Otto-von-Guericke-Universität Magdeburg

Faculty of Computer Science

Master's Thesis

Self-Driving Vertical Partitioning with Deep Reinforcement Learning

Author:

Rufat Piriyev

February, 8th, 2019

Advisors:

M.Sc. Gabriel Campero Durand

Data and Knowledge Engineering Group

Prof. Dr. rer. nat. habil. Gunter Saake

Data and Knowledge Engineering Group

Piriyev, Rufat:
Self-Driving Vertical Partitioning with Deep Reinforcement Learning
Master's Thesis, Otto-von-Guericke-Universität Magdeburg, 2019.

Abstract

Selecting the right vertical partitioning scheme is one of the core database optimization problems. It is especially significant because when the workload matches the partitioning, the processing is able to skip unnecessary data, and hence it can be faster [Bel09]. During the last four decades numerous different algorithms have been proposed to reach efficiently the optimal vertical partitioning. Mainly these algorithms rely on two aspects. On one side, cost models, which approximate how well a workload might be supported with a given configuration. On the other side, they rely on pruning heuristics that are able to narrow down the combinatorial search space, trading optimality for lesser runtime complexity [GCNG16]. Nowadays modified and improved variations of these methods can be expected to be included in commercial DBMSs [ANY04b, JQRD11] (although precise information on the solutions used by commercial systems are kept confidential [BAA12a]). Nonetheless, though these methods are efficient, they do not employ any machine learning technique, and thus are unable to improve their performance based on previous executions, nor are they able to be learn from cost-model errors.

Deep reinforcement learning (DRL) methods could address these limitations, while still providing an optimal and timely solution. In our research we aim to evaluate the feasibility of using DRL methods for self-driving vertical partitioning. To this end we propose a novel design that maps the process from the current domain to the domain of DRL. Furthermore, we validate our design experimentally by studying the training and the inference process for a given TPC-H data and workload, on a prototype we implement using Open AI Gym, and the Google Dopamine framework for DRL.

We train 3 different DQN agents for bottom-up partitioning with 3 selected cases: a fixed workload and table, a set of fixed workload and tables, and finally a fixed table with a random workload. We find that convergence is easily achievable for the first two scenarios, but that generalizing to random workloads requires further work. In addition, we report the impact of hyperparameters in the convergence.

Regarding inference we compare the predictions of our agents with that of state-of-the-art algorithms like HillClimb, AutoPart and O2P [JPPD], finding that our agents have indeed learned the optima during the training carried out. We also report competitive runtimes for our agents on both GPU and CPU inference, which are able to outperform some state of the art algorithms, and the brute-force algorithm as the table size increases.

Acknowledgements

First of all, I would like to thank my supervisor M.Sc. Gabriel Campero Durand for introducing me to this challenging topic. Without his permanent guidance, patience and constant encouragement this work could not have been possible.

I would like to thank Prof. Dr. rer. nat. habil. Gunter Saake for giving me the opportunity to write my Master's thesis with the DBSE research group.

I would like to thank M.Sc. Marcus Pinnecke for assisting us in finding a computer with GPU support for our long-running experiments.

I would like to thank M.Sc. David Broneske, M.Sc. Sabine Wehnert, Mahmoud Mohsen and others for their constructive feedback which I found very useful during my work.

Special thanks to the team of SAP UCC Magdeburg and Dr. Pape Co. Consulting GmbH for allowing me to take a break during the last phase of this thesis.

Finally, I would like to thank my family, relatives and friends for their continuous support during my studies.

Declaration of Academic Integrity

I hereby declare that this thesis is solely my own work and I have cited all external sources used.

Magdeburg, February 8th 2019

Rufat Piriyev

Contents

List of Figures

1. Introduction

In this chapter, we provide the motivation for our work. We start by introducing the three main approaches (manual, partially or fully automated) to selecting design configurations, presenting the vertical partitioning task as an optimization problem and the idea of applying deep reinforcement learning for solving this problem in a fully automated manner. We also establish the assumptions of our study, providing the scope for our work (Section 1.1). Next, we outline the contributions of this thesis (Section 1.2), followed by a description of the methodology that we adopted (Section 1.3). We conclude this chapter by presenting the structure for the subsequent chapters (Section 1.4).

1.1 Motivation

With the digitization of all aspects of life, both companies and end-users require efficient and scalable data management tools that could assist them in understanding their latest information either through traditional SQL querying, or through more complex kinds of analysis.

One essential component for the efficiency of these tools is their physical design (i.e., how logical database models are actually mapped to be stored and represented in memory). On live systems, database administrators and developers must perform several physical design tasks. These tasks are concerned with selecting physical configurations of a database on the storage system, with the goal of making a selection that benefits the expected workload (i.e., that helps to improve the memory footprint, runtime of operations, or maintainability). This includes tasks like: index selection, data partitioning, materialized views definition, data layout selection, among others.

For effective design and tuning it is required to understand database workloads. [KS86]. If a database's physical design is not being properly maintained (i.e, when the physical design does not matches adequately the workload), degradation of performance can occur, leading to less efficient data management systems. On a practical side, this implies that when considering rapidly changing workloads, database administrators and developers face several challenges, such as: evaluating at great speed a high number of possible configurations to determine the best, dealing with possible correlation between configurable parameters[VAPGZ17], and finally facing to the uncertainty of predicting the impact of configurations by using cost models and assumptions that might not fully match the real-world database system (i.e., as happens for other database choices, like join order optimization with mistaken cardinality estimates[Loh14]).

To alleviate these challenges research has proposed two approaches that extend the *purely manual* configuration management (i.e., the case where database administrators and developers are not supported for configuring the system). The first approach is the use of *partially automated* tools, such as database tuning advisors, which recommend database administrators certain configurations based on an expected workload[ABCN06]. This is usually done in a batch-wise fashion. The second approach is the use of *fully automated* solutions. From this last group solutions can either follow a *heuristic-driven* approach, which means that they do not use machine learning models and are based on domain-specific knowledge; or they can adopt a *self-driving* approach, which proposes to let the database itself deal with configuration choices automatically[PAA$^+$17], in an online manner, by using machine learning models and letting the optimizers learn from their mistakes, without involving the database administrators.

Tools produced by the first approach have been shown to create difficulties for database administrators, such as inaccurate estimates or failures in modeling update costs, creating unpredictable results [BAA12b].

The second approach, and specifically the self-driving design, though still in active development, seems specially promising as it could speed the time-to-deployment of configuration changes and, through the use of continuous learning, it could address failures of partially automated solutions.

In this thesis we research on the task of developing a self-driving tool to aid in a specific physical design case which, to date, has mostly been studied following a batch-wise, either heuristic-driven or partially automated approach: vertical partitioning. This is the problem of finding the optimal combination of column groups (i.e., vertical partitions), for an expected database workload. This choice is highly relevant, since it can achieve to optimize the amount of data that needs to be considered by a workload, helping to avoid the loading of unnecessary column groups.

Vertical partitioning is not a new optimization problem, and researchers have offered many different algorithms and solutions during the last forty years, like one of the first works by Hammer and Niamir[HN79a]. At its core, this optimization problem consists of evaluating with a cost model the combinatorial space of possible column groupings and determining which solution is estimated to be the best.

One of the biggest problems that algorithmic solutions face here is to find the right trade-off between the robustness of the solution (i.e., optimality) and optimization complexity[JPPD]. There are many popular non-machine learning based approaches for choosing the optimal vertical partition in a database relation, by using different heuristics that prune the search space of possible solutions that are evaluated.

For our study, following the goal of adopting a machine learning solution to this problem, we propose to study the applicability of a reinforcement learning (RL) model for mapping a traditional vertical partitioning solution to a self-driving solution. RL is one of the three main types of machine learning (i.e., apart from supervised and unsupervised learning), where models are trained using a system of rewards, obtained through interactions with an environment. Here RL is intended to aid in helping the model to navigate efficiently a search space, by learning the long-term value of a given action in a state; such that after a model is trained, online predictions can be made without exploring the complete search space. In specific, we decompose the recommendation for a complete partitioning scheme, into the recommendation of piece-wise actions (in a discrete action space) that when applied, should result in increasing improvements in the partitioning scheme and, by the end of our recommendations, in an optimal partitioning.

However, there are challenges in adopting a traditional RL solution: On one hand, the challenge of learning over a large action-state space (as the number of columns increases), which, in the absence of a function approximation, would require very large storage and numerous training episodes to explore such vast search space. On the other hand, there is also the challenge of feature selection (i.e., choosing the most informative features from the workload, device, database system and table description, that would help the self-driving component to learn from real-world observations, instead of from a synthetic cost-model alone).

For this reason we select for our study a specific approach to RL: deep reinforcement learning (DRL); an approach in which neural networks perform function approximation, bounding the storage requirements of the model, and aiding in the generalization of experience to unvisited states[FLHI$^+$18].

With our research we aim to establish whether building a DRL vertical partitioning solution is feasible and convergence during training can be achieved. For accomplishing this we also seek to design (and validate such design) solutions for aspects such as: how to accomplish normalization of rewards across different states (i.e., tables with different characteristics)? Finally, by developing a prototype, we also aim to study how competitive, in terms of the optimality of the solutions and of optimization time, is our proposed DRL approach over traditional algorithms.

In order to carry out our research we make several assumptions:

- We focus solely on a DRL solution for self-driving vertical partitioning. We do not consider neither in our study, nor in our discussion, the possibility of using alternative models (e.g. other RL approaches, genetic algorithms, frequent pattern mining, or clustering-based solutions).

- We limit our implementation and design to only use a synthetic cost model, specifically an HDD cost model (i.e., instead of a real world signal, like the execution time for a set of queries). Our use of a cost model for training has some implications that limit the generality of our current study:

  - We only evaluate using this cost model, and meta-data about the TPC-H tables and workload, instead of actually evaluating our recommendations on a database system. By removing this assumption it is possible that some changes could occur to our formulation (e.g. training might have to be done over logs), and convergence might require more training effort (i.e., since real-world signal can be noisy and influenced by external variables, when compared to synthetic cost models).
  - We limit our features to a small set that includes, for table description, attribute sizes and number of rows, and for workload description, only a group of queries which are simplified to be represented as a set of flags indicating for each query whether an attribute was chosen or not. This selection makes our current solution comparable (in terms of features used) to traditional vertical partitioning algorithms [JPPD]. When moving beyond a cost model, it can be possible to expand the feature set (e.g. adding the selectivity of queries) without too many changes to our proposed DRL model.
  - In the current implementation we do not consider costs or penalties for performing actions. It seems likely that this can be added to our formulation in future work, without affecting the main design choices.

- In the same way that most traditional vertical partitioning algorithms, we assume a fixed workload, deterministic state transitions and complete state observability. Changes in these assumptions could be added (should they be relevant for a given partitioning use case), as extensions to our model, in future work.

- We evaluate our design only on state-of-the-art DQN agents, without considering alternative (e.g. model-based solutions) or more specialized DRL agent designs (e.g. the Wolpertinger architecture for learning over large discrete action spaces[DAEvH$^+$15]). In addition we do not consider further solutions that could include hierarchical task design, or multi-agent scenarios.

- We model our solution to encompass only bottom-up vertical partitioning (i.e., merging of attributes, assuming that the system starts entirely in a column-only way).

- Due to time and resource constraints, we only evaluate the impact a very limited set of hyper-parameters, over default configurations of the selected reinforcement learning framework. Similarly, we reuse available neural network designs which were originally tailored to learning from pixels in arcade games with the Arcade Learning Environments[CMG$^+$18, BNVB13].

These assumptions provide the scope for our research, and outline aspects that can be considered in future work, building on our findings.

1.2 Our contributions

Our contributions can be listed as follows:

- We propose a novel design for action space (with only actions that allow merging of fragments, which amounts to a bottom-up approach), observation space, and rewarding scheme, that facilitates the task transfer from traditional vertical partitioning algorithms to a reinforcement learning formulation. In addition, we provide a concept for how the knowledge of an expert can be included in the rewarding scheme, leading to a normalization of the rewards across different cases.

- We offer a prototypical implementation of our solution using Open AI Gym and the Google Dopamine framework for reinforcement learning, employing 3 agents based on variations over DQN. Using this implementation we provide early results regarding the ease of the agents to converge to a solution during training, for vertical partitioning on TPC-H tables and workloads.

- We demonstrate empirically the ability of our agents to predict the optimal partitioning, once trained, for the selected cases.

- We report the inference time taken once the agent is trained, using CPU and GPU executions, compared to state-of-the-art implementations of vertical partitioning algorithms. We evaluate how this performance changes with regards to the size of the table.

- We provide first results regarding the ability of the agents to be trained as general agents, for all possible workloads over a given table.

Hence we scope our work to the design of an environment, the evaluation for training and inference for a given set of tables, either considering a fixed workload per table, or given a table, considering varying workloads; and a comparison with existing algorithms in terms of resulting partitions and optimization time.

1.3 Research Methodology

We follow the CRISP-DM [Wir00] process model as our research methodology. This method is mainly applied for data mining projects and has already become standardized. Though our project is not strictly a data mining project, we decided that due to the generality of CRISP-DM, it could be easily adopted to guide this study.

Below we detail the phases of CRISP-DM (Figure 1.1), as we adopt them for our research.

[1]https://en.wikipedia.org/wiki/Cross-industry_standard_process_for_data_mining

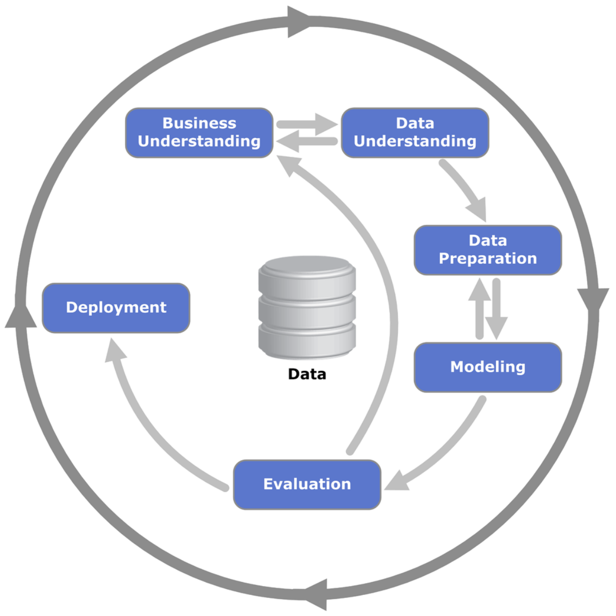


Figure 1.1: CRISP-DM Process Diagram
1

- **Business understanding**
  In this phase we need to answer to the following question: What are the current state-of-the art algorithms for database vertical partitioning, and how could we categorize them? In addition we study reinforcement learning (Chapter 2).

- **Data understanding**
  Here we study the TPC-H benchmark, the tables and queries, and the hyper-parameters available for our models (Chapter 4).

- **Data preparation Modeling**
  In this phase we develop the detailed design and concept for our solution (Chapter 3). Since this is closely tied to our choice of baseline and implementation with Google Dopamine, we include in this phase some study of such framework.

- **Evaluation**
  In this phase we concern ourselves with comparing the performance of the discussed DRL agents, with each other, as well as against adapted versions of traditional vertical partitioning algorithms (Chapter 5).

1.4 Thesis structure

The rest of the thesis is structured as follows:

- In Chapter 2 we give document some necessary background information. Mainly, we discuss traditional approaches to database vertical partitioning, and we establish the basic necessary RL and DRL concepts.
- In Chapter 3 we present the detailed design of our solution, describing the observation space, the action space and semantics of actions and the rewarding scheme we designed. Since the design is closely related to some implementation aspects, in this chapter we also introduce the Google Dopamine framework, and our choices in selecting vertical partitioning algorithms as baselines.
- In Chapter 4 we introduce the precise research questions that will be answered by our evaluation. We also describe the benchmarking data selected, the experimental setup for our study and some relevant implementation details.
- Chapter 5 is dedicated to our experiments using different TPC-H tables, workloads and hyper parameters. In this chapter we report and discuss our experimental results, answering to our research questions.
- In Chapter 6 we discuss related work in applying reinforcement learning methods for different kind of database problems, providing a better context to understand our research results and the outlook of our work. For keeping a cohesive presentation of our work, we place the majority of our studies on database self-driving tasks in this section.
- In Chapter 7 we conclude this thesis and give the further directions for future research.

2. Background

In this chapter, we present a brief overview of the theoretical background and state of the art relevant to this thesis. We organize this chapter as follows:

- **Horizontal and vertical partitioning:**
  We start by describing the general goal and workings of partitioning approaches (Section 2.1).
- **Vertical partition optimization algorithms:**
  We follow by describing in detail four chosen vertical partitioning algorithms. This is important since these algorithms are the state of the art, and they represent the baseline for evaluating our DRL-based solutions (Section 2.2).
- **Reinforcement Learning:**
  Next, we discuss the fundamental ideas behind reinforcement learning and deep reinforcement learning methods, as they are relevant to our research (Section 2.3).
- **Summary**
  We conclude this chapter by summarizing our studies (Section 2.4).

2.1 Horizontal and vertical partitioning

One central problem in database physical performance optimization is that of dividing an existing logical relation into optimally-defined physical partitions. 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.

There are several partitioning strategies, but the two basic ones for relational data are horizontal and vertical partitioning [NCWD84a]. Horizontal partitioning is the process of dividing a relation into a set of tuples, called fragments, where each fragment has the same attributes as the original relation [NCWD84a]. Here each tuple must belong to at least one of the fragments defined, such that we are able to reconstruct the original relation after its partitioning [KS86].

As a basic illustration, a generic employee relation (as shown in Figure 2.1) could be splitted into two fragments based on the *id* attribute, as follows:

$$\begin{aligned} employee_1 &= \sigma_{id<4}(employee) \\ employee_2 &= \sigma_{id>4}(employee) \end{aligned}$$

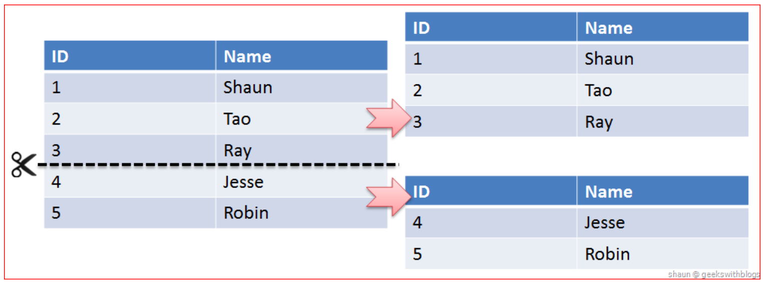
These fragments could be disjoint or not, with the latter case occurring when one particular tuple is simultaneously assigned to several fragments. Generally, for horizontal fragmentation we employ a selection predicate for a global relation r. This is sufficiently expressive to encompass range-based partitioning (i.e., where a range of values from an attribute is used for partitioning) or set-based partitioning (i.e., where sets of individual values are used for partitioning).

The reconstruction of the original relation r is done by simply computing the union of fragments:

$$r = r_1 \cup r_2 \cup ... \cup r_n \tag{2.1}$$

Horizontal partitioning can be used for distributed database scenarios, when data is accessed from geographically separated places, and it makes sense to provide the user with data closer to him/her location, to avoid overheads from network latencies. Such practice is also done by default with hash-based row-group sharding in NoSQL systems, like Amazon Dynamo[DHJ+07]. In these cases replication is also done, for improving availability, and the consistency between replicas needs to be managed with consensus protocols.

The problem of finding the optimal horizontal partitioning has been shown to be NP-complete [SW85].



ID	Name
1	Shaun
2	Tao
3	Ray
4	Jesse
5	Robin

ID	Name
1	Shaun
2	Tao
3	Ray

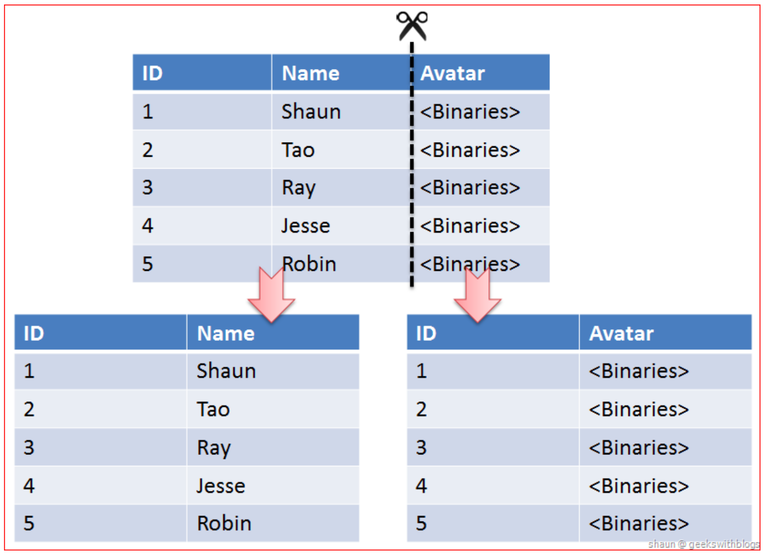
ID	Name
4	Jesse
5	Robin

Figure 2.1: Example of horizontal partitioning
1

[1]https://blog.marksylee.com/2017/01/28/structures-n-architectures-for-optimizing-database-en/

In comparison to horizontal partitioning, vertical partitioning is the process of dividing the attributes of a logical relation into several physical tables, called fragments too. Generally, at least one attribute (mainly primary key attributes) needs to be assigned to several fragments, in order to reconstruct the original relation. As a basic illustration, assuming that our *employee* relation (as shown in Figure 2.2) has three attributes namely *id*, *name* and *avatar* (which stores an image in a binary form), then this relation could be partitioned as two different fragments:

$$P_1 : (id, name)$$
$$P_2 : (id, avatar)$$



ID	Name	Avatar
1	Shaun	<Binaries>
2	Tao	<Binaries>
3	Ray	<Binaries>
4	Jesse	<Binaries>
5	Robin	<Binaries>

ID	Name
1	Shaun
2	Tao
3	Ray
4	Jesse
5	Robin

ID	Avatar
1	<Binaries>
2	<Binaries>
3	<Binaries>
4	<Binaries>
5	<Binaries>

Figure 2.2: Example of vertical partitioning [2]

Choosing the right vertical partitioning for a workload can improve query performance and also have positive impact on other database physical design decisions like indexing[JPPD]. Hence it is very important to select the optimal partitions for a relation.

2.2 Vertical partitioning algorithms

As discussed in (Section 2.1) one way to reduce I/O related costs and make queries run faster is to apply an efficient vertical partitioning. The problem of finding, with an algorithmic approach, such partitioning is not new and during the last few decades there have been numerous algorithms and solutions offered, (e.g. [NCWD84b, HN79b, CY90, JD12, PA04b]).

Mainly, finding the optimal vertical partitions of a given relation is an NP-complete problem, according to Agrawal et al.[ANY04a], which is unfeasible to solve with the

[2]https://blog.marksylee.com/2017/01/28/structures-n-architectures-for-optimizing-database-en/

optimal solution in a sub-exponential time, therefore is difficult to manage in case of real scenarios, without heuristics to reduce the search space[GCNG16, Ape88].

Therefore to find the best possible solution there have been algorithms proposed using different heuristics. One of the main problem here is that these algorithms mainly are not universal solutions, and the selection of the proper algorithm must be done in consideration of different characteristics of the database like the database/query processing paradigm, the hardware used, the workload specifications, among others.

The proposed vertical partitioning algorithms in the literature could be classified from different dimensions based the ideas behind them. Jindal et al. [JPPD] offered a three-dimensional classification:

- Search strategy
- Starting point
- Candidate Pruning

The first of these dimensions, *search strategy*, refers to how algorithms search the solution space. There are three main approaches: brute force, top-down and bottom-up:

- **Brute force:**
  This constitutes the naive approach towards solving the optimization problem. Here the algorithm checks all possible partitions and selects the one which guarantees the best query performance. The total number of possible different combinations follows the *Bell numbers*. A bell number for a number of columns n can be computed recursively in many ways. For example, we can do so in the following way:

  $$B_n = \sum_{k=0}^{n-1} \tbinom{n-1}{k} B_k \qquad (2.2)$$

  Therefore, despite the fact that a brute force approach guarantees that the global optimal would be found, in practise it is unfeasible to use without any space-pruning heuristics, due to the huge number of different combinations. In case of the TPC-H LineItem table, the total number of different vertical partitions is already 10.5 million. [JPPD]

  Other type of algorithms use several heuristics to reduce the solution-space following mainly two approaches, as we discuss next:

- **Top-down:**
  Algorithms based on this approach start with the full relation and try to break it into smaller partitions on each iteration. The main assumption here is that in each step there will be improvements in the cost given for an expected query workload on the partition currently obtained, based on a predefined cost model.

Algorithms stop the partitioning process when there are no longer improvements in subsequent iterations. The earliest vertical partitioning algorithms followed this approach. [NCWD84a, NCWD84b] The ideas of this approach is also used for recent algorithms proposed for an online environment [JD12, JQRD11]. Algorithms with this approach are more suitable when the queries in the workload access mainly the same attributes, since this might mean that the algorithm starts from a solution that is close to the optima.

- **Bottom-up:**
  In comparison to the former approach, these type of algorithms start with individual attributes as partitions, and recursively merge them at each iteration. This process continues until there are no longer improvements in query costs. The majority of state-of-the art algorithms follow this approach [HP03, PA04a]. This type of algorithms are mainly suitable, and have advantages over top down solutions when the attributes show a highly-fragmented access pattern (i.e., this too might mean that the algorithm starts from a solution that is close to the optima).

The second dimension to classify the vertical partitioning algorithms, is the *starting point*. This refers to whether the algorithm carries out some data pre-processing that would reduce the attribute set or the workload, that is fed into the partitioning solution.

Considering this dimension, some algorithms (e.g., Navathe [NCWD84a], O2P [JD12], Brute Force) start from the whole workload (i.e., considering all queries and attributes at the start). Algorithms like AutoPart [PA04b] and HillClimb [HP03]) extract subsets of the attribute space, choosing as a starting point a configuration where the original attributes are sub-divided into groups using a k-way partitioner, and then letting the partitioner compute vertical partitioning in each group. All sub-partitions are next combined for generating the complete solution. Attribute subset solutions decrease the complexity of the partitioning problem but the generated final solution could remain stuck in local optimas. The third type of selection takes as starting point a subset of the query space (or workload). This approach is relatively recent, and is based on query similarities inside a workload.This approach is used in Trojan [JQRD11].

The third classification dimension is *candidate pruning*. According to [JQRD11], according to [JQRD11] all algorithms except Trojan, do not apply any candidate pruning technique (save for the iterative computation) and only Trojan used a so-called threshold-based tuning in order to reduce the search space.

In our experiments we evaluate brute force, next to the Navathe, O2P, AutoPart and HillClimb algorithms, as baselines to provide an evaluation context for our DRL models. Given this choice, we consider that it is important to review these algorithms in detail, highlighting commonalities and differences between them. To conclude this section we summarize some aspects about these algorithms. In the next section we discuss each of them (except brute force) in detail.

In Table 2.1 we classify the given algorithms based on the different dimensions discussed. In the following, Table 2.2, we report further details from the original implementations.

Namely, we highlight the original hardware cost model adopted, the kind of workload the algorithm presupposes, and whether the algorithm allows for attribute/column replication.

Parameters	**Category**	**Navathe**	**O2P**	**AutoPart**	**HillClimb**
Search strategy	Brute Force				
	Top-down	+	+		
	Bottom-up			+	+
Starting point	Whole workload	+	+		
	Attribute subset			+	+
	Query subset				
Candidate Pruning	No pruning	+	+	+	+
	Threshold-based				

Table 2.1: Classification of Vertical-partitioning algorithms, adapted from [JPPD], (Original characteristics of algorithms)

Parameters	**Values**	**Navathe**	**O2P**	**AutoPart**	**HillClimb**
Hardware	HARD DISK	+	+	+	+
	MAIN MEMORY				
Workload	OFFLINE	+		+	+
	ONLINE		+		
Replication	PARTIAL			+	
	FULL				
	NONE	+	+		+

Table 2.2: Settings for vertical-partitioning algorithms, adapted from [JPPD]

2.2.1 Navathe

One of the earliest top-down algorithm is the approximation-based approach offered by Navathe et al. [NCWD84a]. In the first step of algorithm this algorithm an attribute affinity (AA) matrix is constructed. Here the affinity among the attributes is defined as follows:

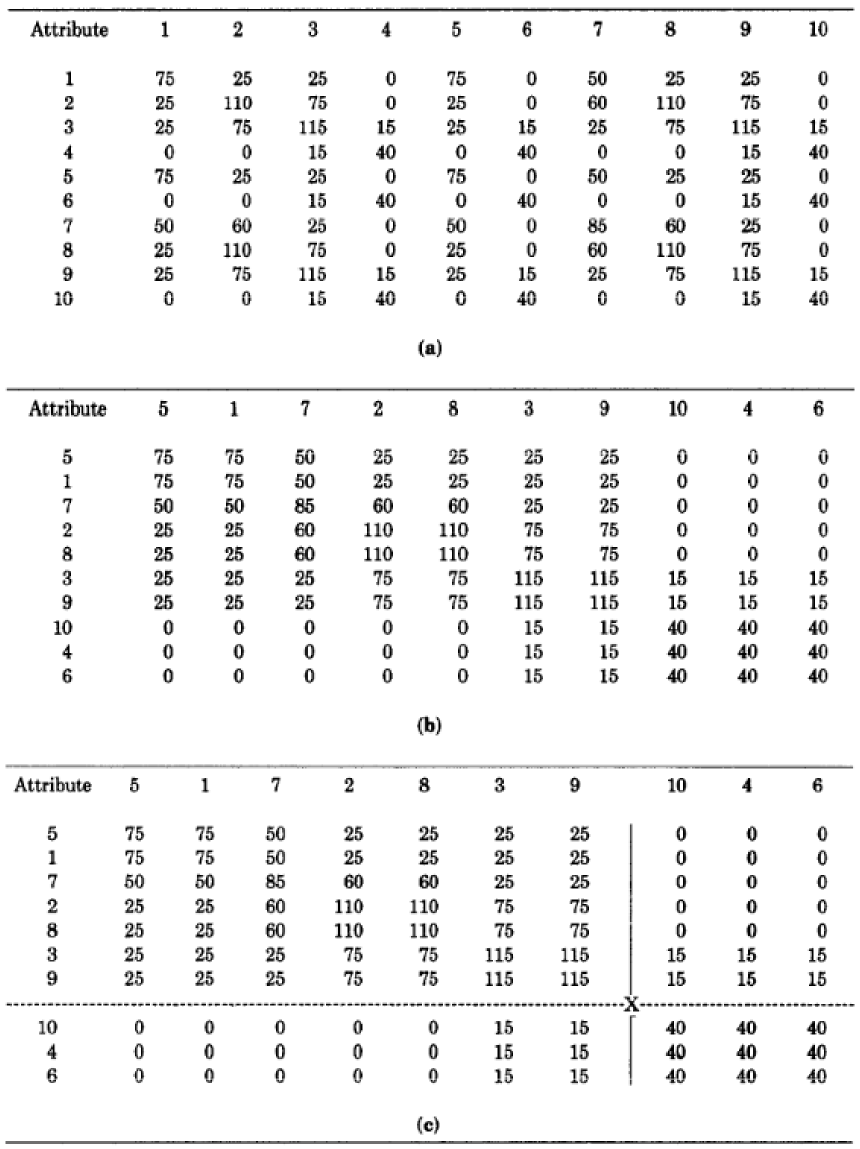
For each query k:

$$u_{ki} = \begin{cases} 1, & \text{if query k uses the attribute } a_i \\ 0, & \text{otherwise} \end{cases}$$

With u_{ki} - representing the affinity, as stored in a cell of the AA matrix.

In the second phase the generated AA matrix is clustered by permutating rows and columns, until obtaining a block diagonal matrix. For this purpose authors offered an algorithm called *CLUSTER* which employs a bond energy algorithm [72].

In the third phase, these blocks are recursively clustered, and the resulting partitioning is finally formulated. In Figure 2.3 some steps of the algorithm are illustrated.



Attribute	1	2	3	4	5	6	7	8	9	10
1	75	25	25	0	75	0	50	25	25	0
2	25	110	75	0	25	0	60	110	75	0
3	25	75	115	15	25	15	25	75	115	15
4	0	0	15	40	0	40	0	0	15	40
5	75	25	25	0	75	0	50	25	25	0
6	0	0	15	40	0	40	0	0	15	40
7	50	60	25	0	50	0	85	60	25	0
8	25	110	75	0	25	0	60	110	75	0
9	25	75	115	15	25	15	25	75	115	15
10	0	0	15	40	0	40	0	0	15	40

(a)

Attribute	5	1	7	2	8	3	9	10	4	6
5	75	75	50	25	25	25	25	0	0	0
1	75	75	50	25	25	25	25	0	0	0
7	50	50	85	60	60	25	25	0	0	0
2	25	25	60	110	110	75	75	0	0	0
8	25	25	60	110	110	75	75	0	0	0
3	25	25	25	75	75	115	115	15	15	15
9	25	25	25	75	75	115	115	15	15	15
10	0	0	0	0	0	15	15	40	40	40
4	0	0	0	0	0	15	15	40	40	40
6	0	0	0	0	0	15	15	40	40	40

(b)

Attribute	5	1	7	2	8	3	9	10	4	6
5	75	75	50	25	25	25	25	0	0	0
1	75	75	50	25	25	25	25	0	0	0
7	50	50	85	60	60	25	25	0	0	0
2	25	25	60	110	110	75	75	0	0	0
8	25	25	60	110	110	75	75	0	0	0
3	25	25	25	75	75	115	115	15	15	15
9	25	25	25	75	75	115	115	15	15	15
10	0	0	0	0	0	15	15	40	40	40
4	0	0	0	0	0	15	15	40	40	40
6	0	0	0	0	0	15	15	40	40	40

(c)

Figure 2.3: Initial attribute affinity matrix (AA); (b) attribute affinity matrix in semiblock diagonal form; (c) non-overlapping splitting of AA into two blocks L and U ([72]).

Despite, Navathe being one of the first approaches for vertical partitioning with the top-down scenario, it is still used as part of more advanced algorithms.

2.2.2 O2P

Recently another top down algorithm was offered by Jindal et.al [JD12], being mainly designed for an online environment. Authors suggest that the main disadvantages of the existing vertical partitioning algorithms is that they are mainly offline solutions. So there are problems when the workload is changing during the time. This is especially important if there is concept drift (or rapidly changing workloads). In such cases it becomes difficult for database administrator to adopt the recommendations from partitioning algorithms appropriately. The offered solution *AUTOSTORE* works in an *online* environment, dynamically monitoring query workloads and dynamically clustering the partitions using interval-based analysis over the affinity matrix. For analyzing the partitions, the authors offered an online algorithm O2P which employs a greedy solution. Therefore, despite the partitions generated by O2P being non-optimal ones, in practise they show decent and comparable results to other algorithms.

As described, O2P employs Navathe's algorithm and dynamically updates the affinity matrix for every incoming query. Dynamic programming is using for keeping the optimal split lines from previous steps.

This algorithm works by defining a partitioning unit, which corresponds to the smallest indivisible unit of storage (e.g. attribute groups that are always mentioned together on all queries). Next it defines an ordering inside each unit (i.e., the order of attributes in the attribute affinity matrix, to help the block-wise clustering). Based on this order a split vector S can be proposed, which defines with 0 if a given attribute is placed next to the former one, or with 1 if there is a split. O2P does one dimensional partitioning, because it considers only one split line at a time. Finally, partitions are then defined by split lines, partitioning units and the corresponding orders.

In 1 the pseudocode of O2P is shown. As mentioned, the call to O2P is recursive, taking as input the previous split vector (S), and information from previously evaluated split lines in the left and right groups. It also takes as input the best in all previous partitions (PrevPartions). As mentioned, the algorithm reuses or computes the information regarding the best split line found in all previous partition, and the minimum cost for the best split in the two groups along the current split vector (lines 1-6). The algorithm stops if one split line is invalid (lines 7-8). Next it evaluates the minimum cost obtained for the three lines (line 10), and chooses the one having the minimum cost, setting the next parameters such that computation can be reused (lines 11-35).

Algorithm 1 O2P algorithm- dynamicEnumerate

```
    Input S, left, right, PrevPartitions
    Output Enumerate over possible split vectors
 1: SplitLine sLeft = BestSplitLine(S,left);
 2: Cost minCostLeft = BestSplitLineCost(S,left);
 3: SplitLine sPrev = BestSplitLine(S,PrevPartitions);
 4: Cost minCostRight = BestSplitLineCost( S,right);
 5: SplitLine sPrev = BestSplitLine(S,PrevPartitions);
 6: Cost minCostPrev = BestSplitLineCost(S,PrevPartitions);
 7: if invalid(sLeft) and invalid(sRight) and invalid(sPrev) then
 8: return
 9: end if
10: Cost minCost = min(minCostLeft, minCostRight, minCostPrev);
11: if $minCost == minCostLeft$ then
12:     SetSplitLine(S, sLeft);
13:     if $sRight > 0$ then
14:         AddPartition(right, sRight, minCostRight);
15:     end if
16:     right = sLeft+1;
17: else if $minCost == minCostRight$ then
18:     SetSplitLine(S, sRight);
19:     if $sLeft > 0$ then
20:         AddPartition(left, sLeft, minCostLeft);
21:     end if
22:     left = right;
23:     right = sRight+1;
24: else
25:     SetSplitLine(S, sPrev);
26:     if $sRight > 0$ then
27:         AddPartition(left, sLeft, minCostLeft);
28:     end if
29:     if $sLeft > 0$ then
30:         AddPartition(right, sRight, minCostRight);
31:     end if
32:     RemovePartition(sPrev);
33:     left = pPrev.start();
34:     right = sPrev+1;
35: end if
36: O2P algorithm- dynamicEnumerate(S, left, right, PrevPartitions);
```

2.2.3 AutoPart

Papadomanolakis et al. offered a bottom-up algorithm called AutoPart [PA04b].

Authors offered two types of fragments:

1. Atomic - the thinnest possible fragment of a given relation, which are accessed atomically (i.e there is no query which access the subset of atomic fragment)
2. Composite - union of several atomic fragments

The general outline of algorithm is described in Figure 2.4.

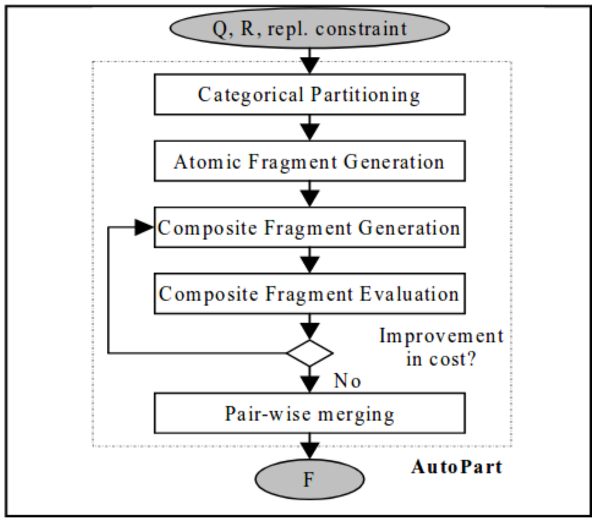
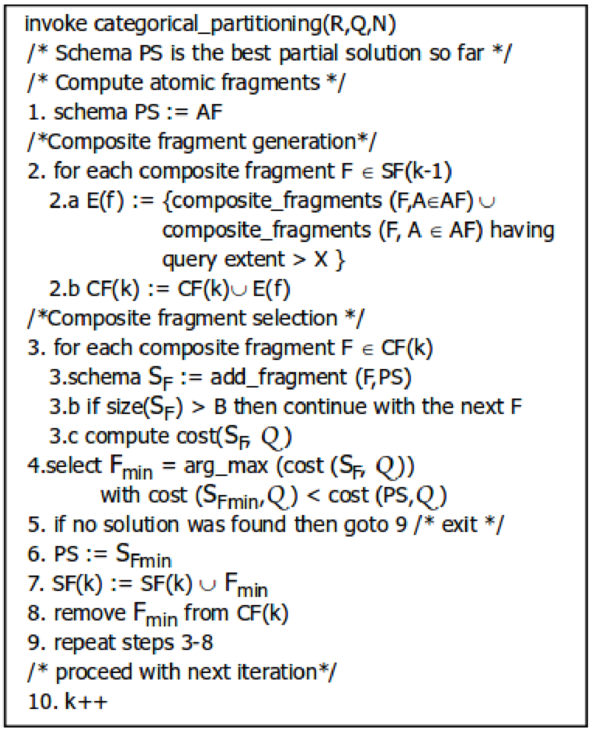


Figure 2.4: Outline of the partitioning algorithm used by AutoPart ([PA04a]).

As a first step the algorithm identifies predicates based on workload, to avoid “false sharing” between queries. This constitutes categorical partitioning. In the second step atomic fragments are generated (i.e, those that appear together in all queries), in the next step composite fragments are generated either by merging atomic fragments or combining those with composite fragments from the previous step. On each iteration an estimated cost is calculated and if there are no improvements, then the process of composite fragment generation finishes.

The pseudocode of the algorithm is shown in Figure 2.5. Here, atomic fragments are computed initially (line 1); they constitute the initial solution or partial schema, PS. Next, composite fragments are calculated (line 2), which are formed by combining pairs of the fragments selected in the previous step with the atomic fragments. These are fragments that need to have a support in queries that exceed a certain threshold (line 2a), which means that for fragments to be considered useful they should appear in a number X of queries. Following this a set of composite fragments can be considered for inclusion in the partial schema (line 3). First a selected fragment is considered (line 3a), which the union of the given composite fragment and the partial schema. There is a pruning condition that could be given (line 3b), and then the cost is simply computed (line 3c). Next from the set of fragments studied (lines 3a-3c), the one with the lowest cost is selected and the algorithm evaluates if there is an improvement on the cost (line 4). If there is an improvement, the union of this fragment with the current partial schema (PS) is selected as the new partial schema (line 6) and removed from the composite fragments (line 8). The process then repeats (lines 3-8) until no solution is found, or there are no composite fragments left.



```
invoke categorical_partitioning(R,Q,N)
/* Schema PS is the best partial solution so far */
/* Compute atomic fragments */
1. schema PS := AF
/*Composite fragment generation*/
2. for each composite fragment F ∈ SF(k-1)
   2.a E(f) := {composite_fragments (F,A∈AF) ∪
               composite_fragments (F, A ∈ AF) having
               query extent > X }
   2.b CF(k) := CF(k)∪ E(f)
/*Composite fragment selection */
3. for each composite fragment F ∈ CF(k)
   3.schema $S_F$ := add_fragment (F,PS)
   3.b if size($S_F$) > B then continue with the next F
   3.c compute cost($S_F$, $Q$)
4.select $F_{min}$ = arg_max (cost ($S_F$, $Q$))
        with cost ($S_{Fmin}$,$Q$) < cost (PS,$Q$)
5. if no solution was found then goto 9 /* exit */
6. PS := $S_{Fmin}$
7. SF(k) := SF(k) ∪ $F_{min}$
8. remove $F_{min}$ from CF(k)
9. repeat steps 3-8
/* proceed with next iteration*/
10. k++
```

Figure 2.5: The AutoPart algorithm ([PA04a]).

2.2.4 HillClimb

Hankins et al. proposed a straight-forward bottom-up partition algorithm which is simply a HillClimb [HP03] optimization algorithm. The steps of algorithm are as follows:

- Initialize the algorithm with a columnar layout, where each attribute is inside a different partition.
- In each iteration: Find and merge two of the current partitions which guarantee the best improvement (across all candidates in the iteration) in terms of a predefined cost function
- The process stops when the best improvement in the current iteration does not lead to an overall performance gain with respect to the previous iteration, or when there are no more columns to merge.

Here should be noted that on each iteration the overall number of partitions is decreasing by one.

The pseudocode for HillClimb is presented in 2.

Algorithm 2 Hill Climb

```
function HILL-CLIMB($Q$, $G$)

    ▷ Q is the set of all queries
    ▷ G is the set of all groups
    ▷ Compute the cost, table[i], of each group i
    for i = 1 to length(G) do
        for j = 1 to length(Q) do
            table[i] = model(Q[j],G[i])
        end for
    end for
    ▷ Compute the cost for each partition
    $Cand$ = {{1}, {2},···,{n}}
    $R = \emptyset$
    do
        $R = Cand$
        $mincost = candcost$
        $Cand = \emptyset$
        for i = 1 to length(R) do
            for j = i + 1 to length(R) do
                $s$ = {{$R_1$, ...  , $R_i$ $\bigcup$R$_j$, ...  }}
                $Cand$ = $Cand \bigcup s$
            end for
        end for
        ▷ Compute lowest cost partition
         $candcost$ = $min_{i=1...|Cand|}(cost(Cand_i))$
        $Cand$ = min$_=$1...|Cand|(Cand$_i$)
    while  $candcost$ <$mincost$
     return R
end function
```

With this we conclude our detailed explanation of state-of-the-art algorithms for vertical partitioning. These constitute the baseline, or heuristic-driven approach, to the self-driving approach we study in this work. In the next sections we introduce the area of reinforcement learning, and of deep reinforcement learning, as they constitute the theoretical framework of the solution we study.

2.3 Reinforcement learning

In this section we briefly discuss reinforcement learning (RL). Since, the topic is very broad we decide to focus only on the essential concepts relevant to our current research.

We structure this section as follows:

- We start by discussing general reinforcement learning, introducing the basic terminology (Section 2.3.1).
- Subsequently we discuss Deep Q-learning (DQN) based approaches, since they are a state-of-the art family of approaches in deep RL algorithms, and we use models based on these approaches for our study (Section 2.3.2).

2.3.1 RL basics

RL is a class of machine learning solution, where an agent learns the proper behavior (i.e, how to choose actions, given a state) for solve a given task through trial-and-error, by interacting with an environment and obtaining rewards. Formally speaking, the purpose of RL is to solve a Markov Decision Process (MDP), when the decision policy is unknown. The interaction in the MDP process is shown in Figure 2.6.

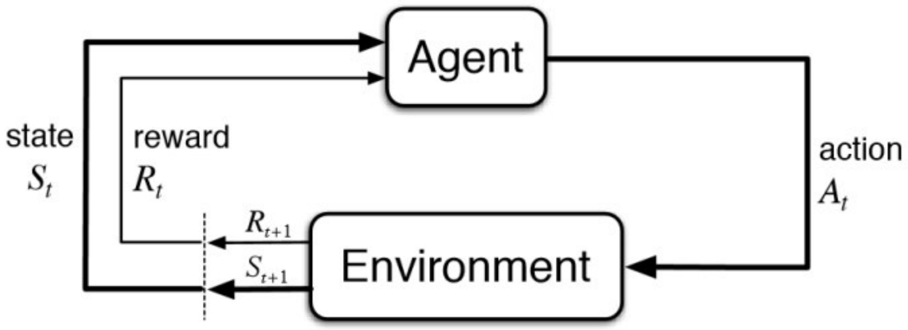


Figure 2.6: The Agent-Environment interaction in MDP [SB98]

Here an agent observes the state of an environment S_t. Based on this, the agent takes an action A_t from the predefined action set A based on a chosen strategy, and applies this action on the environment, changing the current state to a next one S_{t+1} and gaining in the process a so-called reward R_t.

Below we describe some basic terminology of RL:

- *Environment* - The system the agent interacts with. The general mathematical framework for defining an environment is a MDP. A MDP is a set of finite environment states S, a set of possible actions $A(s)$ in each state, a real valued reward function R(s) and a transition probability $P(s', s|a)$ from one particular state to another, given an action. Generally, the main aim of RL is to solve a MDP when the probabilities of rewards are unknown.

- *Episode* - A sequence of $< s_0, a_0, r_1, s_1 >$, $< s_1, a_1, r_2, s_2 >$,..., $< s_{n-1}$, a_{n-1}, r_n, $s_n>$ tuples (steps) from the start to the end. Episodes end with some terminal state.

- *State* - Current position of the agent in the environment. In addition to the state, Observations are how the state is represented to an agent.

- *Reward* - The feedback signal given by the environment for reaching a particular state given an action and a previous state.

- *Actions* - The set of signals from which an agent is able to select for transitioning the environment from one state to another.

- *Transition model* - $T(s_t, a_t, s_{t+1}) = p(s_{t+1}|s_t, a_t)$ - The probability of transitioning between states s_t and s_{t+1} in case of an action a_t.

- *Policy* - The function(strategy) that specifies the action which an agent will choose in a given state s. One of the core problems of RL is to find the optimal policy which will maximize the long-term cumulative reward.

- *Value* - The future reward that an agent expects from following a particular policy.

Some of the challenges in reinforcement learning include the so-called exploration vs. exploitation dilemma, which expresses during training whether the agent should explore actions not performed thus far, or exploit the current knowledge. Another challenge is the credit-assignment problem, wherewith it is difficult to determine for an agent which action is responsible for a given reward. Different models have been proposed in order to help the agents overcome these challenges.

2.3.1.1 Basic components of RL models

The agent's main target is to progressively collect more rewards in comparison to previous episodes, through a learning process. There are some basic components which RL models need to have to accomplish this task. Among them, a direct representation of a policy, a value function or a complete model of the environment couple with a planning algorithm. Approaches that do not include the last component are called model-free. Combinations of model-based and model-free RL also exist, but they, the same as model-based approaches, fall outside the scope of our review.

In next we detail the components from model-free RL, which are relevant to our study since we adopt these for our solution.

- **Policy** - Policies in an agent can be deterministic (Equation 2.3) or stochastic when an action is selected based on certain probability (Equation 2.4)

$$\pi(s_t) = a_t, s_t \subset S, a_t \subset A \tag{2.3}$$

$$\pi(a_t|s_t) = p_i, s_t \subset S, a_t \subset A, 0 \leq p_i < 1 \tag{2.4}$$

The main target of an RL agent is to learn an *optimal policy* π^* which a guarantees maximal reward for a particular environment while following all steps of this policy. In practice, an exploration strategy continues till convergence happens on "optimal" or "sub-optimal" policies.

- **Value function** - These are used to evaluate the usefulness of the particular policy π at the state $s_t \subset S$ and following the same policy. The usefulness of a policy is calculated as a sum of discounted rewards [SB98] (Equation 2.5)

$$V : V^\pi \to \mathbb{R}, V^\pi(S) = \mathbb{E}_\pi\{R_t|s_t = s\} = \mathbb{E}_\pi\{\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}|s_t = s\} \tag{2.5}$$

Value functions could be estimated by "trial-and-error" and calculated using dynamic programming approaches. They also have a recursive nature as described in the (Equation 2.6) [SB98].

$$V^\pi(S) = \mathbb{E}_\pi\{R_t|s_t = s\} = \mathbb{E}_\pi\{\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}|s_t = s\} = \mathbb{E}_\pi\{r_{t+1} + \sum_{i=0}^{\infty} \gamma^i r_{t+i+2}|s_t = s\} \tag{2.6}$$

Equation (Equation 2.6) unravels to the *Bellman equation* of V^π.

Value functions usually replace direct policy representations in implemented models, since value functions can be used to define a policy.

- **Quality function** - Some RL methods try obtain to find optimal policies empirically. For that reason they employ so-called Quality function. A quality function has a similar definition to a value function but they also take an action into consideration[SB98].

$$Q : SXA \to \mathbb{R}, Q^\pi(s, a) = \mathbb{E}_\pi\{R_t|s_t = s, a_t = a\} = \mathbb{E}_\pi\{\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}|s_t = s, a_t = a\} \tag{2.7}$$

In an optimal policy π^*, the $V^{\pi^*} = argmax_a \mathrm{Q}^\pi(s_t, a_t)$ when taking an optimal policy [SB98]

2.3.1.2 Q-learning and SARSA

The idea of temporal-difference learning (TD) [SB98] is based on dynamic programming (DP) and is related to Monte Carlo methods. In comparison to Monte Carlo methods where the entire episode needs to finished to able to be able to update the value function, TD-learning is able to learn value function within each step. In comparison to DP where the model of the environment's dynamic is necessary, TD is suitable for uncertain and unpredictable tasks. There are two possible algorithms offered for TD-learning tasks.

- off-policy **Q-learning algorithm** offered by Watkins et al [WD92]
- on-policy **SARSA** algorithm offered by Sutton et al [SB98]

For updating the action-value-function *on-policy* methods update the value given for the current action by considering the future actions based on the current policy, whereas *off-policy* methods use policies different that the current one for calculating the discounted reward (i.e., the next action) for a given action-value-function.

Q-learning is an off-policy algorithm because to approximate optimal Q-value function Q^* it does not consider the current policy. The update step is described in Equation 2.8.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{2.8}$$

Here α is a learning rate ($0<\alpha \leq 1$) that determines how fast the old Q-value will be updated based on new experiences. γ is a discount factor that is used to balance how much the value of future states will impact the value of the current state.

It has been proven that Q-learning converges to the optimal provided that the state-action pairs are represented discretely, and that during exploration all actions are repeatedly sampled in all states (which, as authors point out [FLHI$^+$18], ensures sufficient exploration)[WD92].

In comparison to Q-learning, in SARSA the Q-value is updated via the interaction with the environment and updating policy is depends from on the taken action. The update step for SARSA is almost the same as for Q-learning except it does not need to select the action which guarantees the maximum reward by following the current policy Equation 2.9.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{2.9}$$

2.3.1.3 ϵ-greedy and Boltzmann exploration strategies

There is a well-known exploration vs. exploitation dilemma in RL:
Should the agent exploit already learned knowledge about the environment (following what is currently known to be the most rewarding policy), or does the agent need

to explore more unknown states to be able to find a better policy? Obviously, at the beginning of its training the agent should explore larger amount of states, and then he could exploit the gathered knowledge till he is confident enough of the results. The problem here is finding the right trade-off for how to switch between exploration and exploitation, and this could be hard to achieve in the case of uncertain/dynamic environments.

There are several methods proposed for managing this challenge:

- ϵ - greedy strategy: In this series of strategies, given the Q-function $Q(s,a)$ an agent randomly selects a action based on predefined ϵ - probability ($0 \leq \epsilon < 1$) and selects the best action which maximizes the Q-value for a given state in (1- ϵ) cases. In most cases, it is better to select a high ϵ at the beginning of the task and decrease it gradually as the training increases (decrease exploration over exploitation). This is the idea of decaying ϵ - greedy approaches, where a variety of functions could be used to decrease ϵ over time.

- Boltzmann Exploration - One of the drawbacks of the ϵ-greedy approach is that it considers one action as the best for a given strategy and considers all others as equally probable bad actions. One of the major problems here, actions that would rank as second best are being treated equal to the worst possible actions. To be able to instead use the knowledge about every $Q(s_t, a_t)$ value, a Boltzmann distribution is used as in the formula (Equation 2.10) for getting the probability of a given action:

$$p(a_t|s_t, Q_t) = \frac{e^{Q(s_t,a_t)/\tau}}{\sum_{b \subset A_S} e^{Q(s_t,b)/\tau}} \tag{2.10}$$

  Here increasing the parameter τ leads for exploration, decreasing τ with the number of episodes make exploration more *greedy* and the probability of several promising actions is increased, such that the agent explores more.

2.3.2 Deep RL

Pure value-function methods discussed in Section 2.3 are suitable when the actions are mainly discrete and we have small state spaces. However these approaches become impractical in case of high-dimensional and continuous action spaces and for environments with huge state spaces, since it will be difficult to create and update such large Q-tables. Another problem is ineffective feature representation, which could lead to slowing down the learning process and delayed convergence.

Nowadays, deep Learning (DL) solutions are becoming very popular in different domains such as computer vision, speech recognition, machine translation, etc. where they are able to demonstrate nice results[DY$^+$14]. Deep neural network approaches are able to learn very complex features from raw input. Deep learning refers to the use of artificial neural networks for machine learning. Nowadays these models have become mainstream thanks to developments in specialized kinds of neurons (e.g. convolutional, recurrent, GANs), optimization algorithms (e.g. Adam, Nesterov) and the standardization in supporting technologies, including libraries such as Caffe, Keras or Tensorflow-slim.

DL approaches show promising results too when combined with RL, being used for *function approximation* (i.e., such that the neural network maps between a state to the Q-values predicted for the actions). The idea here is that it is overwhelming to store each state in memory, specially if these states are very similar (in computer games there could be only one single pixel difference between two sequential frames(states)). Therefore, instead of getting exact Q-values of a concrete state, we would involve deep neural networks to approximate Q-values (Equation 2.11). Here the parameters of the neural network (weights, biases, activation functions, etc., are represented as θ)

$$Q^*(s_t, a_t, \theta) \approx Q^*(s_t, a_t) \tag{2.11}$$

Figure 2.7 provides a taxonomic overview on recent classes of RL algorithms. As can be seen, the first large division concerns model-free or model-based approaches. As we have already mentioned, in our work we have decided to focus on model-free scenarios. From the model-free cases there are value-based methods (such as Q-learning, and it's deep variant, DQN) and policy optimization methods (like basic policy gradient, and some deep variants like PPO).

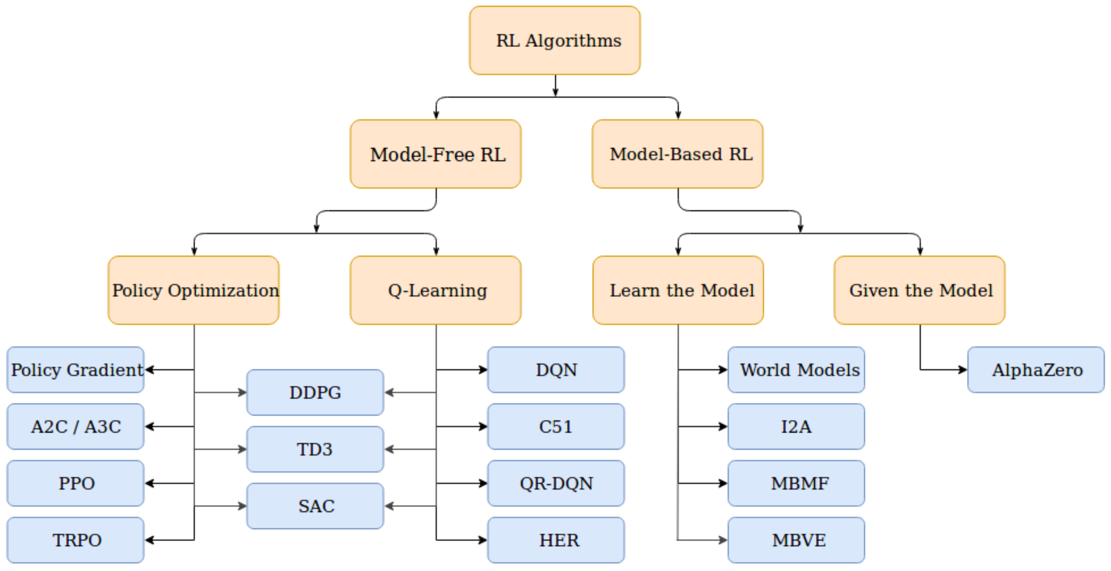


Figure 2.7: Taxonomy of recent RL algorithms[3]

[3]Source: https://spinningup.openai.com

In the further sections we provide a summarized overview of the DRL approach which we used for our models. In detail we review a subset of methods from the DQN class.

2.3.2.1 Deep Q-Network (DQN)

The deep Q-network (DQN), approach of applying deep reinforcement learning as offered by Mnih et al. [MKS+13],
[MKS+15] was able to successfully learn how to play various Atari Games.

In the original DQN algorithm, a neural network was added a substitute of Q-tables in Q-learning, and it as fed first pre-processed images from atari game emulators as an input. A convolutional neural network (CNN) was employed. In this case the agent had access only to a game's score, which it considered as a reward.

After defining the Q-network architecture, taking as input the state (e.g. pre-processed images) and giving as output the series of Q-value predictions for all possible actions given the state, we train it by minimizing a loss function (Section 2.3.2.1) $L_i(\theta_i)$ which changes at each iteration i.

$$L_i(\theta_i) = E_{(s_t,a_t,s_{t+1},r_{t+1})r\sim D}[(\underbrace{r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_i)}_{\text{target}} - \underbrace{Q(s_t, a_t; \theta_t)}_{\text{current}})]^2 \qquad (2.12)$$

For function approximation a naive DQN implementation includes some conditions that could cause the model to not converge (e.g. the high correlation during training in observations from the same episode). In order to deal with such conditions, DQN proposed some basic improvements.

DQN employs so-called *Experience Replay*, a mechanism to avoid correlations between data samples and provide data set for training purposes. Here $\mathbb{D}$ is the *Replay memory* and contains tuples from previous experiences $(s_t, a_t, s_{t+1}, r_{t+1})$. These tuples are sampled in a batch for the DQN training. This sample can be uniformly chosen (the default), or performed with some prioritized sampling.

DQN also incorporate a so-called *Fixed-Q target*. In the original algorithm (Section 2.3.2.1) when we calculate the *loss function* we calculate difference between TD-target (Q_{target}) and current estimated Q-value ($Q_{current}$). The problem here is that we use the same parameters (weights) to update both TD-Target and Q-Value. Therefore, there is a big correlation between target and changed weights. It means, at every training step target values are also changing with along Q-values. It makes convergence slower.

In case of, fixed Q-target we employ two separate networks (θ, θ^-). And update of weights from Q-value network to target network happens at every τ steps. The idea is that during training the current network is continuously update, while the target network (used to predict the Q-values for the next step) is not updating while training. Instead, at seldom intervals the weights of the current network are copied (i.e., synchronized) to the target network. This allows to “fix” the q-values of the next step during training, such that the q-values being updated do not affect those predictions, and the training converges better. This allows to have more stable learning due to target network stay fixed for a while. This is also shown in .

The pseudo code of the algorithm explaining how DQN is trained is shown in 3.

Algorithm 3 Deep Q-learning with Experience Replay [MKS+13]

```
1: Initialize replay memory $D$ to capacity $N$
2: Initialize action-value function $Q$ with random weights
3: for episode = 1, M do
4:     Initialize sequence $s_1 = x_1$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
5:     for episode = 1, T do
6:         With probability $\epsilon$ select a random action $a_t$
7:         otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
8:         Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
9:         Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
10:        Store transition $(\phi t, a_t, r_t, \phi_{t+1})$ in $D$
11:        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
12:        $Set y_j = \begin{cases} r_j, & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta), & \text{for non-terminal } \phi_{j+1} \end{cases}$
13:        Perform a gradient descent step on $(y_j - Q(\phi j, a_j; \theta))^2$
14:    end for
15: end for
```

2.3.2.2 Basic extensions to DQN

There have been several additional improvements offered as basic extensions to the original DQN algorithm.

- **Double DQN** - Hasselt et al. proposed Double DQN [HGS16] to handle overestimation of Q-values. Basically, the max operation uses the same value to select and evalaute an action. Because of this there is a bias to have overestimations. Double DQN proposed the use of two estimators such that the errors introduced are decoupled, reducing the positive bias.

- **Prioritized Experience replay (PER)** - Proposed by Schaul et al. [SQAS15] introduces the idea of improving the sampling from the experience replay by selecting more "important"/"useful" experiences from the replay buffer during training. In the original experience replay the sampling of the batch happens uniformly, which makes it hard to select "important" experiences. In the proposed PER solution we sample a batch by defining the priority for each tuple in the experience buffer, and then ranking them. Here the algorithm "prefers" experiences with bigger difference between the Q-value and TD-Target values.

More advanced extensions to DQN, affecting the neural network design have also been proposed. We discuss two of them, as they correspond to models we use in our solution.

2.3.2.3 Distributional RL

In their work Bellemare et al. proposed an approach called Distributional RL, which mainly learns to approximate the complete distribution rather than the approximate expectation of each Q-value. [BDM17] In RL we use the *Bellman equation* for expected value approximation. But in case of a stochastic environment choosing actions based on an expected value could be a reason for non-optimal solutions.
In disributional RL we directly work with the full distribution of returns. Here we define a random variable $Z(s, a)$ - starting from the state s, and performing action a for the current policy. In that case we could define value-function in terms of a Z-function as shown in Equation 2.13.

$$Q^{\pi}(s,a) = \mathbb{E}[Z^{\pi}(s,a)] \tag{2.13}$$

The Bellman equation can then be rewritten as a distributional Bellman, shown in Equation 2.14.

$$Z^{\pi}(s,a) \overset{\mathrm{D}}{=} R(x,a) + \gamma Z^{\pi}(x^{'},a^{'}) \tag{2.14}$$

Here x' $\approx$p$(\cdot|x,a)$ and a$^{'}$ $\approx\pi(\cdot|x')$ and Z is a value distribution.

In their approach authors used the *Wasserstein Metric* to describe the distance between probability distributions of Z(s, a) and $Z^{\pi}(x^{'},a^{'})$. Here value distribution could be represented in a different form but in the proposed **C51** algorithm authors used a categorical distribution. Authors proposed and proved the convergence of the distributional Bellman equation.

Building on this previous work Dabney et al. offered an implicit quantile network(IQN) [DOSM18], trained as a deterministic parametric function to reparameterize samples from a base distribution to the quantile values of a return distribution. In comparison to **C51** in Implicit quantile case the output is a single sample instead of a reward distribution per action. Here algorithm takes an input at 2 different stages, at the first stage IQN takes current state as a vector and transforms it into another vector with a fixed dimension (i.e., vector V). Then algorithm takes some random scalar value τ in the range of [0,1] and feeds that value into the function $\Phi(\tau)$. Here we get a vector H with the same dimension as V. V and H vectors are then combined (via vector multiplication or concatenation depending on the forward layers size). The result of the forward-pass is an $|A|$ dimensional vector which contains action-distributions.

In Figure 2.8 the network architectures of DQN and different variations of distributional RL algorithms is illustrated.

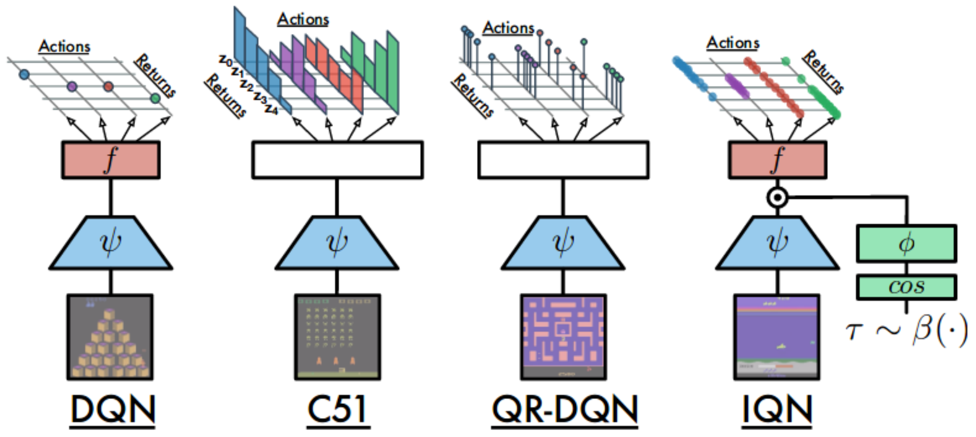


Figure 2.8: Network architectures of different Deep RL algorithms, derived from [DOSM18]

2.3.2.4 Rainbow

In their paper [HMvH$^+$17] Hessel et al. proposed integrating in a single agent several improvements of DQN (as discussed in Section 2.3.2.2), as well as distributional RL (Section 2.3.2.3). Authors report the individual gain from each addition, finding that an approach combining strategies is by large beneficial in the evaluated scenario.

Below we summarize the improvements which have been done in the original paper:

- Replacing 1-step distributional loss with a multi-step variant.
- Combining multi-step distributional loss with Double DQN.
- Adapting prioritized experience replay with KL loss.
- Dueling Network architecture with return distribution

It should be noted that, Rainbow is a pluggable agent so it is possible to add or remove different combination of these and additional extensions and improvements.

In our work we will evaluate models that use basic DQN, distributional DQN with prioritized experience replay (a subset of Rainbow), and a model that uses implicit quantiles, as offered in the Google Dopamine framework.

2.4 Summary

In this chapter we discussed necessary background knowledge which we believe will be helpful to understand the next chapters of this thesis.

First we provided the core ideas behind partitioning approaches, and the reason why DBMS systems need such optimization. We described different vertical partitioning algorithms, categorized them based on different dimensions, and briefly described a selection of them.

Following this we discussed state-of-the art RL approaches, which we use for our experiments and evaluations.

In the next chapter, we describe the design of our solution.

3. Self-driving vertical partitioning with deep reinforcement learning

In this chapter we present the design for our solution. Our design is based on combining two general aspects. First, the use of cost models and experts (i.e., traditional vertical partitioning algorithms) playing a role in normalizing the rewards. Second, the design of the environment itself allowing, in combination with the agents, for an RL process. In this chapter we describe this in detail. We structure our description as follows:

Architecture of our solution:
First, we establish the architecture or our solution, with all constitutent components (Section 3.1).

GridWorld environment:
Second, we present the GridWorld environment, which encompasses all details pertaining to action space, action semantics, observation space and reward engineering (Section 3.2).

Summary:
We conclude the chapter by summarizing the contents.

3.1 Architecture of our solution

Our implementation contains two separate parts:

1. To be able to learn and later to compare our proposed RL models with state-of-the art vertical partitioning algorithms. For this we use adapted versions of these algorithms proposed by [JPPD] and available as an open-source project[1]

[1]https://github.com/palatinuse/database-vertical-partitioning/

implemented in the Java programming language. Since we use a form of *learning from experts* (in our case, we adopt the rewards of the experts to normalize the rewarding scheme), we implemented REST APIs using the dropwizard framework[2] which helped us to interact efficiently with these implementations.

2. The second part comprises the RL framework itself, including our own environment, which we implement as an OpenAI Gym environment[3], such that it can be then called with state-of-the-art agents provided by the RL framework Google Dopamine [4].

To implement our RL based approaches we selected Dopamine - a framework for easy prototyping RL algorithms. Dopamine makes it easy for benchmark experimentation, allows easily integrate our own environment and research ideas to the framework and provide most state-of-the art proven algorithms out-of the box [CMG+18]. Dopamine in turn uses Tensorflow and Tensorflow-slim, for managing neural networks.

In Figure 3.1 the basic architecture of our design is shown. Here each blue box represents individual software components. We adapted our solution to the Dopamine architecture and added/changed necessary software components, as needed. In the following we introduce each component.

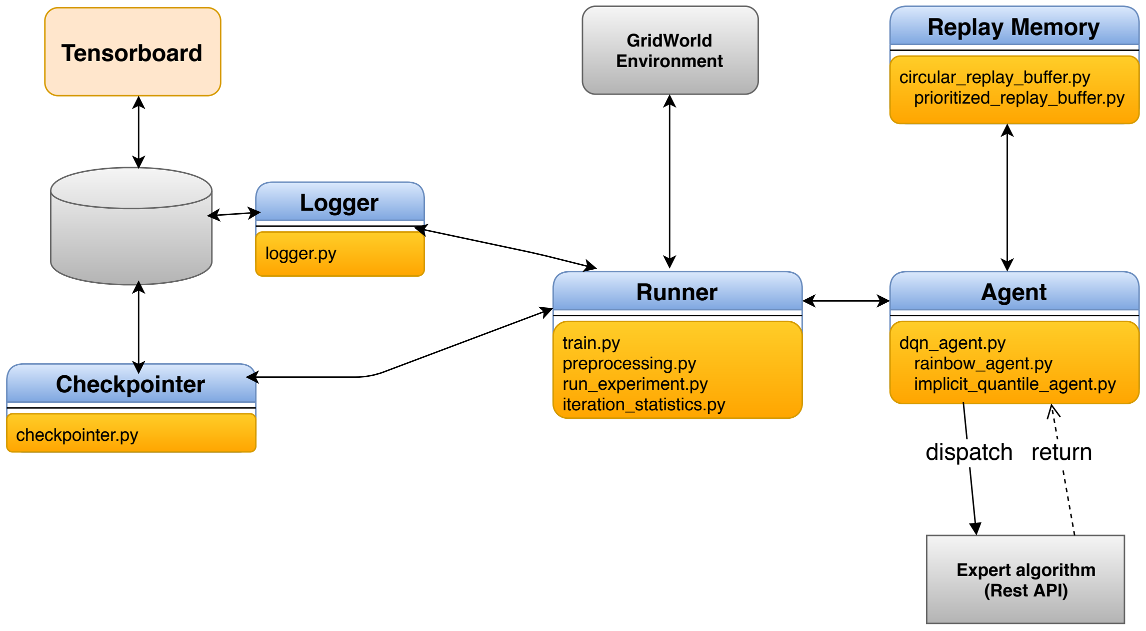


Figure 3.1: The architecture of our proposed solution, adapted from the original Dopamine architecture

[2]https://www.dropwizard.io/
[3]https://gym.openai.com/
[4]https://github.com/google/dopamine

1. **Runner** - organizes the learning process, launching an experiment decomposed into iterations, in turn composed of test and evaluate phases. The runner acts as a middleware, initializing the environment, connecting agents with the environment, and providing agents with the states from environment for inference and learning. The runner also manages the logging.

2. **Agents** - A DQN family of models is provided out-of-the box (requiring only minimal changes to run with new environments) and their hyper parameters are easily configurable using gin files. We had to adapt some agent behaviors for our use case (e.g. by introducing action pruning, or by adding novel parameters not available in Dopamine, like soft-updates).

3. **Replay Memory** - As described previously, the DQN family of algorithms uses a replay memory for effective learning. Dopamine provides several advanced implementations of this. Almost no changes were required to adopt these.

4. **GridWorld Environment** - Our learning environment, encompassing the majority of our choices for modeling vertical partitioning as a DRL task. We discuss it in detail in Section 3.2.

5. **Logger** - This component is used for saving experiment statistics for further visualization and plot analysis, using Colab or Tensorboard.

6. **Checkpointer** - For long running experiments a checkpointer component is provided that allows to periodically save the experiment states for weight re-use later on.

7. **Experts** - We require to compare with experts, as a baseline. Four our work we propose that the expertise of existing solutions can also be used. We discuss this in detail in Section 3.2.

3.2 Grid World Environment

Below we describe the main characteristics of our environment. We called our environment Grid World

Actions - We consider a learning environment with discrete actions. In our implementation we follow a "bottom-up" approach. At the beginning of each episode we consider each attribute as separated partitions. At each step we do only one action and merge two available partitions.

Since our experiments are based on the TPC-H benchmark and the largest relation on this benchmark (LINEITEM) contains 16 attributes the overall action set contains the following number of actions:

$$\binom{16}{2} = 120$$

.

As an example for how we map the actions to a numerical representation we can provide the following:

- **Action 0** $= [0, 1]$ - merging first and second attributes of relation
- ...
- **Action 119** $= [14, 15]$ - merging 15th and 16th attributes of relation

State representation - We experimented with several different approaches to represent the state, deciding at the end to use an approach which we describe next.

Our state is represented as a $[23 \times 16]$ matrix. This matrix can be interpreted as follows:

- **The first row** of the state matrix is the vector of attributes sizes (given a pre-defined ordering), multiplied by the number of rows of the relation of the TPC-H benchmark; For example for the LINEITEM table with a single row, the first row in our matrix will be:
  $[4, 4, 4, 4, 4, 4, 4, 4, 1, 1, 10, 10, 10, 25, 10, 44]$ If relations contain less than 16 attributes, the rest of this vector can be simply filled with zeros (0s) .
- **The next 22 rows** of the state matrix are the workload for the specific TPC-H or randomly generated workload, which contain a maximum of 22 queries.
  Each query vector here contains 16 elements filled with 0s or 1s.
  - 0 - the query does not touch the attribute at this position
  - 1 - query touches the attribute at this position

  In our implementation we do not consider the selectivity of queries on an attribute, since this does not affect our reward function. Future work could consider this aspect.

Action semantics/Merging strategy - If an agent makes a step and the step does not lead to an “end of game”, we can “merge” the attributes based on the selected action. During merging we follow the below steps:

- **Attribute row (first row)** - Sum the merged attribute values and keep the value in the lowest column index. In this case the highest column is deleted from the matrix and all right columns after current highest change decrease their indexes by one (shifting by one position to the left).
- **Workload** - The same rule applies as for the attribute row, except instead of summing we do a logical **OR** operation between values.

- The newly created matrix can now be fed to the NN as the next state.

On each next step the state matrix is shrinking by one in column size. The row size remains the same.

In figure Figure 3.2 one case of our merging strategy has been shown in *CUSTOMER* table example. This is not an optimal partition rather for illustration purposes we are considering the agent to be selecting randomly. For simplicity we skipped row numbers and trailing 0(zero) attributes.

Reward engineering - Our reward function is based on the HDD Cost model offered by [JPPD]. We reimplemented it as a Python function and included it in our environment.

We define our rewarding scheme as follows:

The value of a particular current state is defined as (Equation 3.1):

$$Value_{state} = \begin{cases} \frac{1000}{HDDCost}, & \text{if } HDDCost \neq 0 \\ 1000, & \text{otherwise} \end{cases} \tag{3.1}$$

The reward itself is calculated with (Equation 3.2)

$$Reward_{current\_state} = \begin{cases} 0, & \text{if not end of the game} \\ 100 * \frac{V_{current\_state} - V_0}{\Delta_{best}}, & \text{otherwise} \end{cases} \tag{3.2}$$

Here V_0 - is the value at the beginning of the episode (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 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 very different costs based on table and workload characteristics.

The game reaches a terminal state (game over) when one of the below 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)

a) Original settings of CUSTOMER table with 2 random queries

TableAttributes/ Random Queries	CUSTKEY	NAME	ADDRESS	NATIONKEY	PHONE	ACCTBALL	MKTSEGMENT	COMMENT
Attribute sizes	4	25	40	4	15	4	10	117
Q1	1	1	0	1	0	0	0	0
Q2	1	1	1	0	0	0	1	0

b) Step 1. Agent Choose an action 65 (merge attributes 5 (ACCTBALL) and 6 (MKTSEGMENT))

TableAttributes/ Random Queries	CUSTKEY	NAME	ADDRESS	NATIONKEY	PHONE	ACCTBALL, MKTSEGMENT (new partition)	COMMENT
Attribute sizes	4	25	40	4	15	14	117
Q1	1	1	0	1	0	0	0
Q2	1	1	1	0	0	1	0

c) Step 2. Agent Choose an action 15 (merge attributes 1 (NAME) and 2 (ADDRESS)))

TableAttributes/ Random Queries	CUSTKEY	NAME, ADDRESS (new partiton)	NATIONKEY	PHONE	ACCTBALL, MKTSEGMENT	COMMENT
Attribute sizes	4	65	4	15	14	117
Q1	1	1	1	0	0	0
Q2	1	1	0	0	1	0

d) Step 3. Agent Choose an action 32 (merge attributes 2 (ADDRESS) and 6 (MKTSEGMENT)))
Since Attributes ADDRESS and MKTSEGMENT are inside complex partitions we choose the smalliest number on each partition for merging
In this case we merge 1 and 5

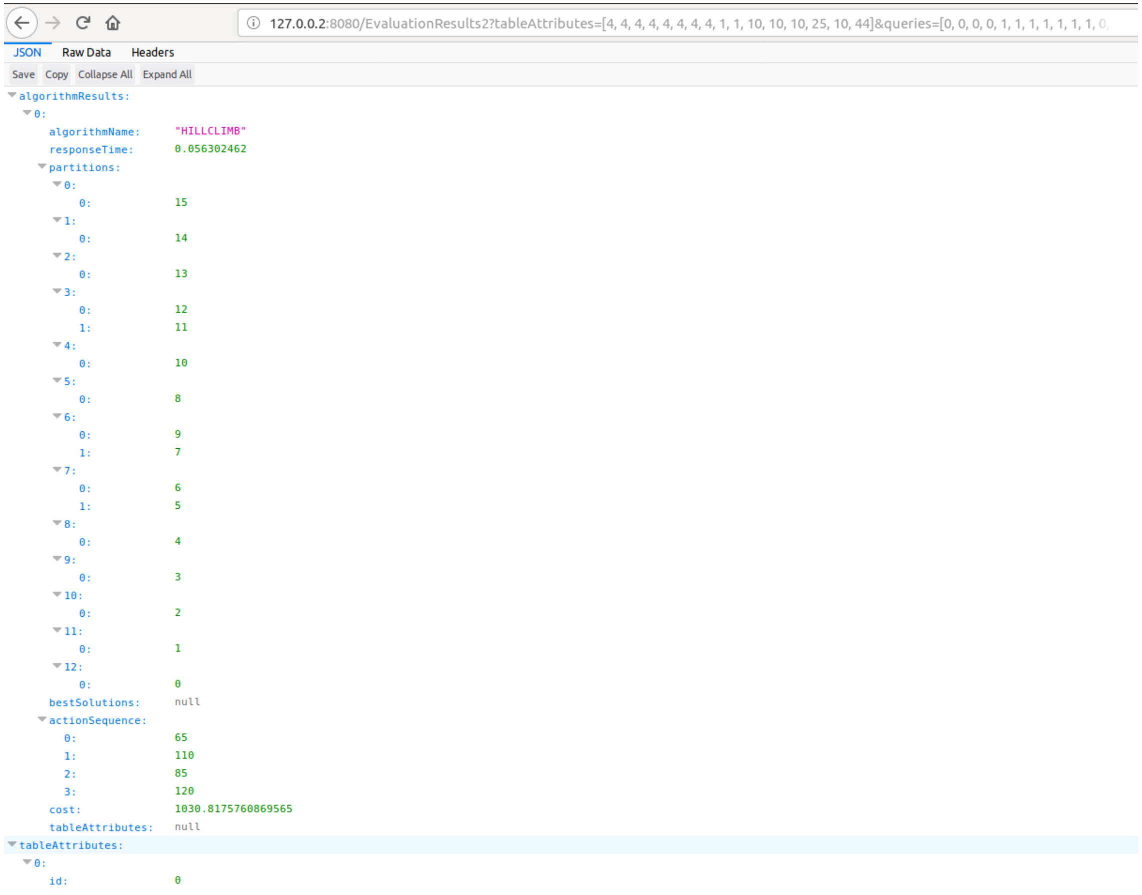
TableAttributes/ Random Queries	CUSTKEY	NAME, ADDRESS, ACCTBALL, MKTSEGMENT (new partiton)	NATIONKEY	PHONE	COMMENT
Attribute sizes	4	79	4	15	117
Q1	1	1	1	0	0
Q2	1	1	0	1	0

e) Step 4. Agent Choose an action 111 (merge attributes 11 and 13))
This is an invalid action since we don't have attributes 11 and 13 in the CUSTOMER relation

Figure 3.2: An example of state merging

Learning from experts - Since we need to measure the goodness of agents behaviour we need a sequence of steps and a final cost proposed by an expert algorithm for a given table and fixed or randomly generated workload.

As an expert we chose the *HILLCLIMB* algorithm offered by [JPPD]. We interacted with this expert through a REST API call. Our API is a GET method and we pass the table in the form of attributes size array and workload configuration and returns us the best cost achieved by *HILLCLIMB*, and the sequence of piecewise actions (i.e, the items merged at each iteration of HILLCLIMB) needed to reach this reward. In Figure 3.3 an example of our REST API response is shown.



```
127.0.0.2:8080/EvaluationResults2?tableAttributes=[4, 4, 4, 4, 4, 4, 4, 4, 1, 1, 10, 10, 10, 25, 10, 44]&queries=[0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0
```

```
▼algorithmResults:
  ▼0:
      algorithmName:        "HILLCLIMB"
      responseTime:         0.056302462
    ▼partitions:
      ▼0:
          0:                15
      ▼1:
          0:                14
      ▼2:
          0:                13
      ▼3:
          0:                12
          1:                11
      ▼4:
          0:                10
      ▼5:
          0:                8
      ▼6:
          0:                9
          1:                7
      ▼7:
          0:                6
          1:                5
      ▼8:
          0:                4
      ▼9:
          0:                3
      ▼10:
          0:                2
      ▼11:
          0:                1
      ▼12:
          0:                0
      bestSolutions:        null
    ▼actionSequence:
        0:                  65
        1:                  110
        2:                  85
        3:                  120
      cost:                 1030.8175760869565
      tableAttributes:      null
▼tableAttributes:
  ▼0:
      id:                   0
```

Figure 3.3: REST API response example

3.3 Summary

In this chapter we provided an architectural view of our solution. Such view combined some implementation and general design aspects. Next we presented the Grid World environment, describing in detail how we mapped the vertical partitioning process to be deep reinforcement learning solution. In the next chapter we establish our research questions and provide details about our experimental setup.

4. Experimental setup

In this chapter we establish our research questions and overview the experimental setup we used for evaluation.

The chapter is organized as follows:

- **Research Questions:**
  First we provide several research questions that we aim to address in our work (Section 4.1).

- **Hyper-parameters for DQN agents:**
  We define how the DQN agents are configured in our experiments (Section 4.2).

- **Unified settings for proposed algorithms:**
  We discuss adapted and unified versions of vertical partitioning algorithms discussed in (Section 2.2), which were required for our evaluation ().

- **Measurement Process:**
  We introduce our basic measurement methodology (Section 4.4).

- **Evaluation Environment:**
  We list relevant characteristics from the execution environment of our tests (Section 4.5).

- **Benchmarks:**
  We describe the TPC-H benchmark which has been used for our evaluations (Section 4.6).

- **Summary:**
  We conclude the whole chapter in (Section 4.7).

4.1 Research Questions

The aim of this research is to provide early results about the feasibility of using state-of-the art RL models for self-driving vertical partitioning.

Accordingly, we would like to answer to the following research questions:

1. What is the training cost, for cases with different levels of generality? Is there any impact with regards to the complexity of the learning instance (i.e., the number of steps)? What techniques are important to speed-up convergence? To answer these questions we consider three cases of different specificity/generality:

    (a) RL agent is training on a fixed workload and table.

    (b) RL agent is training on a set of fixed workload and table pairs.

    (c) RL agent is training on different workloads but the same table.

2. Once the model(s) has converged (or partially converged), how does the performance on inference (i.e, solving the partitioning task) compare with state-of-the-art vertical partitioning algorithms? How does this scale with respect to complexity of the task or number of attributes in the relation?

4.2 Hyper-parameters for DQN agents

In order to describe our experimental setup, we start by describing the relevant parameters of the DQN agents.

Neural Network (NN) design - since we use DQN-based approaches we need to feed the state by adopting a neural network. We checked several NN architectures that we could change in the Dopamine framework and we found that the default convolutional architecture surprisingly worked well for our given task. In (Figure 4.1) the architecture of the adopted solution is shown.

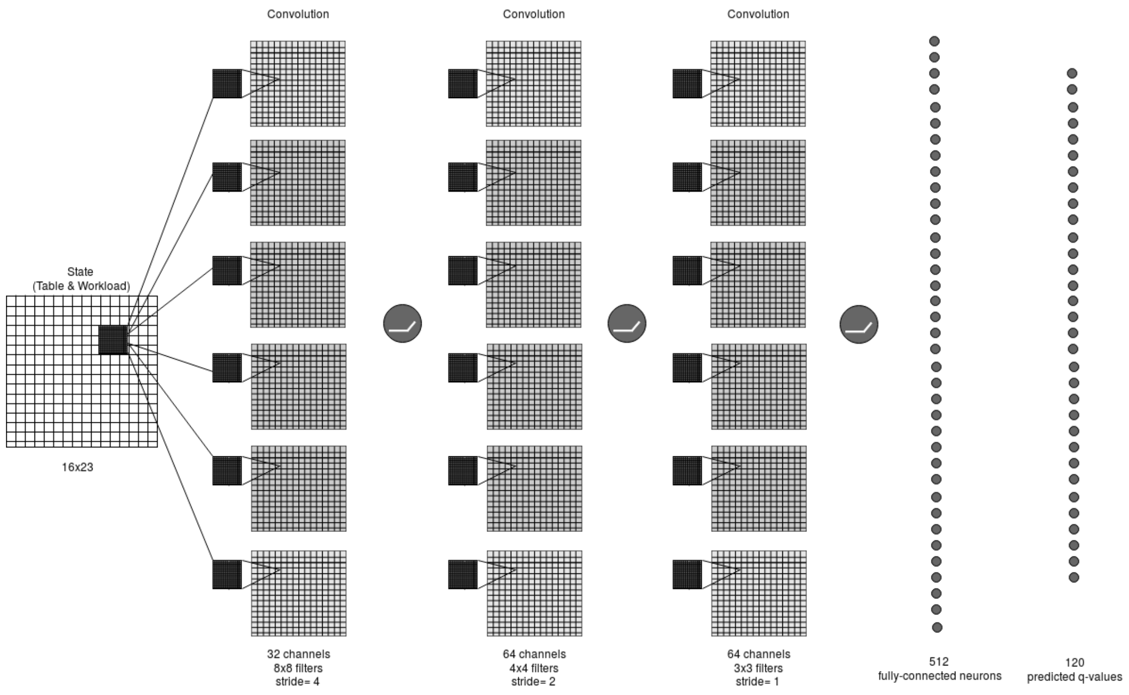


Figure 4.1: Proposed NN architecture

Dopamine provides a number of tunable hyper-parameters for DQN, Rainbow and Implicit Quantile agents. In our experiments we keep a majority of parameters in their default values. In this section we briefly explain the meaning of the offered parameters by categorizing them into several tables. Understanding this is crucial for our further tests about the impact of some of them on the learning process.

Table 4.1 provides a brief explanation of general environment and experiment parameters:

Parameter	**Explanation**
tf_device	GPU or CPU
game_name	Used environment
training_steps	Number of training steps within one iteration
evaluation_steps	Number of evaluation steps within one iteration
max_steps_per_episode	Maximum number of steps within episode before end of episode
num_iterations	Number of iterations for the experiment

Table 4.1: General experiment parameters for all agents

Table 4.2 explains parameters for general reinforcement learning, which are common to all agents.

Parameter	Explanation
gamma(γ)	discount factor(weights for future rewards)
epsilon_train	ϵ value in ϵ-greedy approach during training
epsilon_decay_period	ϵ needs to decay during the learning process according to a predefined approach, this parameter controls it
boltzmann	Instead of ϵ-greedy using Boltzmann exploration
prune_actions	Heuristics helps for faster convergence, we remove a-priori invalid actions from action dictionary before chosing an action

Table 4.2: General RL parameters for all agents

In Table 4.3 we provide explanations for parameters specific to DQN agents.

Parameter	Explanation
update_period	the period of learning
target_update_period	update period for the target network
update_horizon	horizon at which updates are performed (n-step Q-Learning)
min_replay_history	Min. number of transitions that should be experienced before training n
replay_capacity	Max. number of transitions in exp.replay buffer
batch_size	randomly sampled elements of the transitions of size batch_size
optimizer	Optimization algorithms used to minimize objective function of NN
learning_rate	Learning rate of Gradient descent

Table 4.3: DQN-specific parameters

Implicit quantile and Rainbow agents extend DQN, they also offer additional parameters. Table 4.4 shows some of these parameters.

Parameter	**Explanation**
kappa	Huber-loss cutoff value
num_tau_samples	Number of quantile samples for loss estimation
num_tau_prime_samples	Number of target quantile samples for loss estimation
num_atoms	Number of atoms (51 according to C51 algorithm)
replay_capacity	Max. number of transitions in exp.replay buffer
batch_size	randomly sampled elements of the transitions of size batch_size
optimizer	Optimization algorithms used for minimize Objective function of NN
learning_rate	Learning rate of Gradient descent

Table 4.4: Implicit quantile-specific parameters

Implicit quantile uses the Adam Optimizer instead of RMSProp, which is used by DQN.

4.3 Unified settings for proposed algorithms

As discussed in (Section 2.2) each of proposed algorithms (Navathe, O2P, AutoPart, HillClimb) has been used for a specific scenario, based on different characteristics of database system, hardware etc. This makes it slightly odd to compare algorithms with their original settings in one common benchmark, therefore proposed algorithms has been adapted for the same configuration. These common configurations makes it possible to compare the algorithms on a fair ground. Below the common configuration are described:

1. *Data granularity* - We use meta-data of TPC-H tables and workloads.

2. *Hardware* - All selected algorithms are optimized for Hard disk optimization. Therefore the cost models we use follow disk-based storage models.

3. *Workload* - We are using a fixed set of queries (offline workload) and only scan and project queries of TPC-H workload (22 queries in total). We also are considering a random workload in RQ(1.(c)) (the maximum number of queries limited to 22 in this case). In the original settings scale factor has been assumed as 10. We don't consider any specific DBMS in our experiments.

4. *Replication* - We don't consider data replication since only AutoPart algorithm supports it, but we want to keep the same characteristics for all algorithms.

4.4 Measurement Process

In our evaluation we focus on convergence time and inference time of our agents as well as time need to be able (for the traditional algorithms) to formulate the partitions. Since our implementation contains both Java and Python parts we use:

- System.nanoTime() - Java system timer provide sufficient precision.
- from timeit import default_timer as timer - measure execution time for Python

To guarantee more reliable results we replay each experiments several times, and disclose the number of repetitions in each case.

4.5 Experiment Environment

Our experiments were executed on a multi-core machine running on Ubuntu 16.04 system. Detailed parameters of the system as well as used software versions are given below:

- Intel® Core^{TM} i7-6700HQ CPU @ 2.60GHz 8 (8 cores in total
- RAM - 15.5 GiB of memory
- GPU - GeForce GTX 960M/PCIe/SSE2
- Java version: Java SDK 8u111-linux-x64
- Python version:Python3.5
- Dopamine framework: Dopamine 1.0.5
- Tensorflow: Tensorflow-GPU 1.12.0
- Numpy 1.15.4
- Gym 0.10.9

4.6 Benchmark

We evaluate our RL models using the TPC-H Benchmark. The schema of this benchmark is shown in Figure 4.2.

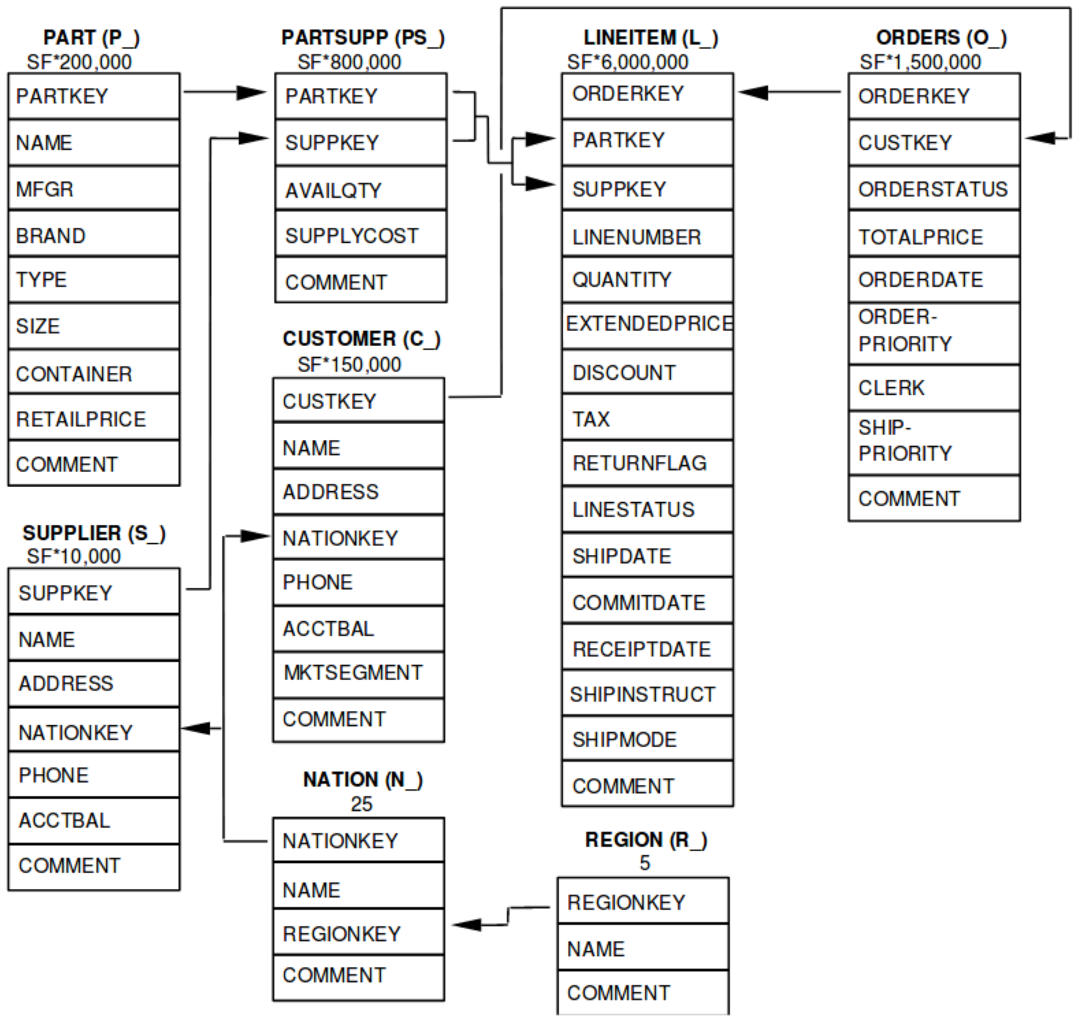


Figure 4.2: TPC-H schema [1]

The TPC-H database contains eight tables and 22 ad-hoc queries to simulate some decision support queries occurring together with operational transactions. TPC-H's schema represents a simple data warehouse for holding data about customer, sales, and part suppliers. TPC-H suggests several rules regarding partitioning strategies. LINEITEM and ORDERS contain about 80% of TPC-H data. TPC-H allows different scaling factors like (1, 10, 30, 100 etc.). As we mentioned earlier we use 10 as a scale factor in our experiments.

[1]http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf

4.7 Summary

In this chapter we introduce our research questions. Next we describe parameters, unified settings for vertical partititioning algorithms, the measurement process, benchmarks we used and finally parameters of the hardware that we use for our evaluations.

5. Evaluation and Results

In this chapter we discuss the evaluation and results of our experiments. This chapter contains two parts:

- In (Section 5.1) we discuss problems related to training costs, convergence results and the impact of task complexity for the three cases of generality discussed in RQ 1.
- In (Section 5.2) we compare the performance of our learning models in terms of solving partitioning tasks with state-of-the-art vertical partition algorithms.

5.1 Training of models

In this evaluation section we focus on our first research question. We test different factors which might have potential influence for our models convergence and performance. As discussed in Chapter 4 we use the Dopamine framework and use models offered by Dopamine.
In our evaluations we used three DRL agents:

- Deep Q-Network (DQN)
- Rainbow
- Implicit Quantile

Rainbow and Implicit Quantile agents extend the DQN agent but also add additional hyper-parameters for tuning. Since, the network architecture used inside these modelsare mainly based on convolutional architecture offered by Dopamine therefore we keep all parameters of network as it is.

5.1.1 Complexity of learning instance

Since we use the TPC-H workload and HillClimb algorithm as an expert for our experiments we need to know the sequence of actions which need for HillClimb to get optimal partitions for each individual table of TPC-H with its exact workload and scale factor 10 (SF=10).

In table (Table 5.1) we summarized those actions:

TPC-H table	**Actions**
PART	[0]
SUPPLIER	[2, 42, 0]
PARTSUPP	[0, 0]
CUSTOMER	[31, 55, 30, 2, 0]
ORDERS	[0]
LINEITEM	[65, 105, 76, 0]
NATION	[0, 0, 0]
REGION	[0, 0]

Table 5.1: Actions(steps) needs to get optimal partitions for HillClimb expert

Is it noticeable that we need more actions to get optimal partitions for LINEITEM, CUSTOMER and SUPPLIER. Instead we don't need any actions for PART and ORDERS to get partitions since initial partitions for these tables are already optimal ones according to HillClimb's cost model.

5.1.2 Convergence in case of fixed workload and tables

In this tests we train first on TPC-H tables individually, with their own workloads. Since this is a relatively simple case we expect that all 8 tables converge to the maximum possible reward in very early iterations. Therefore, we limited the number of iterations to 150.

The results of convergence for all three models is shown in Figure 5.1 and Figure 5.2.

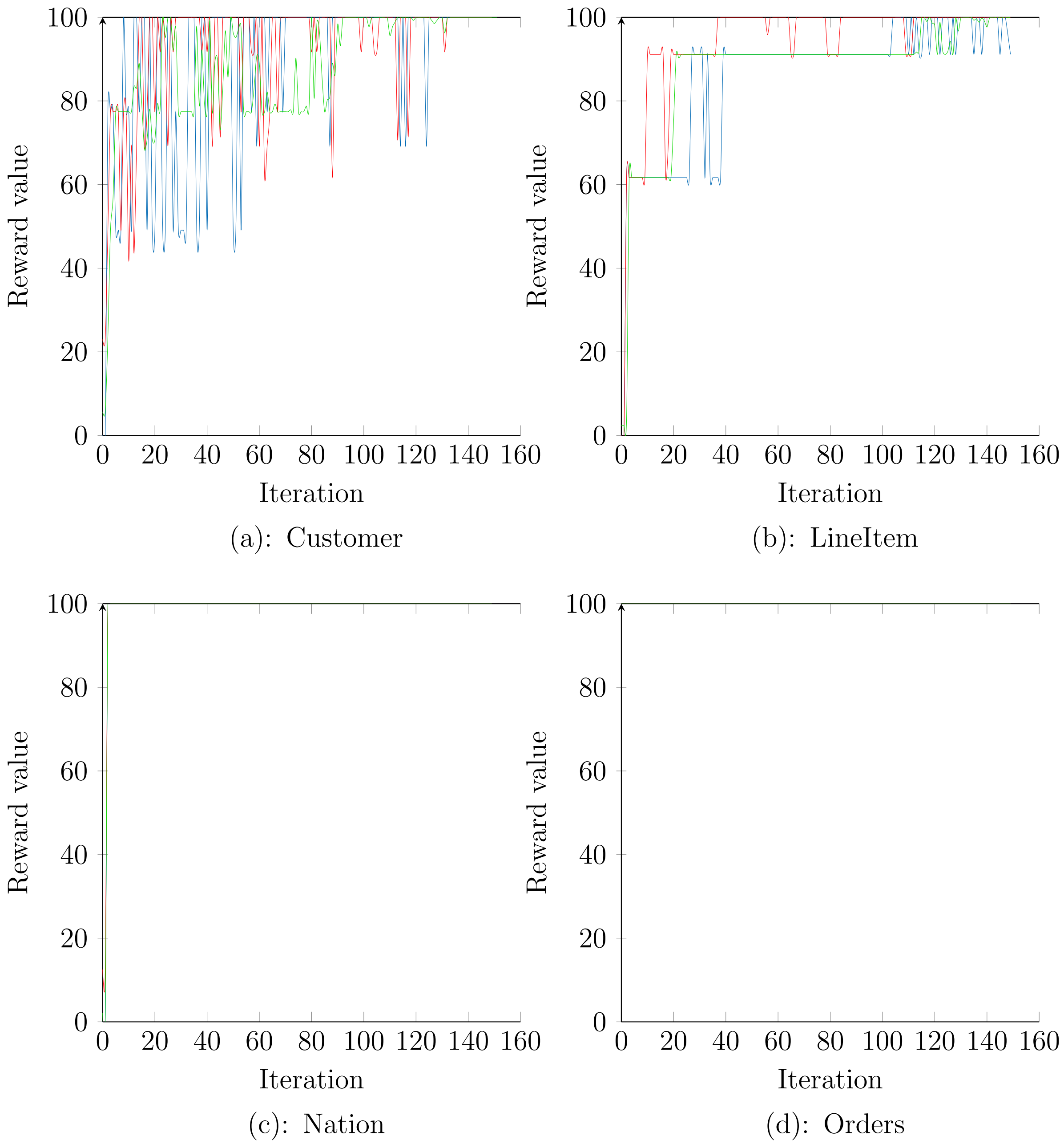
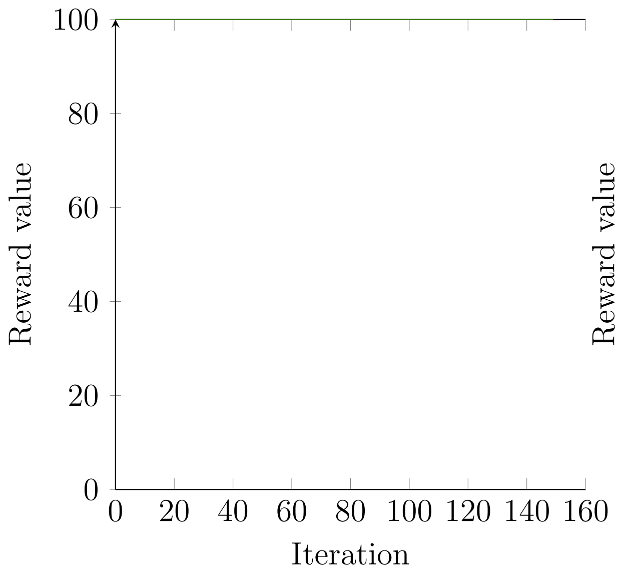
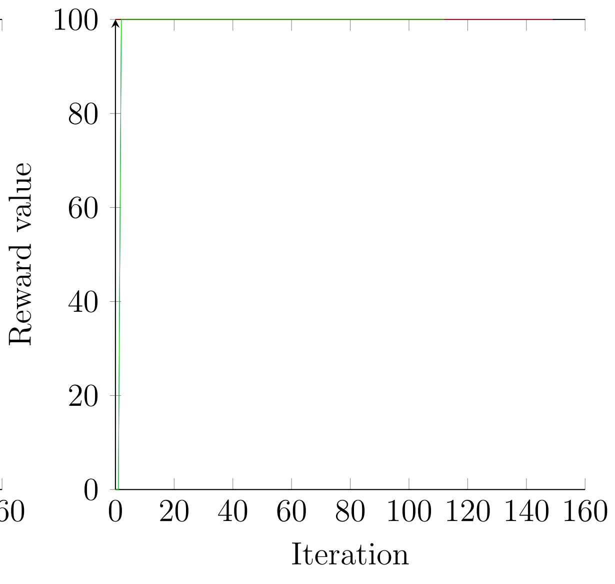


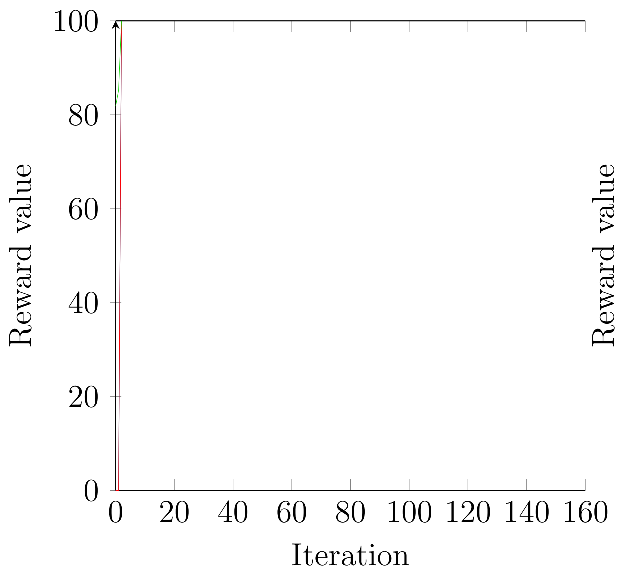
Figure 5.1: Single table, Blue line - DQN agent, Red line - Rainbow agent, Green line - Implicit quantile agent



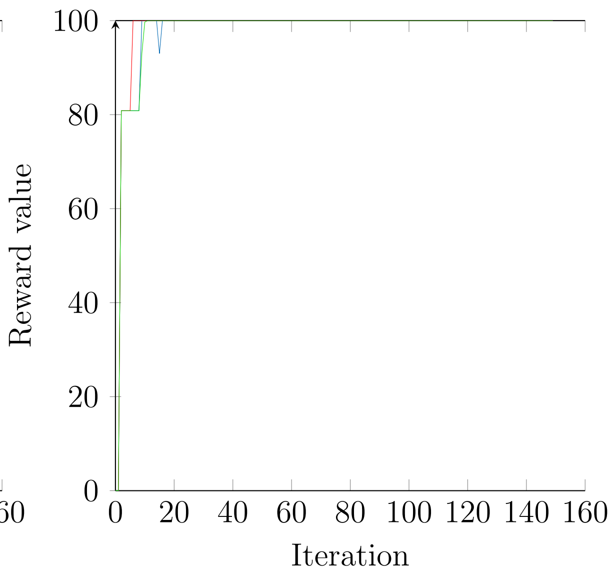
(a): Part



(b): PartSupp



(c): Region



(d): Supplier

Figure 5.2: Single table learning

It is noticeable that after 150 iterations all models have converged. For the tables *Orders* and *Part* convergence happens immediately for all three agents since there is no action needed for convergence for them. For table *Customer* convergence happens around iteration 100 but some fluctuations continues for the DQN agent. For *Lineitem* convergence happens around iteration 50 for all agents, and Implicit quantile agents gain 100% of possible reward, whereas DQN and Rainbow agents fluctuate around 90-100% by the 150th iteration. For other tables convergence happens very fast within the first

10 iterations, since the these tables require at maximum three steps to gain optimal partitions.

This first case shows that agents are able to learn simple cases (fixed table, fixed workload) very fast.

5.1.3 Convergence in case of fixed workload and table pairs

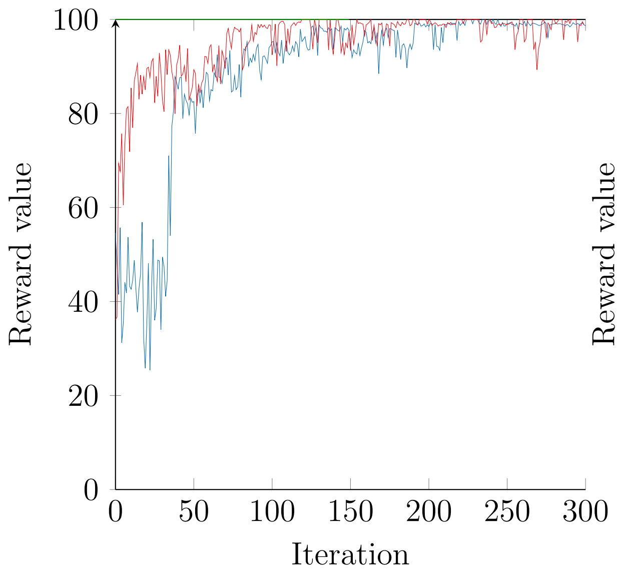
Since our agents were successfully able to learn and converge in case of fixed table-fixed workloads, we decided to add additional complexity to the learning process. Here we mixed up all eight tables and the corresponding eight workloads, training and evaluating on a random selection of them. Here we test several combinations of hyper-parameters, which we describe as follows:

- All parameters are the same, except update_horizon (UH) is different: We test UH (UH=1, UH=3 and UH=5) just to choose the best option for our further experiments

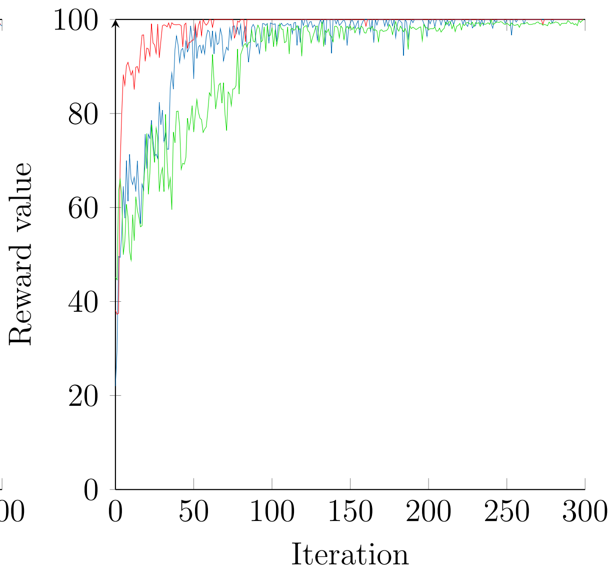
- Prune actions versus non prune actions

- Soft-update versus hard-update: This refers to how the synchronization between current and target network occurs. In a soft update the synchronization occurs on each step, but instead of weights being totally copied within one network and another, only a small percentage (tau) of the weights gets updated.

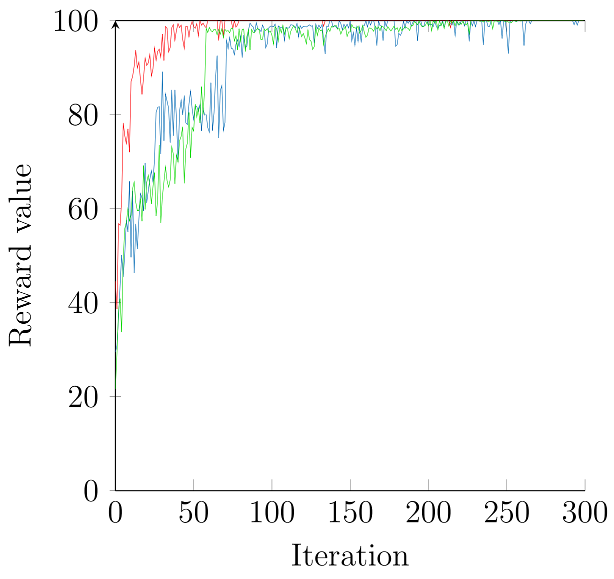
- ϵ-greedy versus Boltzmann



(a): Update horizon (UH=1)



(b): Update horizon (UH=3)



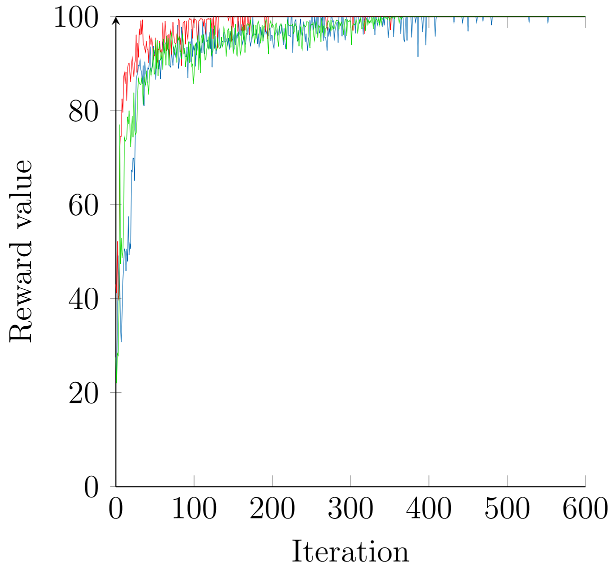
(c): Update horizon (UH=5)

Figure 5.3: Update horizon tuning

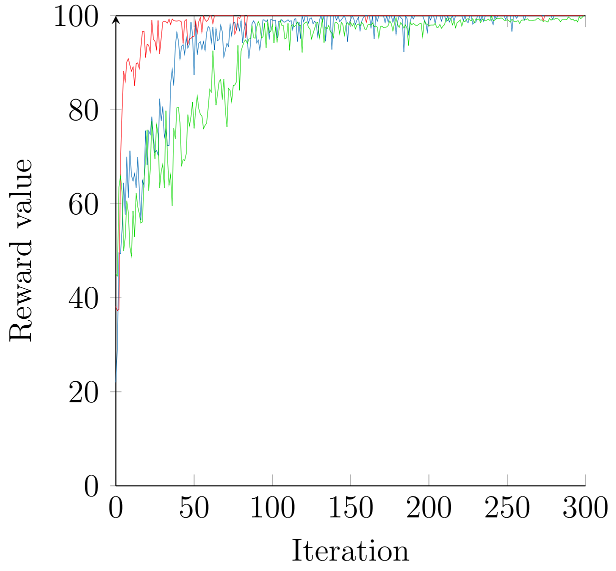
It is noticeable that in all three cases all three models converge relatively fast. In case of (UH=1) and (UH=5) fluctuations continue even after 300 iterations. Therefore we chose the most stable UH=3 for the later experiments.

In our further experiments we used advantages of action pruning heuristic. It helps eliminate a-priory invalid actions beforehand and accelerates the learning process. In specific, with this approach we filter out invalid actions from the Q-value predictions provided from the neural network, before using such predicitions to select the next action. By doing so we are able to prevent an agent to avoid stepping on an action that will be invalid, resulting in game over.

In (Figure 5.4) we compared agents with pruning and non pruning actions.



(a): Non-pruning actions



(b): Pruning actions

Figure 5.4

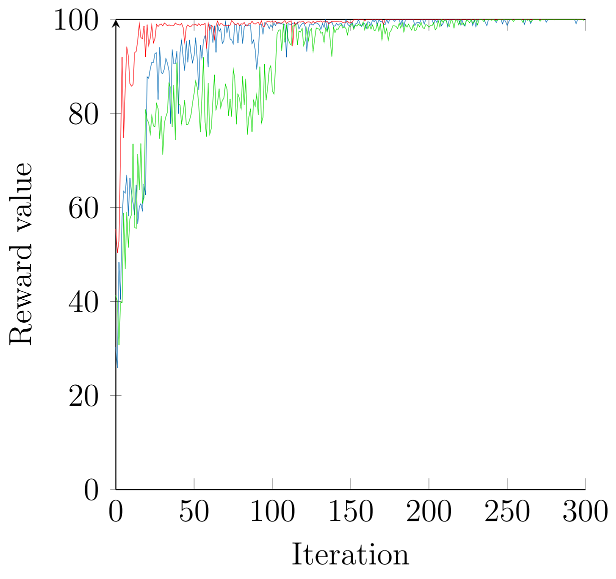
As we can see, in case of action pruning convergence happens around the step 200 for all learning agents, but in case of non-pruning convergence for all agents happens around

400. Therefore, for the next experiments we keep the action pruning heuristic for faster learning.

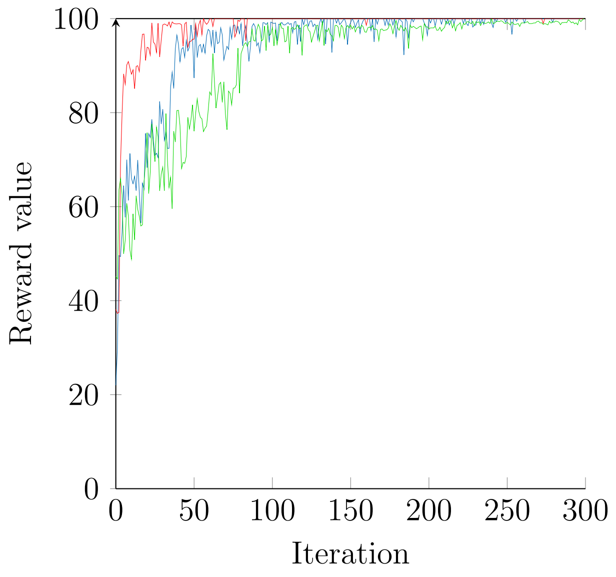
The next set of experiments were dedicated to comparing hard versus soft update of weights from the NN. In the case of hard-updates we follow the fixed-Q-target strategy where we move weights of learning value network to target value network after certain amount of time described via the hyper parameter target update period.

In case of soft-update we follow the same strategy but also change targets weight very slightly gradually on every step.

In Figure 5.5 we compared agents with hard and soft updates.



(a): Hard-update

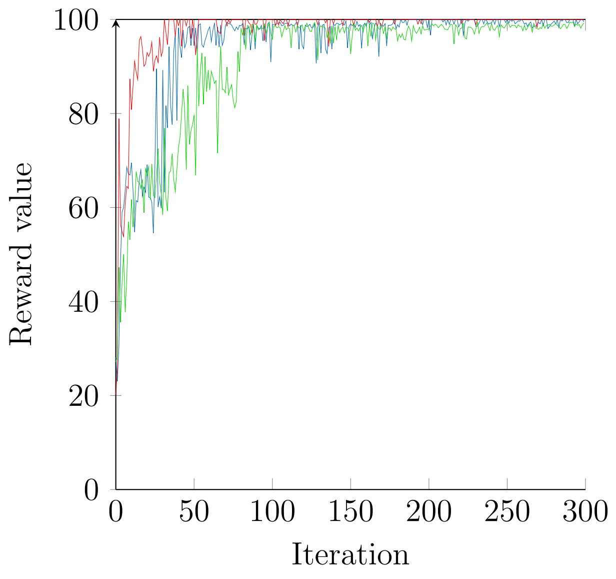


(b): Soft update

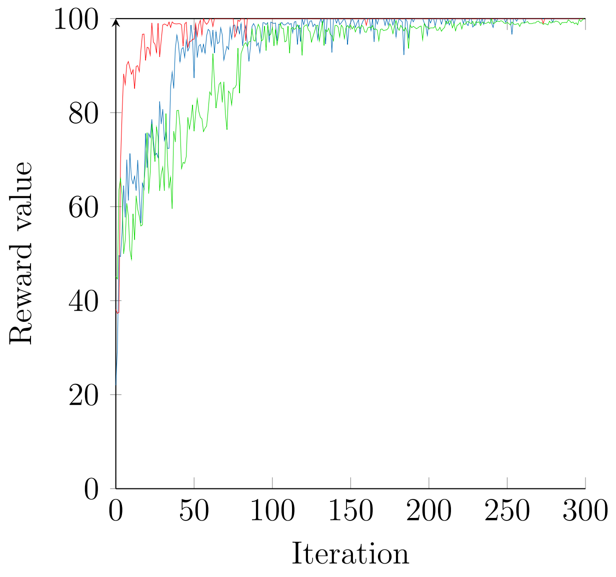
Figure 5.5: Hard and soft updates, UH=3, with pruning actions

In our experiments we found that there is no dramatic improvements in terms of convergence between soft-update and hard-updates but in case of soft-update the average rewards increase more gradually and fluctuations are lower. Therefore we consider soft-updates for our later experiments, but we did not so when testing only running times, since it could introduce runtime overheads.

Next we test the effect of a Bolztmann exploration strategy to the learning.



(a): Bolztmann approach



(b): ϵ-greedy approach

Figure 5.6: Bolztmann and ϵ-greedy strategies (Soft update, UH=3, with pruning actions)

Surprisingly we find that the ϵ-greedy approach makes the convergence more stable after 200 iterations. Therefore for our next experiments we use it instead of Boltzmann exploration.

In (Figure 5.7) the average time needed for training has been shown for our three agents in case of GPU and CPU usages. We run each experiment five times and collect the average result for each case.

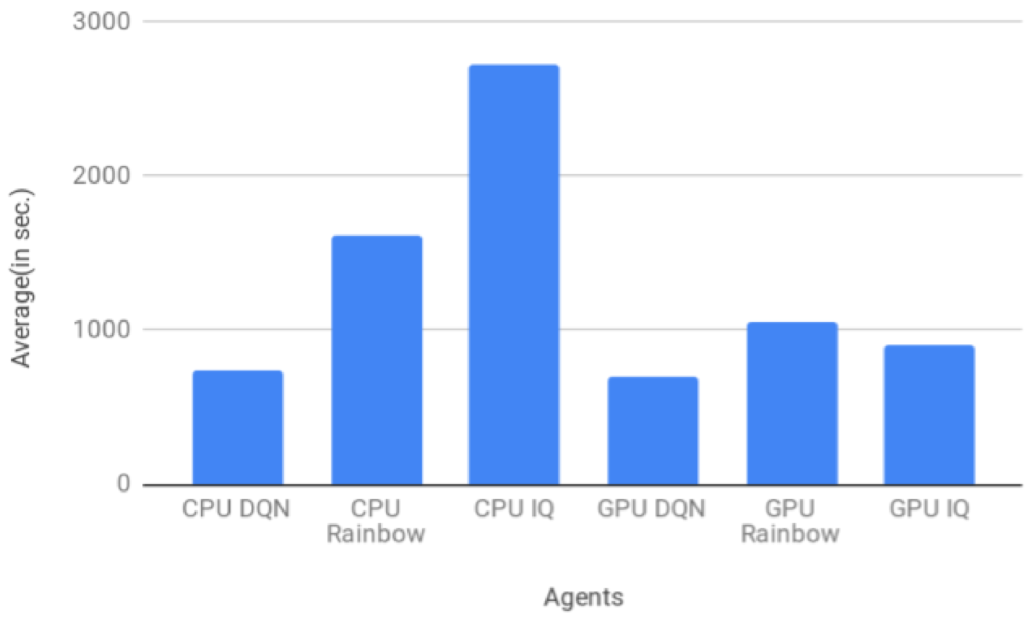


Figure 5.7: Average training time for agents

It is noticeable that for the DQN agent there is no big impact brought forward by the use of a GPU in our learning models, whereas for Rainbow and Implicit Quantile agents GPU made the learning process about two and three times faster, respectively.

5.1.4 Case of fixed table and random workload

We consider this third case as the most challenging one, since our agent needs to deal with much more randomness. As our experiment we select the *CUSTOMER* table and on each episode we randomly generated a new workload matrix and fed it to the agent. The only limitation here is that the workload could contain maximum 22 queries. Theoretically, in this case the number of possible workloads is 2^{352}. Since from previous experiments we found that the implicit quantile agent works better in comparison to DQN and Rainbow, we use only this agent for training. Also, from previous experiments we kept the below settings of the agent.

- We use UH=3
- We use prune actions
- We use soft-update
- We use ϵ-greedy

Unfortunately, in this case our agent is not able to converge and oscillated around 78-85% after 72 hours of training (we stop it at 4000 iterations).

In Figure 5.8 and Figure 5.9 the evaluation and training results of the agent are shown.

According to our observation, the reason for the slow convergence could be possibly the long-sequence of actions (in some cases 7) as seen in some cases for the expert algorithm. An agent could learn up to 5 steps relatively easy, but faces a problem to learn more complex patterns. Another reason could be the very strict rules we defined for reward functions, with delayed rewards.

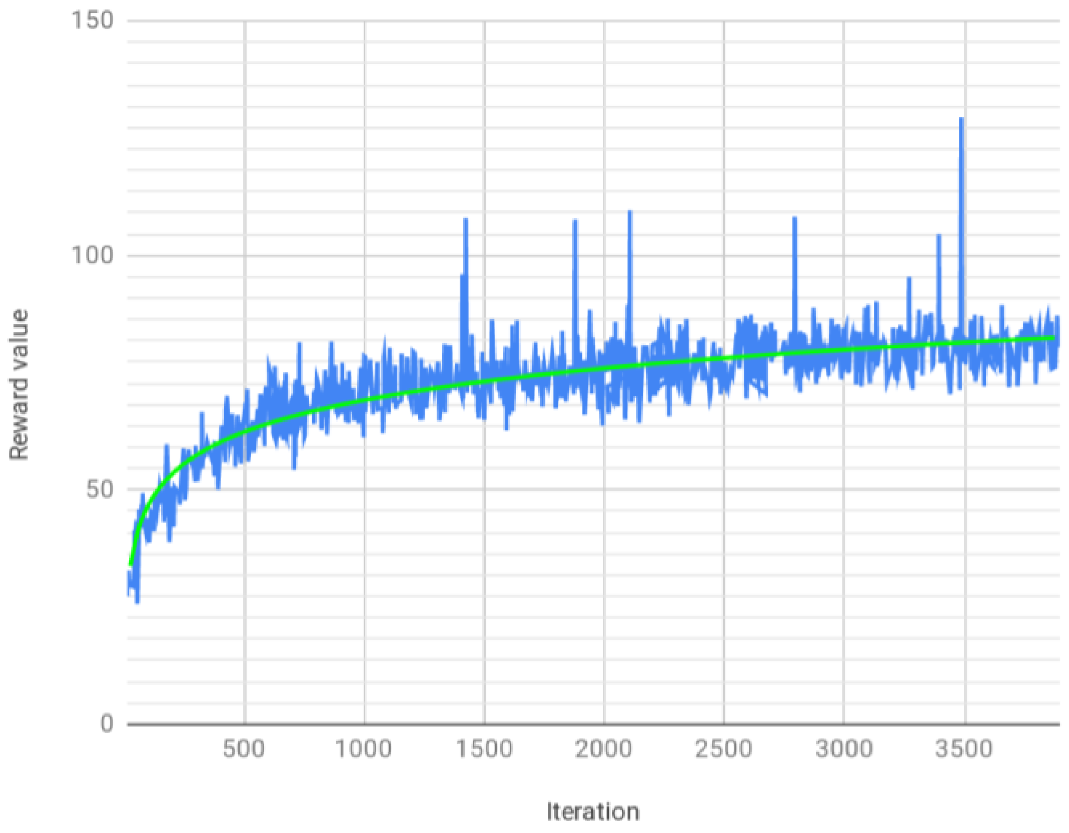


Figure 5.8: Evaluation plot for random workflow case, with trendline

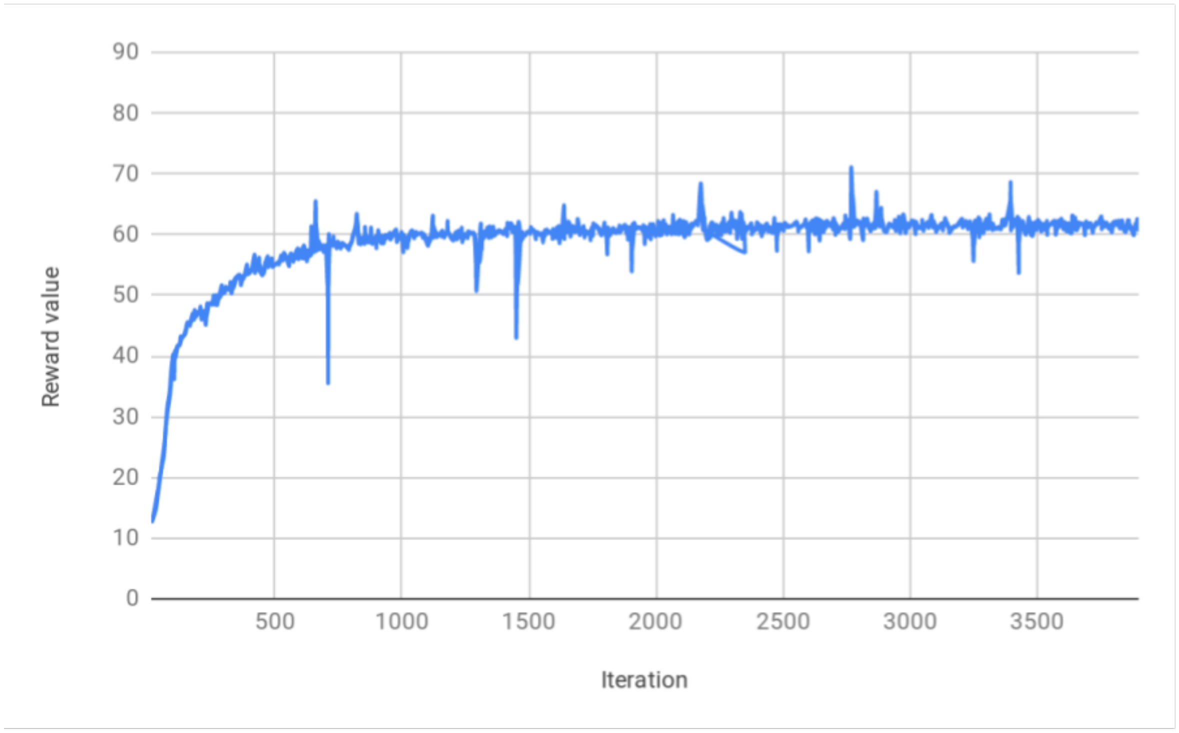


Figure 5.9: Training plot for random workload case

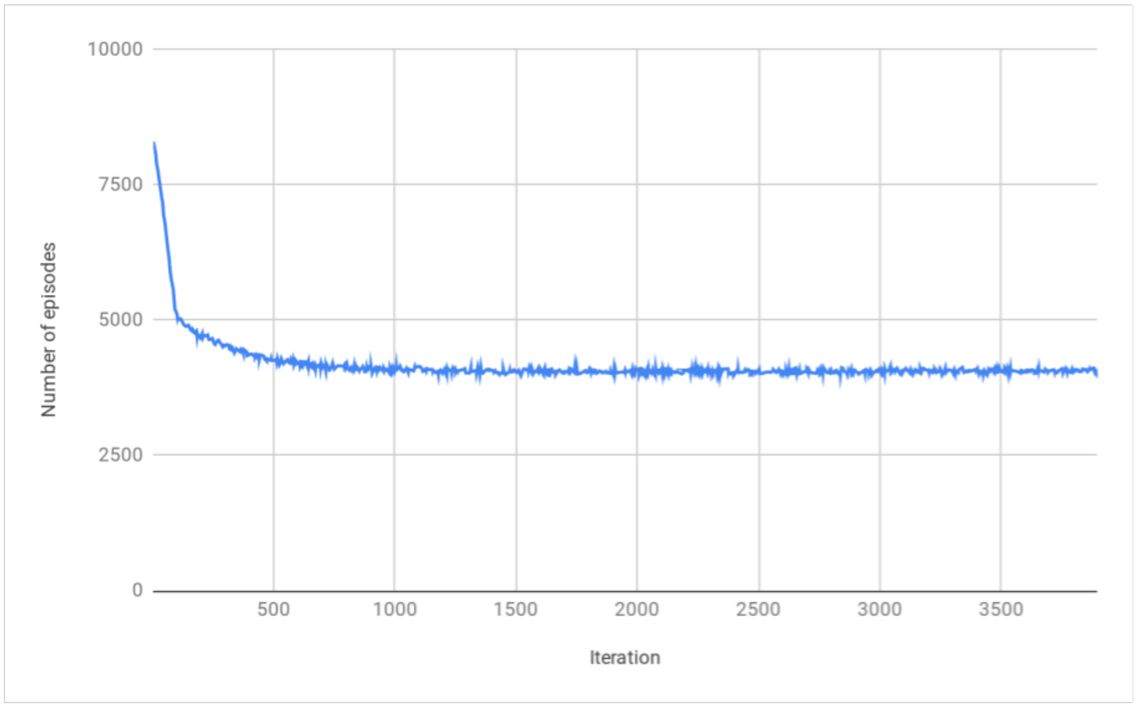


Figure 5.10: Number of episodes per iteration for random workload case

It is noteworthy that in some iterations the reward value exceed 100%. The reason of this could be that our agent found a solution better than HillClimb, hence surpassing the V_{best_case}. In that case, according to our reward calculation it will exceed 100% but this happen quite rarely.

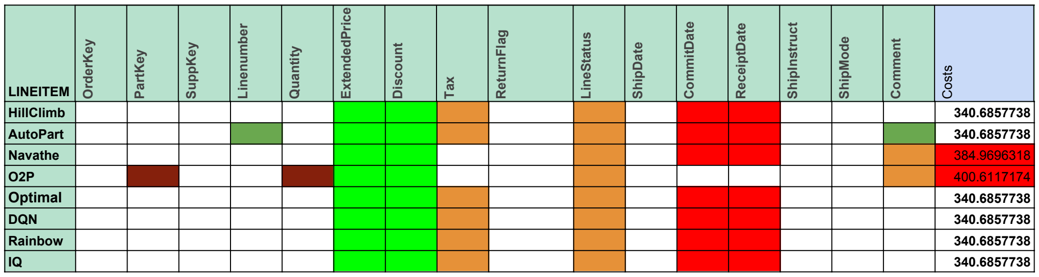
5.2 Algorithms comparison

In this section we compare our already trained agents with state-of the art algorithms, as discussed in (Section 2.2).

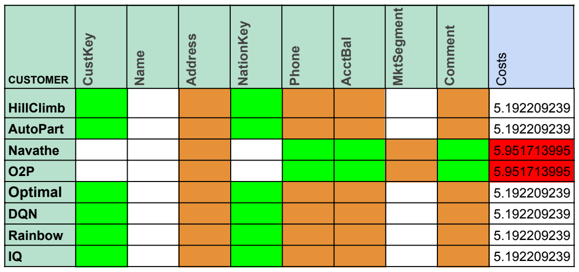
5.2.1 Generated partitions

We start with the partitions generated by our RL agents and by state-of-the-art algorithms. Here we consider the own workload for each given table, while evaluating on agents trained on the 8 tables each with their workload.

In Figure 5.11, Figure 5.12, Figure 5.13 and Figure 5.14 we show, side-by-side, the results.

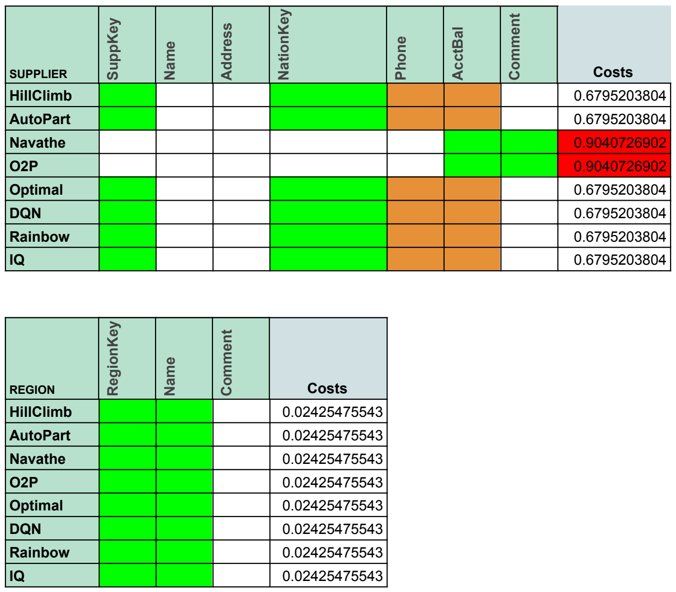


LINEITEM	OrderKey	PartKey	SuppKey	Linenumber	Quantity	ExtendedPrice	Discount	Tax	ReturnFlag	LineStatus	ShipDate	CommitDate	ReceiptDate	ShipInstruct	ShipMode	Comment	Costs
HillClimb						🟩	🟩	🟧		🟧		🟥	🟥				340.6857738
AutoPart				▨		🟩	🟩	🟧		🟧		🟥	🟥			▨	340.6857738
Navathe						🟩	🟩			🟧		🟥	🟥			🟧	🟥 384.9696318
O2P		🟫			🟫	🟩	🟩			🟧						🟧	🟥 400.6117174
Optimal						🟩	🟩	🟧		🟧		🟥	🟥				340.6857738
DQN						🟩	🟩	🟧		🟧		🟥	🟥				340.6857738
Rainbow						🟩	🟩	🟧		🟧		🟥	🟥				340.6857738
IQ						🟩	🟩	🟧		🟧		🟥	🟥				340.6857738



CUSTOMER	CustKey	Name	Address	NationKey	Phone	AcctBal	MktSegment	Comment	Costs
HillClimb	🟩		🟧	🟩	🟧	🟧		🟧	5.192209239
AutoPart	🟩		🟧	🟩	🟧	🟧		🟧	5.192209239
Navathe			🟧		🟩	🟩	🟧	🟩	🟥 5.951713995
O2P			🟧		🟩	🟩	🟧	🟩	🟥 5.951713995
Optimal	🟩		🟧	🟩	🟧	🟧		🟧	5.192209239
DQN	🟩		🟧	🟩	🟧	🟧		🟧	5.192209239
Rainbow	🟩		🟧	🟩	🟧	🟧		🟧	5.192209239
IQ	🟩		🟧	🟩	🟧	🟧		🟧	5.192209239

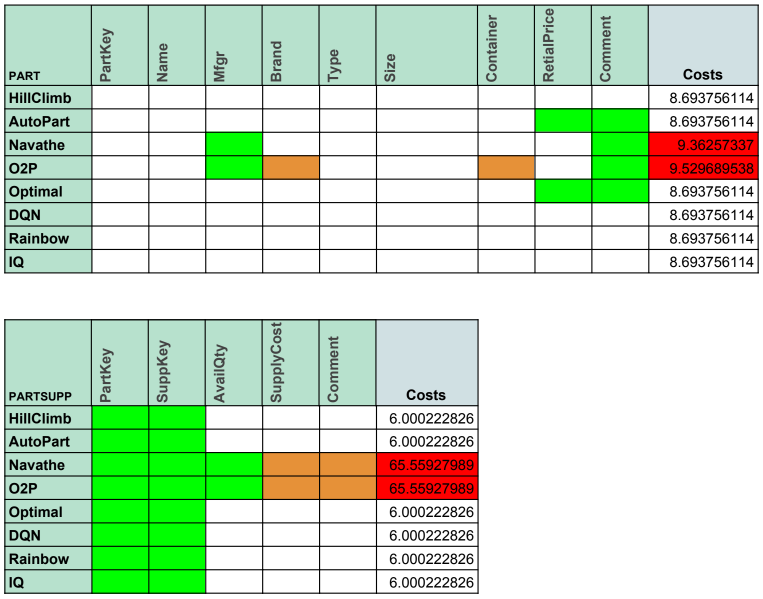
Figure 5.11: Customer and Lineitem tables with corresponding costs for each algorithm



SUPPLIER	SuppKey	Name	Address	NationKey	Phone	AcctBal	Comment	Costs
HillClimb	🟩			🟩	🟧	🟧		0.6795203804
AutoPart	🟩			🟩	🟧	🟧		0.6795203804
Navathe						🟩	🟩	0.9040726902
O2P						🟩	🟩	0.9040726902
Optimal	🟩			🟩	🟧	🟧		0.6795203804
DQN	🟩			🟩	🟧	🟧		0.6795203804
Rainbow	🟩			🟩	🟧	🟧		0.6795203804
IQ	🟩			🟩	🟧	🟧		0.6795203804

REGION	RegionKey	Name	Comment	Costs
HillClimb	🟩	🟩		0.02425475543
AutoPart	🟩	🟩		0.02425475543
Navathe	🟩	🟩		0.02425475543
O2P	🟩	🟩		0.02425475543
Optimal	🟩	🟩		0.02425475543
DQN	🟩	🟩		0.02425475543
Rainbow	🟩	🟩		0.02425475543
IQ	🟩	🟩		0.02425475543

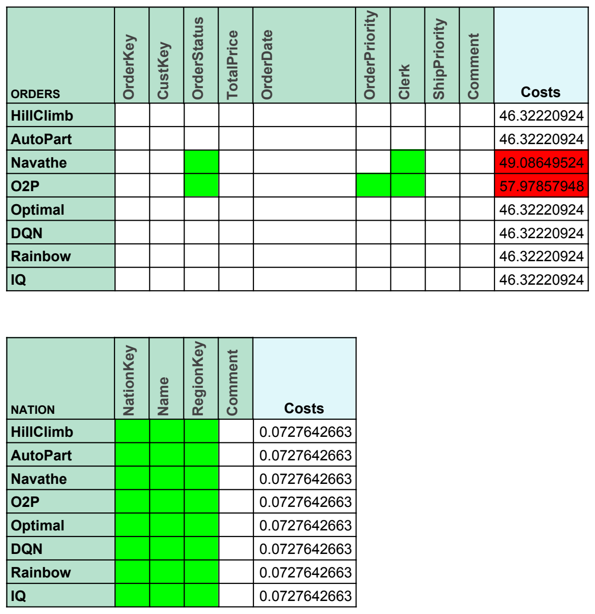
Figure 5.12: Supplier and Region tables with corresponding costs for each algorithm



PART	PartKey	Name	Mfgr	Brand	Type	Size	Container	RetialPrice	Comment	Costs
HillClimb										8.693756114
AutoPart								🟩	🟩	8.693756114
Navathe			🟩						🟩	🟥 9.36257337
O2P			🟩	🟧			🟧		🟩	🟥 9.529689538
Optimal								🟩	🟩	8.693756114
DQN										8.693756114
Rainbow										8.693756114
IQ										8.693756114

PARTSUPP	PartKey	SuppKey	AvailQty	SupplyCost	Comment	Costs
HillClimb	🟩	🟩				6.000222826
AutoPart	🟩	🟩				6.000222826
Navathe	🟩	🟩	🟩	🟧	🟧	🟥 65.55927989
O2P	🟩	🟩	🟩	🟧	🟧	🟥 65.55927989
Optimal	🟩	🟩				6.000222826
DQN	🟩	🟩				6.000222826
Rainbow	🟩	🟩				6.000222826
IQ	🟩	🟩				6.000222826

Figure 5.13: Part and PartSupp tables with corresponding costs for each algorithm



ORDERS	OrderKey	CustKey	OrderStatus	TotalPrice	OrderDate	OrderPriority	Clerk	ShipPriority	Comment	Costs
HillClimb										46.32220924
AutoPart										46.32220924
Navathe			🟩				🟩			🟥 49.08649524
O2P			🟩			🟩	🟩			🟥 57.97857948
Optimal										46.32220924
DQN										46.32220924
Rainbow										46.32220924
IQ										46.32220924

NATION	NationKey	Name	RegionKey	Comment	Costs
HillClimb	🟩	🟩	🟩		0.0727642663
AutoPart	🟩	🟩	🟩		0.0727642663
Navathe	🟩	🟩	🟩		0.0727642663
O2P	🟩	🟩	🟩		0.0727642663
Optimal	🟩	🟩	🟩		0.0727642663
DQN	🟩	🟩	🟩		0.0727642663
Rainbow	🟩	🟩	🟩		0.0727642663
IQ	🟩	🟩	🟩		0.0727642663

Figure 5.14: Nation and Orders tables with corresponding costs for each algorithm

Here individual partitions are coloured with white. The columns inside the same partitions have the same color.

Here Optimal means the Brute-Force algorithm offered and implemented by [JPPD]. Theoretically, Optimal should return the best partition in terms of partitions and cost.

It is noticeable that "Top-down" algorithms (Navathe and O2P) have the highest cost in all cases except for small tables (Region and Nation), and are often not converging to the optimal. As expected, since our agents converged based on HillClimb expert in the cases (1.a, 1.b) then the partitions and the cost are all the same as in HillClimb.

5.2.2 Inference times

For this experiment we made 1000 repetitions and take average time for running the algorithms vs. inference. This was true for all cases except optimal ver the lineitem and customer table, where we only did 5 and 100 repetitions, respectively, due to the high latency of the operation.

Our observations show that for most cases traditional algorithms still perform very fast in comparison to our implemented RL agents. One reason for it could be that we are not using an implementation of the neural network that is optimized for inference. Another aspect is that state-keeping and transitioning for each step adds overheads.

We also found that as models becomes more complex inference times do not increase, but the time taken for traditional algorithms does.

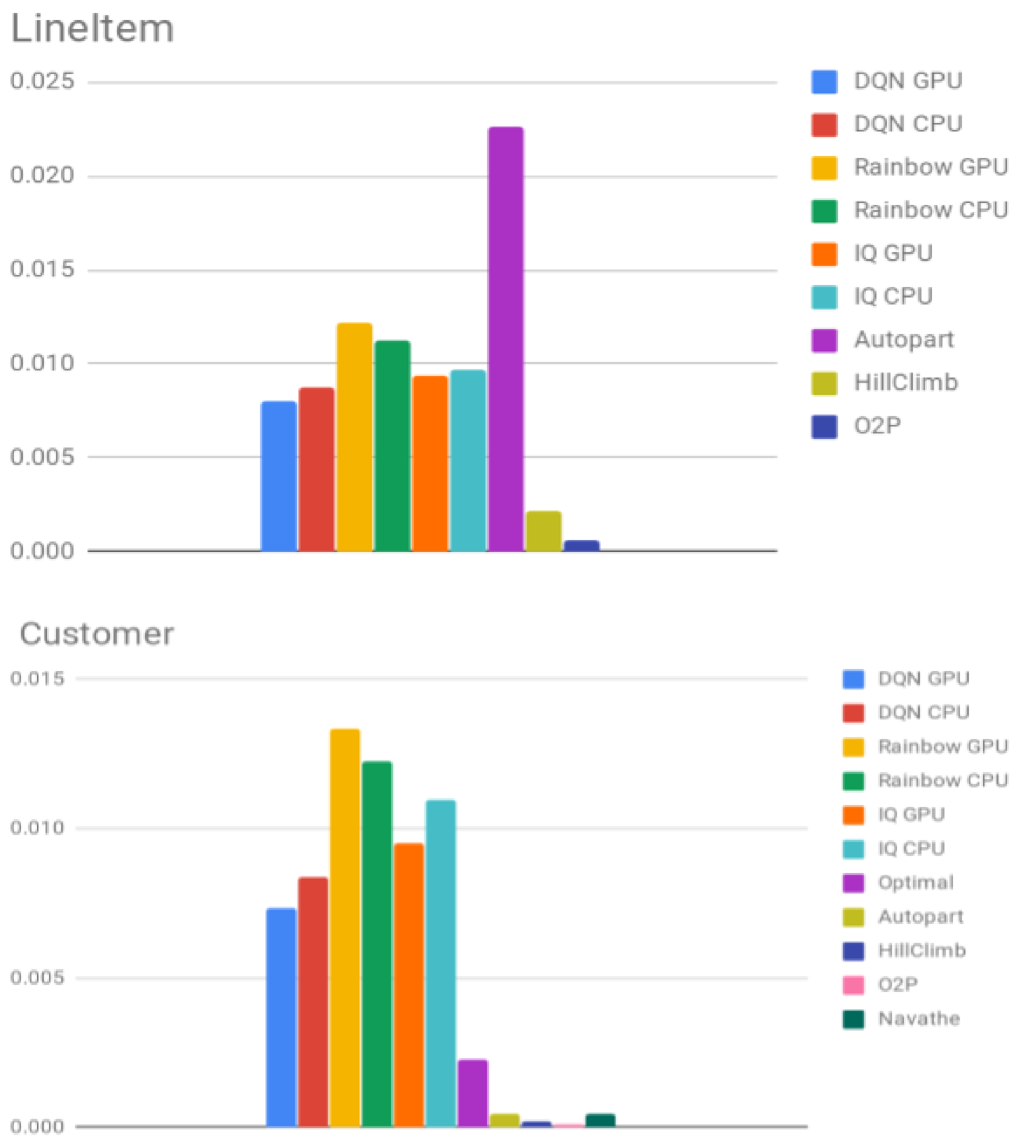


Figure 5.15: Inference times for LineItem and Customer tables (seconds)

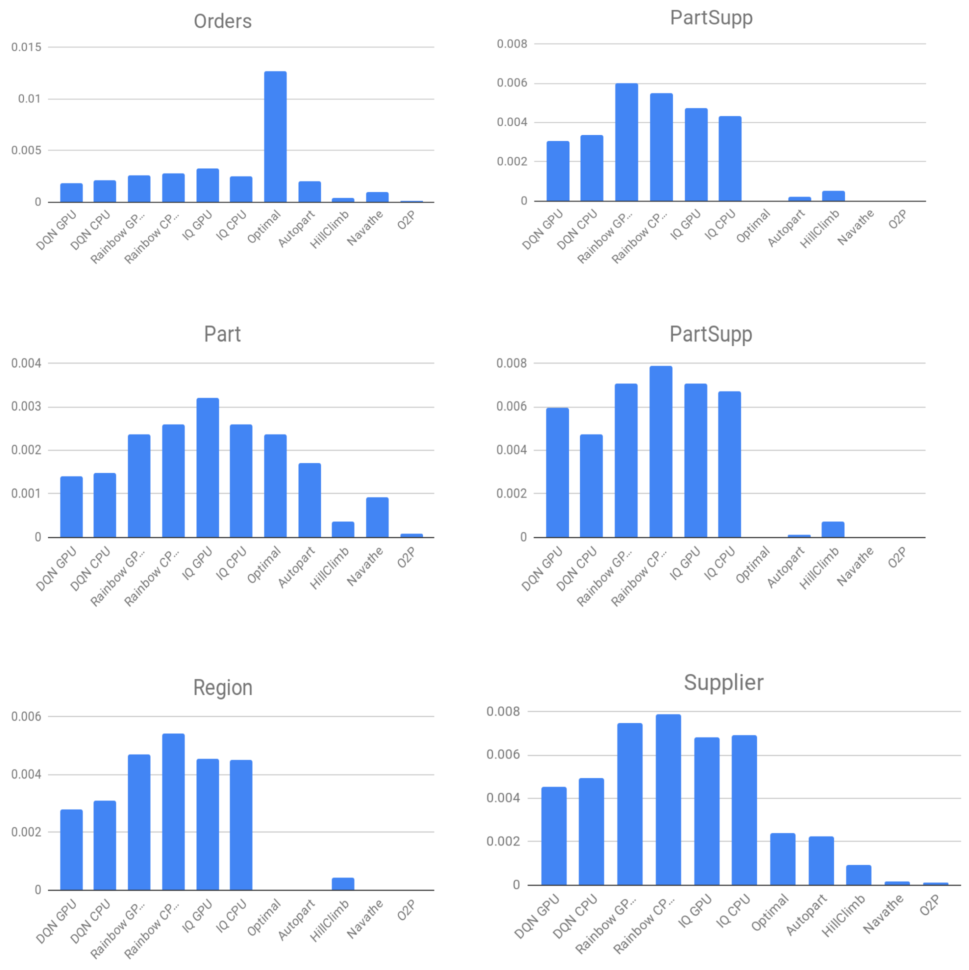


Figure 5.16: Inference times for the remaining 6 tables (seconds)

In (Section B.3) we put the complete inference time table. There we see that a high number of attributes (i.e. the lineitem table) DRL solutions are able to outperform Navathe, O2P and AutoPart in terms of optimization time.

5.3 Summary

In this chapter we discuss the evaluation and results of our experiments. First we checked the possibility of our agents to learn in three cases with different hyper-parameters. We observe that in the first two cases agents are able to converge relatively easy. But in the most complex case we evaluated, we did not find a 100% convergence. In spite of this result we are able to find some cases where our solution outperforms, during training, HillClimb.

In the second part of chapter we compared partitions and the cost need to achieve these partitions. We found that our "top-down" algorithms result in less efficient solutions in comparison to "bottom-up" ones, in most cases. We also observe that partitions generated by our learning agents are exactly the same as for the expert HillClimb algorithm, matching the optima. This proves that our agents correctly converge and at simple cases are able to mimic the behavior of an expert.

We tested inference times for each table and its default workload. Results show that improvements in the implementation of the inference procedure will be required (including improvements for state-transitions) to make DRL-based solutions also competitive in terms of inference time.

6. Related Work

In this chapter we give an overview on state-of-the-art works of applying RL approaches to different database optimization problems. Generally,reviewed papers are very recent, mostly published in 2018, since research on the use of RL and especially DRL methods for optimizations of database operations seems to be only started.
(Table 6.1) presents an overview of the works we found regarding this topic. Below, we discuss some of these works in more detail.s

Area of Application	**Task**	**Existing Work**
Storage Engine	Index Selection	[BLC$^+$15, SSD18]
	Data Partitioning	[DPP$^+$18]
Query Engine	Join-order Enumeration	[MP18]
	Operator-variant Selection	[KBCG18]
	Query-plan Optimization	[OBGK18]
	Adaptive Query Processing	[TSJ08]

Table 6.1: Categorization of work using RL for database management

- **Index selection** - Two works from different research groups propose applying RL to aid in the automatic index selection task. Basu et al. offered a solution called COREIL for index selection, without previous knowledge about the cost model[BLC$^+$15]. The cost model is learned by way of a more general RL formulation. In their pioneering work authors focus on indexes, but they provide a design that is sufficiently generic to be applied to other physical design tasks.

  Building Blocks: In their proposal, one logical database schema R could be described as a set of physical database configurations S, where two configurations s and s' have only a difference in terms of indexes, views, replications and other parameters. Despite these differences the result of queries and updates are logically the same for s and s'. The task of their work is the determination of the cost of changing from one physical configuration to another, alongside the benefits of the action. As a workload set Q they considered a stack of queries and updates. The cost of execution of query q on the configuration s is denoted as cost(s,q). They modeled query q as a random variable which only becomes observable at the time t. The environment for training the model is initialized with a randomly chosen initial configuration, s_0 . Authors describe the transition process as a four step algorithm: Transitions from one state to another are deterministic. Penalties and rewards, however, are considered stochastic and uncertain.

Based on the formulation described above, the RL problem is framed as that of finding the sequence of configurations that minimizes the sum of per-stage costs. To find an optimal policy, authors adopted a policy iteration approach.

Evaluation: Authors implemented a prototype and compare the results with the WFIT algorithm, which they identify as state-of-the-art for automatic index selection. As a dataset they used the TPC-C specification with scale factor 2, with transactions consisting of 5 queries (mixing updates and queries) from an OLTP-bench generated workload. Authors evaluate the process from 0 to 3000 transactions. As database authors used IBM DB2.

Results shows that despite COREIL converges slower than WFIT, it can achieve less processing time after convergence (at around 2000 transactions) with only a small amount of peaks (i.e. cases where it does not achieve better runtime per transaction than WFIT). After convergence COREIL does not induce higher costs (to create indexes) than does WFIT. However the runtime for making an inference exceeds WFIT by at least an order of magnitude. However, COREIL is more effective than WFIT because it creates shorter-compound-attribute indexes. Authors conclude that despite COREIL uses a non existent cost-model beforehand, learning it iteratively, its performance is comparable with WFIT. Finally, it should also be noted that the slow convergence of COREIL is feasibly improvable by starting with cost approximations that are close to the real case.

More recently, Sharma et al. offered the concept of NoDBA [SSD18] which also seeks to automate physical design for indexing, based on Deep Reinforcement learning approaches.

Building Blocks: They assume a database schema S which contains n columns, and the maximum number of indexes is k (k<n). The workload has almost the same meaning as proposed in Basu et al., except for taking more queries into consideration (i.e., a set of n queries, each described by the usage of certain columns, and the selectivity of their predicates on those columns). As input parameters of Neural networks they proposed the workload and the current configuration of indexes separately: $I_{workload}$ and $I_{indexes}$ accordingly. $I_{workload}$ is a matrix of n*m where n is the number of queries, and m is number of columns in the database schema. As an element of the matrix they propose the selectivity function Sel(Q_i;C_j), whose arguments are: the i-th query and the j-th column. Selectivities are computed for every query once per input. $I_{indexes}$ is a bitlist of size m and

shows on which columns indexes exist for a given configuration. They limited the number of actions by considering only single actions using episodic RL. On each episode initially there is no index, but on final configurations there is a maximum number of actions (k), and a number of indexes. This process continues iteratively. For each configuration they propose their own reward function, but the reward could also be an estimate using the cost function like EXPLAIN, for a concrete DBMSs. After several trial and error tests, they select these hyperarameters: k = 3, number of hidden layers = 4, number of neurons for each hidden layer = 8, activation function = RELU, output layer = SOFTMAX, learning algorithm = CEM. For the evaluation they categorize random workloads as training and define some simple test workloads. The test workloads are divided into three groups and the query complexities is increased for each group, according to the columns participating in the queries.

Evaluation: Their evaluation used the Postgres database with the TPC-H dataset. Authors compared their solution with that of the edge cases (NoIndex and IndexedAll). Results show that the runtime of NoDBA is always better for W1 and W2 with improvements of one order of magnitude, and is almost similar with IndexedAll in the W3 workload.

- **Join-order optimization**: Marcus et al. offered a concept called ReJOIN [MP18], which is based in deep reinforcement learning and seeks to help decision making for finding optimal join orders. This is important since different join orders are possible for a query, and they lead to different costs. For this problem the traditional approach consists on evaluating the cost of different, by making assumptions on the selectivity of join predicates (based on statistics) for all pairs of joins. This is called the Seeliger-style join order evaluation.

  In ReJOIN this problem is tackled differently: By giving SQL queries as input to ReJOIN, the RL agent should be able to return the cheapest ordering for execution, based on a pre-explored search space.

  Building Blocks: Fig. Figure 6.1 shows the ReJOIN framework. In their approach, a join tree was considered as a binary tree where each node represents one relation. This constitutes the state of the problem. Each query which is sent to ReJOIN and also to the benchmarking optimizer is considered as a complete episode. In each step of the episode ReJOIN selects one pair of tables to join, receives a reward, and moves to a state where the subtree is without the chosen pair.

  ReJOIN iteratively learns through several episodes. By state authors consider the possible subtrees of the input join tree. Actions in ReJOIN are limited to the merging of two subtrees. An episode ends when all relations have become a part of the binary join tree.

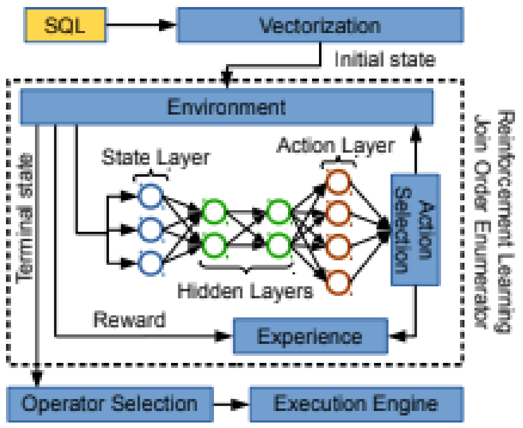


Figure 6.1: The ReJOIN framework [MP18]

The RL model implemented in ReJoin was Proximal Policy Optimization algorithm with a deep reinforcement learning technique; the number of hidden layers are two, each with 128 neurons with a rectified linear unit activation function (ReLU).

Evaluation: The evaluation was done on the so called Join Order Benchmark (JOB) over the IMDB dataset. Authors report that the training converges at around of 10k episodes with each episode consisting of a randomly chosen query from the same test dataset. The Evaluation contains three metrics: Cost of join orderings, Latency of the generated plans and optimization time. Results show that ReJOIN outperforms PostgreSQL's optimizer in terms of the cost of join orderings and generates better query plans for all queries over PostgreSQL. In average, query plan costs generated by ReJOIN were 20% cheaper than in PostgreSQL. In comparison to PostgreSQL, the optimization time of ReJOIN is better starting from 8 relations onwards. The main problems with ReJOIN are policy update overhead and latency optimization which will be investigated in future work.

- **Query-plan optimization**

  Ortiz et al. proposed a deep-reinforcement approach for query optimization [OBGK18]. Authors offer a DRL method for learning the properties of subquery states (such as cardinality estimates, and others) where queries have been built incrementally. The main challenge is the representation of the state transition function, as explained by authors.
  Authors propose that the main problem of using DRL for database-related issues is the representation of queries and models. The problem is: the complexity of model could make solutions impractical since it will require too many training examples. Authors develop an incremental model which generates a short representation of each the intermediate results for each subquery. The model predicts the properties of subquery by subquery itself and one database operation considering as an action in terms of RL. As the second contribution authors use this representation via

Reinforcement learning to improve query plan enumeration.

Building Blocks: Initially the model has a Query and the state of the database as input and it incrementally executes query plans via series of state transitions. Here are several assumptions considered by authors: Initial state t - entire database, action is some query operator and selecting using RL, $t+1$ the next state. Each state except initial represents one subquery result. NN_{st} - state transition function which is recursive and takes as input subquery and action at time t, and produce subquery for time step t+1. Learning process of State Transition Function shown on Fig. Figure 6.2.

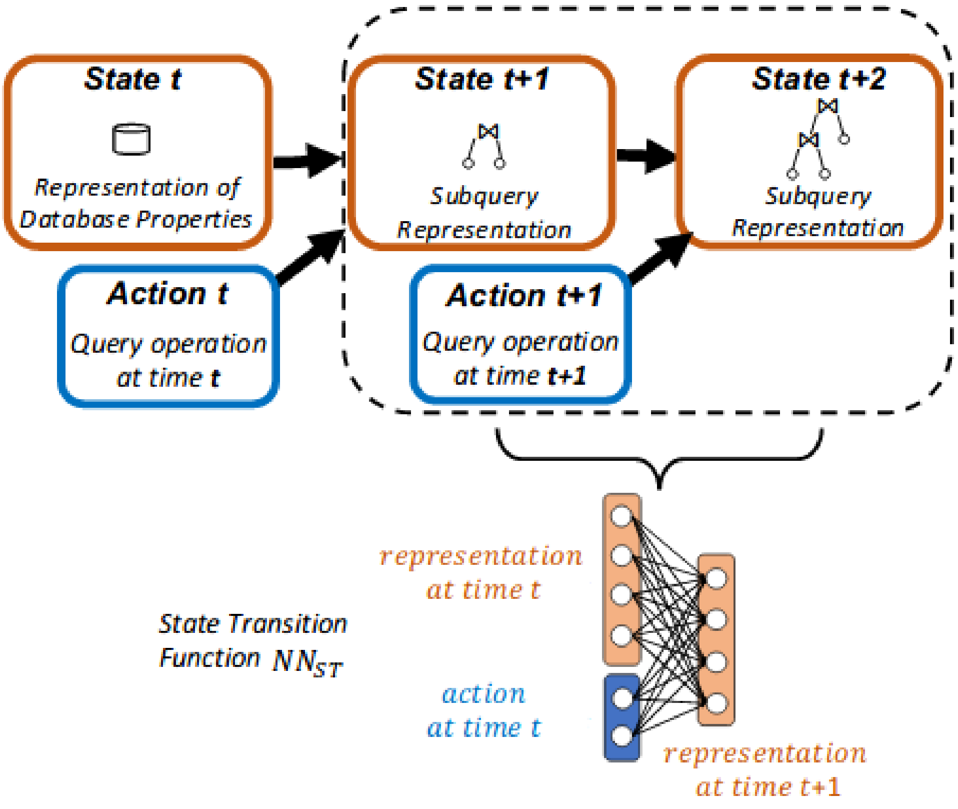

State t
Representation of Database Properties
Action t
Query operation at time t
State t+1
Subquery Representation
Action t+1
Query operation at time t+1
State t+2
Subquery Representation
State Transition Function NN_{ST}
representation at time t
action at time t
representation at time t+1

Figure 6.2: State representation. RL schema[OBGK18]

Here the model takes as input pairs (h_t, a_t) where h_t - is vector representation of a subquery, and a_t - is a relational operation on h_t. h_t is a latent representation that model learns itself. $NN_{observed}$ function is using for mapping observed values to subquery representation. For learning initial state authors offer NN_{init} function which takes some properties of database and randomly one database operator as an action. Authors combine all these models and train them together.

Evaluation: For evaluation authors choose the IMDB database since this database is more realistic and there are definitely correlation and skew between columns. As an initial experiment authors initialized x_0 with properties from the IMDB dataset. As an initial action a_0 authors used conjunctive selection operator. Authors generate and used 15k queries for training the model and 5k for testing purposes. The solution contains 50 hidden nodes for one hidden layer. Authors used Stochastic gradient-descent based algorithm for optimization, and a learning rate of 0.01. Authors compared the cardinality estimation with the value from SQL Server. Results show that in case of 3 columns (m=3) there are heavy errors in short epochs (less than 3) but their approach starts out-performing SQL server's estimation after 6th epochs only. There is an observation that in case of increasing the number of columns convergence will happen slowly. But even in that case it is shown to outperform SQL Server on the 9th epoch.

With this we conclude our review of related work in applying deep reinforcement learning to database tasks.

7. Conclusion and Future work

In this chapter, we summarize results of this thesis and give directions for further research.

7.1 Conclusion

In this work we introduced an early approach for using state-of-the art DRL methods for the database vertical partitioning task. As a benchmark we used the TPC-H workbench. Our main contributions are:

1. Adaptation and unification of four traditional learning algorithms of vertical partitioning, REST interface between learning RL models and these algorithms
2. Implementation and design of our own RL environment
3. Design of suitable reward function, proper action representation
4. Adaptation and integration of three existing RL agents (DQN, Rainbow, Implicit quantile) from Dopamine framework for the given task.
5. Test combination of different hyper-parameters during the training process, and evaluation of the inference process.
6. Validation of the feasibility of the current approach, though questions remain about two core issues: making the inference process more competitive, and scaling-up to training over a larger amount of workloads.
7. Discussed evaluation results
8. Brief discussion of recent works applying RL methods for database optimization problems.

We proposed two research questions. In the first we tested how learning models converge considering three different case scenarios. According to our evaluation for the case 1.a (single table single workload) the iterations needed for convergence strictly depend from the number of steps needing to get final partition. We found that Implicit Quantile agent shows better results in terms of convergence. In case 1.b (mixed pair of tables and workloads) we tested several hyper-parameters. We only found reasonable improvements in terms of convergence while we use the action pruning heuristic. For the other hyper-parameters convergence happens relatively similar despite slightly better results noticeable in case of update horizon = 3, ϵ-greedy, and soft-update combinations. Here again, Implicit Quantile agent outperforms DQN and Rainbow agents.

In the most complex case 1.c (single table, random workload) we tested only one case (CUSTOMER table with random workloads) using Implicit quantile agent since this case is much harder in comparison to previous two cases and we choose the best combinations of hyper-parameters from previous cases. Unfortunately, we did not reach absolute convergence in this case and this case needs additional investigation.

Our second research question was dedicated to inference. We found that in most cases the solutions of algorithms following a "Top-down" approach were less efficient than those of algorithms following a "Bottom-up" approach. This could be partly caused by the workload itself, which could favor columnar approaches, making bottom up approaches start from a better position. Since, our learning agents follows HillClimb algorithm as an expert, the cost is the same as in HillClimb. Also, generated partitions were shown to be the same as HillClimb, and the same as the optimal.

7.1.1 Threats to validity

- **Non-machine-learning algorithms**

  These algorithms are available as an open-source project [JPPD] therefore their implementation could contain some minor bugs. During our implementation we fixed several of them (i.e., mostly in the implementation of optimal) but there are always possibility that we miss some bugs.

- **Simplification in non-machine-learning algorithms**

  In our experiments we considered algorithms with unified settings. Since we do not differentiate between projection and selection queries our result partitions could be different in case of these query types will be considered. This is also true regarding data replication as well as data granularity. We use meta-data of TPC-H workbench, but in real DBMS we need to consider effect of File system, database blocks and other characteristics of real database systems.

- **Heuristics regarding pruning actions**

  For the fast convergence, we made a heuristic of pruning actions. Despite without pruning of actions our models also able to converge but it took 2 times more iterations. We are not able to prove if it is correct to use this pruning technique or this could considering as a bias.

7.2 Future work

In this thesis we show feasibility of applying RL based methods for Database vertical partitioning optimization task. But we need to admit that we do a lot of unifications, simplifications and at the current stage the results of our research is not practical. Therefore, we need to extend this research in the several directions:

- **Dedicated study for generalization**

  As shown, training over a stream of randomly generated cases is a big challenge for our agents. As future work, this needs to be studied with care, assessing the precise limits of the current DRL models with regards to their ability to learn for general cases, and how this aspect could be enhanced by different techniques.

- **Performance improvements to inference process**

  In our study we report inference times that are competitive with the traditional algorithms, but only manage to outperform some of them for large tables. Further optimization to the inference process (which apart from inference encompasses state transitions and initialization) is highly relevant to create a more competitive solution.

- **Better reward engineering**

  In our implementation we need an “expert” algorithm’s best value and action sequences for calculation reward on each step. This means that we tightly depend on expert algorithm. As a future we could consider to avoid this “expert” help, perhaps we could chose a static configuration as a baseline for normalization.

- **Other RL approaches**

  In our research we employed value-based DRL algorithms offered by Dopamine. But probably, for the given task policy-based approaches are more suitable. Therefore for a future we could employ agents based on Policy gradient methods (DPPO, TRPO etc.). Similarly, DRL algorithms for large discrete action spaces would be worthwhile to study.

- **Deep learning and better feature representation**

  In our work we tightly rely on the NN architecture offered by Dopamine framework. Since offered convolutional NN works better for the image input, research could be continue in the direction of choosing more suitable network structures like recurrent and recursive networks (RNN) for a given problem. Also correctly selected features and hyper-parameters could make the model more stable and universal.

- **Integration with real world DBMS system**

  RL introduces new possibilities for DBMSs to expose their design and internals to learning models. Action engineering is required to decide on the granularity, the learning goal, and the way in which RL can be leveraged for DBMS design. This is a specially pertinent research area for self-driving database designs. More specialized open questions, pertaining to each optimization problem could be considered.

A. Structural views of using agents, derived from Tensorboard

Here we place schematic overviews of the used RL agents, which we derived from Tensorboard.

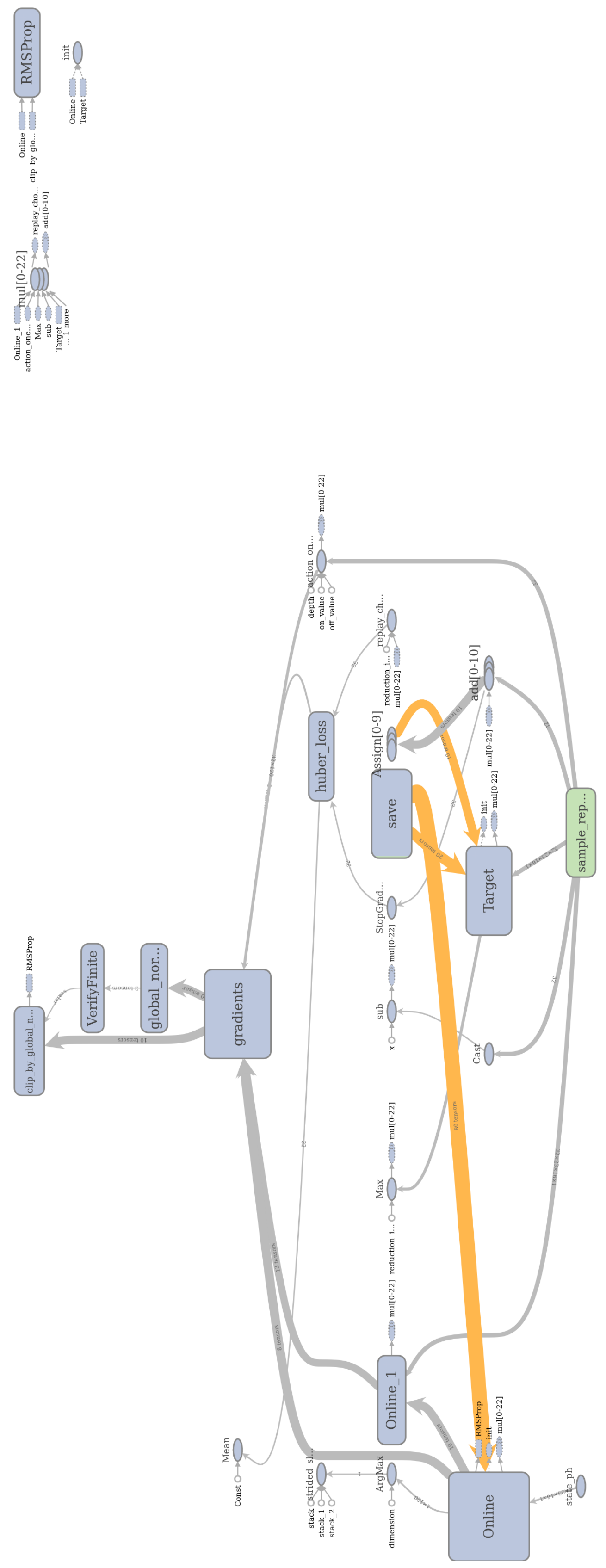


Figure A.1: DQN agent, structural view derived from Tensorboard.

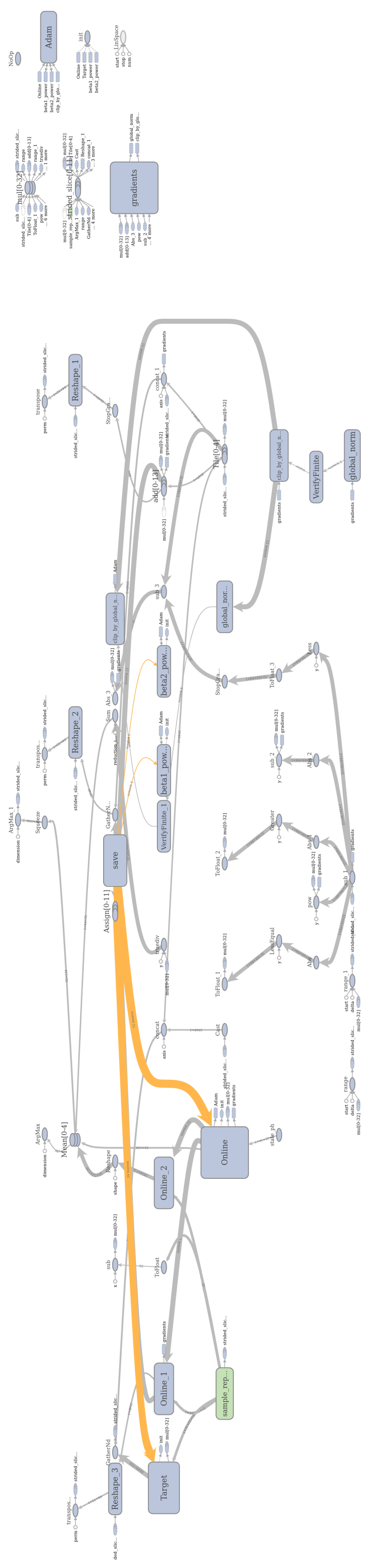


Figure A.2: Implicit quantile agent, structural view derived from Tensorboard.

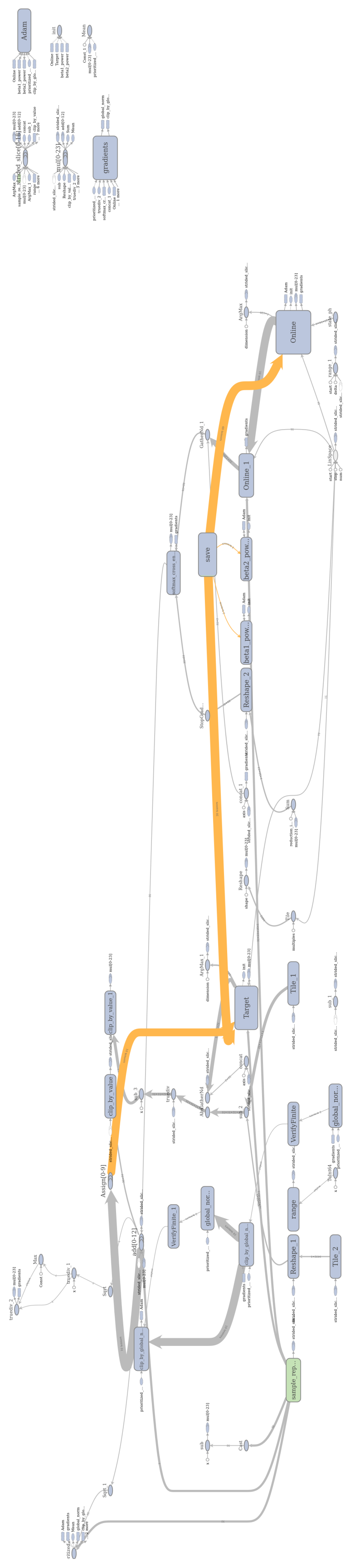


Figure A.3: Rainbow agent, structural view derived from Tensorboard.

B. Hyper-parameters used for each agent

In this section we provide hyper-parameters used for each agent during learning. In Dopamine these parameters are configurable and storing in .gin template files. Majority of used parameters provided by Dopamine and used as default.

B.1 DQN agent parameters

```
# Hyperparameters follow Dabney et al. (2018), but we modify as necessary to
# match those used in Rainbow (Hessel et al., 2018), to ensure apples-to-
apples
# comparison.
import gin.tf.external_configurables
DQNAgent.gamma = 0.99
DQNAgent.min_replay_history = 2000  # agent steps
DQNAgent.update_period = 4
DQNAgent.target_update_period = 1 # agent steps, was 400
DQNAgent.epsilon_train = 0.01
DQNAgent.epsilon_eval = 0.0
DQNAgent.epsilon_decay_period = 1000000  # agent steps
DQNAgent.tf_device = '/gpu:0'  # use '/gpu:*' for non-GPU version
DQNAgent.optimizer = @tf.train.RMSPropOptimizer()
DQNAgent.learning_from_demo = False
tf.train.RMSPropOptimizer.learning_rate = 0.0000625
tf.train.RMSPropOptimizer.decay = 0.0
tf.train.RMSPropOptimizer.momentum = 0.0
tf.train.RMSPropOptimizer.epsilon = 0.00015
tf.train.RMSPropOptimizer.centered = True
```

```
WrappedReplayBufferLocal.replay_capacity = 1000000
WrappedReplayBufferLocal.batch_size = 32
run_experiment.Runner.game_name = 'GridFormationEnvSimple-v0'
# Sticky actions with probability 0.25, as suggested by (Machado et al., 2017).
run_experiment.Runner.sticky_actions = False
run_experiment.Runner.training_steps = 1000  # agent steps
run_experiment.Runner.evaluation_steps = 200 # agent steps
run_experiment.Runner.max_steps_per_episode = 25  # agent steps
run_experiment.Runner.num_iterations = 1000
```

B.2 Implicit Quantile agent parameters

```
# Hyperparameters follow Dabney et al. (2018), but we modify as necessary to
# match those used in Rainbow (Hessel et al., 2018), to ensure apples-to-
apples
# comparison.
import gin.tf.external_configurables

IQ.kappa = 1.0
IQ.num_tau_samples = 64
IQ.num_tau_prime_samples = 64
IQ.num_quantile_samples = 32
RainbowAgent.num_atoms = 51
RainbowAgent.vmax = 200.
RainbowAgent.gamma = 0.99
RainbowAgent.update_horizon = 3
RainbowAgent.min_replay_history = 2000  # agent steps
RainbowAgent.update_period = 4
RainbowAgent.target_update_period = 1 # agent steps, was 400
RainbowAgent.epsilon_train = 0.01
RainbowAgent.epsilon_eval = 0.0
RainbowAgent.epsilon_decay_period = 1000000  # agent steps
# IQN currently does not support prioritized replay.
RainbowAgent.replay_scheme = 'uniform'
RainbowAgent.tf_device = '/gpu:0'  # use '/cpu:*' for non-GPU version
RainbowAgent.optimizer = @tf.train.AdamOptimizer()
IQ.learning_from_demo = False
IQ.boltzmann=False
IQ.prune_actions=True
# Note these parameters are different from C51's.
tf.train.AdamOptimizer.learning_rate =  0.0000625
tf.train.AdamOptimizer.epsilon = 0.00015
WrappedPrioritizedReplayBufferLocal.replay_capacity = 10000
WrappedPrioritizedReplayBufferLocal.batch_size = 32
```

```
run_experiment.Runner.game_name = 'GridFormationEnvSimple-v0'
# Sticky actions with probability 0.25, as suggested by (Machado et al., 2017).
run_experiment.Runner.sticky_actions = False
run_experiment.Runner.training_steps = 1000  # agent steps
run_experiment.Runner.evaluation_steps = 200 # agent steps
run_experiment.Runner.max_steps_per_episode = 25  # agent steps
run_experiment.Runner.num_iterations = 150
```

B.3 Rainbow agent parameters

```
# Hyperparameters follow Dabney et al. (2018), but we modify as necessary to
# match those used in Rainbow (Hessel et al., 2018), to ensure apples-to-
apples
# comparison.
import gin.tf.external_configurables
Rainbow.num_atoms = 51
Rainbow.vmax = 200.
Rainbow.gamma = 0.99
Rainbow.min_replay_history = 2000  # agent steps
Rainbow.update_period = 4
Rainbow.target_update_period = 1  # agent steps
Rainbow.epsilon_train = 0.01
Rainbow.epsilon_eval = 0.0
Rainbow.epsilon_decay_period = 1000000  # agent steps
Rainbow.replay_scheme = 'prioritized'
Rainbow.tf_device = '/gpu:0'  # use '/cpu:*' for non-GPU version
Rainbow.optimizer = @tf.train.AdamOptimizer()
Rainbow.learning_from_demo = False
# Note these parameters are different from C51's.
tf.train.AdamOptimizer.learning_rate =  0.0000625
tf.train.AdamOptimizer.epsilon = 0.00015
WrappedReplayBufferLocal.replay_capacity = 1000000
WrappedReplayBufferLocal.batch_size = 32
run_experiment.Runner.game_name = 'GridFormationEnvSimple-v0'
# Sticky actions with probability 0.25, as suggested by (Machado et al., 2017).
run_experiment.Runner.sticky_actions = False
run_experiment.Runner.training_steps = 1000  # agent steps
run_experiment.Runner.evaluation_steps = 200 # agent steps
run_experiment.Runner.max_steps_per_episode = 25  # agent steps
run_experiment.Runner.num_iterations = 1000
```

C. Inference times

Algorithms	LineItem	Customer	Orders	Part	PartSupp	Nation	Region	Supplier
DQN GPU	8.02E-03	7.34E-03	1.83E-03	1.41E-03	3.06E-03	5.96E-03	2.81E-03	4.54E-03
DQN CPU	8.72E-03	8.35E-03	2.14E-03	1.48E-03	3.39E-03	4.72E-03	3.09E-03	4.94E-03
Rainbow GPU	1.22E-02	1.34E-02	2.56E-03	2.38E-03	6.00E-03	7.05E-03	4.69E-03	7.45E-03
Rainbow CPU	1.12E-02	1.23E-02	2.75E-03	2.61E-03	5.50E-03	7.88E-03	5.42E-03	7.88E-03
IQ GPU	9.41E-03	9.53E-03	3.25E-03	3.21E-03	4.75E-03	7.06E-03	4.54E-03	6.84E-03
IQ CPU	9.68E-03	1.10E-02	2.53E-03	2.61E-03	4.35E-03	6.71E-03	4.49E-03	6.93E-03
Optimal	1.40E+03	2.28E-03	1.27E-02	2.36E-03	2.52E-05	1.14E-05	4.15E-06	2.42E-03
Autopart	2.26E-02	4.82E-04	2.06E-03	1.72E-03	1.99E-04	1.44E-04	2.48E-05	2.27E-03
HillClimb	2.21E-03	2.05E-04	4.17E-04	3.69E-04	5.38E-04	7.26E-04	4.46E-04	9.17E-04
Navathe	2.57E-01	4.70E-04	9.84E-04	9.19E-04	3.82E-05	2.34E-05	9.98E-06	1.85E-04
O2P	6.00E-04	1.08E-04	1.02E-04	9.21E-05	2.22E-05	2.83E-05	8.45E-06	9.48E-05

Figure C.1: Inference times for default TPC-H table-workload pairs (seconds)

Bibliography

[72] Problem decomposition and data reorganization by a clustering technique. *Oper. Res.*, 20(5):993–1009, October 1972. (cited on Page xi, 14, and 15)

[ABCN06] Sanjay Agrawal, Nicolas Bruno, Surajit Chaudhuri, and Vivek R Narasayya. Autoadmin: Self-tuning database systemstechnology. *IEEE Data Eng. Bull.*, 29(3):7–15, 2006. (cited on Page 2)

[ANY04a] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370. ACM, 2004. (cited on Page 11)

[ANY04b] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD Conference*, 2004. (cited on Page iii)

[Ape88] Peter M. G. Apers. Data allocation in distributed database systems. *ACM Trans. Database Syst.*, 13:263–304, 1988. (cited on Page 12)

[BAA12a] Renata Borovica, Ioannis Alagiannis, and Anastasia Ailamaki. Automated physical designers: What you see is (not) what you get. In *Proceedings of the Fifth International Workshop on Testing Database Systems*, DBTest '12, pages 9:1–9:6, New York, NY, USA, 2012. ACM. (cited on Page iii)

[BAA12b] Renata Borovica, Ioannis Alagiannis, and Anastasia Ailamaki. Automated physical designers: what you see is (not) what you get. In *Proceedings of the Fifth International Workshop on Testing Database Systems*, page 9. ACM, 2012. (cited on Page 2)

[BDM17] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *CoRR*, abs/1707.06887, 2017. (cited on Page 29)

[Bel09] Ladjel Bellatreche. *Optimization and Tuning in Data Warehouses*, pages 1995–2003. Springer US, Boston, MA, 2009. (cited on Page iii)

[BLC$^+$15] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. Cost-model oblivious database tuning with reinforcement learning. In *Proceedings, Part I, of the 26th International Conference on Database and Expert Systems Applications - Volume 9261*, DEXA 2015, pages 253–268, 2015. (cited on Page 69)

[BNVB13] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013. (cited on Page 4)

[CMG$^+$18] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G Bellemare. Dopamine: A research framework for deep reinforcement learning. *arXiv preprint arXiv:1812.06110*, 2018. (cited on Page 4 and 34)

[CY90] D. Cornell and P. Yu. An effective approach to vertical partitioning for physical design of relational databases. *IEEE Transactions on Software Engineering*, 16:248–258, 02 1990. (cited on Page 11)

[DAEvH$^+$15] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015. (cited on Page 4)

[DHJ$^+$07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007. (cited on Page 10)

[DOSM18] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning. *CoRR*, abs/1806.06923, 2018. (cited on Page xi, 29, and 30)

[DPP$^+$18] Gabriel Campero Durand, Marcus Pinnecke, Rufat Piriyev, Mahmoud Mohsen, David Broneske, Gunter Saake, Maya S Sekeran, Fabián Rodriguez, and Laxmi Balami. Gridformation: Towards self-driven online data partitioning using reinforcement learning. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, page 1. ACM, 2018. (cited on Page 69)

[DY$^+$14] Li Deng, Dong Yu, et al. Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014. (cited on Page 25)

[FLHI+18] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, Joelle Pineau, et al. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018. (cited on Page 3 and 24)

[GCNG16] Vyacheslav Galaktionov, George Chernishev, Boris Novikov, and Dmitry Grigoriev. Matrix clustering algorithms for vertical partitioning problem: an initial performance study. In *DAMDID/RCDL*, 2016. (cited on Page iii and 12)

[HGS16] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2094–2100. AAAI Press, 2016. (cited on Page 28)

[HMvH+17] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017. (cited on Page 30)

[HN79a] Michael Hammer and Bahram Niamir. A heuristic approach to attribute partitioning. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 93–101, New York, NY, USA, 1979. ACM. (cited on Page 2)

[HN79b] Michael Hammer and Bahram Niamir. A heuristic approach to attribute partitioning., 01 1979. (cited on Page 11)

[HP03] Richard A. Hankins and Jignesh M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *VLDB*, 2003. (cited on Page 13 and 19)

[JD12] Alekh Jindal and Jens Dittrich. Relax and let the database do the partitioning online. In Malu Castellanos, Umeshwar Dayal, and Wolfgang Lehner, editors, *Enabling Real-Time Business Intelligence*, pages 65–80, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. (cited on Page 11, 13, and 16)

[JPPD] Alekh Jindal, Endre Palatinus, Vladimir Pavlov, and Jens Dittrich. A comparison of knives for bread slicing. (cited on Page iii, 2, 4, 11, 12, 14, 33, 37, 39, 65, and 76)

[JQRD11] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. Trojan data layouts: Right shoes for a running elephant. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 21:1–21:14, New York, NY, USA, 2011. ACM. (cited on Page iii and 13)

[KBCG18] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. Cuttlefish: A lightweight primitive for adaptive query processing. *CoRR*, abs/1802.09180, 2018. (cited on Page 69)

[KS86] Henry F. Korth and Abraham Silberschatz. *Database System Concepts*. McGraw-Hill, Inc., New York, NY, USA, 1986. (cited on Page 1 and 10)

[Loh14] Guy Lohman. Is query optimization a “solved” problem. In *Proc. Workshop on Database Query Optimization*, page 13. Oregon Graduate Center Comp. Sci. Tech. Rep, 2014. (cited on Page 2)

[MKS$^+$13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. 2013. (cited on Page 27 and 28)

[MKS$^+$15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015. (cited on Page 27)

[MP18] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, page 3. ACM, 2018. (cited on Page xii, 69, 71, and 72)

[NCWD84a] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems*, 9:680–710, 1984. (cited on Page 9, 10, 13, and 14)

[NCWD84b] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4):680–710, December 1984. (cited on Page 11 and 13)

[OBGK18] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, page 4. ACM, 2018. (cited on Page xii, 69, 72, and 73)

[PA04a] S. Papadomanolakis and A. Ailamaki. Autopart: automating schema design for large scientific databases using data partitioning. In *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.*, pages 383–392, June 2004. (cited on Page xi, 13, 18, and 19)

[PA04b] Stratos Papadomanolakis and Anastassia Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, SSDBM '04, pages 383–, Washington, DC, USA, 2004. IEEE Computer Society. (cited on Page 11, 13, and 18)

[PAA$^+$17] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. Self-driving database management systems. In *CIDR*, 2017. (cited on Page 2)

[SB98] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. (cited on Page 21, 23, and 24)

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015. (cited on Page 28)

[SSD18] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. The case for automatic database administration using deep reinforcement learning. *ArXiv e-prints*, January 2018. (cited on Page 69 and 70)

[SW85] Domenico Sacca and Gio Wiederhold. Database partitioning in a cluster of processors. *ACM Transactions on Database Systems (TODS)*, 10(1):29–56, 1985. (cited on Page 10)

[TSJ08] Kostas Tzoumas, Timos Sellis, and Christian S Jensen. A reinforcement learning approach for adaptive query processing. *History*, 2008. (cited on Page 69)

[VAPGZ17] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM, 2017. (cited on Page 2)

[WD92] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992. (cited on Page 24)

[Wir00] Rüdiger Wirth. Crisp-dm: Towards a standard process model for data mining. In *Proceedings of the Fourth International Conference on the Practical Application of Knowledge Discovery and Data Mining*, pages 29–39, 2000. (cited on Page 5)