



# Automated Vertical Partitioning with Deep Reinforcement Learning

Gabriel Campero Durand<sup>(✉)</sup>, Rufat Piriyeve, Marcus Pinnecke,  
David Broneske, Balasubramanian Gurumurthy, and Gunter Saake

Otto-von-Guericke-Universität, Magdeburg, Germany  
{campero,piriyeve,pinnecke,david.broneske,gurumurthy,saake}@ovgu.de

**Abstract.** Finding the right vertical partitioning scheme to match a workload is one of the essential database optimization problems. With the proper partitioning, queries and management tasks can skip unnecessary data, improving their performance. Algorithmic approaches are common for determining a partitioning scheme, with solutions being shaped by their choice of cost models and pruning heuristics. In spite of their advantages, these can be inefficient since they don't improve with experience (e.g., learning from errors in cost estimates or heuristics employed). In this paper we consider the feasibility of a general machine learning solution to overcome such drawbacks. Specifically, we extend the work in GridFormation, mapping the partitioning task to a reinforcement learning (RL) task. We validate our proposal experimentally using a TPC-H database and workload, HDD cost models and the Google Dopamine framework for deep RL. We report early evaluations using 3 standard DQN agents, establishing that agents can match the results of state-of-the-art algorithms. We find that convergence is easily achievable for single table-workload pairs, but that generalizing to random workloads requires further work. We also report competitive runtimes for our agents on both GPU and CPU inference, outperforming some state-of-the-art algorithms, as the number of attributes in a table increases.

**Keywords:** Vertical partitioning · Deep reinforcement learning

## 1 Introduction

The efficiency of data management tools is predicated on how well their configuration (e.g. physical design) matches the workload that they process. Database administrators are commonly responsible for defining such configuration, but even for the most experienced practitioners finding the optimal remains challenging given: (a) the high number of configurable knobs & possible settings, (b) rapidly changing workloads, and finally (c) the uncertainty in predicting

---

We thank Dr. Christoph Steup, Milena Malysheva and Ivan Prymak for valuable feedback. This work was partially funded by the DFG (grant no.: SA 465/50-1).

the impact of choices when based on cost models or assumptions (e.g. knob independence) that might not match real-world systems. To alleviate these challenges either fully or partially automated tools are used (e.g. physical design advisors [1]). Specially relevant in these tools is the incorporation of machine learning models, helping tools to learn from experience, relying less on initial assumptions. In recent years, due to reinforcement learning methods (RL) outperforming humans in highly complex game scenarios (including Atari games, Go, Poker and real-time strategy games), teams from academia [3, 7, 9, 10, 12] and industry<sup>1</sup> propose data management solutions that learn from real-world signals using RL or deep RL (DRL, i.e., the combination of RL methods, with neural networks for function approximation). In this context, DRL enables models to have a limited memory footprint, a competitive inference process that can use massively-parallel processors and models can generalize from past experiences to unknown states. In this work we study how vertical partitioning can be supported with a novel DRL solution, in order to learn from experience and accelerate the decision making process on real-world signals. Vertical partitioning is a core physical design task, responsible for dividing an existing logical relation into optimally-defined physical partitions, where each partition is a group of attributes from the original relation. The main purpose of this operation is to reduce I/O related costs, by keeping in memory only data that is relevant to an expected workload. The right vertical partitioning improves query performance and other database physical design decisions [8].

Our contributions in this study are: (1) Building upon previous work [3], we design necessary components to make vertical partitioning a problem amiable to be learned with DRL models. (2) We offer a prototypical implementation of our solution, using OpenAI Gym and the Google Dopamine framework for DRL [4], adapting 3 standard value-based agents: DQN, distributional DQN with prioritized experience replay [2, 6] and distributional DQN with implicit quantiles [5]. (3) We contribute a study on the learnability of vertical partitioning for a TPC-H workload, showing that DRL can indeed learn to produce the same solutions as algorithms. We find single table-workloads simple to learn, whereas generalizing to random workloads requires more effort. (4) We show that the out-of-the-box inference of DRL is competitive with the performance of state-of-the-art algorithms, specially as table sizes increase.

We structure our work as follows: We start with a brief background (Sect. 2) and the design of our solution (Sect. 3), covering the rewarding scheme, the observation and action spaces. Our evaluation and results follow (Sect. 4). We conclude by summarizing our findings and outlining future work (Sect. 5).

---

<sup>1</sup> <https://blogs.oracle.com/oracle-database/oracle-database-19c-now-available-on-oracle-exadata>.

## 2 Background

**Automated Vertical Partitioning:** The problem of algorithmically finding the best vertical partitioning is not new – specially for complementing DBA’s manual partitioning [8]. Since this problem is NP complete [1], most solutions seek a trade-off between finding the optimal and the runtime required to produce the result. This trade-off is managed with the adoption of heuristics that prune the search process. Among some solutions, as covered by Jindal et al. [8], we can mention the following: Navathe, one of the earliest algorithms, consisting of keeping an affinity matrix and clustering as a means to create partitions. HillClimb, a simple approach that makes the process iterative, with each iteration exploring all possible combinations into two, of the existing partitions, passing to the next iteration the most promising candidate. This process ends when there is no cost improvement with respect to former iterations. AutoPart, a variant of HillClimb, including categorical partitioning and pruning the search space based on query coverage. O2P is a variant of Navathe, which seeks to perform faster, making more concessions on optimality. Apart from algorithmic solutions, the task has also been studied with unsupervised learning and genetic algorithms.

**Deep Reinforcement Learning:** RL is a class of machine learning solution, where agents are set to learn how to act by interacting with an environment through actions, observing the states of the environment and the rewards obtained. Formally, the scenario is modeled as a Markov decision process. In developing RL solutions, the ability to scale to cases where the space of possible states is quite large poses challenges to the exploration process and to the learned model. Rather than storing all observations, a function approximation solution is required to limit the memory employed, and also to generalize knowledge from visited states to unvisited ones. DRL adopts neural networks for the function approximation task. There are 3 general families of RL and DRL algorithms. Namely, value-based solutions, policy gradient solutions and model-based solutions. The latter form a model of the transitions and expected rewards in the environment, reducing the learning task to a planning problem. From the former approaches, they either learn the long-term value of performing all actions in a state, or simply the policy to pick the best action. In our study we employ 3 value-based methods: DQN, a deep approach to Q-learning relying on experience replay and fixed-Q targets; Distributional DQN with prioritized experience replay (in Dopamine, this model is called Rainbow), which mainly learns to approximate the complete distribution rather than the approximate expectation of each Q-value [2] (in our study we use a categorical distribution, as considered for C51), in addition to using a priority sampling strategy; and Distributional DQN with implicit quantiles, this model extends the previous, using a deterministic parametric function to reparameterize samples from a base distribution to the quantile values of a return distribution [5].

### 3 Design

**Observation Space:** Our state space is represented as a 2D array. For the case of our study, it is of size  $16 \times 23$ . Each column represents one partition. If there is a state with only two partitions, these should take the two leftmost positions of the state array. The remaining columns are set to zero. Hence, this observation space can represent tables of up to 16 columns. This choice fits well the TPC-H database, since the largest table consists of 16 columns (LINEITEM). The first row in the observation space collects the size of each partition, which is the sum of the attribute sizes in the partition, multiplied by the row count of the table. The remaining rows represent 22 queries of a workload. At each position the query holds a 1 if the attribute is accessed by the query, or a 0 otherwise.

**Actions:** Actions can be either to merge or to split partitions, forming bottom-up or top-down processes. In our case we focus on merging. Given the number of columns in our implementation, we have a maximum of 120 actions (which is the number of combinations of 16 items into groups of 2, without repetitions or order). The semantic of actions is given by their number, with action 0 being joining the columns  $[0, 1]$ , and action 119 being joining  $[14, 15]$ . Once joined, the sizes of the columns are added (first row), and the entries in the remaining rows are resolved by a logical OR between the two entries. Finally, the rightmost of the two columns is removed from the observation space, and the result of the operation is stored in the other column, shifting all non-zero columns to the left.

**Rewarding Scheme:** Before defining a reward, we calculate the value (V) of a state as the inverse of the cost (in our case the HDD cost). The reward itself is calculated with (1).

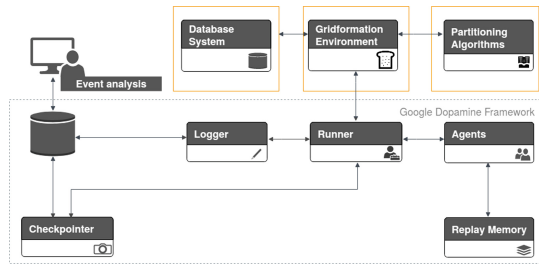
$$Reward_{current\_state} = \begin{cases} 0, & \text{if not end of the game} \\ 100 * \frac{V_{current\_state} - V_0}{\Delta_{best}}, & \text{otherwise} \end{cases} \quad (1)$$

Here  $V_0$  - is the value at the beginning of the episode (with all the attributes in different partitions),  $\Delta_{best} = V_{beststate} - V_0$  - where  $V_{beststate}$  is the value of the state reached when following all the expert steps (getting this value from the expert). By expressing the final reward in terms of the percentage compared to the expert, we are able to normalize the rewards obtained, solving the problem of having rewards at different magnitudes based on table and workload characteristics. As an expert we employ HillClimb. Alternatively, the expert could also be a fixed heuristic (e.g. the cost of supporting the workload with a columnar partitioning), as considered in related work [12]. The game reaches a terminal state (game over) when one of these conditions occur: if the number of columns = 1 (only one partition left), if  $V_{state} < V_{state-1}$ , if the selected action is not valid (e.g. it refers to a column that does not exist in the current table).

## 4 Evaluation

### 4.1 Experiment Configuration

The main components of our prototype are shown in Fig. 1. We created an OpenAI Gym environment, encapsulating the logic of our solution. The *Runner* class from Dopamine was adapted to select the table-workload pair for each episode. This class, in its role of organizing the learning process, also launches an experiment decomposed into iterations, in turn composed of test and evaluate phases. During the experiment the *Runner* fulfills the task of keeping up-to-date the states of the agent and environment, communicating between them, while logging and checkpointing the experimental data. In our implementation, for initializing the environment, it interacts with the vertical partitioning algorithms (e.g. HillClimb, AutoPart, etc.), such that the cost and actions performed by the algorithms/expert can be used to normalize the rewards obtained by the agents during an episode, across diverse cases.



**Fig. 1.** Architecture of our implemented solution, based on the Google Dopamine Framework [4]

**Rewarding and Cost Model:** For our experiments we implemented in our environment the rewarding scheme described previously (Sect. 3). This is based on the value of a state, which in turn depends on a cost model applied to each partition in the given state. For evaluating the cost of each partition we used an HDD cost model employed in previous work [8]. Though we use a cost model to calculate the rewards (i.e., cost-model boot-strapping, as considered in other research [11]) given by our environment to the agents, it should be possible to replace this component, either with other cost models, or with real-world signals, without affecting to a large extent the overall solution.

**Agents:** We adopted the provided agents from the Google Dopamine Framework (DQN, distributional DQN with prioritized experience replay/Rainbow, and implicit quantiles), using provided hyper-parameters. Agents differ in the last layers, which are crucial in determining the predicted values. In our adoption of the agents we changed the last layer, to match the number of actions, and we introduced a hidden layer preceding it, with 512 neurons. As activation function for this layer we used ReLu. When appropriate, we used multi-step variants of the agents.

**Benchmark Data:** We report results on a TPC-H database of scale factor 10 (SF10), using all tables and the complete TPC-H workload. For our observation space we create “views” for how the workload is seen from the perspective of a table. We use state-of-the-art open source Java implementations of the algorithms [8]. For normalization, in our results we use the cost obtained by HillClimb. We could have also used a simple heuristic.

**Hardware:** We used a commodity multi-core machine running Ubuntu 16.04, with an Intel® Core™i7-6700HQ CPU @ 2.60 GHz (8 cores in total), an NVIDIA® GeForce GTX 960M graphic processing unit and 15.5 GiB of memory.

## 4.2 Training

**TPC-H Workloads - Single Table-Workload Pair, or Set of Table-Workload Pairs:** In order to give an optimal partitioning for the tables in the TPC-H workload at SF10 cases generally require few actions, with 4 being the highest (CUSTOMER), and some cases requiring no action (PART). Considering the simplicity of each case, we started training by evaluating the convergence of the agents in finding a solution for each table-workload pair. The results from our evaluation of convergence are shown in Fig. 2. For conciseness, we omit the results for other tables, since they all converge in less than 10 iteration (with 500 training steps per iteration, each iteration corresponding to an average over 100 evaluation steps). Figure 3 considers the training on the set of 8 TPC-H table-workload pairs, using an update horizon of 3 (i.e., multi-step of 3). From these results we see, at first, that when going beyond 1 table-workload pair, convergence requires up to 300 iterations, overall. In this experiment we compare agents with regards to pruning or not pruning actions. This refers to the heuristic of pruning the Q-values obtained from the neural network, by artificially removing those that correspond to actions that would be invalid (i.e., we hard-code it before choosing an action, rather than letting the agent explore bad actions). Results show that by not pruning actions, convergence on very simple cases takes more time.

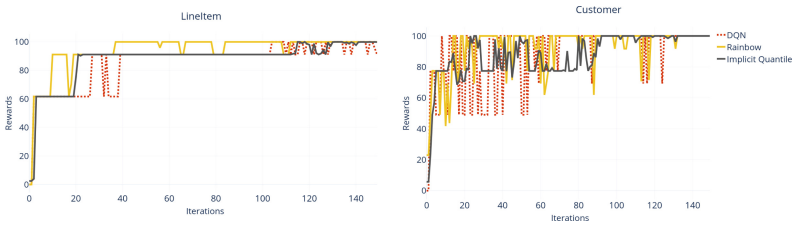


Fig. 2. Training for single table-workload pairs

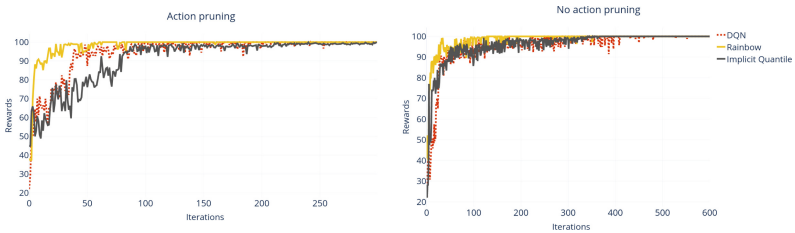
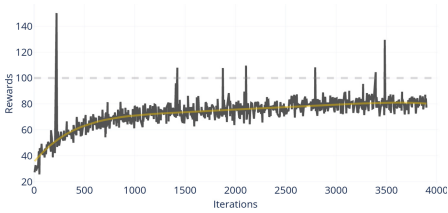


Fig. 3. Pruning or non-pruning invalid actions

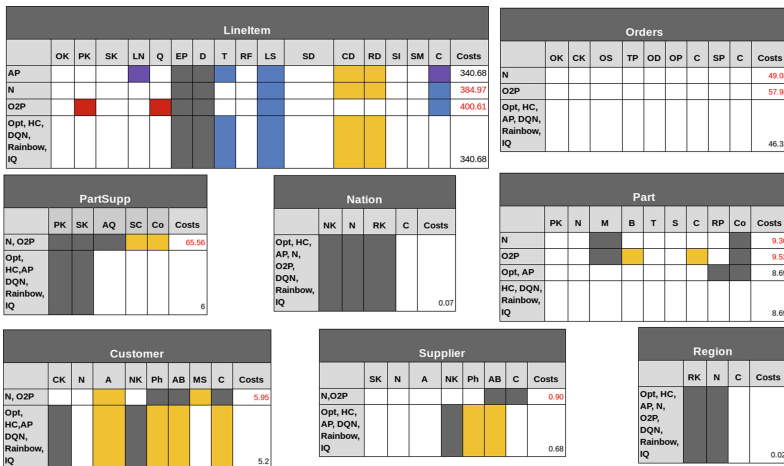


**Fig. 4.** Implicit quantile agent with update horizon 3 (multi-step of 3), training on CUSTOMER table, random workloads

Hence, scoped stochastic workloads might be better to evaluate generalization. Figure 4 also shows cases where the agent outperforms HillClimb (HC). This is understandable as HC does not guarantee the optima.

### 4.3 Inference

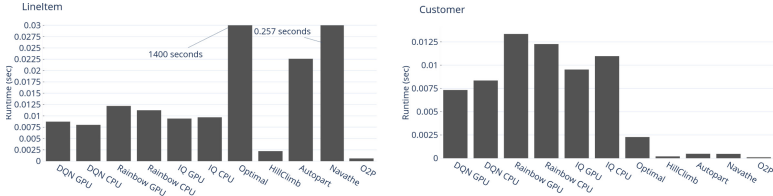
**Generalizing to Random Workloads:** We evaluate the learning process with a fixed table (CUSTOMER), and a series of random workloads. Figure 4 portrays the results of an Implicit Quantile agent. At 4k iterations there’s only convergence to 80% of expertise. To master such challenging scenario, extra work is required. In practice, however, workloads are not entirely random.



**Fig. 5.** Resulting partitions (in same color, except when white) and costs obtained, for TPC-H tables. Agents predict the cost optimal partitioning. Algorithms: AutoPart (AP), Brute force (Opt), HillClimb (HC), Navathe (N), O2P. (Color figure online)

Figure 5, presents the exact costs and partitions of alternative approaches. DRL, trained on the 8 TPC-H tables-workloads, reach for this case the exact costs of the optimal (from brute force). Concerning the time the agents require, after training, to make predictions (and change state based on actions), Fig. 6 shows average runtimes across 100 repetitions for the given tables. Out-of-the-box inference is able to outperform brute-force approaches for tables with increasing number of attributes. This inference also remains in the same order of magnitude,

being competitive with solutions like HillClimb, as the number of attributes in a table increases. Vanilla DQN provides a faster inference, as expected, given its simpler end layers. Still, more studies are required, on a tuned inference process.



**Fig. 6.** Inference time of the agents, compared with partitioning algorithms.

## 5 Conclusion

In this paper we present a novel DRL solution for vertical partitioning, and early evaluations using a cost model and a TPC-H database & workload. We report that the partitioning for individual table-workload pairs or for a small set of pairs is simple to learn. However, generalizing to entirely random workloads given a table, or to random tables, remains challenging, requiring further work. In terms of inference, we validate that our agents are able to learn the same behavior of algorithmic approaches. We show that on cases (when training for generalizing) it is possible to spot instances where our approach outperforms algorithmic solutions. We find that the brute force algorithm can be outperformed (in optimization time) by our off-the-shelf solution, and that our solution is competitive with algorithms, remaining in the same order of magnitude, and becoming more competitive as the number of attributes increases. Future work will consider more tuned model serving procedures (inference and observation space transitions) and training with a database. Further aspects relevant to the production-readiness of our solution should be studied in future work: more challenging workloads, extensions to observation and action (bottom-up and top-down actions) spaces, model improvements (specialized architectures and designs enabling the model to scale to a larger number of columns), different forms of partitioning (horizontal, hybrid), learning from demonstrations and, finally, interface design such that DRL solutions can be offered as useful plugins to data management systems.

## References

1. Agrawal, S., Narasayya, V., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: SIGMOD, pp. 359–370. ACM (2004)
2. Bellemare, M.G., Dabney, W., Munos, R.: A distributional perspective on reinforcement learning. In: ICML, pp. 449–458 (2017)



3. Durand, G.C., Pinnecke, M., Piriye, R., et al.: GridFormation: towards self-driven online data partitioning using reinforcement learning. In: AIDM@SIGMOD, p. 1. ACM (2018)
4. Castro, P.S., Moitra, S., Gelada, C., et al.: Dopamine: a research framework for deep reinforcement learning (2018). arXiv preprint [arXiv:1812.06110](https://arxiv.org/abs/1812.06110)
5. Dabney, W., Ostrovski, G., Silver, D., et al.: Implicit quantile networks for distributional reinforcement learning (2018). arXiv preprint [arXiv:1806.06923](https://arxiv.org/abs/1806.06923)
6. Hessel, M., Modayil, J., Van Hasselt, H., et al.: Rainbow: combining improvements in deep reinforcement learning. In: AAAI (2018)
7. Hilprecht, B., Binnig, C., Röhm, U.: Towards learning a partitioning advisor with deep reinforcement learning. In: AIDM@SIGMOD (2019)
8. Jindal, A., Palatinus, E., Pavlov, V., et al.: A comparison of knives for bread slicing. Proc. VLDB Endow. **6**(6), 361–372 (2013)
9. Krishnan, S., Yang, Z., Goldberg, K., et al.: Learning to optimize join queries with deep reinforcement learning (2018). arXiv preprint [arXiv:1808.03196](https://arxiv.org/abs/1808.03196)
10. Marcus, R., Papaemmanouil, O.: Deep reinforcement learning for join order enumeration. In: AIDM@SIGMOD, p. 3. ACM (2018)
11. Marcus, R., Papaemmanouil, O.: Towards a hands-free query optimizer through deep learning (2018). arXiv preprint [arXiv:1809.10212](https://arxiv.org/abs/1809.10212)
12. Sharma, A., Schuhknecht, F.M., Dittrich, J.: The case for automatic database administration using deep reinforcement learning (2018). arXiv preprint [arXiv:1801.05643](https://arxiv.org/abs/1801.05643)