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

Guoliang Li[1], Xuanhe Zhou[1], Shifu Li[2], Bo Gao[2]

[1]Department of Computer Science, Tsinghua University, Beijing, China

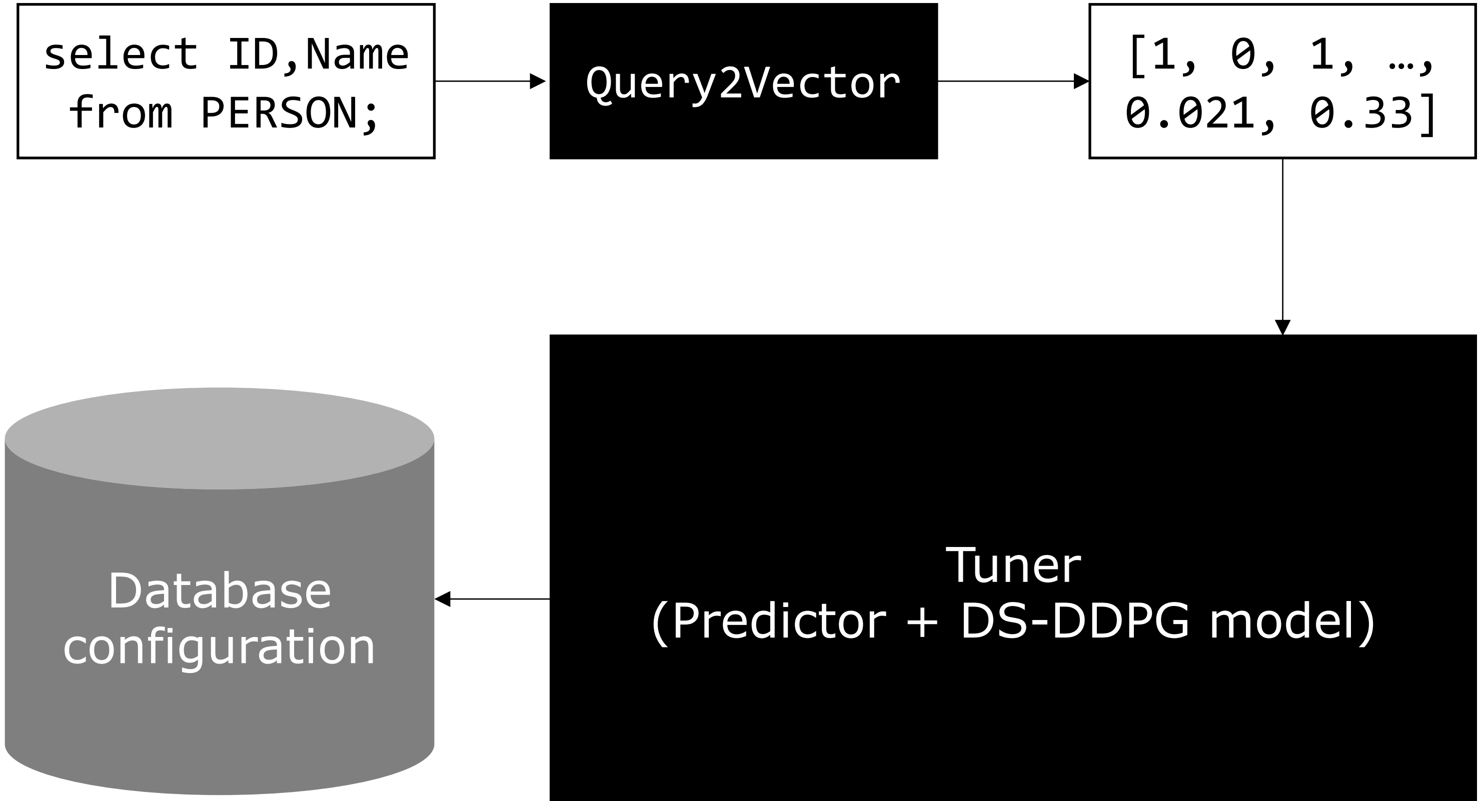
[2]Huawei Company

Presentation by Antonia Boca | R244

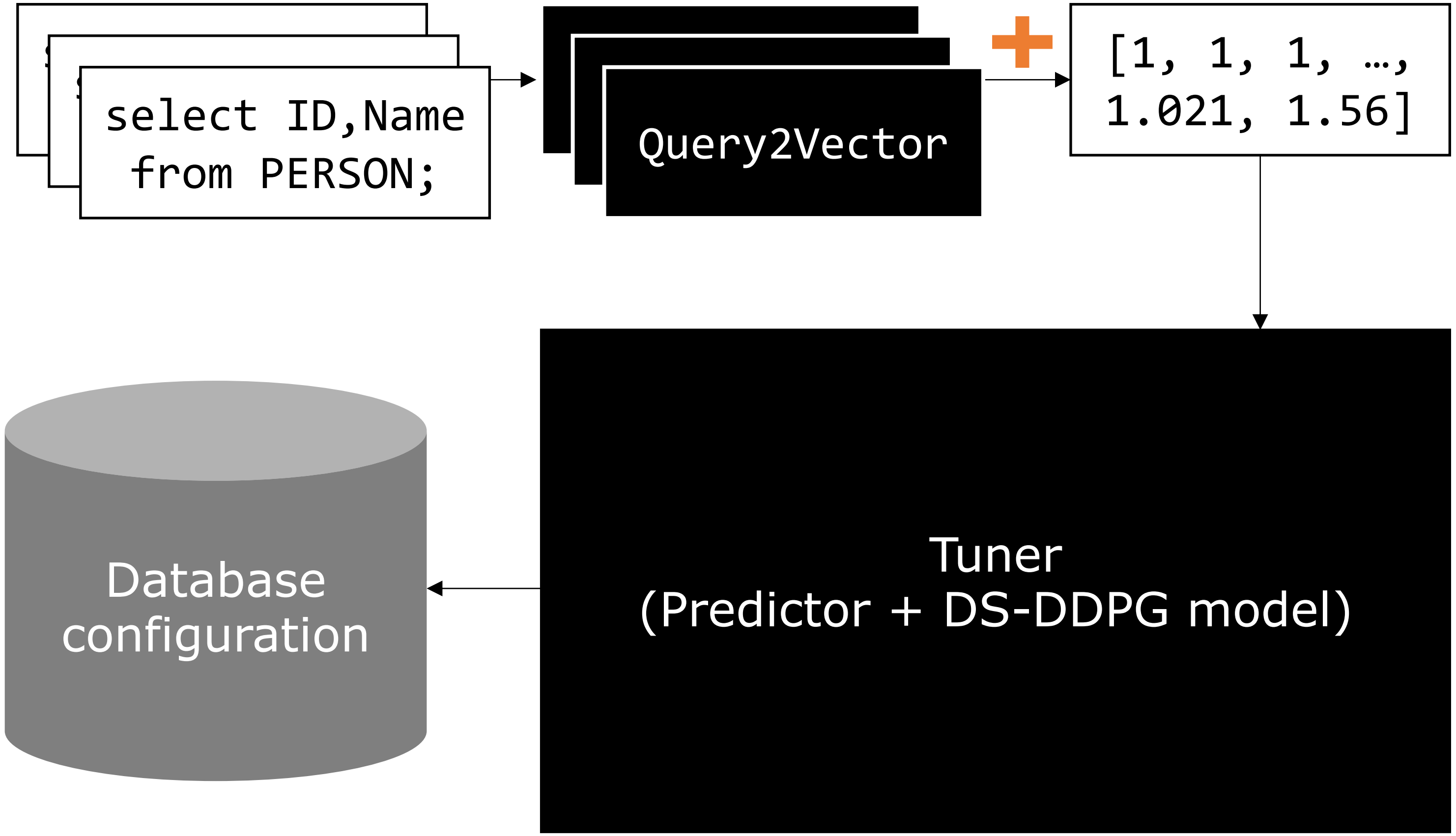
Motivation: Database tuning

- 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!

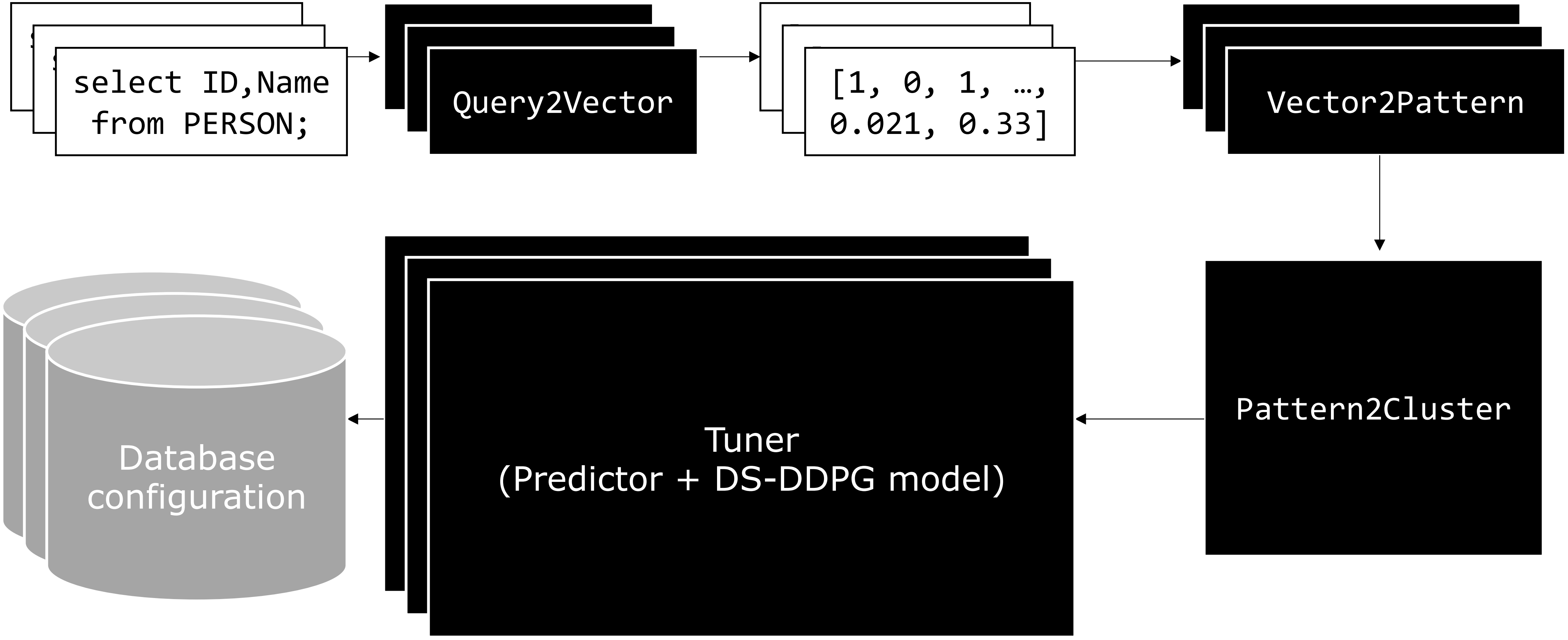
The Architecture (single query)



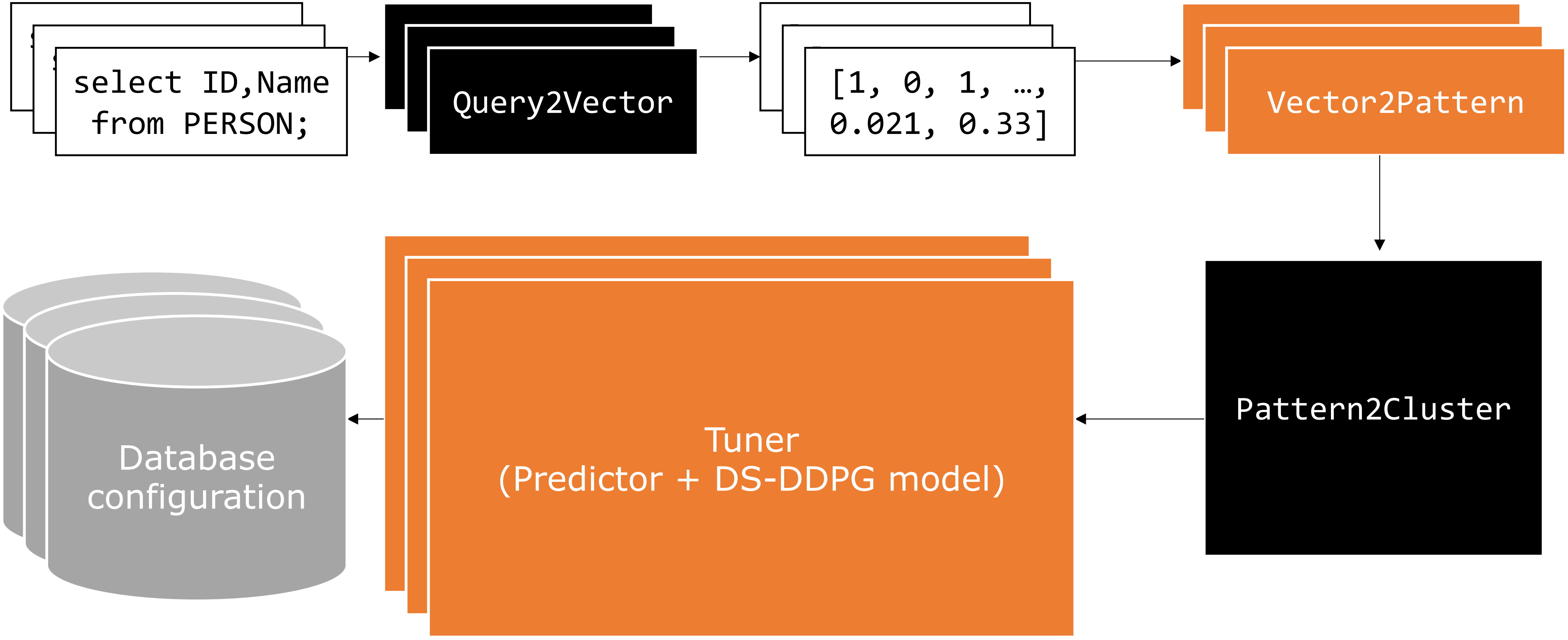
The Architecture (multiple queries)



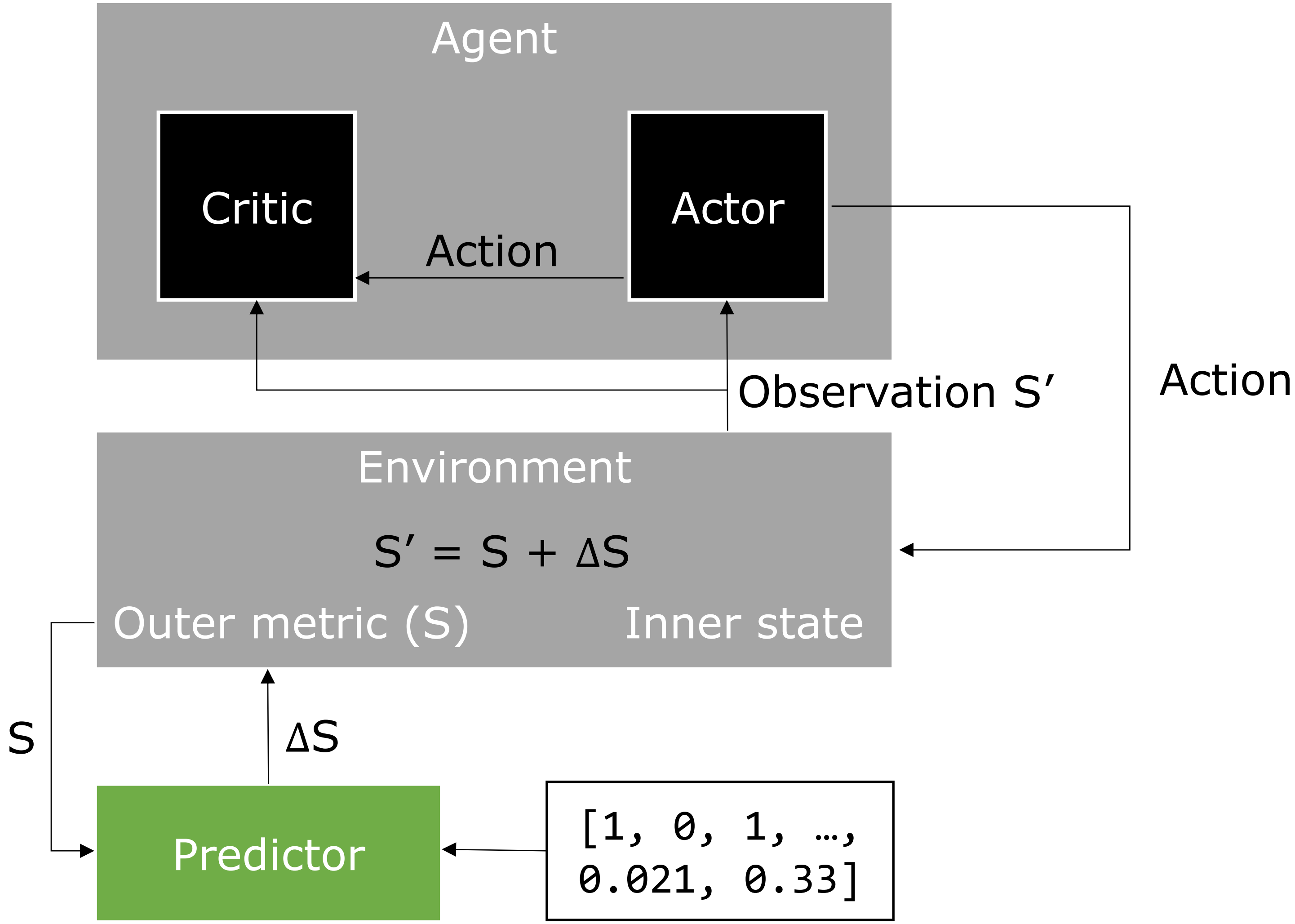
The Architecture (clusters)



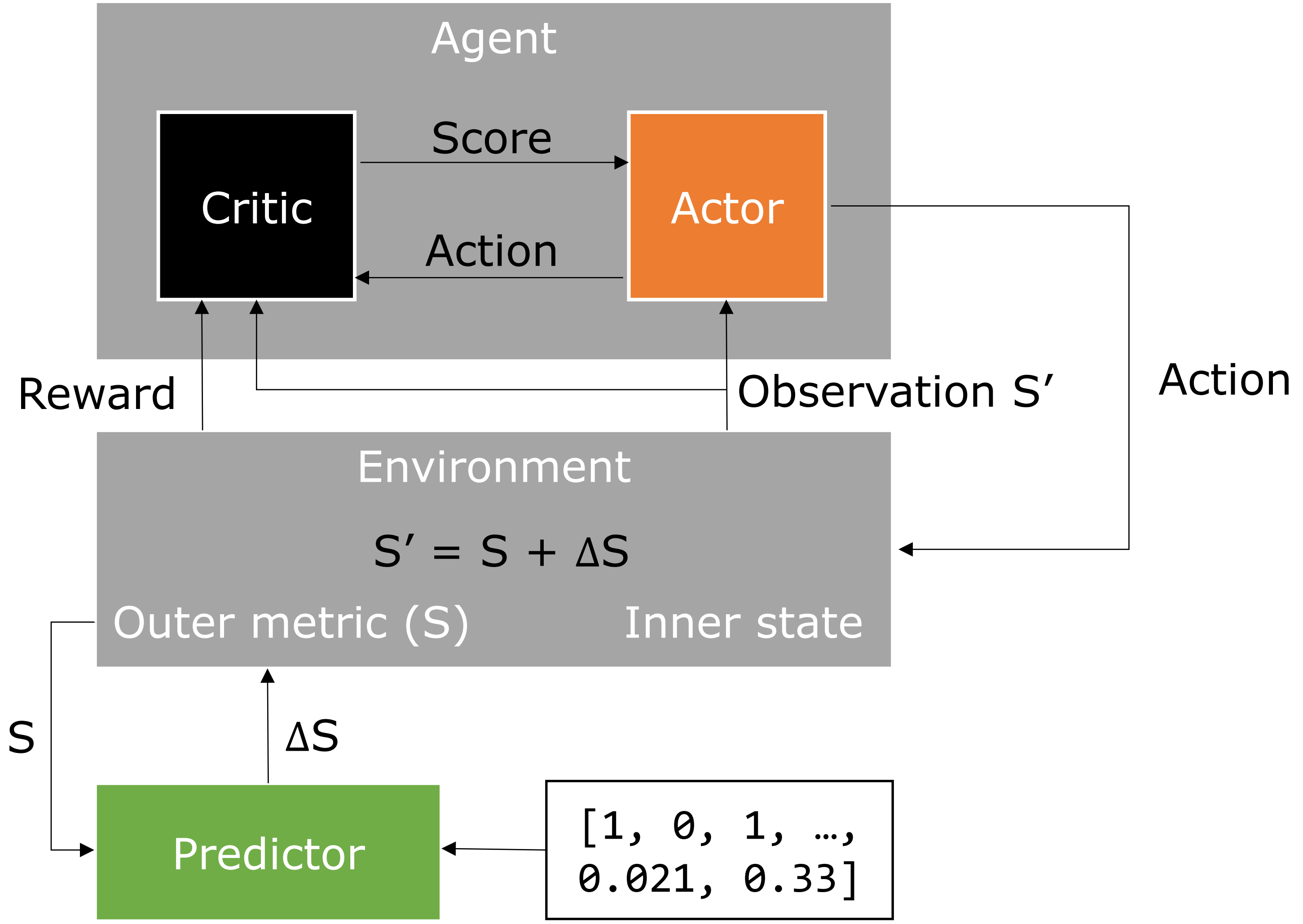
The Architecture (clusters)



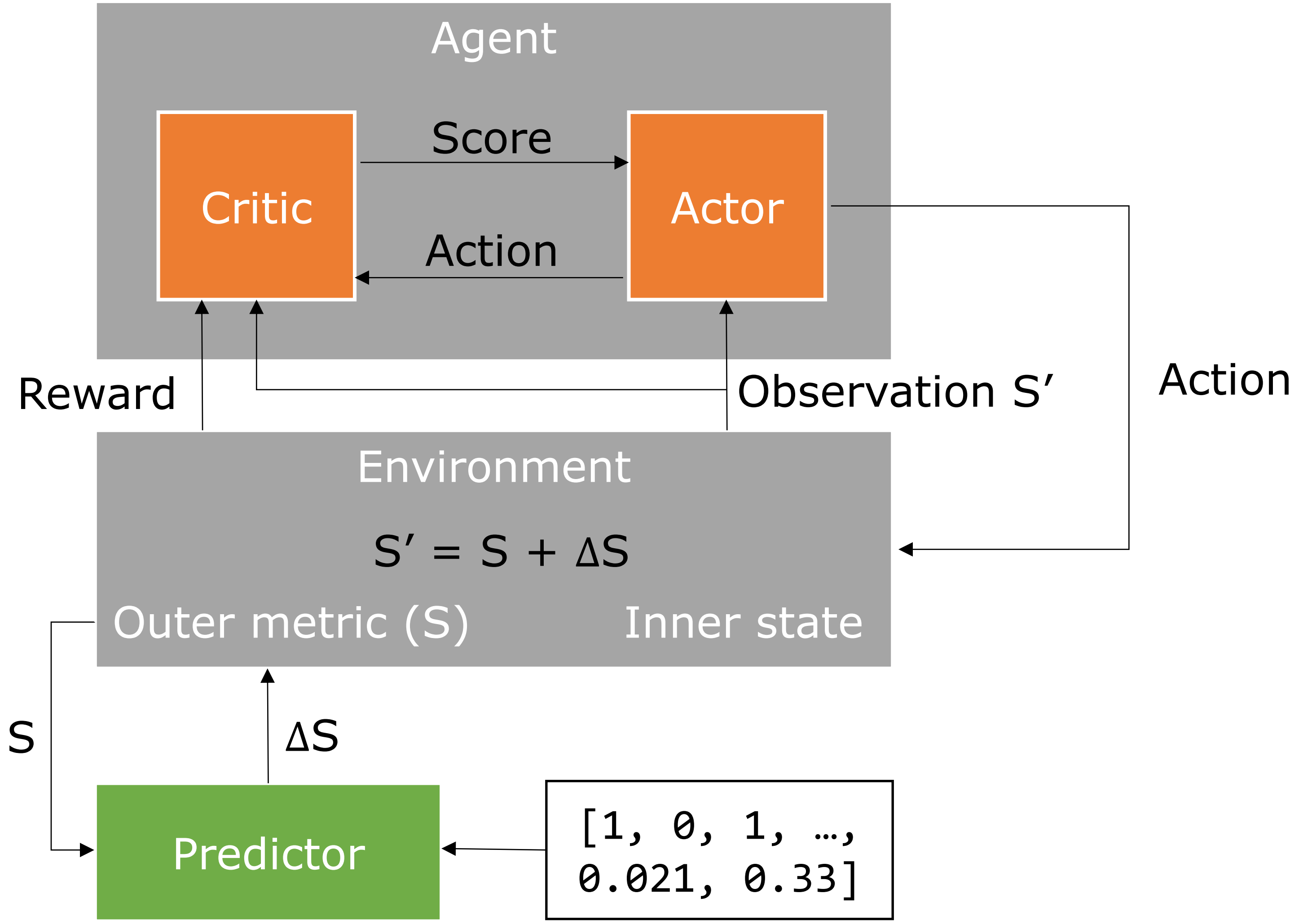
Training the Tuner (DS-DDPG)



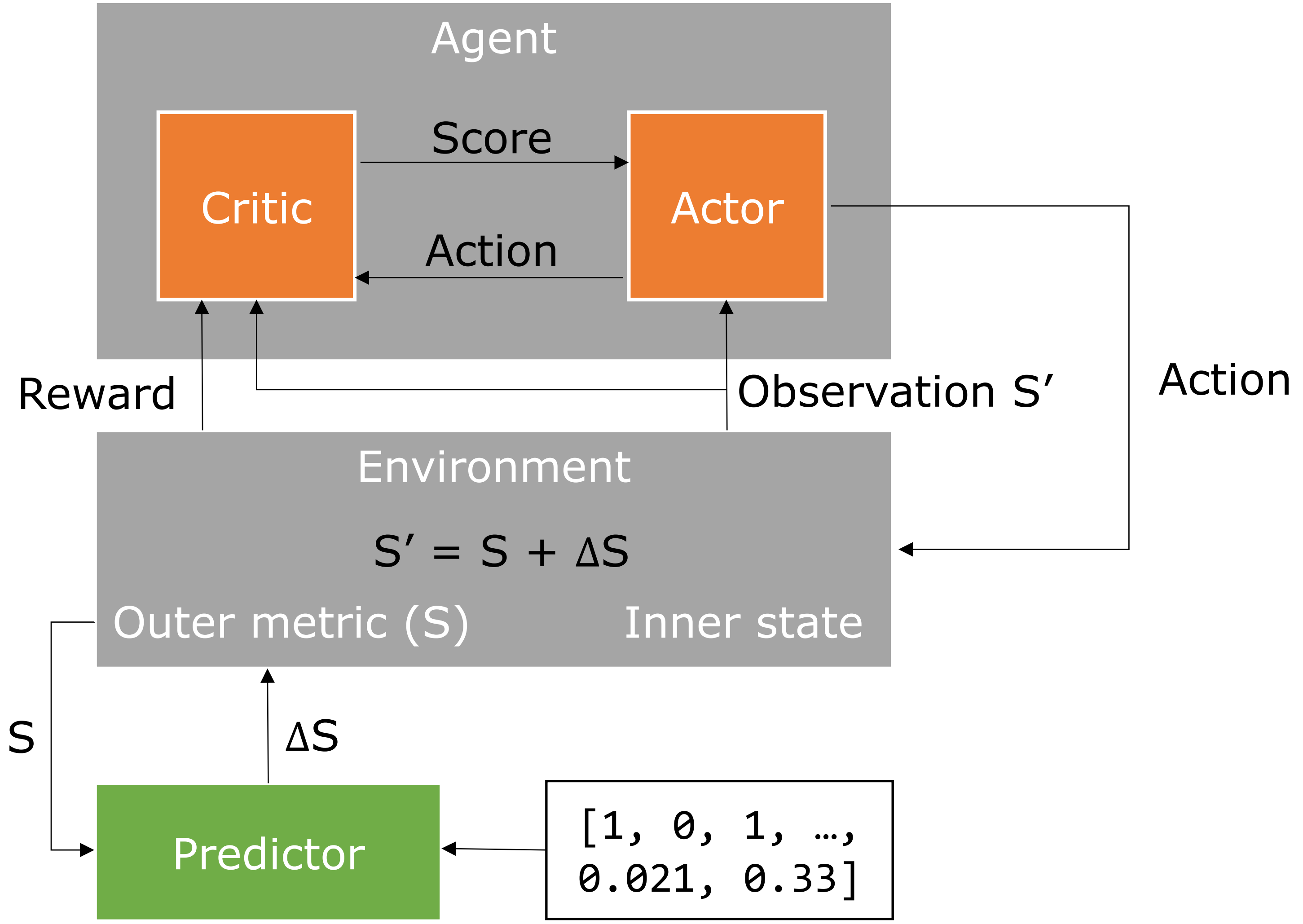
Training the Tuner (DS-DDPG)



Training the Tuner (DS-DDPG)

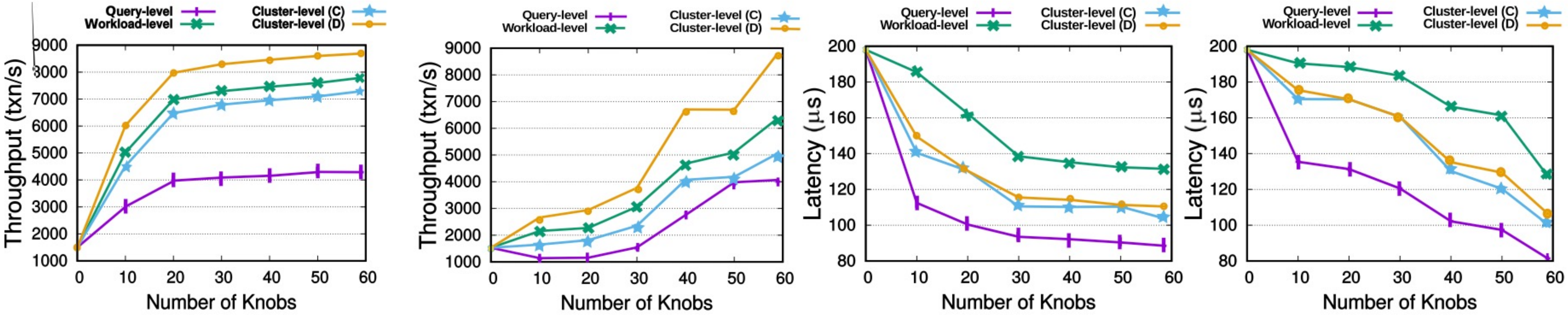


The Tuner


Agent
Critic
Score
Action
Actor
Reward
Observation S′
Action
Environment
S′ = S + ΔS
Outer metric (S)
Inner state
S
ΔS
Predictor
[1, 0, 1, …,
0.021, 0.33]

Evaluation

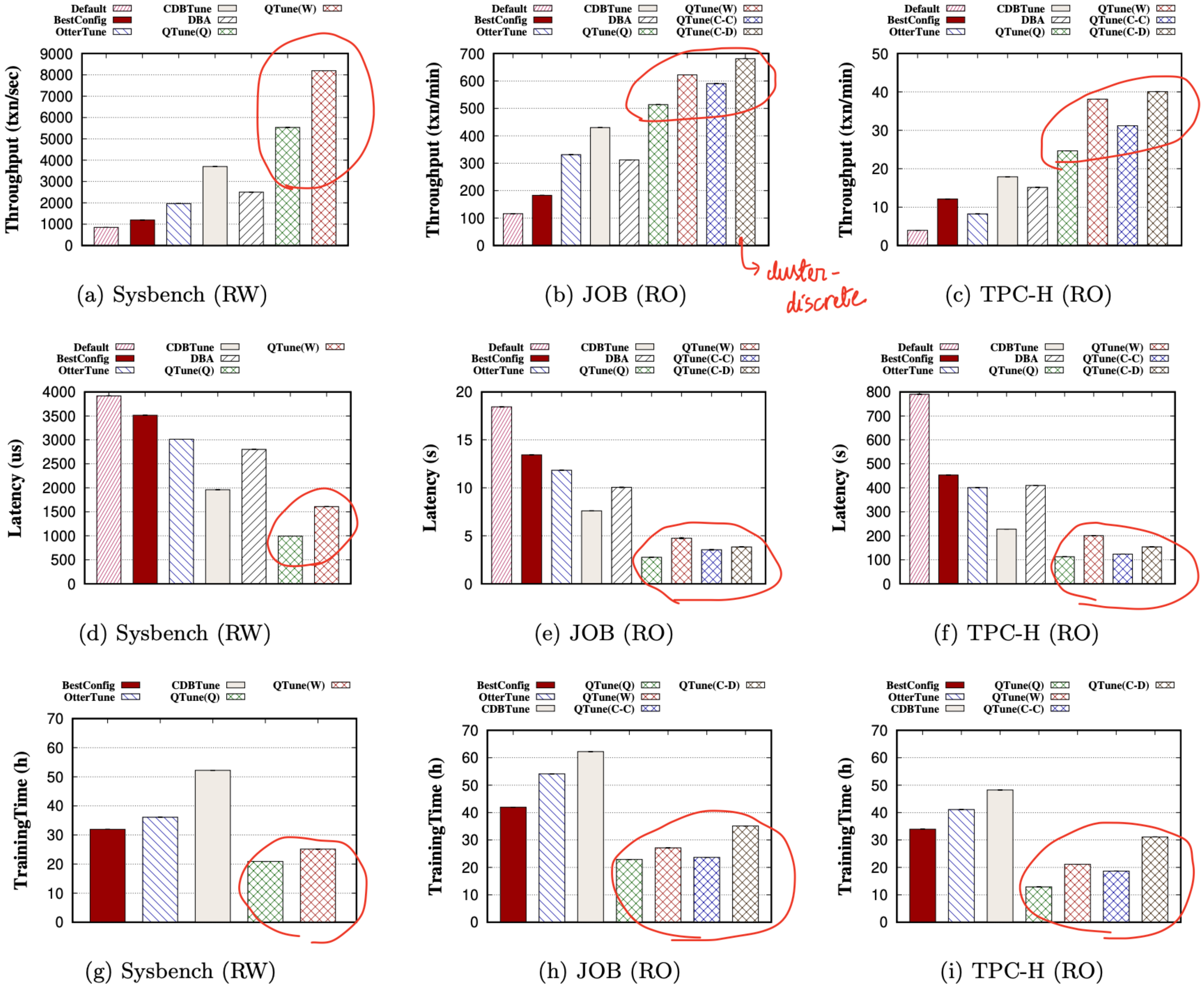
- Bulk of evaluation done on PostgreSQL with 3 datasets;
- Discrete Cluster-level tuning achieves the best throughput;
- Query-level tuning achieves the best latency.



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

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.

Appendix 2: Experiment settings

Table 2: Database information

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

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

Name	Mode	Table	Cardinality	Size(G)	Query
JOB	RO	21	74,190,187	13.1	113
TPC-H	RO	8	158,157,939	50.0	22
Sysbench	RO, RW	3	4,000,000	11.5	474,000

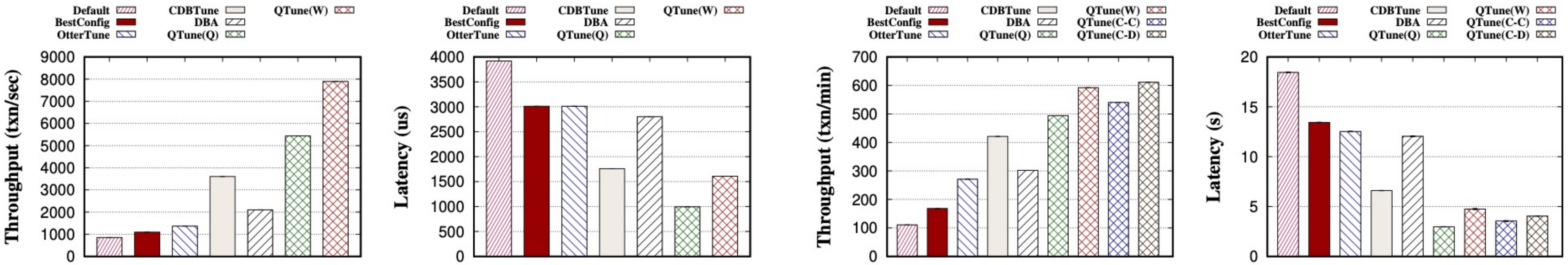
Table 4: The number of training samples for the DL model in query clustering, the Predictor and the Actor-Critic module in DS-DDPG.

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

Table 6: Two hardware configurations

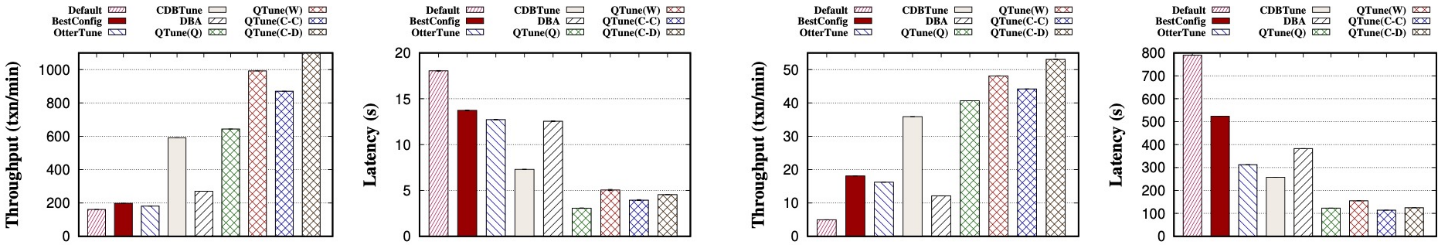
Instance	RAM (GB)	Disk (GB)	CPU (GHz)
A	16	780	2.49
B	128	5000	4.00

Appendix 3: Generalisation



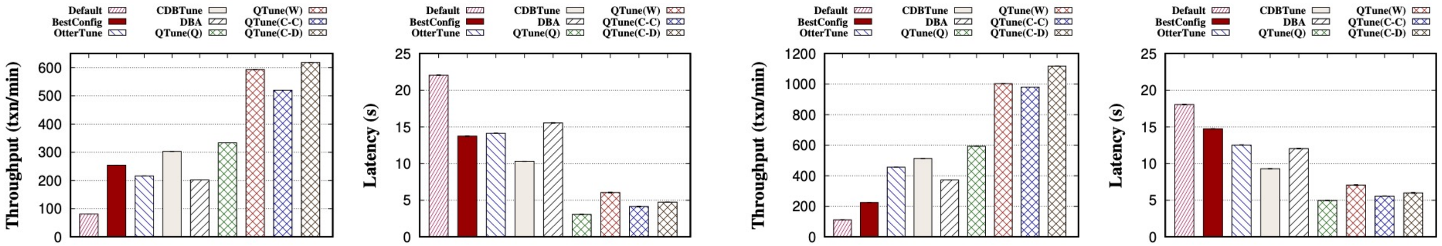
(a) JOB(RO) to Sysb.(RW) (b) JOB(RO) to Sysb.(RW) (c) TPC-H(RO) to JOB(RO) (d) TPC-H(RO) to JOB(RO)

Figure 9: Performance when workload changes on PostgreSQL.



(a) MySQL (b) MySQL (c) MongoDB (d) MongoDB

Figure 10: Performance for different databases.



(a) M_A on A (b) M_A on A (c) M_B on A (d) M_B on A

Figure 11: Performance for different hardware environments.

Appendix 4: Training details

Function TrainPredictor(π_P, T_P)

Input: π_P: The weights of a neural network; T_P: The training set

Initiate the weights in π_P;
while *!converged* **do**
  for *each* $(v, S, I, \Delta S) \in T_P$ **do**
    Generate the output G of $\langle v, S, I \rangle$;
    Accumulate the backward propagation error: $E = E + \frac{1}{2}||G - \Delta S||^2$;
  Compute gradient ∇_{θ_s}(E), update weights in π_P;

Function TrainAgent(π_A, π_C, T_A)

Input: π_A: The actor's policy; π_C: The critic's policy; T_A: training data

Initialize the actor π_A and the critic π_C;
while *!converged* **do**
  Get a training data $T_A^1 = (S_1', A_1, R_1), (S_2', A_2, R_2), \ldots, (S_t', A_t, R_t)$;
  for $i = t - 1$ *to* 1 **do**
    Update the weights in π_A with the action-value $Q(S_i', A_i|\pi_C)$;
    Estimate an action-value $Y_i = R_i + \tau Q(S_{i+1}', \pi_A(S_{i+1}'|\theta^{\pi_A})|\pi_C)$;
    Update the weights in π_C by minimizing the loss value $L = (Q(S_i', A_t|\pi_C) - Y_i)^2$;

Algorithm 1: Training DS-DDPG

Input: U: the query set $\{q_1, q_2, \cdots, q_{|U|}\}$
Output: π_P, π_A, π_C

Generate training data T_P;
TrainPredictor(π_P, T_P);
Generate training data T_A;
TrainAgent(π_A, π_C, T_A);