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

Paper review by Victor

Structure

1. Motivations
2. How QTune works
3. Results
4. Review

Motivations

- Automatically find the best config for database
- Minimize latency
- Maximize throughput
- Individual queries or groups of queries

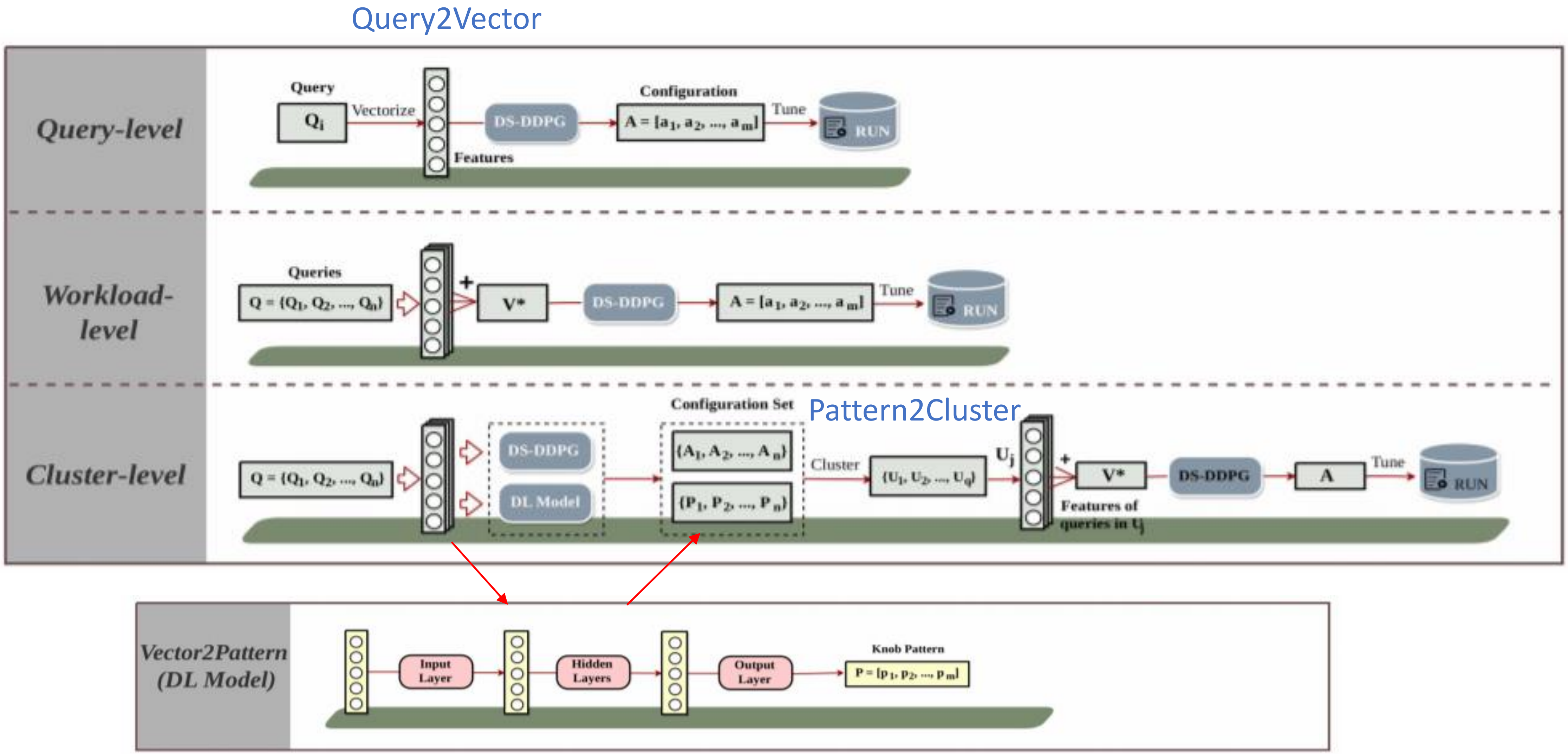
Existing solutions

- BestConfig, OtterTune, CDBTune
- Need large volume of high-quality training data
- Coarse-grained tuning for queries (Read-only, entire workload etc…)
- Doesn’t consider both actions and queries change the environment

QTune - Overview

- Query tuning
  - Minimizes latency
- Workload tuning
  - Maximizes throughput
- Cluster tuning
  - Group queries into different clusters
  - Optimal latency-throughput trade-off
- RL: Double-State Deep Deterministic Policy Gradient (DS-DDPG)

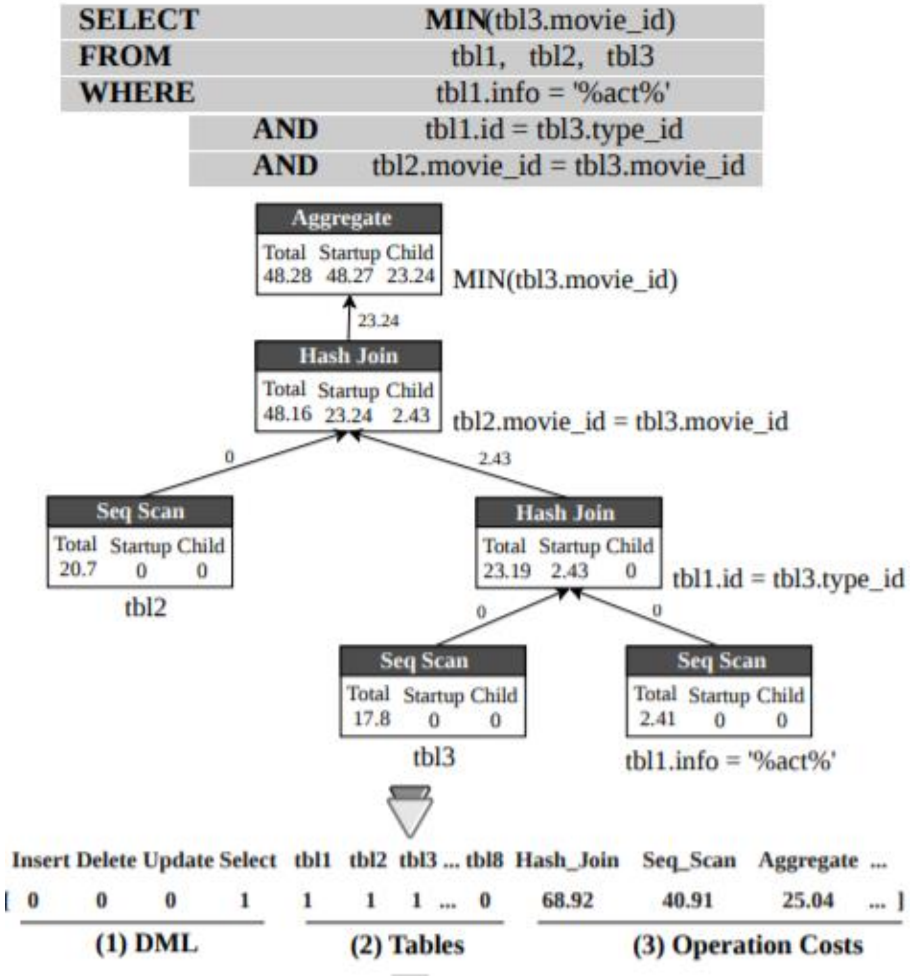
QTune - Overview



QTune – Query Featurization

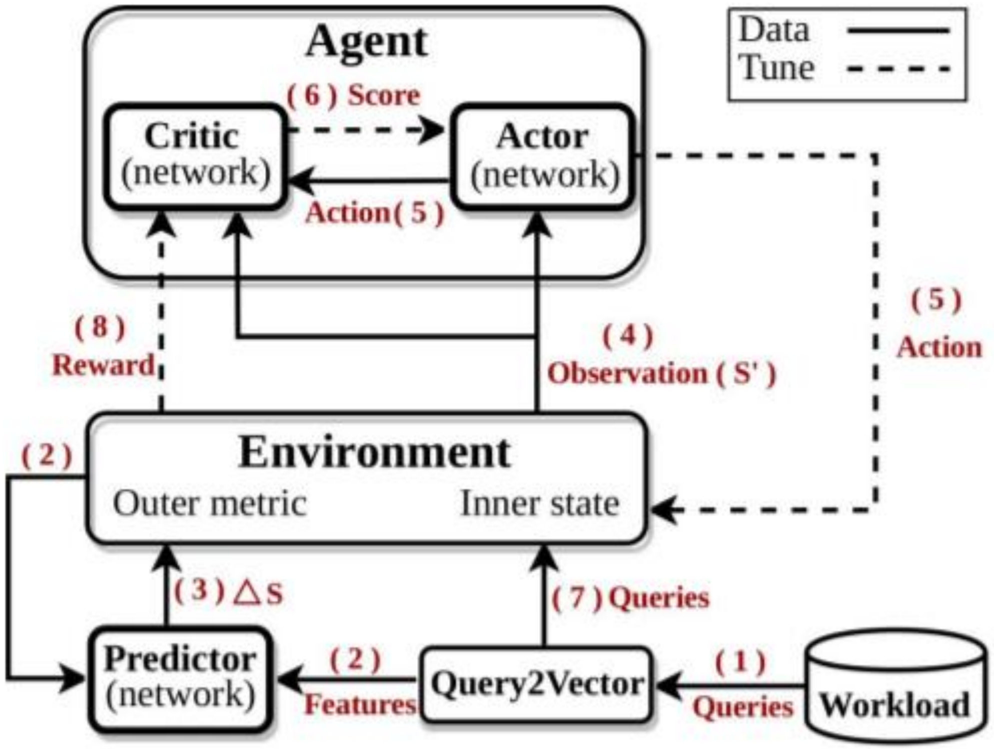
- “Query2Vector” in paper
- Turn a query into a feature vector
- Query type (Select, insert, update, delete) 4-bits of boolean flags
- Tables involved |T|-bits of boolean flags
- No operations (join, groupby etc…)!
- Cost information from query plan (from query optimizer) |P| floats
- Sum up the cost for each operation and normalize
- Unify multiple feature vectors into one:
    - Union query types (bitwise OR)
    - Sum up tables (?)
    - Sum up costs

QTune – Query Featurization



QTune – DS-DDPG

- Double state: both inner state and outer metrics
- DDPG solves the problem of infinite actions (continuous config values)
- Predictor is also DL model
- 3) Predicts change in outer metrics but not the next set of actual metrics (S’ = S + ∆S)



QTune – DS-DDPG – Predictor

- Training data: [(query feature vector, outer metrics, inner state, change in S)]
- Four fully connected layers
- ReLU in hidden layers
- MSE for loss function

QTune – DS-DDPG – Actor-Critic

- Training data: [(S', Action, Reward)] from list of queries
- Update actor policy with gradient of Q-value and state
- Estimate actual state-action value with Bellman equation, reward and Q-value
- Calculate loss with squared error

QTune – DS-DDPG – Reward function

1. Define percentage changes

$$\Delta_{0,t} = \begin{cases} \frac{m_t - m_0}{m_0}, & \textit{the higher the better} \\ \frac{m_0 - m_t}{m_0}, & \textit{the lower the better} \end{cases}$$

$$\Delta_{t-1,t} = \begin{cases} \frac{m_t - m_{t-1}}{m_{t-1}}, & \textit{the higher the better} \\ \frac{m_{t-1} - m_t}{m_{t-1}}, & \textit{the lower the better} \end{cases}$$

2. Calculate a specific metric m

$$r_m = \begin{cases} ((1 + \Delta_{t-1,t})^2 - 1)|1 + \Delta_{0,t}|, & \Delta_{0,t} > 0 \\ -(((1 - \Delta_{t-1,t})^2 - 1)|1 - \Delta_{0,t}|), & \Delta_{0,t} \leq 0 \end{cases}$$

3. Weighted mean for combining multiple metrics into reward R

$$R = \sum w_m r_m$$

4. (Same as CDBTune)

QTune – Query clustering

- DS-DDPG has access to the configs tried for a specific query
- But expensive to use DS-DDPG to get continuous values
- Turn continuous values into discrete values like {-1, 0, 1}
- Only use the top-k frequently changed configs
- Again train a DL model to map queries to discrete config pattern vectors
- Training data: [(query vector, discrete recommended config from DS-DDPG)]
- Use DBSCAN (clustering algorithm) to group the discrete config vectors

Evaluation and results

- Evaluation on QTune techniques and internals
- Evaluation on config tuning as compared to existing methods
- Evaluation on ability to generalize with changes in a few factors

Table 2: Database information

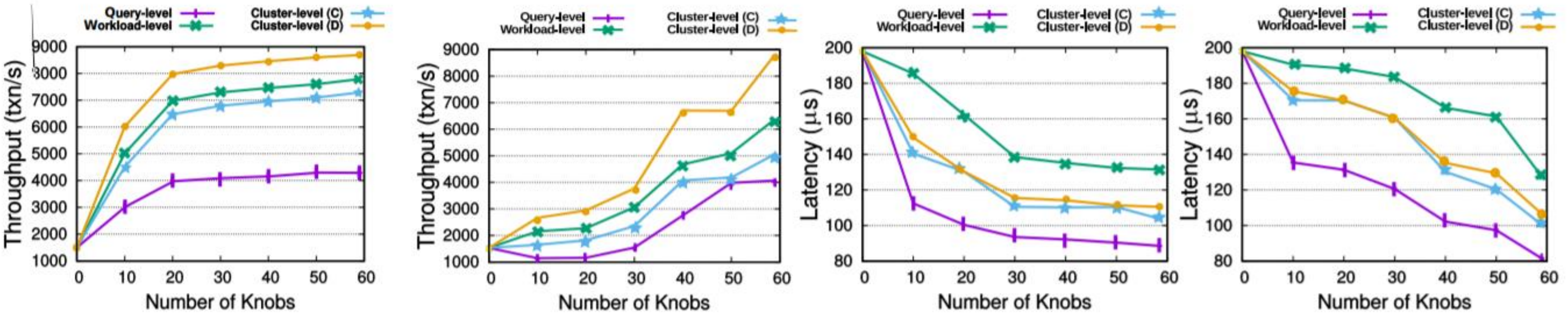
Database	Knobs without restart	State Metrics
PostgreSQL	64	19
MySQL	260	63
MongoDB	70	515

Name	Sysbench	JOB	TCP-H
DL	3792	8000	40,000
Predictor	3792	8000	40,000
Actor-Critic	1500	480	300

- Single machine with 128GB RAM, 5TB disk, and 4GHz CPU
- Huawei Gauss Database

Evaluation and results – QTune

- Three tuning methods



(a) IF-Throughput (b) RC-Throughput (c) IF-Latency (d) RC-Latency

Figure 6: Performance by increasing knobs in Important First (IF) and Randomly Choosing (RC) respectively when running Sysbench (RO) on PostgreSQL.

- Throughput: Cluster > Workload > Query
- Latency: Query > Cluster > Workload (> means better)

Evaluation and results – QTune

- Running time

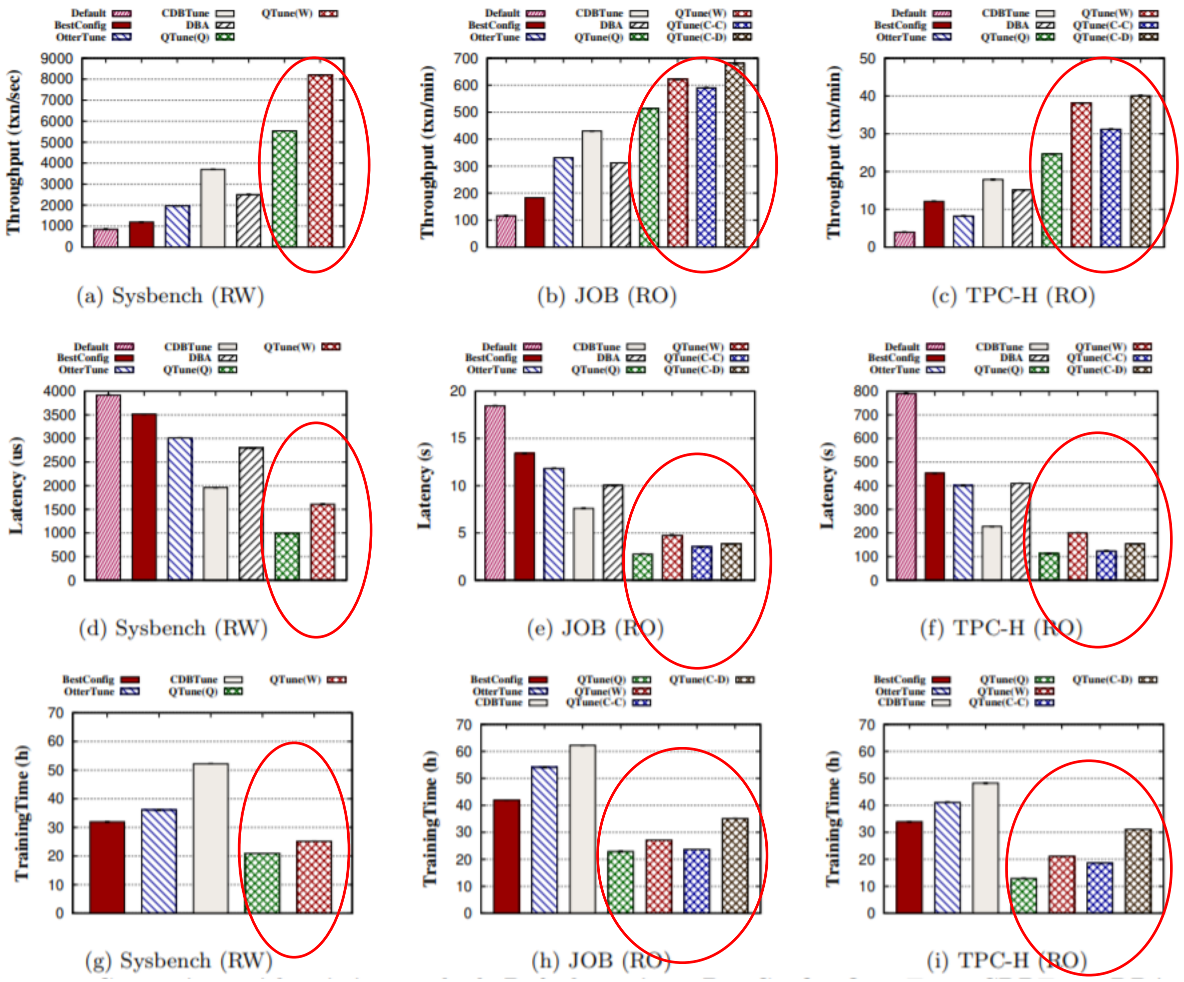
Database	Featurization	Tuner	Vector2Pattern	Clustering	Recommendation	Execution	Overhead
MySQL	9.37 ms	2.23 ms	0.29 ms	1.64 ms	4.36 ms	0.45 s - 262.9 s	3.8 % - 0.0068 %
PostgreSQL	9.46 ms	2.38 ms	0.39 ms	2.51 ms	5.01 ms	0.46 s - 263.3 s	4.1 % - 0.0075 %
MongoDB	13.48 ms	2.16 ms	0.36 ms	2.32 ms	4.31 ms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

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.

- Low overhead during running time (and probably only running time)

Evaluation and results – Comparison with others

- BestConfig, OttorTune, CDBTune, and database admins

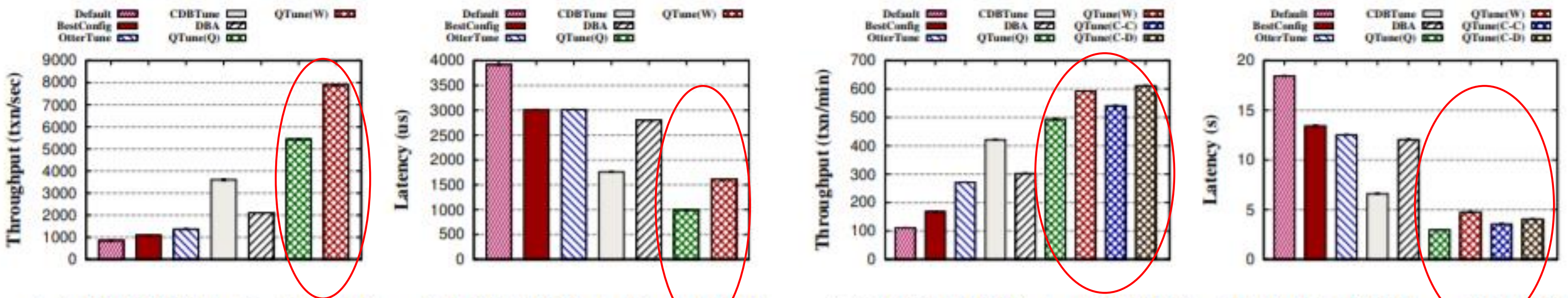


Evaluation and results – Comparison with others

- QTune achieves the highest throughput and lowest latency
- Better than CDBTune because CDBTune only has outer metrics in environment

Evaluation and results – Adaptability to changes

- Use models trained with one workload to tune PostgreSQL under another workload



(a) JOB(RO) to Sysb.(RW) (b) JOB(RO) to Sysb.(RW) (c) TPC-H(RO) to JOB(RO) (d) TPC-H(RO) to JOB(RO)

- QTune performs the best because it takes queries into account

Evaluation and results – Adaptability to changes

- Evaluate QTune on different databases

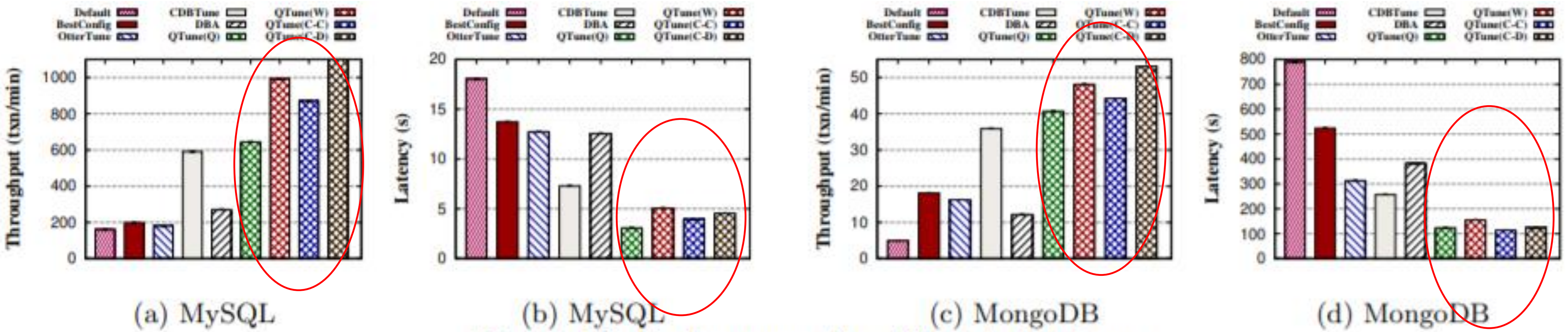


Figure 10: Performance for different databases.

- QTune outperforms other methods

Review

- Improvement upon CDBTune
- The lead author (Li) was involved in CDBTune
- Difference is query featurization, clustering according to pattern, and predictor model
- But makes the whole system really complicated (2 + 2 + 1 = 5 models)
- Probably takes a long time to train and tune the system itself

Discussion