HMAB: Self-Driving Hierarchy of Bandits for Integrated Physical Database Design Tuning

R. Malinga Perera
University of Melbourne
Melbourne, Australia
malinga.perera@student.unimelb.edu.au

Bastian Oetomo
University of Melbourne
Melbourne, Australia
b.oetomo@student.unimelb.edu.au

Benjamin I. P. Rubinstein
University of Melbourne
Melbourne, Australia
brubinstein@unimelb.edu.au

Renata Borovica-Gajic
University of Melbourne
Melbourne, Australia
renata.borovica@unimelb.edu.au

ABSTRACT

Effective physical database design tuning requires selection of several physical design structures (PDS), such as indices and materialised views, whose combination influences overall system performance in a non-linear manner. While the simplicity of combining the results of iterative searches for individual PDSs may be appealing, such a greedy approach may yield vastly suboptimal results compared to an integrated search. We propose a new self-driving approach (HMAB) based on hierarchical multi-armed bandit learners, which can work in an integrated space of multiple PDS while avoiding the full cost of combinatorial search. HMAB eschews the optimiser cost misestimates by direct performance observations through a strategic exploration, while carefully leveraging its knowledge to prune the less useful exploration paths. As an added advantage, HMAB comes with a provable guarantee on its expected performance. To the best of our knowledge, this is the first learned system to tune both indices and materialised views in an integrated manner. We find that our solution enjoys superior empirical performance relative to state-of-the-art commercial physical database design tools that search over the integrated space of materialised views and indices. Specifically, HMAB achieves up to 96% performance gain over a state-of-the-art commercial physical database design tool when running industrial benchmarks.

PVLDB Reference Format:
R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. HMAB: Self-Driving Hierarchy of Bandits for Integrated Physical Database Design Tuning. PVLDB, 16(2): 216 - 229, 2022.
doi:10.14778/3565816.3565824

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://go.unimelb.edu.au/k53i.

Proceedings of the VLDB Endowment, Vol. 16, No. 2 ISSN 2150-8097.
doi:10.14778/3565816.3565824

1 INTRODUCTION

Physical design tuning has enjoyed significant research attention in databases over a period of decades. New challenges such as dynamic and multi-tenant environments, and complex workloads, have challenged the status quo, necessitating new research in this established area. Despite the significant attention, most existing efforts focus separately on the tasks of index selection [36] or materialised view selection [47]. Only a handful of off-line solutions have shown capability to work in the complex combined space of multiple physical design structures [1, 3, 78].

While indices work on a single table, materialised views (simply referred to as 'views' hereafter) provide the option to create indexing structures on multiple tables and use high-level constructs such as group-bys and order-bys. This rich structure of views gives rise to a massive action space for even the simplest workloads. Recommending views and indices together for a given workload is an arduous task for a human or an automated tool. Therefore, some tools limit themselves to a single PDS [5, 36, 67]. While it might look straightforward to iteratively select each PDS in isolation to develop a configuration made of multiple PDS, it has been shown that interactions between PDS can lead to sub-optimal recommendations under the iterative approach [1].

Nevertheless, an exhaustive search of a combined search space can quickly explode in combinations. As a solution, [1] uses filtration techniques (candidate selection) to reduce the search space, whereas [78] follows a hybrid approach to only use integrated search when the interaction between two PDS is strong. These solutions depend on the optimiser cost model, usually exposed via a "what-if" interface [13]. Yet, it has been shown that the estimates from the optimiser cost models can significantly differ from actual performance [21], resulting in severe performance regressions [8].

Contextual combinatorial multi-armed bandits (MABs) have been successfully used to avoid shortcomings of optimiser-based recommendation tools [59]. Unlike systems that rely on a misspecified optimiser cost model, bandits learn from the actual performance observations, eschewing the optimiser (mis)estimates, which allows them to adapt to dynamic environments and ad-hoc workloads. The MAB framework facilitates the search strategy by efficiently balancing exploring new actions and exploiting so-far best actions to maximise cumulative rewards under uncertainty. However, such an

architecture can easily get too complex (in context, reward, and oracle design) when considering multiple physical design structures at once. Furthermore, larger contexts directly impact the running time of the solution. In search of truly extensible design, we introduce a hierarchy of contextual and combinatorial MABs for physical design tuning. Our proposed solution consists of two layers. The first layer is responsible for candidate PDS selection, and the second layer considers all the candidate structures together to select the final configuration. Unlike past efforts [1], MABs can handle a vast action space allowing the first layer of bandits to be more flexible in candidate selection and pass on any promising design structure to the second level, rather than limiting ourselves to the per-query optimal structures only. Furthermore, MAB's simpler reinforcement learning formulation comes with a provable guarantee on the average performance, which is hardly the case for general forms of deep reinforcement learning [67].

Despite many advantages of not relying on the optimiser cost model, actual cost observations require PDS materialisations. While this has been shown to work well with the limited action space of indices [59], when it comes to the massive action space of views or the combined action space of multiple PDS, the cost of exploration becomes unaffordable. Our framework thus employs a hybrid approach that learns the benefit of PDSs through the actual query execution, while minimally using the optimiser knowledge to reduce the exploration space. Specifically, the optimiser cost model is leveraged to prune out exploration paths that the optimiser will not consider for query execution.

The key contributions of the paper are:

- We present the first online learned approach to tune multiple PDS in an integrated search space. This novel solution combines the best of both integrated and iterative search approaches for PDS tuning.
- We introduce a new bandit flavour that extends the existing contextual and combinatorial bandit to build a hierarchy of bandits that can handle a large action space and make use of parallel processing capabilities.
- We provide a search method that combines the optimiser knowledge and actual execution statistics to make better recommendations and cut down on PDS creation time (up to 58% reduction in creation time compared to methods that rely solely on execution statistics).
- Our extensive experiments showcase the HMAB's ability to surpass a state-of-the-art commercial physical design tool by providing up to 96% speed-up under dynamic workloads tuning integrated search space of views and indices.
- We demonstrate HMAB's superiority over nine index-tuning solutions including state-of-the-art DBABandit [59], Extend [64], DTA Anytime [14], Relaxation [9], AutoAdmin [15], DB2 Advisor [70] and reinforcement learning based UDO [74] against gold standard industrial benchmarks.

2 PROBLEM FORMULATION

A formal definition of *online physical database design tuning problem* follows [59]. The PDS recommendation tool's goal is to propose a *configuration* sequence $S = (s_0, s_1, \ldots, s_T)$ for the workload sequence $W = (w_1, w_2, \ldots, w_T)$, which minimises the *total workload time* $C_{tot}(W, S)$, when the workload sequence or system run times are not known in advance. Configuration s_t refers to a set of PDS and $C_{tot}(W, S)$ is defined as:

$$C_{tot}(W, S) = \sum_{t=1}^{T} C_{rec}(t) + C_{cre}(s_{t-1}, s_t) + C_{exc}(w_t, s_t) \quad .$$

Here $C_{rec}(t)$ refers to the *recommendation time* in round t (running time of the recommendation tool) and $C_{cre}(s_{t-1}, s_t)$ refers to the incremental *PDS creation time* in transitioning from configuration s_{t-1} to s_t. Finally, $C_{exc}(w_t, s_t)$ denotes the *execution time* of mini-workload w_t under the configuration s_t (sum of response times of individual queries). Each s_t should fit in the memory budget M and can include any PDS type that is considered by the PDS tuning tool.

At each round t, the system chooses a configuration s_t in preparation for the mini-workload w_t. The system does not have any idea about w_t and therefore, the selection of s_t completely depends on the historical observations $(w_1, \ldots, w_{t-1})$ and respective actions $(s_1, \ldots, s_{t-1})$. The recommendation system performs minimum required actions to change the configuration from s_{t-1} to s_t. That means addition of a new set of PDSs in the set difference $s_t \setminus s_{t-1}$ and dropping PDSs in the set difference $s_{t-1} \setminus s_t$. Thereafter, new workload w_t will be executed and the query plans including execution statistics will be collected.

3 BANDIT BACKGROUND

In this work, we propose using contextual combinatorial bandits for online PDS tuning. This section presents relevant MAB background that will be useful when understanding this paper. We discuss important properties of contextual combinatorial bandits, such as context and combinatorial arms, which allows the bandit to converge to performant configurations quickly.

Notation: We use boldface for non-scalar values, vectors use lowercase letters, matrices use uppercase and transpose of a vector or matrix is denoted with a prime.

3.1 A Simple Bandit Setting

The basic stochastic bandit setting [12] repeatedly selects from k potential actions, over rounds $t = 1, 2, \ldots$, in each of which the MAB:

(1) Selects or *pulls* one action or *arm* $i_t \in [k]$; where $[k] = \{1, 2, \ldots, k\}$ for $k \in \mathbb{N}$ and
(2) Observes a random *reward* $R_t(i_t)$ independently drawn from fixed but unknown reward distribution.

A MAB's goal is to maximise cumulative expected rewards. The MAB *only observes rewards on pulled arms*. Thus, the MAB must balance *exploration* of under-observed arms to increase prospects of long-term rewards, with *exploitation* of knowledge gained so far for immediate rewards. To account for the cost of exploration, performance is measured relative to a best fixed policy:

Definition 1. Cumulative regret *to time T, is defined as the difference between total expected rewards under the best fixed-arm policy and the MAB's cumulative expected reward, $\mu^\star T - \sum_{t=1}^{T} \mathbb{E}[R_t(i_t)]$, where $\mu^\star = \max_{i \in [k]} \mathbb{E}[R(i)]$.*

3.2 Contextual Combinatorial Bandit Setting

Practically, a viable bandit formulation of online physical database design tuning problem requires some additional features, which are only offered in the contextual combinatorial bandit variant. This advanced variant makes some noticeable changes to the classic setting:

(1) MAB gets additional side information (context) with each action or *arm* $i \in [k]$, denoted as $\boldsymbol{X}_t = \{\boldsymbol{x}_t(i)\}_{i\in[k]}$, for $\boldsymbol{x}_t(i) \in \mathbb{R}^d$, along with their costs, c_i;
(2) MAB can pull a set of arms (as opposed to a single arm), which is termed as the *super arm* $s_t \in \mathcal{S}_t$, where we restrict the class of possible super arms using the total cost C. $\mathcal{S}_t = \{s \subseteq [k] \mid \sum_{i\in s} c_i \leq C\} \subseteq 2^{[k]}$; and
(3) Rather than receiving a single reward for the super arm, for each arm i_t in the super arm, MAB observes a *score* $r_t(i_t)$ (known as a *semi-bandit* feedback). $r_t(i_t)$ is drawn from fixed but unknown reward distribution that solely depends on the arm i_t and its context $\boldsymbol{x}_t(i_t)$ (as opposed to depending only on the arm).

Note that still the bandit can only observe scores for arms included in the super arm, requiring delicate balance between exploration and exploitation.

3.3 An Implementation

MAB needs to maximise the cumulative expected reward $\sum_t \mathbb{E}[R_t(s_t)] = \sum_t g(\boldsymbol{r}_t^\star, \boldsymbol{X}_t, s_t)$ for a known function g, which is based on each arm's true expected scores, $\boldsymbol{r}_t^\star$. It might seem like the only way to estimate arms scores is by including them in the super arm. This is practically impossible with a large number of arms. A solution for this is provided in the Contextual Combinatorial Upper Confidence Bound (C^{2}UCB) algorithm [61], an implementation of a contextual combinatorial bandit (see Algorithm 1).

Sweep away the need to explore each arm: Context in C^{2}UCB bandit allows the learner to generalise the learned knowledge to unseen arms by modelling arm rewards as linearly dependent on their context. C^{2}UCB estimates $\mathbb{E}[r_t(i)|\mathbf{x}_t(i)]$ with $\boldsymbol{\theta}'\boldsymbol{x}_t(i) + \varepsilon_t(i)$ for unknown zero-mean (subgaussian) random variable ε_t where $\boldsymbol{\theta} \in \mathbb{R}^d$ are trained coefficients of a ridge regression on arm i's observed rewards against contexts. It is important to notice that, all learned knowledge is encapsulated in $\boldsymbol{\theta}$, which is shared between all arms. Simply knowing the arm context $\mathbf{x}_t(i)$ would allow us to compute the estimate of its score removing the need to explore each arm. The estimation accuracy of an arm's score will depend on exploration of the arm's context dimensions. Therefore, bandit exploration should now target unexplored regions of the context rather than unexplored arms. The number of dimensions in the context is orders of magnitude smaller than the number of arms.

Optimism in the face of uncertainty: C^{2}UCB uses an exploration boost to the arms where it is less certain about the score (i.e., arms with unexplored context dimensions). As the name suggests, C^{2}UCB uses the upper confidence bound (UCB) [45] as shown below, to balance the exploration and exploitation.

$$\hat{r}_t(i) \quad = \quad \hat{\boldsymbol{\theta}}_t'\boldsymbol{x}_t(i) + \alpha_t\sqrt{\boldsymbol{x}_t(i)'\boldsymbol{V}_{t-1}^{-1}\boldsymbol{x}_t(i)}\ , \tag{1}$$

Algorithm 1 The C^{2}UCB Algorithm

Input: $\lambda, \alpha_1, \ldots, \alpha_T$
Initialize $\boldsymbol{V}_0 \leftarrow \lambda\boldsymbol{I}_d$, $\boldsymbol{b}_0 \leftarrow \mathbf{0}_d$
for $t \leftarrow 1, \ldots, T$ **do**
  Observe $\mathcal{S}_t$
  $\hat{\boldsymbol{\theta}}_t \leftarrow \boldsymbol{V}_{t-1}^{-1}\boldsymbol{b}_{t-1}$ ▷ estimate via ridge regression
  for $i \in [k]$ **do**
    Observe context $\boldsymbol{x}_t(i)$
    $\hat{r}_t(i) \leftarrow \hat{\boldsymbol{\theta}}_t'\boldsymbol{x}_t(i) + \alpha_t\sqrt{\boldsymbol{x}_t(i)'\boldsymbol{V}_{t-1}^{-1}\boldsymbol{x}_t(i)}$
  end for
  $s_t \leftarrow \mathcal{A}(\hat{\boldsymbol{r}}_t, \boldsymbol{X}_t)$ ▷ using α-approximation oracle
  Play s_t and observe $r_t(i)$ for all $i \in s_t$
  $\boldsymbol{V}_t \leftarrow \boldsymbol{V}_{t-1} + \sum_{i\in s_t}\boldsymbol{x}_t(i)\boldsymbol{x}_t(i)'$ ▷ regression update
  $\boldsymbol{b}_t \leftarrow \boldsymbol{b}_{t-1} + \sum_{i\in s_t} r_t(i)\boldsymbol{x}_t(i)$ ▷ regression update
end for

First term (i.e., $\hat{\boldsymbol{\theta}}_t'\boldsymbol{x}_t(i)$) of this equation is the point estimate of the arm score, whereas the second term (i.e., $\alpha_t\sqrt{\ldots}$) denotes the exploration boost. Here $\alpha_t > 0$ is the exploration boost factor, and $\boldsymbol{V}_{t-1}$ is the positive-definite $d \times d$ scatter matrix of contexts for the chosen arms up to and including round $t-1$. The value of the second term is larger when the context includes unexplored context dimensions and will be explored by the bandit.

The approximate oracle: Once we have the estimated score for each arm, the next task is to pick a set of arms (i.e., super arm $s_t \in \mathcal{S}_t$) which maximise $g(\hat{\boldsymbol{r}}_t, \boldsymbol{X}_t, s_t)$ keeping within the memory budget. This resembles the well known 0-1 knapsack problem. We settle for a near-optimal solution with an approximate algorithm (known as an *α-approximation oracle*), since the optimal solution for index and view selection is considered NP-hard [17, 28].

Definition 2. *[59] An α-approximation oracle is an algorithm $\mathcal{A}$ that outputs a super arm $\bar{s} = \mathcal{A}(\boldsymbol{r}, \boldsymbol{X})$ with guarantees $g(\bar{s}, \boldsymbol{r}, \boldsymbol{X}) \geq \alpha \cdot \max_s g(s, \boldsymbol{r}, \boldsymbol{X})$, for some $\alpha \in [0, 1]$ and given input $\boldsymbol{r}$ and $\boldsymbol{X}$.*

Approximation errors due to the oracle limit the ability of a bandit to learn as measured by regret. Therefore, in the C^{2}UCB setting, α-regret is defined to focus only on performance due to the bandit learner—cost due only to the oracle is not counted.

Definition 3. *[59] Cumulative α-regret is the sum of expected instantaneous regret, $Reg_t^\alpha = \alpha \cdot \max_s g(s, \boldsymbol{r}_t^\star, \boldsymbol{X}_t) - g(\bar{s}_t, \boldsymbol{r}_t^\star, \boldsymbol{X}_t)$, where $\bar{s}$ is a super arm returned by an α-approximation oracle as a part of the bandit algorithm.*

When the reward function g is chosen to be monotonic and Lipschitz continuous, C^{2}UCB enjoys $\tilde{O}(\sqrt{T})$ α-regret [56, 61]. The sub-linear regret bound implies that a per-round average cumulative regret approaches zero after sufficiently many rounds. The latter provides us with a much needed *safety guarantee* on worst-case performance, which is critical in production systems.

This regret bound builds on the several assumptions we made along the way. While these assumptions are satisfied in our design, the linearity assumption of rewards on the context requires more explanation. While a context that perfectly represents the environment and the arm would meet the assumption, it can lead to a lengthy context. Such a context would provide an optimal yet slow convergence. Fortunately, bandits are robust to environmental

noise in practice, allowing flexible context design even if it deviates from the linearity assumption. Therefore, a trade-off can be made to have a well-designed smaller context that results in more knowledge sharing and faster convergence.

4 DESIGN AND IMPLEMENTATION

Bandits have been successfully employed in the context of index tuning before [59]. The previous attempt has addressed some of the initial challenges for MAB-based physical design tuning. Amongst them, the design of the reward, context, and oracle, are vital. Despite the promising preliminary progress in the narrowed scope of index tuning, practical physical design tuning must address several additional challenges. a) Ability to tune several PDS holistically and b) Ability to handle exploration over a vast action space efficiently.

It would be relatively straightforward to design bandits that focus on a smaller scope (e.g. tuning indices for one table only) and possibly combine the results later, similar to an iterative solution [6, 62]. Previous work in physical database design tuning however discusses the importance of holistically searching through the combined space of PDS to achieve optimal results [1, 78]. We introduce a new bandit architecture that uses a hierarchy of bandits and optimiser knowledge to overcome these challenges. Our architecture allows for a straightforward and customised design for individual PDS, while considering all PDS holistically.

4.1 An Overview

The hierarchical architecture comprises two layers of bandits (see Figure 1). Encouraged by successful uses of candidate selection to reduce the search space [1, 15], the first layer bandits (L1) select the candidates for the responsible scope. All the candidates selected by L1 bandits form the arms for the second layer (L2) bandit, which searches through the combined action space. Finally, the L2 bandit learns from all the observations and feeds the relevant rewards to the L1 bandits.

One of the crucial properties of this architecture is that we can easily add and remove bandits in L1 without any disruption to the L2 bandit and its already learned knowledge. This property allows our system to change the action space at any moment. For example, moving from the combined space of indices and views to an index-only space can be done without any disruption. In those instances, we can save the current state of the bandits before shutting them down. The saved state can be leveraged to warm-start the bandits if we want to (re)create them in the future [57]. Furthermore, this architecture facilitates parallel execution of bandits in L1, which can be crucial in multi-core and distributed processing environments.

4.2 Design of L1 and L2 Bandits

Bandits in L1 are divided into two main sets (referred to as *clusters*). Cluster 1 bandits tune views for the whole database, whereas cluster 2 bandits tune indices for each table. Although we used 2 clusters, any clustering of L1 bandits is valid. As explained later, clustering will allow knowledge sharing between bandits. Each bandit picks a set of arms that fits inside the memory budget.

4.2.1 Arm Generation. Naïve arm generation based solely on the database schema can quickly lead to an explosion in the number of arms. Thanks to the natural skewness in the workloads, some unused column subsets can be ignored entirely [2]. Workload-based arm generation can focus on smaller subsets of columns, which are present in the workload at a time. Workload-based dynamic arm addition is only possible because our bandit setting permits the definition of arms at the start of each round.

Index arms are generated based on the combinations and permutations of query predicates and payloads. The arm generation for views starts by identifying *frequent table subsets*. The frequent table subsets refer to table subsets that frequently occur in queries while noticeably contributing to the total workload time. This computation follows [1], and depends on two crucial measurements: 1) *subset value:* total execution time of queries where the table subset occurs and 2) *subset size:* sum of the total number of rows in the tables included in the table subset. After identifying frequent table subsets, we generate view arms for each query where frequent table subsets are present. Arms will be generated using the subset of predicates and payload belonging to the frequent table subset, with and without the 'GROUP BY' clause.

4.2.2 Context Design. Each bandit in HMAB can have a different context. Oppose to monolithic bandit design [59], any changes to the database schema will only impact a single bandit working in a narrowed scope. Furthermore, context design only needs to capture attributes in the narrowed scope. Oppositely, a monolithic context that includes attributes of all the physical design structures would require a sparse but high-dimensional representation, which would negatively impact the convergence and running time of the solution.[1] We use three different context designs across different bandits in HMAB (see Figure 1 for examples).

a) Context design for index tuning: Following the successful context design of [59] for index tuning, we use a similar two-part context with some modifications. The first part of the context encodes the index prefix and makes a reward distribution biased towards the columns that come earlier in the order. Accordingly, there is a context feature representing each column in the table with a value 10^{-j} where j is the corresponding column's position in the index (if a column is included in the index, 0 otherwise). The second part of the context includes three context features: 1) a Boolean that marks high-value arms,[2] 2) a Boolean indicating a covering index and 3) *size and existence feature*. The *size and existence feature* of a PDS provides information about the PDS' existence as well as its size. It contains the estimated size of the PDS as a fraction of database size if not materialised already, 0 otherwise.

b) Context design for view tuning: The context for view bandit has three parts. The first part is similar to the index tuning bandit with two crucial differences. First, as this bandit works in the scope of the entire database, we have context features representing each column in the database. Second, unlike index tuning context, we use a simple one-hot representation. Columns included in a view do not have a clear order unless the view query specifically forces an order, which is optional. The second part of the context comprises features representing each table in the database and uses one-hot encoding to mark the tables included in the view. In the third part of

[1] C^{2}UCB has $O(t(d^3 + kd + h))$ time complexity, where h is the complexity of the oracle [61]. Note that the complexity heavily depends on the size of the context d.
[2] arms that can avoid expensive table scans that are worth at least 10% of the total execution time

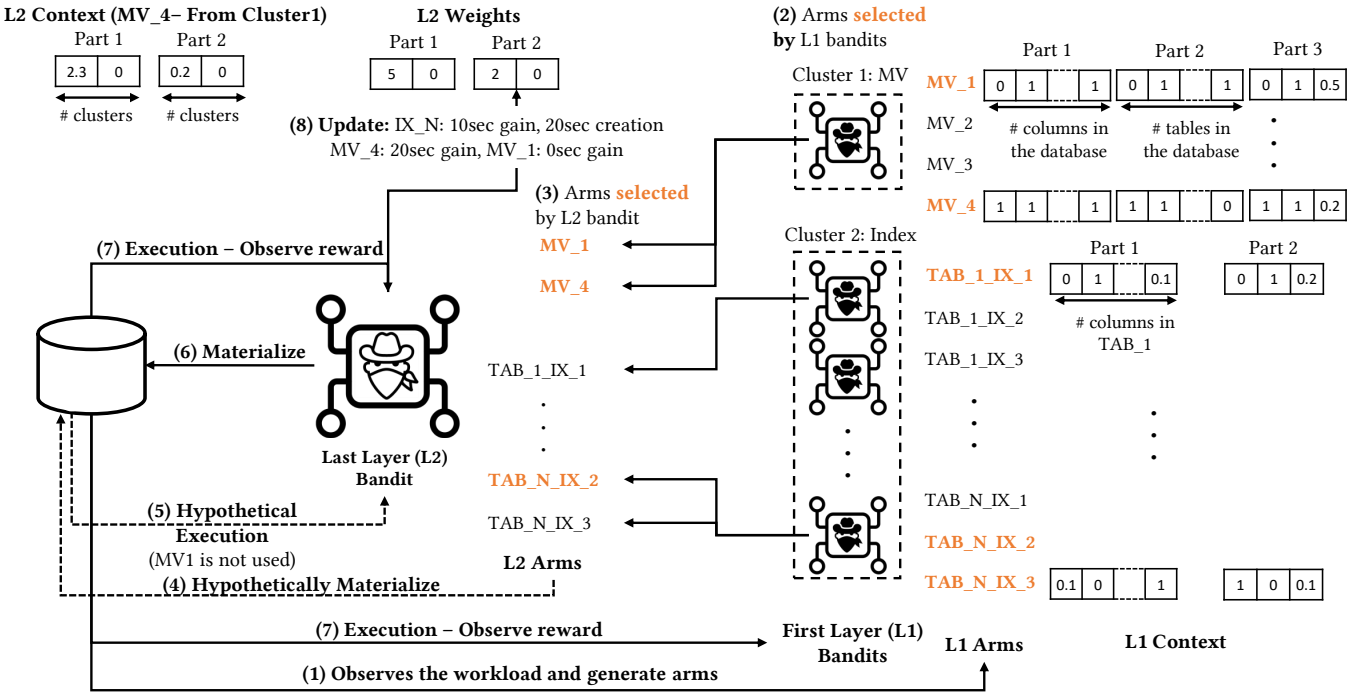


Figure 1: An example overview: index and view recommendation with HMAB.

the context, we have three context features: 1) a Boolean indicating if the view query has filtrations (a *WHERE* clause), 2) a Boolean indicating if the view query has grouped aggregations (a *GROUP BY* clause), and 3) *the size and existence feature.*

Changes to the schema could invalidate the learned knowledge from the responsible table bandit and view bandit. Nevertheless, this design substantially improves over the state-of-the-art bandit-based solution [59], which invalidates all learned knowledge on schema change. In practice, this issue could further be alleviated by leveraging warm-start techniques [57].

c) Context design for L2 bandit: L2 bandit context is divided into two main parts, each with a length equal to the number of L1 clusters. The context will only have values in the features relevant to the origin cluster (since all the L2 arms originate from L1 bandits) and 0 otherwise. The first part will hold the expected gain for that arm as predicted by the L1 bandit, and the second part contains the *the size and existence feature.* Having more clusters can provide more accurate convergence, whereas fewer clusters provide faster convergence through better knowledge sharing.

4.2.3 Design of the Oracle. In C^{2}UCB, the oracle's goal is to select a super arm based on the scores of the individual arms. Submodular (diminishing returns) super-arm value functions represent a well-motivated class with greedy oracles with $\alpha = 1 - 1/e$ that are efficient and near-optimal [55]. We use a greedy oracle coupled with filtration steps to encourage diversity. The super arm selection process happens in multiple iterations. In each iteration, we greedily select the arm with the maximum score from the available set of arms. Afterwards, we filter similar arms based on prefix matching along with arms that are not viable under the remaining memory budget. This process continues until we exhaust the memory budget or viable actions.

4.2.4 Design of the Reward. The reward design is essential to guide the bandit in the correct direction. Our implementation aims to minimise the total workload time, which is the sum of PDS creation time, query execution time, and PDS recommendation time. Recommendation time depends on our algorithm and does not depend on the bandit's actions. Therefore, we incorporated only creation time and query execution time in our reward. Nevertheless, we measure and report the recommendation time for each tool in our experiments.

We now define the score $r_t(i)$ for each arm i, based on creation time and execution time *gain*, in round t as below:

$$r_t(i) = G_t(i, w_t, s_t) - C_{cre}(s_{t-1}, \{i\}) \quad .$$

$C_{cre}(s_{t-1}, \{i\})$ is the creation cost of arm (PDS) i for configuration s_{t-1} and $G_t(i, w_t, s_t)$ is the execution time gain provided by each arm i for a workload w_t under configuration s_t.

The creation time is taken as a negative reward, only if i is materialised in round t, and 0 otherwise. It is relatively straightforward to capture the creation cost from execution statistics, however execution time gain from a PDS is scattered across multiple operators in the query plan. The most obvious one among them is the data scan operator. The gain from the data scan (G_t^{ds}) operations can be computed using the data scan times before and after PDS creation. However, there are some costs in the query plan which can be difficult to attribute to a single PDS, which we refer to as unclaimed gains (G_t^{un}). As shown below, we take the total execution time gain as a sum of these two types of gains. These gains are computed

only for PDS used by the optimiser and taken as 0 otherwise.

$$G_t(i, w_t, s_t) \quad = \quad G_t^{ds}(i, w_t, s_t) + G_t^{un}(i, w_t, s_t) \ .$$

Data scan gains: By defining $\mathcal{U}(s, q)$ as the list of PDS used by the query optimiser for data access in query q under a given configuration s, the data scan gain by arm i for query q is defined as:

$$G_t^{ds}(i, \{q\}, s_t) = \sum_{\tau \in \mathcal{B}} [C_{tab}(\tau, q, \emptyset) - C_{tab}(\tau, q, \{i\})] \, \mathbb{1}_{\mathcal{U}(s,q)}(i) \ .$$

Where $\mathcal{B}$ represents all the tables over which the PDS i is created. For indices $\mathcal{B}$ only includes the table for which an index belongs to, whereas for views $\mathcal{B}$ can contain multiple tables. $C_{tab}(\tau, q, \emptyset)$ represents the full table scan time for table τ in query q.[3] Notice that data gain can be negative if the use of PDS leads to a performance regression.

Unclaimed gains: The use of PDS can impact the query plan in very subtle ways which cannot be easily attributed to a single PDS. For example, introducing a new index can trigger the optimiser to choose a different query plan. Even when the index use provides a faster data scan, new query execution can take more time due to an inefficient nested loop join. Even though this issue arises from the optimiser, HMAB needs to synchronise with the optimiser and possibly take corrective actions to trigger a different query plan to improve execution time. These gains can be computed by comparing the query running times before and after index creation. We compute the total query gain (G_t^{to}) as:

$$G_t^{to}(\{q\}, s_t) = [C_{to}(q, \emptyset) - C_{to}(q, s_t)] \quad .$$

where $C_{to}(q, s_t)$ represents the total running time under configuration s_t. Once the data scan gain is calculated, we calculate the total unclaimed gains for a query by subtracting the data scan gain from the total query gain. Then, we equally divide this cost amongst participating PDS ($\mathcal{U}(s, q)$). The gain for a workload relates to the gain for individual query by:

$$G_t(i, w_t, s_t) = \sum_{q \in w_t} G_t(i, \{q\}, s_t) \, .$$

It can be shown that maximising individual arm rewards is equal to the original goal of minimising the total workload time C_{tot}. Using the execution time gain in the place of execution time and ignoring the recommendation time:

$$\begin{aligned} R_t(s_t) &= [C_{exc}(w_t, \emptyset) - C_{exc}(w_t, s_t)] - C_{cre}(s_{t-1}, s_t) \\ &\approx \sum_{i \in s_t} G_t(i, w_t, s_t) - \sum_{i \in s_t} C_{cre}(s_{t-1}, \{i\}) \\ &= \sum_{i \in s_t} r_t(i) \quad . \end{aligned}$$

The reward includes creation cost and execution cost gain guiding the HMAB to optimise for efficiency and recommendation quality.

[3]The bandit only identifies the arms for a query after its first appearance in the workload. Therefore, naturally, when a query is observed for the first time, the system does not have a supporting PDS for the query, and we observe a full table scan time for each table. When we do not observe full table scan time, we estimate it with the maximum secondary index scan/seek time.

Algorithm 2 Hierarchy of Bandits for PDS Tuning

```
QS ← QueryStore()                                   ▷ keeps query information
L2Bandit ← getL2Bandit()                            ▷ A1, L 1-2
while (TRUE) do
    config ← readDynamicConfig()
    queries ← getLastRoundWorkload()
    for all queries do
        if (isNewTemplate) then
            QS.add(query)
        else
            QS.update(query)
        end if
    end for
    QoI ← QS.getQoI()                               ▷ get queries of interest
    L1Bandits[] ← getL1Bandits(config, QoI)         ▷ A1, L 1-2
    L1arms[] ← generateL1Arms(QoI)
    L1X[] ← generateL1Context(L1arms, QoI)
    for all L1Bandits do
        rec_t[bId] ← bandit.recommend(arms[bId], X[bId])
    end for
    L2arms ← generateL2Arms(rec_t)
    L2X ← generateL2Context(L2arms, L1X)
    s_t ← L2Bandit.recommend(L2arms, L2X)           ▷ A1, L 4-10
    s_t ← hypCheck(s_t)
    C_cre ← materialise(s_t)
    C_exc ← executeCurrentWorkload()
    L2Bandit.updateWeights(C_cre, C_exc)            ▷ A1, L 12-13
    for all L1Bandits do
        bandit.updateWeights(C_cre, C_exc)          ▷ A1, L 12-13
    end for
end while
```

4.3 Hypothetical Checks: A Minimal Use of the Optimiser to Reduce Exploration Cost

Previous efforts using bandits for index tuning [59], which depends on observations from actual PDS materialisations, suffered from high creation costs while exploring. This is compounded with views. While it is imperative to learn from the actual PDS materialisations, we cannot depend on them entirely, due to the high creation cost.

Hypothetical checks are a simple yet effective solution to drastically reduce creation cost with minimal use of optimiser knowledge. A hypothetical check happens between the arm selection and materialisation, under a what-if analysis based on the optimiser cost model. While the optimiser cost model can be wrong in ranking the best index for a query, it has the last say in what indices are to be used in query execution. Therefore, it is necessary to be in sync with the optimiser recommendations. In our bandit setting, we use hypothetical checks to find the arms that will not be picked and prune them (i.e., give 0 reward) without ever materialising them.

4.4 Putting it all together

Consider a scenario where HMAB's target is to recommend a set of indices and views (see Figure 1). First, the system observes the workload and generates bandit arms for L1 bandits (step 1). For instance, the view bandit in cluster 1 has four arms generated. Notice that

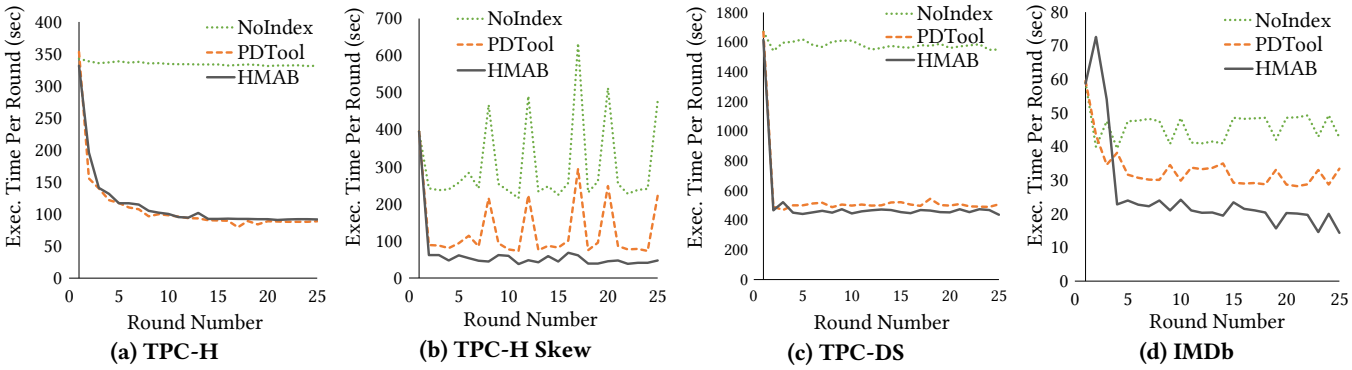


Figure 2: HMAB vs. PDTool, execution time convergence for integrated *view and index* tuning under *static* workloads.

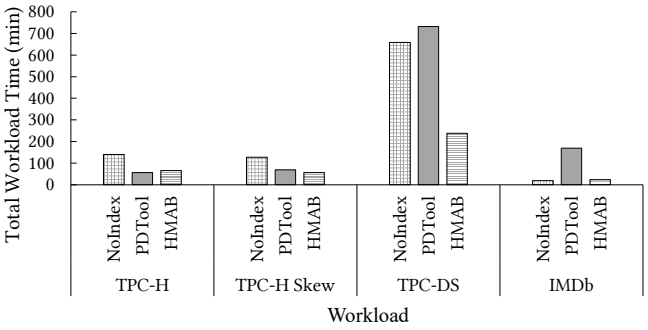


Figure 3: HMAB vs. PDTool, end-to-end total workload time for *view and index* tuning under *static* workloads.

each arm is accompanied by a context that will provide additional information to the learner. Next, each L1 bandit selects the arms for the L2 bandit (step 2: selected arms are coloured orange). After that, the L2 bandit selects arms (step 3) that will be checked using the what-if interface (steps 4 and 5). In this example, the learner understands that the optimiser does not use $MV1$ and the rest of selected PDS ($MV4$, TAB_N_IX2) will be materialised in the database (step 6). Then, both L1 and L2 bandits will observe the actual workload execution to understand the actual gain/reward by the created PDS (step 7). Finally, the calculated reward will update all the bandits and learn the weight vectors (step 8).[4] The bandit can choose to forget the learned knowledge based on the workload shift intensity (i.e., the number of newly introduced query templates) [59].

4.5 Implementation

Algorithm 2 outlines the implementation of our C^{2}UCB [61] bandit system. First, we use a query store to track the queries we observe and their properties (last seen time, first seen time, frequency, table level selectivity, running times). The query store is later used to generate the *queries of interest* (QoI). While QoI can consist of all queries the learner has observed, the filtration step allows selection of a subset of queries for arm generation. For example, queries that were not observed for many rounds, or fast queries that contribute negligibly to the total workload time may be pruned.

[4]In this step, we leverage a 'focus update' introduced in [60].

While the L2 bandit can be initialised at the beginning, the L1 bandit initialisation depends on the current workload and current configurations (i.e., what PDS types to tune). Bandits will be initialised only when not already available; otherwise, the old instance of the bandit will be used. After setting up, each L1 bandit will be called for recommendations along with their estimated scores. The L1 recommendations and estimated scores help generate arms and context for the L2 bandit, which makes final recommendations (i.e., super arm). Note that the bandit initialisation, arm recommendation, and weight update, call the functions from the C^{2}UCB Algorithm 1.

5 EXPERIMENTAL RESULTS

In this experimental section, we demonstrate the ability of our tool to outperform state-of-the-art physical design tuning systems for index and materialised view selection. There are not many PDS tuning tools that can work in an integrated search space of multiple physical design structures, and none of them is learning-based. Therefore, we compare against a state-of-the-art commercial physical design tool that works in the integrated search space, which we refer to as the Physical Design Tool (PDTool). Recent studies have shown that the PDTool outperforms other tuning tools available on the market [7, 36]. Furthermore, we test against nine index selection algorithms covering a wide range of approaches.

5.1 Experimental Setup

We test against four publicly available benchmarks. Amongst them, there are three industrial benchmarks (a) TPC-H (uniform) [68]: A widely used decision support benchmark. (b) TPC-H Skew [52] with Zipfian factor 4: TPC-H with skewed data distribution, allowing the readers to understand the impact of data skewness when all the other aspects are kept identical. (c) TPC-DS [54]: A complex and modern benchmark commonly used as the gold standard. Other than these three industrial benchmarks, we use well known real-world IMDb dataset with Join Order Benchmark (JOB) [44]. A massive action space generated by a high number of joins in JOB queries makes it extremely challenging for view selection. All the experiments are performed over the combined action space of indices and views, on 10GB (i.e., SF10) databases, with a memory budget approximately equal to the size of the data unless mentioned otherwise.[5] We experiment using two workload types:

[5]The IMDb dataset has fixed size of 6GB.

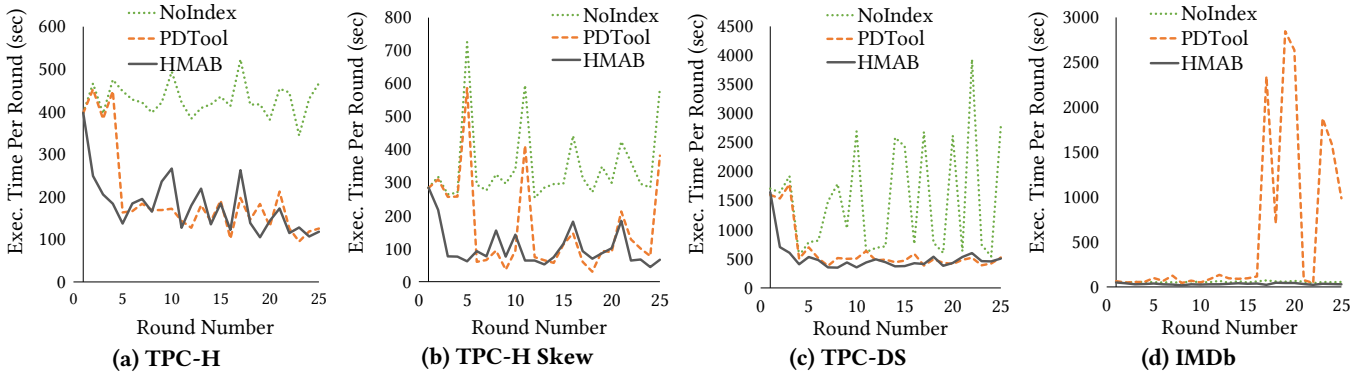


Figure 4: HMAB vs. PDTool, execution time convergence for integrated *view and index* tuning under *dynamic random* workloads.

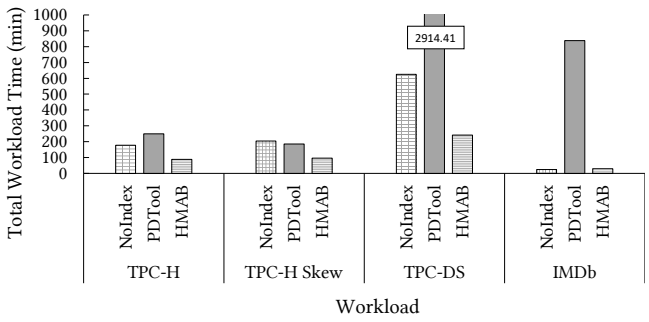


Figure 5: HMAB vs. PDTool, end-to-end total workload time for *view and index* tuning under *dynamic random* workloads.

Static: A repeating workload similar to what is typically found in reporting applications. In the absence of dynamic environment complexities, this setting is used to apprehend the accuracy and overhead of the bandit search strategy. The static workload runs for 25 rounds giving sufficient time to observe the convergence. In each round, a different query instance for every query template will be executed (i.e., 22, 99 and 33 templates for TPC-H, TPC-DS and IMDb, respectively). PDTool is invoked once per experiment at the start of round two with round one workload as input.

Dynamic random: A natural dynamic query sequence comprises new and returning queries mimicking modern ad-hoc workloads (such as cloud services). While tuning tools need to adapt to the new queries, reacting too soon can result in unwanted PDS oscillations. A tool's effectiveness depends on the delicate balance between swift and careful adaptation. The number of queries in the dynamic random setting is similar to the static setting, but queries are chosen randomly and contain around 45-55% of repeating query templates (with different query instances). Identifying the representative workload for an offline tuning tool is almost impossible when the workload is dynamic. Therefore, it is a common practice to periodically (e.g., nightly or weekly) invoke the PDTool with all the queries observed from the last invocation. In our setup, we invoke PDTool every four rounds, providing the queries from the last four rounds as the representative workload.

Hardware: All experiments are conducted on an industrial-grade server running Windows Server 2016 equipped with two 24 Core Xeon Platinum 8260 processors, 1.1TB RAM, and 50TB disk (10K RPM). We report cold runs, clearing the database buffer caches before each query execution.

5.2 Index and View Recommendation

In addition to comparing our approach against the state-of-the-art commercial tuning tool, we use a baseline without any indices (NoIndex). With the NoIndex baseline, we compare the execution time gain and recommendation time loss against an untuned system.

Static: Figure 3 summarises HMAB performance under all benchmarks comparing the total workload time of all three baselines. Figure 2 (a-d) shows the convergence of execution time for each benchmark. The convergence graph explains the effectiveness of the recommended configurations. The exact total workload times and component-wise breakdown (across *recommendation time*, *creation time* and *execution time*) are reported in Table 1.

HMAB provides significant gains of 19%, 67% and 86% for TPC-H Skew, TPC-DS, and IMDb benchmarks (see Figure 3). However, under TPC-H, PDTool provides a 13% better total workload time. This result is anticipated since the TPC-H benchmark provides a uniform dataset, perfectly matching the offline tuning tool's assumptions. Under TPC-H, as shown in Figure 2 (a), both tools perform equally well and recommend configurations that converge to similar execution times with HMAB slightly behind (PDTool provides 3% better total execution time). PDTool's gain in the total workload time under TPC-H is mainly due to the higher creation costs of HMAB, which is expected from a system that forms decisions on actual PDS materialisations and execution statistics. The bandit learns better configurations with skewed datasets (TPC-H Skew, TPC-DS and IMDB), justifying the creation time overhead (see Figure 2 (b-d)). Based on last round execution time gains, 20%, 12%, 1.2% and 0.8% of the improvement for HMAB under TPC-H, TPC-H skew, TPC-DS and IMDb benchmarks, respectively, come from views, whereas the rest comes from indices. Views tend to provide substantial support for very few queries, whereas indices support a wider range of queries albeit to a smaller extent. Creating views that target a few templates is thus of limited help when we have a large number of query templates (such as in TPC-DS, IMDb).[6] Other than that, in

[6]On the contrary, in a test done with the SSB benchmark, over 10GB dataset and static workload with only 13 templates, 76% of the execution time gain is provided by views.

Table 1: HMAB vs. PDTool total workload time breakdown (min): the best option is highlighted in blue.

Workload		Recommendation		Creation		Execution		Total		Exploration**	
		PDTool	MAB	PDTool	MAB	PDTool	MAB	PDTool	MAB	PDTool	MAB
Static	TPC-H	2.12	1.28	8.58	15.97	45.66	47.54	56.36	64.78	10.7	17.25
	TPC-H Skew	2.12	1.74	13.15	27.16	53.82	26.64	69.08	55.54	15.27	28.9
	TPC-DS	499.54	5.99	4.01	20.89	228.72	211.4	732.27	238.29	503.55	26.88
	IMDb	154.15	9.32	1.25	3.41	13.81	10.64	169.21	23.37	155.4	12.73
Random	TPC-H	139.77	1.38	27.79	12.99	82.09	74.63	249.66	89	167.56	14.37
	TPC-H Skew	80.08	1.35	37.76	51.77	68.18	43.16	186.02	96.27	117.84	53.12
	TPC-DS	2639.14	17.25	14.6	10.68	260.68	214.43	2914.41	242.36	2653.74	27.93
	IMDb	594.37	11.5	2.76	2.88	240.36	14.22	837.49	28.6	597.13	14.38

**Exploration time = Creation time + Recommendation time

TPC-DS, we have one index responsible for more than 85% of the gain, compared to which the view contribution looks smaller. Other than the gain from execution time, HMAB's lightweight execution in TPC-DS and IMDb provides noticeable gains in recommendation time compared to costly PDTool invocations (see Table 1). Large recommendation times for PDTool are attributed to a large number of view candidates under IMDb and TPC-DS benchmarks, which HMAB swiftly handles (e.g., HMAB works with 11700 and 500 view-arms for IMDb and TPC-DS, respectively)

While the TPC-H line appears steady, it is possible to observe slight variations in execution times for other benchmarks. While we are repeating the same set of templates in each round, it is essential to note that we use different instances. Consequently, different instances of the same template can have different execution times as the data is skewed. For example, we see some instances of IMDb Q20 taking five times longer than other instances, creating the pattern we see in the IMDb convergence graph (Figure 2 (d)). Other than these minor variations, we can observe a few big spikes in rounds 8, 12, 17, 20 and 25 for PDTool under the TPC-H skew benchmark (see Figure 2 (b)). These spikes are due to PDTool missing critical index *Orders.O_custkey* which supports some instances of Q22. Similar TPC-H Skew spikes can be observed in other experiments as well.

Dynamic random: Figure 5 provides a summary of HMAB performance under dynamic random workloads. HMAB outperforms PDTool under all benchmarks, providing 64%, 48%, 91% and 96% gains under TPC-H, TPC-H Skew, TPC-DS and IMDb benchmarks, respectively. The percentage execution time gain from views has gone down to 19%, 8%, 4% and 0% for TPC-H, TPC-H skew, TPC-DS and IMDb benchmarks. Higher creation costs in dynamic environments favour indices with a low d footprint. Furthermore, as observable from Figure 4 (a-d) due to the random selection of queries, we do not see a steady line in convergence graphs as we did in static settings, even for TPC-H. In addition, there are interesting spikes in the IMDb graph (Figure 4 (d)) that we need to explain.

The IMDb benchmark challenges optimisers with numerous joins, leading to sub-optimal PDS choices. Consequently, IMDb queries frequently run faster without any PDS. The spikes are caused by Q29 of IMDb, which occurs multiple times per round giving rise to spikes of different sizes (in round 17, there are three occurrences, whereas, in round 18, there is only one). IMDb Q29 is a seemly negligible query with 4.5 s no index running time. However, after the fourth invocation (round 16), PDTool creates an index on *title*, *production_year* columns in the *title* table. This index leads to a nested loop that runs for more than 10 mins, due to erroneous cardinality estimations. In such a situation, HMAB's approach is to drop the PDS that triggered the erroneous execution plan.

5.3 Index Selection

Due to the broad applicability of indices and their low memory footprint, index-only configurations are popular amongst DBAs. In this section, we showcase the capability of this new bandit framework to surpass PDTool, Microsoft's AutoAdmin [15], Relaxation [9] and DTA Anytime [14], IBM's DB2 Advisor [70], iterative Extend [64] algorithm, heuristic-based Drop [75] algorithm, PostgreSQL's Dexter [34] and reinforcement learning based UDO [74] and DBABandit [59]. We extended the implementations found in [34, 37, 73] to work with the commercial database system we use. We experiment against TPC-H, TPC-H Skew and TPC-DS benchmarks under static workloads. We have used the default configurations from these implementations for all the tests. We further experiment and discuss the impact of varying the main parameters.

As evident from results in Table 2, HMAB outperforms all the tools in total workload time under both TPC-H Skew and TPC-DS whereas, under TPC-H, PDTool performs the best but is closely followed by HMAB and Extend. Considering execution time alone, HMAB performs best in TPC-H Skew and TPC-DS whereas, under TPC-H, Relaxation performs better. As expected HMAB records high creation times. In recommendation time, HMAB is only second to DB2 Advisor, which provides lightning-fast recommendations without much loss of recommendation quality.

Index width: While the default configuration runs with an index width of 2 columns we tried a few other index widths (6 and 10) in search of the best configurations.[7] Higher index widths help to reduce the execution time while they greatly increase the recommendation time leading to a higher total workload time. However, in some cases, the overall gain was positive. We used the best timings across tested index widths for Table 2.

Other parameters: Dexter is tested with different minimal cost-savings percentages and we report the best results in the table (with 5%). For AutoAdmin and Drop the total number of indices was set to fill the memory budget.

Recommendation time: Externally invoking hypothetical cost requests take much longer compared to an internalised implementation. This is an unfair comparison for tools that externally use

[7]Dexter and Drop were not expandable beyond 2 columns.

Table 2: Total workload time (min) breakdown for index tuning under TPC-H, TPC-H Skew and TPC-DS static workloads: the best option is highlighted in blue.

	TPC-H				TPC-H Skew				TPC-DS			
	Rec.	**Cre.**	**Exec.**	**Total**	**Rec.**	**Cre.**	**Exec.**	**Total**	**Rec.**	**Cre.**	**Exec.**	**Total**
DBABandit	0.08	9.02	58.63	67.73	0.1	15.1	43.55	58.75	1.47	12.86	262.88	277.21
PDTool	0.88	8.07	46.79	55.74	0.89	12.59	55.32	68.8	16.39	3.8	277.22	297.41
HMAB	0.22	8.62	47.76	56.6	0.15	20.86	30.39	51.4	1.14	7.76	219.98	228.88
UDO	552	4.35	96.4	652.75	1113.38	14.39	44.54	1172.31	N/A	N/A	N/A	N/A
Anytime	5.68	5.66	45.34	56.68	7.33	8.88	35.36	51.57	39.88	7.29	308.47	355.64
AutoAdmin	1.94	4.61	80.4	86.95	2.73	11.91	44.32	58.96	28.99	4.94	273.87	307.8
DB2Advis	0.01	4.09	89.2	93.3	0.01	15.52	43.23	58.76	0.09	4.27	279.97	284.33
Dexter	0.05	1.37	103.46	104.88	4.7	5.97	53.78	64.45	9.22	1.86	674.06	685.14
Drop	0.32	5.33	88.75	94.4	0.32	13.82	75.41	89.55	56.35	0.34	694.39	751.08
Extend	1.09	3.16	52.35	56.6	0.52	8.91	81.51	90.94	9.49	3.41	702.73	715.63
Relaxation	25.91	3.37	43.23	72.51	8.75	15.01	35.67	59.43	567.39	4.3	365.38	937.07

Rec. = Recommendation time, Cre. = Creation time, Exec. = Execution time, N/A = Not available

the hypothetical index simulations. Therefore to create a level playing field, we discounted the recommendation times for all taking 12.7 ms for a cost request and 0.25 ms for index simulations based on the findings from [36]. For example, the recommendation time for DTA Anytime algorithm drops from 40.63 minutes to 3.44 minutes after discounting. While this discounting favours the tools that heavily rely on hypothetical indices, still HMAB performs better than all the other tools in TPC-H Skew and TPC-DS.

5.4 View Selection

Here we test HMAB against PDTool and three heuristic baselines from [29] (see Figure 6). These baselines greedily select the best views based on three different metrics collected using isolated hypothetical executions: a) TopValue: execution time gain (G^{to}), b) TopUValue: G^{to}/size, and c) TopFreq: view usage frequency. In TopFreq, ties are broken using G^{to}. These heuristic approaches do not consider the view interactions, thus leading to less performant view recommendations. The views created by PDTool and HMAB differ in their use of filters with specific predicates. While filtered views use less space, they are unusable with other predicate values. For example, a filter *PART.P_NAME like '%pink%'* gets used only three times, creating sudden drops in the convergence graph (rounds 8, 17 and 19). On the other hand, HMAB does not create any views with such filters allowing them to be widely used across multiple queries. HMAB converges to a 21.6% better configuration. However, due to high execution times in the early rounds and higher creation costs HMAB records an 8.8% longer total workload time.

5.5 Impact of Changing L1 bandits

One of the crucial features of HMAB is the ability to add and remove L1 bandits without any disruption to the learned knowledge of L2 bandit and other L1 bandits. In this experiment, we require the tuning tool to first tune indices and materialised views. After the 10th round, we change the requirement to tune only indices and then revert back to indices and views after round 20. PDTool is invoked at rounds 1, 11 and 21 to tune for respective PDS sets. We run the experiment for a total of 30 rounds. Figure 7 plots the

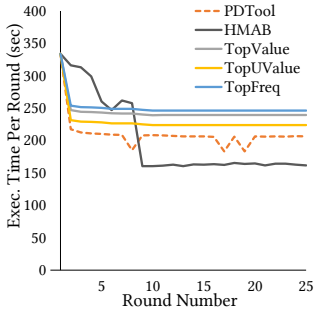


Figure 6: Execution time convergence for *view* tuning under *static* workloads.

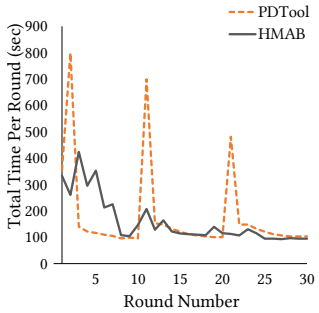


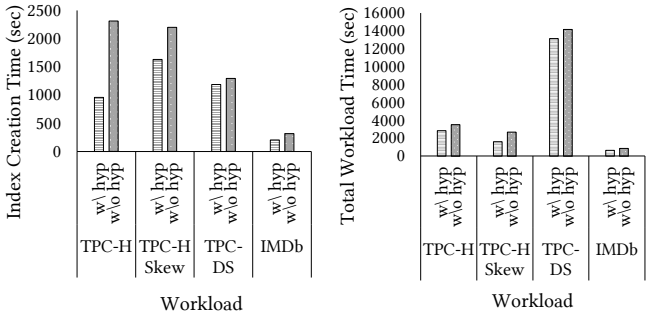
Figure 7: HMAB vs. PDTool total end-to-end workload time convergence for *static* workloads with changing PDS sets.

total workload time per round which showcases the overhead of changing PDS types as it includes the recommendation and creation times.

We can notice large spikes for PDTool in rounds 1, 11, and 21 due to recommendation times and index creation times. On the other hand, HMAB adapted to the changes smoothly. In rounds 11-13, there are some spikes in the case of HMAB in search of the best indices to fill up the space vacated by dropping views. At round 21, HMAB created only 2 views at the expense of several small indices. View creation took less than 5 seconds as both views had GROUP BY clauses. This demonstrates that the learning from the first 10 rounds (i.e., views with a group by clause provide better rewards with low space consumption), has been carried over without any disruption. HMAB provides a 10.4% improvement in total workload time over PDTool in this experiment.

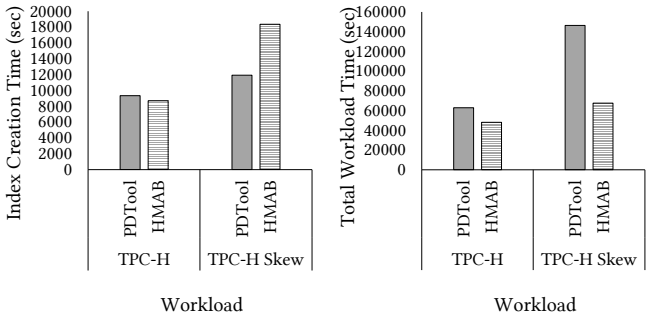
5.6 Impact of Hierarchical Bandit Structure on Recommendation Time

The recommendation time of HMAB contains two main components: (a) running time of the bandit (i.e., running time of the code ignoring the time taken for external dependencies such as index creation and query execution) (b) Time taken for hypothetical checks.



(a) Total Creation time **(b) Total workload time**

Figure 8: HMAB with hypothetical checks vs. without.



(a) Total creation time **(b) Total workload time**

Figure 9: HMAB vs. PDTool, view and index tuning under 100GB databases and TPC-H *static* workloads.

In this section, we focus on the impact of hierarchical bandit structure on the running time of the bandit. To understand the impact, we compare the running time of HMAB against DBABandit in a static index tuning setting. In the index tuning setting, the main difference between the two tools is that DBABandit uses one large bandit, whereas HMAB uses a hierarchy of small bandits. We use the TPC-DS benchmark for this experiment as it has the highest number of tables resulting in 28 Level 1 bandits for HMAB. Under the TPC-DS static setting, DBABandit records a 88 second running time, whereas HMAB runs in 29 seconds (66% drop). While this drop in recommendation time is negligible compared to the total workload time, it demonstrates the scalability of the new architecture. While running all the L1 bandits in parallel is possible; we did not do that in the current implementation. Allowing the bandits to run in parallel would enable even higher gains for HMAB, noting its suitability for modern multi-processor systems.

5.7 Impact of Hypothetical Checks

Hypothetical checks noticeably reduce creation time, improving the usability of the solution. This section compares the previous results from four benchmarks against the identical bandit runs that do not use hypothetical checks. The initial observation shows three effects of hypothetical checks. Hypothetical checks: 1) reduce the total creation time by removing unnecessary PDS creations, 2) reduce total execution time by providing faster convergence, and 3) increase the recommendation time attributed to 'what-if' calls.

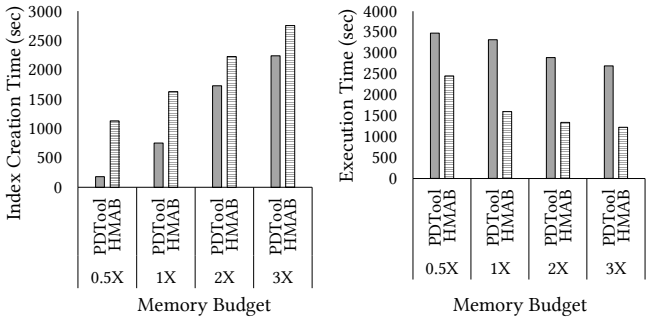
As shown in Figure 8 (a), hypothetical checks provide a 58%, 25%, 8% and 34% reduction in creation cost under TPC-H, TPC-H Skew, TPC-DS and IMDb benchmarks, respectively. At the same time, hypothetical checks add 0.8, 1.2, 6.0, and 6.8 minutes to the total workload time under the benchmarks mentioned above. The ultimate time saving from hypothetical checks can be identified by comparing the total workload time in Figure 8 (b), where bandits using hypothetical checks provide 33%, 32% and 5% gain under TPC-H, TPC-H Skew, TPC-DS benchmarks, and 5% loss under IMDb. A 6.8 minute addition to the total workload time is excessive compared to IMDb's short total workload time. It is important to note that HMAB issues a minimal number of cost requests ('what-if' calls). For example, PDTool considers around 1500 configurations and issues more than 1.5M cost requests under the TPC-DS static setting, where 8% of this is non-cached (around 130K requests) [36]. On the contrary, HMAB only performs 25 configurations checks (one per round) with around 2.5k cost requests for a 25 round HMAB run. Thus, the number of optimiser calls by HMAB is negligible compared to thousands of checks done by traditional tuning tools. The 'what-if' analysis done through an external interface provided by the commercial database however incurs a considerable recommendation time increase for HMAB. An internal implementation of the bandit system should reduce such recommendation costs.

5.8 HMAB's Scalability for Larger Databases

We repeat the TPC-H uniform and TPC-H Skew static experiment on 100 GB (SF 100) databases to answer two primary concerns: 1) practicality of materialisation-based search strategy for large databases, 2) HMAB performance under large databases (compared to SF 10 experiments). Overall, the results demonstrate the solution fitness for larger databases and the potential for big data era.

Comparing the creation time against PDTool, HMAB records 6% gain and 35% loss under TPC-H and TPC-H Skew, respectively (see Figure 9 (a)). As a percentage of total workload time, we notice a clear reduction in HMAB creation time from SF 10 to SF 100 experiments (TPC-H: 24% → 18%, TPC-H Skew: 48% → 27%). This reduction is due to the clear difference between arm scores in large databases: the bandit can converge faster with less exploration.

Comparing the total workload time, HMAB outperforms PDTool with 23% and 53% gains under TPC-H and TPC-H Skew, respectively (see Figure 9 (b)). In both cases, the gains are higher compared to SF 10 experiments. Under TPC-H skew, the impact of Q22 is much higher (long-running Q22 instances takes around 5 hours), demonstrating the catastrophic impact of sub-optimal PDS choices in large databases. The result is exciting in TPC-H as this is the first time we observe HMAB outperforming PDTool under TPC-H static workload. This gain (4.2 hours) comes from execution time, which cannot be assigned to a single query but is distributed across 14 queries. Drilling down, we observe that PDTool creates five memory heavy views leaving less than 20% of the space for indices. While views provide excellent support for a few queries (e.g., Q5, Q17, Q19, Q20), many others are not supported by any PDS (e.g., Q1, Q3, Q6, Q7, Q14). HMAB finds a different balance and uses 94% space for indices and the rest for views. This result portrays the importance of finding the right balance when tuning multiple PDSs.



(a) Total creation time **(b) Total execution time**

Figure 10: HMAB vs. PDTool, view and index tuning for different memory budgets with TPC-H Skew *static* workloads.

5.9 HMAB's Performance under Different Memory Budgets

This section presents experiments across four different memory budgets, 0.5X, 1X (original memory budget), 2X, 3X, under TPC-H skew benchmark, to understand the memory budget's impact on solution fitness. As shown in Figure 10 (a), HMAB continuously takes more creation time under all memory budgets. Higher creation times result from higher exploration in HMAB, which is required to understand the highly skewed datasets. We detect that more view exploration happens with larger memory budgets (e.g., 10 view arms are explored for 0.5x, whereas 15 view arms are explored for the 3x memory budget). For lower memory budgets, HMAB sticks to aggregated views with a low memory footprint (7 out of 10 selected view arms used GROUP BY), whereas with higher memory budgets, more views without aggregation are tried (only 6 out of 15 views used GROUP BY in the case of 3x setting). With higher memory budgets, HMAB can choose to preserve most of the PDS it builds, reducing waste. Consequently, the percentage difference between creation times from the two tools decreases with higher memory budgets. Despite higher PDS creation costs, HMAB always finds better configurations (see Figure 10 (b)). When considering the total workload time, HMAB ends up providing 2%, 19%, 22%, 19% gain, under 0.5X, 1X, 2X and 3X memory budgets, respectively.

5.10 Success of HMAB's Search Strategy

With the experimental results, it is clear that actual materialisation and execution helped HMAB converge to better configurations in terms of query execution time. However, this comes at the cost of the creation time, where HMAB records higher creation times in 6 out of 8 times (see Table 1). The above observation poses questions about the efficiency of the HMAB search strategy.

It is essential to note that PDTool focuses on exploration using the 'what-if' analysis, and HMAB uses a combination of the 'what-if' analysis and actual PDS creation. To truly understand how each of these techniques performs, we need to compare the sum of recommendation time and creation time, which we will refer to as the *exploration time*. As observable in Table 1, 6 out of 8 times HMAB records a better exploration time. It falls short only under the TPC-H and TPC-H Skew static settings, where PDTool records short recommendation times due to the low number of queries.

6 RELATED WORK

PDS search space. The PDS tuning problem has been studied for years [36, 47]. However, most tools work with a single design structure. Index tuning has been the entry point for many PDS tuning tools due to indices' simpler and effective nature. There are many traditional and learned solutions for index tuning [5, 10, 11, 20, 21, 29, 30, 41, 59, 63, 65, 66, 76]. Similarly, some efforts work in the area of view tuning [4, 16, 24, 27, 39, 40, 42, 43, 53, 71, 72]. Nevertheless, there are only a few tools out there that can work in the combined space of indices and views [1, 18, 78]. To the best of our knowledge, there are not any learned tuning tools that work in the combined action space of views and indices.

Fixing the optimiser misestimates using execution statistics. All state-of-the-art physical design solutions are based on the query optimiser cost estimates [13]. Such estimates are severely misleading for non-uniform data distributions and under complex workloads [21]. Recent efforts attempted to leverage execution statistics to alleviate these mistakes [19, 21]. [21] used regression to mitigate the optimiser misestimates for the index tuning. This effort however focused only on the index tuning and has not incorporated views. While presenting highly promising results by avoiding cost misestimates, the solution adds up to 10% recommendation time.

Learning based solutions in other areas of databases. Reinforcement learning has been used in many areas of database systems, including join ordering, query optimisation and configuration tuning [33, 35, 51, 58, 69]. Furthermore, there are bandit solutions used in the areas of monitoring, query optimisation and join ordering [23, 26, 48]. Hierarchical contextual bandit architectures have been used for performance prediction in cloud databases [49]. There are modern, adaptive data structures [25, 31] as well as learned models [22, 32, 38] that can replace the traditional data structures. These efforts complement our efforts, and HMAB can be extended to work with those index structures.

Use of workload forecasting to avoid cost start problem Workload forecasting [46, 50, 77] is an exciting research area that can complement both traditional and learned PDS tuning approaches. Forecasted workloads can be used to identify potential PDS early on and curtail PDS oscillations, reducing the exploration overhead.

7 CONCLUSIONS

This paper proposes a hierarchical multi-armed bandit framework for physical database design tuning, the first learned solution to work in the combined space of indices and views. Tapping into the optimiser knowledge while learning based on strategic exploration and observation allows our solution to eschew costly optimiser misestimates and heavy PDS creation times to provide significant performance gains. Our comprehensive experiments against a state-of-the-art commercial physical design tool under well known complex industrial benchmarks demonstrate the solution fitness by providing 40% and 75% average speed-up in static and random settings, respectively.

ACKNOWLEDGMENTS

We gratefully acknowledge support from the Australian Research Council Discovery Project DP220102269, as well as Discovery Early Career Researcher Award DE230100366.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*. 496–505. http://www.vldb.org/conf/2000/P496.pdf

[2] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB: Efficient Query Execution on Raw Data Files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 241–252.

[3] Kamel Aouiche and Jérôme Darmont. 2009. Data mining-based materialized view and index selection in data warehouses. *Journal of Intelligent Information Systems* 33, 1 (2009), 65–93.

[4] Kamel Aouiche, Pierre-Emmanuel Jouve, and Jérôme Darmont. 2006. Clustering-based materialized view selection in data warehouses. In *East European conference on advances in databases and information systems*. Springer, 81–95.

[5] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. 2016. Regularized Cost-Model Oblivious Database Tuning with Reinforcement Learning. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXVIII*. Springer, 96–132.

[6] Ladjel Bellatreche, Kamalakar Karlapalem, and Michel Schneider. 2000. On efficient storage space distribution among materialized views and indices in data warehousing environments. In *Proceedings of the ninth international conference on Information and knowledge management*. 397–404.

[7] Renata Borovica, Ioannis Alagiannis, and Anastasia Ailamaki. 2012. Automated Physical Designers: What You See is (Not) What You Get. In *Proceedings of the Fifth International Workshop on Testing Database Systems, DBTest 2012*. 9.

[8] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. 2018. Smooth Scan: Robust Access Path Selection Without Cardinality Estimation. *The VLDB Journal* 27, 4 (2018), 521–545.

[9] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. In *SIGMOD*. 227–238. https://doi.org/10.1145/1066157.1066184

[10] Nicolas Bruno and Surajit Chaudhuri. 2006. To Tune or Not to Tune?: A Lightweight Physical Design Alerter. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 499–510.

[11] Nicolas Bruno and Surajit Chaudhuri. 2007. An Online Approach to Physical Design Tuning. In *Proceedings - International Conference on Data Engineering*. 826–835.

[12] Nicolò Cesa-Bianchi and Gabor Lugosi. 2006. *Prediction, Learning, and Games*. Cambridge University Press.

[13] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "What-if"; Index Analysis Utility. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (Seattle, Washington, USA). ACM, New York, NY, USA, 367–378.

[14] Surajit Chaudhuri and Vivek R. Narasayya. [n.d.]. *Anytime Algorithm of Database Tuning Advisor for Microsoft SQL Server*. Retrieved April, 2022 from https://www:microsoft:com/en-us/research/publication/anytime-algorithm-of-databasetuning-advisor-for-microsoft-sql-server

[15] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. 146–155. http://www.vldb.org/conf/1997/P146.PDF

[16] Leonardo Weiss F Chaves, Erik Buchmann, Fabian Hueske, and Klemens Böhm. 2009. Towards materialized view selection for distributed databases. In *Proceedings of the 12th international conference on extending database technology: advances in database technology*. 1088–1099.

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

[18] Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zaït, and Mohamed Ziauddin. 2004. Automatic SQL Tuning in Oracle 10g. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*. 1098–1109.

[19] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2019 International Conference on Management of Data*. 666–679.

[20] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *Proc. VLDB Endow.* 4, 6 (mar 2011), 362–372. https://doi.org/10.14778/1978665.1978668

[21] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the 2019 International Conference on Management of Data*.

[22] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data*.

[23] Vahid Ghadakchi, Mian Xie, and Arash Termehchy. 2020. Bandit join: preliminary results. In *aiDM-SIGMOD 20*. 1–4.

[24] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing queries using materialized views: a practical, scalable solution. *ACM SIGMOD Record* 30, 2 (2001), 331–342.

[25] Goetz Graefe and Harumi A. Kuno. 2010. Self-Selecting, Self-Tuning, Incrementally Optimized Indexes.. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*. 371–381. https://doi.org/10.1145/1739041.1739087

[26] Hagit Grushka-Cohen, Ofer Biller, Oded Sofer, Lior Rokach, and Bracha Shapira. 2020. Using Bandits for Effective Database Activity Monitoring. In *PAKDD*. Springer, 701–713.

[27] Himanshu Gupta. 1997. Selection of views to materialize in a data warehouse. In *International Conference on Database Theory*. Springer, 98–112.

[28] Himanshu Gupta and Inderpal Singh Mumick. 1999. Selection of views to materialize under a maintenance cost constraint. In *International Conference on Database Theory*. Springer, 453–470.

[29] Yue Han, Guoliang Li, Haitao Yuan, and Ji Sun. 2021. An Autonomous Materialized View Management System with Deep Reinforcement Learning. 2159–2164. https://doi.org/10.1109/ICDE51399.2021.00217

[30] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 143–157. https://doi.org/10.1145/3318464.3389704

[31] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research*. 68–78.

[32] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the 2018 International Conference on Management of Data*. 535–550.

[33] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. 2018. Cuttlefish: A Lightweight Primitive for Adaptive Query Processing. (2018). unpublished.

[34] Andrew Kane. 2021. *Dexter Repository - The automatic indexer for Postgres*. Retrieved September, 2022 from https://github.com/ankane/dexter

[35] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *Conference on Innovative Data Systems Research*.

[36] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2382–2395.

[37] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2021. *Magic Mirror Repository*. Retrieved September, 2022 from https://github.com/hyrise/index_selection_evaluation

[38] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[39] TV Vijay Kumar and Santosh Kumar. 2012. Materialized view selection using simulated annealing. In *International Conference on Big Data Analytics*. Springer, 168–179.

[40] TV Vijay Kumar and Santosh Kumar. 2013. Materialized view selection using iterative improvement. In *Advances in Computing and Information Technology*. Springer, 205–213.

[41] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An Index Advisor Using Deep Reinforcement Learning. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management* (Virtual Event, Ireland) *(CIKM '20)*. Association for Computing Machinery, New York, NY, USA, 2105–2108. https://doi.org/10.1145/3340531.3412106

[42] Michael Lawrence. 2006. Multiobjective genetic algorithms for materialized view selection in olap data warehouses. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. 699–706.

[43] Michael Lawrence and Andrew Rau-Chaplin. 2006. Dynamic view selection for OLAP. In *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 33–44.

[44] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215.

[45] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. 2010. A Contextual-Bandit Approach to Personalized News Article Recommendation. In *Proceedings of the 19th International Conference on World Wide Web*. 661–670.

[46] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-Based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data*. 631–645.

[47] Imene Mami and Zohra Bellahsene. 2012. A Survey of View Selection Methods. *SIGMOD Rec.* 41, 1 (April 2012), 20–29. https://doi.org/10.1145/2206869.2206874

[48] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2020. Bao: Making Learned Query Optimization Practical. In *SIGMOD*.

[49] Ryan Marcus and Olga Papaemmanouil. 2017. Releasing Cloud Databases for the Chains of Performance Prediction Models.. In *CIDR*.

[50] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[51] Ryan Marcus and Olga Papaemmanouil. 2019. Towards a Hands-Free Query Optimizer through Deep Learning. In *Conference on Innovative Data Systems Research*.

[52] Microsoft. [n.d.]. *TPC-H Skew Benchmark*. Retrieved March, 2021 from https://www.microsoft.com/en-us/download/details.aspx?id=52430

[53] Hoshi Mistry, Prasan Roy, S Sudarshan, and Krithi Ramamritham. 2001. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. 307–318.

[54] Meikel Nambiar, Raghunath Othayoth and Poess. 2006. The Making of TPC-DS. *Proceedings of the 32nd International Conference on Very Large Data Bases* (2006), 1049–1058.

[55] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. 1978. An Analysis of Approximations for Maximizing Submodular Set Functions–I. *Mathematical programming* 14, 1 (1978), 265–294.

[56] Bastian Oetomo, Malinga Perera, Renata Borovica-Gajic, and Benjamin I P Rubinstein. 2019. A Note on Bounding Regret of the C^2UCB Contextual Combinatorial Bandit. *arXiv preprint arXiv:1902.07500* (2019).

[57] Bastian Oetomo, R. Malinga Perera, Renata Borovica-Gajic, and Benjamin I. P. Rubinstein. 2021. Cutting to the Chase with Warm-Start Contextual Bandits. In *2021 IEEE 21th International Conference on Data Mining (ICDM)*.

[58] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems. In *Conference on Innovative Data Systems Research*.

[59] R Malinga Perera, Bastian Oetomo, Benjamin IP Rubinstein, and Renata Borovica-Gajic. 2021. 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, 600–611.

[60] R Malinga Perera, Bastian Oetomo, Benjamin IP Rubinstein, and Renata Borovica-Gajic. 2021. No DBA? No regret! Multi-armed bandits for index tuning of analytical and HTAP workloads with provable guarantees. *arXiv preprint arXiv:2108.10130* (2021).

[61] Lijing Qin, Shouyuan Chen, and Xiaoyan Zhu. 2014. Contextual Combinatorial Bandit and its Application on Diversified Online Recommendation. In *Proceedings of the 2014 SIAM International Conference on Data Mining*. 461–469.

[62] Stefano Rizzi and Ettore Saltarelli. 2003. View materialization vs. indexing: Balancing space constraints in data warehouse design. In *International Conference on Advanced Information Systems Engineering*. Springer, 502–519.

[63] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. 2003. QUIET: Continuous querydriven index tuning. In *Proceedings 2003 VLDB Conference*. Elsevier, 1129–1132. https://doi.org/10.1016/B978-012722442-8/50122-1

[64] Rainer Schlosser, Jan Kossmann, and Martin Boissier. 2019. Efficient Scalable Multi-Attribute Index Selection Using Recursive Strategies. https://doi.org/10.1109/ICDE.2019.00113

[65] Rainer Schlosser, Jan Kossmann, and Martin Boissier. 2019. Efficient Scalable Multi-attribute Index Selection Using Recursive Strategies. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1238–1249. https://doi.org/10.1109/ICDE.2019.00113

[66] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. 2007. On-Line Index Selection for Shifting Workloads. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*. 459–468.

[67] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. (2018). unpublished.

[68] TPC. [n.d.]. *TPC-H Benchmark*. Retrieved March, 2021 from http://www.tpc.org/tpch/

[69] Immanuel Trummer, Samuel Moseley, Deepak Maram, Saehan Jo, and Joseph Antonakakis. 2018. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. *Proc. VLDB Endow.* 11, 12 (2018), 2074–2077.

[70] Gary Valentin, Michael Zuliani, Daniel Zilio, Guy Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend its own Indexes. *Proceedings - International Conference on Data Engineering*, 101–110. https://doi.org/10.1109/ICDE.2000.839397

[71] TV Vijay Kumar and Kalyani Devi. 2012. Materialised view construction in data warehouse for decision making. *International Journal of Business Information Systems* 11, 4 (2012), 379–396.

[72] TV Vijay Kumar and Santosh Kumar. 2014. Materialised view selection using differential evolution. *International Journal of Innovative Computing and Applications* 6, 2 (2014), 102–113.

[73] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. *UDO Repository*. Retrieved September, 2022 from https://github.com/jxiw/UDO

[74] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: Universal Database Optimization Using Reinforcement Learning. 14, 13 (sep 2021), 3402–3414. https://doi.org/10.14778/3484224.3484236

[75] Kyu-Young Whang. 1987. *Index Selection in Relational Databases*. Springer US, Boston, MA, 487–500. https://doi.org/10.1007/978-1-4613-1881-1_41

[76] Xuanhe Zhou, Luyang Liu, Wenbo Li, Lianyuan Jin, Shifu Li, Tianqing Wang, and Jianhua Feng. 2022. Autoindex: An incremental index management system for dynamic workloads. ICDE.

[77] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries Using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (may 2020), 1416–1428. https://doi.org/10.14778/3397230.3397238

[78] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*. 1087–1097.