Reinforcement Learning for Database Indexing

Gabriel Paludo Licks
Graduate Program in Computer Science - School of Technology
Pontifical Catholic University of Rio Grande do Sul - PUCRS
Porto Alegre, Brazil
gabriel.paludo@acad.pucrs.br

Abstract

The index configuration of a database has a direct impact on its performance and response time. However, choosing the right set of indexes is not a trivial task. This is a task carried out by database administrators and is often time consuming, especially when analyzing complex workloads and a number of possible combinations of columns to index. We propose a reinforcement learning agent using the Q-Learning algorithm to explore different index configurations in a relational database model, using a benchmark to evaluate their performance. The agent has the ability to explore many index configurations through as many iterations as defined for training. Among the configurations explored by the agent, our experiments show that the agent provides at least equivalent, and often superior, indexing choices than other baseline configurations we compare.

Introduction

Databases with large volumes of data have a constant challenge of achieving satisfactory response time for its users. Among many other techniques, the use of indexes in a database is one of the solutions for performance improvement, especially when it comes to processing complex queries (Ramakrishnan and Gehrke 2000). More than finding the correct columns to be indexed, it is crucial to have a balance in the amount of indexed columns. Too many indexes, for example, could result in an overhead during the index look-up process. Thus, this is a fundamental task that needs to be performed with diligence, as the indexing configuration directly impacts on a database's overall performance.

A study from Basu et al. proposed the use of reinforcement learning for suggesting indexes to create or drop one query at a time. Another study, from Sharma, Schuhknecht, and Dittrich, used a reinforcement learning approach to recommend a column to be indexed, given a set of queries provided by the TPC-H Benchmark. Both authors used reinforcement learning for index recommending, being the first one limited to working with one query at a time, and the latter by recommending only one column to be indexed given a set of queries.

Given that reinforcement learning is a potential approach for database index tuning, we use it to model an agent for performing over a database exploring different index configurations by creating or dropping column indexes. It uses the Q-Learning algorithm for exploring different configurations and a benchmark for database workload and performance evaluation. In addition, unlike previous work, this proposal aims to recommend indexes in between whole query workloads. Lastly, the goal is to retrieve the best resulting set of indexes among the ones explored by the agent.

We carry out experiments in which we evaluate the index configurations obtained by the agent and compare it to three other baseline configurations: (1) the standard index configuration of the database (primary and foreign keys indexed only); (2) all columns indexed; and (3) an expert-based index configuration from a DBA-style analysis. Among the index configurations explored by the agent, results show that these are at least equivalent and often better than an expert-based configuration.

Approach

Reinforcement Learning (RL) is an approach to learn optimal agent policies in stochastic environments modelled as a Markov Decision Process (MDP) (Bellman 1978). In this model, an environment is framed as a set of states and a stochastic transition system whose transitions the agent influences by choosing from a set of actions. The goal is to control the system in a way that the expected agent reward is maximized over time (Bellman 1978).

In order to solve an MDP, an agent needs to know the state-transition and the reward functions. However, for some cases, we are unable to define them precisely. Thus, an agent has to interact with the environment by taking sequential actions to collect information and explore the search space by trial and error (Sutton and Barto 2018). One of the algorithms for solving MDPs with unknown reward and transition functions is the Q-learning. This method learns values of state-action pairs denoted by $Q(s, a)$, which is the value of taking action a in state s (Sutton and Barto 2018). The basic idea of the algorithm is to use the Q-function (Equation 1) to incrementally estimate values of $Q(s, a)$ based on reward signals from each action taken.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma max_{a'} Q(s',a') - Q(s,a)) \quad (1)$$

These values learned by the agent are stored and represented in tabular structure with one output value for each input tuple (Russell and Norvig 2016). However, when we have larger state spaces and branching factors, such as our case, it becomes unfeasible to visit all states many times and store them in a tabular structure (Russell and Norvig 2016). One way to handle such problems is to use function approximation, where we use a weighted linear function of a set of features from our states to approximate our values (Equation 2):

$$\hat{Q}(s,a) \leftarrow \theta_0 + \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s) \quad (2)$$

The compression achieved by function approximation allows the agent to generalize from states it has visited to states it has not. In addition, for temporal-difference learners, we need to adjust the parameters to try to reduce the temporal difference between successive states and move the parameters in the direction of decreasing the error after each trial (Russell and Norvig 2016). For that, we use Equation 3.

$$\theta_i \leftarrow \theta_i + \alpha[R(s) + \gamma max_{a'} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)] \frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i} \quad (3)$$

Considering this background knowledge, we model and implement a software agent in order to explore different index configurations in a database using the Q-learning algorithm with function approximation. Moreover, for modelling a RL agent, there are a few aspects that we need to formalize: the environment in which the agent will act in; the state representation of this environment; the actions the agent will be able to execute; and a reward measure as a feedback from the environment to the agent.

Implementation

This section details how we modelled and implemented the agent to perform index tuning in a database. We describe the database and the benchmark we use to evaluate performance. Then, we formalize the problem we use our agent to address and detail how we implement it.

Database and benchmark

Throughout this project, we intensively make use of the TPC-H benchmark and database. TPC stands for Transaction Processing Performance Council, a well-known environment that produces benchmarks to measure transaction processing and database performance (TPC 1998). The TPC in its version H provides a relational model with 8 tables, along with a query generator (QGen) that creates 22 queries with different levels of complexity, and a data generator (DBGen) which is able to create up to 100 TB of data to load in a Database Management System (DBMS). The queries and the data populating the database are chosen to have broad industry-wide relevance (TPC 1998). We create this database using MySQL DBMS and populate it with 1 GB of data provided by DBGen.

We also use the TPC-H benchmark protocol for evaluating database performance. The performance metric reported by TPC-H is called the Composite Query-per-Hour Performance Metric ($QphH@Size$), where $Size$ means the scale factor of the database (1 GB in our case). This metric, however, is composed of two other metrics that have to be computed beforehand: the $Power@Size$ and the $Throughput@Size$. The $Power@Size$ evaluates how fast the system computes the answers to a raw sequential execution of the 22 queries provided by QGen (TPC 1998). The $Throughput@Size$ measures the ability of the system to process parallel executions of queries, aiming to represent system performance against a multi-user workload (TPC 1998). The resulting $QphH@Size$ metric summarizes both single and multi user overall database performance calculated by the geometric mean of the two previous metrics.

Table 1: TPC-H Database - Tables and indexable columns

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

Problem formalization

Given the amount of 8 tables available in our database, we have in total 45 columns that are available to be indexed. In Table 1, we map the number of columns per table in the TPC-H database, the number of columns that come indexed by default (primary and foreign keys) and the number of available columns for indexing. Considering that a column is either indexed or not, from the 45 available columns for indexing, we have a total of 2^{45} (over 35 trillion) possible index configurations for our database. This is the number of options that a person who is not a DBA nor familiar with database tuning would have to test manually, in order to analyze the results and get an optimal solution regarding the performance of a set of queries performed in a database. The amount of index combinations reinforces our need of implementing an agent that uses function approximation in order to generalize states.

Agent modelling

We implement an agent which uses the Q-learning algorithm with function approximation in order to automatically choose indexes for the database using reinforcement learning. We organize the agent executions in episodes and steps within each episode. Each step consists of the agent choosing an action, executing it in the environment and receiving a reward signal. Then, at the end of each step, we update our set of Θ parameters used for approximating our state-action values based on Equation 3.

The environment in which the agent will be acting in is the TPC-H database we previously mentioned. The database was implemented using MySQL, with the relational model and data provided by the TPC-H Benchmark. We define our environment state by the database's index configuration, which informs the columns that are indexed or not. For

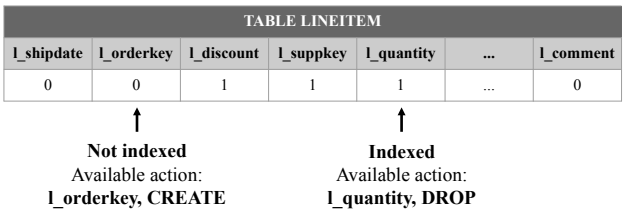

Figure 1: Representation of state and available actions.

that, we represent it by a binary vector mapping all the table columns, for each of them containing 1 if the column is indexed, or 0 if the column is not indexed in the database. Figure 1 illustrates a snippet of this binary vector, using columns of the LINEITEM table as an example.

As illustrated in Figure 1, we can also extract the available actions in a given state by iterating through the binary vector. For every column in the vector, there is one available action, which is create or drop an index on the column. If a column is not indexed, the available action is CREATE, otherwise available action is to DROP. For each episode step, the agent chooses an action to take based on an epsilon-greedy exploration function. Therefore, with an ϵ-probability, the agent chooses whether to take a $random\,action_a$ or an $arg\,max_a$ action at a given state. Since our agent does not have a terminal state, the agent will perform creation and deletion actions until the agent gets to the last episode.

We also need to decide a reward function, which is a very important assignment whereas it impacts on the learning quality. The $QphH@Size$ performance metric reported by TPC-H reflects multiple aspects of the capability of a database to process queries (TPC 1998). In order to get a feedback whether a given index configuration has increased or decreased performance, we use the TPC-H $QphH@Size$ performance metric as the agent's reward signal. Thus, whenever the agent transitions to a new state, the benchmark is executed in order to retrieve the corresponding reward for that state. The highest performance metric rewarded, then, will be used to evaluate whether the agent found an optimizing index configuration among the explored ones.

Finally, we fixed each episode to execute 100 steps in the environment. Specifically, at any given step, the agent follows its policy and executes an action of either creating or dropping an index in a given column, this index is then created or dropped and the resulting performance metric is returned by the benchmark in order to evaluate how good is that action in that state. In addition, after each episode, we also decay the epsilon value used in our exploration function in a logarithmic scale according to the episode number in order to start exploiting states with higher utility.

Experiments

We report experiments where we compare the index configuration obtained by the agent to three other baseline index configurations: (1) with all table columns indexed; (2) with the TPC-H default index configuration; and (3) with an expert-based index configuration. We start by explaining the baseline configurations we tested and later report the results obtained using the reinforcement learning agent.

Table 2: TPC-H Metrics - Index Configuration Comparison

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**

The configuration we established as a baseline for our experiments are:

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.

For the expert-based configuration, we made it through a traditional DBA-style analysis: we examined the database and queries performed in the workload. We analyze the columns contained in each "WHERE" clause of each of the 22 queries. Then, we employ the MySQL EXPLAIN command to visualize the cost of each query step, which allows us to identify indexing opportunities for columns that impact on higher costs. We create candidate indexes and we run the explain plan again to evaluate whether the queries cost actually decreased. Finally, we keep the indexes that impacted in a reduction in the queries cost, and we drop the indexes that did not improve performance.

Then, we compute the performance metrics to each baseline configuration, using a Python script we implemented for executing the TPC-H benchmark. We carried out these experiments on a 12-core Intel Xeon @ 2.40GHz CPU and 16 GB of RAM running Ubuntu Linux 18.04 and Python 3.6.6. For each execution of the TPC-H benchmark, we first calculated the Power (Equation 1) and Throughput (Equation 2) metrics in a scale factor of 1 and running the 22 queries in MySQL, according to the TPC-H benchmark protocol (TPC 1998). Finally, we calculated the QphH (Equation 3) for each configuration, which is the geometric mean of both metrics calculated beforehand.

Table 2 shows the results of our experiments for each metric in each configuration with the best results in each row highlighted in bold. These results suggest that the expert-based indexing configuration shows better performance than the initial, but not better than indexing all columns. Indexing all columns just incurs in greater performance because our database is only 1GB, thus it is still possible to maintain all tables and indexes in main memory, consequently improving query performance. However, when available memory is an issue, we have to take in consideration the size in disk occupied by all these indexes.

As for the agent configuration, we carried out agent training over the course of 21 episodes, which took approxi-

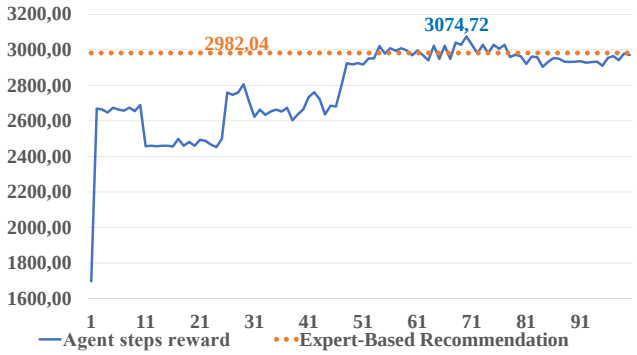

Figure 2: Agent's tenth episode training.

mately 5 days of training. As we mentioned earlier, each episode is composed of 100 steps, thus our agent visited 2100 states during its training and for each state we calculate the reward by executing the TPC-H benchmark, which is a 3-4 minutes long execution. From these states visited, 1711 of them were distinct states and the remainder are states that were visited more than once. Considering the amount of 2^{45} possible states, with only 1711 of them visited, we have already obtained configurations with better performance if compared to the results of the baseline configurations.

Among the configurations explored by the agent over the episodes trained, we obtained the highest rewarded configuration during the tenth episode, with a QphH@1GB of 3074.72. If compared to the expert-based configuration, which resulted in a QphH@1GB of 2982.04, approximately 92 more queries could be ran per hour with the configuration obtained by the agent, representing an increase of 3.11%. We also had a performance increase of 2.30% on Power@1GB and 3.93% on Throughput@1GB.

In Figure 2 we plot the rewards obtained during the tenth episode. We have the resulting reward in queries per hour (QphH@1GB) in the Y-axis and the corresponding step to that reward in the X-axis throughout the whole episode. The blue line shows the actual reward the agent obtained in each state. With the green dashed line, we show the QphH@1GB we obtained with expert-based indexing, which we plot in order to compare it to the explored configurations of the agent during the training. From this plot, we can visualize that, after a few steps of training, the agent starts exploring index configurations that are superior or equivalent in queries per hour than the expert-based.

Related work

As we mentioned in the introduction, a few studies used reinforcement learning for index tuning. A study from Basu et al. proposed the use of reinforcement learning for suggesting indexes to create or drop one query at a time. However, their implementation was limited to suggest one index one query at a time, and not in between whole workloads as we implement. Also, they did not consider metrics from a standardized benchmark, as they only used response time related to the requesting queries for evaluating performance.

Another study, from Sharma, Schuhknecht, and Dittrich, applied reinforcement learning to recommend columns to be indexed given a set of queries from TPC-H workload. The authors, however, decided not to use the TPC-H protocol to evaluate performance. Instead, they chose a set of queries to measure the execution time of them. We, on the other hand, strictly follow the TPC-H protocol in order to execute whole query workloads in between index recommendations and use the benchmark provided by TPC-H itself to evaluate performance.

Conclusion

Analyzing the cost that an index implies in the database performance is a recurrent and time consuming task to be performed. Specifically, when database administrators need to analyze many possible sets of indexes for achieving the best configuration, the use of a reinforcement learning agent is a solution that affords exploring a significantly higher number of combinations throughout as many iterations as defined. In this paper, we developed an agent for automated database indexing. Our empirical evaluation shows that the agent provides at least equivalent, and often superior, indexing choices compared to expert-based indexing recommendations.

We believe, though, that these results can be maximized if we consider using auto-encoded state compression. The agent state knowledge is currently limited to the indexed columns of the database. However, we believe that there are substantial information that can be aggregated in the state description and set of features in order to bring an accurately higher representation for predictions using function approximation. We also limited our experiments to a 1 GB size database, which easily fits a workload into main memory as a consequence, whereas in larger databases this would not happen and thus indexes would have a higher impact in performance.

References

Basu, D.; Lin, Q.; Chen, W.; Vo, H. T.; Yuan, Z.; Senellart, P.; and Bressan, S. 2016. Regularized cost-model oblivious database tuning with reinforcement learning. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXVIII*. Springer. 96–132.

Bellman, R. 1978. *An introduction to artificial intelligence: Can computers think?* The University of Michigan: Boyd & Fraser Pub. Co.

Ramakrishnan, R., and Gehrke, J. 2000. *Database management systems*. McGraw Hill.

Russell, S. J., and Norvig, P. 2016. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Sharma, A.; Schuhknecht, F. M.; and Dittrich, J. 2018. The case for automatic database administration using deep reinforcement learning. *arXiv preprint arXiv:1801.05643*.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

TPC. 1998. Transaction performance council website (tpc). http://www.tpc.org/.