Lecture #19

Carnegie Mellon University

Special Topics:

Self-Driving Database Management Systems

Autonomous Systems I

Wan Shen Lim // 15-799 // Spring 2022

LAST CLASS

- High-level NoisePage Pilot architecture.

TODAY’S AGENDA

- UDO: Universal Database Optimization Using Reinforcement Learning. Junxiong Wang, Immanuel Trummer, Debabrota Basu. VLDB 2022.

- Commentary: I think of this as an intelligent configuration batching paper. Doesn’t rely on models. Splits parameters into cheap / expensive to change. Introduces new RL algorithm with proofs.

MOTIVATION

Cost models

- Require training data.
- Error-prone estimates.

Rely on sample runs instead

- Regime: high-quality, high-overhead optimization.

Tuning Quality \ Overhead	high	low
high	UDO	the dream
low	<bleep>	cost models

MOTIVATION

Primary use cases

- Tuning when a configuration can be reused over an extended period.
- Analysis tool for other tuning approaches.

Generalizes to many tuning problems

- e.g., transaction query order, index selection, knob tuning.

REINFORCEMENT LEARNING

Learning from sample runs

- Classic use case for reinforcement learning.
- **Actions** The tuning actions we've seen in class.
- **Reward** The improvement in target metrics.

REINFORCEMENT LEARNING

Learning from sample runs

- **Algorithm**

  for config in search space

  try next action ← Insert RL here

  evaluate performance

- What's the problem?

REINFORCEMENT LEARNING

Learning from sample runs

- **Algorithm**

```
for config in search space
    try next action
    evaluate performance
```

- What's the problem?
- Trying actions can be expensive!
  - High cost per iteration, slow convergence.
  - Key insight: yet not all actions are equally expensive.

TERMINOLOGY

- **Parameter** Each tuning choice.
  Explicitly: a knob, a create index, etc.
- **Configuration** An assignment from parameters to values.
- **Heavy parameter** A parameter which is expensive to change.
  Currently, anything involving physical data structures or database restarts.
- **Light parameter** A parameter which is cheap to change.
  Anything which isn't heavy.

UDO: KEY IDEAS

Give heavy parameters special treatment

- Use RL algorithm that supports delayed rewards.
- e.g., create an expensive index once, batch evaluate similar configurations with that expensive index.
- For each heavy parameter, the optimization of light parameters is a separate Markov decision process.

Light parameters, business as usual

- Use standard no-delay RL.

UDO: SUPPORTING CAST

Planning component

- Each configuration selected by RL is forwarded here.
- Decides when and in what order to evaluate configs.

New RL algorithms that accept delayed feedback

- Variant of Monte Carlo Tree Search, delayed-HOO.

MCTS, delayed-HOO

Monte-Carlo Tree Search

- Recent publicity: AlphaGo, Total War, Tesla Autopilot.
- Game = tree, nodes are states, edges are actions.
  - Selection — From root, select children until you reach a leaf.
  - Expansion — Expand the leaf (if non-terminal) with children.
  - Simulation — Rollout/playout until the game is won/lost.
  - Backprop — Update weights.

Delayed-Hierarchical Optimistic Optimization

- (same authors) [AAAI22] Procrastinated Tree Search.
- Proofs, details, regret bounds, etc. are there.

FORMAL MODEL: DEFINITIONS

- **Parameter** As previously defined.

Has a **value domain**, e.g., is index built 0/1, query position in txn.

- **Configuration** c = vector [parameter -> value].
- **Configuration space** C = {all possible c}.

  C_H : heavy parameters, C_L: light parameters,

  C = C_H X C_L.
- **Benchmark metric** f : C -> real number, stochastic.
- **UDO instance** (f, C), find c* = argmax E[f(c)]

FORMAL MODEL: UDO -> MDPs

Given a UDO instance (f, C),
where the goal is to find c* = argmax E[f(c)],
map (f, C) to **multiple** episodic Markov decision processes.

Episodic MDP (S, A, T, R, S_d, S_e), in this case,
(S state space, A actions, T : S*A->S deterministic transition,
R : S->real stochastic reward,
S_d episode start states, S_e episode end states)

FORMAL MODEL: Heavy Parameter MDP

Heavy Parameter MDP

- Each action changes one heavy parameter to a new value.
- Start state is default configuration.
- Reward is max over c_L, f(c_H o c_L) - f(c_default).
- End state is all states that are N actions away (they use N=4).

FORMAL MODEL: Light Parameter MDP

Light Parameter MDP M_L[c_h] for each heavy parameter c_h

- Actions are value changes for light parameters.
- End states are a fixed number of light parameter changes.
- Reward is f(c_H o c_L) - f(c_default).

UDO OVERVIEW

Iterate until the time limit is reached

- Note other stopping conditions could be used instead.

Algorithm 1 UDO main function.

Input: Benchmark metric f, configuration space C, RL algorithms Alg_H and Alg_L for heavy and light parameter optimization
Output: a suggested configuration for best performance
function UDO($f, C, \text{Alg}_H, \text{Alg}_L$)
    // Divide into heavy (C_H) and light (C_L) parameters
    $\langle C_H, C_L \rangle \leftarrow$ SSA.SplitParameters(C)
    // Until optimization time runs out
    for $t \leftarrow 1, \ldots, \text{Alg}_H.Time$ **do**
        // Select next heavy parameter configuration
        $c_{H,t} \leftarrow$ RL.Select($\text{Alg}_H, C_H, c_{H,t-1}$)
        // Submit configuration for evaluation
        EVAL.Submit($c_{H,t}, t + \text{Alg}_H.maxDelay$)
        // Receive newly evaluated light configurations
        $E \leftarrow$ EVAL.Receive(Alg_L, f, C_L, t)
        // Update statistics for heavy parameters
        RL.Update(Alg_H, E)
    end for
    return best obtained configuration
end function

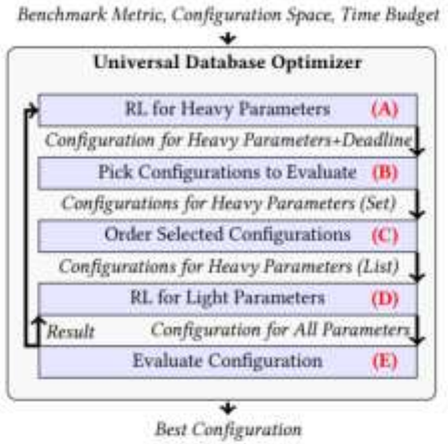


Figure 2: Overview of UDO system (rectangles represent processing steps, arrows represent data flow).

EVALUATING CONFIGURATIONS: API

EVAL.Submit(config, deadline)

- Deadline = max additional future configs that can be buffered before this specific config must be evaluated.

EVAL.Receive(RL algorithm to use, f, light config space, current time)

- Get the next set of evaluated configs.

Algorithm 2 EVAL: Functions for evaluating configurations.

```
// Global variable representing evaluation requests
R ← ∅
Input: heavy configuration c_H to evaluate and time t
Effect: adds new evaluation request
procedure EVAL.SUBMIT(c_H, t)
    R ← R ∪ {(c_H, t)}
end procedure
Input: RL algorithm Alg_L, benchmark metric f, time t, and space C_L
Output: evaluated configurations with reward values
function EVAL.RECEIVE(Alg_L, f, C_L, t)
    // Choose configurations from R to evaluate now
    N ←PICKCONF(R, t)
    // Remove from pending requests
    R ← R \ N
    // Prepare evaluation plan
    P ←PLANCONF(N)
    // Collect evaluation results by executing plan
    E ← ∅
    for s ∈ P.steps do
        // Prepare evaluation of next configurations
        CHANGECONFIG(s.hconf)
        // Find (near-)optimal light parameter settings
        c_L ←RL.OPTIMIZE(Alg_L, s.hconf, C_L, f)
        // Take performance measurements on benchmark
        b ←EVALUATE(f, s.hconf, c_L)
        // Add performance result to set
        E ← E ∪ {(c_L, s.hconf, b)}
    end for
    // Return evaluation results
    return E
end function
```

EVALUATING CONFIGURATIONS: PICKING

PickConf-Threshold

- If you have “too much” work to do, you must do everything now.

PickConf-Secretary

- Do everything that must be done.
- Then, secretary problem style, do whatever doesn’t require “too much” reconfiguration work.

Algorithm 3 PickConf: Methods for picking configurations to evaluate.

```
Input: Evaluation requests R, current timestamp t
Output: Set of configurations to evaluate
function PickConf-Threshold(R, t)
    // Was size threshold reached?
    if |R| ≥ ρ then
        // Return all requests
        return R
    else
        return ∅
    end if
end function

// Initialize maximal cost savings for each request
S = ∅

Input: Evaluation requests R, current timestamp t
Output: Set of configurations to evaluate
function PickConf-Secretary(R, t)
    // Add requests whose deadline is reached
    E ← {⟨c_H, t_D⟩ ∈ R | t_D ≥ t}
    // Remove requests from pending set
    R ← R \ E
    // Iterate over requests
    for r = ⟨c_H, t_D⟩ ∈ R do
        // Calculate re-configuration cost savings
        s ← CostSavings(r, E)
        // Retrieve maximal savings so far
        m ← S(r)
        // Should we evaluate?
        if t − (t_D − δ) ≥ δ/e ∧ s > m then
            E ← E ∪ {r}
        end if
        // Update maximally possible savings
        S(r) ← max(m, s)
    end for
    return E
end function
```

EVALUATING CONFIGURATIONS: ORDERING

Ordering configurations is NP-HARD

- Hamiltonian graph.

PlanConf

- Greedy algorithm.

Integer linear programming

- Optimal solution.

Algorithm 4 PLANCONF: Order configurations for evaluation.

```
Input: Evaluation requests R
Output: Requests in suggested evaluation order
function PLANCONF-GREEDY(R)
    // Initialize list of ordered requests
    O ← []
    // Iterate over all requests
    for r ∈ R do
        // Find optimal insertion point
        i ← arg min_{i∈0,...,|O|} C_R(O[i−1], O[i]) + C_R(O[i], O[i+1]))
        // Insert current request there
        O.insert(i, r)
    end for
    return O
end function
```

REINFORCEMENT LEARNING

Three main subroutines: RL.SELECT, RL.UPDATE, RL.OPTIMIZE.

RL.SELECT	RL.UPDATE	RL.OPTIMIZE
Pick the next action based on some statistics.	Update the statistics used by RL.SELECT.	Invoke the other two repeatedly for optimization.

See paper for details.
$c_{t+1} \triangleq \underset{c}{\operatorname{argmax}}\, \hat{\mu}_c(t) + \sqrt{2.4\hat{\sigma}_c^2(t)\frac{\log(v_{c_t})}{v_c}} + \frac{3b\log(v_{c_t})}{v_c}$
	Update num visits to state-action pairs, present state, sample mean and variance of accumulated rewards.	**Algorithm 5** RL: Monte Carlo Tree Search optimization.
		Input: Algorithm Alg, configuration space C, state c_0, benchmark B
		Output: Final parameter configuration
		function RL.OPTIMIZE(Alg, C, c_0)
		Initialize $Stat \leftarrow \emptyset$
		for $t = 0, \ldots,$ Alg.$Time$ **do**
		$\langle c_{t+1}, a_t \rangle \leftarrow$ RL.SELECT(Alg, C, c_t)
		Evaluate the new configuration $r_t \leftarrow$ B.EVALUATE(c_{t+1})
		Update $Stat \leftarrow Stat \cup \{\langle c_t, a_t, c_{t+1}, r_t, t \rangle\}$
		RL.UPDATE(Alg, $Stat$)
		end for
		return Final parameter configuration c_T
		end function

THEORY

- Minimizes expected regret.
- See paper for details.
- Extended proofs in [AAAI22] Procrastinated Tree Search.

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*

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

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

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.

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, $\mathrm{HOO}(T_l) \triangleq O\left([\log T_l / T_l]^{\frac{1}{d+2}}\right)$.

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 $T_h, T_l \to \infty$.

EXPERIMENTS: SETUP

Hardware

- Server, 2x Intel Xeon Gold 5218, 2.3 GHz, 32 physical cores.
- 384 GB RAM.
- 1 TB HDD.

DBMSs

- MySQL 5.7.29.
- PostgreSQL 10.15.

EXPERIMENTS: SETUP

UDO

- Delay = 10 for heavy MDP.
- b = 3 in UCB-V (RL.SELECT picking the next action).
- Per episode,
  - Up to 8 actions for TPC-H.
  - Up to 13 actions for TPC-C (four heavy parameter changes).

EXPERIMENTS: WORKLOADS

Workloads

- TPC-C (SF 10, 32 terminals), maximize throughput.
  - Reload snapshot every 10 iterations of main loop.
  - Standard mix for 5 seconds.
  - Parameters: 71 index, 19 reorder, 10 MySQL / 15 PostgreSQL knobs.
- TPC-H (SF 1), minimize latency.
  - Parameters: 99 index, 10 MySQL / 15 PostgreSQL knobs.

EXPERIMENTS: IMPLEMENTATION

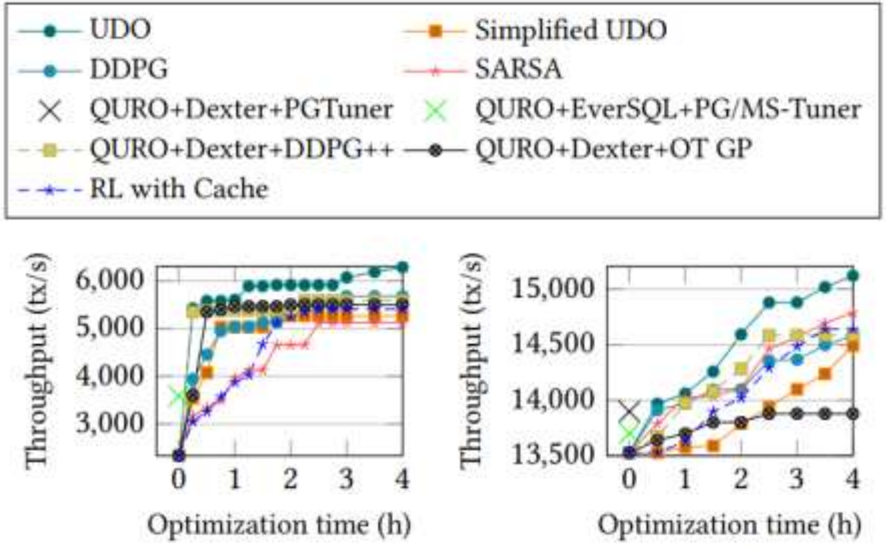
UDO

- Python3 + OpenAI gym
- Gurobi for cost-based planning

Baselines (targeted at no prior training data scenario)

- For RL comparisons, against Keras-RL's SARSA, DDPG.
- Some combination of MySQL-Tuner, PGTuner, Gaussian Process Regression, DDPG++, Quro, Dexter, EverSQL.
- When combining, optimize transaction code, then parameters, then index selection.

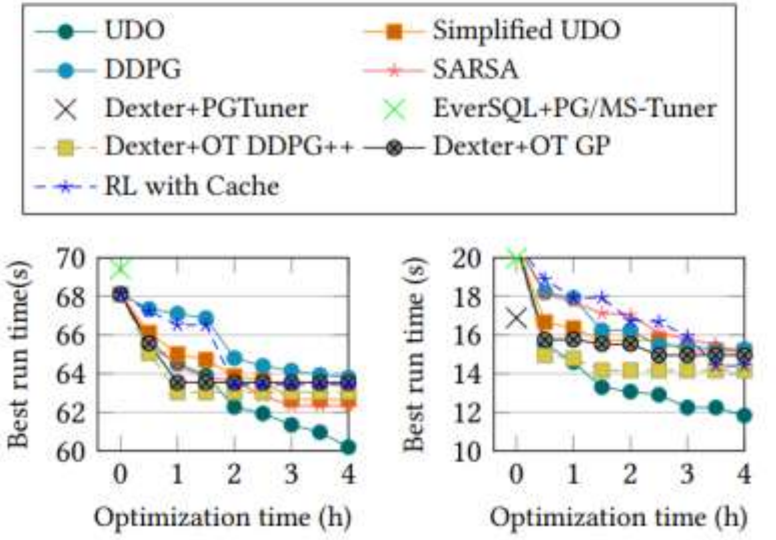
UDO vs BASELINES



(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.

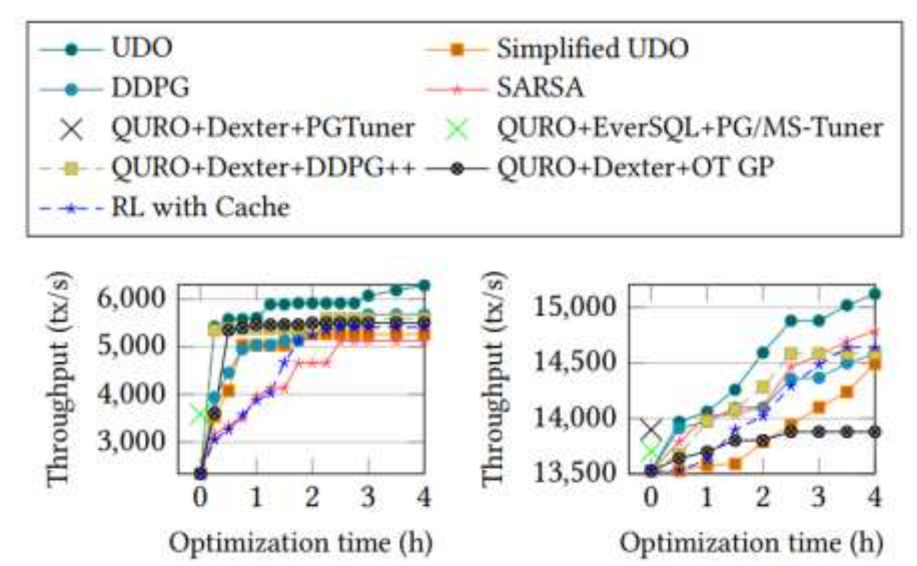


(a) TPC-H performance as a function of optimization time in MySQL.

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

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

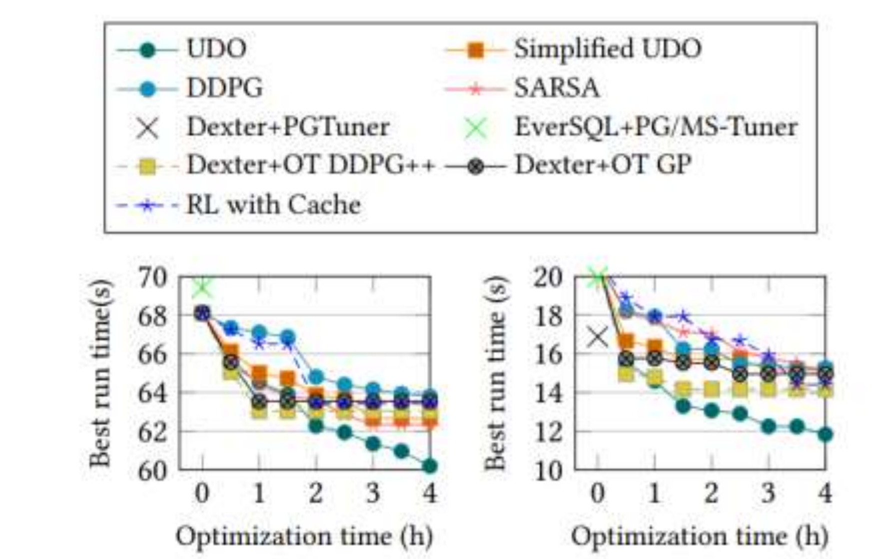
UDO vs BASELINES



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

(b) TPC-C ... optin...

Figure 3: Comparing UDO to bas...



... a function ... MySQL.

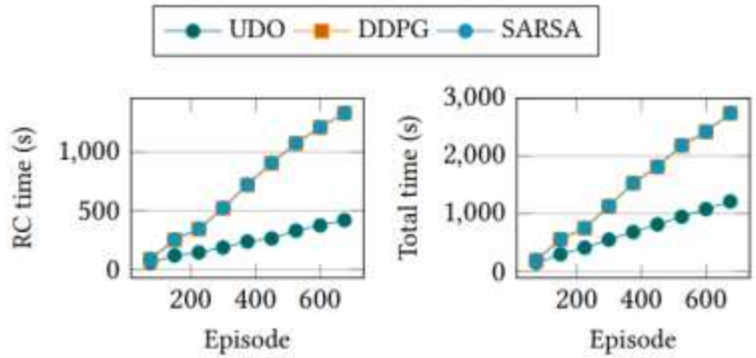
(b) TPC-H performance as a function of optimization time in Postgres.

...ring UDO to baselines on TPC-H.

UDO is always the best

Followed by DDPG++ with Dexter

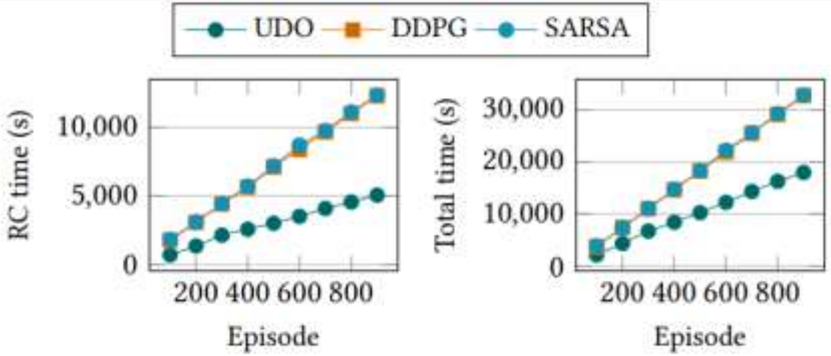
UDO vs BASELINES



(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.

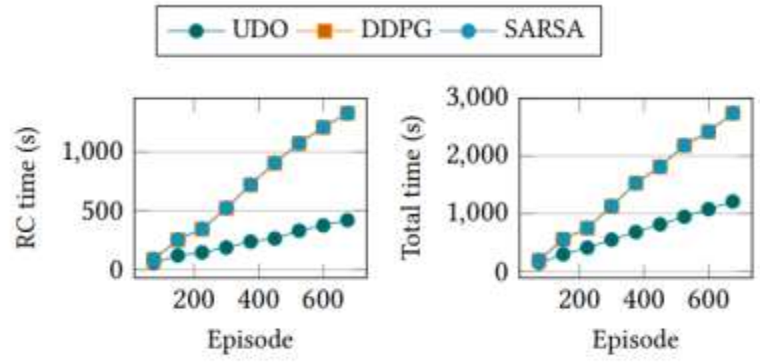


(a) Reconfiguration time of different RL algorithms for Postgres on TPC-H.

(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.

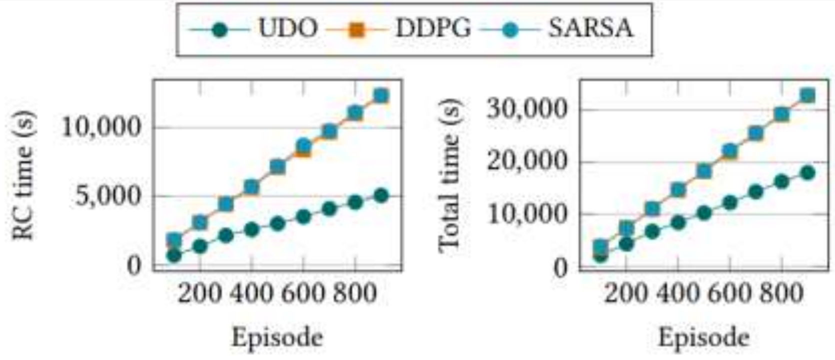
UDO vs BASELINES



(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.

UDO can reduce reconfiguration time by a factor of 3

UDO VARIANTS

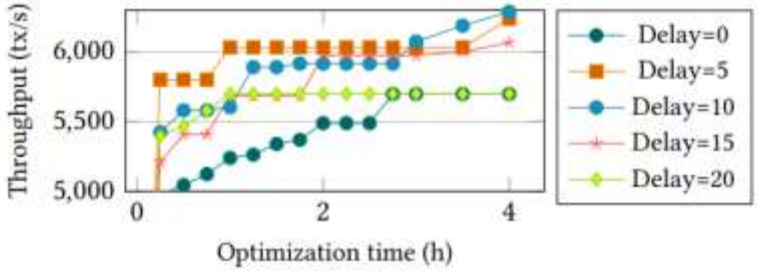


Figure 7: Impact of delayed feedback on UDO performance (MySQL on TPC-C).

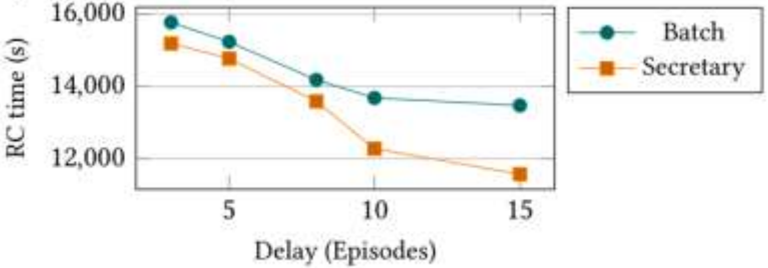
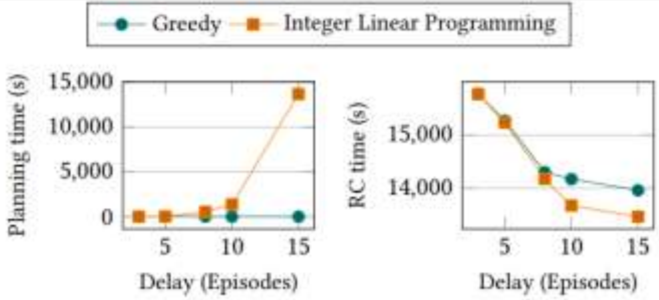


Figure 8: Impact of evaluation time selection on UDO performance (MySQL on TPC-C).



(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).

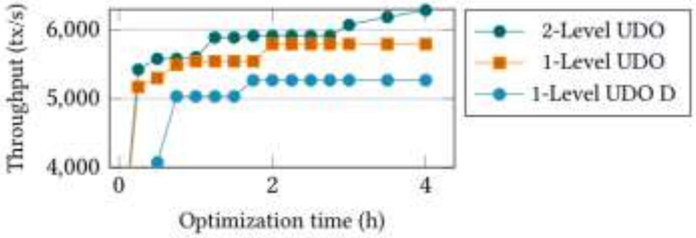
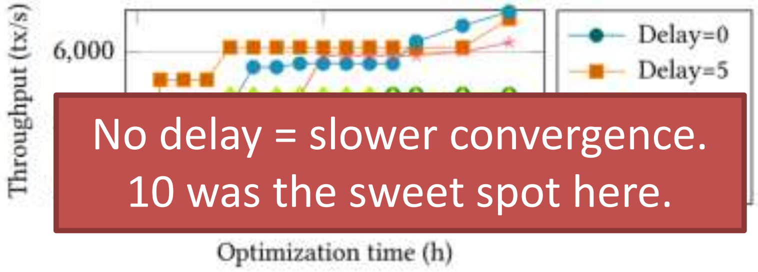


Figure 10: Impact of search space design and search strategy on UDO performance (MySQL on TPC-C).

UDO VARIANTS



No delay = slower convergence.
10 was the sweet spot here.

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

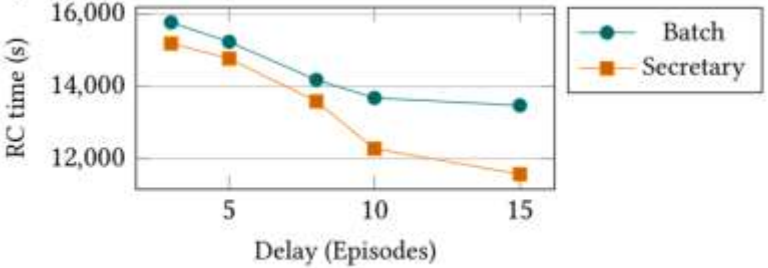
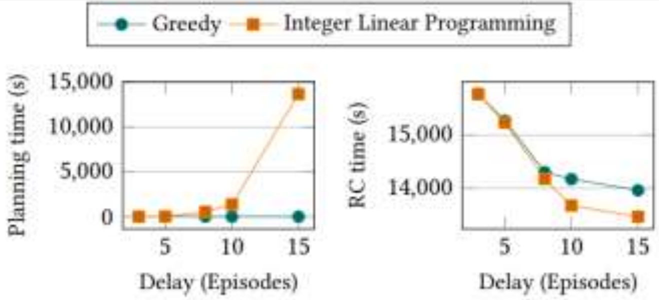


Figure 8: Impact of evaluation time selection on UDO performance (MySQL on TPC-C).



(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).

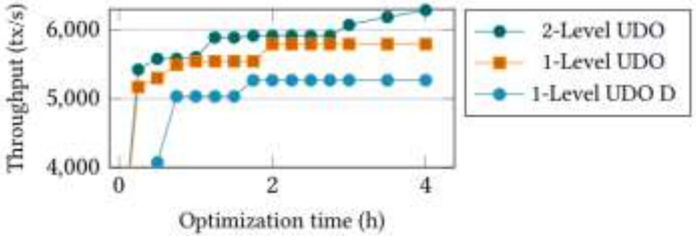
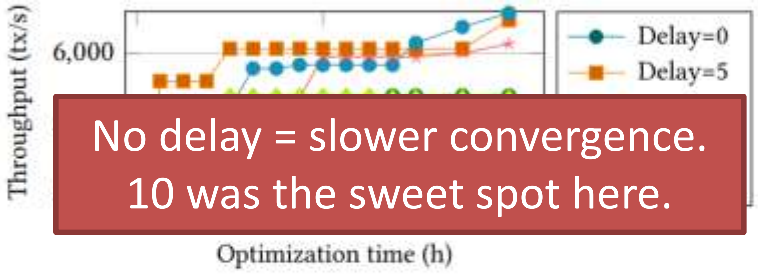


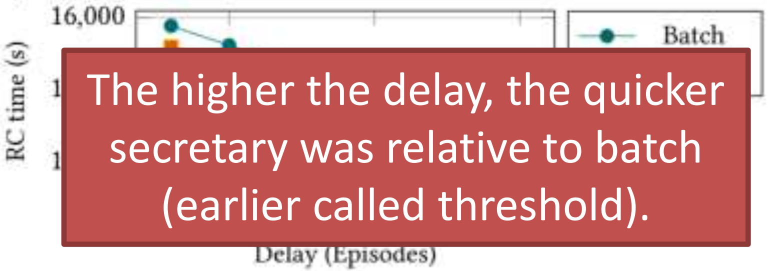
Figure 10: Impact of search space design and search strategy on UDO performance (MySQL on TPC-C).

UDO VARIANTS



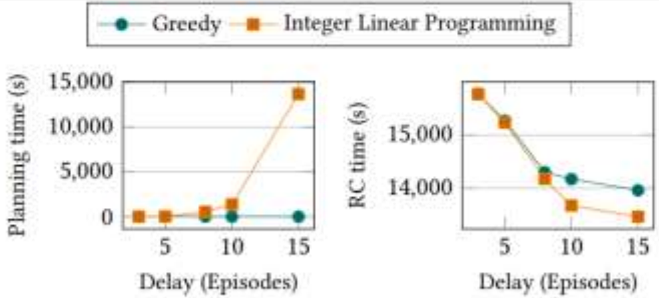
No delay = slower convergence.
10 was the sweet spot here.

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



The higher the delay, the quicker secretary was relative to batch (earlier called threshold).

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



(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).

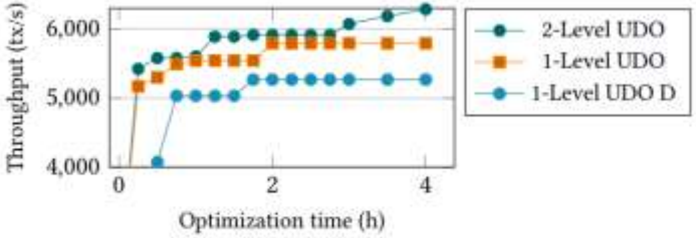
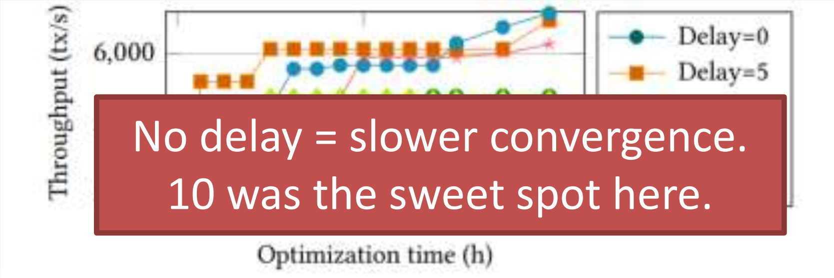


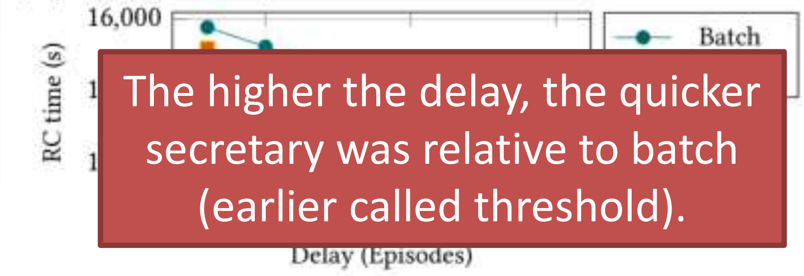
Figure 10: Impact of search space design and search strategy on UDO performance (MySQL on TPC-C).

UDO VARIANTS



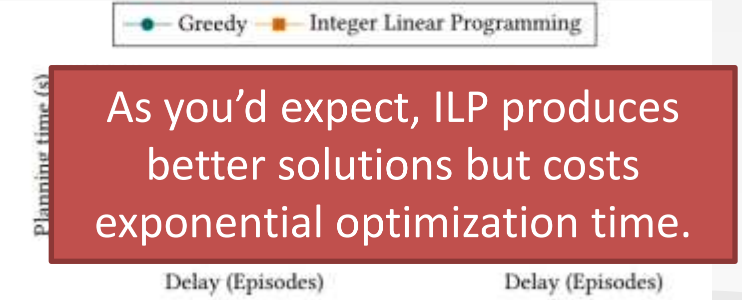
No delay = slower convergence. 10 was the sweet spot here.

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



The higher the delay, the quicker secretary was relative to batch (earlier called threshold).

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



As you'd expect, ILP produces better solutions but costs exponential optimization time.

(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).

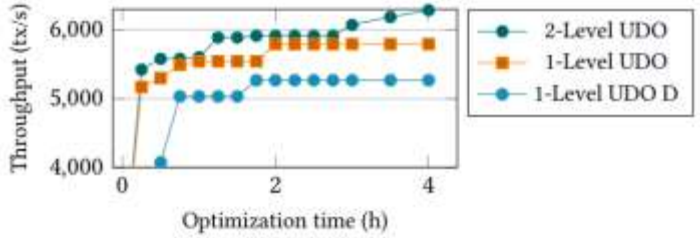
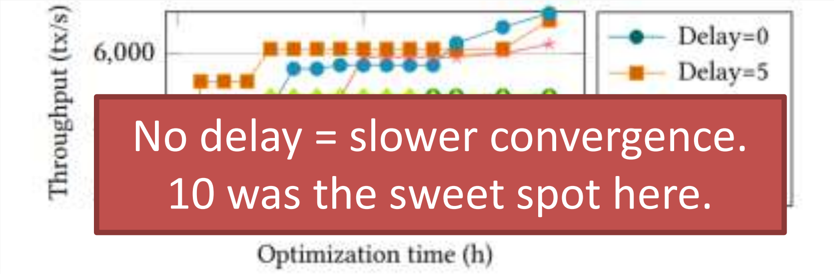


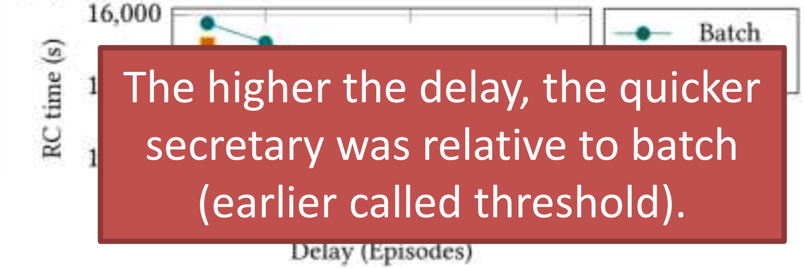
Figure 10: Impact of search space design and search strategy on UDO performance (MySQL on TPC-C).

UDO VARIANTS



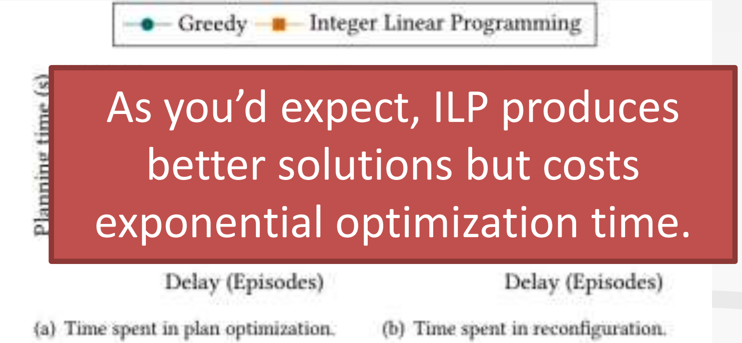
No delay = slower convergence.
10 was the sweet spot here.

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



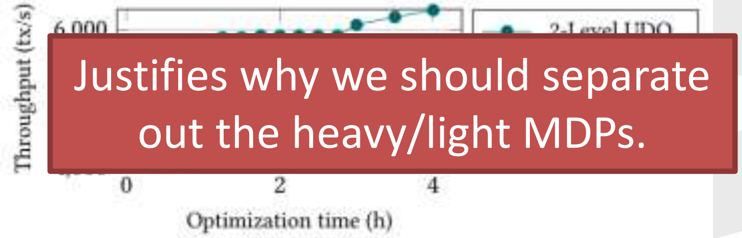
The higher the delay, the quicker secretary was relative to batch (earlier called threshold).

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



As you'd expect, ILP produces better solutions but costs exponential optimization time.

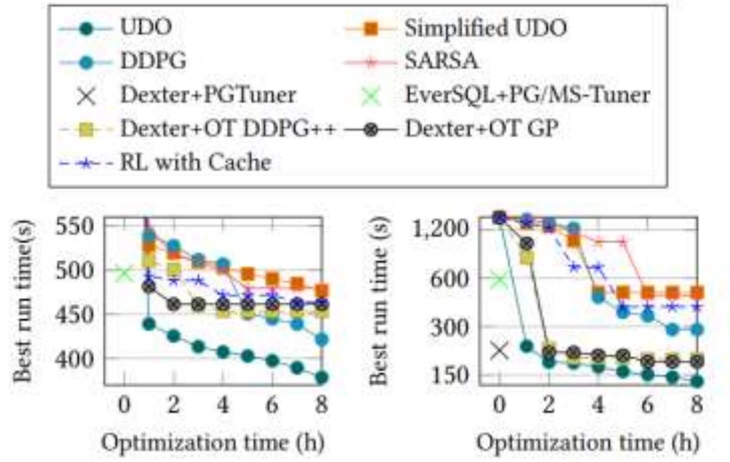
Figure 9: Impact of reconfiguration planning algorithm on UDO performance (MySQL on TPC-C).



Justifies why we should separate out the heavy/light MDPs.

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

SCENARIO VARIANTS



(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.

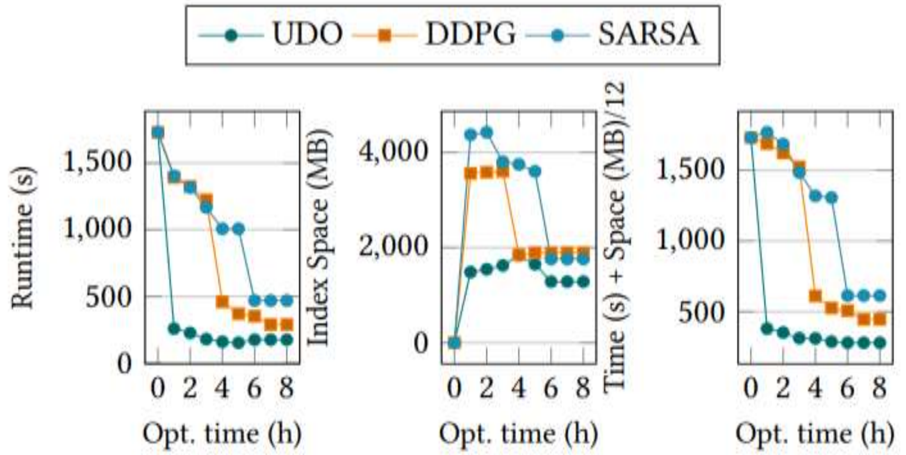
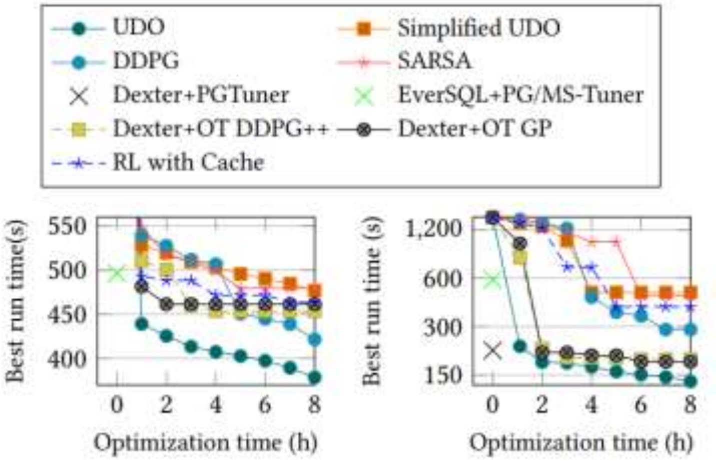


Figure 12: Optimizing weighted sum of run time and disk space for TPC-H SF 10 on Postgres.

SCENARIO VARIANTS



(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.

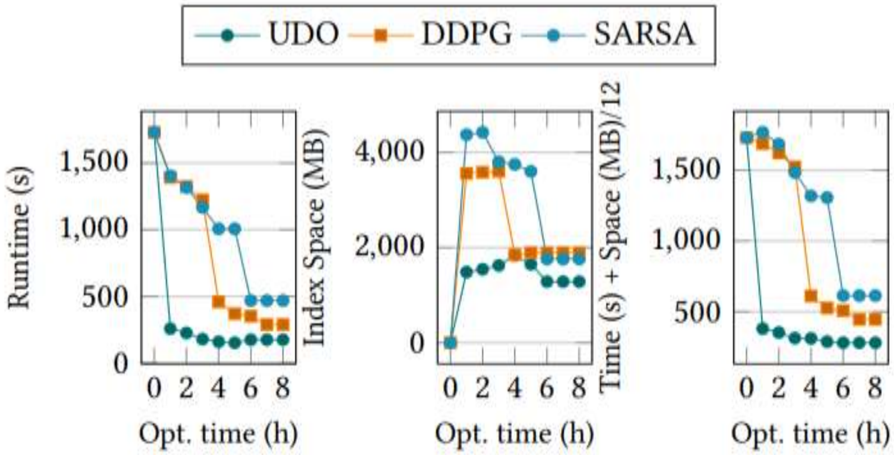
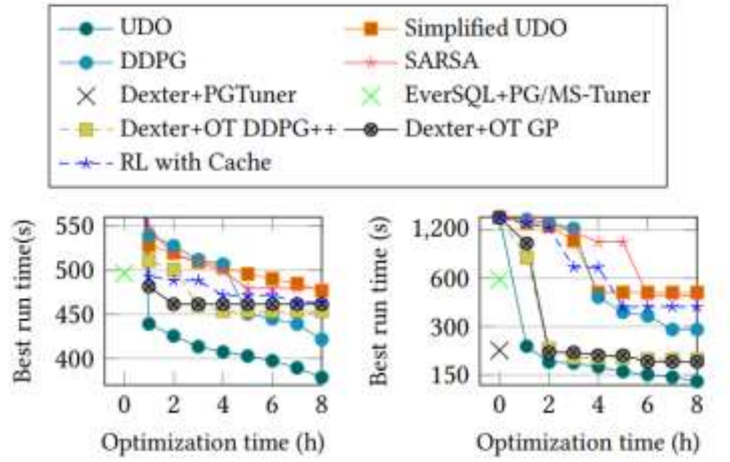


Figure 12: Optimizing weighted sum of run time and disk space for TPC-H SF 10 on Postgres.

Higher scalefactor, similar trends.

SCENARIO VARIANTS



(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.

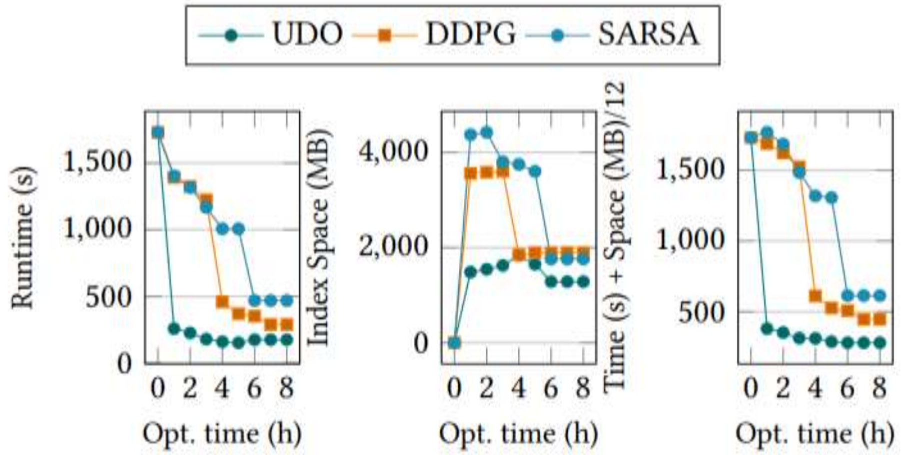
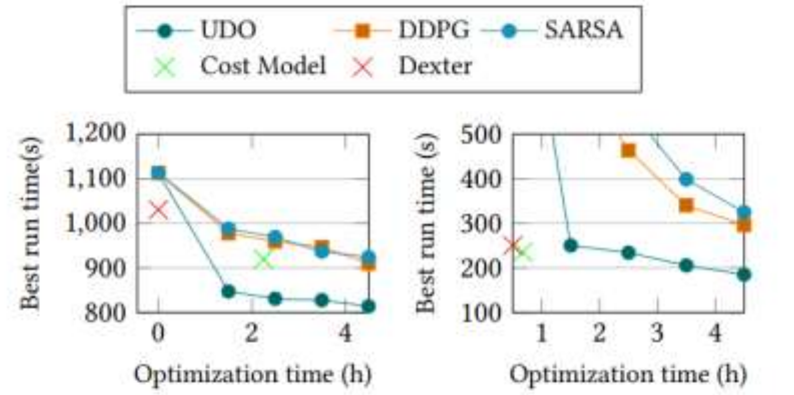


Figure 12: Optimizing weighted sum of run time and disk space for TPC-H SF 10 on Postgres.

Higher scalefactor, similar trends.

Multi-objective optimization.

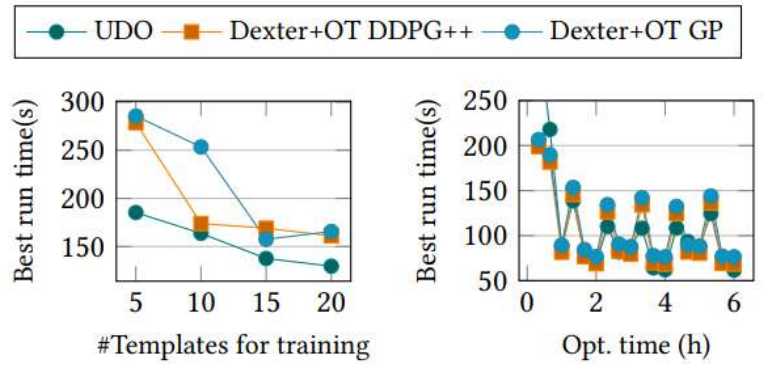
SCENARIO VARIANTS



(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).

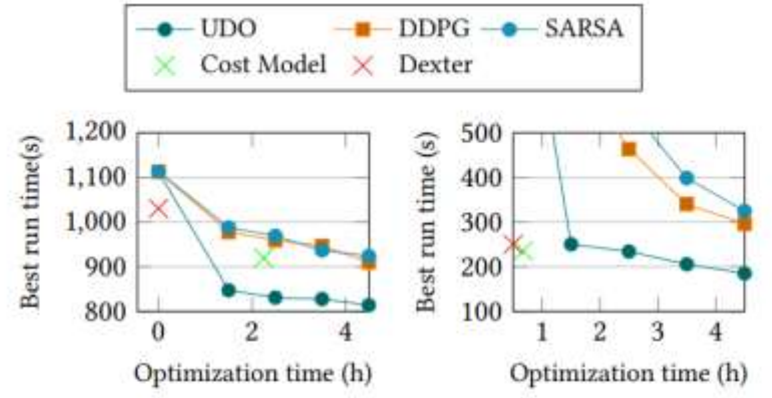


(a) Varying number of TPC-H query templates used for training.

(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).

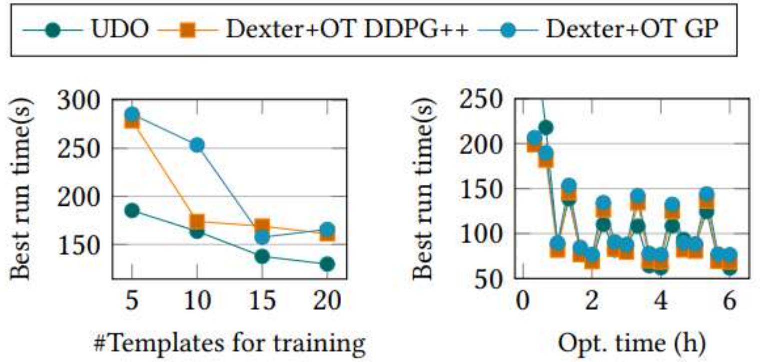
SCENARIO VARIANTS



(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).



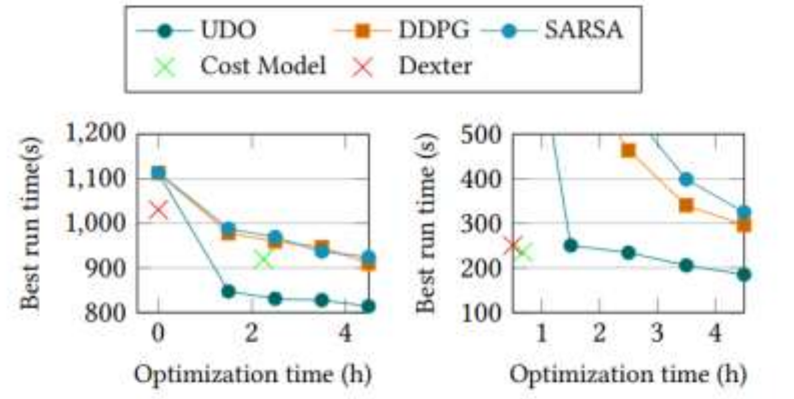
(a) Varying number of TPC-H query templates used for training.

(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).

Restricted to just indexes, still good.

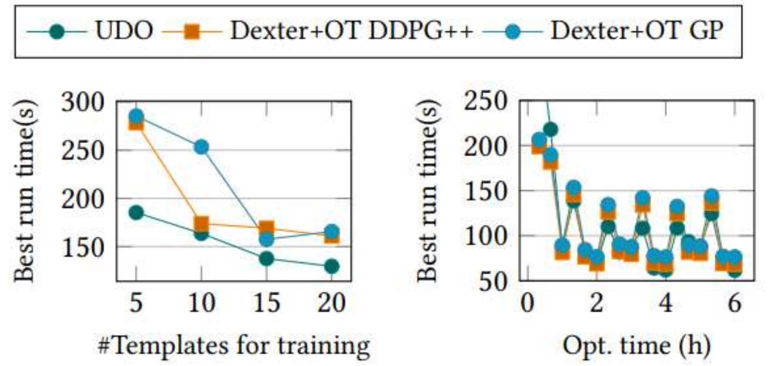
SCENARIO VARIANTS



(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.

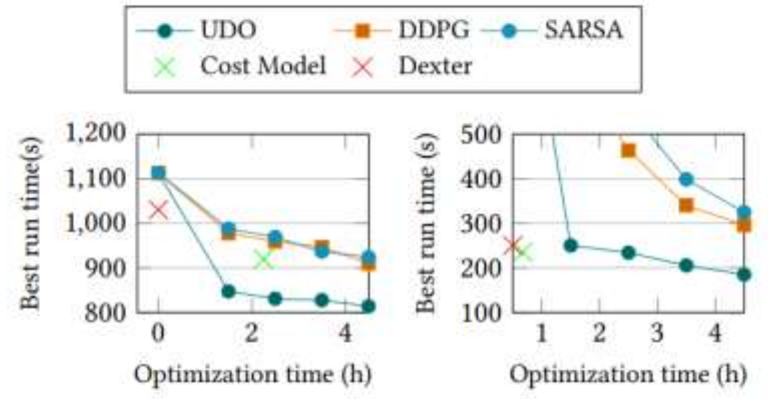
(b) Performance for dynamic workload switching every full hour.

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

Restricted to just indexes, still good.

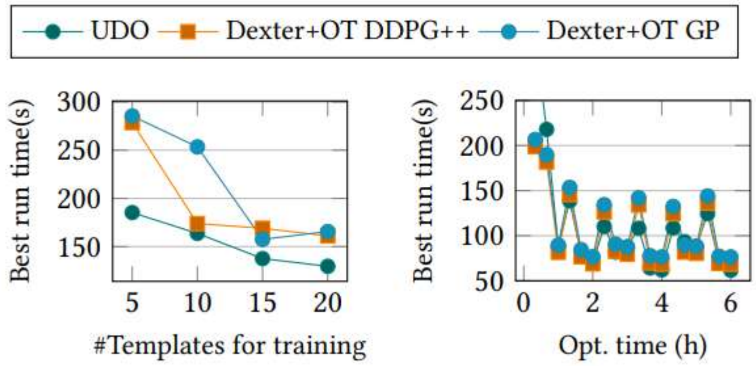
Workload shift is difficult.

SCENARIO VARIANTS



(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).

Restricted to just indexes, still good.

Workload shift is difficult.

Pingpong between even / odd TPC-H.

PARTING THOUGHTS

Parameters are not equal cost.

- Batch light parameters, multiple MDPs.
- delayed-HOO to account for delayed rewards.

Thoughts and commentary.

- Good use of both DBMS and RL domain knowledge; fig 10 cautionary of xkcd1838.
- *Universal*, counterexamples?
- Where do cost models still play a role? Other parts of the quality/overhead regime?