Demonstrating UDO: A Unified Approach for Optimizing Transaction Code, Physical Design, and System Parameters via Reinforcement Learning

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

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

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

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 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 configuration switching overheads. For instance, it aims to amortize the creation of expensive data structures by consecutively evaluating configurations using them. We demonstrate 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.

CCS CONCEPTS

• **Information systems → Query optimization**.

KEYWORDS

Query optimization; Machine learning for data management

ACM Reference Format:
Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. Demonstrating UDO: A Unified Approach for Optimizing Transaction Code, Physical Design, and System Parameters via Reinforcement Learning. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3448016.3452754

1 INTRODUCTION

We demonstrate *UDO*, the *Universal Database Optimizer*. UDO is an offline tuning tool that optimizes settings for various kinds of database tuning 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. 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 [1]). It is suitable for scenarios in which an optimized configuration, obtained in an expensive one-time step, can be used over extended periods of time.

UDO operates on various types of tuning parameters, which are traditionally handled by separate tuning tools. For instance, for our demonstration, we combine optimization of transaction query orders [11], index selections [2], as well as database system configuration parameters [12]. 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). We use the generic term **Parameter** for each tuning choice in the following 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 can be based on multiple metrics such as throughput or latency. We demonstrate optimization with both metrics on different database systems (Postgres and MySQL) on 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 that maximize reward.

RL has been used previously for optimizing database system configuration parameters [7] 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

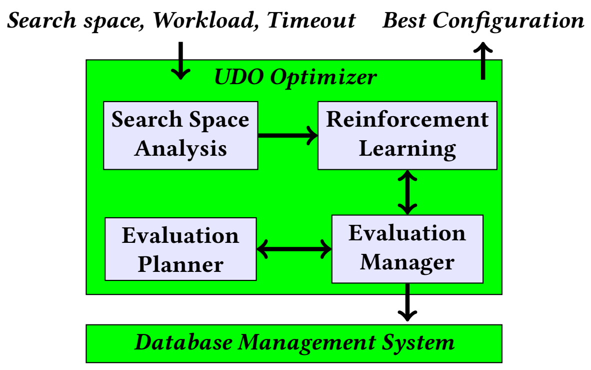


Figure 1: Overview of UDO system.

workload sample. Similarly, configuration parameters requiring a database server restart are relatively expensive to change. As we show in our experiments, a naive reinforcement learning approach is limited by costs of changing heavy parameters. This leads to high costs per iteration and slows down convergence.

UDO avoids this pitfall by giving heavy parameters special treatment. UDO separates heavy from light parameters and uses different reinforcement learning algorithms to optimize them. Specifically for heavy parameters, it uses reinforcement learning algorithms 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. This 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. For instance, it allows us to create an expensive index once to evaluate multiple similar configurations that all include the index. The alternative (alternating between configurations that use or do not use the index, requiring multiple index creations and drops) is less efficient.

For the current setting of heavy parameters, we use RL again to find optimal settings for light parameters. Of course, optimal settings for light parameters may 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 RL is applied to. In contrast to heavy parameters, we use an undelayed RL algorithm to converge faster to near-optimal settings for light parameters.

In our demonstration, we will give participants the chance to compare configurations of their own design against the one identified by UDO. Also, participants will be able to gain insights into how UDO identifies optimal solutions (by visualizing convergence over time for different parameters, based on pre-recorded traces). In the reminder of this paper, we give a more detailed overview of UDO (Section 2), present an extract from our experiments (Section 3), and describe our demonstration plan in more detail (Section 4).

2 SYSTEM OVERVIEW

Figure 1 shows an overview of the UDO system. Next, we discuss its components in more detail.

Input and Output. The input to UDO is threefold. First, we specify a search space for tuning. The search space is defined as a set of tuning parameters that may represent system parameters, physical design choices, or query planning decisions. Each parameter is associated with a set of admissible values and scripts that change the value (e.g., by creating an index). Second, UDO is given a workload to optimize. The workload is represented by a script that evaluates a given configuration with a workload sample and returns a reward value (which may represent throughput or the inverse of execution time). Third, UDO requires a time limit for optimization. The output is the best configuration found until the timeout.

Search Space Analysis. Search space analysis examines the space for configuration optimization. In particular, it divides tuning parameters into "heavy" and "light" parameters. For heavy tuning parameters, changing the value causes non-trivial overheads. Typically, index creations or parameter changes that require restarting the database server are heavy. On the other side, light parameters include system parameters that can be changed without restart or optimizer planning decisions. Here, values can be changed with high frequency. As outlined in more detail next, heavy and light parameters are treated differently during optimization. Currently, we classify parameters based on parameter type, considering parameters related to physical design and parameters requiring database server restarts for changes as heavy, the others as light.

Next, the search space analyzer creates an environment in which learning takes place. UDO uses reinforcement learning as core optimization mechanism. Reinforcement learning generally applies to Markov Decision Processes (MDPs), defined by states $\mathcal{S}$, actions $\mathcal{A}$, state transitions $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ (we consider deterministic transitions), and a function that assigns transitions to reward values. Here, we associate actions with configuration changes (e.g., creating an index or changing a system parameter). States are associated with configurations. Actions transition from a state representing a first configuration to a state representing the configuration obtained, after applying the change represented by the action to the first configuration.

A particularity of our approach is that we divide the search space into two levels. The top level is focused on heavy parameters alone. Each state only represents choices for those parameters and actions refer only to those parameters. The reward for taking an action in a state is linked to the performance improvement obtained by the associated configuration change. The performance depends however not only on choices for heavy parameters but also on choices for light parameters. Ideally, we want to assess settings for heavy parameters, based on the optimal settings for the light parameters. The optimal settings for light parameters may however depend on the choices for heavy parameters (e.g., we may want to disable sequential scans in the optimizer search space only after creating a particularly useful index). Hence, we introduce a second level of optimization that focuses on light parameters only. Here, we assume fixed values for heavy parameters while exploring a state space representing alternative settings for light parameters. As outlined next, this separation enables us to reduce overheads related to configuration changes. The search space analyzer creates an implicit (i.e., space-efficient) representation of that search space, which is used to initialize the reinforcement learning agents.

Example 2.1. Assume our search space is defined by the following parameters with associated binary value domains. I_1 and I_2 representing decisions as to whether index one and two are created or not. Parameters $O_1 \in \{0, \ldots, 7\}$ and $O_2 \in \{0, \ldots, 4\}$ represent alternative query orderings (identified by an integer number) for transactions one and two. Here, I_1 and I_2 should be classified as heavy parameters (as creating indexes typically takes time) while O_1 and O_2 are light (changing query order merely requires switching to a different version of the transaction code). Hence, our search space at level 1 has four states (corresponding to the admissible value combinations for I_1 and I_2). Its actions refer only to changes with regards to the indexes. For each index combination, we introduce a separate MDP (at level 2) to optimize settings for O_1 and O_2. Each MDP at level 2 has $7 \cdot 4 = 28$ possible states. The reward of an index combination (level 1) is calculated based on (near-)optimal, index-specific settings for O_1 and O_2. To find those near-optimal settings, the associated MDP at level 2 is solved.

Reinforcement Learning. UDO applies reinforcement learning to solve the MDPs described before. As discussed before, we decompose the search space into multiple dependent MDPs. Doing so allows us to solve those MDPs by different algorithms. We outline next why that is interesting.

Reinforcement Learning (RL) often benefits from a high frequency of iterations. This allows the RL agent to explore the search space in depth within a reasonable time frame. In our case, iteration frequency is limited by two factors: *time for creating a configuration* and *time for evaluating a configuration*. The time for evaluating a configuration can often be reduced via sampling (from the given workload). As we consider stochastic rewards in general, obtaining performance on a sample is in principle sufficient for convergence. The time for creating configurations, however, cannot easily be reduced. For instance, actions related to physical design changes (e.g., index creations) can take significant amount of time. As we cannot fully avoid such overheads, our goal is to amortize them over multiple evaluations instead.

For instance, to decrease index creation overheads per iteration, we ideally want to create indexes once and then evaluate a set of similar configurations consecutively. Standard reinforcement learning algorithms do not cope with this requirement. They expect reward values immediately after taking an action. This does not leave any slack to collect similar configurations before evaluating them. Hence, specifically for the MDP optimizing heavy parameters, we use algorithms [4, 6] that accept delays between an action is taken and the associated reward becomes known.

Instead of evaluating heavy parameter configurations directly, we forward them to the evaluation manager component. This component, described in more detail next, uses cost-based planning to evaluate configuration sets efficiently. Once performance results become available, the corresponding reward values are forwarded to the RL agent. We use an algorithm that copes with delayed rewards to optimize heavy parameters. On the other side, we use non-delayed reinforcement learning to optimize light parameters (via the UCT algorithm with Rapid Value Estimation [3]). Here, amortization is not required (since changes are cheap) and would only delay convergence unnecessarily. We show in Section 3 the benefits of this two-level approach.

Evaluation Manager. The evaluation manager continuously receives evaluation requests from the RL agent. Each evaluation request references a configuration whose performance to evaluate. Additionally, each request specifies a deadline until which the evaluation result is required. The evaluation manager is responsible for addressing each request before the deadline. Under that constraint, its goal is to minimize evaluation overheads. Such overheads are minimized by optimizing the point in time and the order in which configurations are evaluated.

More specifically, we focus on minimizing overheads for creating configurations to evaluate. For instance, we amortize index creation overheads if we evaluate two configurations with similar indexes consecutively. We amortize overheads for a database server restart (to have system configuration parameter changes take effect) if we evaluate multiple configurations using the current settings consecutively. The evaluation manager tries to take advantages of such effects as much as possible.

Example 2.2. 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.

Evaluation Planner. To reduce overheads, the evaluation manager uses a cost-based planner. This planner formalizes the evaluation of configuration sets as an optimization problem. The goal is to minimize overheads for switching between configurations while generating all results until the corresponding deadline.

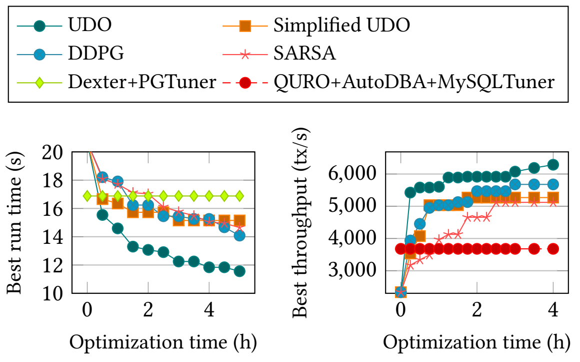
The cost model of the planner is based on time measurements, taken when changing heavy parameters. The search space corresponds to configuration evaluation orders that are admissible, given the current evaluation deadlines. To explore this search space, the planner uses a dynamic programming algorithm.

3 EXPERIMENTS

We present a small extract from our experiments, evaluating UDO on different benchmarks and systems.

Setup. We use a server with 2 Intel Xeon Gold 5218 2.3 GHz CPUs with 32 physical cores and 384 GB of RAM for our experiments. We implemented UDO in Python, using OpenAI gym[1]. We compare against two popular deep reinforcement learning algorithms, DDPG [8] and SARSA [9], implemented in the same framework (but not using the UDO-specific amortization of configuration change cost). Also, we compare against a simplified UDO version that does

[1]https://gym.openai.com/



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

Figure 2: Performance comparison of UDO and baselines.

not separate heavy from light parameters and does not exploit delayed evaluations either. Finally, we compare against more traditional tuning tools and combine their solution for different classes of parameters. Namely, we set system parameters for Postgres (v 10.15) via PGTuner [2] and select indexes via Dexter [5]. For MySQL (v5.7.29), we set system parameters via MySQLTuner [3], using indexes recommended by Dexter for Postgres which turned out to be equivalent to indexes recommended by another baseline we tried directly on MySQL, NoDBA [10], for the following experiments. We use Quro [11] as baseline for optimizing query order in transactions.

Results. We report an extract of our experimental results in Figure 2. We show latency for TPC-H, with scaling factor 1, on Postgres as a function of optimization time in Figure 2(a). We show throughput for TPC-C on MySQL (with scaling factor 10, 32 concurrent requests, and 10 warehourses) in Figure 2(b). For TPC-H, we consider index selection and system parameter tuning, a total of 137 tuning parameters. For TPC-C, we consider indexing, transaction query re-orderings [11], and system parameters, a total of 112 parameters. Given enough time, UDO finds the best solutions for both systems and benchmarks. Compared to other learning-based baselines, UDO achieves significantly faster iterations due to delayed evaluations and cost-based planning (speedup of factor 8 for TPC-C and factor 4 for TPC-H, comparing UDO to simplified UDO).

4 DEMONSTRATION SETUP

Our demonstration consists of two parts. First, we give participants the opportunity to compare alternative configurations against the ones found by UDO. Second, we give participants a chance to visualize convergence of UDO for different parameters, based on pre-recorded traces.

Competition. We will provide participants with access to an EC2 machine on which the TPC-H benchmark is installed. We will point audience members to a public document in which they can register for access during 30 minutes time slots. We will provide registered participants with the login details, a list of parameters considered by UDO, and the final performance numbers achieved. Visitors are free to try out different configurations, also installing tuning tools of their choice. In case of significant changes, we

[2]https://github.com/jfcoz/postgresqltuner
[3]https://github.com/major/MySQLTuner-perl

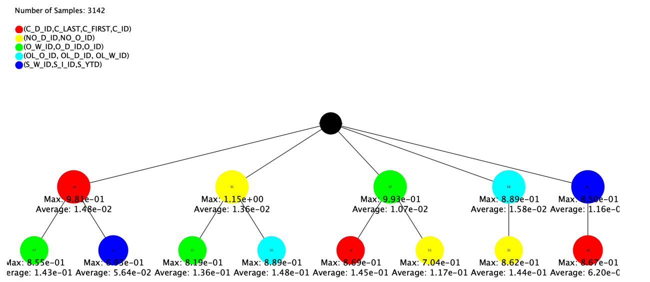

Number of Samples: 3142
(C_D_ID,C_LAST,C_FIRST,C_ID)
(NO_D_ID,NO_O_ID)
(O_W_ID,O_D_ID,O_ID)
(OL_O_ID, OL_D_ID, OL_W_ID)
(S_W_ID,S_I_ID,S_YTD)
Max: 9.81e-01 Average: 1.48e-02
Max: 1.15e+00 Average: 1.36e-02
Max: 9.93e-01 Average: 1.07e-02
Max: 8.89e-01 Average: 1.58e-02
Max: 8.50e-01 Average: 1.16e-0
Max: 8.55e-01 Average: 1.43e-01
Max: 6.93e-01 Average: 5.64e-02
Max: 8.19e-01 Average: 1.36e-01
Max: 8.89e-01 Average: 1.48e-01
Max: 8.69e-01 Average: 1.45e-01
Max: 7.04e-01 Average: 1.17e-01
Max: 8.62e-01 Average: 1.44e-01
Max: 8.67e-01 Average: 6.20e-0

Figure 3: Screenshot of UDO visualization tool.

will be able to restore the defaults quickly via a stored EC2 AMI. Participants may send us their best performing configuration. We will benchmark the configuration ourselves and maintain a public "high-score list" with achieved performance results.

Visualization. Audience members may gain insights into how UDO works internally, using a visualization tool. To save time, we will provide pre-recorded optimization traces for TPC-C on MySQL and Postgres for download, together with the visualization tool. The visualization tool allows users to select subsets of parameters to visualize, and shows a video illustrating UDO's convergence over time (in addition to statistics such as reward values, visit frequencies, and UCT search tree growth). Figure 3 shows a screenshot.

ACKNOWLEDGMENTS

This research project is supported by NSF grant IIS-1910830 ("Regret-Bounded Query Evaluation via Reinforcement Learning") and a Cornell Computer Science Electronic Arts Fellowship.

REFERENCES

[1] 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
[2] 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
[3] Sylvain Gelly and David Silver. 2011. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence* 175, 11 (2011), 1856–1875.
[4] Pooria Joulani, Andras Gyorgy, and Csaba Szepesvári. 2013. Online learning under delayed feedback. In *International Conference on Machine Learning*. 1453–1461.
[5] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic Mirror in My Hand, Which is the Best in the Land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 12 (July 2020), 2382–2395. https://doi.org/10.14778/3407790.3407832
[6] Bingcong Li, Tianyi Chen, and Georgios B Giannakis. 2019. Bandit online learning with unknown delays. In *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 993–1002.
[7] 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
[8] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).
[9] Gavin A Rummery and Mahesan Niranjan. 1994. *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge, UK.
[10] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The case for automatic database administration using deep reinforcement learning. *arXiv preprint arXiv:1801.05643* (2018).
[11] Cong Yan and Alvin Cheung. 2016. Leveraging lock contention to improve OLTP application performance. *Proceedings of the VLDB Endowment* 9, 5 (2016), 444–455.
[12] 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.