **AutoIndex** and **Indexer++** during the stable period of a mini-workload, but UTune typically converges to a better index configuration, resonating with the observation on static workloads. We attribute the fast convergence of UTune to its uncertainty-aware selection policy, which prioritizes indexes with low model confidence and thus improves benefit estimates with fewer explorations. See [27] for more analysis.

3) Cyclic Variation: One common limitation of existing RL-based online index selection policies is "knowledge forgetting." For example, **HMAB** resets its internal parameters when significant workload drifts are detected, whereas **Indexer++** adopts *prioritized sweeping* by focusing on the most important updates to the value function. As a result, these strategies weigh more on their recent search experience than their past search experience. This is in general not a problem if workload keeps changing, such as in the case of continuous variation. However, for recurring workload with a long period, such as in the case of cyclic variation, these strategies are not able to utilize their experience learned from the past history. In contrast, this is not a limitation of UTune, which captures all historical query execution feedback via its operator-level learned index benefit estimators. Figure 9 presents the improvement of the total workload execution time given by different index advisors for cyclically varying mini-workloads. UTune improves the best baseline index advisor by 9% (i.e., 14% vs. 5%) on **TPC-H**, by 8% (i.e., 14% vs. 6%) on **TPC-DS**, and by 13% (i.e., 42% vs. 29%) on **JOB**.

G. Impact of Index Tuning Constraints

In online environments, timely index creation and adaptation are critical. To quantify the overheads and benefits of indexes, Table VI presents a detailed breakdown of index creation

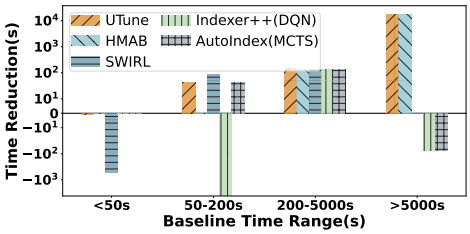


Fig. 10: Excluded long-running **TPC-DS** queries

overhead (including time and space costs) versus workload execution time under varying index budgets (encompassing both quantity and storage limits). We omit the results of **SWIRL** and **Index++** because they are similar to **AutoIndex**. Our evaluation does not include index updates because we focus on **OLAP** workloads in this paper, following previous work on index tuning [30, 31]. However, `UTune` can be extended to handle OLTP workloads, where update costs are critical, by introducing operator-level cost models for modification operators (e.g., `INSERT`, `UPDATE`, and `DELETE`).

The space consumption differs marginally across different algorithms, as it is primarily bounded by the index tuning budget. `UTune` consistently achieves lower execution time than baselines, especially under a tuning budget. Although it incurs higher index creation time as the budget grows (due to the actual index creation during exploration), the corresponding reduction in execution time compensates more for this overhead, making the exploration cost acceptable in practice. In contrast, **HMAB** has lower index creation time but much higher execution time, as its linear reward function over-penalizes candidates on certain tables and therefore leads to suboptimal convergence. **AutoIndex** suffers the most significant performance decline as the tuning budget increases (e.g., from 8 to 12 indexes allowed). This is due to the expanded search space, which dramatically increases the exploration overhead of its MCTS-based selection policy, leading to premature and inadequately explored index choices.

H. Analysis of Long-running TPC-DS Queries

We now analyze the performance of `UTune` for the previously excluded long-running **TPC-DS** queries. The average execution time of these queries is over 1,000 seconds, while the average execution time of the other **TPC-DS** queries is around 6 seconds. We further classify these queries into four groups based on their execution time. As shown in Figure 10, `UTune` achieves orders-of-magnitude performance gains compared to what-if based index advisors (e.g., **SWIRL**, **AutoIndex**, **Index++**). This significant advantage is driven by the inaccurate what-if cost estimates for these queries. On the other hand, both feedback-based index advisors, `UTune` and **HMAB**, achieve comparable performance.

I. Hyper-parameter Sensitivity Analysis

We analyze the impact of several critical hyper-parameters on the behavior of `UTune`. Figure 11 summarizes the results.

Index Estimation: The uncertainty threshold ρ determines which CAM predictions are trustworthy enough to correct the what-if cost estimates. As Figure 11a shows, setting ρ around 0.1 provides a stable baseline across different benchmarks, filtering out highly volatile predictions while allowing necessary adjustments. Complementing ρ, the uncertainty weight α regulates the internal composition of the uncertainty metric, balancing between model and data uncertainty. As Figure 11b shows, $\alpha \in [0.3, 0.7]$ works well for most scenarios.

Index Selection: The hyper-parameters λ_0 and γ govern the trade-off between exploration and exploitation during the index selection process. The initial exploration weight λ_0 determines the magnitude of the exploration incentive during the cold-start phase, while the decay rate γ controls how quickly this incentive diminishes as feedback being collected. `UTune` exhibits low sensitivity to these two parameters, as shown in Figures 11c and 11d. In general, setting $\lambda_0 = 0.5$ and $\gamma = 0.9$ works well across all workloads.

J. Case Study

As a case study to better understand the efficacy of `UTune`, we present a detailed example from **JOB**, which involves two candidate indexes: (1) I_{mi}, `movie_info(movie_id)` and (2) I_{cc}, `complete_cast(status_id, movie_id)`.

SWIRL and **AutoIndex** rely on what-if cost estimates. They fail on this example in two ways:

- (**Overestimation**) I_{mi} is a popular index on different query templates and is selected frequently because of the predicted high global benefit. However, it can cause severe performance regression [36] on query **Q29** (5s $\rightarrow$ 29s) due to a `Nested-Loop Join` with overestimated benefit;
- (**Underestimation**) I_{cc}, a covering index for **Q23**, is usually discarded due to its estimated low benefit (1%), missing a significant actual speedup (23s $\rightarrow$ 0.8s).

`UTune` overcomes the pitfalls with the cost correction mechanism. By learning from execution feedback, the CAM models penalize I_{mi} for the observed regression and boosts I_{cc} for its true benefit. Although **HMAB** also utilizes execution feedback, it fails to recommend I_{cc} due to its misguided exploration with its simplistic linear reward modeling that tends to prioritize complex and high-cost indexes.

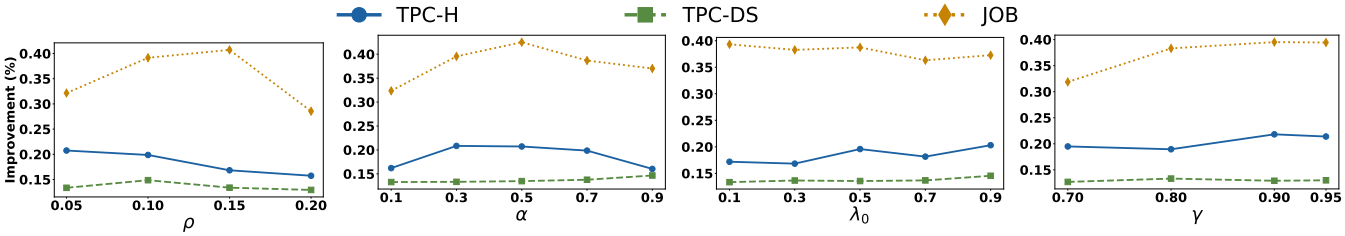
VII. Discussion on Generalizability

Although currently built on top of PostgreSQL, `UTune` can be extended to other DBMSs, distributed settings, and Lakehouse systems. `UTune` is based on correcting the query optimizer's cost model by analyzing the execution feedback. Therefore, it can be extended to other DBMSs as long as they support *cost-based* query optimization and query *execution feedback*. The extension involves updating the operator-level cost models in Section III-D and the cost correction formulas in Table II with respect to the target system.

`UTune` can also be extended to distributed settings (such as CockroachDB) and Lakehouse systems (Databricks Lakehouse) that support query cost estimations and execution feedback. Although lakehouses generally do not support B-trees, which are commonly used by DBMSs, they can still leverage `UTune` as long as they support indexing. For example, Databricks Lakehouse utilizes Z-ordering indexing, allowing `UTune` to be adapted accordingly.

TABLE VI: Analysis of index maintenance overheads under different constraints

Tuning Constraints		Creation(min)			Execution(min)			Space(M)		
		UTune	HMAB	MCTS	UTune	HMAB	MCTS	UTune	HMAB	MCTS
Number(#)	2	0.48	0.06	0.32	38.0	39.7	40.3	1,102	1,807	1,380
	4	0.78	0.46	0.37	35.3	39.3	38.3	1,898	1,712	2,297
	8	0.86	0.4	0.44	33.7	37.3	38.7	2,911	3,572	2,580
	12	1.05	0.57	0.61	33.3	37.6	40.0	3,241	3,758	3,414
Storage(M)	2000	0.26	0.45	0.15	36.3	40.3	41.0	1,413	1,513	1,703
	4000	0.70	0.63	0.30	35.7	38.0	39.3	2,426	2,415	1,874
	6000	0.83	0.70	0.37	34.0	38.7	39.0	3,870	3,729	3,098



(a) Uncertainty threshold ρ (b) Uncertainty weight α (c) Initial exploration weight λ_0 (d) Decay factor γ

Fig. 11: Sensitivity of UTune's performance with respect to varying hyper-parameter values

VIII. Related Work

Classic Index Tuning: The problem of index tuning has been studied for decades in the *offline* setting, where a static workload and certain constraints (e.g., limitation on storage space) are given and the goal is to find a set of indexes under the constraints that minimizes the workload execution time. Most commercial and open-source offline index tuning software [3, 37] employs a cost-based architecture that replies on the what-if API [4] provided by the query optimizer. There has been extensive work on making effective what-if query optimizer calls [30, 38–43] to improve the efficiency of index tuning, but their accuracy remains limited by the underlying cost model of query optimizer due to well-known challenges such as cardinality estimation and analytic cost modeling.

Online Index Tuning: Unlike offline index tuning, *online* index tuning [24] operates under workload drifts, which is arguably a more realistic setup in real-world index tuning applications [9, 10]. It is typically formulated as a Markov decision process (MDP) with RL-based solutions [14–18, 44–47]. While RL-based index tuning methods have shown promise in adapting to dynamic workloads, they often struggle with the *"cold-start"* problem, such that the lack of reliable execution feedback in early tuning rounds leads to poor initial recommendations and slow convergence. As we mentioned in the introduction, recent work on modeling index tuning as contextual bandits [19–21] attempted to mitigate this challenge by incorporating historical knowledge and transfer learning, but failed to capture more complex index benefit functions in real-world workloads due to the simplified assumption of a linear index benefit function.

Learned Index Benefit Estimators: Accurate estimation of query execution cost given an index configuration is therefore central to both offline and online index tuning. What-if cost estimates are relatively cheap to obtain but inaccurate, while true execution feedback is accurate but costly. This inspires a recent line of work on learned index benefit estimators [1, 6, 7, 31, 48, 49], which operate in the "middle ground" using query execution data to train learned cost models. However, existing learned index estimators are mainly designed for offline index tuning and struggle in online settings due to the requirement of large amounts of training data and limited generalization capability over unseen queries.

Uncertainty Quantification: One key idea UTune brings into online index tuning is *uncertainty quantification*, which has various applications in related areas such as query execution time estimation and robust query optimization [50–52]. Uncertainty is essential for learned index benefit estimators trained in an online fashion, especially during early stages of online index tuning where very limited query execution feedback data are available. Our way of quantifying uncertainty by considering both aleatoric uncertainty (i.e., data uncertainty) and epistemic uncertainty (i.e., model uncertainty) is motivated by the recent work [53]. However, unlike their autoencoder-based approach designed for offline tuning, which faces similar generalization challenges over unseen queries as other offline learned index benefit estimators, UTune leverages operator-level CAM predictors, which generalize better with limited online training data and incur lower computational overhead.

IX. Conclusion

We have presented UTune, an uncertainty-aware online index tuner that employs operator-level learned index benefit estimators to address two major challenges in online index tuning, namely, (1) limited query execution feedback data for model training and (2) limited generalization capability over unseen queries due to constant workload drifts. We have designed a low-overhead uncertainty quantification mechanism that incorporates both data uncertainty and model uncertainty, and we have integrated it into both the learned index benefit estimators and the index configuration enumeration component of UTune. Our experimental evaluation demonstrates that factoring uncertainty information into the index selection strategy of UTune can significantly improve the workload execution time and reduce the index exploration overhead in online index tuning, compared to state-of-the-art RL-based index tuners.

AI-Generated Content Acknowledgment

The authors used OpenAI's ChatGPT solely for grammar checking and minor language polishing of the manuscript. The AI system was not used to generate research ideas, technical content, analyses, figures, or experimental results. All scientific contributions, methodologies, and conclusions presented in this paper are entirely the work of the authors.

References

[1] T. Siddiqui and W. Wu, "Ml-powered index tuning: An overview of recent progress and open challenges," *ACM SIGMOD Record*, vol. 52, no. 4, pp. 19–30, 2024.

[2] Y. Wu, X. Zhou, Y. Zhang, and G. Li, "Automatic index tuning: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 12, pp. 7657–7676, 2024.

[3] S. Chaudhuri and V. R. Narasayya, "An efficient, cost-driven index selection tool for microsoft sql server," in *VLDB*, vol. 97, 1997, pp. 146–155.

[4] S. Chaudhuri and V. Narasayya, "Autoadmin "what-if" index analysis utility," *ACM SIGMOD Record*, vol. 27, no. 2, pp. 367–378, 1998.

[5] 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*, 2013, pp. 1081–1092.

[6] 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*, 2019, pp. 1241–1258.

[7] J. Shi, G. Cong, and X.-L. Li, "Learned index benefits: Machine learning based index performance estimation," *Proceedings of the VLDB Endowment*, vol. 15, no. 13, pp. 3950–3962, 2022.

[8] T. Yu, Z. Zou, W. Sun, and Y. Yan, "Refactoring index tuning process with benefit estimation," *Proceedings of the VLDB Endowment*, vol. 17, no. 7, pp. 1528–1541, 2024.

[9] 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*, 2019, pp. 666–679.

[10] R. Yadav, S. R. Valluri, and M. Zaït, "AIM: A practical approach to automated index management for SQL databases," in *ICDE*, 2023, pp. 3349–3362.

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

[12] 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.*, vol. 5, no. 11, pp. 1555–1566, 2012.

[13] W. Wu, "Hybrid cost modeling for reducing query performance regression in index tuning," *IEEE Trans. Knowl. Data Eng.*, vol. 37, no. 1, pp. 379–391, 2025.

[14] 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 *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 600–611.

[15] 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 *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 2196–2208.

[16] 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.

[17] 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*, 2022, pp. 372–380.

[18] Z. Sadri, L. Gruenwald, and E. Leal, "Online index selection using deep reinforcement learning for a cluster database," in *ICDEW*, 2020, pp. 158–161.

[19] B. Oetomo, R. M. Perera, R. Borovica-Gajic, and B. I. Rubinstein, "Warm-starting contextual bandits under latent reward scaling," in *2024 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2024, pp. 360–369.

[20] B. Oetomo, R. M. Perera, R. Borovica-Gajic, and B. Rubinstein, "Cutting to the chase with warm-start contextual bandits," *Knowledge and Information Systems*, vol. 65, no. 9, pp. 3533–3565, 2023.

[21] C. Zhang, A. Agarwal, H. Daumé III, J. Langford, and S. N. Negahban, "Warm-starting contextual bandits: Robustly combining supervised and bandit feedback," *arXiv preprint arXiv:1901.00301*, 2019.

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

[23] 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*, 2018, pp. 631–645.

[24] N. Bruno and S. Chaudhuri, "An online approach to physical design tuning," in *ICDE*, 2007, pp. 826–835.

[25] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: a learned query optimizer," *Proc. VLDB Endow.*, vol. 12, no. 11, p. 1705–1718, Jul. 2019. [Online]. Available: https://doi.org/10.14778/3342263.3342644

[26] M. Grohe, "word2vec, node2vec, graph2vec, x2vec: Towards a theory of vector embeddings of structured data," in *PODS*, 2020, pp. 1–16.

[27] "Extended version of the paper," https://arxiv.org/abs/2601.18199, 2026.

[28] S. Deep, A. Gruenheid, P. Koutris, J. Naughton, and S. Viglas, "Comprehensive and efficient workload com-

pression," vol. 14, no. 3, 2020.

[29] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *ICML*, 2016, pp. 1050–1059.

[30] W. Wu, C. Wang, T. Siddiqui, J. Wang, V. Narasayya, S. Chaudhuri, and P. A. Bernstein, "Budget-aware index tuning with reinforcement learning," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1528–1541.

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

[32] L. Péret and F. Garcia, "On-line search for solving markov decision processes via heuristic sampling," *learning*, vol. 16, p. 2, 2004.

[33] V. Leis, "Join order benchmark," https://github.com/gregrahn/join-order-benchmark.

[34] 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," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2382–2395, 2020.

[35] R. M. Perera, B. Oetomo, B. I. Rubinstein, and R. Borovica-Gajic, "Hmab: self-driving hierarchy of bandits for integrated physical database design tuning," *Proceedings of the VLDB Endowment*, vol. 16, no. 2, pp. 216–229, 2022.

[36] W. Wu, A. Dutt, G. Xu, V. R. Narasayya, and S. Chaudhuri, "Understanding and detecting query performance regression in practical index tuning: [experiments & analysis]," *Proc. ACM Manag. Data*, vol. 3, no. 6, pp. 1–26, 2025.

[37] 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*, 2000, pp. 101–110.

[38] M. Brucato, T. Siddiqui, W. Wu, V. Narasayya, and S. Chaudhuri, "Wred: Workload reduction for scalable index tuning," *Proceedings of the ACM on Management of Data*, vol. 2, no. 1, pp. 1–26, 2024.

[39] X. Wang, W. Wu, C. Wang, V. R. Narasayya, and S. Chaudhuri, "Wii: Dynamic budget reallocation in index tuning," *Proc. ACM Manag. Data*, vol. 2, no. 3, p. 182, 2024.

[40] S. Papadomanolakis, D. Dash, and A. Ailamaki, "Efficient use of the query optimizer for automated physical design," in *VLDB*, 2007, pp. 1093–1104.

[41] 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 *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 660–673.

[42] T. Siddiqui, W. Wu, V. Narasayya, and S. Chaudhuri, "Distill: low-overhead data-driven techniques for filtering and costing indexes for scalable index tuning," *Proceedings of the VLDB Endowment*, vol. 15, no. 10, pp. 2019–2031, 2022.

[43] X. Wang, W. Wu, V. R. Narasayya, and S. Chaudhuri, "Esc: An early-stopping checker for budget-aware index tuning," *Proc. VLDB Endow.*, vol. 18, no. 5, pp. 1278–1290, 2025.

[44] 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*, 2015, pp. 253–268.

[45] A. Sharma, F. M. Schuhknecht, and J. Dittrich, "The case for automatic database administration using deep reinforcement learning," *arXiv preprint arXiv:1801.05643*, 2018.

[46] V. Sharma, C. Dyreson, and N. Flann, "Mantis: multiple type and attribute index selection using deep reinforcement learning," in *Proceedings of the 25th International Database Engineering & Applications Symposium*, 2021, pp. 56–64.

[47] Z. Wang, H. Liu, C. Lin, Z. Bao, G. Li, and T. Wang, "Leveraging dynamic and heterogeneous workload knowledge to boost the performance of index advisors," *Proc. VLDB Endow.*, vol. 17, no. 7, pp. 1642–1654, 2024.

[48] Y. Zhao, G. Cong, J. Shi, and C. Miao, "Queryformer: A tree transformer model for query plan representation," *Proceedings of the VLDB Endowment*, vol. 15, no. 8, pp. 1658–1670, 2022.

[49] J. Gao, N. Zhao, N. Wang, and S. Hao, "Smartindex: An index advisor with learned cost estimator," in *CIKM*, 2022, pp. 4853–4856.

[50] F. C. Chu, J. Y. Halpern, and P. Seshadri, "Least expected cost query optimization: An exercise in utility," in *PODS*, V. Vianu and C. H. Papadimitriou, Eds., 1999, pp. 138–147.

[51] B. Babcock and S. Chaudhuri, "Towards a robust query optimizer: A principled and practical approach," in *SIGMOD*, 2005, pp. 119–130.

[52] W. Wu, X. Wu, H. Hacigümüs, and J. F. Naughton, "Uncertainty aware query execution time prediction," *Proc. VLDB Endow.*, vol. 7, no. 14, pp. 1857–1868, 2014.

[53] T. Yu, Z. Zou, and H. Xiong, "Can uncertainty quantification enable better learning-based index tuning?" *arXiv preprint arXiv:2410.17748*, 2024.