QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning

Guoliang Li¹, Xuanhe Zhou¹, Shifu Li², Bo Gao²
¹Department of Computer Science, Tsinghua University, Beijing, China
²Huawei Company

Presentation by Antonia Boca | R244
Motivation: Database tuning

- Parameters that can be tuned: *max cache size, max concurrent threads, max RAM etc.*
- Manual tuning can take up to several days
- Automatic tuning:
  - Rule-based: BestConfig
  - Learning-based:
    - Traditional ML system: OtterTune [relies on a large number of high-quality training examples from DBAs’ experience data, which are rather hard to obtain]
    - Deep Reinforcement Learning: CDBTune

- CDBTune’s limitation: not using query information!
The Architecture (single query)

```
select ID, Name
from PERSON;
```

Query2Vector

```
[1, 0, 1, ..., 0.021, 0.33]
```

Tuner

(Predictor + DS-DDPG model)

Database configuration
The Architecture (multiple queries)
The Architecture (clusters)

```
(select ID, Name from PERSON)
 查询向量 (Query2Vector)
 [1, 0, 1, ..., 0.021, 0.33]
 向量到模式 (Vector2Pattern)
```

```
数据库配置 (Database configuration)
 调优器 (Tuner)
 (Predictor + DS-DDPG model)
 模式到聚类 (Pattern2Cluster)
```
The Architecture (clusters)

Database configuration

select ID, Name from PERSON;

Query2Vector

[1, 0, 1, ..., 0.021, 0.33]

Tuner (Predictor + DS-DDPG model)

Vector2Pattern

Pattern2Cluster
Training the Tuner (DS-DDPG)

Agent

Critic → Action → Actor

Observation S' → Action

Environment

S' = S + ΔS

Outer metric (S) → Inner state

Predictor

[1, 0, 1, ..., 0.021, 0.33]
Training the Tuner (DS-DDPG)

Agent
Critic
Score
Action
Reward
Observation S'
Environment
S' = S + ΔS
Outer metric (S) Inner state
Actor
Action

Predictor
[1, 0, 1, ..., 0.021, 0.33]
Training the Tuner (DS-DDPG)

Agent

Critic

Score

Actor

Action

Reward

Environment

Observation $S'$

$S' = S + \Delta S$

Outer metric ($S$)  Inner state

$S'$ = [1, 0, 1, ..., 0.021, 0.33]

Predictor

[1, 0, 1, ..., 0.021, 0.33]
The Tuner

The Tuner is a framework for tuning parameters in a system. It consists of four main components:

1. **Agent**: This component evaluates the system and provides feedback on the performance. It receives an observation of the system state and provides an action to improve the system.

2. **Critic**: The critic evaluates the actions taken by the agent and provides a score based on the performance of the action. The critic receives the action and the system state as input and outputs a score.

3. **Actor**: The actor takes the action provided by the agent and applies it to the system. It also receives the system state and provides an observation of the system state after the action is applied.

4. **Environment**: The environment is the system that the agent operates on. It provides the initial state, receives actions from the actor, and updates the state based on the action. The environment also provides rewards to the agent based on the performance.

The outer metric (S) and inner state are updated as follows:

\[ S' = S + \Delta S \]

The inner state is represented as:

\[ [1, 0, 1, ..., 0.021, 0.33] \]

The diagram illustrates the flow of information and the interaction between these components.
Evaluation

- Bulk of evaluation done on PostgreSQL with 3 datasets;
- Discrete Cluster-level tuning achieves the best throughput;
- Query-level tuning achieves the best latency.
Evaluation

• QTune **outperforms** all other SOTA methods on all types of tuning

• Qtune **generalizes** to other databases, datasets, and hardware platforms
Limitations

- Cost information is dependent on the SQL query optimizer;
- Their feature vectorization method makes it hard to add or delete new tables;
- Paper is unclear on whether QTune is fine-tuned before being evaluated on different databases/hardware platforms;
- Paper does not provide training metrics (e.g. loss, acc, hyperparameters);
- Evaluation is done only on open-source DBMSs;
- Did not provide cluster-level evaluation on one of the datasets;
Conclusion

• QTune’s DRL model is not a novel idea
  • CDBTune uses the same actor-critic architecture
• It’s innovation comes from:
  • **query-awareness**
    • Paper provides a feature vectorization method
    • Also provides a way to predict the cost of an SQL query
  • **Clustering approach**
    • They discretize feature vectors for faster clustering
    • They show how this achieves both **high throughput** and **low latency**
• Not much follow-up work
  • paper suggests the method is used in the Huawei data centres
  • but the paper was **cited >100 times!**
Any questions?
## Appendix 1: Overhead

<table>
<thead>
<tr>
<th>Database</th>
<th>Featurization</th>
<th>Tuner</th>
<th>Vector2Pattern</th>
<th>Clustering</th>
<th>Recommendation</th>
<th>Execution</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>9.37 ms</td>
<td>2.23 ms</td>
<td>0.29 ms</td>
<td>1.64 ms</td>
<td>4.36 ms</td>
<td>0.45 s - 262.9 s</td>
<td>3.8 % - 0.0068 %</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>9.46 ms</td>
<td>2.38 ms</td>
<td>0.39 ms</td>
<td>2.51 ms</td>
<td>5.01 ms</td>
<td>0.46 s - 263.3 s</td>
<td>4.1 % - 0.0075 %</td>
</tr>
<tr>
<td>MongoDB</td>
<td>13.48 ms</td>
<td>2.16 ms</td>
<td>0.36 ms</td>
<td>2.32 ms</td>
<td>4.31 ms</td>
<td>0.63 s - 264.5 s</td>
<td>3.5 % - 0.0085 %</td>
</tr>
</tbody>
</table>

Table 5: Time distribution of queries in JOB (RO) benchmark on MySQL, PostgreSQL and MongoDB respectively. Execution is the range of time the database executes a query. Overhead is the percentage of tuning in the total time for a query.
Appendix 2: Experiment settings

Table 2: Database information

<table>
<thead>
<tr>
<th>Database</th>
<th>Knobs without restart</th>
<th>State Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>64</td>
<td>19</td>
</tr>
<tr>
<td>MySQL</td>
<td>260</td>
<td>63</td>
</tr>
<tr>
<td>MongoDB</td>
<td>70</td>
<td>515</td>
</tr>
</tbody>
</table>

Table 3: Workloads. RO, RW and WO denote read-only, read-write and write-only respectively.

<table>
<thead>
<tr>
<th>Name</th>
<th>Mode</th>
<th>Table</th>
<th>Cardinality</th>
<th>Size(G)</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB</td>
<td>RO</td>
<td>21</td>
<td>74,190,187</td>
<td>13.1</td>
<td>113</td>
</tr>
<tr>
<td>TPC-H</td>
<td>RO</td>
<td>8</td>
<td>158,157,939</td>
<td>50.0</td>
<td>22</td>
</tr>
<tr>
<td>Sysbench</td>
<td>RO, RW</td>
<td>3</td>
<td>4,000,000</td>
<td>11.5</td>
<td>474,000</td>
</tr>
</tbody>
</table>

Table 4: The number of training samples for the DL model in query clustering, the Predictor and the Actor-Critic module in DS-DDPG.

<table>
<thead>
<tr>
<th>Name</th>
<th>Sysbench</th>
<th>JOB</th>
<th>TCP-H</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL</td>
<td>3792</td>
<td>8000</td>
<td>40,000</td>
</tr>
<tr>
<td>Predictor</td>
<td>3792</td>
<td>8000</td>
<td>40,000</td>
</tr>
<tr>
<td>Actor-Critic</td>
<td>1500</td>
<td>480</td>
<td>300</td>
</tr>
</tbody>
</table>

Table 6: Two hardware configurations

<table>
<thead>
<tr>
<th>Instance</th>
<th>RAM (GB)</th>
<th>Disk (GB)</th>
<th>CPU (GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>16</td>
<td>780</td>
<td>2.49</td>
</tr>
<tr>
<td>B</td>
<td>128</td>
<td>5000</td>
<td>4.00</td>
</tr>
</tbody>
</table>
Appendix 3: Generalisation

Figure 9: Performance when workload changes on PostgreSQL.

Figure 10: Performance for different databases.

Figure 11: Performance for different hardware environments.
Appendix 4: Training details

Function TrainPredictor($\pi_P, T_P$)

**Input:** $\pi_P$: The weights of a neural network; $T_P$: The training set

1. Initiate the weights in $\pi_P$;
2. while !converged do
3.     for each $(v, S, I, \Delta S) \in T_P$ do
4.         Generate the output $G$ of $(v, S, I)$;
5.         Accumulate the backward propagation error:
6.             $E = E + \frac{1}{2} ||G - \Delta S||^2$;
7.         Compute gradient $\nabla_{\theta_A}(E)$, update weights in $\pi_P$;

Function TrainAgent($\pi_A, \pi_C, T_A$)

**Input:** $\pi_A$: The actor’s policy; $\pi_C$: The critic’s policy; $T_A$: training data

1. Initialize the actor $\pi_A$ and the critic $\pi_C$;
2. while !converged do
3.     Get a training data $T_A^i = (S_i^t, A_i, R_i)$, $(S_2^t, A_2, R_2), \ldots, (S_t^t, A_t, R_t)$;
4.     for $i = t - 1$ to 1 do
5.         Update the weights in $\pi_A$ with the action-value $Q(S_{i+1}^t, A_{i+1}^t|\pi_C)$;
6.         Estimate an action-value
7.             $Y_i = R_i + \tau Q(S_{i+1}^t, \pi_A(S_{i+1}^t|\theta^A)|\pi_C)$;
8.         Update the weights in $\pi_C$ by minimizing the loss value $L = (Q(S_i^t, A_i|\pi_C) - Y_i)^2$;

Algorithm 1: Training DS-DDPG

**Input:** $U$: the query set $\{q_1, q_2, \ldots, q_{\lceil \theta \rceil}\}$

**Output:** $\pi_P, \pi_A, \pi_C$

1. Generate training data $T_P$;
2. TrainPredictor($\pi_P, T_P$);
3. Generate training data $T_A$;
4. TrainAgent($\pi_A, \pi_C, T_A$);