UDO: Universal Database Optimization using Reinforcement Learning

Junxiong Wang
Cornell University
Ithaca, NY, USA
junxiong@cs.cornell.edu

Immanuel Trummer
Cornell University
Ithaca, NY, USA
itrummer@cornell.edu

Debabrota Basu
Scool, Inria Lille- Nord Europe
Lille, France
debabrota.basu@inria.fr

ABSTRACT

UDO is a versatile tool for offline tuning of database systems for specific workloads. UDO can consider a variety of tuning choices, reaching from picking transaction code variants over index selections up to database system parameter tuning. UDO uses reinforcement learning to converge to near-optimal configurations, creating and evaluating different configurations via actual query executions (instead of relying on simplifying cost models). To cater to different parameter types, UDO distinguishes heavy parameters (which are expensive to change, e.g. physical design parameters) from light parameters. Specifically for optimizing heavy parameters, UDO uses reinforcement learning algorithms that allow delaying the point at which the reward feedback becomes available. This gives us the freedom to optimize the point in time and the order in which different configurations are created and evaluated (by benchmarking a workload sample). UDO uses a cost-based planner to minimize reconfiguration overheads. For instance, it aims to amortize the creation of expensive data structures by consecutively evaluating configurations using them. We evaluate UDO on Postgres as well as MySQL and on TPC-H as well as TPC-C, optimizing a variety of light and heavy parameters concurrently.

PVLDB Reference Format:
Junxiong Wang, Immanuel Trummer, and Debabrota Basu. UDO: Universal Database Optimization using Reinforcement Learning. PVLDB, 14(13): 3402 - 3414, 2021.
doi:10.14778/3484224.3484236

1 INTRODUCTION

We introduce *UDO*, the *Universal Database Optimizer*. UDO is an offline tuning tool that optimizes various kinds of tuning choices (e.g., physical design decisions as well as settings for database system configuration parameters), given an example workload and a tuning time limit. UDO does not rely on simplifying cost models to assess the quality of tuning options. Also, it does not require any kind of training data upfront. Instead, it relies only on feedback obtained via sample runs, after creating a tuning configuration to evaluate. This makes the optimization process expensive but avoids sub-optimal choices due to erroneous cost estimates, which are otherwise common [8].

Proceedings of the VLDB Endowment, Vol. 14, No. 13 ISSN 2150-8097.
doi:10.14778/3484224.3484236

Given the tradeoff realized by UDO (i.e., high-quality, high-overhead optimization), we see two primary use cases. First, UDO is useful in scenarios where a configuration, obtained via expensive optimization, can be used over extended periods. This is possible if data and query workload properties do not change too frequently. Also, UDO is useful as an analysis tool for other tuning approaches. For instance, as UDO does not rely on cost or cardinality models, it can be used to uncover weaknesses in other recommender tools that are based on the latter. In this scenario, UDO adopts a similar role as previously proposed methods for query optimizer testing [11, 30], which generate guaranteed optimal plans via an expensive process (but are specific to query plans, as opposed to other tuning choices).

UDO operates on various types of tuning parameters, which are traditionally handled by separate tuning tools. For instance, in our experiments, we consider optimization of transaction query orders [35], index selections [10, 14], as well as database system configuration parameters [37, 38]. Considering various parameter types together can be advantageous as optimal choices for one parameter type may depend on settings for other parameters (e.g., we may disable sequential scans, a configuration parameter, only if specific indexes are created). Hereby, we use the generic term **Parameter** for each tuning choice and the term **Configuration** for an assignment from parameters to values. UDO handles all parameters by a unified approach.

UDO explores the search space iteratively: selecting configurations to try, creating them (e.g., creating index structures or setting system parameters as specified by the configuration), and evaluating their performance on a workload sample. Evaluation is flexible to incorporate multiple metrics such as throughput or latency. We demonstrate optimization with both metrics on different database systems (Postgres and MySQL) and standard benchmarks (TPC-C and TPC-H). UDO uses *Reinforcement Learning (RL)* to determine which configurations to try next. Improvement in performance measurements translate into reward values that guide an RL agent during search towards actions, i.e. configurations, that maximize the accumulated reward, i.e. the resulting performance.

RL has been used previously for optimizing database system configuration parameters [23, 38] in particular. The main novelty of UDO lies in the fact that it broadens the scope of optimization to a much larger class of parameters. This becomes particularly challenging due to what we call **Heavy Parameters** (we distinguish them from **Light Parameters** in the following). For heavy parameters, it is expensive to change the parameter value. For instance, parameters that relate to index creations are expensive to change. Creating an index, in particular a clustered index, may take an amount of time that dominates query or transaction evaluation

time for a small workload sample. Similarly, configuration parameters requiring a database server restart are relatively expensive to change. As we show in our experiments, a naïve RL approach is limited by costs of changing heavy parameters. This incurs high costs per iteration and slows down convergence.

UDO avoids this pitfall by giving heavy parameters special treatment. *UDO separates heavy parameters from light ones and uses different reinforcement learning algorithms to optimize them.* Specifically for heavy parameters, it uses an RL algorithm that can adjust with delays until reward values for previous choices become available. We leverage such delayed feedbacks as follows. All configurations selected by the RL algorithm are forwarded to a *planning component*. The planning component decides, when and in which order to create and to evaluate configurations. Depending on those choices, we are able to amortize cost for changing heavy parameters over the evaluation of many similar configurations. E.g., it allows us to create an expensive index once to evaluate multiple similar configurations that all include the index. Alternating between configurations that use or do not use the index, requiring multiple index creations and drops, is less efficient.

Given settings for heavy parameters, we use RL again to find optimal settings for light parameters. Of course, optimal settings for light parameters depend on the values for heavy parameters. UDO takes that into account and models the optimization of light parameters for each heavy parameter setting as a separate Markov Decision Process (MDP), to which an RL algorithm is applied to. In contrast to heavy parameters, we use a no-delay RL algorithm to converge faster to near-optimal settings for light parameters.

We propose a new Monte Carlo Tree Search (MCTS) variant, called delayed-Hierarchical Optimistic Optimization (HOO), that can be used for optimizing both, heavy and light parameters (with and without delays). Using this approach, we show that UDO converges to near-optimal configurations.

We demonstrate via experiments that the resulting system finds better configurations, compared to baselines, given the same amount of optimization time. We consider multiple standard benchmarks (TPC-H and TPC-C), multiple optimization metrics (throughput and latency), as well as different database management systems (Postgres and MySQL). In summary, our original, scientific contributions are the following.

- We introduce an approach for optimizing various database tuning decisions using reinforcement learning. This approach is characterized by a factorisation of heavy and light parameters, the use of RL algorithms accepting delayed feedback, and a planner component that reduces re-configuration overheads by carefully planning evaluation orders.
- We experimentally demonstrate that UDO finds better configurations than baselines, given the same amount of optimization time. Our experiments cover various benchmarks and metrics.
- We propose a new MCTS variant, delayed-HOO, that can be used to optimize light and heavy parameters. We show that UDO converges to near-optimal solutions, using that approach, under moderately simplifying assumptions.

The remainder of this paper is organized as follows. First, in Section 2, we introduce our formal problem model and terminology used throughout the paper. Then, in Section 3, we give a high-level overview of the UDO system. We analyze UDO formally in Section 6. In Section 4, we describe mechanisms by which UDO evaluates batches of configurations efficiently. In Section 5, we introduce UDO's learning algorithms and analyze them formally in Section 6. Then, in Section 7, we report results of our experimental evaluation. We discuss related work in Section 8 before concluding the paper.

2 FORMAL MODEL

We introduce our problem model and associated terminology here.

Definition 2.1. A **Tuning Parameter** represents an atomic decision, influencing performance of a database management system for a specific workload. It is associated with a (discrete) **Value Domain**, representing admissible parameter values. It may be subject to **Constraints**, restricting its values based on the values of other tuning parameters.

We use the term "parameter" in a broad sense, encompassing system configuration parameter settings as well physical design decisions. In the following, we give examples for tuning parameters.

Example 2.2. Considering a set of candidate indices for a given database, we associate one tuning parameter with each candidate. Such index-related parameters have a binary value domain, representing whether the index is created or not. Equally, we can introduce a tuning parameter to represent the `random_page_cost` configuration parameter of the Postgres system (together with a set of values to consider). Finally, we may associate a query in a transaction template with a tuning parameter, representing the position within the template at which it is evaluated (the set of admissible positions is restricted via control flow and data dependencies).

Definition 2.3. Given fixed, ordered parameters, a **Configuration** c is a vector, assigning a specific value to each parameter. The **Configuration Space** C is the set of all possible configurations.

Our goal is to find configurations that optimize a benchmark.

Definition 2.4. A **Benchmark Metric** f maps a configuration $c \in C$ to a real-valued performance result (i.e., $f : C \mapsto \mathbb{R}$), which represents the performance of a configuration according to a specific metric for a specific benchmark. Higher performance results are preferable. We assume that f is stochastic (i.e., evaluating the same configuration twice may not yield exactly the same performance).

Our definition of f is deliberately generic, covering different types of benchmarks and metrics. A few examples follow.

Example 2.5. In our experiments, we use the following two benchmark metrics among others. We consider a benchmark metric f_1 that maps configurations to the average throughput, measured over a fixed time period, when processing TPC-C transactions generated randomly according to a fixed distribution. Also, we consider a benchmark metric f_2 that maps configurations to a weighted sum between disk space d consumed (e.g., for created indexes) and run time t of all TPC-H queries (i.e., $f(c) = -d - \sigma \cdot t$ where c is a configuration and $\sigma \in \mathbb{R}^+$ a user-defined scaling factor). Both benchmark metrics are implemented as a script (a black box from UDO's perspective) that returns a numerical performance result.

We present a system, UDO, that solves the following problem.

Definition 2.6. An instance of **Universal Database Optimization** is characterized by a benchmark metric f and a configuration space C. The goal is to find an optimal configuration c^*, maximizing the stochastic benchmark metric f in expectation (i.e., $c^* = \text{argmax}_{c \in C} \mathbb{E}[f(c)]$). We denote the optimal expected performance (that of c^*) as f^*.

For iterative approaches, the problem specification may also include a user-defined timeout for optimization. The qualification "Universal" attest to the fact that our approach is broadly applicable, in terms of parameter types, workloads, and performance metrics. We map an UDO instance to multiple episodic Markov Decision Processes, using the following definition, and solve UDO using RL.

Definition 2.7. An **Episodic Markov Decision Process** (MDP) is defined by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{S}_D, \mathcal{S}_E \rangle$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ a set of actions, and $\mathcal{T} : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$ a transition function linking state-action pairs to new states. $\mathcal{R} : \mathcal{S} \mapsto \mathbb{R}$ is a reward function mapping states to a reward value. We consider deterministic transitions but stochastic rewards. Optimization models an agent that performs steps. In each step, the agent selects an action, receives a reward, and transitions to the next state, based on the selected action. Optimization is divided into episodes. In each episode, the agent starts in state $\mathcal{S}_D \in \mathcal{S}$. The episode ends once it reaches one of the end states $\mathcal{S}_E \subseteq \mathcal{S}$. The goal is to find a policy (here: a sequence of actions as we consider deterministic transitions) that maximizes expected rewards per step.

We introduce two scenario-specific instances of this formalism, associated with different parameter types.

Definition 2.8. We distinguish **Heavy** and **Light Parameters**, based on the overheads associated with changing their values. Heavy parameters have high reconfiguration overheads, light parameters have negligible overheads. We denote by C_H the configuration space for heavy parameters (i.e., a set of vectors representing all possible heavy parameter settings). We denote by C_L the configuration space for light parameters. Hence, the entire configuration space C can be written as $C = \{c_H \circ c_L | c_H \in C_H, c_L \in C_L\} = C_H \times C_L$ (assuming that heavy parameters are ordered before light parameters and writing vector concatenation as $\circ$).

Currently, UDO considers all parameters representing physical data structures such as indexes as heavy (as changing such parameters means creating or dropping the associated data structure). Also, UDO considers parameters as heavy that require a database server restart to make value changes effective. The other parameters are considered light.

Definition 2.9. For the **Heavy Parameter MDP**, states correspond to configurations for heavy parameters (i.e., $\mathcal{S} \subseteq C_H$) and each action changes one heavy parameter to a new value (i.e., an action is defined as a pair $\langle p, v \rangle$ representing parameter p and new value v). The transition function maps a configuration (i.e., state) with a parameter value change (i.e., action) to a new configuration, reflecting the changed value. The start state $\mathcal{S}_D \in C_H$ represents the default configuration (i.e., no created indices and default values for all system parameters). All states reachable from the start state with a given number of actions are end states (we typically use a threshold of four actions). The reward function $\mathcal{R}$ is scenario-specific and based on the benchmark metric f. The reward for a state representing heavy parameter configuration c_H is proportional to $\text{argmax}_{c_L \in C_L} f(c_H \circ c_L)$, i.e. to the value of the benchmark metric when combining c_H with the best possible configuration c_L for light parameters. We scale raw rewards by subtracting rewards for the default configuration (e.g., if f measures throughput for a specific benchmark, UDO considers the throughput improvement compared to default settings as reward function).

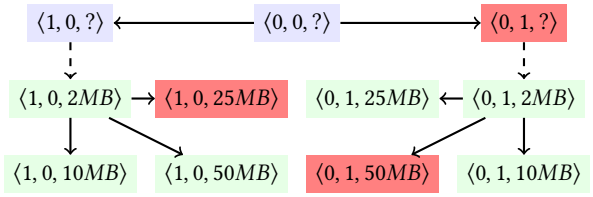


Figure 1: Extract of heavy (top) and light (bottom) parameter MDPs for a space with two heavy and one light parameter. Optimal states for each MDP are marked up in red.

The definition above uses optimal configurations for light parameters, leading to the second MDP version.

Definition 2.10. A **Light Parameter MDP** $\mathcal{M}_L[c_h]$ is introduced for each heavy parameter configuration c_h (in practice, we limit ourselves to configurations explored by UDO). Its states represent configurations for light parameters, its actions represent value changes for light parameters (analogue to the previous definition). The start state represents default values for all light parameters and end states are defined by a fixed number of light parameter changes, compared to the default. The reward function $\mathcal{R}_L[c_H]$ is defined as $\mathcal{R}_L[c_H](c_L) = f(c_H \circ c_L) - f_D$ where f_D is performance of the default configuration.

As shown above, we model optimization for light parameters as a family of MDPs, where each MDP is associated with a specific heavy parameter configuration. Our problem model assumes that only the reward function (i.e., performance) depends on heavy parameters. It is possible to extend the definition above to cover cases where admissible values for light parameters depend on currently chosen heavy parameters. In that case, states, transitions, and actions must be instantiated for specific heavy parameter settings as well. The following example illustrates the interplay between heavy and light parameter MDPs.

Example 2.11. Figure 1 illustrates part of a two-level MDP. In the illustrated scenario, the configuration space contains two heavy parameters (e.g., index creation decisions) as well as one light parameter (e.g., the maximal amount of main memory used per operator). Rectangles represent states in the figure and are annotated with configuration vectors (reporting parameter in the aforementioned order). The upper part of the figure illustrates the heavy parameter MDP (note that the light parameter is not specified). Solid lines mark transitions due to actions changing the configuration. Dashed

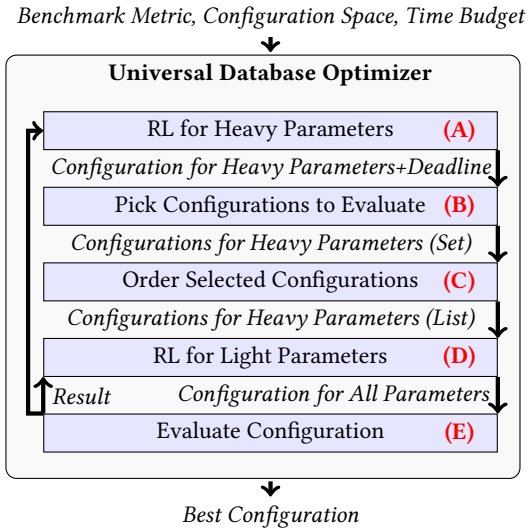


Figure 2: Overview of UDO system (rectangles represent processing steps, arrows represent data flow).

lines mark mappings between heavy parameter states and the start state of the associated light parameter MDP. As index-related parameters are binary, the heavy parameter MDP has four states (out of which three are shown). In total, the figure illustrates three MDPs (the one for heavy parameters and light parameter MDPs for two heavy configurations). Optimal states are marked up in red, showing that the optimal settings for light parameters may depend on the heavy parameter settings. Identifying optimal settings for heavy parameters requires obtaining optimal, associated settings for light parameters first.

3 SYSTEM OVERVIEW

Figure 2 shows an overview of UDO and the interplay between its components. The input to UDO is a benchmark metric to optimize, a configuration space, and an optimization time budget. The configuration space is specified as a set of index candidates to consider, a set of database system parameters with alternative values to try, and (optionally) a set of alternative versions for each query or transaction template. UDO considers index parameters as heavy and the others as light. The output is the best configuration found until the time limit.

UDO iterates until the time limit is reached[1]. In each iteration, UDO first chooses a configuration of heavy parameters to explore (Component A). UDO uses reinforcement learning for this decision, balancing the need for exploration (analyzing configurations about which little information is available) and exploitation (refining configurations that seem to well) in a principled manner. Evaluating a new configuration for heavy parameters can however be expensive. It involves changing the current database configuration to the one to evaluate, e.g. by creating indexes. Doing so becomes cheaper if the current configuration is close to the one to evaluate. Hence, UDO tries to optimize the point in time at which heavy parameter configurations are evaluated. UDO uses a specialized reinforcement learning algorithm that does not expect evaluation results immediately after selecting a configuration. Instead, it allows for a certain delay (measured as the number of iterations between selection and evaluation result). Selected configurations for heavy parameters are added to a buffer, associated with a deadline until which the result must become available. Note that the learning algorithm does not consider the current database state for *deciding which configuration to explore* (doing so may prevent UDO from finding promising configurations that are far from the current one). Instead, it merely creates opportunities for cost reductions by other system components.

In each iteration, UDO selects a set of heavy parameter configurations to evaluate from the aforementioned buffer (Component B). Configurations are selected if either their deadline has been reached (in this case, there is no choice) or if their evaluation is cheaper than usual (e.g., because they share indexes with configurations that must be evaluated). Selected configurations are ordered for evaluation (Component C). The goal of evaluation is to reduce reconfiguration cost by placing similar configurations consecutively. For instance, if configurations with similar indexes are evaluated consecutively, some index creation cost can be amortized.

Next, UDO selects values for light parameters (Component D). The best configuration for light parameters may depend on the heavy parameter configuration. For instance, we may want to enable or disable specific join algorithms (by setting parameters such as enable_nestloop for Postgres), depending on which indexes are available. For specific configurations of heavy parameters, UDO learns suitable settings for light parameters via reinforcement learning. Here, reconfiguration is cheap. Hence, UDO uses a standard reinforcement learning algorithm without delays. Light parameters are optimized for the current heavy configuration for a fixed number of iterations of the latter learning algorithm. Note that statistics for light parameters are saved and will be used as starting point if the same heavy configuration is selected again. A fully specified configuration (i.e., for light and heavy parameters) is evaluated via the benchmark metric. This involves executing a script that executes a sample workload and returns the performance metric to optimize. The evaluation results are used to update the statistics of the two learning algorithms (Components D and A).

Algorithm 1 describes the main loop, executed by UDO, in more formal detail. Beyond benchmark metric and configuration space, it obtains two parameters specifying hyper-parameters for the two reinforcement learning algorithms used. These include optimization time as well as other parameters (e.g., the maximal amount of allowed delay) whose impact we analyze in Section 7. Users only need to specify optimization time while defaults are available for the other algorithm parameters (hence, Figure 2 only references the former parameter).

Algorithm 1 first classifies parameters as heavy or light (Line 5). We use a simple heuristic and classify parameters requiring index creations or database server restarts as heavy, the other ones as light. Next, Algorithm 1 iterates until optimization time runs out. It selects interesting heavy parameter configurations to evaluate via reinforcement learning (Line 9). It submits requests for evaluation, setting a deadline until which the result must be available (Line 11).

[1]Instead of a fixed optimization time budget, we could use other termination criteria as well. For instance, the algorithm could terminate once a given number of iterations does not yield improvements above a configurable threshold.

Algorithm 1 UDO main function.

```
1: Input: Benchmark metric f, configuration space C, RL algorithms
   Alg_H and Alg_L for heavy and light parameter optimization
2: Output: a suggested configuration for best performance
3: function UDO(f, C, Alg_H, Alg_L)
4:     // Divide into heavy (C_H) and light (C_L) parameters
5:     ⟨C_H, C_L⟩ ← SSA.SplitParameters(C)
6:     // Until optimization time runs out
7:     for t ← 1, ..., Alg_H.Time do
8:         // Select next heavy parameter configuration
9:         c_{H,t} ←RL.Select(Alg_H, C_H, c_{H,t−1})
10:        // Submit configuration for evaluation
11:        EVAL.Submit(c_{H,t}, t + Alg_H.maxDelay)
12:        // Receive newly evaluated light configurations
13:        E ←EVAL.Receive(Alg_L, f, C_L, t)
14:        // Update statistics for heavy parameters
15:        RL.Update(Alg_H, E)
16:    end for
17:    return best obtained configuration
18: end function
```

It receives evaluation results for previously submitted requests (potentially, but not necessarily, including the one submitted in the current iteration). The results are used to update the statistics for learning (Line 15). Finally, the best configuration is returned.

We discuss the sub-functions related to evaluating configurations (Functions EVAL.Submit and EVAL.Receive) in Section 4. In Section 5, we discuss the learning algorithms used (Functions RL.Select and RL.Update).

4 EVALUATING CONFIGURATIONS

We discuss how configurations are selected and ordered for evaluation. In Section 4.1, we describe the implementation of the evaluation functions invoked by Algorithm 1. In Section 4.2, we describe how configurations to evaluate are selected (Component B in Figure 2). In Section 4.3, we discuss the method used to order configurations to evaluate for minimal cost.

4.1 Evaluation Overview

The evaluation interface offers two functions, represented in Algorithm 2 (and used in Algorithm 1). First, it accepts evaluation requests (EVAL.Submit), allowing to submit configurations for evaluation, together with an evaluation deadline. Second, it allows triggering evaluations via the EVAL.Receive function. Algorithm 2 maintains a global variable R (whose state persists across different calls to the two interface functions). This variable contains pending requests for evaluating specific configurations. Each configuration to evaluate is only partially specified (i.e., it assigns values to a subset of configuration parameters). More precisely, configurations to evaluate only contain specific values for heavy parameters. During evaluation, we learn suitable values for light parameters to accurately assess the potential of the heavy parameter settings. Items in R correspond to tuples, combining a heavy parameter configurations with an evaluation deadline. This deadline specifies the latest possible time (measured as the number of main loop iterations, as per Algorithm 1) at which evaluation results must be generated.

Algorithm 2 EVAL: Functions for evaluating configurations.

```
1: // Global variable representing evaluation requests
2: R ← ∅
3: Input: heavy configuration c_H to evaluate and time t
4: Effect: adds new evaluation request
5: procedure EVAL.Submit(c_H, t)
6:     R ← R ∪ {⟨c_H, t⟩}
7: end procedure
8: Input: RL algorithm Alg_L, benchmark metric f, time t, and space C_L
9: Output: evaluated configurations with reward values
10: function EVAL.Receive(Alg_L, f, C_L, t)
11:     // Choose configurations from R to evaluate now
12:     N ←PickConf(R, t)
13:     // Remove from pending requests
14:     R ← R \ N
15:     // Prepare evaluation plan
16:     P ←PlanConf(N)
17:     // Collect evaluation results by executing plan
18:     E ← ∅
19:     for s ∈ P.steps do
20:         // Prepare evaluation of next configurations
21:         ChangeConfig(s.hconf)
22:         // Find (near-)optimal light parameter settings
23:         c_L ←RL.Optimize(Alg_L, s.hconf, C_L, f)
24:         // Take performance measurements on benchmark
25:         b ←Evaluate(f, s.hconf, c_L)
26:         // Add performance result to set
27:         E ← E ∪ {⟨c_L, s.hconf, b⟩}
28:     end for
29:     // Return evaluation results
30:     return E
31: end function
```

The submission function (Procedure EVAL.Submit) simply adds one more tuple to set variable R.

Calling Function EVAL.Receive triggers evaluation of a subset of pending configurations. It is up to that function itself to choose, within certain boundaries, the set of configurations to evaluate. The function returns the results of those evaluations. As input, Function EVAL.Receive obtains a configuration parameter Alg_L, specifying the algorithm to use for optimizing light parameters (for fixed heavy parameter values). Also, it receives the benchmark metric f, the space of light configurations C_L, and the current time t as input. The latter is important to decide which configurations must be evaluated in the current invocation.

As a first step (Line 12), Function EVAL.Receive determines the set of configurations to evaluate in the current invocation. If the time t has reached the deadline of any pending configurations, those configurations must be included in that set. For other configurations, the evaluator can choose to evaluate them now or to postpone evaluation. We describe the selection mechanisms in Section 4.2. Having selected configurations to evaluate, the algorithm removes those configurations from the pending set R.

Having selected a set of configurations, Algorithm 2 decides how to evaluate them. Function PlanConf selects a plan to evaluate the given set of configurations. Evaluating configurations in the right order can save significant overheads, compared to a random

Algorithm 3 PickConf: Methods for picking configurations to evaluate.

```
1: Input: Evaluation requests R, current timestamp t
2: Output: Set of configurations to evaluate
3: function PickConf-Threshold(R, t)
4:     // Was size threshold reached?
5:     if |R| ≥ ρ then
6:         // Return all requests
7:         return R
8:     else
9:         return ∅
10:    end if
11: end function
12: // Initialize maximal cost savings for each request
13: S = ∅
14: Input: Evaluation requests R, current timestamp t
15: Output: Set of configurations to evaluate
16: function PickConf-Secretary(R, t)
17:    // Add requests whose deadline is reached
18:    E ← {⟨c_H, t_D⟩ ∈ R | t_D ≥ t}
19:    // Remove requests from pending set
20:    R ← R \ E
21:    // Iterate over requests
22:    for r = ⟨c_H, t_D⟩ ∈ R do
23:        // Calculate re-configuration cost savings
24:        s ←CostSavings(r, E)
25:        // Retrieve maximal savings so far
26:        m ← S(r)
27:        // Should we evaluate?
28:        if t − (t_D − δ) ≥ δ/e ∧ s > m then
29:            E ← E ∪ {r}
30:        end if
31:        // Update maximally possible savings
32:        S(r) ← max(m, s)
33:    end for
34:    return E
35: end function
```

permutation. In particular, ordering them allows to amortize re-configuration overheads (e.g., overheads for creating an index) over the evaluation of multiple, similar configurations. The planner function (PlanConf) exploits this fact and aims at minimizing cost. We describe the planning mechanism in Section 4.3 in detail.

After selecting a plan, Algorithm 2 processes the plan steps in order (loop from Line 19 to Line 28). For each plan step s, the system first executes re-configuration actions required to evaluate specific heavy parameter settings (Line 21). Then, it selects a (near-)optimal setting of light parameters, specifically for the current configuration of heavy parameters (Line 23). Here, we invoke a reinforcement learning algorithm described via tuning parameters Alg_L. We discuss learning algorithms to implement this step in Section 5. Finally, Algorithm 2 benchmarks the current heavy and light parameter setting (Line 25) and adds the result to the set (Line 27). All evaluation results are ultimately returned to the invoking function (Line 30).

4.2 Picking Configurations to Evaluate

We present two strategies for selecting configurations to evaluate (invoked in Line 12 of Algorithm 2 and represented as Component B in Figure 2). Algorithm 3 shows corresponding pseudo-code. The two functions represented in Algorithm 3 (Function PickConf-Threshold or PickConf-Secretary) implement the call in Line 12 of Algorithm 2. Next, we discuss the two strategies in more detail.

The first strategy, represented by Function PickConf-Threshold, is relatively simple. We select all pending evaluation requests for processing if their number has reached a threshold. This threshold is represented as parameter ρ in the pseudo-code. Before reaching the threshold, we simply collect evaluation requests without actually processing them (i.e., the set of selected requests is empty). By evaluating requests in batches, we hope to amortize re-configuration overheads via the planning mechanisms outlined in the next subsection. The threshold ρ is a tuning parameter. It is associated with a tradeoff. Choosing ρ too small reduces chances for cost amortization. Choosing ρ too large means that we introduce significant delays for the RL algorithm between a configuration is selected and evaluated. Delaying feedback may increase time spent in exploring uninteresting parts of the search space. Note that ρ must be smaller or equal to the maximal delay, allowed by the RL algorithm. In our experiments, we typically set ρ to 20. Empirically, we determined this setting to work well for many scenarios.

Our second strategy, written as Function PickConf-Secretary, is more sophisticated and often works better in practice. It is motivated by algorithms for solving the so called "Secretary Problem" [17]. This problem models a job interview for a single position, in which a hiring decision must be made directly after each interview. This decision is hard due to uncertainty with regards to the quality of the remaining candidates. A popular algorithm for this problem reviews a fraction of $1/e$ of candidates without hiring any. Then, it selects the first candidate better than all previously seen candidates (or the last candidate, if no such candidate emerges). It can be shown that this strategy makes a near-optimal choice likely. We use an adaption of this algorithm for our problem.

In our case, candidates correspond to evaluation times for a fixed configuration. The re-configuration cost, required to test a specific configuration, decreases if similar configurations were evaluated before. E.g., we do not have to create an expensive index, part of a configuration to evaluate, if that index was created before. So, instead of immediately evaluating a configuration, we may want to wait until similar configurations are requested. Of course, we cannot know precisely which configurations will be submitted for evaluation in the future. This is akin to the uncertainty about the quality of future job candidates.

Algorithm 3 keeps track of possible cost savings for specific configurations. Global variable S keeps track of maximal savings in re-configuration costs for specific configurations, over different invocations of PickConf-Secretary. We compare current cost savings to the maximum seen so far to decide when to evaluate. Intuitively, we want to evaluate configurations in invocations, during which we can obtain particularly high cost savings.

Function PickConf-Secretary first selects all evaluation requests whose evaluation deadline has been reached (Line 18). Then, we iterate over the remaining requests. For each request, we calculate re-configuration cost savings, assuming that we evaluate it after the configurations selected for evaluation so far. Next, we retrieve maximal cost savings observed for this configuration so far (Line 26). We select the configuration for evaluation if we have

observed cost savings over a sufficiently large period (condition $t - (t_D - \delta) \geq \delta/e$ where δ is the maximal delay and e Euler's number) and if current savings exceed the previous optimum (condition $s > m$).

4.3 Optimizing Evaluation Order

Given a set of configurations to evaluate, we re-order them to minimize evaluation overheads (invoked in Line 16 of Algorithm 2 and represented as Component C in Figure 2). The following example illustrates the principle.

Example 4.1. We describe configurations by vectors in which each vector component represents a parameter value. Assume we have to evaluate configurations $(1, 1, 16MB)$, $(0, 0, 12MB)$, and $(0, 1, 16MB)$. Here, the first two components indicate whether two specific indexes are created or not, the third component represents the (configurable) amount of working memory. Assume that the latter parameter requires a server restart with a duration of 10 seconds to take effect. For simplicity, we assume that creating an index takes 20 seconds while dropping one is free. Evaluating the configurations in the given order creates (pure configuration switching) overheads of $2 \cdot 20 + 10 + 10 + 20 + 10 = 90$ seconds (assuming that no indexes are initially created and an initial setting of $8MB$ for memory). If we evaluate them in the order $(0, 0, 12MB)$, $(0, 1, 16MB)$, and $(1, 1, 16MB)$ instead, those overheads reduce to $10 + 10 + 20 + 20 = 60$ seconds. Relative savings tend to increase with the size of configuration batches.

We introduce the associated optimization problem formally.

Definition 4.2. An instance of **Reconfiguration Cost Minimization** is defined by a set $R = \{r_i\}$ of requested (heavy parameter) configurations to evaluate and a cost function $C : R \times R \mapsto \mathbb{R}^+$ that maps a pair r_1, r_2 of requested configurations to the cost for switching from r_1 to r_2 (e.g., by creating indexes that appear in r_2 but not in r_1). A solution is a permutation $\Pi : \mathbb{N} \mapsto R$ of configurations, representing evaluation order with cost $\sum_i C(\Pi(i), \Pi(i+1))$. An optimal evaluation order minimizes cost.

In the current implementation, we approximate $C(r_1, r_2)$ by only considering indexes that appear in r_2 but not r_1 and summing up the cardinality of the indexed table over all added indexes. Next, we analyze the computational complexity of this problem (called "reconfiguration cost minimzation" in the following).

Theorem 4.3. *Reconfiguration cost minimization is NP-hard.*

Proof. Consider an instance of the Hamiltonian graph problem. This instance is described by a graph G, the goal is to construct a path visiting each node once. We reduce to reconfiguration cost minimization as follows. For each node i in G, we introduce one evaluation request r_i. For each pair of nodes i and j, connected by an edge in G, we set the reconfiguration cost $C(r_i, r_j)$ to zero, otherwise to one. Assume we find an evaluation order with a reconfiguration cost of zero. In this case, we obtain a Hamiltonian path in the original problem instance (visiting nodes, associated with requests, in the order in which requests are selected for evaluation). As each request is evaluated once, the associated graph node is visited once. As the reconfiguration cost is zero, all visited nodes are connected by edges. □

Algorithm 4 PlanConf: Order configurations for evaluation.

```
1:  Input: Evaluation requests R
2:  Output: Requests in suggested evaluation order
3:  function PlanConf-Greedy(R)
4:      // Initialize list of ordered requests
5:      O ← []
6:      // Iterate over all requests
7:      for r ∈ R do
8:          // Find optimal insertion point
9:          i ← arg min_{i∈0,...,|O|} C_R(O[i−1], O[i]) + C_R(O[i], O[i+1]))
10:         // Insert current request there
11:         O.insert(i, r)
12:     end for
13:     return O
14: end function
```

Hence, we must choose between efficient optimization and guaranteed optimal results. In the following, we present a greedy and an exhaustive algorithm to solve this problem.

Algorithm 4 generates evaluation orders via a simple, greedy approach. The input is a set of evaluation requests (each one referencing a configuration to evaluate). Starting from an empty list, we expand the evaluation order gradually, by adding one more request in each iteration. We insert each request greedily at the position where it leads to minimal reconfiguration overheads. We measure re-configuration overheads via function $C_R(c_1, c_2)$, measuring reconfiguration overheads to move from configuration c_1 to configuration c_2. Those overheads include for instance index creation overheads for indices that appear in c_2 but not in c_1. After identifying the position with minimum overheads, we expand the order accordingly.

Next, we show how to transform the problem of ordering evaluations into an integer linear program. After doing so, we can use corresponding solvers to find an optimal solution quite efficiently. Our decision variables are binary: we introduce variables e_t^r to indicate whether request r is evaluated at time t. We introduce variables for each request $r \in R$ to evaluate and for $|R|$ time steps. We evaluate one configuration at each time step, represented by constraints of the form $\sum_r e_t^r = 1$ (for each time step t). Also, we must evaluate each configuration once which we represent by the constraint $\sum_t e_t^r = 1$ (for each request r)[2]. The objective function is determined by reconfiguration costs. For each pair of configuration requests r_1 and r_2, we can estimate reconfiguration cost $C_R(r_1, r_2)$ by comparing the associated configurations. We introduce binary variables of the form $i_t^{r_1,r_2}$, indicating whether reconfiguration costs for moving from r_1 to r_2 is incurred at time t. We introduce those variables for each pair of configurations and for each time step. The objective function is given as $\sum_{t,r_1,r_2} c_R(r_1, r_2) \cdot i_t^{r_1,r_2}$ (our goal is to minimize this function). Lastly, we need to ensure that the value assignments for variables $i_t^{r_1,r_2}$ and e_t^r are consistent. Due to the objective function, variables $i_t^{r_1,r_2}$ will be set to zero if possible. Hence, we only must constrain them to one if the context requires it. We do so by introducing constraints of the form $i_t^{r_1,r_2} \geq (e_t^{r_1} + e_{t+1}^{r_2})/2$ for each pair of requests and for each time step. The optimal solution to this linear program describes an optimal evaluation order.

[2]Strictly speaking, the last constraint is redundant as we evaluate exactly one configuration in each times step.

5 REINFORCEMENT LEARNING

UDO uses RL algorithms from the family of Monte Carlo Tree Search (MCTS) [13] methods. UDO can be instantiated with different algorithms, for optimizing light and heavy parameters respectively. Our implementation supports multiple algorithms as well. We discuss some of them in the following.

Throughout the pseudo-code presented so far, we used three sub-functions that relate to RL: RL.Select, RL.Update, and RL.Optimize. Those functions were used in Algorithm 1 and 2. The implementation of those functions depends on the RL algorithm used (as indicated by the Alg parameter). The first function, RL.Select, selects the next action to take, based on algorithm-specific statistics. The second function, RL.Update, updates those statistics based on feedback. Function RL.Optimize is based on the latter two functions and invokes them repeatedly for optimization.

Next, we show how to implement those functions for one specific RL algorithm. This algorithm follows the *Hierarchical Optimistic Optimization* (HOO) [9] framework, a generalized version of the well-known UCT algorithm [22]. We extend that algorithm with a mechanism for accepting delayed feedback. We call this algorithm *Delayed Hierarchical Optimistic Optimization* (Delayed-HOO). This is the algorithm used for our experiments for optimizing both, heavy and light parameters (when optimizing light parameters, we set the allowed delay to zero). While based closely on existing components for action selection [6] and delayed feedback management [20], the combination of those components is novel.

First, we discuss Function RL.Select. If the current state is an end state, this function returns the state representing the default configuration. Otherwise, we use the UCB-V selection policy [6], adapted for delayed feedback. Given the state representing the current configuration at time t, c_t, we choose the action a_t leading to configuration c_{t+1} that maximizes the upper confidence bound:

$$c_{t+1} \triangleq \underset{c}{\operatorname{argmax}}\, \hat{\mu}_c(t) + \sqrt{2.4\hat{\sigma}_c^2(t)\frac{\log(v_{c_t})}{v_c}} + \frac{3b\log(v_{c_t})}{v_c}, \quad (1)$$

Here, v_{c_t} and v_c are the number of visits to the parent configuration c_t and child configuration c respectively. The average reward obtained till time t after considering delay τ, i.e. $\hat{\mu}_c(t) = \sum_{i=\tau}^{t} f(c_{i-\tau})\mathbb{1}(c_{i-\tau} = c)$. Similarly, $\hat{\sigma}_c^2(t)$ is the empirical variance of reward for configuration c after considering the delay τ. As a practical alternative to the aforementioned estimates, our implementation also supports another estimate of average and variance of reward, following the RAVE (Rapid Action Value Estimation) [15] approach. This approach shares reward statistics for the same action, invoked in different states, thereby obtaining quality estimates faster. It is known to work well for particularly large search spaces.

Function RL.Update updates all of the aforementioned statistics, based on reward values received. More precisely, we update the number of visits to state-action pair (c_t, a_t), present state c_{t+1}, and sample mean and variance of accumulated rewards ($\hat{\mu}(c_t, a_t)$ and $\hat{\sigma}^2(c_t, a_t)$). Algorithm 5 shows simplified pseudo-code for Function RL.Optimize. Given a start state and a search space, this function iterates until a timeout. In each iteration, it selects actions via Function RL.Select (discussed before), evaluates the performance impact on benchmark B, and updates statistics accordingly (using Function RL.Update). It returns the most promising configuration found until the timeout.

Algorithm 5 RL: Monte Carlo Tree Search optimization.

```
1:  Input: Algorithm Alg, configuration space C, state c_0, benchmark B
2:  Output: Final parameter configuration
3:  function RL.Optimize(Alg, C, c_0)
4:      Initialize Stat ← ∅
5:      for t = 0, ..., Alg.Time do
6:          ⟨c_{t+1}, a_t⟩ ← RL.Select(Alg, C, c_t)
7:          Evaluate the new configuration r_t ← B.Evaluate(c_{t+1})
8:          Update Stat ← Stat ∪ {⟨c_t, a_t, c_{t+1}, r_t, t⟩}
9:          RL.Update(Alg, Stat)
10:     end for
11:     return Final parameter configuration c_T
12: end function
```

6 THEORETICAL ANALYSIS

We show, under moderately simplifying assumptions, that UDO converges to optimal configurations. UDO uses an extension of HOO algorithm, which provides this type of guarantee (Theorem 6, [9]). However, we decompose our search space (into a space for heavy and one for light parameters) and delay evaluation feedback (to amortize re-configuration costs). In this section, we sketch out our reasoning for why those changes do not prevent convergence. We provide proofs for the following theorems online[3].

In doing so, we use *expected regret* [7] as the metric of convergence. Given a time horizon T, expected regret $\mathbb{E}[\text{Reg}_T]$ is the sum of differences between the expected performance of the optimal configuration and the configuration achieved by the algorithm at any time step $t \leq T$. If the expected regret of an algorithm grows sublinearly with horizon T, it means the algorithm asymptotically converges to optimal configuration as $T \to \infty$.

Theorem 6.1 (Regret of HOO (Theorem 6, [9])). *If the performance metric f is smooth around the optimal configuration (Assumption 2 in [9]) and the upper confidence bounds on performances of all the configurations at depth h create a partition shrinking at the rate $c\rho^h$ with $\rho \in (0, 1)$ (Assumption 1 in [9]), expected regret of HOO is*

$$\mathbb{E}[\text{Reg}_T] = O\left(T^{1-\frac{1}{d+2}}(\log T)^{\frac{1}{d+2}}\right) \quad (2)$$

for a horizon $T > 1$, and $4/c$-near-optimality dimension[4] d of f.

Typically, configuration space C is a bounded subset of $\mathbb{R}^P$ and performance metric $f : C \to [a, b] \subset \mathbb{R}$. Here, d is of the same order as the number of parameters P. HOO uses UCB1 [7] rather than UCB-V [6]. For brevity of analysis, we follow the same though the proof technique is similar for any UCB-type (Upper Confidence Bound) algorithm.

[3] https://www.cs.cornell.edu/database/supplementary_proofs.pdf

[4] c-near-optimality dimension is the smallest $d \geq 0$, such that for all $\varepsilon > 0$, the maximal number of disjoint balls of radius $c\varepsilon$ whose centres can be accommodated in $\mathcal{X}_\varepsilon$ is $O(\varepsilon^{-d})$ (Def. 5 in [9]). Here, ε-optimal configurations $\mathcal{X}_\varepsilon \triangleq \{x \in C | f(x) \geq f^* - \varepsilon\}$. Near-optimality dimension encodes the growth in number of balls needed to pack this set of ε-optimal configurations as ε increases.

6.1 Regret of Delayed-HOO

Now, we prove that using delayed-UCB1 [20] instead of UCB1 allows us to propose delayed-HOO and also achieves similar convergence properties.

Theorem 6.2 (Regret of Delayed-HOO). *Under the same assumptions as Thm. 6.1, the expected regret of delayed-HOO is*

$$\mathbb{E}[\mathrm{Reg}_T] = O\left((1+\tau)T^{1-\frac{1}{d+2}}(\log T)^{\frac{1}{d+2}}\right) \quad (3)$$

for delay $\tau \geq 0$, horizon T, and $4/c$-near-optimality dimension d of f.

The bound in Eq. (3) is the same as Eq. (2) with an additional factor $(1+\tau)$, which does not change the convergence in terms of T. For delay $\tau = 0$, we retrieve the regret bound of original HOO.

The expected error in estimated expected performance (or reward) of any given configuration at time T is $r(T) = \mathbb{E}[f^* - \hat{f}(c_T)] = \frac{1}{T}\mathbb{E}[\mathrm{Reg}_T]$. Thus, the expected error $\epsilon(T)$ in estimating the expected performance (or reward) of a configuration using delayed-HOO converges at the rate $O\left((1+\tau)\left[\log T/T\right]^{1/(d+2)}\right)$, where T is the number of times the configuration is evaluated.

6.2 Regret of UDO

As we have obtained the error bound of the delayed-HOO algorithm, now we can derive bounds for UDO when using delayed-HOO for heavy and light parameters with two different delays.

Theorem 6.3 (Regret of UDO). *If we use the delayed-HOO as the delayed-MCTS algorithm with delays τ and 0, and time-horizons T_h and T_l for heavy and light parameters respectively, the expected regret of UDO is upper bounded by*

$$\mathbb{E}[\mathrm{Reg}_T] = O\left((1+\tau)T_h^{1-\frac{1}{d+2}}(\mathrm{HOO}^2(T_l)\log T_h)^{\frac{1}{d+2}}\right), \quad (4)$$

under the assumptions of Thm. 6.1. Here, $\mathrm{HOO}(T_l) \triangleq O\left(\left[\log T_l/T_l\right]^{\frac{1}{d+2}}\right)$.

Deviation in expected performance of the configuration returned by UDO from the optimum is $O\left((1+\tau)\left[\mathrm{HOO}^2(T_l)\mathrm{HOO}(T_h)\right]^{\frac{1}{d+2}}\right)$. Here, T_h and T_l are the number of steps allotted for the heavy and light parameters respectively. Deviation in expected performance of the configuration selected by UDO vanishes as $T_h, T_l \to \infty$.

7 EXPERIMENT EVALUATION

We describe our experimental setup (Section 7.1), compare UDO to baselines (Section 7.2) and variants (Section 7.3), and vary the benchmark scenario (Section 7.4).

7.1 Experimental Setup

We consider two standard benchmarks, TPC-C (with 10 warehouses and 32 concurrent requests) and TPC-H (with scaling factor one). We maximize throughput for TPC-C and minimize latency for TPC-H. We automatically tune two popular database management systems, MySQL (version 5.7.29) and Postgres (version 10.15), for maximal performance on those benchmarks. Our parameters include indexing choices (we consider index candidates that are referenced in queries), DBMS configuration parameters, as well as query order in transaction templates (for TPC-C). For TPC-C, we sample configuration quality by running a mix of 4% `STOCK_LEVEL`, 4% `DELIVERY`, 4% `ORDER_STATUS`, 43% `PAYMENT` and 45% `NEW_ORDER` transactions for five seconds. Also, we reload a fixed TPC-C snapshot every 10 iterations of UDO's main loop. For TPC-H, we evaluate queries. We consider 100 tuning parameters for MySQL and 105 parameters for Postgres. The majority of parameters (71) relate to indexing decisions, followed by 19 parameters related to reordering (each parameter represents the position of a query within a transaction template [35]), and, finally, parameters representing DBMS configuration parameters (10 parameters for MySQL and 15 for Postgres). For TPC-H, we consider 109 parameters for MySQL and 114 parameters for Postgres (99 of them are related to indexes, the other ones represent DBMS configuration parameters).

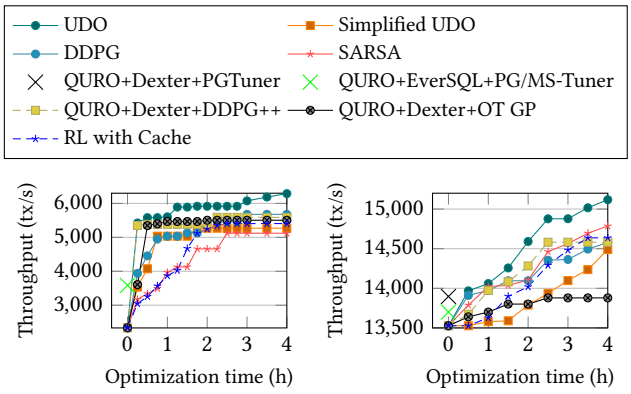
UDO itself is implemented in Python 3, using the OpenAI gym framework. It uses Gurobi (version 9) for cost-based planning. We compare UDO against several baselines that apply RL for universal database optimization without specialized treatment for heavy parameters. Those baselines use out of the box learning algorithms, SARSA [27] and Deep Deterministic Policy Gradient (DDPG) [24], provided by the Keras-RL framework [2] for Open AI gym. Prior work on database tuning via reinforcement learning [23, 38] has applied the same framework but to more narrowly defined tuning problems. We also consider a variant of the latter (using UDO's UCT algorithm without evaluation delays or configuration reordering) that exploits cached configurations. Here, we create database copies for each new heavy parameter configuration encountered and reuse previously created configurations, if available. Our cache uses up to 100 such slots, except for experiments with TPC-H with scaling factor ten where we reduce the number of slots to ten due to higher storage consumption per slot. All baselines discussed so far can optimize the same search space as UDO. In addition, we compare against combinations of tools that are each targeted at specific tuning problems such as index selection, configuration parameter tuning, or query reordering. Here, we compose solutions proposed for sub-problems by different tools, considering *MySQLTuner* [3], *PGTuner* [1], and the Gaussian Process and DDPG++ algorithms [32], as implemented in the *OtterTune* [12] tool, for system parameter tuning, *Quro* for selecting query orders [35], and *Dexter* [4] and *EverSQL* [5] for selecting indexes. When combining tools, we first optimize transaction code, then parameters, and finally index selections.

Note that UDO uses no prior training data but optimizes from scratch. Hence, we only consider baselines targeted at the same scenario (i.e., no prior training data). Unless noted otherwise, we set UDO's delay $\tau = 10$ for the heavy parameter MDP and $b = 3$ in UCB-V (Eq. (1)). We allow up to eight actions (i.e., tuning parameter changes compared to the defaults) per episode for TPC-H and up to 13 for TPC-C (four heavy parameter changes).

All of the following experiments were executed on a server with two Intel Xeon Gold 5218 CPUs with 2.3 GHz (32 physical cores), 384 GB of RAM, and 1 TB of hard disk.

7.2 Comparison to Baselines

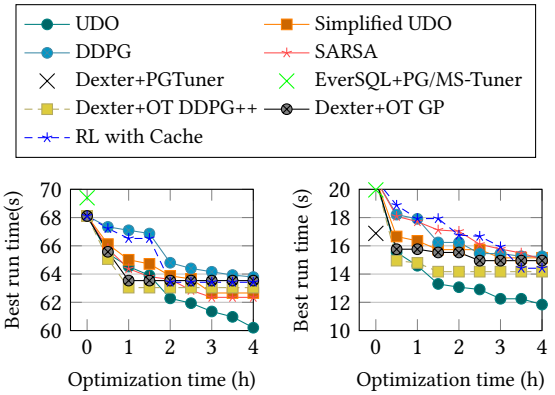
Next, we compare UDO against several baselines. Figure 3 reports experimental results for the TPC-C benchmark. Figure 4 reports results for TPC-H. For TPC-C, we report throughput (of the best configuration found so far) as a function of optimization time. Note



(a) TPC-C performance as a function of optimization time in MySQL.

(b) TPC-C performance as a function of optimization time in Postgres.

Figure 3: Comparing UDO to baselines on TPC-C.



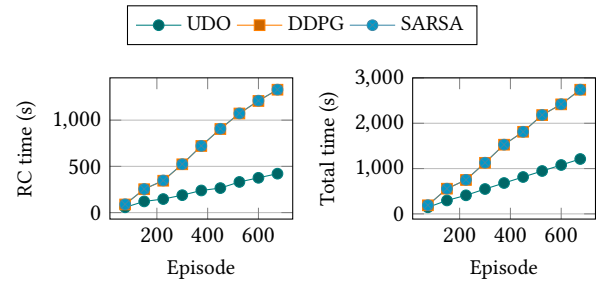
(a) TPC-H performance as a function of optimization time in MySQL.

(b) TPC-H performance as a function of optimization time in Postgres.

Figure 4: Comparing UDO to baselines on TPC-H.

that non-iterative baselines are represented as a dot while iterative baselines are represented as a curve. For TPC-H, we report execution time (for TPC-H queries) with the best configuration as a function of optimization time. We optimize for four hours with all baselines. Within the search space defined by our tuning parameters, UDO finds the best configurations for all four combinations of systems and baselines. Generally, the approaches based on reinforcement learning eventually find better solutions than the non-iterative methods. Among them, UDO performs best, followed in most cases by the combination of DDPG++ (for parameter tuning) and Dexter (for index selection). Configuration caching can sometimes improve over baselines without caches. It is however not as effective as parameter separation and evaluation order optimization, as implemented by UDO.

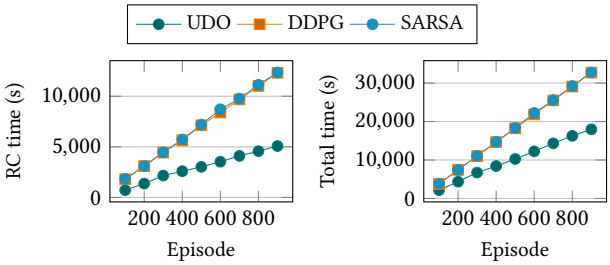
Digging deeper, we analyzed how different RL algorithms spend their time during optimization. Figure 5 shows total (right) and reconfiguration time (left) as a function of the number of episodes for three RL algorithms. Figure 6 shows the corresponding numbers for TPC-H. Clearly, UDO iterates faster as it reduces reconfiguration time. For instance, for MySQL running TPC-C, UDO reduces reconfiguration time by a factor of approximately three. This means UDO



(a) Reconfiguration time of different RL algorithms for MySQL on TPC-C.

(b) Total time of different RL algorithms for MySQL on TPC-C.

Figure 5: Time spent per episode by different RL algorithms when optimizing MySQL for TPC-C.



(a) Reconfiguration time of different RL algorithms for Postgres on TPC-H.

(b) Total time of different RL algorithms for Postgres on TPC-H.

Figure 6: Time spent per episode by different RL algorithms when optimizing Postgres for TPC-H.

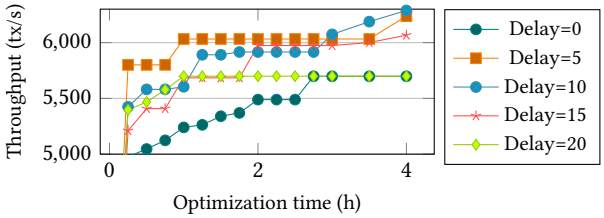


Figure 7: Impact of delayed feedback on UDO performance (MySQL on TPC-C).

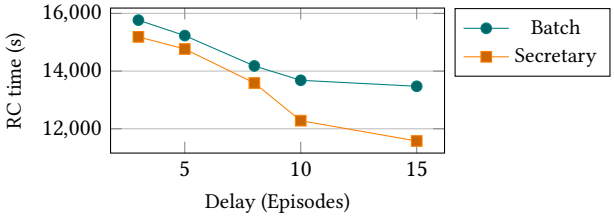
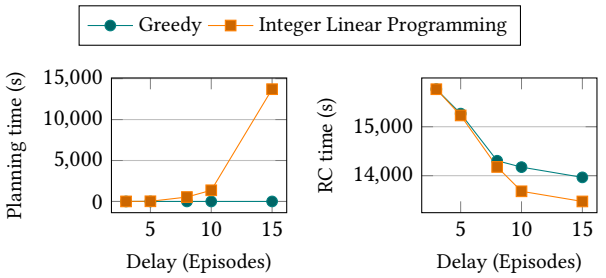


Figure 8: Impact of evaluation time selection on UDO performance (MySQL on TPC-C).

performs significantly more iterations within the same amount of optimization time, thereby finding promising configurations faster.

7.3 Comparison of UDO Variants

Delays. UDO delays the evaluation of configurations to amortize reconfiguration costs. Of course, there is a trade-off. While delays



(a) Time spent in plan optimization. (b) Time spent in reconfiguration.

Figure 9: Impact of reconfiguration planning algorithm on UDO performance (MySQL on TPC-C).

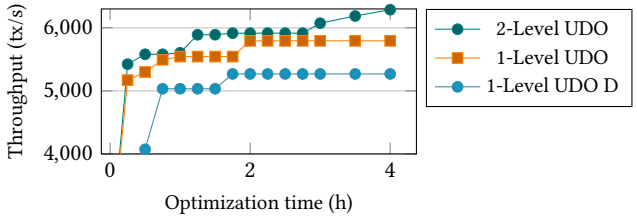


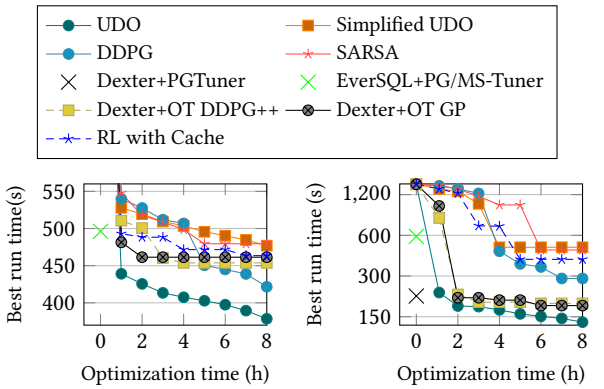
Figure 10: Impact of search space design and search strategy on UDO performance (MySQL on TPC-C).

decrease reconfiguration costs, they may also increase convergence time. Figure 7 evaluates UDO with different delay parameters (we measure delay by the maximal number of episodes between request and evaluation). Clearly, disabling delays (Delay=0) leads to slower convergence. Using a delay of five to ten turns out to be the optimal setting (ten is the default setting for our experiments).

Note that the results in Figure 7 relate to the quality of the solution, not to the overheads of optimization. Typically, the rate of improvements slows down as optimization progresses (this effect appears for instance in Figures 3 and 4). Hence, even small gains in throughput in Figure 7 likely translate into significant advantages in terms of optimization time (i.e., optimization time required by weaker approaches to close the gap).

Picking configurations. Delays are exploited by the evaluation manager to optimize time and order of evaluations. We propose two mechanisms to choose evaluation time. The first one, rather simple, evaluates once the batch of pending evaluation requests reaches a certain size. The second one is more sophisticated and tries to optimize the context in which configurations are evaluated. We compare both methods in Figure 8. Reporting reconfiguration time on the y-axis, we find that "Secretary-selection" works best. The gap between the two approaches increases with the delay (a higher delay means more choices in terms of evaluation time).

Ordering configurations. The second decision made by the evaluation manager relates to the order in which requests, selected for evaluation in a given time slot, are processed. We describe two approaches for request ordering in Section 4.3: a simple, greedy algorithm and an approach based on integer linear programming. Figure 9 compares the two approaches in terms of optimization time (left) and in terms of reconfiguration time (right), i.e. the quality of the generated solution. Clearly, the integer programming



(a) TPC-H performance as a function of optimization time in MySQL. (b) TPC-H performance as a function of optimization time in Postgres.

Figure 11: Comparing UDO to baselines on TPC-H for SF 10.

approach finds better solutions. The gap increases as the delay (and the number of potential orderings) increases. However, this advantage comes at a steep price. As shown on the left hand side, optimization time increases exponentially and becomes prohibitive for a delay of more than ten (which corresponds to around 100 requests). For our implementation, we switch to the greedy algorithm once delays become prohibitive.

1-level vs. 2-level MDP. Finally, we compare different representations of the search space. Figure 10 shows corresponding results. Our main version of UDO uses a two-level representation of the search space (separating heavy and light parameter MDPs). In a first step, we remove the separation between the two and apply the same RL algorithm to the 1-Level UDO MDP, introduced in Section 2. In a second step, we additionally delay feedback by evaluating configurations only at the end of each episode. Clearly, both of those changes degrade performance, compared to the original version.

7.4 Scenario Variants

Scaling up. We increase the scaling factor for TPC-H from one to ten. Figure 11 reports results for all baselines. The relative tendencies are similar to Figure 4. However, the spread of run times across different methods is larger. The impact of tuning decisions on performance grows with the data size. For Postgres, at the end of optimization, UDO achieves a 25% improvement in run time over the second-best baseline (136 versus 181 seconds).

Multi-criteria optimization. UDO can optimize composite performance metrics. To demonstrate this feature, we optimize a weighted sum between execution time and disk space consumed for indexes. Figure 12 shows run time, space for indexes, and the weighted sum (from left to right). We compare against DDPG and SARSA (configured to optimize the same objective). Compared to the baselines, UDO generates near-optimal solutions faster and ultimately finds the best tradeoff between disk space and run time.

Index recommendation. UDO is designed to optimize diverse parameters. Nevertheless, we can use it for more narrow problem variants. We evaluate UDO exclusively for index recommendation in Figure 13 (using default settings for all database system parameters). We add a new baseline that exploits the query optimizer's cost

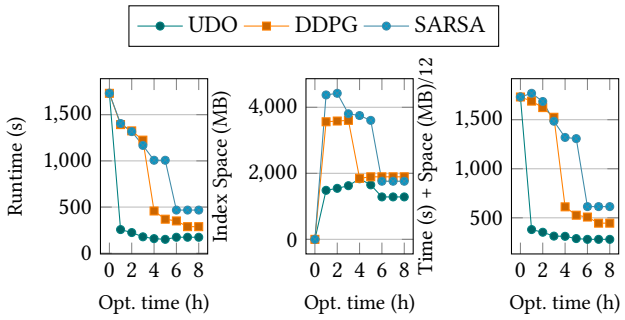
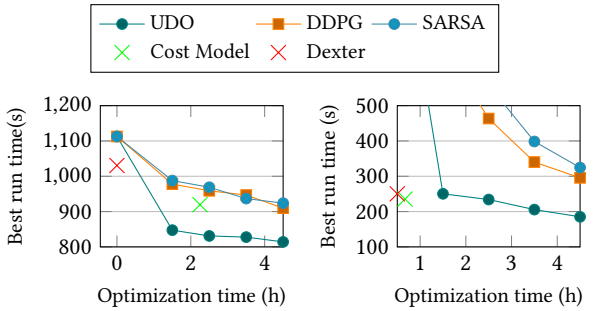


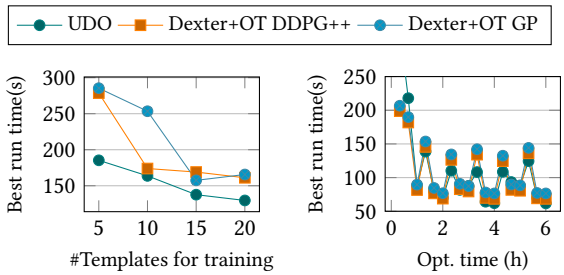
Figure 12: Optimizing weighted sum of run time and disk space for TPC-H SF 10 on Postgres.



(a) TPC-H performance as a function of optimization time in MySQL.

(b) TPC-H performance as a function of optimization time in Postgres.

Figure 13: Comparing UDO to baselines for index recommendation (TPC-H SF 10).



(a) Varying number of TPC-H query templates used for training.

(b) Performance for dynamic workload switching every full hour.

Figure 14: Performance for non-representative training sets and changing workloads (TPC-H SF 10, Postgres).

model: we generate all index candidates and estimate execution costs (via "explain" commands) if subsets of indexes are visible to the optimizer. We consider all subsets of up to three index candidates (the number of indexes selected by UDO in the final configuration). While not particularly efficient (the query optimizers of Postgres and MySQL do not directly support what-if analysis), this process identifies the index set that works best according to the optimizer's cost model. UDO ultimately finds better solutions than the baselines. However, the margins are smaller, compared to Figure 11. UDO works best for diverse tuning parameters.

Generalization. To test generalization, we train UDO and baselines for eight hours on a subset of TPC-H query templates. We show performance of the final configuration for *all queries* in Figure 14(a). Clearly, training with fewer queries degrades performance on the entire workload. The generalization overheads of UDO are comparable to baselines. E.g., for UDO, performance degrades by about 40% when considering five instead of 20 templates during training. It is around 70% for baselines based on Dexter.

Shifting workload. In Figure 14(b), we report results for a dynamic workload. We switch back between TPC-H query templates with odd numbers (i.e., Q1, Q3, etc.) and templates with even numbers every hour. Figure 14(b) reports run time for the current half of queries as a function of optimization time. For DDPG++ and OT GP, we use indexes proposed by Dexter for each of the two workload parts. As the indexes proposed by Dexter lead to one problematic query running for more than one hour, we added one more index from the final configuration generated by UDO for the baseline (index on the "L_PARTKEY" column of the "Lineitem" table). The presented results therefore correspond to upper bounds on performance for all approaches except for UDO. We see spikes for all baselines, whenever the workload changes. The magnitude of the spikes decreases over time, showing that all approaches converge to a configuration that compromises between the two workload parts. Considering aggregate run times for both workload parts, UDO still performs about 5% better than the nearest baseline.

8 RELATED WORK

Recently, there has been significant interest in using machine learning for database tuning [18, 25, 26, 31, 34]. Our work falls into the same, broad category as it exploits RL. Prior work typically focuses on specific tuning choices such as system configuration parameters [23, 37, 38], index selection [28, 29], or data partitioning [19, 36]. We support a broad set of tuning choices via one unified approach.

Traditionally, tuning decisions in a database system are made based on simplifying execution cost models. This often leads to sub-optimal choices in practice [8, 16]. UDO does not use any simplifying cost model. Instead, it exclusively uses feedback obtained via trial runs to identify promising configurations. In that, it also differs from a significant fraction of prior work using machine learning for database tuning [21, 37]. Many corresponding approaches rely on a-priori training data, obtained from representative workloads. UDO assumes no prior training data and learns (near-)optimal configurations from scratch. This makes optimization expensive (in the order of hours for our experiments) but avoids generalization errors and the need for training data. UDO will be demonstrated at the upcoming SIGMOD'21 conference [33].

9 CONCLUSION

We presented a system, UDO, for optimizing various tuning parameters by a unified approach. Our experiments show that parameter separation and delayed learning yield significant improvements.

ACKNOWLEDGMENTS

This research project is supported by NSF grant IIS-1910830 ("Regret-Bounded Query Evaluation via Reinforcement Learning").

REFERENCES

[1] 2020. https://github.com/jfcoz/postgresqltuner.

[2] 2020. https://github.com/keras-rl/keras-rl.

[3] 2020. https://github.com/major/MySQLTuner-perl.

[4] 2021. https://github.com/ankane/dexter.

[5] 2021. https://www.eversql.com/.

[6] Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári. 2007. Tuning Bandit Algorithms in Stochastic Environments. In *Algorithmic Learning Theory*, Marcus Hutter, Rocco A. Servedio, and Eiji Takimoto (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 150–165.

[7] P Auer, N Cesa-bianchi, and P Fischer. 2002. Finite time analysis of the multiarmed bandit problem. *Machine Learning* 47, 2-3 (2002), 235–256.

[8] 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*. 9:1—-9:6. https://doi.org/10.1145/2304510.2304522

[9] Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. 2011. X-Armed Bandits. *Journal of Machine Learning Research* 12, 5 (2011).

[10] Surajit Chaudhuri. 2004. Index selection for databases: A hardness study and a principled heuristic solution. *KDE* 16, 11 (2004), 1313–1323. http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=1339260

[11] Surajit Chaudhuri, V Narasayya, and Ravi Ramamurty. 2009. Exact cardinality query optimization for optimizer testing. In *VLDB*. 994–1005. https://doi.org/10.14778/1687627.1687739

[12] CMU Database Group. 2020. https://github.com/cmu-db/ottertune.

[13] Pierre-Arnaud Coquelin and Rémi Munos. 2007. Bandit Algorithms for Tree Search. *Arxiv preprint cs0703062* 23, March (2007), 67–74. arXiv:0703062v1 [arXiv:cs] http://arxiv.org/abs/cs/0703062

[14] 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 *SIGMOD*. 1241–1258. https://doi.org/10.1145/3299869.3324957

[15] Sylvain Gelly and David Silver. 2007. Combining online and offline knowledge in UCT. *Proceedings of the 24th international conference on Machine learning - ICML '07* (2007), 273–280. https://doi.org/10.1145/1273496.1273531

[16] Andrey Gubichev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3 (2015), 204–215.

[17] Theodore P. Hill. 2009. Knowing when to stop. *American Scientist* 97, 2 (2009), 126–133. https://doi.org/10.1511/2009.77.126

[18] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2019. Towards learning a partitioning advisor with deep reinforcement learning. *SIGMOD* (2019). https://doi.org/10.1145/3329859.3329876 arXiv:arXiv:1904.01279v1

[19] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2020), 143–157. https://doi.org/10.1145/3318464.3389704

[20] Pooria Joulani, András György, and Csaba Szepesvari. 2013. Online learning under delayed feedback. *30th International Conference on Machine Learning, ICML 2013* PART 3 (2013), 2503–2511. https://doi.org/10.14288/1.0044651 arXiv:1306.0686

[21] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: estimating correlated joins with deep learning. In *CIDR*. arXiv:1809.00677 http://arxiv.org/abs/1809.00677

[22] Levente Kocsis and C Szepesvári. 2006. Bandit based monte-carlo planning. In *European Conf. on Machine Learning*. 282–293. http://www.springerlink.com/index/D232253353517276.pdf

[23] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2018. QTune: A QueryAware database tuning system with deep reinforcement learning. *PVLDB* 12, 12 (2018), 2118–2130. https://doi.org/10.14778/3352063.3352129

[24] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings* (2016). arXiv:1509.02971

[25] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. 2020. Active Learning for ML Enhanced Database Systems. In *SIGMOD*. 175–191. https://doi.org/10.1145/3318464.3389768

[26] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2020. QuickSel: Quick Selectivity Learning with Mixture Models. In *SIGMOD*. 1017–1033. https://doi.org/10.1145/3318464.3389727 arXiv:1812.10568

[27] Gavin A Rummery and Mahesan Niranjan. 1994. *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge, Department of Engineering Cambridge, UK.

[28] Zahra Sadri, Le Gruenwald, and Eleazar Lead. 2020. DRLindex: Deep reinforcement learning index advisor for a cluster database. *ACM International Conference Proceeding Series* (2020). https://doi.org/10.1145/3410566.3410603

[29] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The case for automatic database administration using deep reinforcement learning. *arXiv* (2018), 1–9. arXiv:1801.05643

[30] Immanuel Trummer. 2019. Exact cardinality query optimization with bounded execution cost. In *SIGMOD*. 2–17.

[31] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: regret-bounded query evaluation via reinforcement learning. In *SIGMOD*. 1039–1050.

[32] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. 2021. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1241–1253. https://doi.org/10.14778/3450980.3450992

[33] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. Demonstrating UDO: A Unified Approach for OptimizingTransaction Code, Physical Design, and System Parameters via Reinforcement Learning. In *SIGMOD*.

[34] Lucas Woltmann, Claudio Hartmann, Maik Thiele, and Dirk Habich. 2019. Cardinality estimation with local deep learning models. In *aiDM*.

[35] Cong Yan and Alvin Cheung. 2016. Leveraging Lock Contention to Improve OLTP Application Performance. In *VLDBJ*, Vol. 9. 444–455. https://doi.org/10.14778/2876473.2876479

[36] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning data layouts for big data analytics. *arXiv* 2 (2020), 193–208.

[37] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J Gordon. 1910. A demonstration of the OtterTune automatic database management system tuning service. *VLDB* 11, 12 (1910), 1910–1913.

[38] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*. 415–432. https://doi.org/10.1145/3299869.3300085