Reinforcement Learning for Database Indexing

Gabriel Paludo Licks

Automated Planning 2018/2

Overall scenario and approach

Overall scenario

Databases with large volumes of data have a constant challenge of achieving satisfactory response time for its users

WE USE INDEXES!

One of the solutions for performance improvement, especially when it comes to processing complex queries

- ➔ Crucial to have a balance in the amount of indexed columns
- ➔ Overhead during the index look-up process
- ➔ Finding the correct columns to be indexed

Overall scenario

RL is a potential approach for index tuning!

- ➔ Basu et al. (2016). Sharma, Shuhknecht, and Dittrich (2018)

Implement a RL agent using the Q-Learning algorithm to act over a database

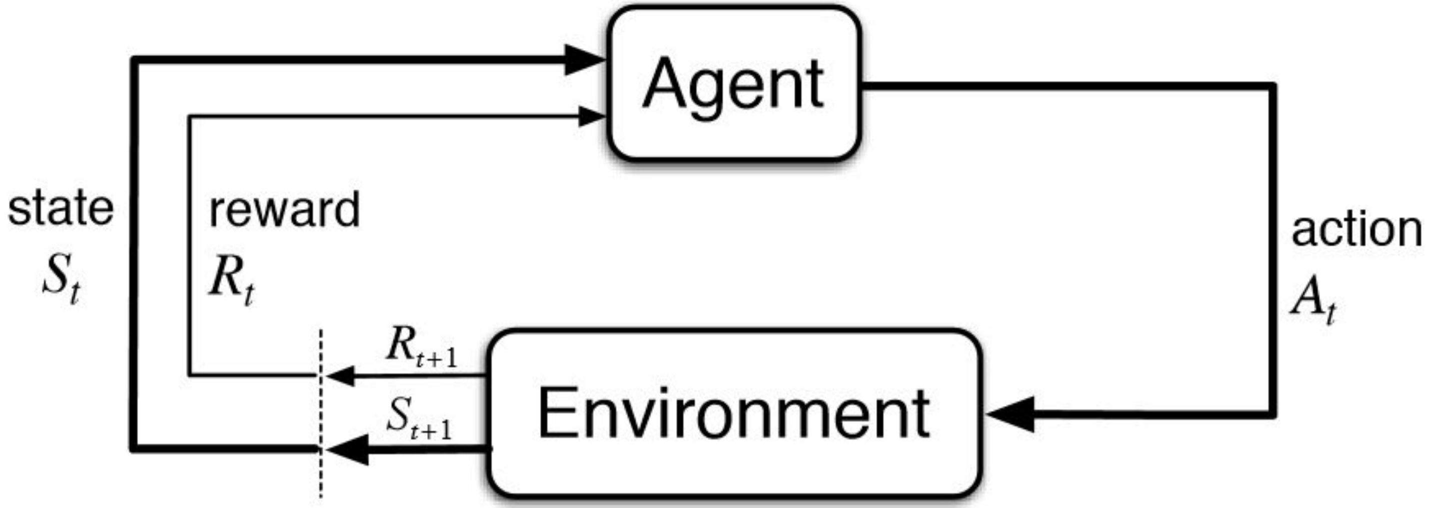
- ➔ Alter its index configuration by creating or dropping column indexes
- ➔ Explore different index recommendations and combinations

Recommend indexes in between query workloads and evaluate performance

We carry out agent training and we evaluate the index configurations obtained by the agent to three other baseline configurations

1. The standard index configuration of the database (PKs and FKs only)
2. All table columns indexed
3. Expert-based index configuration based on a DBA-style analysis

Approach



Source: Sutton, R. S., Barto, A. G., & Bach, F. (1998). Reinforcement learning: An introduction. MIT press.

Approach

- The Q-learning is one of the algorithms for solving MDPs
- It learns values of state-action pairs denoted by *Q(s, a)*:

$$Q(s,a) \leftarrow Q(s,a)+\alpha(r+\gamma max_{\mathrm{a}'}Q(s',a')-Q(s,a))$$

- However… unfeasible to incrementally update values for large state spaces
- We use Linear Function Approximation to generalize from states it has visited to states it has not

$$\hat{Q}(s,a) \leftarrow \theta_0 + \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s)$$

$$\theta_i \leftarrow \theta_i + \alpha[R(s)+\gamma max_{a'}\hat{Q}_\theta(s',a')-\hat{Q}_\theta(s,a)]\frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

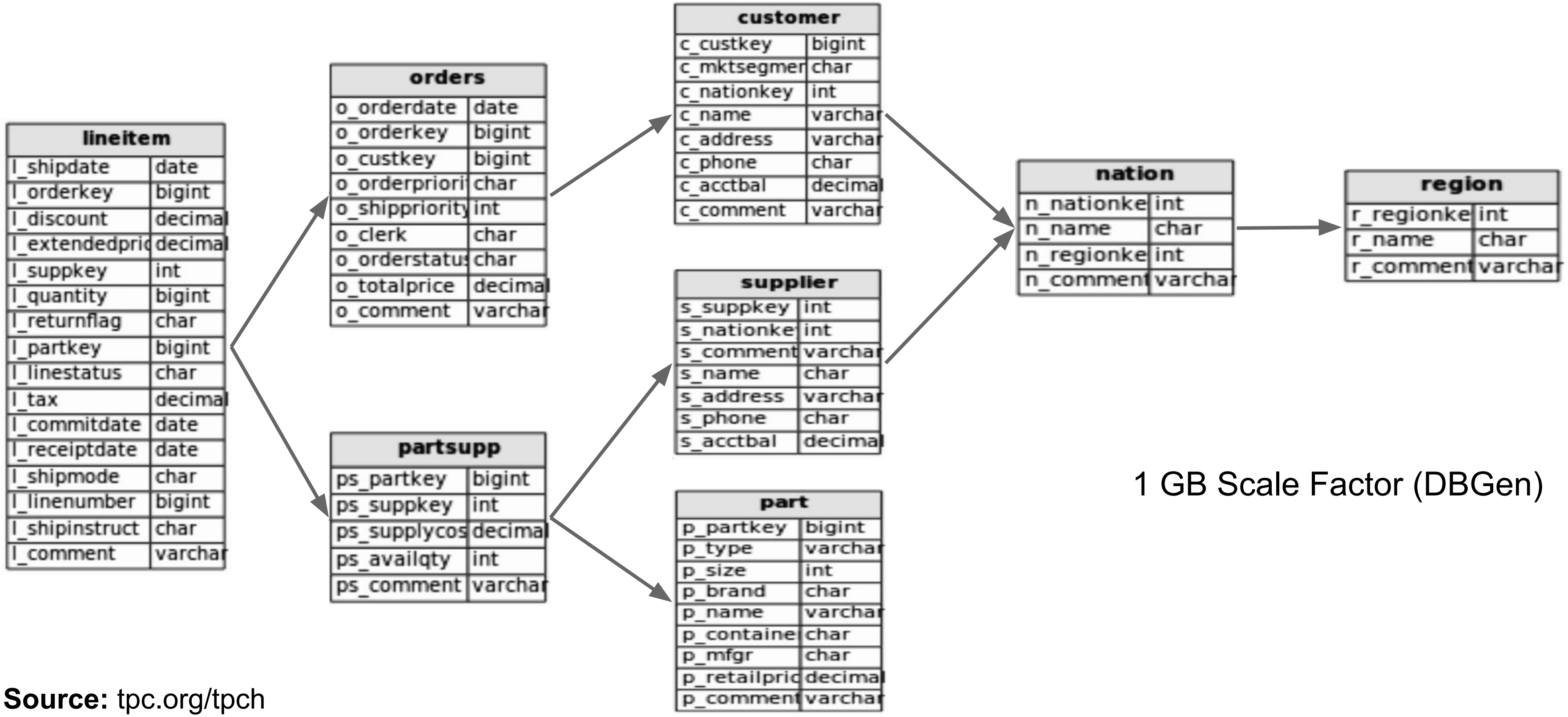
Implementation

Implementation

- ➔ **Environment:** TPC-H database in MySQL DMBS
- ➔ **States:** database index configuration
- ➔ **Actions:** create or drop indexes (non-composite)
- ➔ **Reward:** TPC-H benchmark performance metric

- We use a reinforcement learning agent to explore the space of possible index configurations in the database
- Evaluate each configuration explored using a benchmark

Environment - TPC-H Relational model



lineitem	
l_shipdate	date
l_orderkey	bigint
l_discount	decimal
l_extendedpric	decimal
l_suppkey	int
l_quantity	bigint
l_returnflag	char
l_partkey	bigint
l_linestatus	char
l_tax	decimal
l_commitdate	date
l_receiptdate	date
l_shipmode	char
l_linenumber	bigint
l_shipinstruct	char
l_comment	varchar

orders	
o_orderdate	date
o_orderkey	bigint
o_custkey	bigint
o_orderpriori	char
o_shippriority	int
o_clerk	char
o_orderstatu	char
o_totalprice	decimal
o_comment	varchar

partsupp	
ps_partkey	bigint
ps_suppkey	int
ps_supplycos	decimal
ps_availqty	int
ps_comment	varchar

customer	
c_custkey	bigint
c_mktsegmer	char
c_nationkey	int
c_name	varchar
c_address	varchar
c_phone	char
c_acctbal	decimal
c_comment	varchar

supplier	
s_suppkey	int
s_nationke	int
s_comment	varchar
s_name	char
s_address	varchar
s_phone	char
s_acctbal	decimal

part	
p_partkey	bigint
p_type	varchar
p_size	int
p_brand	char
p_name	varchar
p_containe	char
p_mfgr	char
p_retailpric	decimal
p_comment	varchar

nation	
n_nationke	int
n_name	char
n_regionke	int
n_comment	varchar

region	
r_regionke	int
r_name	char
r_comment	varchar

1 GB Scale Factor (DBGen)

Source: tpc.org/tpch

Environment - TPC-H Relational model

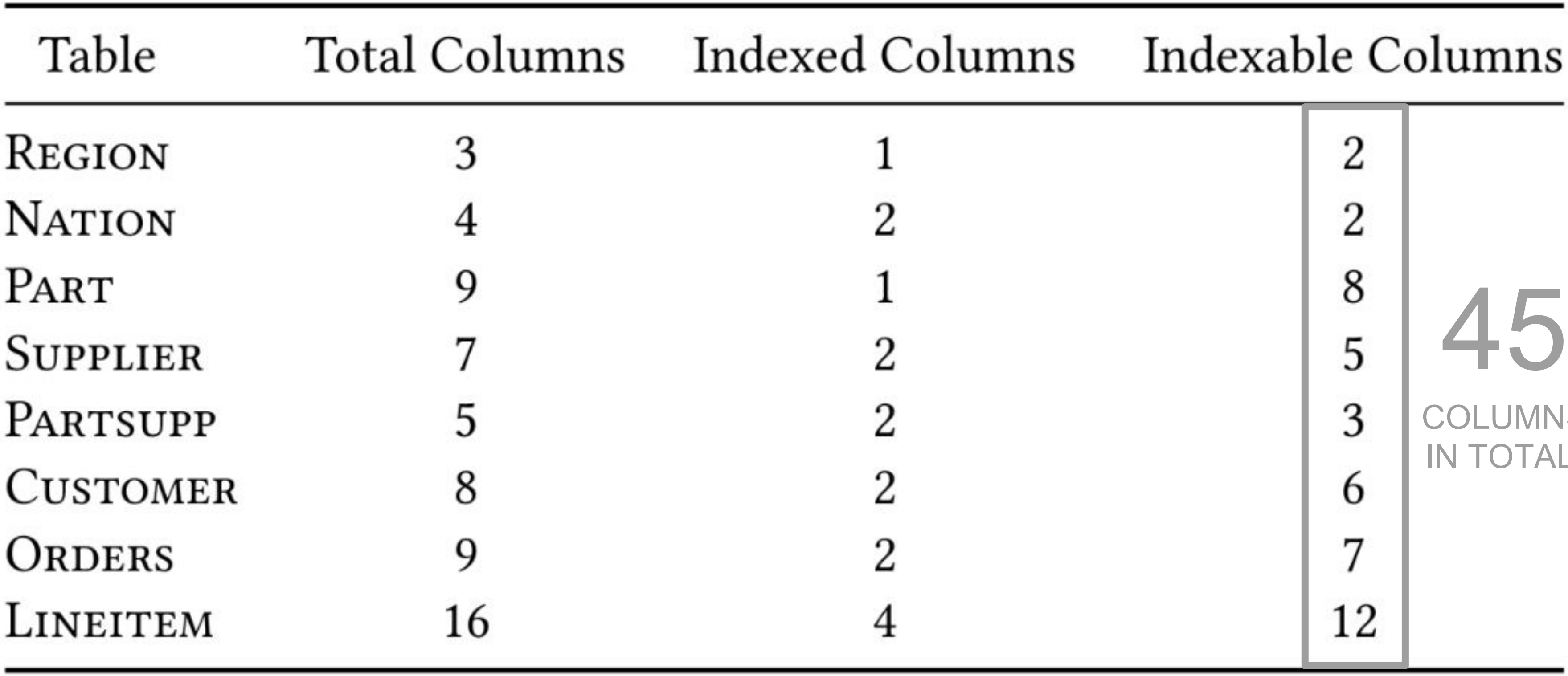


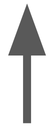
Table	Total Columns	Indexed Columns	Indexable Columns
REGION	3	1	2
NATION	4	2	2
PART	9	1	8
SUPPLIER	7	2	5
PARTSUPP	5	2	3
CUSTOMER	8	2	6
ORDERS	9	2	7
LINEITEM	16	4	12

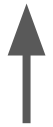
45

COLUMNS
IN TOTAL!

State and actions

TABLE							
COLUMN 1	**COLUMN 2**	**COLUMN 3**	**COLUMN 4**	**COLUMN 5**	**COLUMN 6**	**...**	**COLUMN N**
0	0	1	0	1	1	...	0





Not indexed

Available action
COLUMN 2, CREATE

Indexed

Available action
COLUMN 6, DROP

Reward - TPC-H Performance metrics

We need a feedback from the environment!

We have implemented the TPC-H Benchmark to evaluate DB performance

$$QphH@Size = \sqrt{Power@Size \times Throughput@Size}$$

This metric reflects the DB's performance for processing queries, inserts/updates

We run the TPC-H benchmark every time the agent transitions to a new state

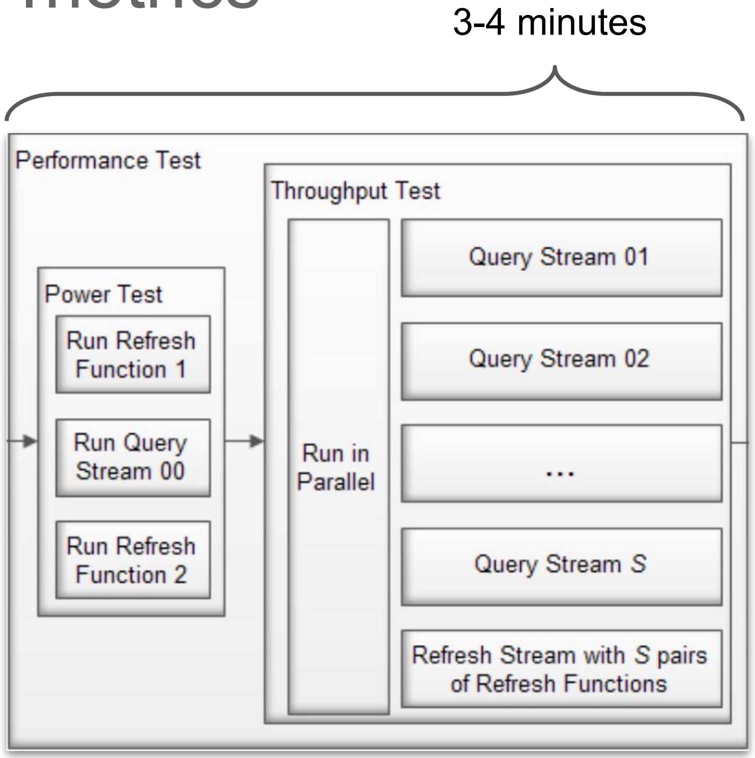
The highest metric rewarded is used to evaluate whether it found an optimizing index configuration

Reward - TPC-H Performance metrics

$$Power@Size = \frac{3600}{\sqrt[24]{\prod_{i=1}^{22} QI(i,0) \times \prod_{j=1}^{2} RI(j,0)}} \times SF$$

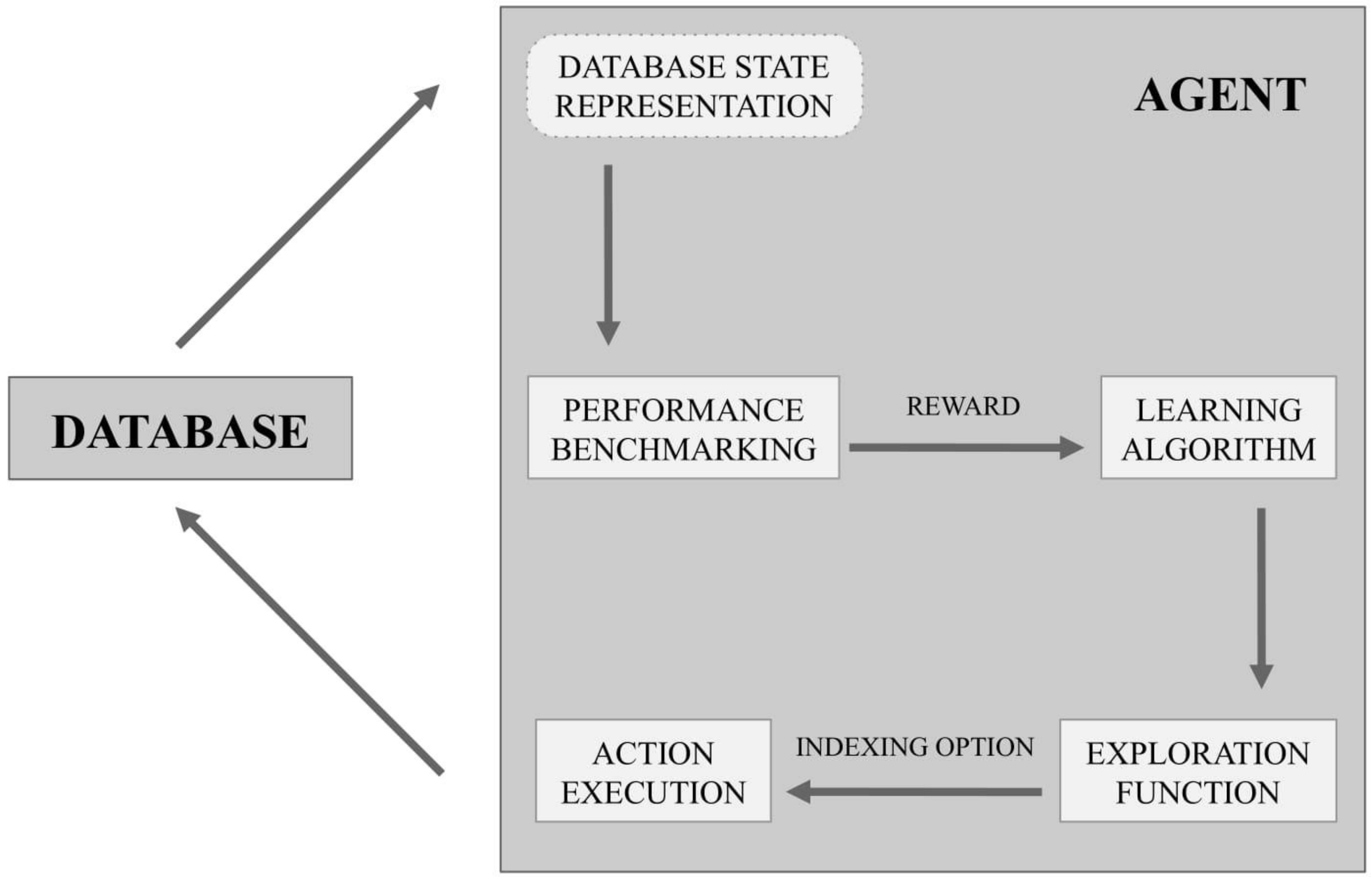
$$QphH@Size = \sqrt{Power@Size \times Throughput@Size}$$

$$Throughput@Size = \frac{S \times 22}{T_s} \times 3600 \times SF$$



Source: Thanopoulou, A., Carreira, P. and Galhardas, H., 2012. *Benchmarking with TPC-H on off-the-shelf Hardware*. ICEIS (1), pp.205-208.

Overview


DATABASE STATE REPRESENTATION
AGENT
DATABASE
PERFORMANCE BENCHMARKING
REWARD
LEARNING ALGORITHM
ACTION EXECUTION
INDEXING OPTION
EXPLORATION FUNCTION

Experiments

Baseline configurations

We have defined three baseline configurations:

1. **TPC-H default configuration:** contains only primary and foreign keys, and secondary indexes from the TPC-H benchmark default configuration.
2. **Expert-based configuration:** a configuration based on a DBA-style analysis of the database workload.
3. **All tables indexed:** with this configuration, we aim to evaluate the performance of the database workload having all table columns indexed.

We benchmarked these configurations (12 executions and a trimmed mean) and we later compare to the results obtained by the agent.

Experiments

Resources used

- ➔ 12-core Intel Xeon @ 2.40GHz CPU
- ➔ 16 GB of RAM
- ➔ Ubuntu Linux 18.04 and Python 3.6.6

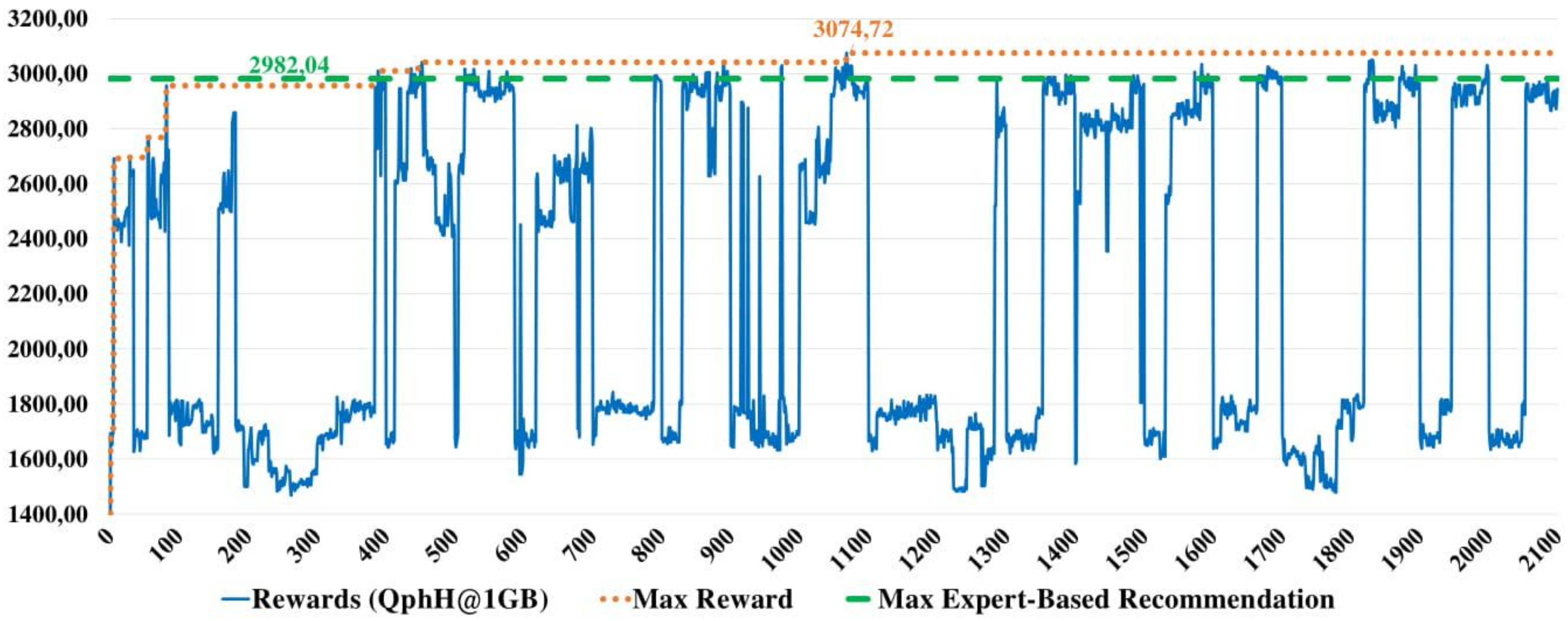
We trained the agent over the course of **21 episodes**

- ➔ Each episode composed of **100 steps**
- ➔ Approximately **5 days** of execution

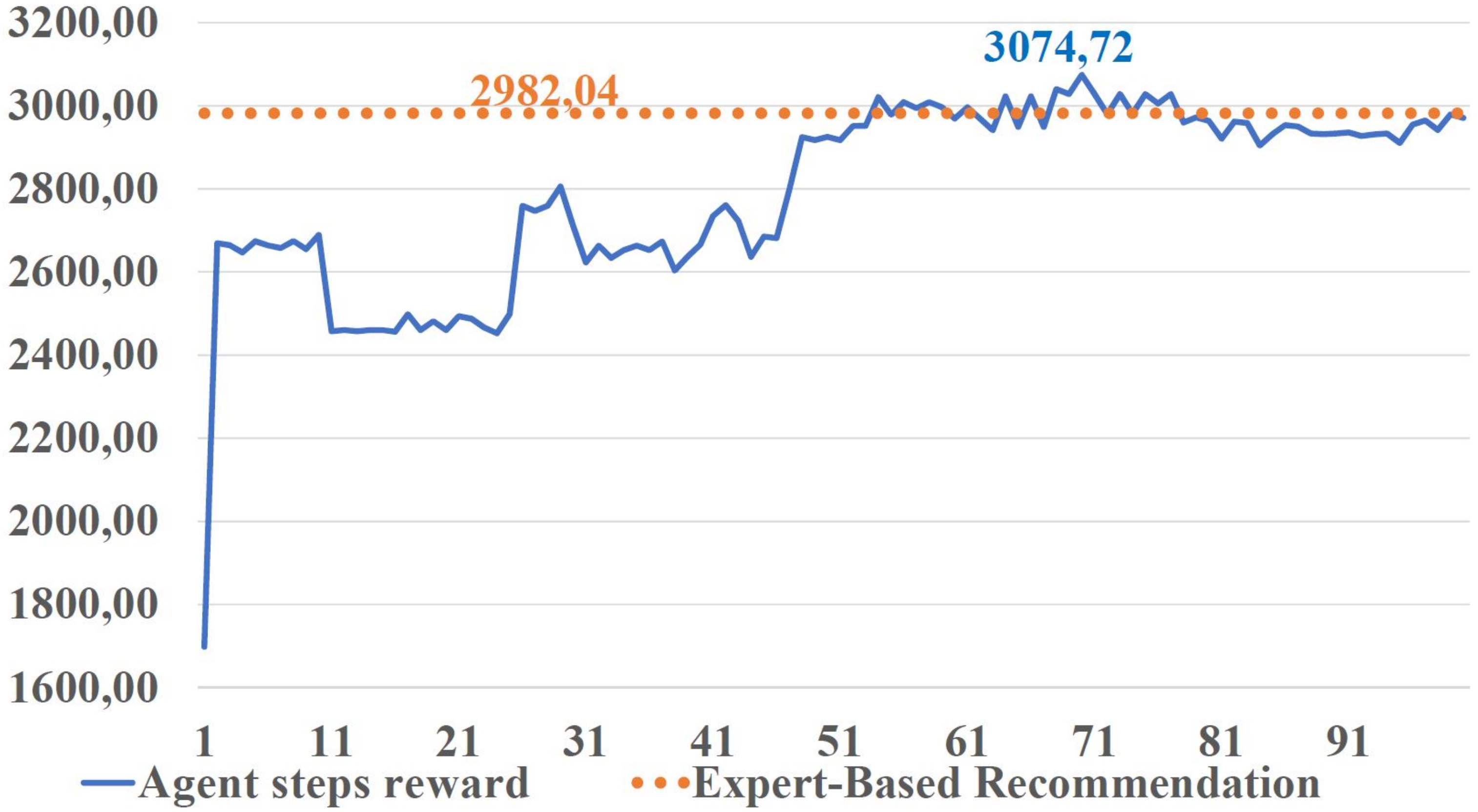
Epsilon decay in a logarithmic scale according to the episode number

From **2100 states visited**, 1711 of them were distinct states

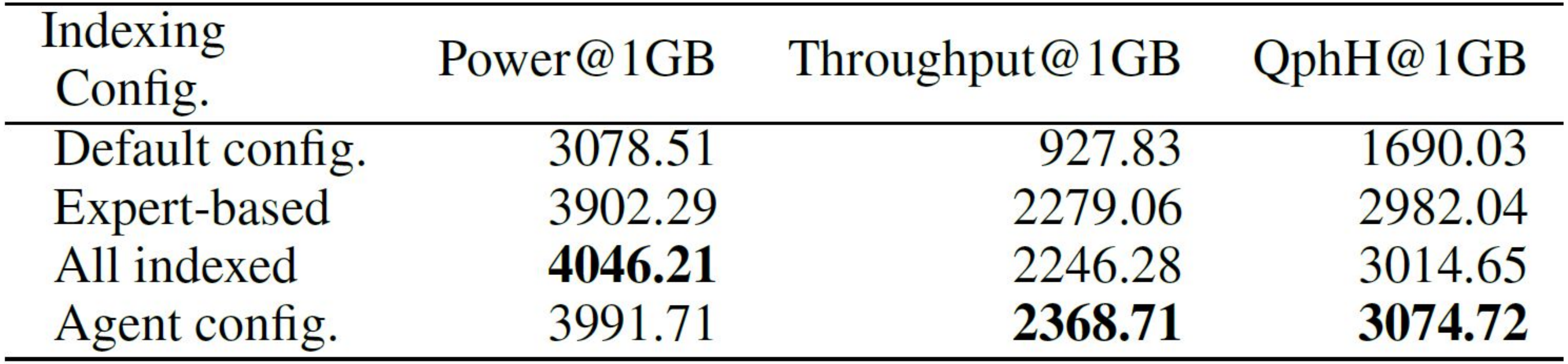
Experiments



Experiments



Experiments



Indexing Config.	Power@1GB	Throughput@1GB	QphH@1GB
Default config.	3078.51	927.83	1690.03
Expert-based	3902.29	2279.06	2982.04
All indexed	**4046.21**	2246.28	3014.65
Agent config.	3991.71	**2368.71**	**3074.72**

Agent increase comparing to expert-based:

- ➔ **2.30%** on Power@1GB
- ➔ **3.93%** on Throughput@1GB
- ➔ **3.11%** on QphH@1GB

Approximately 92 more queries could be ran per hour with the configuration obtained by the agent.

Limitations and future work

- ➔ These results can probably be maximized with auto-encoded state compression
- ➔ The agent knowledge is still limited to the current indexed columns
  - ◆ We could consider the cost of transition from one state to another
  - ◆ Aggregate other useful information to the state description
  - ◆ Consider only the indexes being used by queries for faster convergence
- ➔ Consider creating composite indexes and other types of indexes
- ➔ We limited our experiments to a 1 GB database

Final considerations

Creating indexes is a recurring task for DBAs

- ➔ In reactive situations as well!

Analyzing the cost that an index implies on a database is not easy

- ➔ Time consuming task to be performed frequently
- ➔ Especially when analyzing many possible configurations

The RL agent has the ability to explore a higher number of combinations

Our empirical evaluation shows that the agent provides at least equivalent, and often superior, indexing choices compared to expert-based recommendations

Thank you!

Questions?

Gabriel Paludo Licks
gabriel.paludo@acad.pucrs.br