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


• 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.

<table>
<thead>
<tr>
<th>Tuning Quality</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td>UDO</td>
</tr>
<tr>
<td>low</td>
<td>&lt;bleep&gt;</td>
</tr>
<tr>
<td>high</td>
<td>cost models</td>
</tr>
<tr>
<td>low</td>
<td></td>
</tr>
</tbody>
</table>

<bleep> the dream
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.
Learning from sample runs

- **Algorithm**
  
  for config in search space
  try next action
  evaluate performance

- **What’s the problem?**

  Insert RL here
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.
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.
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 intxn.

- **Configuration**
  - \( c = \text{vector } [\text{parameter } \rightarrow \text{value}]. \)

- **Configuration space**
  - \( C = \{\text{all possible } c\}. \)
  - \( C_H : \text{heavy parameters}, \ C_L: \text{light parameters}, \)
  - \( C = C_H \times C_L. \)

- **Benchmark metric**
  - \( f : C \rightarrow \text{real number, stochastic}. \)

- **UDO instance**
  - \( (f, C), \text{find } c^* = \text{argmax } \mathbb{E}[f(c)] \)
Formal Model: UDO -> MDPs

Given a UDO instance \((f, C)\),
where the goal is to find \(c^* = \text{argmax } E[f(c)]\),
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 $c_L$, $f(c_H \circ c_L) - f(c_{default})$.
• 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 $f(c_H \circ c_L) - f(c_{default})$. 
UDO OVERVIEW

Iterate until the time limit is reached

• Note other stopping conditions could be used instead.

```
Algorithm 1 UDO main function.
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
```
**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.

```plaintext
function EVAL.RL.submit(config, deadline):
    R ← 0
    // Global variable representing evaluation requests
    Input: heavy configuration c₁ to evaluate and time t
    Effect: adds new evaluation request
    procedure EVAL.RL.submit(ε₁, t):
        R ← R ∪ \{c₁, t\}
        end procedure
    Input: RL algorithm Alg₁, benchmark metric f, time t, and space C₁
    Output: evaluated configurations with reward values
    function EVAL.RL.receive(Alg₁, f, C₁, 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 ← 0
        for s ∈ P.steps do
            // Prepare evaluation of next configurations
            CHANGECONF(s, hconf(ε₁))
            // Find (near-optimal) light parameter settings
            c₂ ← RL.OPTIMIZE(Alg₁, s, hconf, C₂, f)
            // Take performance measurements on benchmark
            b ← EVALUATE(f, s, hconf, c₂)
            // Add performance result to set
            E ← E ∪ \{c₂, 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.

```
1: function PickConf-Threshold(R, t)
2: "Was size threshold reached?"
3: if \(|R| \geq T\) then
4: return all requests
5: else
6: return 0
7: end if
8: end function

9: function PickConf-Secretary(R, t)
10: "Add requests whose deadline is reached"
11: E ← \{(e_j, t_j) ∈ R | t \geq t_j\}
12: "Remove requests from pending set"
13: R ← R \ E
14: "Iterate over requests"
15: for r = (e_j, t_j) ∈ R do
16: "Calculate re-configuration cost savings"
17: s ← CostSavings(r, E)
18: "Retrieve maximal savings so far"
19: m ← S(r)
20: "Should we evaluate?"
21: if t \geq (t_j - \delta) \geq \delta / s \wedge s > m then
22: E ← E U \{r\}
23: end if
24: "Update maximally possible savings"
25: S(r) ← max(m, s)
26: end for
27: return E
28: end function
```
Ordering configurations is NP-HARD
• Hamiltonian graph.

PlanConf
• Greedy algorithm.

Integer linear programming
• Optimal solution.

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:   $O \leftarrow []$
6:   // Iterate over all requests
7:   for $r \in R$ do
8:     // Find optimal insertion point
9:     $i \leftarrow \arg\min_{i=0,\ldots,|O|} \left( C_R(O[i-1], O[i]) + C_R(O[i], O[i+1]) \right)$
10:    // Insert current request there
11:    $O.insert(i, r)$
12:   end for
13:   return $O$
14: **end function**
### REINFORCEMENT LEARNING

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

<table>
<thead>
<tr>
<th>RL.SELECT</th>
<th>RL.UPDATE</th>
<th>RL.OPTIMIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pick the next action based on some statistics.</td>
<td>Update the statistics used by RL.SELECT.</td>
<td>Invoke the other two repeatedly for optimization.</td>
</tr>
<tr>
<td>See paper for details.</td>
<td>Update num visits to state-action pairs, present state, sample mean and variance of accumulated rewards.</td>
<td></td>
</tr>
</tbody>
</table>
THEORY

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

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

for a horizon \( T > 1 \), and 4/c-near-optimality dimension \( d \) of \( f \).

\[ \mathbb{E}[\text{Reg}_T] = O \left( C \left(1 + \tau\right) T^{\frac{1}{2}} (\log T)^{\frac{1}{2}} \right) \]

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

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

under the assumptions of Thm. 6.1. Here, \( \text{HOO}(T_i) \geq O \left( [\log T_{th}]^{\frac{1}{2}} \right) \).

Deviation in expected performance of the configuration returned by UDO from the optimum is \( O \left( (1 + \tau) [\text{HOO}^2(T_i) \text{HOO}(T_{th})]^{\frac{1}{2}} \right) \).

Here, \( T_{th} \) and \( T_i \) 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 \( T_{th}, T_i \to \infty \).
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**

---

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

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

UDO is always the best
Followed by DDPG++ with Dexter
U DO 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

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).

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

The higher the delay, the quicker secretary was relative to batch (earlier called threshold).
No delay = slower convergence. 10 was the sweet spot here.

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

As you’d expect, ILP produces better solutions but costs exponential optimization time.
UDO VARIANTS

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

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

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

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

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

Higher scalefactor, similar trends.

Multi-objective optimization.
SCENARIO VARIANTS

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

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

(a) Varying number of TPC-H query templates used for training.  
(b) Performance for dynamic workload switching every full hour.
**SCENARIO VARIANTS**

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

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

(a) Varying number of TPC-H query templates used for training.
(b) Performance for dynamic workload switching every full hour.

Restricted to just indexes, still good.
**SCENARIO VARIANTS**

Restrict to just indexes, still good.

Workload shift is difficult.

---

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

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

(a) Varying number of TPC-H query templates used for training. (b) Performance for dynamic workload switching every full hour.
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?