2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00058

DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees

R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, Renata Borovica-Gajic
{malinga.perera, b.oetomo}@student.unimelb.edu.au, {brubinstein, renata.borovica}@unimelb.edu.au
School of Computing and Information Systems
University of Melbourne

***Abstract*—Automating physical database design has remained a long-term interest in database research due to substantial performance gains afforded by optimised structures. Despite significant progress, a majority of today's commercial solutions are highly manual, requiring offline invocation by database administrators (DBAs) who are expected to identify and supply representative training workloads. Even the latest advancements like query stores provide only limited support for dynamic environments. This status quo is untenable: identifying *representative* static workloads is no longer realistic; and physical design tools remain susceptible to the query optimiser's cost misestimates.**

We propose a self-driving approach to online index selection that eschews the DBA and query optimiser, and instead *learns* the benefits of viable structures through strategic exploration and direct performance observation. We view the problem as one of sequential decision making under uncertainty, specifically within the bandit learning setting. Multi-armed bandits balance exploration and exploitation to *provably* guarantee average performance that converges to policies that are optimal with perfect hindsight. Our simplified bandit framework outperforms deep reinforcement learning (RL) in terms of convergence speed and performance volatility. Comprehensive empirical results demonstrate up to 75% speed-up on shifting and ad-hoc workloads and 28% speed-up on static workloads compared against a state-of-the-art commercial tuning tool and up to 58% speed-up against the deep RL alternatives.

***Index Terms*—Automated Indexing, Autonomous Databases, Reinforcement Learning, Multi-armed Bandits**

I. Introduction

With the increasing complexity and variability of database applications and their hosting platforms (e.g., multi-tenant cloud environments), automated physical design tuning, particularly automated index selection, has re-emerged as a contemporary challenge for database management systems. Most database vendors offer automated tools for physical design tuning within their product suites [1]–[3]. Such tools form an integral part of broader efforts toward fully automated database management systems which aim to: a) decrease database administration costs and thus total costs of ownership [4], [5]; b) help non-experts use database systems; and c) facilitate hosting of databases on dynamic environments such as cloud-based services [6]–[9]. Most physical design tools take an *offline* approach, where the representative training workload is provided by the database administrator (DBA) [10]. Where *online* solutions are provided [8], [11]–[15], questions remain: How often should the tools be invoked? And importantly, is the quality of proposed designs in any way guaranteed? How can tools generalise beyond queries seen to dynamic ad-hoc workloads, with unpredictable and non-stationary queries?

Modern analytics workloads are dynamic in nature with ad-hoc queries common [16], e.g., data exploration workloads adapt to past query responses [17]. Such ad-hoc workloads hinder automated tuning since: a) inputting representative information to design tools is infeasible under time-evolving workloads; and b) reacting too quickly to changes may result in performance variability, where indices are continuously dropped and created. Any robust automated physical design solution must address such challenges [11].

To compare alternative physical design structures, automated design tools use a cost model employed by the query optimiser, typically exposed through a "what-if" interface [18], as the sole source of truth. However such cost models make inappropriate assumptions about data characteristics [19], [20]: commercial DBMSs often assume attribute value independence and uniform data distributions when sufficient statistics are unavailable [20]–[22]. As a result, estimated benefits of proposed designs may diverge significantly from actual workload performance [8], [9], [22]–[24]. Even with more complex data distribution statistics such as single- and multi-column histograms, the issue remains for complex workloads [22].

In this paper, we demonstrate that even in ad-hoc environments where queries are unpredictable, there are opportunities for index optimisation. We argue that the problem of online index selection under ad-hoc, analytical workloads can be efficiently formulated within the multi-armed bandit (MAB) learning setting—a tractable form of Markov decision process. MABs take arms or actions (selecting indices) to maximise cumulative rewards, trading off exploration of untried actions with exploitation of actions that maximise rewards observed so far (see Figure 1). MABs permit learning from observations of actual performance, and need not rely on potentially misspecified cost models. Unlike initial efforts with applying learning for physical design, e.g., more general forms of reinforcement learning [25], bandits offer regret bounds that *guarantee* the fitness of dynamically-proposed indices [26].

The key contributions of the paper are summarised next:

- We model index tuning as a multi-armed bandit, proposing design choices that lead to a practical, competitive solution;
- Our proposed design achieves a worst-case safety guarantee against any optimal fixed policy, as a consequence

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00058

of a corrected regret analysis of the C^{2}UCB bandit; and
- Our comprehensive experiments demonstrate MAB's superiority over a state-of-the-art commercial physical design tool, with up to 75% speed-ups under dynamic, analytical workloads.

II. Problem Formulation

The goal of the *online database index selection problem* is to choose a set of indices (referred to as a *configuration*) that minimises the total running time of a workload sequence within a given memory budget. Neither the workload sequence, nor system run times, are known in advance.

We adopt the problem definition of [13]. Let the *workload* $W = (w_1, w_2, \ldots, w_T)$ be a sequence of *mini-workloads* (e.g., a sequence of single queries), I the set of *secondary indices*, $C_{mem}(s)$ represent the memory space required to materialise a configuration $s \subseteq I$, and $\mathcal{S} = \{s \subseteq I \,|\, C_{mem}(s) \leq M\} \subseteq 2^I$ be the class of *index configurations* feasible within our total memory allowance M. Our goal is to propose a configuration sequence $S = (s_0, s_1, \ldots, s_T)$, with $s_t \in \mathcal{S}$ as the configuration in round t and $s_0 = \emptyset$ as the starting configuration, which minimises the *total workload time* $C_{tot}(W, S)$ defined as:

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

Here $C_{rec}(t)$ refers to the *recommendation time* in round t (defined as running time of the recommendation tool) and $C_{cre}(s_{t-1}, s_t)$ refers to the incremental index 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, namely the sum of response times of individual queries.

At round t, the system:

1) Chooses a set of indices $s_t \in \mathcal{S}$ in preparation for upcoming workload w_t, without direct access to w_t. s_t only depends on observation of historical workloads $(w_1, \ldots, w_{t-1})$, corresponding sets of chosen indices, and resulting performance;
2) Materialises the indices in s_t which do not exist yet, that is, all indices in the set difference $s_t \backslash s_{t-1}$; and
3) Receives workload w_t, executes all the queries therein, and measures elapsed time of each individual query and each operator in the corresponding query plan.

III. Contextual Combinatorial Bandits

In this paper, we argue that online index selection can be successfully addressed using multi-armed bandits (MABs) from statistical machine learning, where different arms correspond to chosen indices. We first present necessary background on MABs, outlining the essential properties that we exploit in our work (i.e., bandit context and combinatorial arms) to converge to highly performant index configurations.

We use the following notation. We denote non-scalar values with boldface: lowercase for vectors and uppercase for matrices. We also write $[k] = \{1, 2, \ldots, k\}$ for $k \in \mathbb{N}$, and denote the transpose of a matrix or a vector with a prime.

The contextual combinatorial bandit setting under *semi-bandit* feedback involves repeated selections from k possible actions, over rounds $t = 1, 2, \ldots$, in which the MAB:

1) Observes a *context* feature vector (possibly random or adversarially chosen) of 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) Selects or *pulls* a set of arms (referred to as *super arm*) $s_t \in \mathcal{S}_t$, where we restrict the class of possible super arms $\mathcal{S}_t \subseteq \mathcal{S}'_t = \left\{ s \subseteq [k] \,\middle|\, \sum_{i\in s} c_i \leq C \right\} \subseteq 2^{[k]}$; and
3) For each $i_t \in s_t$, observes random *scores* $r_t(i_t)$ drawn from fixed but unknown arm distribution which depends solely on the arm i_t and its context $\boldsymbol{x}_t(i_t)$, whose true expected values are contained in the unknown variable $\boldsymbol{r}^\star_t = \{\mathbb{E}[r_t(i)]\}_{i\in[k]}$.

A MAB's goal is to maximise the cumulative expected reward $\sum_t \mathbb{E}[R_t(s_t)] = \sum_t g(\boldsymbol{r}^\star_t, \boldsymbol{X}_t, s_t)$ for a known function g. This function g need not be a simple summation of all the scores. The core challenge in this problem is that the expected scores for all arms $i \in [k]$ are unknown. Refinement of a bandit learner's approximation for arm i is *generally* only possible by including arm i in the super arm, as the score for arm i is not observable when i is not played. This suggests solutions that balance *exploration* and *exploitation*. Even though at first glance it may seem that each arm needs to be explored at least once, placing practical limits on large numbers of arms, there is a remedy to this as will be discussed shortly.

The C^{2}UCB algorithm. Used to solve the contextual combinatorial bandit problem, the C^{2}UCB Algorithm [26] models the arms' scores as linearly dependent on their contexts: $r_t(i) = \boldsymbol{\theta}' \boldsymbol{x}_t(i) + \varepsilon_t(i)$ for unknown zero-mean (subgaussian) random variable ε_t, unknown but fixed parameter $\boldsymbol{\theta} \in \mathbb{R}^d$, and known context $\boldsymbol{x}_t(i)$. It is crucial to notice that this implies that **all learned knowledge is contained in estimates of $\boldsymbol{\theta}$, which is shared between all arms, obviating the need to explore each arm**. Estimation of $\boldsymbol{\theta}$ can be achieved using ridge regression, with $|s_t|$ new data points $\{(\boldsymbol{x}_t(i), r_t(i))\}_{i\in s_t}$ available at round t, further *accelerating the convergence rate* of the estimator $\hat{\boldsymbol{\theta}}$, over observing only one example as might be naïvely assumed.

Point estimates on the expected scores can be made with $\bar{r}_t(i) = \hat{\boldsymbol{\theta}}'_t \boldsymbol{x}_t(i)$, where $\hat{\boldsymbol{\theta}}_t$ are trained coefficients of a ridge regression on observed rewards against contexts. However, this quantity is oblivious to the variance in the score estimation. Intuitively, to balance out the exploration and exploitation, it is desirable to add an *exploration boost* to the arms whose score we are less sure of (i.e., greater estimate variance). This suggests that the upper confidence bound (UCB) should be used, in place of the expected value, and which can be calculated [27] as:

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

where $\alpha_t > 0$ 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 first term

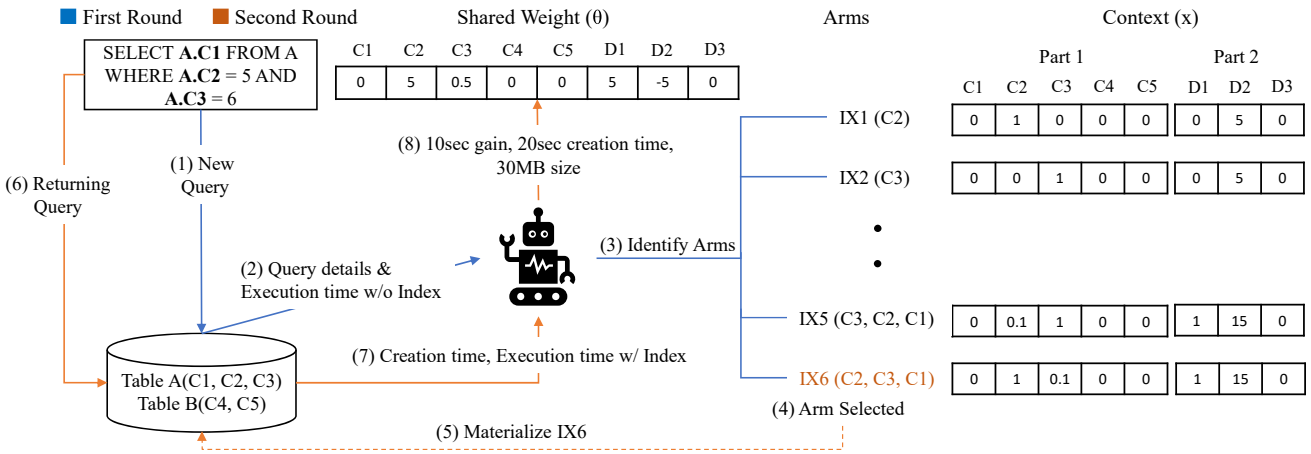

Fig. 1. An abstract view of the proposed bandit learning-based online index selection.

of $\hat{r}_t(i)$ corresponds to arm i's immediate reward, whereas its second term corresponds to its exploration boost, as its value is larger when the arm is sensitive to the context elements we are less confident of (i.e., the underexplored context dimension). Hence, by using $\hat{r}_t(i)$ in place of $\bar{r}_t(i)$, arms with contexts lying in the underexplored regions of context space are more likely to be chosen, as higher scores yield higher g, assuming that g is monotonic increasing in the arm rewards.

Ideally, the super arm $s_t \in \mathcal{S}_t$ is chosen such that $g(\hat{\boldsymbol{r}}_t, \boldsymbol{X}_t, s_t)$ is maximised. However, it is sometimes computationally expensive to find such super arms. In such cases, it is often good enough to obtain a solution via some approximation algorithm where $g(\hat{\boldsymbol{r}}_t, \boldsymbol{X}_t, s_t)$ is *near* maximum. With this criterion in mind, we now define an *α-approximation oracle*.

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

Note that knapsack-constrained submodular programs are efficiently solved by the greedy algorithm (iteratively select a remaining cost-feasible arm with highest available score) with $\alpha = 1 - 1/e$. C^{2}UCB is detailed in Algorithm 1.

The performance of a bandit algorithm is usually measured by its *cumulative regret*, defined as the total expected difference between the reward of the chosen super arm $\mathbb{E}[R_t(s_t)]$ and an optimal super arm $\max_{s \in \mathcal{S}_t} \mathbb{E}[R_t(s)]$ over T rounds. Such a metric is unfair to C^{2}UCB since its performance depends on the oracle's performance. This suggests measuring C^{2}UCB's performance with a metric using the oracle's performance guarantee as its measuring stick, as follows.

Definition 2. *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, while* $\boldsymbol{r}_t^\star$ *is a vector containing each arms' true expected scores.*

When g is assumed to be monotonic and Lipschitz continuous, [26] claimed that C^{2}UCB enjoys $\tilde{O}(\sqrt{T})$ α-regret. We have corrected an error in the original proof, as seen in our technical note [28], confirming the $\tilde{O}(\sqrt{T})$ α-regret. This expression is sub-linear in T, implying that the per-round average cumulative regret approaches zero after sufficiently many rounds. Consequently, online index selection based on C^{2}UCB comes endowed with a *safety guarantee* on worst-case performance: selections become at least as good as an α-optimal policy (with perfect access to true scores); and potentially much better than any fixed policy.

Algorithm 1 The C^{2}UCB Algorithm

Input: $\lambda, \alpha_1, \ldots, \alpha_T$
Initialize $\boldsymbol{V}_0 \leftarrow \lambda \boldsymbol{I}_d$, $\boldsymbol{b}_0 \leftarrow \boldsymbol{0}_d$
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

IV. MAB for Online Index Selection

Performant bandit learning for online index tuning demands arms covering important actions and no more, rewards that are observable and for which regret bounds are meaningful, and contexts and oracle that are efficiently computable and predictive of rewards. Each workload query is monitored for characteristics such as running time, query predicates, payload, etc. (see Figure 1). These observations feed into generation of relevant arms and their contexts. The learner selects a desired configuration which is materialised. After query return, the

system identifies benefits of the materialised indices, which are then shaped into the reward signal for learning.

Dynamic arms from workload predicates. Instead of enumerating all column combinations, *relevant* arms (indices) may be generated based on queries: combinations and permutations of query predicates (including join predicates), with and without inclusion of payload attributes from the selection clause. Such workload-based arm generation drastically reduces the action space, and exploits natural skewness of real-life workloads that focus on small subsets of attributes over full tables [17]. Workload-based arm generation is only viable due to dynamic arm addition (reflecting a dynamic action space) and is allowed by the bandit setting: we may define the set of feasible arms for each round at its start.

Context engineering. Effective contexts are predictive of rewards, efficiently computable, and promote generalisation to previously unseen workloads and arms. We form our context in two parts (see Figure 1).

Context Part 1: Indexed column prefix. We encode one context component per column. However unlike a bag-of-words or one-hot representation appropriate for text, similarity of arms depends on having similar column prefixes; common index columns is insufficient. This reflects a ***novel bandit learning aspect of the problem.*** A context component has value 10^{-j} where j is the corresponding column's position in the index, *provided* that the column is included in the index and is a workload predicate column. The value is set to 0 otherwise, including if its presence only covers the payload. Unlike a simple one hot encoding, this context enables the bandit to differentiate between arms with the same set of columns but different ordering, and reward the columns differently based on the column's position in the index.

Example 3. *Under the simplest workload (single query) in Figure 1, our system generates six arms: four using different combinations and permutations of the predicates, two including the payload (covering indices). Index IX5 includes column C1, but the context for C1 is valued as* 0*, as this column is considered only due to the query payload.*

Context Part 2: Derived statistical information. We represent statistical and derived information about the arms and workload, details available during query execution, and sufficient statistics for unbiased estimates. This statistical information includes: a Boolean indicating a covering index, the estimated size of the index divided by the database size (if not materialised already, 0 otherwise), and the number of times the optimiser has picked this arm in recent rounds. This is shown in Figure 1 under D1, D2 and D3, respectively.

Reward shaping. As the goal of physical design tuning tools is to minimise end-to-end workload time, we incorporate index creation time and query execution time into the reward for a workload. We omit index recommendation time, as it is independent of arm selection. However, we measure and report recommendation time of the MAB algorithm in our experiments. Recall that MAB depends only on observed execution statistics from implemented configurations and generalisation of the learned knowledge to unseen arms thereafter.

The implementation of the reward for an arm includes the execution time as a *gain* $G_t(i, w_t, s_t)$ for a workload w_t by each arm i under configuration s_t. By defining $\mathcal{U}(s, q)$ as the set of indices used by the optimiser for query q for a given configuration s, the gain by index i for a query q is defined:

$$\begin{aligned} & G_t(i, \{q\}, s_t) \\ = \; & \left[C_{tab}(\tau(i), q, \emptyset) - C_{tab}(\tau(i), q, \{i\})\right] \mathbb{1}_{\mathcal{U}(s,q)}(i)\,, \end{aligned}$$

where $\tau(i)$ is the table which i belongs to and $C_{tab}(\tau(i), q, \emptyset)$ represents the full table scan time for table $\tau(i)$ and query q.[1] The gain for a workload relates to the gain for individual query by:

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

By this definition, gain $G_t(i, w_t, s_t)$ will be 0 if i is not used by the optimiser in the current round t and can be negative if the index creation leads to a performance regression. Creation time of i is taken as a negative reward, only if i is materialised in round t, and is 0 otherwise:

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

Minimising the end-to-end workload time, or rather, maximising the end-to-end workload time gained, is the goal of the bandit. As defined earlier, the total workload time C_{tot} is the sum of *execution*, *recommendation* and *creation* times accumulated over rounds. As such, minimising each round's summand is an equivalent problem. Modifying the execution time to the time gain while ignoring the recommendation time yields per-round super arm reward of:

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

Selection of the execution plan depends on the query optimiser, and as noted, the query optimiser may resolve to a sub-optimal query plan. As we show, the bandit is nonetheless resilient as it can quickly recover from any such performance regressions. Observed execution times encapsulate real-world effects, e.g., the interaction between queries, application properties, runtime parameters, etc. Since the end-to-end workload time includes the index creation and query execution times, we are indirectly optimising for both efficiency and the quality of recommendations.

A greedy oracle for super-arm selection. Recall that C^{2}UCB leverages a near-optimal oracle to select a super arm, based on individual arm scores [26]. As a sum of individual arm rewards, our super-arm reward has a (sub)modular objective function and (as easily proven) exhibits monotonicity

[1]Due to the reactive nature of multi-armed bandits, we mostly observe a full table scan time for each table $\tau(i)$ and query q. When we do not observe this, we estimate it with the maximum secondary index scan/seek time.

Algorithm 2 MAB Simulation for Index Tuning

```
1:  QS ← QueryStore()                          ▷ keeps query information
2:  C²UCB ← InitialiseBandit()                 ▷ A1, L 1-2
3:  while (TRUE) do
4:      queries ← getLastRoundWorkload()
5:      for all queries do
6:          if (isNewTemplate) then
7:              QS.add(query)
8:          else
9:              QS.update(query)
10:         end if
11:     end for
12:     QoI ← QS.getQoI()                      ▷ get queries of interest
13:     arms ← generateArms(QoI)
14:     X ← generateContext(arms, QoI)
15:     s_t ← C²UCB.recommend(arms, X)         ▷ A1, L 4-10
16:     C_cre ← materialise(s_t)
17:     C_exc ← executeCurrentWorkload()
18:     C²UCB.updateWeights(C_cre, C_exc)      ▷ A1, L 12-13
19: end while
```

and Lipschitz continuity. Approximate solutions to maximise submodular (diminishing returns) objective functions can be obtained with greedy oracles that are efficient and near-optimal [29]. Our implementation uses such an oracle combined with filtering to encourage diversity. Initially, arms with negative scores are pruned. Then arm selection and filtering steps alternate, until the memory budget is reached. In the selection step, an arm is selected greedily based on individual scores. The filtering step filters out arms that are no longer viable under the remaining memory budget, or those that are already covered by the selected arms based on prefix matching. If a covering index is selected for a query, all other arms generated for that query will be filtered out. Note that filtering is a temporary process that only impacts the current round.

Bandit learning algorithm. Algorithm 2 shows the MAB algorithm, which wraps Algorithm 1 and handles the domain specific aspects of the implementation. We have divided Algorithm 1 into three main parts, initialisation (lines 1-2), arm recommendation (lines 4-10) and weight vector update (lines 12-13). These segments are utilised in the Algorithm 2 as C^{2}UCB function calls. After initialising the bandit, Algorithm 2 summarises workload information using templates; these track frequency, average selectivity, first seen and last seen times of the queries which help to generate the best set of arms per round (i.e., QoI). The context is updated after each round based on the workload and selected set of arms. The bandit then selects the round's set of arms, forming a configuration to be materialised within the database. The reward will then be calculated based on observed execution statistics on a new set of queries, and will be used to update the shared weight. To support shifting workloads, where users' interests change over time, the learner may forget learned knowledge depending on the workload shift intensity (i.e., the number of newly introduced query templates).

V. Experimental Evaluation

We evaluate our MAB framework across a range of widely used industrial benchmarks, comparing it to a state-of-the-art physical design tool shipped with a commercial database product referred to as the Physical Design Tool (PDTool). This is a mature product, proven to outperform other physical design tools available on the market [23], [30].

A. Experimental Setup

Benchmarks. We use five publicly available benchmarks: TPC-H (with uniform distribution) [31] and TPC-H Skew [32] with zipfian factor 4, allowing the reader to understand the impact of data skewness when all the other aspects are kept identical; TPC-DS [33], which demonstrates the solution fitness under a large number of candidate configurations; SSB [34] with easily achievable high index benefits; and finally, Join Order Benchmark (JOB) with IMDb dataset (a real-world dataset) [20] (henceforth referred to as IMDb) a challenging workload for index recommendations, with index overuse leading to performance regressions.

Unless stated otherwise, all experiments use scale factor (SF) 10, resulting in approximately 10GB of data per workload, except in the case of the IMDb dataset which has a fixed size of 6GB. We consider three broad types of workloads, allowing us to compare different aspects of the recommendation process:

Static: The workload sequence is known in advance, and repeating over time (modelling workloads used for reporting purposes). In absence of dynamic environment complexities, this simpler setting allows us to single-out the effectiveness (ability to find a better configuration) and the efficiency (additional overhead) of the MAB search strategy.

Dynamic shifting: The region of interest shifts over time from one group of queries to another (modelling data exploration). This experiment evaluates the adaptation speed to workload shifts and the cost of exploration when adapting.

Dynamic random: A query sequence is chosen entirely at random (modelling more dynamic settings, such as cloud services). Dynamic random experiments test the delicate balance between swift and careful adaptation under returning workloads, which can lead to unwanted index oscillations.

Both PDTool and MAB are given a memory budget approximately equal to the size of the data (1x; 10GB for SF 10 datasets and 6GB for IMDb dataset) for the creation of secondary indices. We have experimented with different memory budgets ranging from 0.25x to 2x (since benefits of additional memory seem to diminish beyond a 2x limit) under TPC-H and TPC-H skew benchmarks, and observed the same patterns throughout that range.[2] We have naturally picked the middle of the active region (1x) as our memory budget. All these workloads come with original primary and foreign keys that influence the choice of indices. We grant the aforementioned memory budget on top of this.

[2]Both tools converge to the same execution cost by the final round, when enough memory was given to fit the entire useful configuration.

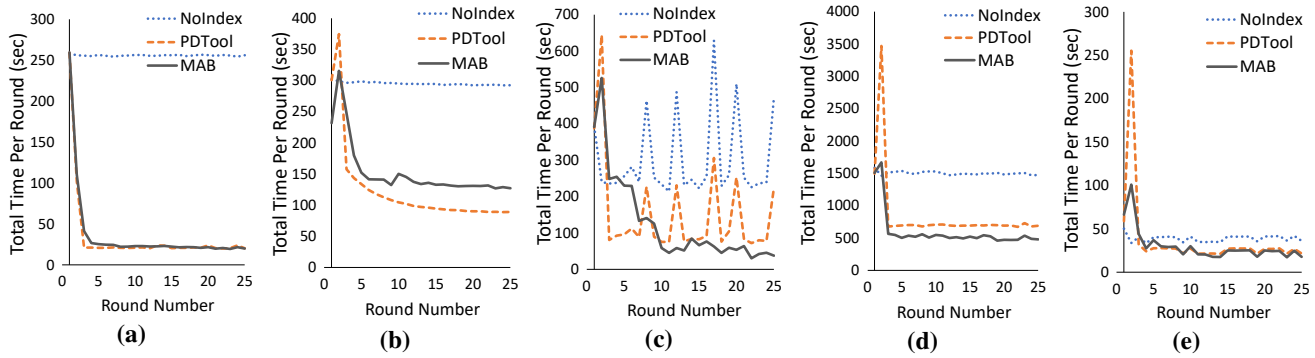

Fig. 2. MAB vs. PDTool Convergence for *static* workloads: (a) SSB, (b) TPC-H, (c) TPC-H Skew, (d) TPC-DS and (e) IMDb.

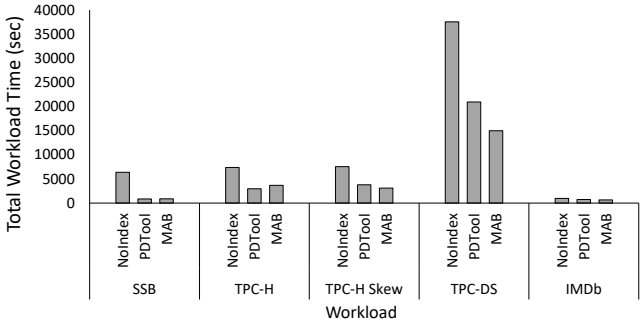

Fig. 3. MAB vs. PDTool total end-to-end workload time for *static* workloads.

In search of the best possible design, we do not constrain the running time of PDTool, with one exception: In TPC-DS dynamic random, PDTool was uncompetitive due to long running times,[3] hence the PDTool running time of each invocation was restricted to 1 hour. All proposed indices are materialised and queries invoked over the same commercial DBMS in both cases (MAB and PDTool).

Across experiments, each group of templatized queries is invoked over rounds, producing different query instances. For static and dynamic settings, PDTool is invoked every time after the first round of new queries, with those queries given as the training workload, since this workload will become representative of future rounds. This setting is somewhat unrealistic and favourable for PDTool, since in real-life the PDTool will seldom truly have knowledge of the representative workload (i.e., what is yet to arrive in the future), advantaging the PDTool in our experiments. However, it presents a viable comparison against the workload-oblivious MAB. Bandits do not use any workload information ahead of time, but instead observe workload sequence and react accordingly.

Hardware. All experiments are performed on a server equipped with 2x 24 Core Xeon Platinum 8260 at 2.4GHz, 1.1TB RAM, and 50TB disk (10K RPM) running Windows Server 2016. We report cold runs, clearing database buffer caches prior to every query execution.

[3]A single PDTool invocation took around 8 hours (default limit). The total recommendation time was around 40 hours, which is not competitive compared to the end-to-end workload time of 4 hours under MAB.

Metrics. In addition to reporting total end-to-end workload time for all rounds, we also report the total workload time per round used to demonstrate the convergence of different tools. Additionally, we present the total workload time broken down by recommendation time (when invoking the PDTool or the MAB framework), index creation time, and workload execution time. For completeness, we show original query times, without any secondary indices (denoted as NoIndex). In addition to convergence graphs of individual benchmarks, we present a summary graph with total end-to-end workload time for all rounds under MAB and PDTool tuning of SSB, TPC-H (uniform), TPC-H skew, TPC-DS and IMDb benchmarks.

B. Experimental Results

1) MAB versus the PDTool: We report on wide ranging empirical comparisons of MAB and PDTool.

Static workloads. Static workloads over uniform datasets are the best case for offline physical design tools, as a pre-determined workload sequence may perfectly represent future queries. However, when underlining data is skewed, recommendations based on a pre-determined workload alone can have unfavourable outcomes. While used for reporting, static workloads do not reflect modern dynamic workloads (e.g., data exploration). In static workloads, all query templates in the benchmark (22, 13, 99 and 33 templates for TPC-H, SSB, TPC-DS and IMDb, respectively) are invoked once every round, each with a different query instance of the template, for a total of 25 rounds, providing sufficient time for convergence.

Figure 3 displays overall workload time (including recommendation and index creation time) for all 25 rounds under MAB and PDTool. For skewed datasets (TPC-H Skew, TPC-DS, IMDb) MAB outperforms PDTool. MAB shows over 17%, 28% and 11% performance gain against PDTool, under TPC-H Skew, TPC-DS, IMDb benchmarks, respectively. Under uniform datasets (TPC-H and SSB), both MAB and PDTool provide significant performance gains over NoIndex (over 50% and 85%, respectively), while PDTool outperforms the MAB (by 19% and 5%). This is not surprising since for uniform, static experiments usually align with PDTool assumptions and the future can be perfectly represented by a pre-determined workload.

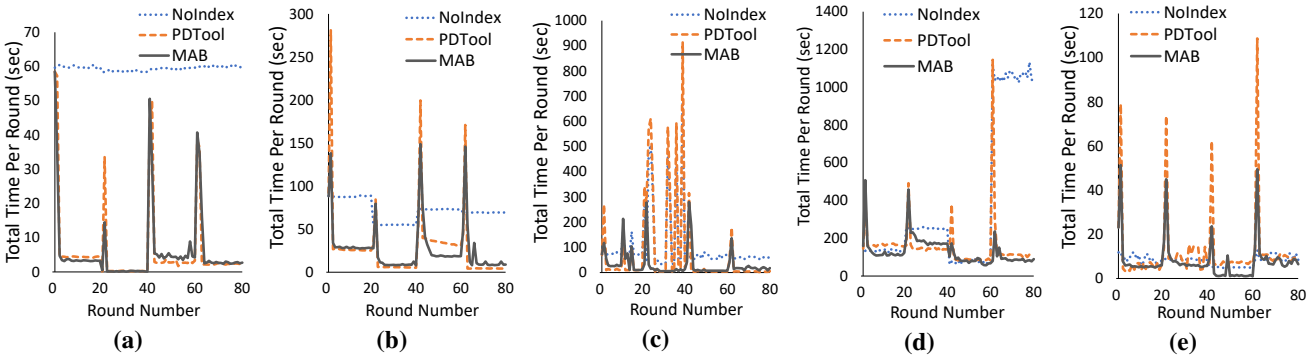

Fig. 4. MAB vs. PDTool Convergence for *dynamic shifting* workloads: (a) SSB, (b) TPC-H, (c) TPC-H Skew, (d) TPC-DS and (e) IMDb.

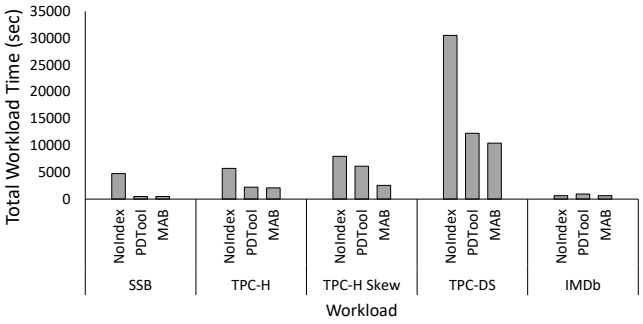

Fig. 5. MAB vs. PDTool total end-to-end workload time for *dynamic shifting* workloads.

Convergence plots in Figure 2(a–e), show MAB's gradual improvement over 25 rounds. Both MAB and PDTool have large spikes after the first round for all the workloads. For both tools, this is due to recommendation and creation of indices. However, MAB might drop proposed indices and create new ones later on, generating relatively smaller spikes in subsequent rounds. Nonetheless, MAB efficiently balances the exploration of new indices, reducing exploration with time.

What is the best search strategy? Comparison of execution times in the final round of the static experiment provides a clear idea about the benefit of using execution cost guided search. As evident from Figure 2(a–e), in 4 out of 5 cases, MAB converges to a better configuration than PDTool. MAB provides over 5%, 84%, 31% and 19% better execution time by the last round (25^{th}) compared to PDTool under SSB, TPC-H Skew, TPC-DS and IMDb, respectively.

For TPC-H skew, PDTool misses an index on $Orders.O\_custkey$. This index boosts the performance of some queries (Q22 in particular) which MAB correctly detects and materialises. Missing this leads to large execution times in a few rounds including the last round (8, 12, 17, 20 and 25) for PDTool. These experiments illustrate the risk of relying on the query optimiser and imperfect statistics as a single source of truth.

The only case when MAB is outperformed by the PDTool is under TPC-H (PDTool delivers over 21% better execution time by the last round): different indices are proposed, as our current MAB framework does not support an index merging phase employed by some physical design tools [35]. Instead, MAB uses individual queries to propose index candidates. We plan to address index merging in future work. In addition to TPC-H (uniform) and TPC-H skew with Zipfian factor 4, we experimented with different degrees of skewness (Zipfian factors) ranging from 1 to 3. Under Zipfian factors 2 and 3, MAB showed over 51% and 58% performance gain against PDTool, respectively. Whereas under Zipfian factor 1, PDTool outperformed the MAB by 16%.

Dynamic shifting workloads. Under the dynamic shifting workloads, all query templates in the benchmark are randomly divided into 4 equal-sized groups. A group of query templates is then executed for 20 rounds, after which the workload switches to a new group of unseen queries (no overlap with the previous queries). When the workload switches, PDTool is invoked and trained on the new sequence of queries (whose templates will be used in the next 19 rounds).[4] Thus, PDTool is invoked four times in total (in rounds 2, 22, 42, 62). On the other hand, the MAB framework does not assume any workload knowledge.

Figure 5 displays MAB's end-to-end workload time as substantially lower compared to the alternatives, under all benchmarks. MAB provides over 3%, 6%, 58%, 14% and 34% speed-up compared to PDTool, under SSB, TPC-H, TPC-H Skew, TPC-DS and IMDb, respectively.

Interestingly, NoIndex performs better than PDTool against the IMDb workload. PDTool has a higher total workload time as well as higher execution time compared to NoIndex. NoIndex provides 3.5% (24 seconds) speedup in execution time over PDTool. This is mainly due to misestimates of the optimiser [8]. As an example (out of many), query 18 takes less than 1 second under NoIndex, whereas with the created indices by PDTool some instances of this query take around 7-8 seconds due to a suboptimal plan chosen favoring the index usage. This affects both MAB and PDTool, but MAB identifies the indices with a negative impact based on the reward and drops them. For the IMDb workload which does not get much support from indices, MAB provides 3% total performance gain and 26% execution time gain compared to NoIndex.

[4]This relaxation assumes a DBA with knowledge that the previous workload will not be repeated, placing PDTool at an advantage. In reality, proposing training workloads might be much more challenging for dynamic workloads.

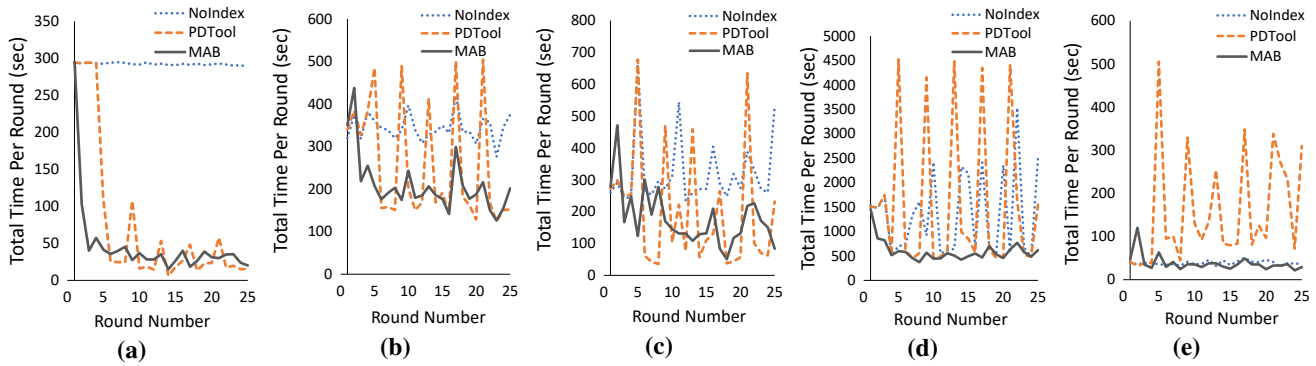

Fig. 6. MAB vs. PDTool Convergence for *dynamic random* workloads: (a) SSB, (b) TPC-H, (c) TPC-H Skew, (d) TPC-DS and (e) IMDb

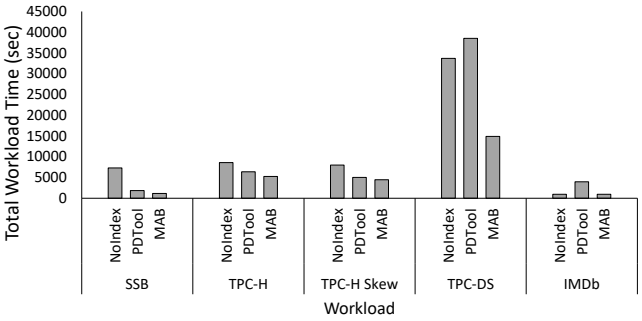

Fig. 7. MAB vs. PDTool total end-to-end workload time for *dynamic random* workloads.

One can easily observe the workload shifts in Figure 4(a–e) due to the spikes in rounds 2, 22, 42, and 62. For PDTool, this is due to the invocation of PDTool and index creation after the workload shifts. Similar spikes can be seen in the MAB line with automatic detection of workload shifts. Further random spikes can be observed, for PDTool, from rounds 20-40 in TPC-H skew and rounds 30-40 under IMDb, due to the issues discussed in the previous paragraphs (Q22 in TPC-H Skew, Q18 in IMDb).

Dynamic random workloads. We simulate modern data analytics workloads that are truly ad-hoc in nature. For instance, cloud providers, hosting millions of databases, seldom can detect representative queries, since they frequently change [8]. In such cases, it is common to invoke the PDTool periodically (e.g., nightly or weekly) using queries since the last invocation as the training workload. In this setting, we invoke the PDTool every 4 rounds, using queries from the last 4 rounds as the representative workload. In the dynamic random setting, the number of total training queries in the complete sequence is similar to the number of queries we had in the static setting. However, we have no control over the selection of queries for the workload and they are chosen completely randomly. The sequence is then divided into 25 equal-sized rounds. In all cases, the round-to-round repeat workload was between 45-54%.

As shown in Figure 7, again we see a considerably lower total workload time of MAB compared to PDTool. MAB provides over 37%, 17%, 11%, 61% and 75% speed-up compared to PDTool, under SSB, TPC-H, TPC-H Skew, TPC-DS and IMDb, respectively. It is notable that in Figure 7, the total workload time of PDTool climbs higher than NoIndex on two occasions, in TPC-DS and IMDb. In IMDb, this is due to the same issue discussed previously under dynamic shifting workloads (due to the optimiser's misestimates, favouring the usage of sub-optimal indices, e.g., IMDb Q18). While PDTool has a much better execution time than NoIndex under TPC-DS (execution time of 5.3h under PDTool vs 9.3h under NoIndex), due to high recommendation time (5.1 hours, see Table I), PDTool ends up with a higher total workload time. Under these 2 benchmarks (TPC-DS and IMDb), MAB provides over 55% and 1.5% performance gain over NoIndex, respectively. In Figure 6(a–e), we can see five major spikes for PDTool due to the tuning invocations (in rounds 5, 9, 13, 17, 21).

2) The impact of database size: To examine the impact of database size, we run TPC-H uniform and TPC-H Skew static experiments on SF 1, 10 and 100 databases. As previously discussed, under SF 10, MAB performed better in the case of TPC-H Skew and PDTool performed better on TPC-H (see Table II). The impact of sub-optimal index choices is even more evident for larger databases, leading to a huge gap between total workload times of MAB and PDTool for TPC-H Skew (44 hours in the former vs 20 hours in the latter case). In TPC-H, PDTool results in a higher total workload time (14.8 hours vs. 13.2 hours for MAB). This is mainly due to sub-optimal optimiser decisions, where the optimiser favours the usage of indices (coupled with nested loops joins) when alternative plans would be a better option. For instance, under the recommended indices from PDTool, some instances of Q5 run longer than 8 minutes (using index nested loops join), where others finish in 1.5 minutes (using a plan based on hash joins). We notice that, with larger database sizes, execution time dominates contributing more than 91% to the total workload time. We observe faster and more accurate convergence of MAB under larger databases, due to a clear difference between rewards for different arms, highlighting MAB's excellent potential benefits for larger databases.

3) Hypothetical index creation vs actual index creation: Managing the exploration-exploitation balance under a large number of candidate indices, and an enormous number of combinatorial choices is non trivial. PDTool explores using

TABLE I
TOTAL TIME BREAKDOWN (MIN): THE BEST CHOICE IS IN BOLD TEXT.

Workload		Recommendation		Creation		Execution		Total	
		PDTool (#)	MAB	PDTool	MAB	PDTool	MAB	PDTool	MAB
Static	SSB	0.34 (0.34)	**0.02**	**0.95**	1.86	**12.9**	13.15	**14.19**	15.03
	TPC-H	0.6 (0.6)	**0.08**	**2.45**	5.66	**46.35**	55.64	**49.4**	61.38
	TPC-H Sk.	0.58 (0.58)	**0.11**	**8.37**	19.82	54.17	**32.06**	63.12	**51.99**
	TPC-DS	44.86 (44.86)	**1.53**	**1.45**	5.94	302.63	**242.15**	348.94	**249.62**
	IMDB	0.34 (0.34)	**0.31**	**1.1**	1.3	11.01	**9.42**	12.41	**11.03**
Dynamic	SSB	1.28 (0.32)	**0.05**	**1.5**	2.21	**5.42**	5.69	8.2	**7.95**
	TPC-H	1.55 (0.32)	**0.12**	**9.36**	9.74	26.35	**25.14**	37.25	**35**
	TPC-H Sk.	1.65 (0.41)	**0.16**	**14.98**	20.96	85.49	**21.44**	102.11	**42.56**
	TPC-DS	11.13 (2.78)	**1.66**	**6.08**	16.48	187.08	**155.65**	204.29	**173.79**
	IMDB	3.09 (0.77)	**0.29**	**1.59**	2.24	11.21	**7.93**	15.89	**10.46**
Random	SSB	2.83 (0.57)	**0.02**	**1.77**	2.37	26.59	**16.83**	30.85	**19.22**
	TPC-H	7.55 (1.51)	**0.08**	14.68	**7.06**	84.14	**80.43**	106.37	**87.57**
	TPC-H Sk.	3.3 (0.66)	**0.08**	**31.74**	34.68	48.71	**39.44**	83.75	**74.2**
	TPC-DS	310.22 (62.04)	**1.4**	**8.23**	19.81	323.57	**227.02**	642.01	**248.24**
	IMDB	14.74 (2.94)	**0.28**	2.72	**1.14**	48.55	**14.47**	66.01	**15.89**

The average time of a single PDTool invocation

TABLE II
TOTAL END-TO-END WORKLOAD TIME FOR STATIC WORKLOADS UNDER DIFFERENT DATABASE SIZES (MIN)

Workload	SF	PDTool	MAB
TPC-H	1	**2.02**	2.03
	10	**49.4**	61.38
	100	891.01	**793.40**
TPC-H Skew	1	4.17	**3.83**
	10	63.12	**51.99**
	100	2640.64	**1219.33**

TABLE III
RECOMMENDATION TIMES (MIN) VS. WORKLOAD COMPLEXITY

Workload	SSB	TPC-H	TPC-DS
PDTool	0.84	1.36	44.86
MAB	0.05	0.14	1.53

the "what-if" analysis, which comes under the tool's recommendation time, whereas MAB explores using index creations.

Cost of hypothetical index creation: When analysing PDTool's average invocation times in dynamic shifting (small workloads) and dynamic random (large workloads) settings, it becomes evident that PDTool invocation cost grows noticeably with the training workload size, under all benchmarks (see Table I). As an example, PDTool tuning of the TPC-DS benchmark grows from 3 minutes in the dynamic shifting setting (25-query workload) to 1 hour in the dynamic random setting (400-query workload). Furthermore, multiple invocations required in dynamic random and shifting settings aggravate the problem further for PDTool (see Table I). On the other hand, PDTool recommendation time rapidly increases with the complexity of the workloads. In an experiment with 100 query workloads from SSB, TPC-H and TPC-DS (with the complexity of SSB $<$ TPC-H $<$ TPC-DS), it is evident that the complexity of a workload has a considerable impact on the PDTool recommendation time (see Table III).

MAB recommendation times stay significantly lower and stable despite the workload shifts and changes in complexity or size (see Table I and Table III). In all experiments, MAB takes less than 1% of the total workload time for recommendation, except for IMDb where it takes around 2% (due to low total workload time and a high number of query templates). More than 80% of this recommendation time is spent on the initial setup (1st round) and the continuous overhead is negligible.

Cost of actual index creation: While actual execution statistics based search allows the MAB to converge to better configurations, as a down side, MAB spends more time on index creation (see Table I). For instance, under TPC-H and TPC-H Skew static experiments, MAB spends 5.6 and 19.8 minutes on index creation where PDTool only spends 2.4 and 8.3 minutes, respectively. Under skewed data, rewards show more variability which delays the convergence for MAB. This leads to higher exploration and greater creation costs. While MAB is still competitive due to efficient exploration, we consider ways to improve its convergence in future work.

Final verdict: Comparing the total of recommendation and index creation times (henceforth referred to as *exploration cost*) between MAB and PDTool presents a clear picture about these two exploration methods. From Table I we can observe that, in most cases (9 out of 15) MAB archives a better exploration cost compared to PDTool. However when the workload is small (e.g., dynamic shifting) PDTool tends to perform better. TPC-DS, with the highest number of candidate indices among these benchmarks (over 3200 indices), provides a great test case for exploration efficiency. Under TPC-DS, MAB exploration cost is significantly lower in shifting and random settings, and marginally higher in the static setting. Despite the efficient exploration, MAB does not sacrifice recommendation quality in any way (better execution costs in 12 out of 15 cases, with significantly better execution costs under all cases of TPC-DS).

This efficient exploration is promoted by the linear reward-context relationship along with C^{2}UCB's weight sharing (Section III), resulting in a small number of parameters to learn. An arm's identity becomes irrelevant and context (Section IV) becomes the sole determining factor of each arm's expected score, which allows MAB to predict the UCB of a newly arriving arm with known context *without* trying it even once.

4) The impact of round size: In the original TPC-H Skew static experiment (1x), each bandit round includes all the benchmark templates (22 queries). To analyse the impact of the round size (bandit invocation frequency), we conduct experiments with single-query (1 query), 0.5x (11 queries) and 2x (44 queries) round sizes on TPC-H Skew benchmark. All three round sizes converge to the same performant configurations by the last round. We observe a faster convergence with small round sizes, resulting in lower execution costs in the first few rounds. While the execution cost gain from 1x to 0.5x is noticeable, dividing the round further (single query) does not provide a considerable benefit compared to the added creation and recommendation overhead. With larger round sizes, we observe lower creation costs due to less frequent bandit updates (see Table IV). MAB performs better under all round-sizes compared to PDTool. A DBA can decide on

TABLE IV
TPC-H SKEW BENCHMARK UNDER DIFFERENT ROUND SIZES (MIN)

Round size	Recommendation	Creation	Execution	Total
Single Query	1.11	27.77	30.16	59.04
0.5x	0.13	22.39	30.39	52.92
1x	0.11	19.82	32.06	51.99
2x	0.08	12.66	43.53	56.27

the round size (bandit invocation frequency) based on the application and DBA's primary goal (faster convergence vs low creation cost). We leave auto-tuning of this parameter as an interesting future work avenue.

5) Optimal configurations: With many possible configurations for even the smallest databases, finding an optimal configuration under given memory budget, while considering performance regressions, is a non trivial task. However, when the memory budget limitation is lifted, we can estimate optimal configurations with a set of covering indices. This configuration occupies around 40 GB of space for both TPC-H and TPC-H Skew. Under this budget, PDTool and MAB, both converged to the optimal configuration by the final round.

C. Why Not (General) Reinforcement Learning?

Past efforts have considered more general reinforcement learning (RL) for physical design tuning [25], [36]. Compared to most MAB approaches, deep RL invites over parameterisation, which can slow convergence (see Figure 8), whereas MAB typically provides better convergence, simpler implementation, and *safety guarantees* via strategic exploration and knowledge transfer (see Section III). Due to its randomisation, RL can also suffer from performance volatility as compared to C^2UCB, a deterministic algorithm.

Experimental setup. The above intuition is supported by experiments with more general RL, where we evaluate the popular DDQN RL agent [37]. We run the static 10GB TPC-H and TPC-H Skew benchmark over 100 rounds and present results in Figure 8. For a fair comparison, we combine all of MAB's arms' contexts as DDQN state. We also present the same set of candidate indices to the DDQN. For the DDQN's neural network hyperparameters, we followed the experiment of [25] by setting 4 hidden layers, with 8 neurons each. The discount factor γ is set to 0.99 and the exploration parameter is set to 1 at the first sample, decaying to 0 with exponential rate reaching 0.01 in the 2400^{th} sample. One sample corresponds to one index chosen by the agent. In the beginning of the round, if the agent decides to explore, then the choice of the set of indices will be randomly made for that entire round. These experiments are repeated ten times, reporting either average value (Figure 8 (a) and (b)) or median ((c) and (d)) along with inter-quartile range. For completeness, we include the case of only using single column indices (DDQN-SC in Figure 8), as originally proposed in [25].

Evaluation. Due to DDQN-SC's reduced search space in some scenarios, it might not be possible to find an optimal configuration for a workload. This is evident from Figure 8 (a) and (b) where DDQN shows 33% and 21% speedup, compared to DDQN-SC, in execution time under TPC-H and TPC-H Skew, respectively. Interestingly, under TPCH-Skew, where the demand for exploration is higher, DDQN-SC has a lower total workload time than DDQN due to the noticeably small index creation times of single column indices (1.5 hours vs 5.8 hours, respectively). Under both TPC-H and TPC-H Skew, MAB performs significantly better, providing 35% and 58% speed-up against the better RL alternative, respectively.

No state transitions. A strength of more general reinforcement learning is its ability to take into account (random) state transitions when actions are taken. However, the importance of state transition in online index selection is unclear. While modelled state could include the collection of indices that exist in the system, actions (i.e., choosing an index) deterministically govern the ensuing state. State could also model characteristics of the next round's queries. However these queries do not depend on the prior action, thus it is appropriate to take successor query state as independent of action, as promoted by bandits. Hence, adopting more general RL provides no clear benefit over MAB, while imposing delay to convergence as demonstrated in Figure 8 (c) and (d), and passing over MAB-style performance guarantees (see Section III).

Hyperparameter search space. Deep RL is notorious for challenging hyperparameter tuning. For example, in this experiment, we have to decide: the number of layers of the neural network, the neurons per layer, activation functions and loss, the exploration parameter ε, and discount factor γ. C^2UCB has just λ—which becomes less relevant as rounds are observed—and α which controls exploration.

Volatility of deep RL. Most deep RL algorithms randomise to explore vast state-action spaces. This is not the case with C^2UCB. Extending UCB, deterministic C^2UCB is capable of identifying underexplored arms through their context vectors. The only (rare and as such not strictly necessary) case when C^2UCB is random is where the MAB must tie-break arms. A more significant cause of stability of our MAB is its small parametrisation compared to deep learner-based RL. Combined, the stable MAB yields a more consistent result, as can be seen in Figure 8(a) and (b). Much wider variance on the DDQN plot demonstrates how the performance of DDQN can vary significantly, compared to the narrow error bars on the MAB, which demonstrates the algorithm's stability.

VI. RELATED WORK

Automated physical design tuning. Most commercial DBMS vendors nowadays offer physical design tools in their products [1]–[3]. These tools rely heavily on the query optimiser to compare benefits of different design structures without materialisation [18]. Such an approach is ineffective when base data statistics are unavailable, skewed, or change dynamically [10]. In these dynamic environments, the problem of physical design is aggravated: a) deciding *when* to call a tuning process is not straightforward; and b) deciding *what* is a representative training workload is a challenge.

Online physical design tuning. Several research groups have recognised these problems and have offered lightweight solutions to physical design tuning [11]–[14]. While such

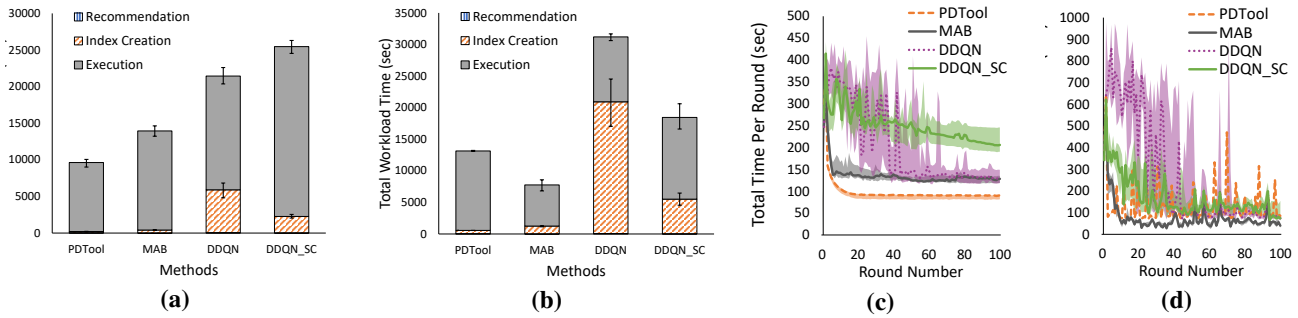

Fig. 8. DDQN vs. MAB for static workloads: (a) End-to-end workload time for TPC-H, (b) End-to-end workload time for TPC-H Skew, (c) TPC-H convergence, (d) TPC-H Skew convergence.

solutions are more flexible and need not know the workload in advance, they are typically limited in terms of applicability to new unknown workloads (generalisation beyond past), and do not come with theoretical guarantees that extend to actual runtime conditions. Moreover, by giving the optimiser a central role, the tools remain susceptible to its mistakes [9]. [8] extends [1] with the use of additional components, in a narrowed scope of index selection to mimic an online tool. This takes corrective actions against the optimiser mistakes through a validation process.

Adaptive and learning indices. Another dimension of online physical design tuning is database cracking and adaptive indexing that smooth the creation cost of indices by piggybacking on query execution [38], [39]. Recent efforts have gone a step further and proposed replacing data structures with learned models that are smaller in size and faster to query [40]–[42]. Such approaches are complementary to our efforts: once the data structures (or models) are materialised inside a DBMS, the MAB framework can be used to automate the decision making as to which data structure should be used to speed-up query analysis.

Learning approaches to optimisation and tuning. Recent years have witnessed new machine learning approaches to automate decision-making processes within databases. For instance, reinforcement learning approaches have been used for query optimisation and join ordering [43]–[46]. In [9], regression has been used to successfully mitigate the optimiser's cost misestimates as a path toward more robust index selection. [9] shows promising results when avoiding query regressions. However, this classifier incurs up to 10% recommendation time, impacting recommendation cost in all cases, especially where recommendation cost already dominates the cost for PDTool (e.g., TPC-DS, IMDb).

When it comes to tuning, the closest approaches employ variants of RL for index selection or partitioning [25], [36], [47] or configuration tuning [5]. [36] describes RL-based index selection, which depends solely on the recommendation tool for query-level recommendations and is affected by decision combinatorial explosion, both issues addressed in our work. Unlike its more general counterpart (RL), MABs have advantages of faster convergences as demonstrated in Section V-C, simple implementation, and theoretical guarantees. There has been recent interest in using bandits for database tasks such as monitoring, query optimisation and join ordering [48]–[50].

VII. Discussion and Future Avenues

This paper scratches the surface of the numerous opportunities for applying bandit learners to performance tuning of databases. We now discuss a rich research vision for the area.

Multi-tenant and HTAP environments. A crucial advantage of the MAB setting is theoretical guarantees on the fitness of proposed indices to observed run-time conditions. This is critical for production systems in the cloud and multi-tenant environments [7]–[9], where analytical modelling is impossible due to unpredictable changes in run-time conditions. Similar requirements hold for hybrid OLTP/OLAP processing environments (HTAP) where the presence of transactions hinders the usefulness of indices, making analytical modelling next to impossible. The MAB approach on the contrary eschews the optimiser and modelling completely, choosing indices based on observed query performance and is thus equally applicable to these challenging environments.

Beyond index choices. Despite focusing solely on the task of index selection in this paper, the MAB framework is equally applicable to other physical design choices, such as materialised views selection, statistics collection, or even selection of design structures that are a mix of traditional and approximate data structures, such as learned models [40] or other fine-grained design primitives [42]. Furthermore, MABs can be used in other areas in databases which require strategic exploration under theoretical guarantees like dynamic memory allocation, query optimisation, and database monitoring.

Cold-start problem. Under the current setup, MAB starts without any secondary indices or knowledge about their benefits, forming a cold-start problem and leading to higher creation costs. While MAB is already superior against PDTool even with the creation cost burden (see Section V-B3), even faster convergence and better creation costs can be provided by pre-training models in hypothetical rounds (using what-if) or workload forecasting [15] to improve context quality.

Opportunities for bandit learning. The increased search space of possible design choices calls for advancements to bandits algorithms and theory, where unbounded/infinite numbers of arms will be increasingly important. Similarly, various flavours of physical design might ask for novel bandits that adopt the notion of heterogeneous arms (indices, views, or

statistics), or hierarchical models where individual choices at lower levels (e.g., the choice of indices or materialised views) influence decisions at a higher level (e.g., index merging due to memory constraints).

VIII. Conclusions

This paper develops a multi-armed bandit learning framework for online index selection. Benefits include eschewing the DBA and the (error-prone) query optimiser by learning the benefits of indices through strategic exploration and observation. We justify our choice of MAB over general reinforcement learning for online index tuning, comparing MAB against DDQN, a popular RL algorithm based on deep neural networks, demonstrating significantly faster convergence of the MAB. Furthermore, our extensive experimental evaluation demonstrates advantages of MAB over an existing commercial physical design tool (up to 75% speed up, and 23% on average), and exemplifies robustness to data skew and unpredictable ad-hoc workloads.

References

[1] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala, "Database tuning advisor for Microsoft SQL Server 2005," in *VLDB*, 2004.

[2] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden, "DB2 design advisor: Integrated automatic physical database design," in *VLDB*, 2004.

[3] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin, "Automatic SQL tuning in oracle 10g," in *VLDB*, 2004.

[4] D. Zilio, S. Lightstone, K. Lyons, and G. Lohman, "Self-managing technology in IBM DB2 universal database," in *ACM CIKM*, 2001.

[5] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah *et al.*, "Self-driving database management systems," in *CIDR*, 2017.

[6] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich, "Relational cloud: a database service for the cloud," in *CIDR*, 2011.

[7] V. R. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri, "SQLVM: Performance isolation in multi-tenant relational database-as-a-service," in *CIDR*, 2013.

[8] 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.

[9] 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.

[10] S. Chaudhuri and V. Narasayya, "Self-tuning database systems: A decade of progress," in *VLDB*, 2007.

[11] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis, "On-Line Index Selection for Shifting Workloads," in *ICDEW*, 2007.

[12] K.-U. Sattler, E. Schallehn, and I. Geist, "Autonomous query-driven index tuning," in *IDEAS*, 2004.

[13] N. Bruno and S. Chaudhuri, "An Online Approach to Physical Design Tuning," in *ICDE*, 2007.

[14] ——, "To tune or not to tune?: A lightweight physical design alerter," in *VLDB*, 2006.

[15] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based workload forecasting for self-driving database management systems," in *SIGMOD*, 2018.

[16] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou, "The researcher's guide to the data deluge: Querying a scientific database in just a few seconds," *VLDB Endow.*, 2011.

[17] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "NoDB: Efficient query execution on raw data files," in *SIGMOD*, 2012.

[18] S. Chaudhuri and V. Narasayya, "AutoAdmin "what-if"; index analysis utility," in *SIGMOD*, 1998.

[19] S. Christodoulakis, "Implications of certain assumptions in database performance evaluation," *TODS*, 1984.

[20] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *PVLDB*, 2015.

[21] A. Aboulnaga and S. Chaudhuri, "Self-tuning histograms: Building histograms without looking at data," *ACM SIGMOD Record*, vol. 28, no. 2, pp. 181–192, 1999.

[22] R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski, and C. Fraser, "Smooth scan: Robust access path selection without cardinality estimation," *The VLDB Journal*, 2018.

[23] R. Borovica, I. Alagiannis, and A. Ailamaki, "Automated physical designers: What you see is (not) what you get," in *DBTest*, 2012.

[24] K. E. Gebaly and A. Aboulnaga, "Robustness in automatic physical database design," in *EDBT*, 2008.

[25] A. Sharma, F. M. Schuhknecht, and J. Dittrich, "The case for automatic database administration using deep reinforcement learning," 2018, unpublished.

[26] L. Qin, S. Chen, and X. Zhu, "Contextual combinatorial bandit and its application on diversified online recommendation," in *SDM*, 2014.

[27] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *WWW*, 2010.

[28] B. Oetomo, M. Perera, R. Borovica-Gajic, and B. I. P. Rubinstein, "A note on bounding regret of the C^2UCB contextual combinatorial bandit," *arXiv preprint arXiv:1902.07500*, 2019.

[29] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions–i," *Mathematical programming*, 1978.

[30] J. Kossmann, S. Halfpap, M. Jankrift, and R. Schlosser, "Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2382–2395, 2020.

[31] TPC, "TPC-H benchmark," http://www.tpc.org/tpch/.

[32] Microsoft, "TPC-H skew benchmark," https://www.microsoft.com/en-us/download/details.aspx?id=52430.

[33] R. O. Nambiar and M. Poess, "The making of tpc-ds," in *VLDB*, 2006.

[34] P. O. Neil, B. O. Neil, and X. Chen, "Star schema benchmark," 2009, unpublished.

[35] S. Chaudhuri and V. Narasayya, "Index merging," in *ICDE*, 1999.

[36] D. Basu, Q. Lin, W. Chen, H. T. Vo, Z. Yuan, P. Senellart, and S. Bressan, "Regularized cost-model oblivious database tuning with reinforcement learning," in *TLDKS XXVIII*, 2016.

[37] H. v. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *AAAI*, 2016.

[38] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking," in *CIDR*, 2007.

[39] G. Graefe and H. Kuno, "Self-Selecting, Self-Tuning, Incrementally Optimized Indexes," in *EDBT*, 2010.

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

[41] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *SIGMOD*, 2019.

[42] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo, "The data calculator: Data structure design and cost synthesis from first principles and learned cost models," in *SIGMOD*, 2018.

[43] T. Kaftan, M. Balazinska, A. Cheung, and J. Gehrke, "Cuttlefish: A lightweight primitive for adaptive query processing," 2018, unpublished.

[44] I. Trummer, S. Moseley, D. Maram, S. Jo, and J. Antonakakis, "SkinnerDB: Regret-bounded query evaluation via reinforcement learning," *PVLDB*, 2018.

[45] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper, "Learned cardinalities: Estimating correlated joins with deep learning," in *CIDR*, 2019.

[46] R. Marcus and O. Papaemmanouil, "Towards a hands-free query optimizer through deep learning," in *CIDR*, 2019.

[47] B. Hilprecht, C. Binnig, and U. Röhm, "Towards learning a partitioning advisor with deep reinforcement learning," in *aiDM*, 2019.

[48] H. Grushka-Cohen, O. Biller, O. Sofer, L. Rokach, and B. Shapira, "Using bandits for effective database activity monitoring," in *PAKDD*. Springer, 2020, pp. 701–713.

[49] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in *SIGMOD*, 2020.

[50] V. Ghadakchi, M. Xie, and A. Termehchy, "Bandit join: preliminary results," in *aiDM-SIGMOD 20*, 2020, pp. 1–4.