Qtune : A query-aware database tuning system with deep reinforcement learning

연세대학교 컴퓨터과학과 염찬호

2022년 3월

SW STAR LAB

Software Technology Advanced Research

과제명: IoT 환경을 위한 고성능 플래시 메모리 스토리지 기반 인메모리 분산 DBMS 연구개발

과제번호: 2017-0-00477

과학기술정보통신부 Ministry of Science and ICT

연세대학교 YONSEI UNIVERSITY

IITP 정보통신기술진흥센터 Institute for Information & communications Technology Promotion

Index

- INTRODUCTION
- SYSTEM OVERVIEW
- QUERY FEATURIZATION
- DRL FOR KNOB TUNING
- QUERY CLUSTERING
- EXPERIMENT

Index

- INTRODUCTION
- SYSTEM OVERVIEW
- QUERY FEATURIZATION
- DRL FOR KNOB TUNING
- QUERY CLUSTERING
- EXPERIMENT

Traditional tuning (DBA)

- Limitation
  - DBAs **can only tune a small percentage of the knobs** and **may not find a good global knob configuration**
  - DBAs require to **spend a lot of time**
  - DBAs are usually good at **tuning a only specific database**

  - → These limitations are extremely severe for tuning cloud databases, because they have to tune a lot of database instances on different environments (e.g., different CPU, RAM and disk).

Automatic knob tuning

- **BestConfig**
- **OtterTune**
- **CDBTune**

Automatic knob tuning

- **BestConfig**
    - **heuristic method** to search for the optimal configuration from the history and **may not find** good knob values if there is no similar configuration in the history

Automatic knob tuning

- **OtterTune**
  - **machine-learning** techniques to collect, process and analyze knobs and tunes the database by learning DBAs' experiences from the historical data

  - relies on a large number of high-quality training examples from DBAs' experience data, which are rather hard to obtain

Automatic knob tuning

- **CDBTune**
  - deep reinforcement learning (DRL) to tune the database by using a try-and error strategy
  - has 3 limitations

CDBTune - limitation

- **First**
  - CDBTune **requires** to run a SQL query workload **multiple times** in the database to get an appropriate configuration, which is rather time consuming
- **Second**
  - CDBTune only provides a **coarse-grained tuning** (i.e., tuning for read-only workload, read-write workload, write-only workload), but cannot provide a fine-grained tuning (i.e., tuning for a specific query workload).
- **Third**
  - it directly uses the existing DRL model, which assumes that the environment can only be affected by reconfiguring actions, but **cannot utilize** the **query information**, which is more important for configuration tuning and environment updates.

Proposed model (**Qtune**)

- Step
  - first **featurizes the SQL queries** by considering rich features of the SQL queries(query type, tables, and query cost)
  - Then feeds the **query features into the DRL** model to dynamically choose suitable configurations

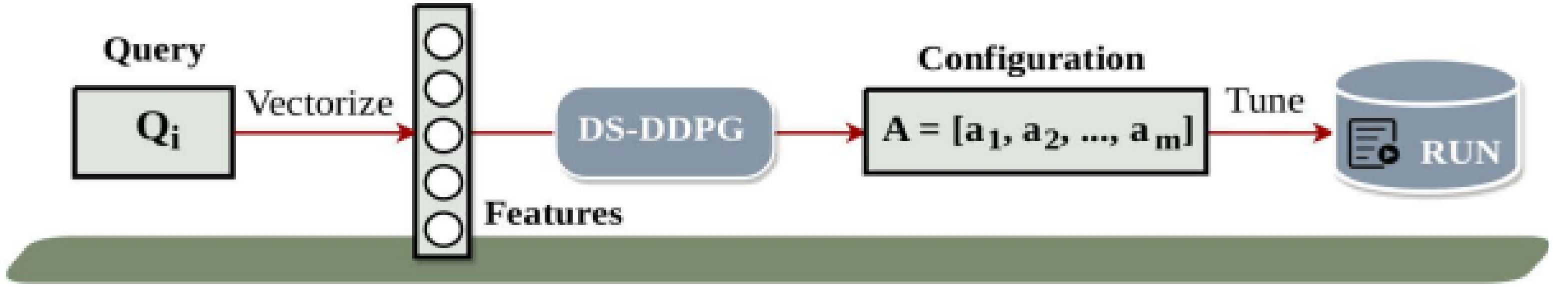
Index

- INTRODUCTION
- SYSTEM OVERVIEW
- QUERY FEATURIZATION
- DRL FOR KNOB TUNING
- QUERY CLUSTERING
- EXPERIMENT

Three types of tuning requests

- **Query-level Tuning**
- **Workload-level Tuning**
- **Cluster-level Tuning**

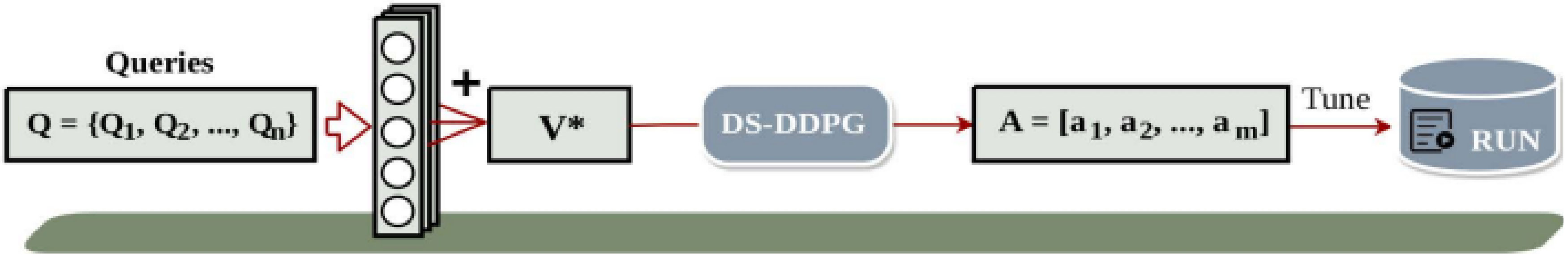
Three types of tuning requests



- **Query-level Tuning**
  - For each query, it first tunes the database knobs and then executes the query
  - **can optimize the latency**(=low latency)
  - but **may not achieve high throughput.**

  - Because query-level tuning cannot run ths SQL queries in parallel

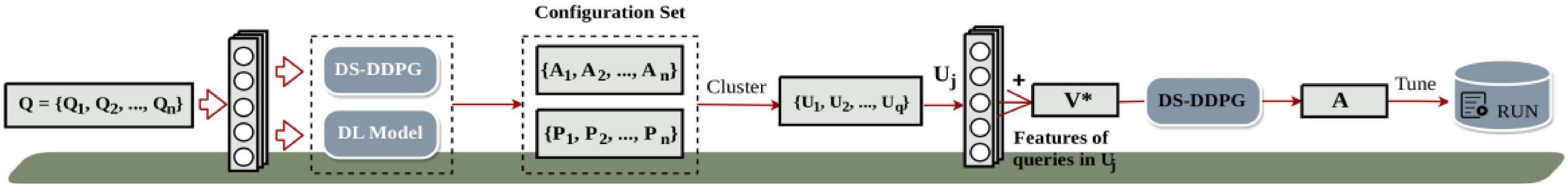
Three types of tuning requests



- **Workload-level Tuning**
  - It tunes the database knobs for the whole query workload
  - **cannot optimize the query latency**
  - **can achieve high throughput**

  - Because it cannot find a good configuration **for every SQL query**

Three types of tuning requests



- **Cluster-level Tuning**
  - It **partitions** the queries into different **groups**
  - Next it **tunes the knobs for each query group** and executes the queries in each group in parallel. This method can optimize both the latency and throughput.
  - Because it **can find the good configuration** for a group of queries and **run** the queries in each group in **parallel**

Index

- INTRODUCTION
- SYSTEM OVERVIEW
- QUERY FEATURIZATION
- DRL FOR KNOB TUNING
- QUERY CLUSTERING
- EXPERIMENT

QUERY FEATURIZATION

- 3.1 Query Information
- 3.2 Cost Information
- 3.3 Character Encoding

```
SELECT   MIN(tbl3.movie_id)
FROM     tbl1, tbl2, tbl3
WHERE    tbl1.info = '%act%'
    AND  tbl1.id = tbl3.type_id
    AND  tbl2.movie_id = tbl3.movie_id
```

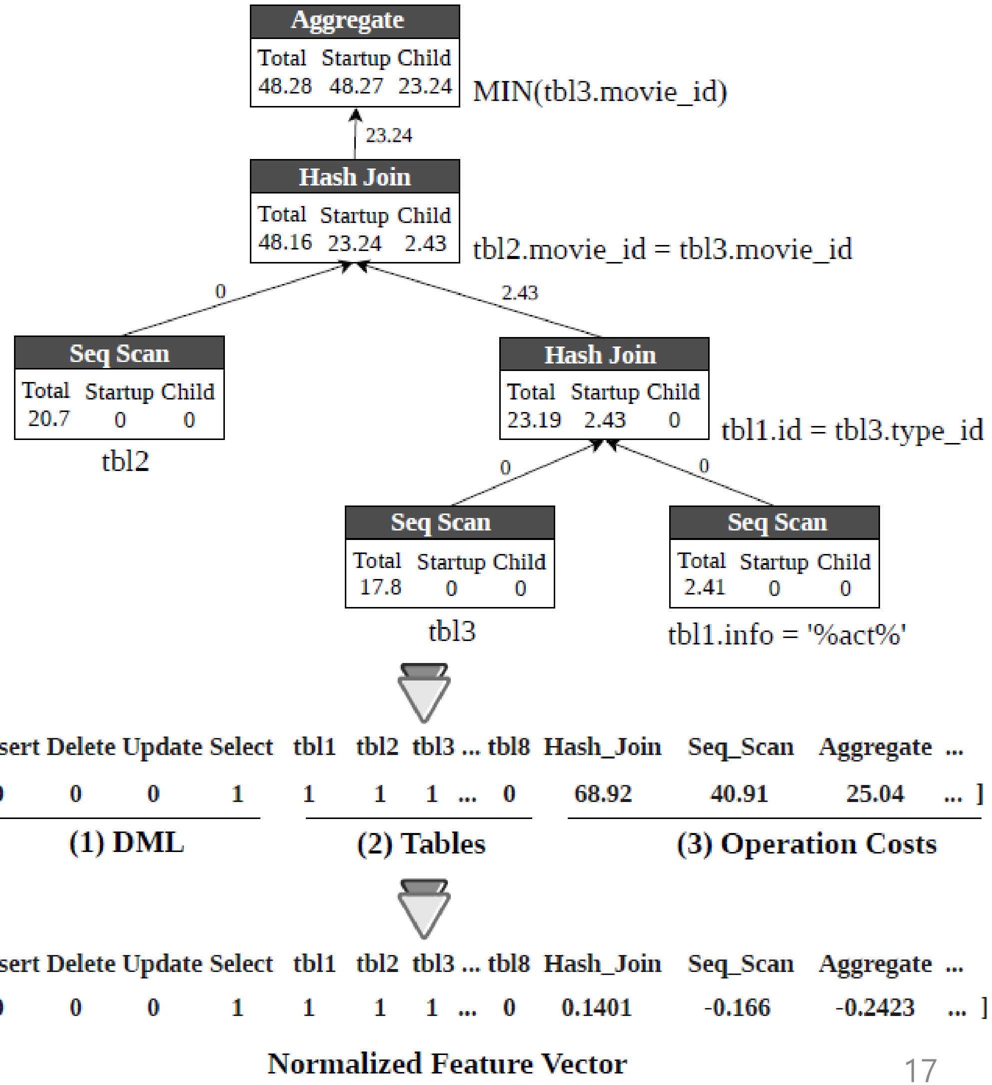


Figure 3: Character Encoding.

3.1 Query Information

- **SQL query**
  - **Query type(e.g., insert, delete, select, update), table, attributes, operations(e.g., selection, join, groupby)**

- **Query type - different query types have different query cost**
- **Tables - data volumes and structures of tables will signficantly affect the database performance**

3.1 Query Information

- **Note that we do not featurize the attributes (i.e., columns) and operations (i.e., selection conditions) due to three reasons.**
- **First, the query cost will capture the operation information and cost, and we do not need to maintain duplicated information.**
- **Second, operations are too specific and adding specific operations into the vectors will <u>reduce the generalization ability</u>.**
- **Third, the attributes and operations will be frequently updated and it requires to redesign the model for the updates.**

- **Query information → 4 + |T| dimensional vector**
  - **4 : query types, (e.g., insert, select, updat**
  - **|T| : table**

Insert	Delete	Update	Select	tbl1	tbl2	tbl3	...	tbl8
[0	0	0	1	1	1	1	...	0
(1) DML				(2) Tables				

3.2 Cost Information

- **utilize the query plan generated by the query optimizer, which has a cost estimation for each operation.**

- Fig3 is **the vector of a SQL query.**

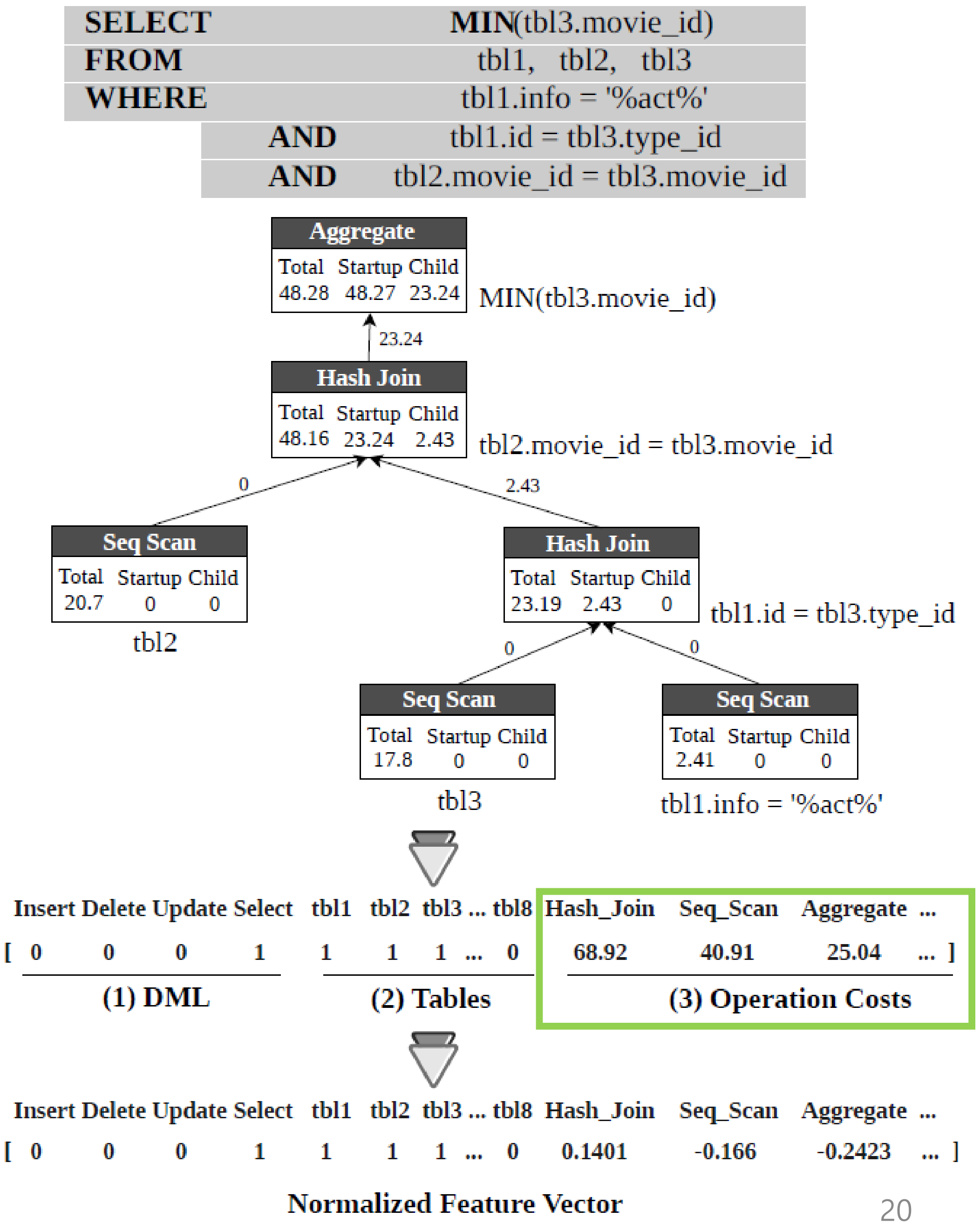


Figure 3: Character Encoding.

3.3 Character Encoding

- **To tune the database for this query workload, we need to combine the vectors together**
- **concatenate the query vector and cost vector to generate an overall vector of a query**
- **for each query vector, we need to consider all the query types and tables, and thus we compute the union of the query vectors.**
- **And for each table, if the value is 1, we replace it with the row number of the table. Thus it can capture the actions like deleting/inserting rows and improve system's adaptivity**
- **for cost vector, we need to sum up all the costs**

Insert	Delete	Update	Select	tbl1	tbl2	tbl3	...	tbl8	Hash_Join	Seq_Scan	Aggregate	...
[0	0	0	1	1	1	1	...	0	0.1401	-0.166	-0.2423	...]

Index

- INTRODUCTION
- SYSTEM OVERVIEW
- QUERY FEATURIZATION
- DRL FOR KNOB TUNING
- QUERY CLUSTERING
- EXPERIMENT

DRL FOR KNOB TUNING

- **Since there are hundreds of knobs in a database and many of them are in continuous space, the database tuning problem is NP hard and it is rather expensive to find highquality configurations.**
- **We utilize the deep reinforcement learning model, which combines reinforcement learning and neural networks to automatically learn the knob values from limited samples.**
- **existing DRL models cannot utilize the query features as they ignore the effects to the environment state from the query, and we propose a Double-State Deep Deterministic Policy Gradient (DS-DDPG) model to enable query-aware tuning**

4.1 DS-DDPG

Table 1: Mapping from DS-DDPG to Tuning

DS-DDPG	The tuning problem
Environment	Database being tuned
Inner state	Database knobs (e.g., work_mem)
Outer metrics	State statistics (e.g., updated tuples)
Action	Tuning database knobs
Reward	Database performance changes
Agent	The Actor-Critic networks
Predictor	A neural network for predicting metrics
Actor	A neural network for making actions
Critic	A neural network for evaluating Actor

4.1 DS-DDPG

- **Environment**
  - **contains the database information, which includes the inner state (i.e., knob configurations) and the outer metrics (e.g., database key performance indicators).**
- **Query2Vector**
  - **generates the feature vector for a given query (or a workload).**

4.1 DS-DDPG

- **Environment**
  - **contains the <u>database information</u>, which includes the inner state (i.e., knob configurations) and the outer metrics (e.g., database key performance indicators).**

- **Query2Vector**
  - **generates the feature vector for a given query (or a workload).**

- **Predictor**
  - **is a deep neural network, which predicts <u>the changes in outer metrics (ΔS)</u> of before/after processing the queries.**
  - **observation S' = S + ΔS (S : original metrics)**

- **Agent**
  - **is used to <u>tune the inner state based on the observation S'</u>. Agent contains two modules, Actor and Critic, which are two independent neural networks.**

4.1 DS-DDPG

Actor

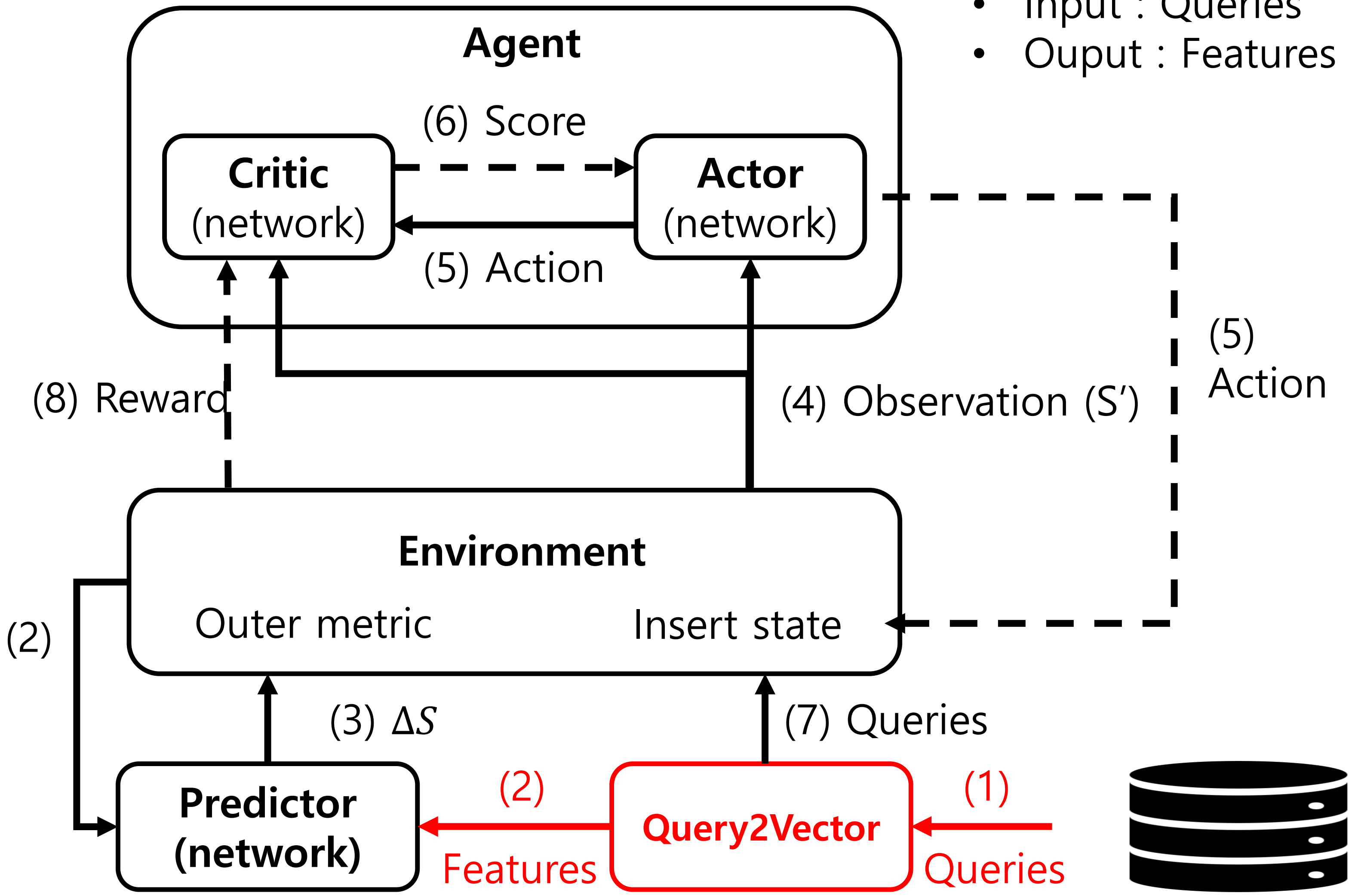
- takes S′ as input, and outputs an action (a vector of tuned knob configurations). **Environment** executes the query workload and computes a reward based on the performance.

- **updates the weights** of its neural network **based on the Q-value**

Critic

- takes the observation S′ and the action as input, and outputs a score (Q-value), which reflects whether the action tuning is effective.

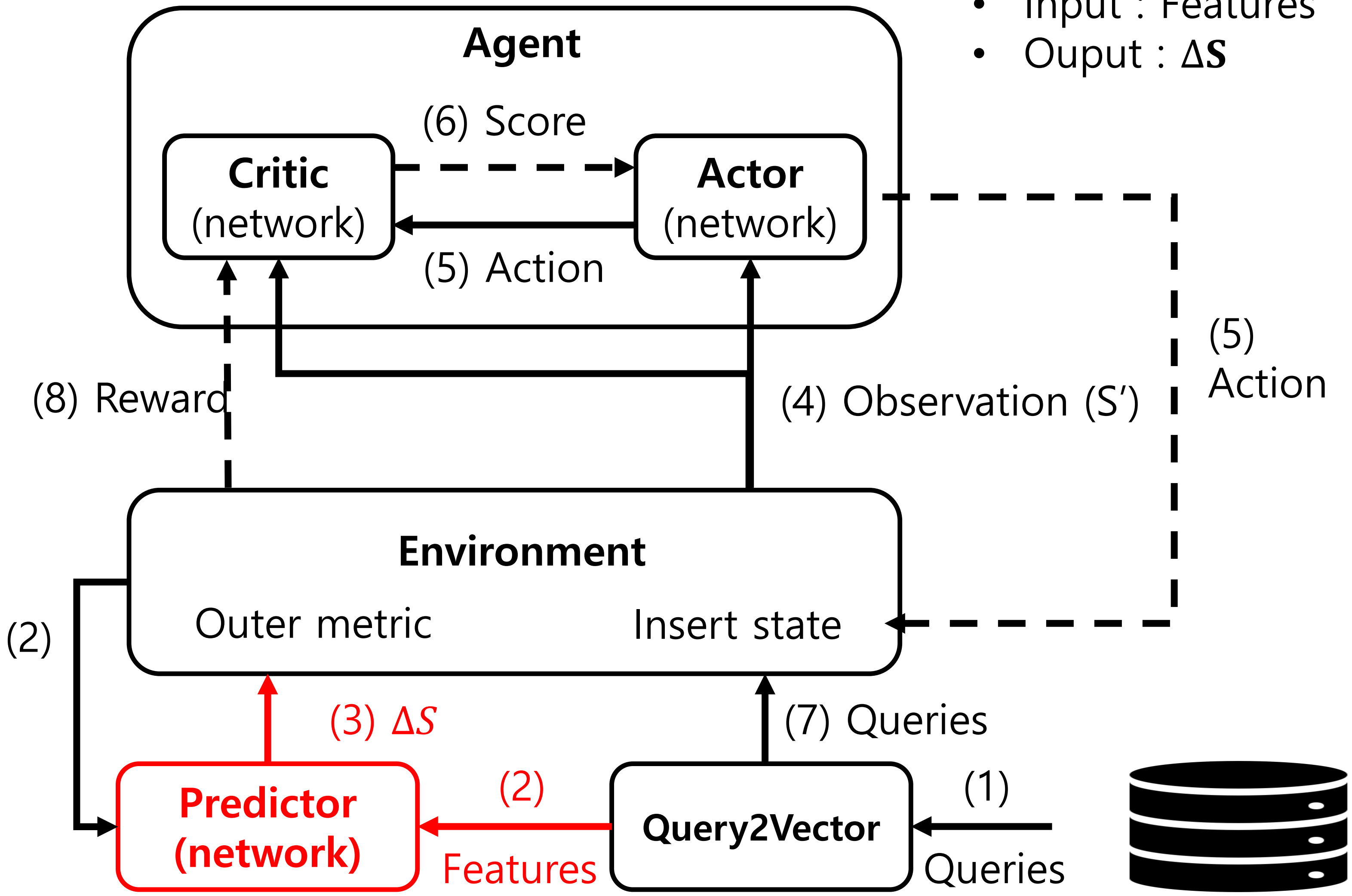
- **updates the weights** of its neural network **based on the reward value**.



- Input : Queries
- Ouput : Features

Step 1

Query2Vector가 주어진 쿼리로부터 Feature vector를 생성



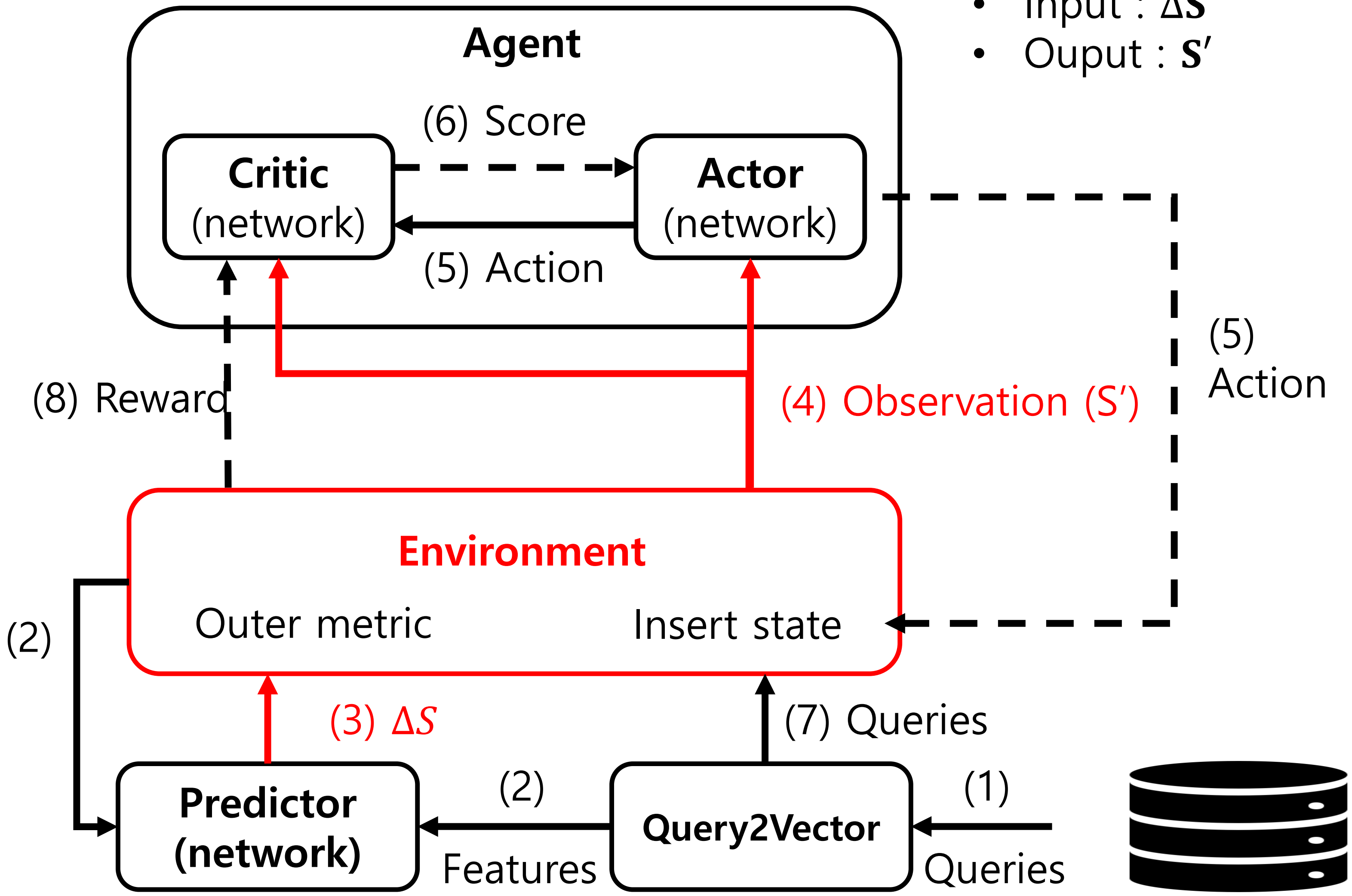
- Input : Features
- Ouput : ΔS

Step 2

Query 실행 후의 Outer metrics 변화량 예측

- Outer metrics of **before/after processing the query**
- **Difference**
- ΔS

ΔS : query 작업을 진행하기 전과 후의 Outer metrics 차이값



- Input : $\Delta\boldsymbol{S}$
- Ouput : $\boldsymbol{S}'$

Step 3

Query 실행 후의 Outer metrics 값 계산

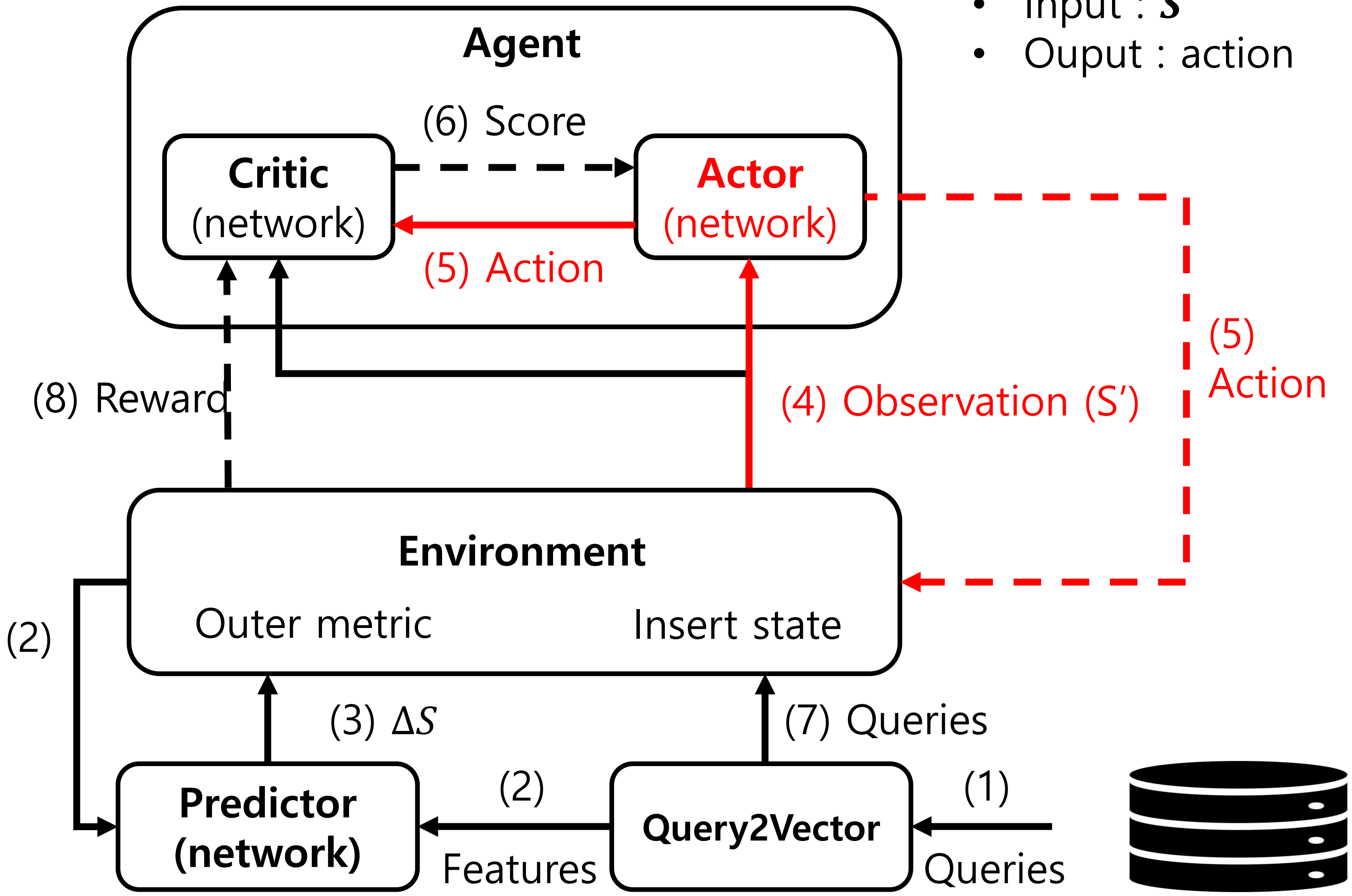
$$\boldsymbol{S}' = \boldsymbol{S} + \Delta\boldsymbol{S}$$

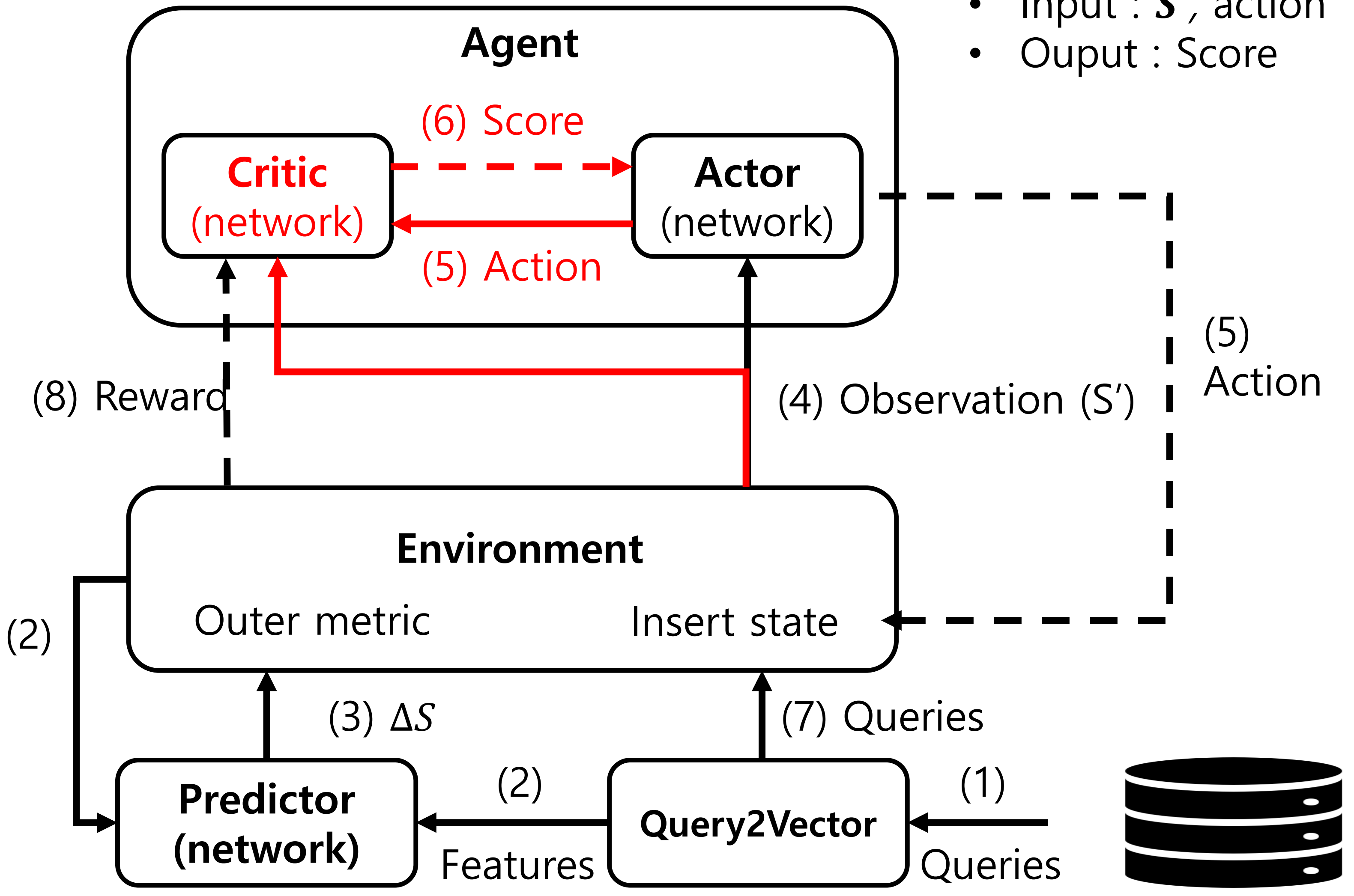
- $\boldsymbol{S}$: 기존의 Outer metrics 값, query 작업 진행하기 전의 Outer metrics 값
- $\Delta\boldsymbol{S}$: query 작업을 진행하기 전과 후의 Outer metrics 차이값
- $\boldsymbol{S}'$: 기존 Outer metrics 값에 query 작업을 진행한 결과가 반영된 값

- Input : S'
- Ouput : action

Step 4

New Knob values(Action) 획득



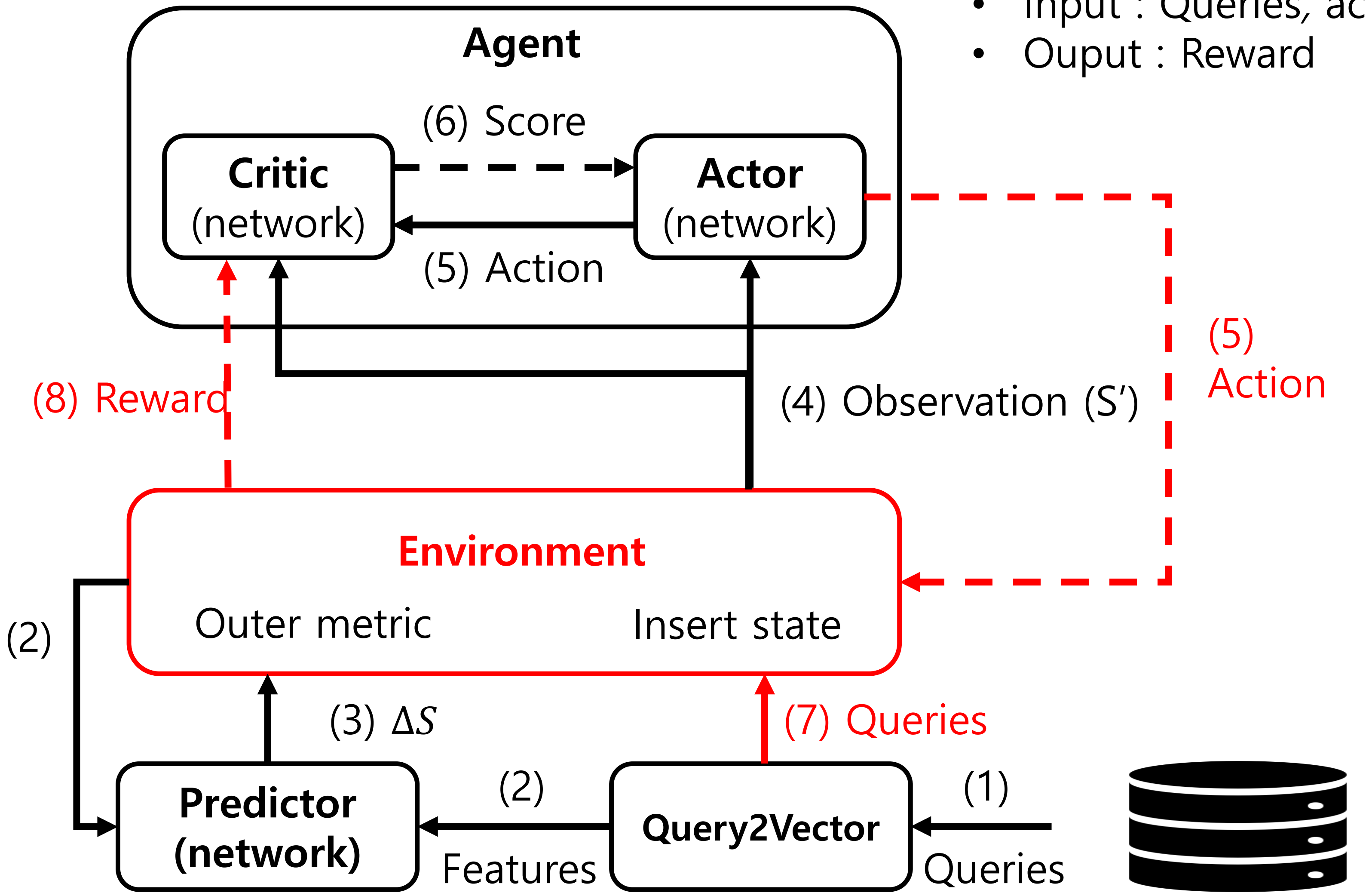


- Input : S', action
- Ouput : Score

Step 5

Query를 실행하고 난 후의
outer metrics 예측

➡ Actor의 Weight update



- Input : Queries, action
- Ouput : Reward

Step 6
New knob values에 대한 Reward 계산

- Action(New Knob values)로 update
- Query 실행 후 Performance 비교
- Reward 계산

4.2 Training DS-DDPG

- **4.2.1 Training the Predictor**
- **4.2.2 Training the Actor-Critic Module**

Algorithm 1: Training DS-DDPG

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

1 Generate training data T_P;
2 TrainPredictor(π_P, T_P);
3 Generate training data T_A;
4 TrainAgent(π_A, π_C, T_A);

4.2.1 Training the Predictor

- **Predictor aims to predict the database metrics change if processing a query in the database.**
  $T_P = \{< v, S, I, \Delta S >\}$
- **For each < v, S, I >, we train Predictor to output a value that is close to ΔS**

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;

- v : a vector of a query
- S: the outer metrics
- I : inner state
- ΔS : the outer metrics change
- **G : the output value by** Predictor **for query qi,**
- **U : the query set.**
- **E : error function**

4.2.2 Training the Actor-Critic Module

- **The agent (the Actor-Critic module) aims to judiciously tune the database configurations**

1) **We generate its feature vector in via Query2Vector**
2) **Predict a database metrics S'_1 via Predictor**
3) **Get an action A_1 via Actor**
4) **Deploy the actions in the database**
5) **Run the database to get reward R_1**

- **In the next step, we get a new database metrics S'_2 by updating S'_1 using the new metrics, and repeat The above steps to get A_2 and R_2 Until the average reward value is good enough(the average reward of ten runs is larger than 10)**
  $T_P^1 =< (S_1', A_1, R_1), (S_2', A_2, R_2), \dots, (S_t', A_t, R_t) >$

4.2.2 Training the Actor-Critic Module

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 $L = (Q(S'_i, A_t | \pi_C) - Y_i)^2$;

Index

- INTRODUCTION
- SYSTEM OVERVIEW
- QUERY FEATURIZATION
- DRL FOR KNOB TUNING
- QUERY CLUSTERING
- EXPERIMENT

5.1 Configuration Pattern

- **DS-DDPG to generate a continuous knob configuration and take the knob configuration as the pattern**

→ But It is expensive to get the continuous knob values,
approximate patterns are good enough to cluster the queries

- **So we discretize the continuous values into discrete values (i.e. {-1, 0, +1})**

→Using Deep Learning
(input : feature vector, output : pattern)

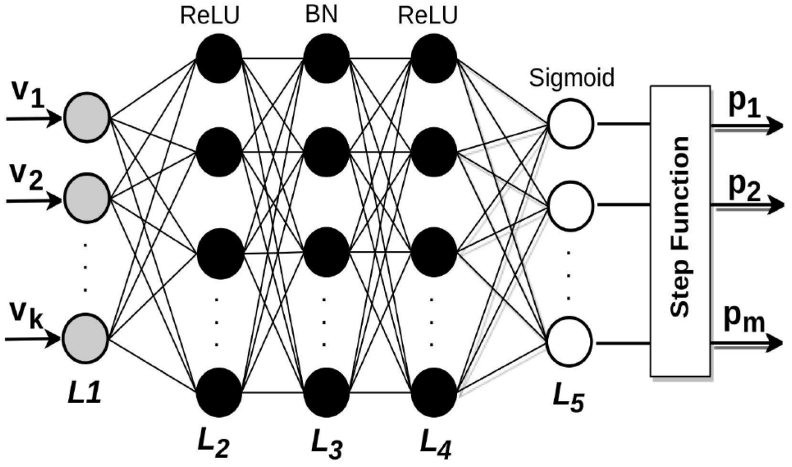


Figure 5: Architecture of the DL model

5.2 Query Clustering

- **After gaining the suitable configuration pattern for each query, we classify the queries into different clusters based on the similarity of these patterns**
- **we take** DBSCAN(Density-based spatial clustering of applications with noise) **as a clustering algorithms**

Index

- INTRODUCTION
- SYSTEM OVERVIEW
- QUERY FEATURIZATION
- DRL FOR KNOB TUNING
- QUERY CLUSTERING
- EXPERIMENT

Table 2: Database information

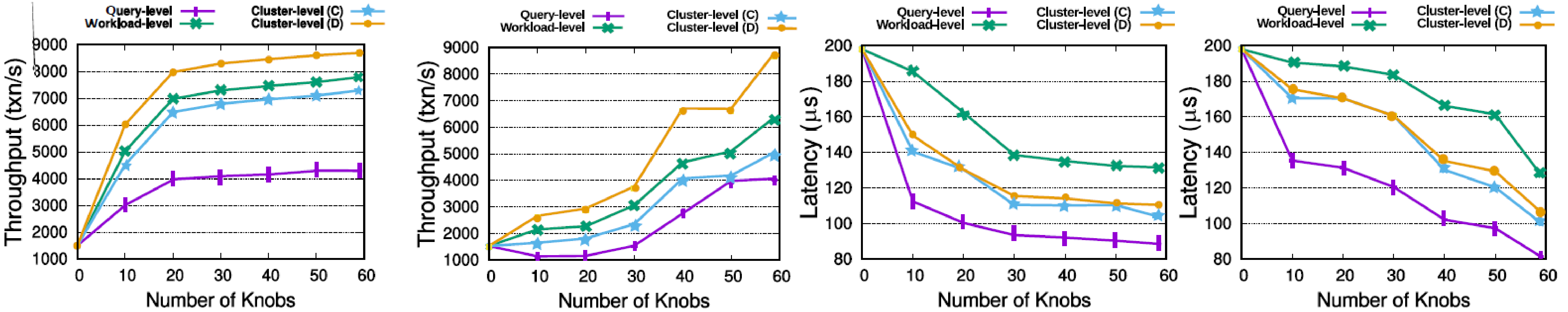
Database	Knobs without restart	State Metrics
PostgreSQL	64	19
MySQL	260	63
MongoDB	70	515

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.

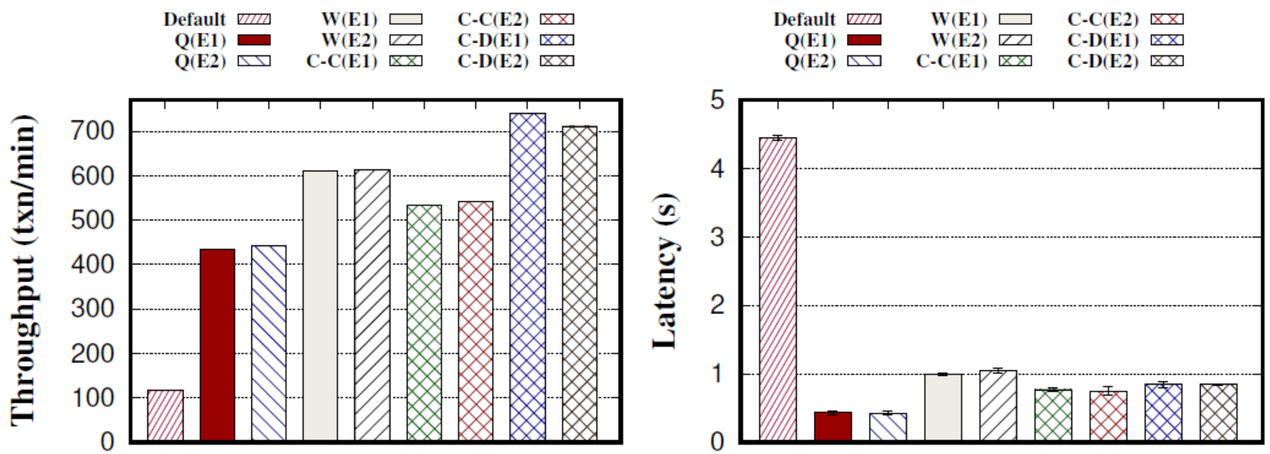
Name	Sysbench	JOB	TCP-H
DL	3792	8000	40,000
Predictor	3792	8000	40,000
Actor-Critic	1500	480	300



(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.

(1) Randomly Choosing : We permute the knobs in a random way. If we tune k knobs, we select the first k knobs.
(2) Important first : We sort the knobs based on their importance (e.g., which knobs were tuned more in the query workload).

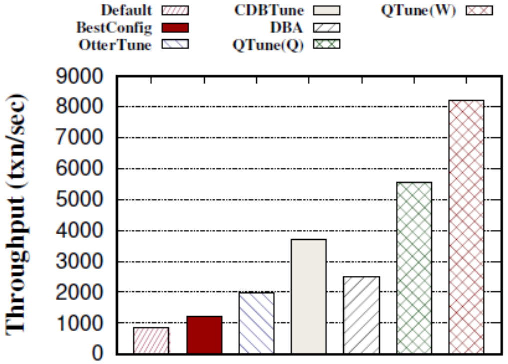


(a) Throughput (b) Latency

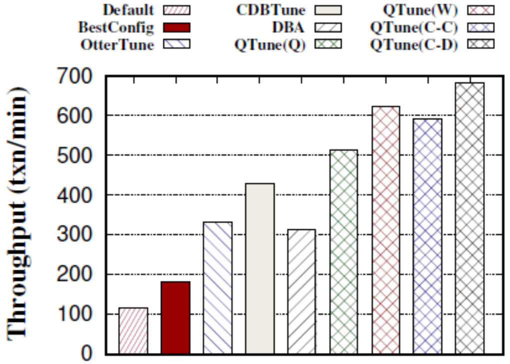
Figure 7: Performance comparison of 2 Featurization methods (E1, E2) when running JOB (RO) on PostgreSQL. (Query-level(Q), Workload-level(W), Cluster-level-C (C-C)), Cluster-level-D(C-D)

E1 : uses query type, tables, costs

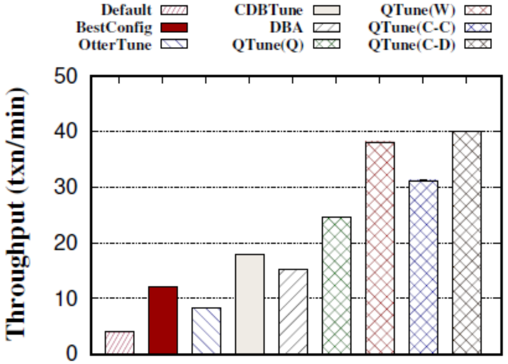
E2 : uses query type, tables, costs, attributes, operations



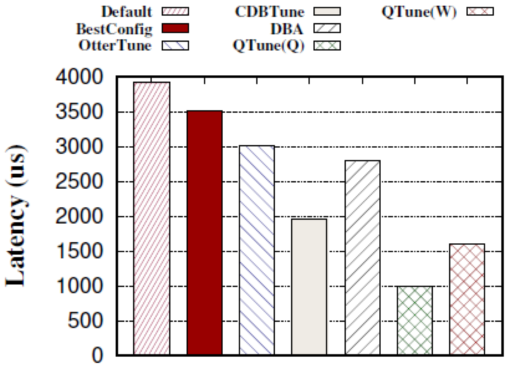
(a) Sysbench (RW)



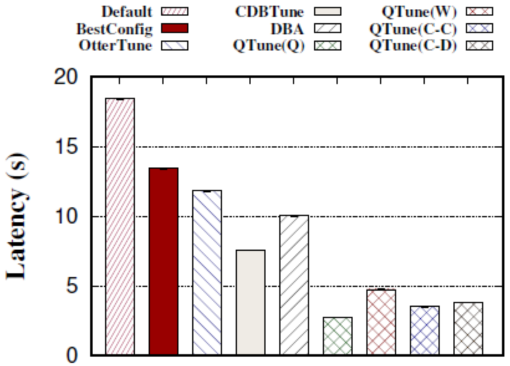
(b) JOB (RO)



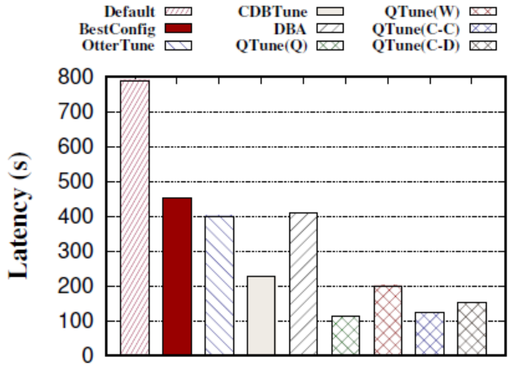
(c) TPC-H (RO)



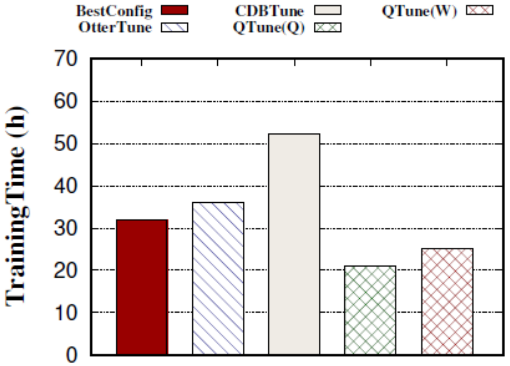
(d) Sysbench (RW)



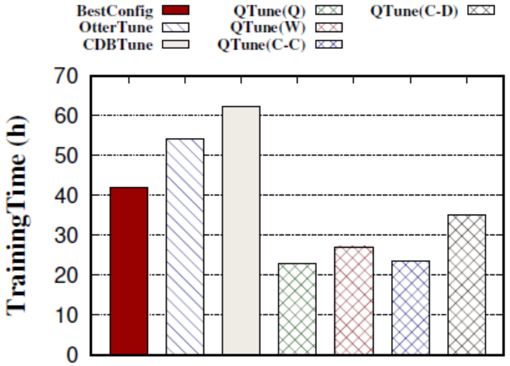
(e) JOB (RO)



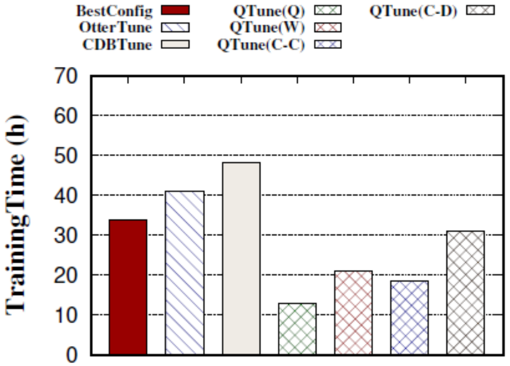
(f) TPC-H (RO)



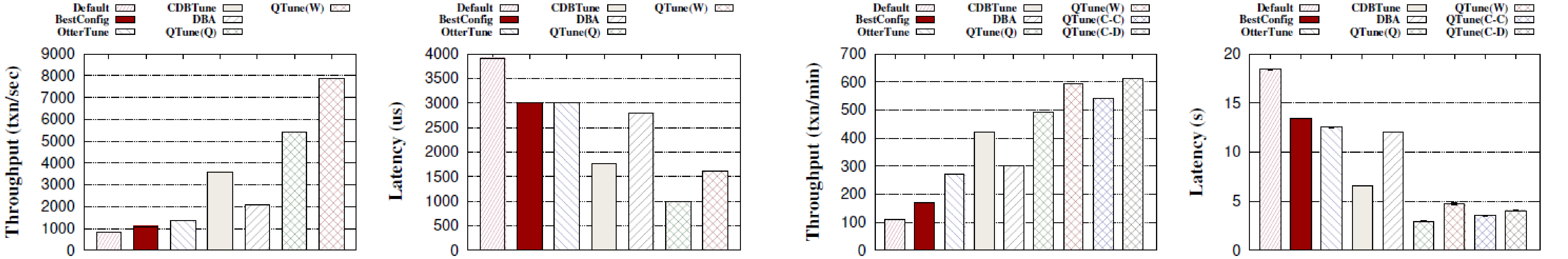
(g) Sysbench (RW)



(h) JOB (RO)

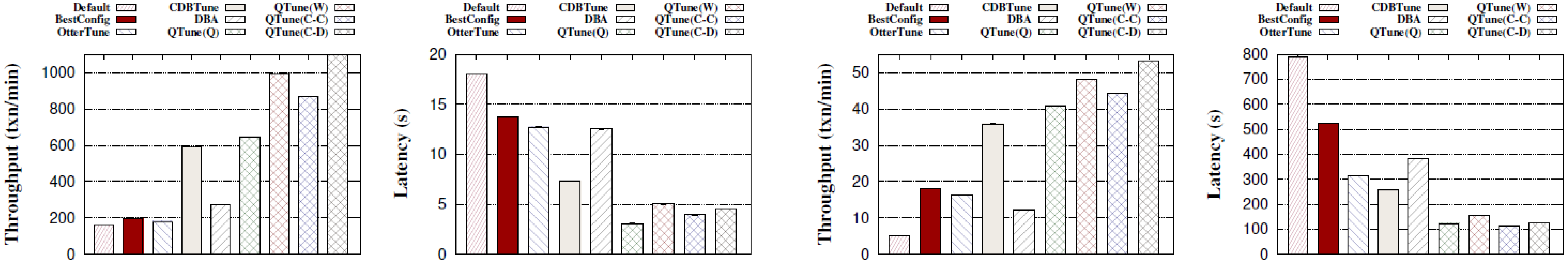


(i) TPC-H (RO)



(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

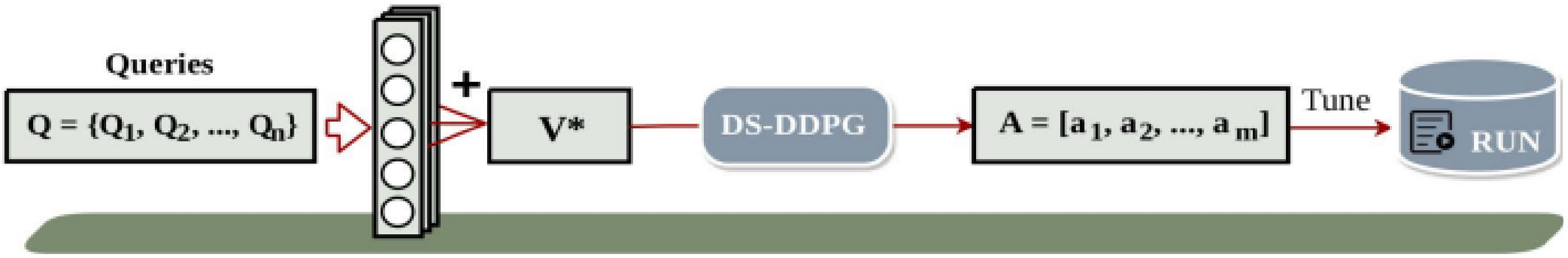
Figure 10: Performance for different databases.

Proposed model (**Qtune**)

- **(1)** We propose a **query-aware database tuning system using deep reinforcement learning**, which provides three database tuning granularities
- **(2)** We propose a **SQL query featurization model** that featurizes a SQL query to a vector by using rich SQL features
- **(3)** We propose the **DS-DDPG model**, which embeds the query features and utilizes the actor-critic algorithm to learn the relations among queries, database state and configurations to tune database configurations
- **(4)** We propose a **deep learning based query clustering method** to classify queries according to the similarity of their suitable configurations

V*

- Query2Vector **generates a feature vector for each query in the workload and merges them to generate a unfied vector.(→ V*)**

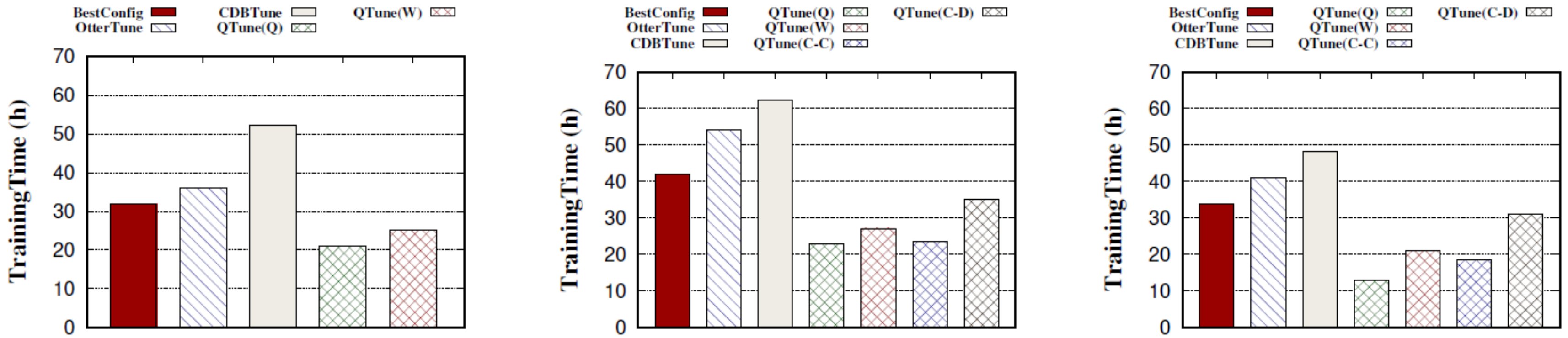


튜닝시 DB 직접 실행

- it first tunes the database knobs and **then executes the query.**

- the session-level knobs (e.g., bulk write size) **can be concurrently tuned** for different queries, while the system-level knobs (e.g., working memory size) cannot be concurrently tuned because when we tune these knobs for a query, **the system cannot process other queries.**

튜닝시 DB 직접 실행



(g) Sysbench (RW) (h) JOB (RO) (i) TPC-H (RO)

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.

Query plan, query optimizer

- DBMS에 내장된 optimizer를 통해 계산(selection cost, join cost)

https://www.oracle.com/search/results?Ntt=query%20plan&Dy=1&Nty=1&cat=mysql&Ntk=SI-ALL5
https://www.postgresql.org/docs/10/using-explain.html
https://docs.mongodb.com/manual/core/query-plans/

Predictor의 outer metrics

- Database의 metric을 의미 (e.g., latency, throughput)

Experiment

- As restarting database is not acceptable in many real business applications, here we only **use the knobs that do not need to restart databases**.

- **MongoDB** is a document-oriented NoSQL Database. **It uses json format queries** rather than SQL. To run a SQL benchmark, we **convert the data sets into json documents** before injecting them into the database and transforms the SQL queries to json format queries.

- use three query workloads **JOB**, **TPC-H** and **Sysbench**.

- http://initd.org/psycopg,scikit-learn.org,numpy.org
- https://www.mongodb.com/
- https://github.com/gregrahn/join-order-benchmark
- http://www.tpc.org/tpch/
- https://github.com/akopytov/sysbench