Deep Reinforcement Learning from Demonstrations for Automatic Database Index Selection

Author:
Fabian Rodriguez

March 2nd, 2020

Advisors:
M.Sc. Gabriel Campero Durand
Data and Knowledge Engineering Group

Prof. Dr. rer. nat. habil. Gunter Saake
Data and Knowledge Engineering Group
Rodriguez, Fabian:
Deep Reinforcement Learning from Demonstrations for Automatic Database Index Selection
Abstract

In recent years, the amount of data and the speed in which it is being created continues to grow exponentially. To keep pace with this trend, exhaustive research is being conducted for improving data management tools, with regards to aspects such as index selection, query processing and join order optimization.

Novel research has shown that Deep Reinforcement Learning (DRL) models can excel in solving several complex tasks, like playing a large variety of video games or mastering the game of Go. Hence, researchers have been applying this approach to several aspects of Database Performance tuning.

The index selection problem is an area that has attracted a lot of attention by researchers in recent years, with some promising solutions using DRL. In this thesis we provide a comprehensive study of several of these state-of-the-art solutions. We observe that these studies are mostly concerned online DRL, failing to consider to what extent offline learning with DRL, from logs of an existing index advisor, can improve the process.

In our work we take upon the task of researching the contribution of learning from demonstrations with DRL, to improve the agents at the index selection problem. We conducted a proof of concept based on previous work, defining a scoped testing environment for the problem, using PostgreSQL. Then we concentrated our research on using learning from demonstrations of an index advisor. We evaluated different configurations to train an agent from past experiences, to see how it behaves, and how it could it help online training on unseen workloads.

We hope that the conducted research will encourage the research community to experiment further in starting with models that learn from demonstrations, instead of adopting models that must learn from scratch. In this way we hope to contribute to solutions to the index selection problem using DRL, which could fulfill their goal of efficiently complementing the experience of Database Administrators.
Acknowledgements

First and foremost, I would like to thank my thesis advisor M.Sc. Gabriel Campero Durand, whose expertise, guidance and continuous encouragement allowed not only the formulation of the thesis topic, but also helped me bring it to completion.

I would also 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.

Finally, I must express my most profound gratitude to my family, friends and to my girlfriend, for providing constant support and encouragement during the process of researching and writing of this thesis. Without them, this work could not have been possible. Thank you.
Declaration of Academic Integrity

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

Magdeburg, March 2nd 2020

__________________________________
Fabian Rodriguez
Contents

List of Figures xiv

1 Introduction 1
  1.1 Motivation ................................................. 1
  1.2 Main Contributions ......................................... 2
  1.3 Thesis Structure ........................................... 3

2 Background 5
  2.1 Indexes in DBMS ............................................ 5
  2.2 Index Advisory Tools ....................................... 6
    2.2.1 The Case for Automatic Database Administration using Deep
    Reinforcement Learning (NoDBA) .................................. 6
      2.2.1.1 Problem Definition .................................... 7
      2.2.1.2 Design ................................................. 7
      2.2.1.3 Evaluation ............................................ 8
    2.2.2 AI Meets AI: Leveraging Query Executions to Improve Index
    Recommendations ............................................. 10
      2.2.2.1 Classification Task .................................... 10
      2.2.2.2 Featurization ......................................... 11
      2.2.2.3 Training .............................................. 12
      2.2.2.4 Evaluation and Results ................................. 13
    2.2.3 LIFT: Reinforcement Learning in Computer Systems by Learning
    From Demonstrations .......................................... 15
      2.2.3.1 LIFT .................................................. 15
      2.2.3.2 Design ............................................... 16
      2.2.3.3 Learning from Demonstrations .......................... 17
      2.2.3.4 Evaluation .......................................... 17
    2.2.4 Learning Index Selection with Structured Action Spaces (STRUCT) 18
      2.2.4.1 Design .............................................. 18
      2.2.4.2 Evaluation .......................................... 19
    2.2.5 Summary of recent approaches for ML in index selection ....... 20
  2.3 Reinforcement learning ..................................... 22
    2.3.1 Introduction ............................................ 22
      2.3.1.1 Formal Definition ..................................... 22
2.3.1.2 Basic components ........................................ 23
2.3.1.3 Q-learning .............................................. 25
2.3.2 Deep Reinforcement Learning ............................... 25
  2.3.2.1 Deep Q-Networks ................................... 26
  2.3.2.2 Rainbow ............................................. 27
  2.3.2.3 Policy optimization algorithms and Policy Gradient .... 28
  2.3.2.4 Proximal Policy Optimization ......................... 29
  2.3.2.5 Asynchronous Advantage Actor-Critic .................. 30
2.3.3 Learning from Demonstrations ............................... 31
  2.3.3.1 Imitation Learning .................................. 31
  2.3.3.2 Inverse Reinforcement Learning ...................... 32
  2.3.3.3 Deep Q-learning from Demonstrations ................. 32
  2.3.3.4 Monotonic Advantage Re-Weighted Imitation Learning (MARWIL) ........................................ 33
2.3.4 DRL Frameworks ........................................... 34
  2.3.4.1 Dopamine ........................................... 35
  2.3.4.2 Ray ................................................ 36
2.4 Other applications of DRL in data management ............... 38
  2.4.1 Join order optimization ................................ 38
  2.4.2 Query Optimization ................................... 41
2.5 Summary ..................................................... 44

3 Design and Query/Workload generations .......................... 45
  3.1 Research Questions ....................................... 45
  3.2 Solution Design .......................................... 46
    3.2.1 Environment ......................................... 46
    3.2.2 DRL frameworks considerations ....................... 48
  3.3 Workload Generator ...................................... 49
    3.3.1 Synthetic TPC-H benchmark ............................ 49
    3.3.2 Generator .......................................... 50
  3.4 HypoPG .................................................. 52
  3.5 Index Advisor ............................................. 52
  3.6 Summary .................................................. 53

4 Experimental setup ............................................... 55
  4.1 Experiments ............................................... 55
    4.1.1 Online agents ....................................... 56
    4.1.2 Learning from Demonstrations ....................... 56
  4.2 Experiments environment .................................... 57
  4.3 Summary .................................................. 57

5 Evaluation and Results ........................................... 59
  5.1 Dopamine ................................................ 59
    5.1.1 DQN ................................................ 60
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1.1</td>
<td>Setup</td>
<td>60</td>
</tr>
<tr>
<td>5.1.1.2</td>
<td>Results and observations</td>
<td>61</td>
</tr>
<tr>
<td>5.1.1.3</td>
<td>Conclusions</td>
<td>62</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Rainbow</td>
<td>63</td>
</tr>
<tr>
<td>5.1.2.1</td>
<td>Setup</td>
<td>63</td>
</tr>
<tr>
<td>5.1.2.2</td>
<td>Results and observations</td>
<td>63</td>
</tr>
<tr>
<td>5.1.2.3</td>
<td>Conclusions</td>
<td>64</td>
</tr>
<tr>
<td>5.1.2.4</td>
<td>Comparison between DQN and Rainbow</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>Ray</td>
<td>65</td>
</tr>
<tr>
<td>5.2.1</td>
<td>DQN</td>
<td>65</td>
</tr>
<tr>
<td>5.2.1.1</td>
<td>Setup</td>
<td>65</td>
</tr>
<tr>
<td>5.2.1.2</td>
<td>Results and observations</td>
<td>66</td>
</tr>
<tr>
<td>5.2.1.3</td>
<td>Conclusions</td>
<td>67</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Learning from demonstrations</td>
<td>68</td>
</tr>
<tr>
<td>5.2.2.1</td>
<td>Setup</td>
<td>68</td>
</tr>
<tr>
<td>5.2.2.2</td>
<td>Hypothesis</td>
<td>70</td>
</tr>
<tr>
<td>5.2.2.3</td>
<td>Results and observations</td>
<td>70</td>
</tr>
<tr>
<td>5.2.2.4</td>
<td>Online Training from offline experiences</td>
<td>73</td>
</tr>
<tr>
<td>5.2.2.5</td>
<td>Conclusions</td>
<td>77</td>
</tr>
<tr>
<td>5.3</td>
<td>Summary</td>
<td>77</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion and Future work</td>
<td>79</td>
</tr>
<tr>
<td>6.1</td>
<td>Conclusions</td>
<td>79</td>
</tr>
<tr>
<td>6.2</td>
<td>Future work</td>
<td>80</td>
</tr>
</tbody>
</table>

### Bibliography

83
List of Figures

2.1 Workloads used for the experiments ....................................... 9
2.2 Workload execution times. NoIndex (baseline), IndexedAll consisting of
having an index on each column and NoDBA recommended indexes from
the DRL model............................................................................... 10
2.3 Example of featurization of a query plan into a vector (Feature Channel). 12
2.4 Number of queries that improved or regressed with the final configuration
after ten iterations....................................................................... 14
2.5 Distribution of workload-level execution cost improvement after ten iter-
tations......................................................................................... 15
2.6 Performance evaluation on the IMDB dataset................................... 18
2.7 Performance evaluation of the agents............................................ 19
2.8 Agent-Environment interaction.................................................... 23
2.9 DQN algorithm as defined in [MKS+13]........................................... 27
2.10 Taxonomy of Reinforcement Learning algorithms.............................. 29
2.11 Deep Q-learning from Demonstrations......................................... 33
2.12 Dopamine’s design. Blue boxes correspond to software components, yellow
ones indicate the file containing the implementation, arrows indicate class
inheritance, whereas numbers the number of non-comment Python lines. 35
2.13 Tradeoffs between tasks and actors............................................... 37
2.14 Ray’s architecture....................................................................... 37
2.15 Neo system model...................................................................... 41
2.16 Relative performance to plans created by the built in optimizer of different
systems for the three workloads....................................................... 43
3.1 Architecture of our solution......................................................... 46
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Selectivity matrix for training workload 1</td>
</tr>
<tr>
<td>3.3</td>
<td>Queries corresponding to training workload 1</td>
</tr>
<tr>
<td>5.1</td>
<td>Average returns of DQN training iterations</td>
</tr>
<tr>
<td>5.2</td>
<td>Average returns of DQN evaluation iterations</td>
</tr>
<tr>
<td>5.3</td>
<td>Duration of the whole set of training iterations in minutes</td>
</tr>
<tr>
<td>5.4</td>
<td>Duration of the evaluation iterations in seconds</td>
</tr>
<tr>
<td>5.5</td>
<td>Average returns of Rainbow training iterations</td>
</tr>
<tr>
<td>5.6</td>
<td>Average returns of Rainbow evaluation iterations</td>
</tr>
<tr>
<td>5.7</td>
<td>Duration of the training iterations in minutes</td>
</tr>
<tr>
<td>5.8</td>
<td>Duration of the evaluation iterations in seconds</td>
</tr>
<tr>
<td>5.9</td>
<td>Average returns of DQN training iterations</td>
</tr>
<tr>
<td>5.10</td>
<td>Average returns of DQN evaluation iterations</td>
</tr>
<tr>
<td>5.11</td>
<td>Duration of the training iterations in minutes</td>
</tr>
<tr>
<td>5.12</td>
<td>Duration of the evaluation iterations in seconds</td>
</tr>
<tr>
<td>5.13</td>
<td>Episodes reward mean for beta 0</td>
</tr>
<tr>
<td>5.14</td>
<td>Episodes reward mean for beta 0.5</td>
</tr>
<tr>
<td>5.15</td>
<td>Episodes reward mean for beta 1.0</td>
</tr>
<tr>
<td>5.16</td>
<td>Episodes reward mean for PG, PPO and A3C - Baseline</td>
</tr>
<tr>
<td>5.17</td>
<td>Episodes reward mean for PG, PPO and A3C - Checkpoint 1</td>
</tr>
<tr>
<td>5.18</td>
<td>Episodes reward mean for PG, PPO and A3C - Checkpoint 2</td>
</tr>
</tbody>
</table>
1. Introduction

1.1 Motivation

The performance of a DBMS depends on the several configuration options available, ranging from page size, replication settings, to the physical database design. One crucial aspect of the physical design that has a huge impact on a DBMS performance is the Index Selection Task.

The goal of this task is to find the optimal set of indexes that reduce the execution cost for a certain workload, without introducing performance regressions and keeping a limit on the storage of the selected indexes. This problem has been exhaustively researched in the past decades, using different approaches [FST88, Gra93], etc.

In recent years, with the evolution of Big Data, databases have been increasing both in size and with regards to the speed in which data is being created. These aspects making the Index Selection Problem more difficult [AMS15]. This situation, in turn, has led the database community to research and develop partial or fully automated solutions that recommend the best set of indexes for a given workload, assisting in this way the necessary database configuration-optimization process.

Recently, researchers have been focusing on using Machine Learning approaches (Reinforcement Learning in particular), in order to solve the Index Selection Problem; due to the ability of these approaches to learn to control behavior from experience. This field of research has shown great success in solving a wide variety of complex tasks and real world problems. It has attained human-level performance at playing several Atari Games [MKS+15], superhuman-level performance at playing Poker [BSM17] and mastering the game of Go [SHM+16], among others. Its application to the Index Selection Process is still a subject of active study.

Reinforcement Learning approaches have been applied to several aspects of Database Performance ranging from Join Order Optimization [KYG+18], Query Optimization [MNM+19], and also the Index Selection Problem [SSD18, SKE+18].
In this work, we concentrate on applying Deep Reinforcement Learning (DRL) approaches to the Index Selection Problem, with a focus on Learning from Demonstrations for relational databases. We do so, since this problem has been studied mostly with online methods, but limited understanding exists on the role of offline learning (based on past experiences), for improving the learning task. To the best of our knowledge, ours is the first research studying how Learning from Demonstrations can be applied for the Index Selection task on relational databases.

Furthermore, DRL is selected for our research, considering it is specially promising due to its ongoing standardization work, that has led to a large amount of efficient frameworks that make it easy to use DRL for everyday applications. It is also promising, because the inference process of DRL agents for recommending indexes can be faster than that of an online optimization algorithms, and it can scale to large problems.

We start with an introduction to the Index Selection Problem; then we review some of the existing state-of-the-art solutions, using Reinforcement Learning.

We design and build a prototype for Index Selection with DRL, based on previous work. We use the PostgreSQL database for our study, with the TPC-H dataset. We create a synthetic workload, which we describe carefully, to evaluate the Index Selection task. Unlike previous work, we integrate the recommendations of an Index Advisor to normalize our rewards (based on PostgreSQL cost estimations). We intend that our normalization choice would help in the interpretation of our results (with rewards indicating to what extent the DRL agents compare to the Index Advisor).

Using our developed model for the Index Selection task, we evaluate value-based agents in online learning. In this area, we compare the performance and inference of the different agents. Then we train different agents in Learning for Demonstrations, analysing the impact of configurations of the agents. Finally, we evaluate the process of online learning on unseen data, by starting the process by using a checkpoint of agents trained by Learning from Demonstrations.

### 1.2 Main Contributions

1. We provide a thorough review of several state-of-the-art Deep Reinforcement Learning and Machine Learning index advisory tools and Deep Reinforcement Learning frameworks that provide implementations for different agents.

2. We present the design and implementation of a Deep Reinforcement Learning solution based on previous research and make it open source, so the community can benefit from it, while also ensuring the reproducibility of our results.

3. We conduct a series of experiments that first show the feasibility of using an online agent to solve the Index Selection Problem. We do so by implementing agents on DRL frameworks, such as Dopamine and Ray.

4. We evaluate the impact of different configuration sof Learning from Demonstrations.
5. We study how learning from demonstration can aid in improving the learning process, by providing a starting point for online learning on unseen data.

6. We outline directions for future research that should provide further insights on how learning from demonstrations can help improving online training and how to adapt our solution to real world examples.

1.3 Thesis Structure

The rest of the thesis is structured as follows:

- In Chapter 2, we describe some of the fundamental concepts that serve as background for the presented problem. We start with a brief introduction to the Index Selection problem, move over to the theory behind Deep Reinforcement Learning and go through some of the state-of-the-art approaches and tools. We close the chapter by reviewing related work that provide a context for other research studying DRL applications in data management.

- Chapter 3 formulates the work into more concise research questions, and provides the definition of the several Reinforcement Learning components.

- In Chapter 4, we provide a detailed explanation of the experimental setup, the data used for training and evaluation, the specification of the resources used for running the experiments and the configuration of the DRL methods used.

- Chapter 5 is dedicated to the review, analysis and discussion of the evaluation results obtained as part of our experiments.

- Finally, we conclude this thesis, and provide directions for future research in Chapter 6.
1. Introduction
2. Background

In this chapter we present an overview of the theoretical aspects related to the index selection problem, together with some of the state of the art solutions relevant to it. The chapter is organized as follows:

- **Indexes in DBMS**: This chapter starts with an introduction to the index selection problem, its goal, challenges and impact on the performance of a DBMS.

- **Index Advisory Tools**: We follow with a detailed description of some state of the art index advisory tools, relevant to the research performed as part of this thesis.

- **Reinforcement Learning**: Next, we introduce the fundamental concepts of Reinforcement Learning (RL), the Deep Reinforcement Learning (DRL) methods studied, and state of the art DRL frameworks, given that they are relevant to the research performed.

- **Selected Applications of DRL in Data Management**: To close the chapter we provide a short review of other applications of DRL in data management, showing applications where our research could also hold some interest.

2.1 Indexes in DBMS

The performance of a DBMS depends on the several configuration options available, ranging from page size, replication settings, to the physical database design. One crucial process of the physical design that has a huge impact on a DBMS performance is the index selection task, whose goal is to find the optimal set of indexes that reduce the execution cost for a certain workload, without introducing performance regressions.
The selection of an optimal index configuration for a given workload is a challenging task, given that it imposes a trade off between the queries’ execution cost and the index maintenance, i.e. storage and update costs. As a matter of fact, a restricted version of this problem has been proven to be an NP-Complete problem [Com78]. Thus, it is a topic that has been exhaustively researched in the past decades using different approaches, for reaching acceptable practical solutions [FST88 Gra93 ZRL+04 DDM+19 SSD18].

Given its difficulty, an entire industry (Database Administration) was developed around tasks like this [KLR99]. Database Administrators (DBAs), select the optimal set of indexes based on statistics, design tools, expectations on future workloads, and experience [ZRL+04].

In recent years, with the evolution of Big Data, databases have been increasing in size, demanding new skills from DBAs [Sim13] and making the index selection problem more difficult [AMS15]. Hence, having a fully automated system that can recommend an optimal set of indexes has been of uttermost importance to researchers.

### 2.2 Index Advisory Tools

Most commercial DBMS vendors come with a built-in advisory tool that provides several recommendations for the different aspects of the physical design of a Database [DDD+04 ACK+05 ZRL+04 BC07]. These advisors, facilitate the tasks of the Database Administrator, and lead the path towards a fully automated management of DBMS [BAAT12].

One important goal in the data management community is to be able to develop solutions that can not only recommend a physical design for a specific workload [SSD18 SKE+18], but also one that can continuously tune the design and adapt it whenever the workload or the data changes [DDM+19 BLC+15].

In recent years, researchers have been focusing on using Machine Learning approaches (Reinforcement Learning in particular) in order to solve the Index Selection Problem, due to the ability of such approaches to learn to control behavior from experience.

In the next sections we will describe in detail some of the state of the art solutions for the Index Selection Problem that rely on using Reinforcement Learning methods, given that they are most relevant to our research.

#### 2.2.1 The Case for Automatic Database Administration using Deep Reinforcement Learning (NoDBA)

Sharma, et. al [SSD18] presented how Deep Reinforcement Learning can be applied in order to solve the Index Selection Problem, by training a neural network to select the optimal set of indexes for a given workload.
2.2.1.1 Problem Definition
Assuming a Database Schema $S$ with $n$ total columns among several tables, a workload $W$ and a maximum number of single-column indexes $K$, authors defined the problem as finding an optimal subset of $S$ of maximum size $K$ that minimizes the runtime of $W$.

2.2.1.2 Design
To perform Reinforcement Learning the Environment design, Action Space, Reward function and hyper-parameters were defined in the following way:

**Environment**

The Environment design proposed consists of an $n+1 \times m$ matrix, where $m$ is the total number of columns in the database schema and $n$ is the number of queries in the given workload, divided into two parts $I_{\text{workload}}$ and $I_{\text{indexes}}$.

$I_{\text{indexes}}$ encodes the current index configuration into a $1 \times m$ boolean vector indicating the presence or absence of an index for each column in the schema.

$$I_{\text{indexes}}[i] = \begin{cases} 1 & \text{if there is an index on column } i, \\ 0 & \text{otherwise} \end{cases} \quad (2.1)$$

$I_{\text{workload}}$, consists of an $n \times m$ matrix that encodes the characteristics of the workload related to data access, where an entry $I_{\text{workload}}(i, j)$ specifies the selectivity $\text{Sel}(Q_i, j)$ of column $j$ for the query $Q_i$.

$$I_{\text{workload}} = \begin{pmatrix} \text{Sel}(Q_0, C_0) & \cdots & \text{Sel}(Q_0, C_{m-1}) \\ \vdots & \ddots & \vdots \\ \text{Sel}(Q_{n-1}, C_0) & \cdots & \text{Sel}(Q_{n-1}, C_{m-1}) \end{pmatrix} \quad (2.2)$$

If query $Q_i$ has a predicate that contains column $j$, the selectivity $\text{Sel}(Q_i, j)$ returns the ratio between the number of records that match that predicate and the total number of records for that table, meaning that the lower the amount of records that match the predicate, the lower the selectivity. The expectation is that columns with many queries with low selectivity will benefit from an index on them, whereas columns with many queries with a high selectivity will not benefit from an index, since full-column scans would work better.

In case $Q_i$ does not include column $j$ in one of its predicates, then the selectivity $\text{Sel}(Q_i, j)$ is set to 1, encouraging the model to avoid creating an index on that column.

$$\text{Sel}(Q_i, C_j) = \begin{cases} \frac{\# \text{ selected records of } Q_i \text{ on } C_j}{\# \text{ total records of } C_j} & \text{if } Q_i \text{ has a predicate containing } C_j, \\ 1 & \text{if } Q_i \text{ does not have a predicate containing } C_j. \end{cases} \quad (2.3)$$
2. Background

Actions

The set of actions $A$ that the Agent can perform was restricted to only creating indexes. Given they use an episodic Deep Reinforcement Learning approach, where an episode consists of multiple steps in which an index is created, once they reached the maximum number of indexes Section 2.2.1.1, they conclude the episode, drop the existing indexes and start with the next episode. Thus, dropping indexes was not considered as a possible action.

Reward

Given that goal is to minimize the response time of the queries for the given workload, the authors defined a cost function which will serve as the basis for the Reward function.

The cost function returns the sum of the execution cost of each query in the workload. The baseline was defined as the cost of executing all the queries in the workload $\text{cost}(\emptyset)$ without any indexes. For an index configuration $L$ containing up to $K$ indexes the reward function is defined as:

$$r(L) = \max\left(\frac{\text{cost}(\emptyset)}{\text{cost}(L)} - 1, 0\right)$$

(2.4)

Hyper-parameters

The Deep Reinforcement Learning algorithm used as part of the study was CEM [DBKMR05] with a Neural Network consisting of:

- Four hidden layers with RELU as activation function.
- Each hidden layer with eight neurons.
- Output layer with SOFTMAX as activation function.
- Maximum amount of indexes set to 3.

2.2.1.3 Evaluation

For their experiments, the authors used PostgreSQL as the Database and the TPC-H benchmark with a scale factor of 1.

Training and testing workloads were generated using the LINEITEM table and queries of the form $\text{SELECT COUNT(*) FROM LINEITEM WHERE Selection1 AND Selection2 AND ...}$, where the predicates used were randomly selected from actual values present in the dataset and the operators used were either equality or range selections.
### Workloads

<table>
<thead>
<tr>
<th>Query</th>
<th>Selections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td><code>l.orderkey &lt; 10000, l.partkey &lt; 10000, l.quantity = 1, l.linenumber = 1, l.discount = 0.02</code></td>
</tr>
<tr>
<td>Q2</td>
<td><code>l.orderkey &lt; 100000, l.partkey &lt; 10000, l.quantity = 1, l.linenumber = 1</code></td>
</tr>
<tr>
<td>Q3</td>
<td><code>l.quantity = 1, l.partkey &lt; 100000, l.supkey &lt; 10000, l.orderkey &lt; 100000</code></td>
</tr>
<tr>
<td>Q4</td>
<td><code>l.orderkey &lt; 100000, l.discount = 0.0, l.supkey &lt; 10000, l.linenumber = 1</code></td>
</tr>
<tr>
<td>Q5</td>
<td><code>l.orderkey &lt; 100000, l.partkey &lt; 10000, l.supkey &lt; 10000, l.linenumber = 1, l.discount = 0.01</code></td>
</tr>
</tbody>
</table>

(a) **Workload 1.**

<table>
<thead>
<tr>
<th>Query</th>
<th>Selections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td><code>l.orderkey &lt; 100000, l.partkey &lt; 10000, l.supkey &lt; 10000, l.quantity = 1</code></td>
</tr>
<tr>
<td>Q2</td>
<td><code>l.quantity = 1, l.partkey &lt; 100000, l.orderkey &lt; 100000, l.linenumber = 1</code></td>
</tr>
<tr>
<td>Q3</td>
<td><code>l.orderkey &lt; 100000, l.partkey &lt; 10000, l.supkey &lt; 10000, l.quantity = 1, l.discount = 0.0</code></td>
</tr>
<tr>
<td>Q4</td>
<td><code>l.orderkey &lt; 100000, l.partkey &lt; 10000, l.supkey &lt; 10000, l.linenumber = 1, l.quantity = 1, l.discount = 0.02</code></td>
</tr>
<tr>
<td>Q5</td>
<td><code>l.orderkey &lt; 100000, l.partkey &lt; 10000, l.supkey &lt; 10000, l.linenumber = 1, l.discount = 0.02</code></td>
</tr>
</tbody>
</table>

(b) **Workload 2.**

<table>
<thead>
<tr>
<th>Query</th>
<th>Selections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td><code>l.orderkey &lt; 10, l.supkey &lt; 50000, l.extendedprice &lt; 5000, l.receiptdate &lt; '1993-12-31', l.returnflag = 'A', l.linenstatus = 'D'</code></td>
</tr>
<tr>
<td>Q2</td>
<td><code>l.orderkey &lt; 1500, l.extendedprice &lt; 10000, l.shipinstr = 'TAKE BACK RETURN', l.receiptdate &lt; '1993-06-30'</code></td>
</tr>
<tr>
<td>Q3</td>
<td><code>l.supkey &lt; 100, l.shipdate &lt; '1993-01-01', l.receiptdate &lt; '1992-06-29', l.linenumber = 4, l.shipmode = 'SHIP'</code></td>
</tr>
<tr>
<td>Q4</td>
<td><code>l.orderkey &lt; 1500, l.supkey &lt; 1000, l.shipdate &lt; '1995-03-31', l.linenumber = 4, l.tax = 0.0, l.returnflag = 'N'</code></td>
</tr>
<tr>
<td>Q5</td>
<td><code>l.supkey &lt; 50000, l.extendedprice &lt; 1000, l.comidate &lt; '1996-01-28', l.receiptdate &lt; '1992-06-29', l.quantity = 1, l.linenstatus = 'D'</code></td>
</tr>
</tbody>
</table>

(c) **Workload 3.**

Figure 2.1: Workloads used for the experiments

The workloads used for testing in the experiments are listed in figure Figure 2.1. **Workload 1** and **Workload 2** consist of queries with up to six selections on a fixed set of attributes with high selectivity, whereas **Workload 3** uses all the attributes in the table.
As shown in Figure 2.2, the recommended indexes were able to perform as good (and in some cases even better) as having indexes on all columns in the case of Workload 1 and Workload 2. Workload 3 on the other hand seems to perform better while having indexes on all columns, although for some of the queries the recommended index outperforms the execution time.

2.2.2 AI Meets AI: Leveraging Query Executions to Improve Index Recommendations

Most of the state of the art index advisory tools rely on the query optimizer cost estimates in order to recommend indexes that best improve the execution cost \[ \text{LGM}^{+15} \]. However, these estimates have limitations, such as errors in the cardinality estimation or cost-model \[ \text{LGM}^{+15}, \text{Loh}^{14}, \text{WCZ}^{+13} \], which could cause index recommendations that can hurt the performance.

Thus, Ding, Bailu, et al \[ \text{DDM}^{+19} \] used the execution cost to compare execution plans and formulated the problem as a Machine Learning classification task achieving a significantly higher accuracy compared to using a cost model to make the comparison.

2.2.2.1 Classification Task

In the study, a ternary classification task was presented, where given two query plans \( P_1 \) and \( P_2 \) for a query \( Q \), the tuple \( \langle P_1, P_2 \rangle \) gets a label assigned using the median of several executions according to the following:

<table>
<thead>
<tr>
<th>Workload</th>
<th>NoIndex [ms]</th>
<th>IndexedAll [ms]</th>
<th>NoDBA [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( W_1:Q1 )</td>
<td>562.143</td>
<td>13.912</td>
<td>38.350</td>
</tr>
<tr>
<td>( W_1:Q2 )</td>
<td>530.348</td>
<td>31.888</td>
<td>38.921</td>
</tr>
<tr>
<td>( W_1:Q3 )</td>
<td>490.926</td>
<td>82.439</td>
<td>74.666</td>
</tr>
<tr>
<td>( W_1:Q4 )</td>
<td>552.060</td>
<td>46.091</td>
<td>37.583</td>
</tr>
<tr>
<td>( W_1:Q5 )</td>
<td>545.870</td>
<td>39.811</td>
<td>24.067</td>
</tr>
<tr>
<td>( W_1:\text{Total} )</td>
<td>2681.347</td>
<td>214.121</td>
<td>213.587</td>
</tr>
<tr>
<td>( W_2:Q1 )</td>
<td>498.074</td>
<td>127.356</td>
<td>276.205</td>
</tr>
<tr>
<td>( W_2:Q2 )</td>
<td>491.517</td>
<td>999.767</td>
<td>193.690</td>
</tr>
<tr>
<td>( W_2:Q3 )</td>
<td>548.675</td>
<td>29.630</td>
<td>24.653</td>
</tr>
<tr>
<td>( W_2:Q4 )</td>
<td>548.188</td>
<td>25.126</td>
<td>24.617</td>
</tr>
<tr>
<td>( W_2:Q5 )</td>
<td>543.516</td>
<td>25.123</td>
<td>24.086</td>
</tr>
<tr>
<td>( W_2:\text{Total} )</td>
<td>2629.97</td>
<td>1207.002</td>
<td>543.251</td>
</tr>
<tr>
<td>( W_3:Q1 )</td>
<td>705.850</td>
<td>0.057</td>
<td>1028.729</td>
</tr>
<tr>
<td>( W_3:Q2 )</td>
<td>737.364</td>
<td>1.375</td>
<td>977.043</td>
</tr>
<tr>
<td>( W_3:Q3 )</td>
<td>728.996</td>
<td>803.244</td>
<td>743.211</td>
</tr>
<tr>
<td>( W_3:Q4 )</td>
<td>644.503</td>
<td>0.923</td>
<td>621.37</td>
</tr>
<tr>
<td>( W_3:Q5 )</td>
<td>851.598</td>
<td>1065.513</td>
<td>6.602</td>
</tr>
<tr>
<td>( W_3:\text{Total} )</td>
<td>3668.311</td>
<td>1871.112</td>
<td>3376.955</td>
</tr>
</tbody>
</table>

Figure 2.2: Workload execution times. NoIndex (baseline), IndexedAll consisting of having an index on each column and NoDBA recommended indexes from the DRL model.
2.2. Index Advisory Tools

- **regression**: If $P_2$ is more expensive, execution cost-wise (CPU time).
- **improvement**: If $P_2$ is cheaper.
- **unsure**: Otherwise.

### 2.2.2.2 Featurization

Machine Learning techniques rely on feature vectors of fixed dimensions. Thus, the query plans needed to be converted into feature vectors. The featurization process presented considered several aspects, ranging from Database Engine specifics to information from the optimizer.

The key aspects used to encode the query plans were:

- **Schema agnostic**: It should be able to learn in a cross-database manner, considering execution cost from several databases.
- **Leverage the information given by query optimizers.**
- **Do not rely on execution statistics.**
- **Measure of work done**: E.g. number of rows processed or the estimate given by the query optimizer for the cost of an operator.
- **Structural information**: The position of an operator in the query plan or join orders.
- **Physical operator details**: Whether the operator is parallel or serial and the execution mode (row/batch).

The fixed set of physical operators (SQL Server) were used as the feature dimensions, given that they are known beforehand and considering that changes in the set are infrequent. Each physical operator in a query plan gets assigned a key of the form `<Physical Operator>_<Execution Mode>_Parallelism` and the corresponding values for each key are the amount of work and structural information.

In order to keep the feature vectors with a fixed length, all values of the operators with the same key are summed, and a value of zero is assigned to the corresponding key in case an operator is not part of the query plan. Furthermore, different Feature Channels can be created, depending on the type of Structural Information and Measure of work done. The following table summarizes the different Feature Channels considered:
<table>
<thead>
<tr>
<th>Channel</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EstNodeCost</td>
<td>Estimated node cost as node weight (work done).</td>
</tr>
<tr>
<td>EstRowsProcessed</td>
<td>Estimated rows processed by a node as its weight (work done).</td>
</tr>
<tr>
<td>EstBytesProcessed</td>
<td>Estimated bytes processed by a node as its weight (work done).</td>
</tr>
<tr>
<td>EstRows</td>
<td>Estimated rows output by a node as its weight (work done).</td>
</tr>
<tr>
<td>EstBytes</td>
<td>Estimated bytes output by a node as its weight (work done).</td>
</tr>
<tr>
<td>LeafWeightEst - RowsWeightedSum</td>
<td>Estimated rows as leaf weight and weight sum as node weight (structural information).</td>
</tr>
<tr>
<td>LeafWeightEst - BytesWeightedSum</td>
<td>Estimated bytes as leaf weight (structural information).</td>
</tr>
</tbody>
</table>

Figure 2.3 shows an example of the featurization of a query plan containing a join with three tables, executed in a single-threaded manner in row mode (a), as well as how the corresponding feature channels are computed using the raw values from the plan (b).

Figure 2.3: Example of featurization of a query plan into a vector (Feature Channel).

2.2.2.3 Training

Once the query plans have been featurized, Machine Learning approaches were used to train an offline model. As an example of linear learners Logistic Regression was
chosen, whereas among the tree-based family of algorithms, Random Forest, Gradient-Boosted Trees and Light GBM were chosen. This offline model can be constructed on a per-database level or it can even be a general model for several databases.

The ability to adapt to unseen data is critical for the problem at hand and although offline models can be re-trained with new data periodically, this solution does not scale given the size of the infrastructure of current databases. An alternative approach is to train a local model with the execution data for a specific database in order to make predictions for unseen data. The main advantage of this local model is that re-training this model is several orders of magnitude cheaper than re-training the offline model that contains information from several databases with new data.

However, the local model is likely to overfit and thus, make poor predictions. Hence, a new model was defined, meta model \( M_{\text{meta}} \) and is trained to use predictions from both models.

The execution data collected from a database is splitted into two disjoint sets \( D_l \) and \( D_m \), where the former is used to train the local model \( M_{\text{local}} \) and the latter to train the meta model \( M_{\text{meta}} \). The offline model is trained with execution model from other databases.

Meta features are extracted for each data point \( d \) in \( D_m \), such as distances of labels of close neighbors of \( d \) in \( D_l \) and uncertainty scores. Once the features are extracted a Random Forest is trained on \( D_m \) and both \( M_{\text{local}} \) and \( M_{\text{meta}} \) are re-trained whenever new execution data is available.

### 2.2.2.4 Evaluation and Results

The experimental for evaluating the index recommendation quality (in terms of CPU Time) setup consisted on using the TPC-DS benchmark with scale factors 10 and 100 and a real world workload (Customer6), on SQL Server. Both single query and workload level tuning were evaluated.

The results obtained were compared against two baselines:

- **Opt**: original index advisor using the query optimizer cost estimation.
- **OptTr**: same as previous one, except that an index is recommended if the estimated improvement is greater than a threshold (20%).

For the experiments, two different settings of how the training data is collected were used:

- **AdaptivePlan**: the offline model is trained with plans collected from the database being tuned, as well as other databases.
- **AdaptiveDB**: the offline model is trained with data collected from databases different from the one being tuned.
Due to their nature (cost estimates), Opt and OptTr do not take feedback from past executions, the tuning stops when there is no index recommendation available or whenever there is a regression on the performance for a query.

In contrast, both the AdaptivePlan and AdaptiveDB prevent regressions. In case of a regression after an index recommendation, the indexes are reverted and given that the models are continuously being adapted with new data an alternative configuration can be recommended in the next iteration.

Figure 2.4 shows the results of evaluating single query tuning with the before mentioned methods on the different workloads. Each query was tuned independently in order to get optimal query-wise index recommendations.

The following metrics were reported:

- **Improve**: the amount of queries that improved at least by 20% in execution cost with the final configuration, with respect to the initial configuration.

- **Regress**: the amount of queries that regress when the tuning stops.

Figure 2.4: Number of queries that improved or regressed with the final configuration after ten iterations.

These results show that with Opt the amount of queries that end up with a regression in their performance can be as high as 29%, confirming that cost estimates are not the best for index recommendations. In contrast, both adaptive approaches significantly reduce the number of regressions while improving in most cases most of the queries.

For the workload-level tuning, an index configuration is recommended for a set of queries in a 20 query workload. Figure 2.5 shows the workload-level distribution of the execution cost improvements across 60 workloads.
2.2. Index Advisory Tools

<table>
<thead>
<tr>
<th>Workload Improve</th>
<th>TPC-DS 10g</th>
<th>TPC-DS 100g</th>
<th>Customer6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Opt Tr ADB APlan</td>
<td>Opt Tr ADB APlan</td>
<td>Opt Tr ADB APlan</td>
</tr>
<tr>
<td>0.2</td>
<td>5 5 5 9</td>
<td>5 5 3 4</td>
<td>8 6 5 9</td>
</tr>
<tr>
<td>1</td>
<td>3 3 3 3</td>
<td></td>
<td>5 6 9 8</td>
</tr>
<tr>
<td>5</td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>8 8 9 13</td>
<td>5 5 3 4</td>
<td>14 12 14 17</td>
</tr>
</tbody>
</table>

Figure 2.5: Distribution of workload-level execution cost improvement after ten iterations.

All these improvements come at a cost. The major overhead found by the authors was the time taken for retraining the online model, which even on thousands of plans, completes in a few minutes. Though arguably, not significant, this might not be practical for all cases. All in all, the proposed approach on average adds less than 10% of tuning time (in estimation), with respect to a DBA.

2.2.3 LIFT: Reinforcement Learning in Computer Systems by Learning From Demonstrations

Schaarschmidt, et al [SKE+18], proposed a different approach to the Index Selection Problem by using human expertise to learn from demonstrations. They introduced LIFT, a software stack for applying Deep Reinforcement Learning to data management tasks, together with Tensorforce which is a Tensorflow library that provides a unified API for Reinforcement Learning algorithms.

The authors studied learning from demonstrations via log data or demonstrated examples as an alternative to current solutions that apply online or offline models.

In the study, the limitations of the mentioned approaches in data management are enumerated as: the huge training data requirements, the expensive decision evaluation and the cost of hyper-parameter tuning.

Online training can be impractical for the selection of indexes, given that performing a single step and observing its impact could take hours. In addition, Deep Reinforcement Learning algorithms require a lot of configuration, hyper-parameter tuning and design decisions.

2.2.3.1 LIFT

LIFT consists of a set of reusable components and can run in three different execution modes:

- Pre-training: During the pre-training mode, LIFT does not interact with the System. Instead it parses and maps a trace data source to demonstrations, creates
a Reinforcement Learning agent and imports the data. Finally, it monitors and validates the model performance and serializes the model for subsequent use.

- **Agent-Driven: (Active execution)** In this mode LIFT alternates interacting with the system and the Reinforcement Learning agent. During this mode, the majority of the time is spent in interacting with the environment.

- **Environment-Driven: (Passive Execution)** In this type of execution mode, LIFT acts as a passive service. Control flow is driven by external workload, it maps incoming data to states, feeds them to the agent, and performs the necessary configuration changes on the system. It’s primary purpose is to perform incremental updates on trained models that have been deployed.

The user needs to implement a small set of LIFT components in order to perform the parsing and serialization of the data. A schema component is responsible for programmatically constructing the layouts of states, actions and rewards. Then, a model converter component needs to be implemented. This component performs the translation between the Reinforcement Learning model and the system, to be able to:

- Map system output to states and actions for pre-training.
- Map system output to rewards.
- Map agent output to system configuration changes.

2.2.3.2 Design

*MongoDB* was chosen for conducting the experiments. This Database has the particularity that only one index (order sensitive) per query can be used, with the exception of queries using the `$or` operator, for which a different index can be used for each sub-expression. Moreover, the optimizer can make use of index intersections to resolve queries.

**States**

The state is defined by the current index configuration and the current query. Queries are parsed and a sequence of operators and attributes are extracted, and if there is an index for one of the parsed attributes, this information is encoded as part of the query to enable the agent to learn about index intersection.

**Actions**

For every query the possible actions are either creating an index or none. A positional action model was proposed, of which given a query, the attributes are extracted and an action is interpreted as creating an index in the $ith$ attribute. An extra action per attribute is created in order to be able to distinguish sort patterns.
Rewards

In order to express the trade-offs between runtime requirements and memory usage, the reward function was defined as follows:

\[ r(q) = -\omega_1 m(I) - \omega_2 t(q) \tag{2.5} \]

where \( t(q) \) is the execution time of query \( q \), \( I \) an index set and \( m(I) \) the memory usage of the current index configuration.

2.2.3.3 Learning from Demonstrations

The main motivation for this approach is that a Database Administrator using his expertise can determine within a reasonable time-frame an index configuration for an application by accessing profiling tools.

Demonstration Data

For the Index Selection Problem, demonstrations are available in the form of:

- Query Logs from applications tuned by a Database Administrator, where indexes are assumed to be correct, not necessarily optimal.
- Query logs from applications where indexes were created using heuristics. This set of indexes is taken to be non optimal and it could even not reach necessary service objectives.
- Queries and index pairs for which no runtime data is available.

2.2.3.4 Evaluation

The main goal was to use of LIFT for learning an optimal set of compound indexes while minimizing latency (mean 90th and 99th percentile) and memory usage. For the evaluation two datasets were used, the IMDB dataset and a synthethic dataset based on the YCSB benchmark.

After running LIFT in pre-training mode and feeding it with the workloads and demonstrations, it is restarted in online mode and trained with a different set of queries. As baseline, a full index strategy was chosen, in which a compound index consisting of all the attributes present in a query is created.

Four different scenarios were evaluated:

- Default: System without any extra indexes created (MongoDB by default creates an index on the _id attribute).
- Online: online learning was performed without pretraining.
• Pretrain: only pretraining was performed (without online refinement).
• Pretrain+Online: pretraining followed by online refinement.
• Human: a human expert recommendation for the given queries.

The results of the experiments are presented in Figure 2.6. For the evaluation, each query was executed five times and the final results were averaged over five trainings with different random seeds.

![Figure 2.6: Performance evaluation on the IMDB dataset.](image)

The results show that Pretrain+Online outperforms by a huge margin the rest of the strategies, with respect to mean, 90th percentile and 99th percentile latency. In contrast, the index memory consumption of the human expert was marginally bigger than the Pretrain+Online strategy due to the fact that the expert tried to minimize the amount of indexes by leveraging index intersection.

2.2.4 Learning Index Selection with Structured Action Spaces (STRUCT)

Schaarschmidt, et al. [WSY19] investigated how adding structure to an action space like the one in the Index Selection Problem, can improve the recommended index configurations. As a result, a Deep Reinforcement Learning Agent that exploits hierarchical structure was able to find index configurations up to 40% smaller than other approaches without deteriorating the level of latency.

2.2.4.1 Design

State

The state was represented as the current query and index configuration. For the experiments a fixed query structured of the form `SELECT COUNT(*) FORM lineitem WHERE ... AND ...` was used, so only the attributes and operators are extracted from them.

Actions

An action determines which of the attributes of a query will be indexed together with their ordering. In this case, indexes with up to \( m \) keys and queries with up to \( n \) predicates were considered.
2.2. Index Advisory Tools

Reward

The reward function Equation 2.6 consists of the weighted sum of storage space and execution time:

\[ r(q) = -\omega_1 m(I) - \omega_2 t(q) \]  

(2.6)

where \( t(q) \) is the latency of query \( q \) and \( m(I) \) is the size of the index constructed at the current step.

2.2.4.2 Evaluation

For the experiments PostgreSQL was chosen as the Database and a synthetic version of TPC-H (with scale factor 1) workloads was used. The workloads consisted of queries of the form \( \text{SELECT COUNT}(*) \text{ FROM lineitem WHERE ... AND ...} \), where the predicates are also sampled from values derived from the specification, similar to \[SSD18\] \[SKE++18\].

A DQN agent based on the work by \[SKE++18\] was implemented, together with a new Deep Reinforcement Learning agent using the Sinkhorn Policy Gradient (SPG) algorithm \[ER18\].

After training, the DQN and SPG agents are exposed to test workloads, the corresponding indexes are created and finally, those workloads are evaluated against the new index configuration. Each workload consisted of 25 queries with up to three attributes per query.

Baselines

The agents were measured against three baselines:

- Default: Using PostgreSQL’s default indexes for primary keys.
- Full: Single-key indexes on all attributes.
- Tuner: Index configuration recommended by OpenTuner \[AKV++14\].

Three DQN and SPG agents were trained with a hundred workloads and tested using three different workloads. The experimental results (final results averaged over training runs and training and test workloads) are shown in Figure 2.7

![Figure 2.7: Performance evaluation of the agents.](image)
When considering mean latencies, it is clear that Full is similar to Tuner, DQN and SPG. In contrast, when inspecting the 80th percentile, Tuner can identify indexes that speed up slower queries over Full, but it performs a 33% slower than DQN and SPG.

2.2.5 Summary of recent approaches for ML in index selection

To summarize the recent approaches for ML in index selection we offer table Table 2.2. In general, deep reinforcement learning is present in several of the studied papers, with multi-class classification being the other approach considered. In terms of the complexity of the problem formulation, the classification approach employs a sophisticated query-plan featurization of the task, while the others are relatively more simple. The work in STRUCT [WSY19] makes progress in the area of action design, creating a structure for actions based on queries, that considers ranking of indexes. With regards to the mode of learning, offline learning has received attention, but there is no combination of DRL and offline learning for relational databases (MongoDB is a document-oriented storage). In addition, open-source implementations are not available, and the workloads used are not easily comparable, creating the necessity for a more established benchmark.

We should also note that the absence of evaluations of offline learning for relational databases with DRL serves as a research gap which we intend to address with our current research.
Table 2.2: Summary of recent ML-based approaches for index selection

<table>
<thead>
<tr>
<th>Paper</th>
<th>ML Approach</th>
<th>Models Used</th>
<th>Offline approach considered?</th>
<th>Evaluation</th>
<th>Open-source implementation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoDBA[SSD18]</td>
<td>DRL</td>
<td>CEM</td>
<td>No</td>
<td>PostgreSQL Synthetic workloads</td>
<td>No</td>
</tr>
<tr>
<td>AI meets AI[DDM19]</td>
<td>Multi-class classification</td>
<td>Logistic Regression Tree-based Models</td>
<td>Yes</td>
<td>SQL Server Workloads based on TPC-DS and 1 Real-world (private) workload</td>
<td>No</td>
</tr>
<tr>
<td>LIFT[SKF19]</td>
<td>DRL</td>
<td>DQN Deep Q-learning from Demonstrations</td>
<td>Yes</td>
<td>MongoDB Workloads based on the IMDB Dataset and YCSB benchmark</td>
<td>No</td>
</tr>
<tr>
<td>STRUCT[WSY19]</td>
<td>DRL</td>
<td>DQN Sinkhorn Policy Gradient</td>
<td>No</td>
<td>PostgreSQL Workloads based on TPC-H</td>
<td>No</td>
</tr>
</tbody>
</table>
With this summary we conclude our presentation on recent ML-based approaches for index selection. In the next section we discuss on machine learning models and frameworks pertinent to the task.

2.3 Reinforcement learning

In this section we introduce Reinforcement Learning (RL) as a prelude to Deep Reinforcement Learning (DRL), with a focus on the concepts relevant to our research. The section is structured as follows:

- We start with a brief introduction of the terminology and basic concepts.
- Next, we describe DRL, some of the state of the art approaches that are used in our experiments and general applications of this family of algorithms.
- We conclude this section, with a brief summary of DRL frameworks that serve as the foundations for running our experiments.

2.3.1 Introduction

Reinforcement Learning (RL) is a subfield of Machine Learning that focuses on automatic learning of optimal decisions over time. Informally, RL can be described as an agent interacting with an environment that has to make decisions in order to optimize a given notion of cumulative reward \[ SB^{+98} \].

A key aspect of RL is that it does not require a full knowledge of the environment, instead agents learn by trial-and-error, interacting with the environment and gathering information. There are two alternatives in which the agent can gather experience: offline (or batch) and online.

In the batch setting, the experience is collected beforehand and then used for learning, whereas in the online setting, data is available in a sequential order and is used to incrementally update the behavior of the agent.

One of the main challenges that RL faces is the exploration-exploitation dilemma. While an agent gains information about the environment, it has to make a trade-off between learning more about the environment by performing actions not explored so far, or exploiting the current knowledge and following the most promising strategy which could lead to local optima.

2.3.1.1 Formal Definition

The RL problem is formally defined as a Markov Decision Process where at each time step \( t \), the Agent takes actions \( a \in A \) on the Environment by following a policy \( \pi \), to transition from state \( st \) to state \( st+1 \) and receives a reward \( rt \in R \). This process is illustrated in Figure 2.8.
2.3. Reinforcement learning

2.3.1.2 Basic components

The main goal of the Agent is to find an optimal policy that maximizes the long-term cumulative reward. A RL Agent includes one or more of the following components:

- A Value Function that provides a prediction of how good each state (or state/action pair) is.
- A Model of the Environment together with a planning algorithm.
- A Direct representation of the policy \( \pi(s) \) or \( \pi(s, a) \).

Approaches that include either a Value Function or a direct representation of the policy are classified as \textit{model-free}, whereas an approach that includes a model of the environment is classified as \textit{model-based}.

Next, we describe in detail the different components of \textit{model-free} approaches, given that they are relevant to the research conducted.

Policies

Policies define how agents select actions to perform on the environment. They can be categorized as follows:

- Deterministic: where the policy can be described as
  \[ \pi(s) : S \rightarrow A. \]  \hspace{1cm} (2.7)

- Stochastic: where the action is selected based on a probability
  \[ \pi(s, a) : S \times A \rightarrow [0, 1] \]  \hspace{1cm} (2.8)
**Expected Return**

Given:

- S: State space.
- A: Action space.
- $T: S \times A \times S \rightarrow [0, 1]$ transition function.
- $R: S \times A \times S \rightarrow R$ the reward function.
- $\gamma \in [0, 1)$: discount factor used to balance how the value of future steps will impact the value of the current state.

The goal of the Agent is to find a policy $\pi(s, a) \in \Pi$ in order to optimize an expected return (V-value function) $V^\pi(s): S \rightarrow R$, such that:

$$V^\pi(s) = \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, \pi \right]$$

(2.9)

where

$$r_t = \mathbb{E}_{a \sim \pi(s_t, .)} R(s_t, a, s_{t+1})$$

(2.10)

$$\mathbb{P}(s_{t+1} \mid s_t, a_t) = T(s_t, a_t, s_{t+1}) \text{ with } a_t \sim \pi(s_t, .)$$

(2.11)

From Equation 2.9, we can derive the optimal expected return as:

$$V^*(s) = \max_{\pi \in \Pi} V^\pi(s).$$

(2.12)

which translates to the expected discounted reward for state $s$, while following the optimal policy.

**Q-value**

The Q-value function (Quality function) is used for the family of RL methods that seek to empirically obtain the optimal policy. The Q-value function $Q^\pi(s, a): S \times A \rightarrow R$ is defined as follows:

$$Q^\pi(s, a) = \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, a_t = a, \pi \right]$$

(2.13)

Thus, the optimal Q-value function $Q^*(s, a)$ can be defined as:

$$Q^*(s, a) = \max_{\pi \in \Pi} Q^\pi(s, a).$$

(2.14)

The main advantage of the Q-value function over the V-value function is that the optimal policy can be derived directly from $Q^*(s, a)$:

$$\pi^*(s) = \arg \max_{a \in A} Q^*(s, a)$$

(2.15)

The optimal Q-value Equation 2.14 is the expected discounted return for a given state $s$ and action $a$ while following $\pi^*$ thereafter.
2.3. Reinforcement learning

2.3.1.3 Q-learning

In this section we discuss Q-learning [WD92], one of the simplest value-based algorithms (model-free approach that uses a value function).

Q-learning maintains a lookup table Q-table of Q-values \((Q(s,a))\) where each entry corresponds to a state-action pair.

The algorithm applies the Bellman equation [Bel57] to the Q-value function Equation 2.14 in order to learn the optimal value function based on the observations of episodes during the learning process.

Hence, the optimal Q-value function can be defined as:

\[
Q^*(s,a) = \sum_{s' \in S} T(s,a,s')(R(s,a,s') + \gamma \max_{a' \in A} Q(s',a'))
\] (2.16)

Watkins, et. al. [WD92] demonstrated that this learning approach converges to the optimal value function under certain conditions:

- The (state, action) pairs are represented discretely
- Sufficient exploration is ensured, i.e. all actions are repeatedly sampled in all states.

2.3.2 Deep Reinforcement Learning

Value-function approaches are well suited for problems that involve relatively small State spaces and discrete actions. However, due to the memory storage necessary for the Q-table, these approaches become impractical for continuous action spaces or substantially large state spaces. Hence, approximation methods using Deep Neural Networks have been proposed [GBCT16].

Deep Learning [LBH15] refers to the use of Neural Networks for Machine Learning. In recent years, this approach has been used in a wide variety of problems, covering Image Visualization [OMS17], Medical Image Analysis [LKB+17], among others.

Deep Reinforcement Learning (DRL) consists of the combination of Reinforcement Learning and Deep Learning, where the neural networks are used to approximate the Q-values or a policy given a state, serving as the basis for action selection. This field of research has shown great success in solving a wide variety of complex tasks and real world problems. It has attained human-level performance at playing several Atari Games [MKS+15], superhuman-level performance at playing Poker [BSM17] and mastering the game of Go [SHM+16] among others.
2.3.2.1 Deep Q-Networks

Mnih, et. al. [MK$\text{S}$$^+$$\text{13}$], presented the first DRL model that used a combination of Neural Networks (NN) with Q-Learning to learn control policies from high-dimensional inputs; Deep Q-Networks (DQN). In their study, a NN was trained to play Atari games, using Atari frames (images) as inputs and changes in the game score as the Reward.

A Convolutional Neural Network with weights $\theta$ was used to approximate the Q-function, and trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that change at each iteration $i$,

$$L_i(\theta_i) = E_{s,a \sim \rho(.)}[(y_i - Q(s,a;\theta_i))^2] \quad (2.17)$$

where $y_i = E_{s' \sim \epsilon}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a]$ is the target for iteration $i$ and $\rho(s,a)$ is a probability distribution over sequences of states $s$ and actions $a$.

In order to avoid strong correlations between consecutive data samples, DQN implements a technique called Experience Replay [Lin93], where a Replay Memory is used to store the Agent’s experience. During training, experiences are sampled from the Replay Memory.

Two separate neural networks $Q_{\text{target}}$ and $Q_{\text{current}}$ are trained in order to aid convergence. $Q_{\text{current}}$ is continuously updated and only after $N$ iterations the weights from $Q_{\text{current}}$ are copied to $Q_{\text{target}}$. In this way the predictions of Q-values for the following states don’t depend on the same network that attempts to estimate the Q-value for the current state.

The interested reader can refer to Figure 2.9 for a complete explanation of the DQN algorithm. At each step of an episode during training the agent selects an action either randomly or by choosing the action with the maximum Q-value (i.e., the agent follows an epsilon greedy policy). Based on the selected action, the agent interacts with the environment and stores in the replay buffer the relevant information (the reward obtained, the new state, the action performed). After interacting, and before the next interaction, the agent undergoes training. Training consists on sampling a series of transitions from the replay buffer and performing gradient descent on the $Q_{\text{current}}$ neural network, attempting to minimize the error between the predictions currently given and the expected true value, which is considered to be the observed reward for the transition, for the case of terminal states, or the reward plus the discounted prediction of the maximum Q value for the next step, according to the $Q_{\text{target}}$ neural network. Not shown in the algorithm, after some number of steps the weights of the current network are copied to the target network.
2.3. Reinforcement learning

Hessel, et. al. presented the Rainbow Agent [HMVH18] in order to address some of the limitations present in DQN, by combining several approaches, each of which add significant performance improvements in isolation, into one integrated agent. Next, we provide a brief summary of the different approaches, how they contribute to Rainbow’s performance together with its result on the Arcade Learning Environment (ALE) [BNVB13] benchmark.

- **Double Q-learning**: The traditional Q-learning algorithm tends to overestimate action values, due to its maximization step, which harms learning. Double Q-learning [HMVH18] addresses this issue by decoupling the selection of the subsequent action from its evaluation. This approach can be combined with DQN using the following loss function:

\[
(R_{t+1} + \gamma_{t+1} q_\theta(S_{t+1}, \arg \max_a q_\theta(S_{t+1}, a'; \theta)) - q_\theta(S_t, A_t))^2
\] (2.18)

- **Prioritized replay**: DQN samples transitions from the replay buffer uniformly. However, we would like to sample transitions from which we can learn the most. Hence, Shaul, et. al. [SQAS15] proposed to sample transitions with a probability proportional to the last encountered absolute temporal difference error. Furthermore, a bias towards recent transitions is introduced by inserting new transitions in the replay buffer with maximum priority.

- **Multi-step learning**: In this approach, the rewards over n consecutive steps, are truncated and added to the expected return. As demonstrated by Sutton, et. al. [SB+98], multi-step targets with tuned n often lead to faster learning.
• **Distributional RL**: Instead of learning a single-value approximation (i.e., the expected Q-value for an action given a state), Distributional Reinforcement Learning agents [BSHB17] learn to approximate the distribution of the action values. To do so they can learn quantiles or categories (often called atoms). This approach should help exploration.

• **Dueling networks**: Wang, et. al. [WSH+15] proposed a neural network architecture that features two estimators (one for the state value function and another one for the action advantage function) that share a convolutional encoder, and are merged by an aggregator. According to the study, this architecture leads to a better policy evaluation, when there are several similar-valued actions. Similar to distributional RL, this approach enhances the Q-values learned by agents, and should improve the exploration.

• **Noisy nets**: One of the limitations of DQN, is that it uses an $\epsilon$−greedy policy for exploration, which is problematic when many actions need to be executed in order to collect the first reward, e.g. Montezuma’s Revenge Atari game [HMVH+18]. Fortunato, et. al. proposed Noisy Nets [FAP+17], which add a noisy linear layer that combines a deterministic and a noisy stream. Over time, the agent learns to ignore the noise and to exploit the learned policy. This approach intends to help exploration.

In the study, Hessel, et. al. showed that by combining all this approaches into Rainbow, led to state-of-the-art performance on the benchmark suite ALE [BNVB13] of 57 Atari 2600 games, not only in terms of data efficiency but also in terms of final performance.

2.3.2.3 Policy optimization algorithms and Policy Gradient

[Figure 2.10] provides a comprehensive summary of the categorization of recent Reinforcement Learning algorithms. So far, we described several Q Learning algorithms that are fundamental to our research. In this section we scope our discussion to three Policy Optimization algorithms: Policy Gradient, A3C and PPO, which play an important role in our experiments.
As an alternative to function approximation, Sutton, et. al [SMSM00] explored the concept of approximating a stochastic policy using an independent function approximator. They chosen to represent the policy with a neural network whose inputs represent the state, its outputs the action probabilities and its weights the policy parameters.

In the vanilla policy gradient approach, the policy parameters are approximated proportional to the gradient as in:

\[
\Delta \theta \approx \alpha \frac{\delta \rho}{\delta \theta}
\]

where \( \theta \) denotes the vector of policy parameters, \( \alpha \) the step size, and \( \rho \) the performance of the corresponding policy. In [SMSM00], Sutton, et. al, proved that if Equation 2.19 can be achieved, then \( \theta \) converges to a locally optimal policy with respect to \( \rho \).

### 2.3.2.4 Proximal Policy Optimization

[SWD+17] introduced a new family of policy gradient algorithms called Proximal Policy Optimization. The main difference with standard policy gradient algorithms is that PPO uses multiple epochs of stochastic gradient ascent to perform each policy update. Furthermore, PPO provides the option of multiple agents working in parallel at each training step, to collect experience from the current policy.

According to the results presented by Schulman, et. al. [SWD+17], PPO was able to outperform several policy optimization algorithms (PG, A3C, CEM, TRPO) while providing a balance between simplicity, sample complexity and wall time. An example of the PPO algorithm is shown below:

---

2. Background

### 2.3.2.5 Asynchronous Advantage Actor-Critic


Among the variants, Asynchronous Advantage Actor-Critic (A3C) proved to be the best one, outperforming current state-of-the-art methods on the Atari benchmark. The method keeps a policy $\pi_{\theta_{old}}$ and an estimate of the value function $V(s_t; \theta_v)$ and operates in the forward view as defined in [SB+98] using the same combination of steps to update the value function and the policy.

The update perform can be described as

$$\nabla_{\theta'} \log \pi(a_t | s_t'; \theta') A(s_t, a_t; \theta, \theta_v)$$  \hspace{1cm} (2.20)

where $A(s_t, a_t; \theta, \theta_v)$ is an estimate of the advantage function given by

$$\Sigma_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v)$$  \hspace{1cm} (2.21)

where $k$ is upper bounded by a fixed amount of actions and varies from state to state. The pseudo-code for the A3C algorithm is provided below.

---

**Algorithm 2 PPO**

1: for iteration = 1, 2, \ldots do
2: \hspace{1cm} for actor = 1, 2, \ldots, N do
3: \hspace{2cm} Run policy $\pi_{\theta_{old}}$ in environment for $T$ time steps
4: \hspace{2cm} Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
5: \hspace{1cm} end for
6: \hspace{1cm} Optimize surrogate $L$ wrt. $\theta$, with $K$ epochs and minibatch size $M \leq NT$
7: \hspace{1cm} $\theta_{old} \leftarrow \theta$
8: end for

---
2.3. Reinforcement Learning

2.3.3 Learning from Demonstrations

As part of RL, an Agent performs an Action on the Environment and receives a Reward. The reward function defines the goal that the Agent has for the given Environment. However, defining a reward function is not always a simple task due to the increasing complexity of some Environments.

As a result, a family of approaches called Learning from Demonstrations was developed. In these approaches, the Agent is not given any rewards, instead, it is provided with demonstrations of the desired task performed by an expert. Thus, the goal is to make the Agent to perform as similar as possible as the expert. In the following subsection we cover the most common types of approaches.

2.3.3.1 Imitation Learning

Imitation Learning [MJP97] makes use of supervised learning to train a classifier or regressor in order to predict the behavior of an expert, from observations and actions taken by the expert. Several applications for this approach, ranging from visual perception of forest trails [GGC+15] to self-driving cars [BDTD+16] have shown great success.

Given state-action pairs \((s_t, a_t)\) in the data generated by some behavior policy \(\pi\), the idea is to minimize the KL divergence between \(\pi\) and \(\pi_\theta\), in other words, we want to minimize:

\[
D_{KL}^d(\pi || \pi_\theta) = -\mathbb{E}_{s \sim (s)}(\log \pi_\theta(a|s) - \log \pi(a|s))
\] (2.22)
2.3.3.2 Inverse Reinforcement Learning

Ng, et. al. \cite{NR00} defined the Inverse Reinforcement Learning Problem in the context of Machine Learning as to determine a reward function given a finite set of trajectories from an optimal behavior. In \cite{AN04} Abbeel, et. al. demonstrated how an apprentice can learn how to drive a car from trajectories.

2.3.3.3 Deep Q-learning from Demonstrations

In recent years, DRL has had success in several applications. Some notable examples were mentioned at the start of section \textsection{2.3.2}.

However, these approaches require a huge amount of data in order to reach a decent performance, and usually tend to perform poorly during learning.

Real world applications like autonomous vehicles \cite{HS13} require the DRL algorithm to be able to efficiently learn from a few samples while still taking actions in real time. One solution to this problem is to train in simulation, however, there are not many applications with accurate emulators available.

In contrast, most of these applications have demonstration data available from when they were operated under the control of either a machine or human (controller). Based on this idea, Sendora, et. al. \cite{SDL17} presented a new DRL algorithm, Deep Q-learning from Demonstration (DQLFD), which leverages the use of the demonstration data to not only accelerate the learning process but also to perform well from the start.

DQFD starts with a pre-training phase, where the Agent is trained only on the demonstration data using a combination of supervised and temporal difference losses, before running on the actual system. The supervised loss is crucial during this stage allowing the Agent to imitate the demonstrator. Given that the data covers only a small part of the Action Space, this loss forces the values of non taken actions to be at least a margin lower that the ones taken by the demonstrator.

In contrast the temporal difference loss ensures that the network satisfies the Bellman equation, allowing it to learn a valid value function so as to continue learning and improving the policy online with RL. To prevent overfitting on the usually small demonstration data set, an extra L2 regularization loss is applied to the weights and biases of the network.

After pre-training, the Agent having learned an appropriate policy (and stored demonstration data in a demonstration replay buffer $D_{\text{demo}}$) starts interacting with the system, collecting self generated data and storing it in a replay buffer $D_{\text{replay}}$.

Each mini-batch used to update the network contains $n$ samples with data from each buffer with a proportion of $p = \frac{n_{\text{demo}}}{n_{\text{demo}} + n_{\text{replay}}}$ of demonstration data. A pseudocode of the DQLFD algorithm is shown in \textbf{Figure 2.11}. 
2.3. Reinforcement learning

Figure 2.11: Deep Q-learning from Demonstrations.

The presented algorithm was able to outperform RL approaches such as Double DQN on 27 out of 42 (average rewards across four trials) Atari games and Imitation Learning on 31 out of 42 games, and a better initial performance than DQN on 40 of the 42 games. Moreover, DQFD outperformed the best demonstrator on six of the games, and even when poor demonstration data was intentionally given, it received more average rewards than DQN and IL.

2.3.3.4 Monotonic Advantage Re-Weighted Imitation Learning (MARWIL)

Wang, et. al. [WXH+18] considered learning a policy from batched historical data by monotonic advantage re-weighting, that is able to make decisions in complex environments, working with hybrid action spaces (discrete and continuous).

The easiest method to learn a policy from data is with imitation learning [Section 2.3.3.1]. However, this makes no distinction between good or bad actions, it just imitates every action generated by the policy. If, there are rewards \( r_t \) included as part of the data,
the algorithm would be able to know the impact of taking an action \( a_t \), by looking at the next state \( s_{t+1} \) and reward \( r_t \). If we have an estimation of the advantage of action \( a_t \) defined by \( \hat{A}^\pi(s_t, a_t) \), then actions with higher advantages could have higher sample weights, imitating better actions more often.

Hence, they proposed a monotonic advantage reweighted imitation learning method which maximizes Equation 2.23 where \( \beta \) is a hyperparameter, which when set to zero, the algorithm becomes ordinary imitation learning.

\[
\mathbb{E}_{s \sim d_\pi(s), a \sim \pi(a|s)} \exp(\beta \hat{A}^\pi(s, a)) \log \pi_\theta(a|s)
\]  

(2.23)

They found that estimating the advantage function \( A(s_t, a_t) = \mathbb{E}_{\pi|t} (R_t - V^\pi(s_t)) \) using a single path estimation as \( \hat{A}(s_t, a_t) = (R_t - V_\theta(s_t))/c \) yielded the best results, where they normalize the advantage by its average norm \( c \) in order to make the scale of \( \beta \) stable across different environments. The complete algorithm is presented in algorithm 1.

**Algorithm 1: MARWIL algorithm**

- **Input** Historical data \( D \) generated by \( \pi \), hyper-parameter \( \beta \).
- For each trajectory \( \tau \) in \( D \), estimate advantages \( \hat{A}^\pi(s_t, a_t) \) for time \( t = 1, ..., T \).
- Maximize \( \mathbb{E}_{(s_t, a_t) \in D} \exp(\beta \hat{A}^\pi(s_t, a_t)) \log \pi_\theta(a_t|s_t) \) with respect to \( \theta \).

The most notable advantages of the proposed algorithm are:

- Its simplicity.
- The method works with hybrid action spaces of both discrete and continuous types.
- It can be used to learn from an unknown policy.
- After training, the learned agent can be deployed to an agent for online learning (this is not the common case for pure imitation learning).

with our discussion of MARWIL we close our presentation on deep reinforcement learning models. We discussed the general reinforcement learning approach, we introduced Q-value based methods, DQN and its optimizations in Rainbow, policy optimization algorithms (policy gradient, PPO and A3C), before overviewing learning from demonstration with its main approaches of imitation learning, DLFD and MARWIL. In the next section we introduce 2 state-of-the-art DRL frameworks, which we use for our research.

**2.3.4 DRL Frameworks**

In recent years, a lot of research has been conducted on the topic of RL, and it has taken a turn towards more complex agent-environment interactions [CMG+18].
2.3. Reinforcement learning

such as superhuman-level performance at videogames \[ MKS^{+15} \]. Hence, reliable and reusable implementations of DRL agents has become the focus in order to facilitate new developments both in academic and production settings. In the following section, we review some off-the-shelf solutions that provide state-of-the-art implementations of several DRL agents together with trained models and training data.

2.3.4.1 Dopamine

Castro, et. al. \[ CMG^{+18} \] presented Dopamine a Tensorflow-based framework with focus on RL research. In particular, Dopamine focuses on two of the research goals defined by \[ ABC^{+16} \], algorithmic research and instruction.

It provides implementations for several state-of-the-art value-based agents (only value-based agents as of the moment of writing this thesis) for solving the Arcade Learning Environment (ALE) as defined by \[ BNVBI^{13} \] that are performant enough for research at scale, together with a set of trained models and training data.

Dopamine’s architecture is shown in Figure 2.12 built by emphasizing the following principles:

- Self-contained and Compact: In order to provide an easy to comprehend framework for users, Dopamine was built with a few number of files as seen in Figure 2.12. The Runner class manages the interactions between the Agent and the Environment (ALE). The Checkpointer class is responsible of periodically saving the experiment’s state, in order to ensure recovery in the event of a failure or the re-use of learned policies. In contrast, Logger is in charge of saving statistics of the experiment for later visualization. The rest of the classes are self-explanatory.

- Reliable and Reproducible: Dopamine demonstrates reliability by providing a test suite with a coverage of over 98%. Reliability and reproducibility are two

---

1End to end OSS Machine Learning platform: [https://www.tensorflow.org/](https://www.tensorflow.org/)
key aspects in any research, in particular in RL \cite{BTSK17}. In order to reduce inconsistencies and variability, Dopamine makes use of \textit{gin-files}\footnote{Framework for parameter injection, see \url{https://github.com/google/gin-config}.} for configuring different modules (agents, algorithms, etc), making contributions easier to share with the community.

Dopamine has proven to be a reliable, flexible, and reproducible framework for DRL research. However, it is limited to some of the design decisions taken by the authors, e.g. small-scale value-based agents that are mostly developed to run on ALE, which makes it less appealing compared to other solutions presented next.

\subsection*{2.3.4.2 Ray}

Learning a policy in DRL can be described as a two-step process, first policy evaluation, where an Agent interacts with the Environment generating trajectories and policy improvement, where the agent uses these trajectories in order to improve the policy. The main challenges in RL come from the need to scale these two steps \cite{LLM+17}.

However, most of the existing frameworks have not exploited opportunities for distributed computation \cite{LLM+17}. One could argue that is possible to use several systems, one for each stage of the process and combine them in one end-to-end solution, but in practice, this approach has been proven to be untenable given how tightly coupled this systems end up being to the applications and the high communication costs between them.

These issues, led Moritz, et. al. \cite{MNW+18} to develop Ray, a general purpose cluster-computing framework for building and running RL applications in a distributed fashion. It provides a single dynamic execution engine that supports the implementation of a unified interface that can support both task-parallel and actor-based computations.

Furthermore, Ray was built keeping the following requirements in mind:

- Fine-grained, heterogeneous computations. Not only the duration of a computation can range from a few milliseconds to several hours, but also, training could require heterogeneous hardware, e.g., sampling of experiences run often on CPUs whereas policy optimizations on GPUs.

- Flexible computation model: RL applications require both stateless computations for fine-grained simulation and data processing, and stateful computations for maintaining network parameters and replay buffers.

- Dynamic execution. Some components of RL applications require dynamic execution, given that the order in which computations terminates is not always known, and their results can determine even more computations.

Ray models applications as a dependency graph of tasks that evolves during execution. On top of this, it provides an actor and task-parallel programming model. Next, we provide a brief summary of Ray’s programming and computation model.
2.3. Reinforcement learning

Programming Model

Ray provides both tasks and actors. A task represents the execution of a remote function on a stateless worker, and a future representing its result is immediately returned. In contrast, an actor represents a stateful computation. It exposes methods that can be invoked remotely and are executed in a serial manner, and are similar to tasks in that they return a future. Figure 2.13 provides a brief comparison of the properties of both tasks and actors.

<table>
<thead>
<tr>
<th>Tasks (stateless)</th>
<th>Actors (stateful)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fine-grained load balancing</td>
<td>Coarse-grained load balancing</td>
</tr>
<tr>
<td>Support for object locality</td>
<td>Poor locality support</td>
</tr>
<tr>
<td>High overhead for small updates</td>
<td>Low overhead for small updates</td>
</tr>
<tr>
<td>Efficient failure handling</td>
<td>Overhead from checkpointing</td>
</tr>
</tbody>
</table>

Figure 2.13: Tradeoffs between tasks and actors.

Computation Model

As we previously mentioned, Ray implements a dynamic task graph computation model, where the execution of remote functions and actor methods are automatically triggered as soon as their inputs become available.

Architecture

Ray’s architecture consists of an application layer that implements the API and a system layer, which provides fault tolerance and high scalability. Figure 2.14 provides an overview of Ray’s architecture.

Figure 2.14: Ray’s architecture.
The application layer comprises three types of processes:

- **Driver**: process in charge of executing the user program.
- **Worker**: stateless process that executes tasks.
- **Actor**: stateful process that executes when invoked only the methods it exposes.

In contrast, the system layer consists of three major components:

- **Global Control Store**: key-value store with pub-sub functionality that maintains the control state of the system, enabling every component in the system to be stateless. Its main objective is to provide fault tolerance and low latency.

- **Distributed Scheduler**: Ray has a two level hierarchical scheduler consisting of a global and a per-node local schedulers. A local scheduler schedules tasks locally unless the node is overloaded, in which case it forwards the task to the global scheduler. In contrast, the global scheduler consider the load and constraints on each node, and assigns tasks to the node that provides the lowest estimated waiting time.

- **Distributed Object Store**: An in-memory distributed storage system is implemented in order to minimize task latency.

In this section we have discussed frameworks for deep reinforcement learning, following a presentation of deep reinforcement learning models and learning for demonstration approaches. In the following section we return our focus to the data management side, presenting other applications of DRL in data management, which introduce to the reader other areas where our work in understanding better learning from demonstrations might have an impact.

### 2.4 Other applications of DRL in data management

Apart from index selection, there are other applications in data management where DRL is being studied as a solution. In this section we present to the reader developments in two of such areas: Join order optimization and Query optimization.

#### 2.4.1 Join order optimization

Join order optimization has been studied for more than four decades and it is still an interesting topic among researchers \([\text{SAC}^+79]\). Due to its complexity (NP-Hard) it is usually solved using heuristics in order to prune the action space.

Inspired by the advances made by exploring the use of machine learning in solving database problems \([\text{KKR}^+18, \text{MVAH}^+18, \text{MP}18]\), Krishnan, et. al. \([\text{KYG}^+18]\) presented
a Reinforcement Learning based optimizer named \( DQ \), which optimizes blocks of the form \( \text{select-project-join} \).

Their key insight is that join ordering has the same problem structure that is present in Reinforcement Learning, hence, they studied how to embed a RL solution into a query optimizer, by replacing any enumeration algorithm with a policy learned by RL.

The join order optimization problem can be defined as follows: Given \( G \) a query graph with \( K_G \) connected components, \( V \) set of vertexes, and \( J \) a cost model, we want to find a sequence of joins terminating in \( |V| = K_G \) that minimizes:

\[
\min_{c_1, \ldots, c_T} \sum_{i=1}^{T} J(c_i) \quad \text{subject to } G_{i+1} = c(G_i) \tag{2.24}
\]

This problem can be seen as a Markov Decision Process, when we consider the query graph as a representation of the state, the joins selected \( c_i \) (an edge in the graph) as actions, the vertex merging process (after removing a join edge) as the state transition \( P(G, c) \) and the negative cost \(-J\) as a possible reward function.

When we consider the optimization problem for a query graph \( G \) of finding the lowest query join, we can define its value function \( V(G) \) as Equation 2.24. Hence, considering the current join as starting point, we can re-write the value function as:

\[
V(G) = \min_c Q(G, c) \quad \text{with } Q(G, c) = J(c) + V(G') \tag{2.25}
\]

With this function definition, Q-learning was the easiest DRL approach to solve the problem. It allowed not only to take advantage of optimal substructures during training reducing the amount of training data needed, but also allows for top-\( k \) planning rather than just finding the best plan.

Three cost models were considered for the experiments:

- CM1: cost model for main-memory database.
- CM2: accounts for disk usage in hybrid hash joins.
- CM3: allows the re-use of hash tables.

The Q-learning agent was compared against the following baselines:

- QuickPick1000 (QP): [WCHN13] which selects the best of 1000 random join plans.
• IK-KBZ (KBZ): [KBZ86] polynomial-time heuristic that decomposes the query graph into chains and orders them.

• Right-deep (RD).

• Left-deep (LD).

• Zig-zag (ZZ): [ZZBS93] which uses Zigzag trees to process queries in parallel.

• Exhaustive (EX): exhaustive enumeration of join plans.

Table 2.4 shows the results of the experiments. Regarding Cost Model 1, DQ clearly outperforms heuristic solutions such as QP, KBZ and RD and performs similar to ZZ and LD were in the worst case scenario its performance is only 1.45 times worse, proving that a reinforcement learning approach could be a viable alternative to heuristics.

When we consider Cost Model 2, none of the heuristics was able to match the performance of the exhaustive search, suggesting that as memory becomes limited, heuristic tend to diverge from the optimal solution. In contrast, DQ outperforms all the heuristics and gets quite close to the optimal, although not as close as in the first model.

Finally, when the re-use of hash tables is allowed, DQ significantly outperforms the heuristics and was the only approach that was able to determine that for some queries, bushy plans were the best option, hence generating low-cost solution.

In summary, these results prove that DQ is not only robust against different cost models, due to its ability to adapt to different workloads, but also that it can outperform several heuristics, making it a good alternative.

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Cost Model 1</th>
<th>Cost Model 2</th>
<th>Cost Model 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Mean</td>
<td>Max</td>
</tr>
<tr>
<td>QP</td>
<td>1.0</td>
<td>23.87</td>
<td>405.04</td>
</tr>
<tr>
<td>KBZ</td>
<td>1.0</td>
<td>3.45</td>
<td>36.78</td>
</tr>
<tr>
<td>RD</td>
<td>4.70</td>
<td>53.25</td>
<td>683.35</td>
</tr>
<tr>
<td>LD</td>
<td>1.0</td>
<td>1.08</td>
<td>2.14</td>
</tr>
<tr>
<td>ZZ</td>
<td>1.0</td>
<td>1.07</td>
<td>1.87</td>
</tr>
<tr>
<td>EX</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>DQ</td>
<td>1.0</td>
<td>1.32</td>
<td>3.11</td>
</tr>
</tbody>
</table>

Table 2.4: Plan costs relative to exhaustive enumeration.
2.4.2 Query Optimization

Query optimization is an issue that has been of interest to the database community throughout the years. The main requirement of an optimizer is that has to be memory and time efficient, hence most of commercial system use heuristics for cardinality estimation, physical operator selection and query plan costs, leading to bad query plans [LGM+13].

Inspired by recent advances in solving data management problems with Machine Learning, Marcus, et. al. [MNM+19] introduced Neo (Neural Optimizer), an entire query optimizer that relies on NN in order to generate query execution plans, which is able to learn about index selection, physical operator selection and join ordering.

In the first stage, Neo uses a “traditional” query optimizer (in this case PostgreSQL’s optimizer) as an expert, and uses Learning From Demonstration [Section 2.3.3] to bootstrap its learning process and then continues to learn from new queries. Figure 2.15 illustrates an overview of how the system learns.

As part of the study, the queries were restricted to project-select-equijoin-aggregate-queries and the database remain the same throughout the experiments. However, authors show their work to generalize to unseen queries (though it is difficult to understand whether these queries pose a generalization challenge or not), and can be extended to support more types of queries.

![Figure 2.15: Neo system model](https://example.com/neosystemmodel.png)

Neo replaces each component of a query optimizer with machine learning models:
• Query representation: is done with feature vectors instead of object-based operator
trees.
• Cost model: is implemented with a Deep Neural Network.
• Search strategy: is a Deep Neural Network-guided best-first strategy.
• Cardinality estimation: based on histograms in combination with a learned model.

Next, we provide a brief description of all the components as depicted in Figure 2.15 and how Neo uses Reinforcement Learning and Learning from Demonstration to integrate them into a query optimizer.

**Expertise Collection**

In this first phase, a Sample Workload with representative queries of the application, together with an Expert Optimizer (“traditional” query optimizer) are provided. From this expert, the system gathers Experience, i.e. each query is executed through the optimizer, and the generated query execution plan along with the query and its latency are stored.

**Model Building**

Once the Experience has been gathered, Neo builds its initial Value Model, consisting of a Deep Neural Network designed to predict the final execution time of a partial/complete plan for any given query.

**Plan Search**

After the queries have been encoded into feature vectors, they are fed to the Value Model in order to find the plan with the minimum predicted execution time. Neo uses the Value Model as an heuristic in order to perform a best-first search of the space of execution plan for a given query. The generated plan, is then passed to the execution engine for its execution, ensuring that it computes the correct result.

**Model Refinement**

After the query execution plan is sent to the execution engine and the query is processed, the plan, together with its latency are added to the Experience, and the Value Model is retrained, improving, iteratively, its estimates.

For every query entered by the user, the same featurization, search and refinement process is adopted. Neo penalizes with a higher cost, plans that end up performing poorly. This is possible when the learned cost model leads to a query plan with a good performance prediction, whose resulting latency is high. Hence, the cost model learns from its mistakes becoming more accurate over time.

For the experiments three different benchmarks across different Database systems were used:
2.4. Other applications of DRL in data management

- JOB: Join Order Benchmark\cite{LGM15} over the IMDB dataset.
- TPC-H: with a scale factor of 10^3.
- Corp: 2TB dataset with 8k unique queries from an internal application.

In order to evaluate Neo’s performance, the mean execution time of the query plans generated by Neo were compared against the execution time of the plans generated by the built in optimizer of PostgreSQL, SQLServer, SQLite and Oracle, for each of the aforementioned workloads.

Figure 2.16 depicts the relative performance (lower better) of Neo against the native optimizers. It can be seen that Neo generates query plans that have (in average) 40% less execution time that the ones provided by PostgreSQL built-in optimizer. Moreover, Neo outperforms the built-in optimizers in every system for all of the workloads, except for TPC-H under SQLServer and Oracle, but only by a small margin 10%. Marcus, et. al. \cite{MNM19} suspect that the performance of both commercial systems is extremely good because they are tuned towards TPC-H, given that it is a standard benchmark in the industry.

![Figure 2.16: Relative performance to plans created by the built in optimizer of different systems for the three workloads.](image)

To summarize this section, there is a clear tendency in the data management research to use DRL approaches to tackle database performance problems \cite{MP18,MNM19,SKE18,DDM19} and even though in most cases the approaches are still at the research stage, the results obtained so far are promising. This provides a motivation for

\[http://www.tpc.org/tpch/\]
our research, where we seek to contribute in understanding better one training approach for developing DRL models in data management applications.

2.5 Summary

In this chapter, we presented the necessary theoretical background that we consider essential to understand the experiments and results obtained as part of this work. We start the chapter with a brief overview of the Index Selection Problem in databases. Furthermore, we provide some examples of state-of-the-art index advisory tools and give a broad description of deep reinforcement learning, its concepts and relevant methods.

Also in this chapter we reviewed in depth two Deep Reinforcement Learning Frameworks which we will later use as part of our experiments, with a focus on their implementation, architecture and the technology stack they use.

We concluded the chapter by summarizing research on two recent application areas of DRL in data management: join order optimization and query optimization. Through this we expect to provide some context for our research.

In the next chapter we establish our specific research questions and present the design of the prototype we built for our study.
3. Design and Query/Workload generations

In the following chapter, we provide a detailed description of the research questions that have been addressed as part of this work. In Section 3.2 we describe the design of our solution and we close the chapter Section 3.3 with an overview of some of the tools that we needed to implement in order to accomplish the tasks.

3.1 Research Questions

The main goal of this research is to determine the feasibility of using Learning from Demonstrations (Section 2.3.3) to train a model that can recommend an optimal set of indexes for a given workload in a relational Database. In particular, we address the following research questions:

1. How do different online agents behave? We provide a comparison of the performance (training time and convergence) of different state-of-the-art agents with respect to the Index Selection Problem.

2. Once the models are trained, how well can they perform inference (i.e. provide index recommendations for unseen workloads)?

3. How well does a model using Learning From Demonstration from logs perform? In particular, we want to understand how its training requirements compare to that of online learning. Moreover, after the model is trained, how well can it perform inference or help in online training on unseen workloads?
3.2 Solution Design

In this section we present the design of our DRL Environment in detail together with a brief description of the necessary changes made and considerations taken to the state-of-the-art DRL frameworks presented in Section 2.3.4 to enable them to run our environment correctly. Figure 3.1 depicts the architecture of our solution.

![Figure 3.1: Architecture of our solution.](image)

3.2.1 Environment

The implementation of our Environment follows the interface defined by *OpenAI Gym Environment*[^1], which provides a universal interface supported by most of the state-of-the-art DRL frameworks.

*OpenAI Gym* requires each environment to implement the following functions:

- **step**: implements an actual interaction between the Agent and the Environment. It takes an action as an argument and returns four values: *observation*, *reward*, *done* and *info* (diagnostic information).

- **reset**: resets the Environment to an initial *observation*.

[^1]: [https://gym.openai.com/docs](https://gym.openai.com/docs)
and two attributes: **action_space** which describes the set of valid actions and **observation_space** which describes the set of valid observations, in order to implement the Agent-Environment interactions as depicted in Figure 2.8.

Next, we describe the implementation of the different aspects required by *OpenAI Gym’s* interface.

**Observation Space**

We defined the **observation_space** in the same way as [SSD18] which was described in Section 2.2.1.2 as a matrix that combines the current set of indexes and the selectivity of each column with respect to each query.

**Action Space**

The set of Actions that the Agent can perform at any given timestep is defined in the same manner as previous works [SSD18, SKE+18, WSY19].

Given that our queries focus only on the **lineitem** table (see Section 3.3), we reduced the action space to creating indexes on columns belonging to that table. Moreover, we limit the amount of indexes that an agent can create to a fixed number **MAX_INDEXES** to prevent it from creating an index on each column of the table.

**Reset**

The main goal of the reset function is to reset the environment and to return an initial observation. As part of the implementation, we reset the matrix with the original selectivities and set the current set of indexes as empty.

Furthermore, the **initial_cost** and **best_cost** are calculated (only on the first invocation of the program, then the values are cached) so that we can compare the evolution of the cost in every step and the final reward.

The cost is calculated as the sum of the execution time of each of the individual queries, given the selected index configuration. Here, we use explained execution time, which corresponds to the cost provided by the engine. This has criticisms, for the reasons given in Section 2.2.2, making our process susceptible of cardinality and estimation errors. In future work we should be able to use some actual execution for our training and evaluations, or study the impact of the suggested indexes on actual execution times.

**Step**

In each timestep, the Agent performs an Action on the Environment by calling the **step** function and providing the selected Action as an argument. First, we perform some checks in order to know if we should terminate the episode as described in the next section.
Next, we increment the number of steps taken and if the termination conditions are met we calculated the reward and finish the episode, otherwise, we create the new index and calculate the new cost. Furthermore, we update the state by marking that a new index was created on that column and also set the selectivity of that column for each query to one as if the query no longer has a predicate on that column (See Equation 2.3). This is done in order to make the change have a bigger impact on the environment making the gradient descent algorithm converge faster.

Finally, the function returns the new state, the reward, whether the episode is finished or not and some information useful for performing diagnostics.

**Done**

We consider the following scenarios as termination conditions for the current episode:

- When the Agent tries to create more than the maximum allowed amount of indexes.
- When the cost considering the new index is greater than or equal to the current cost, i.e. the new index does not provide any performance improvement.
- When the action is invalid, in case the action does not belong to the action space.
- When the Agent tries to create an index on a column that is not used by any of the queries. We terminate the episode early in this case without even calculating the cost given that we know that the index will not provide any improvements.
- When the Agent tries to create an index that already exists in the system.

**Reward**

The reward function for an index configuration $I$ can be described as follows:

$$
r(I) = \begin{cases} 
\max(\frac{\text{current cost} - \text{initial cost}}{\text{best cost} - \text{initial cost}} \times 100 - \text{amount of steps}, 0) & \\
1 & \text{otherwise}
\end{cases}$$

where amount_of_steps corresponds to the steps taken so far in the current episode.

**3.2.2 DRL frameworks considerations**

In order to improve training times, we made the agent smarter with respect to the actions it could take. In particular, we prune the Action Space after an action was executed by the agent, in order to prevent it from trying to create an already existing index.

In Dopamine [Section 2.3.4.1](#), there is not an interface or configuration option to implement action pruning, hence, we had to edit the source code and alter how the actions are
selected. We do so, by keeping track of the actions that were performed in an array that
gets reseted every time an episode begins.

In each timestep, an action can be selected either: randomly with probability $\epsilon$, or in a
greedy manner by selecting the action with the highest Q-value. Before the new action
gets selected, we go through every performed action and change those Q-values to the
lowest value possible so that they do not get selected.

In contrast, Ray provides an interface for performing action pruning called Parametric
Action Spaces\footnote{https://ray.readthedocs.io/en/latest/rllib-models.html#variable-length-parametric-action-spaces} where a set of valid actions per step can be defined. Here, the idea is
similar, we keep a mask of the actions that have been performed, and in every step we
mask the performed action so that it does not get selected in a future step.

Unfortunately for the Marwil agent, we had to modify the source code of the neural
network used by the agent in order to perform action pruning and improve training
times. Though this is not so entirely an appropriate solution, this facilitated the use
of diverse agents for training after the MARWIL checkpoints, which would have been
cumbersome if we had to implement a parametric model for each of them. The training
times with this change were reduced from 30 minutes to 2 minutes. The approach taken
was the same we used for Dopamine.

### 3.3 Workload Generator

In this section we provide a brief summary of why the standard benchmarks are not
properly suited for the problem at hand, which led us to develop a custom tool based
on previous research.

#### 3.3.1 Synthetic TPC-H benchmark

For our experiments we used a synthetic version of the standard TPC-H benchmark
with scale factor 1. By default, TPC-H benchmark provides a set of 22 query templates
in an OLAP fashion across different tables, making it appropriate for the Index Selection
Problem given that the queries can benefit from different index configurations \cite{WSY19}.

However, a set of 22 queries might not be enough data or variety for a DRL agent to
learn properly (keeping in mind that a workload consists of 25 queries). Hence, we
developed synthetic workloads in the same manner as previous approaches \cite{SSD18,WSY19,SKE18,MNM19}.

Each workload consists of 25 queries on the \textit{lineitem} table of the form \textit{SELECT count(*) FROM lineitem WHERE ... AND ...}, with up to three predicates per query combined
with the \textit{AND} keyword.

The predicates are randomly generated from \textit{lineitem}’s columns and consist on a single
condition on one column with one of the following operators: $>$, $<$, $=$. The set of
possible values used as part of the conditions are extracted from the actual entries in
the DB.
3.3.2 Generator

We developed a tool\(^3\) that generates the workloads defined in Section 3.3.1 based on the approach taken by SSD18, WSY19, SKE\(^++\)18, MNM\(^++\)19.

With our tool we generated ten workloads for training and three for testing purposes, both sets of workloads are stored into disk under the a workloads folder, inside a folder with the corresponding name. The generator also calculates the selectivity and the cost for each query in the workload and stores them in YAML files in order so they can be used later to create the Environment for the RL agent, as described in Section 3.2.

An example of the selectivity matrix for the first training workload is shown in Figure 3.2. This matrix serves as the observation space described in Section 3.2.1. It contains 25 rows, one for each query belonging to the workload and 16 columns that correspond to each of the queries in the lineitem table. Table 3.1 describes the mapping between lineitem’s columns and their index in the selectivity matrix.

![Figure 3.2: Selectivity matrix for training workload 1.](https://github.com/fabianrbz/thesis/tree/master/generator)
3.3. Workload Generator

<table>
<thead>
<tr>
<th>Column name</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>l_orderkey</td>
<td>0</td>
</tr>
<tr>
<td>l_partkey</td>
<td>1</td>
</tr>
<tr>
<td>l_suppkey</td>
<td>2</td>
</tr>
<tr>
<td>l_linenumber</td>
<td>3</td>
</tr>
<tr>
<td>l_quantity</td>
<td>4</td>
</tr>
<tr>
<td>l_extendedprice</td>
<td>5</td>
</tr>
<tr>
<td>l_discount</td>
<td>6</td>
</tr>
<tr>
<td>l_tax</td>
<td>7</td>
</tr>
<tr>
<td>l_returnflag</td>
<td>8</td>
</tr>
<tr>
<td>l_linenstatus</td>
<td>9</td>
</tr>
<tr>
<td>l_shipdate</td>
<td>10</td>
</tr>
<tr>
<td>l_commitdate</td>
<td>11</td>
</tr>
<tr>
<td>l_receiptdate</td>
<td>12</td>
</tr>
<tr>
<td>l_shipinstruct</td>
<td>13</td>
</tr>
<tr>
<td>l_shipmode</td>
<td>14</td>
</tr>
<tr>
<td>l_comment</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 3.1: Mapping of column names.

At the beginning of each episode, the matrix is loaded with the corresponding selectivities that were previously calculated as defined in Equation 2.3 (the lower the value the better impact that an index on that column could have in accelerating the query). Figure 3.2 presents also a heatmap of the selectivities, the ones with the lowest values are depicted with colours closer to red, whereas the highest values are closer to the yellow end of the colour spectrum.

Figure 3.3 presents the queries corresponding to the training workload 1. It can be seen, that the only columns present in Query 1 are l_receiptdate and l_shipdate. The corresponding selectivities are depicted in Figure 3.2. From their colour code, we see that l_receiptdate has a higher selectivity than l_shipdate making it a good candidate for an index.
3.4 HypoPG

HypoPG\(^4\) is a PostgreSQL extension that adds support for Hypothetical Indexes. These types of indexes, are not physical indexes (they do not exist in the database), hence, they do not require any CPU or disk space in order to be created.

Hypothetical indexes are useful when trying to evaluate the impact that a new index can have in the performance, without spending extra resources to create them. The extension provides a convenient API for creating and removing hypothetical indexes, simply by running: `select * from hypopg.create_index('create index on table (column)');` and `select * from hypopg.reset();`.

This type of indexes will be considered as part of the query plan provided by PostgreSQL when running the `explain` command\(^5\) for a given query, confirming whether using the index is a good alternative against other plans. On the contrary, given that the index is not physically created, it is not considered when running `explain analyze`, given that this command performs the actual query and calculates the execution cost.

In Chapter 4 we describe how this type of indexes were useful during our research in order significantly reduce the training and evaluation time of our experiments.

3.5 Index Advisor

The Index Advisor that we used during our research was Dexter\(^6\) which relies on HypoPG. It works in two phases, first it collects queries and then provides index recommendations. It collects queries, either by:

\(^4\)https://hypopg.readthedocs.io/en/latest/
\(^5\)https://www.postgresql.org/docs/11/using-explain.html
\(^6\)https://github.com/ankane/dexter
3.6 Summary

- Passing a single SQL statement to it.
- Passing SQL files with the queries.
- Collecting real time queries by piping log file to it.

The index recommendation process consists of 5 stages.

- Parses the queries, and groups queries with the same parse tree and different values using a hash table.
- Analyzes the tables in order to get the latest statistics from the database.
- Calculates the initial cost of the queries.
- Creates hypothetical indexes using HypoPG on columns that are not indexed.
- Re-calculates the cost of the queries, all the hypothetical indexes used during this calculation are recommended.

In Section 4.1 we explain how we used Dexter in order to calculate which indexes provide the highest performance improvements for each of the workloads we use in our experiments.

3.6 Summary

In this chapter, we describe the design of our solution. We start the chapter with the definition of our Research Questions, then, in Section 3.2 we provide a detail explanation of the design and implementation of each component Reinforcement Learning, Environment, Actions and Rewards.

Furthermore, Section 3.2.2 describes the different considerations that we needed to take in order to implement the corresponding agents with each of the DRL frameworks. We conclude the chapter with a description of the workloads that we use as part of our experiments, the reasons behind why we decided to choose a synthetic version of standard benchmarks and a description of some of the tools that we built in order to create the different workloads.

Based on the presented design and described workload generation approach, we present in the following chapter the experimental setup for our study.
3. Design and Query/Workload generations
4. Experimental setup

In this chapter we provide all the details necessary to reproduce the results obtained as part of the research. First, Section 4.1 covers the different experiments and how they are organized. Then, in Section 4.2 we conclude the chapter with the description of the specification of both the hardware and software resources used.

4.1 Experiments

All our experiments make use of the aforementioned DRL Frameworks in Section 2.3.4 and the tools that we described in Section 3.3.2. They can be classified in two categories: Online agents and Learning from Demonstration which we cover in the next sections.

For all our experiments, we implemented an OpenAI Gym Environment\footnote{https://gym.openai.com/docs} as described in Section 3.2. Then we generated a total of 13 workloads\footnote{https://github.com/fabianrbz/thesis/tree/master/workloads} consisting of 25 different queries, 10 dedicated for training and 3 for testing using the Workload Generator defined in Section 3.3. In order to have a higher statistical significance, each experiment was ran three times.

Each experiment consisted of running several iterations of the Agent-Environment interaction using the aforementioned training workloads for training and then switching to the test workloads for the evaluation process.

The Observation Space of our environment consisted of a matrix of 26 rows, one for each of the queries in a workload representing the corresponding selectivity of each column for each query and one row that keeps track of the current indexes. The 16 columns, correspond to each of the columns in the lineitem table. We also limited the amount of possible indexes to 10.
For the cost calculation in each step, we first create the hypothetical index as described in Section 3.4 corresponding to the specified action and run each of the queries using PostgreSQL Explain command. The new cost after performing the step consists of the sum of the cost extracted from the result of running the aforementioned command for each query of the workload. The Reward is calculated as defined in Section 3.2.1.

In order to calculate the best cost for each workload, we ran it through the index advisor (Section 3.5) and selected the 10 indexes that provided the highest improvements. The best cost is calculated for each workload using the same logic as described in the previous paragraph, with the caveat that the recommended indexes were created (hypothetical ones) beforehand.

The reason behind using hypothetical indexes instead of real ones is the execution time. Creating indexes on a table of the size of lineitem in each step and dropping them after each episode is extremely time consuming. The same reason holds for using Explain instead of Explain Analyze, which performs the query and provides the actual execution time instead of a cost estimate based on statistics.

Our first implementation, created actual indexes and performed Explain Analyze but after running just one iteration of one of our experiments, we realized that it would have taken almost 5 years to conclude all the experiments that we proposed. This is a compromise that we had to make in order to be able to run the experiments. We know there are limitations from learning from estimates as described in Section 2.2.2 but the actual running time of our original experiments made it very impractical to consider.

Given that we want to see how well different agents perform, this compromise is acceptable, considering that in the case the agents are able to learn and to perform reasonably well out of estimates, we should get more accurate results from actual indexes and execution times. This should be possible to achieve with a more powerful computer.

4.1.1 Online agents

We performed the same experiment using the same training and testing workloads for Dopamine using the DQN and Rainbow agents and Ray with its DQN agent. The configuration of the different agents remained as similar as possible across the different frameworks and is presented in the next chapter when we present the experiments.

4.1.2 Learning from Demonstrations

As described in Section 2.3.3 for this type of approach we needed to generate demonstrations of index selections performed by an expert. For this, we generated batches of demonstrations using Ray’s API for offline datasets simulating and Action-Environment interaction.

Each demonstration contains, the action taken, the corresponding reward and the representation of the environment. This demonstration is serialized into a JSON object

by the API, which later can be fed into one of Ray’s agents and used for training an imitation agent.

The batches were generated by using the training workloads and the recommendations provided by the index advisor which in this case acts as the expert. For each workload, the index advisor provides an index recommendation (or none) for each of its queries. These recommendations are treated as the actions performed by the agent as part of the Agent-Environment interaction. We wrote a script that takes the workloads and the corresponding recommendations as input, an outputs a set of demonstrations, containing the simulated interactions.

4.2 Experiments environment

All the experiments were executed on a machine which runs Linux x86_64 kernel version 2.6.32.59 with 72 Intel(R) Xeon(R) Gold 6150 CPU @ 2.70GHz CPUs and 376 GB RAM.

Python version 3.7 and miniconda[^4] were used to set up the different packages and dependencies for managing the different DRL frameworks. Dopamines version 2.0.5 and Ray’s latest version of master at the time of this writing[^5] were used. The reason to point to the latest master was due to some changes that were not included in any release, which were needed in order to run MARWIL that were added to the Ray project after discussions with Ray’s team in their community slack[^6].

For a more detailed information about the packages used for each experiments refer to [https://github.com/fabianrbz/thesis-experiments/tree/master/envs](https://github.com/fabianrbz/thesis-experiments/tree/master/envs).

4.3 Summary

We started this chapter with Section 4.1 providing a detailed explanation of the type of experiments we conducted, how the necessary data was generated and the compromise we needed to make in order to conduct our research.

Then, in Section 4.2 we provided a thorough description of the system in which the experiments were conducted and the software requirements needed in order to reproduce our results.

In the next chapter, we will present the results of our experiments, the configuration of each of the agents, and an analysis of the observations taken.

[^4]: https://docs.conda.io/en/latest/miniconda.html
[^5]: https://github.com/ray-project/ray/commit/f0e62d733fa025e152451b9ff9d90a59510abc7aa
[^6]: https://forms.gle/9TSdDYUg7s88A9e8
4. Experimental setup
5. Evaluation and Results

In this chapter we cover the experiments conducted as part of our research, the remaining details about the different agent configurations and provide an analysis based on the observations. Furthermore, we provide answers to our research questions (Section 3.1). Section 5.1 and Section 5.2.1 address the first two questions, how different online agents behave and how they perform inference, whereas Section 5.2.2 addresses how well Learning from Demonstrations perform and how they can help online training.

1. Section 5.1 we present the setup and corresponding results for the experiments conducted using Dopamine framework and DQN and Rainbow agents.

2. Section 5.2 is divided in two parts. Section 5.2.1 covers the online experiments using the Ray framework and DQN as agent, which served as foundations for learning from demonstrations. Section 5.2.2 is dedicated to learning from demonstrations, the different experiments conducted and how they helped improving the training of different online agents.

3. Finally, we finish this chapter with a summary of the most significant observations we made from the conducted experiments.

5.1 Dopamine

In this section we present the experiments formulations for the different agents using the Dopamine framework (Section 2.3.4.1), the results obtained for each of the three runs together with an in depth analysis based on the observations taken.

Section 5.1.1 covers the results of running the experiments using a DQN agent whereas Section 5.1.2 explains the results obtained while using the Rainbow agent. All the experiments conducted were described in Section 4.1.

1 https://github.com/google/dopamine/tree/master/dopamine/agents/dqn
2 https://github.com/google/dopamine/tree/master/dopamine/agents/rainbow
For the experiments we generated 10 training and 3 testing workloads, each consisting of 25 queries on the lineitem table of the form `SELECT count(*) FROM lineitem WHERE ... AND ...` with up to three predicates per query. A more detailed description was provided in Section 3.3.

Unless stated otherwise, the results presented correspond to three runs of the experiments using different agents, each consisting of 500 iterations of 1000 training steps and one evaluation iteration consisting of 1000 steps.

We expect these two agents to oscillate a lot given their value based nature, although we expect Rainbow to converge faster and provide better performance in general due to more advanced implicit quantiles. Furthermore, we expect Rainbow to have slightly lower performance during evaluation given the extra overhead it has.

5.1.1 DQN

5.1.1.1 Setup

The observation space consisted of a matrix of 26 rows by 16 columns. The first row represent the current set of indexes, and the rest correspond to each of the queries in the workload. The columns correspond to the different columns in the lineitem table.

The neural network we used was the default provided by Dopamine’s DQN, which consists of:

- A fully connected network.
- Three convolutional layers of type Conv2D, with 32, 64 and 64 output filters each.
- Two dense layers, the first one with 512 output filters and the last one with 16 output filters corresponding to the possible actions the agent can perform.

The following hyperparameters were used to configure the DQN agent, for a full list of the configuration refer to https://github.com/fabianrbz/thesis-experiments/blob/master/gin_configs/dopamine_dqn.gin. We found these hyperparameters to be the best, after analysing different configurations.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimizer</td>
<td>AdamOptimizer</td>
</tr>
<tr>
<td>learning rate</td>
<td>0.00025</td>
</tr>
<tr>
<td>epsilon</td>
<td>0.0003125</td>
</tr>
<tr>
<td>discount factor</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 5.1: Values of the DQN Hyperparameters
We had mixed expectations for this experiment. On the one hand, given that DQN is an advanced method of Reinforcement Learning, we expected it to converge. On the other hand, since:

- It only keeps an estimation of the Q-value rather than its distribution. Which might make it act in a greedily on mistaken values.
- It does not have anything more advanced than epsilon greedy as an exploration policy.
- There is relative similarity in the observation spaces of the workloads, hence, we believed that the agent would not explore smartly and fail at finding the best solutions.

5.1.1.2 Results and observations

As shown in Figure 5.1, none of the runs were able to converge to the optimal policy even after 500 iterations. In fact, averaged across all runs, they reached a suboptimal policy that only provides 50 percent of the possible performance improvements compared to the recommendations given by the index advisor. Furthermore, it can be observed that there is a lot of variability across runs, which can be explained given that the agent that has no built a good understanding of its optimization task. In Figure 5.2, this variability increases, which is understandable since the agent knows less on the testing case.

![Figure 5.1: Average returns of DQN training iterations.](image)

For the evaluation, three testing workloads that were previously unseen by the agent were used, and the average reward obtained on average across the runs was around 35, as depicted in Figure 5.2. This means that the agent was able to improve up to
30 percent of the improvements indicated by the index advisor (See Section 3.5). This results would have been better if the agent was able to find the optimal policy during training. To conclude, the second of our hypothesis holds. The described causes make the agent fail.

Performance-wise, DQN presented an average runtime of 350 minutes during training, overall time of all the 500 iterations. The time per training iteration is on average 42 seconds (350 / 500 * 60). The evaluation on the other hand, had an average of 14.5 seconds across runs. This means that the training was about four times slower than testing, which is understandable given the overhead of training the network and the difference in the amount of workloads.

5.1.1.3 Conclusions

Based on the results shown above, DQN does not seem to be the best option for the problem at hand. Given that it was not able to learn the optimal policy during training, it was expected for it to provide a low evaluation performance. We assume that in order to obtain a better performance, a more adequate fine tuning of the hyperparameters is needed, therefore, in future work we will consider finding the best configuration that could exploit the agent’s potential.
5.1.2 Rainbow

5.1.2.1 Setup

The following hyperparameters were used to configure the Rainbow agent, for a full list of the configuration refer to [https://github.com/fabianrbz/thesis-experiments/blob/master/gin_configs/dopamine_rainbow.gin](https://github.com/fabianrbz/thesis-experiments/blob/master/gin_configs/dopamine_rainbow.gin). We found these hyperparameters to be the best, after experimenting with different configurations.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimizer</td>
<td>AdamOptimizer</td>
</tr>
<tr>
<td>learning rate</td>
<td>0.00025</td>
</tr>
<tr>
<td>epsilon</td>
<td>0.0003125</td>
</tr>
<tr>
<td>discount factor</td>
<td>0.99</td>
</tr>
<tr>
<td>num_atoms</td>
<td>51</td>
</tr>
</tbody>
</table>

Table 5.2: Values of Rainbow Hyperparameters

5.1.2.2 Results and observations

As shown in Figure 5.5, all the runs were able to converge to the optimal policy approximately after 65 iterations. One thing to point out is the fact that after the agent finds the optimal set of indexes, it never degraded to a suboptimal solution during the rest of the experiment, implicating that 150 or 200 iterations were more than enough.

![Figure 5.5: Average returns of Rainbow training iterations.](image)

For the evaluation of the agent, three testing workloads that were previously unseen by the agent were used, and the average reward obtained on average across the runs was
around 40, as depicted in Figure 5.6. This means that the agent was able to improve up to 40 percent of the improvements indicated by the index advisor (See Section 3.5).

It is worth noting, that for one of the runs, the average returns were lower, around 32. Interestingly enough, this run correspond to the one that found the optimal policy faster. There is no correlation between the two facts that we could find that would explain this behaviour. Furthermore, we did not expect Rainbow to oscillate so much, in particular more than DQN.

With respect to performance, the training times for Rainbow presented an average runtime of 370 minutes for the 500 iterations as a whole (time per training iteration 44.4 seconds), whereas the evaluation averaged around 15 seconds. The corresponding runtimes are presented next in Figure 5.7 and Figure 5.8. We can see that Rainbow’s training was 20 minutes slower than DQN’s, which is understandable given the extra overhead that Rainbow has.

**Figure 5.6:** Average returns of Rainbow evaluation iterations.

**Figure 5.7:** Duration of the training iterations in minutes.

**Figure 5.8:** Duration of the evaluation iterations in seconds.

### 5.1.2.3 Conclusions

With respect to our Research Questions (Section 3.1), the results presented before have shown that the online agent Rainbow is a good alternative for the Index Selection
Problem, given the short training times and the fact that it was able to provide on average across runs 40 percent of the performance of that provided by the index advisor. It is worth noting, that some of the runs even reached a 55 percent improvement. We think that this performance improvement could be increased by having a better training set and not letting the agent overfit the training data by using cross-validation or other prevention measure, but we left that out of the scope of this work and consider it for future work.

5.1.2.4 Comparison between DQN and Rainbow

From the two agents, Rainbow outperforms DQN in terms of the rewards obtained during training, given that it was able to find and converge to the optimal policy after a few iterations, we believe that the number of atoms in Rainbow made the difference.

In terms of the evaluation, none of the agents were able to find the best set of indexes for the testing set. On average, both agents presented similar performances, although, Rainbow’s runs did not oscillated as much and presented more smoothed graphs. In addition, all three runs for Rainbow were close to the same average returns, indicating that the agent performed similar actions across runs.

DQN on the other hand, shows a more erratic behavior during evaluation and considering its poor performance during training (not reaching the optimal policy), the part of the graphs where it reaches high rewards were probably because it found a good set of actions by chance.

5.2 Ray

5.2.1 DQN

5.2.1.1 Setup

The observation space consisted of a matrix of an array of 432 items. This correspond to a linear representation of a matrix of 26 rows by 16 columns (same one described in Section 5.1.1) with an extra row of 16 columns corresponding to the set of taken actions. These flattening of the original environment was needed in order to implement action pruning using Parametric Action Space.

The following hyperparameters were used to configure the DQN agent, for a full list of the configuration refer to https://github.com/fabianrbz/thesis-experiments/blob/master/gin_configs/ray_dqn.gin. The default configuration of Ray’s implementation of DQN uses dueling and dual DQN, making it similar to a Rainbow agent, hence, we expect to see less oscillation during training compared to the experiments described in the previous section.
<table>
<thead>
<tr>
<th>optimizer</th>
<th>AdamOptimizer</th>
</tr>
</thead>
<tbody>
<tr>
<td>learning rate</td>
<td>0.00025</td>
</tr>
<tr>
<td>epsilon</td>
<td>0.0003125</td>
</tr>
</tbody>
</table>

Table 5.3: Values of Ray Hyperparameters

As stated in Section 3.2.2, in order to be able to implement action pruning and hence, reduce training times, we had to implement parametric action spaces. The idea is that the set of possible action changes based on the observation, in the case of our environment, we wanted to prevent the duplication of indexes.

An extra environment that wraps the original gym environment had to be implemented, that added a set of the available actions as part of the observation. This set, consists of an array with the same length as columns in the lineitem table and a value of 0 is set to for each of the corresponding action that has been taken. Furthermore a custom Model was implemented, that computes the dot product of the actions taken with the output of the underlying neural network, scaling their probability to 0.

### 5.2.1.2 Results and observations

As shown in Figure 5.9, all the runs were able to converge to policy close to the optimal approximately after 40 iterations and kept that policy until the training finished.

![Figure 5.9: Average returns of DQN training iterations.](https://ray.readthedocs.io/en/latest/rllib-models.html#variable-length-parametric-action-spaces)

As with the previous experiments, the same set of testing workloads were used. Figure 5.10 presents the results from the evaluation phase, the three runs achieved on
average in improvement of 46 percent of the index advisor performance over the same set (graph was smoothed to better interpret the results). We also observed that there is a scoped variability across the different runs.

This results are similar to the ones obtained with Dopamine, although each run seems more stable than with the previous agents. As we previously mentioned, given the similarity of the DQN’s configuration to Rainbow, we didn’t see much oscillation during training, but it is definitely present during evaluation.

In terms of runtime, DQN’s training time averaged around 363 minutes (total time for all the training iterations, average time per iteration was 44 seconds) across runs whereas the evaluation time averaged 15 seconds. These numbers allow us to be able to train using different combinations hyperparameters several times a day in order to find the optimal configuration. These training times were similar to the ones presented by Dopamine’s Rainbow and DQN, in fact they took 10 more minutes than DQN and 10 less than Rainbow.

5.2.1.3 Conclusions

In contrast to Dopamine’s implementation of DQN, Ray’s implementenation was able to find the optimal solution during training as shown in Figure 5.9. Given that we kept...
the most relevant hyperparameters fixed for both frameworks, this is related to different neural network configuration the Ray provides and the 51 $num_{\text{atoms}}$.

In terms of evaluation, both frameworks presented similar results as depicted in Figure 5.2 and Figure 5.10 with the caveat that Ray’s results were more stable.

### 5.2.2 Learning from demonstrations

In order to train an agent using learning from demonstrations we needed to generate batches of experiences as described in Section 4.1.2. We trained an agent using MARWIL with the previously generated experiences and a combination of different hyperparameters.

Out of the different configurations, we selected the two that presented the best results and then proceed to train an online agent to see the impact that offline training can have in training times. Due to limitation of the framework, only Policy Gradients were used for the online part of the experiments.

The other hyperparameter that we fixed was the $input_{\text{evaluation}}$. We set the value to $simulation$ given that the implementation of our environment supported this feature. With this configuration, Ray runs background simulations to estimate the current performance of the policy. We leave the training using the other values of this hyperparameter for future work.

#### 5.2.2.1 Setup

The following table depicts the different combinations of the hyperparameters used to train the agent from the offline experiences, their meaning is explained in the following paragraphs. All runs used the default batch size (2000) and were stopped once they reached an average reward mean of 1k (graphics where shortened so they could be better observed). We experimented with different stopping criteria, but found this one to be the best, given that the others tended to overfit the model.
<table>
<thead>
<tr>
<th>Experiment</th>
<th>beta</th>
<th>vf_coeff</th>
<th>batch mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run 1</td>
<td>0.0</td>
<td>0.0</td>
<td>complete</td>
</tr>
<tr>
<td>Run 2</td>
<td>0.0</td>
<td>0.0</td>
<td>truncate</td>
</tr>
<tr>
<td>Run 3</td>
<td>0.0</td>
<td>0.5</td>
<td>complete</td>
</tr>
<tr>
<td>Run 4</td>
<td>0.0</td>
<td>0.5</td>
<td>truncate</td>
</tr>
<tr>
<td>Run 5</td>
<td>0.0</td>
<td>1.0</td>
<td>complete</td>
</tr>
<tr>
<td>Run 6</td>
<td>0.0</td>
<td>1.0</td>
<td>truncate</td>
</tr>
<tr>
<td>Run 7</td>
<td>0.5</td>
<td>0.0</td>
<td>complete</td>
</tr>
<tr>
<td>Run 8</td>
<td>0.5</td>
<td>0.0</td>
<td>truncate</td>
</tr>
<tr>
<td>Run 9</td>
<td>0.5</td>
<td>0.5</td>
<td>complete</td>
</tr>
<tr>
<td>Run 10</td>
<td>0.5</td>
<td>0.5</td>
<td>truncate</td>
</tr>
<tr>
<td>Run 11</td>
<td>0.5</td>
<td>1.0</td>
<td>complete</td>
</tr>
<tr>
<td>Run 12</td>
<td>0.5</td>
<td>1.0</td>
<td>truncate</td>
</tr>
<tr>
<td>Run 13</td>
<td>1.0</td>
<td>0.0</td>
<td>complete</td>
</tr>
<tr>
<td>Run 14</td>
<td>1.0</td>
<td>0.0</td>
<td>truncate</td>
</tr>
<tr>
<td>Run 15</td>
<td>1.0</td>
<td>0.5</td>
<td>complete</td>
</tr>
<tr>
<td>Run 16</td>
<td>1.0</td>
<td>0.5</td>
<td>truncate</td>
</tr>
<tr>
<td>Run 17</td>
<td>1.0</td>
<td>1.0</td>
<td>complete</td>
</tr>
<tr>
<td>Run 18</td>
<td>1.0</td>
<td>1.0</td>
<td>truncate</td>
</tr>
</tbody>
</table>

Table 5.4: Values of MARWIL’s Hyperparameters

The description of the different hyperparameters is as follows, taken from Ray’s documentation:

- **beta**: Scaling of advantages in exponentials terms, when 0 MARWIL is reduced to imitation learning.

- **batch_mode**: Whether to rollout complete_episodes or truncate_episodes. Episode truncation guarantees more evenly sized batches, but increases variance as the reward-to-go will need to be estimated at truncation boundaries. When truncating episodes, the value estimate at the end of the batch is set to 0.0 which could increase the error of the value estimates. Given that the batch size is 2000 steps and our episodes have a maximum amount of 10 steps, we don’t believe that the error introduced could be significantly large.

- **vf_coeff**: Balances value estimation loss and policy optimization loss.

- **input simulation**: Specifies how to evaluate the current policy. Possible values are:

– wis: weighted step-wise importance sampling estimator.
– is: step-wise importance sampling estimator.
– simulation: runs the environment in the background, but uses this data for evaluation only and not for learning.

We expect that when beta is 0 that the agent will convert, given that Marwil is reduced to imitation learning. However, we do not know the impact that batch mode and vf_coeff can have, as they have not been reported, but we expect a vf_coeff of 0.5 to perform well as it balances between policy and the value estimation.

5.2.2.2 Hypothesis

- We expected all the different runs to converge, and to be faster than the online training.
- When Beta is 0, we expected it to train faster and to find a quick convergence.
- Beta 0.5 to show a state that is a middle point between 0 and 1.
- The value of the batch mode does not have an impact on the results.
- We expected that the higher the value of vf_coeff, the longer it would train, due to having a higher loss function to minimize.

5.2.2.3 Results and observations

In this section, we analyze the results obtained for the different experiments. We focus our analysis on the impact the different hyperparameters have on training times and convergence. We split the analysis depending on the value of beta to facilitate the understanding of the results. This hyperparameter correspond to the beta depicted in Equation 2.23.

Beta 0

When beta is set to 0, MARWIL is reduced to imitation learning. In this section, we cover Runs 1 to 6 as depicted in Table 5.4 and provide an analysis of the impact they have. Figure 5.13 presents the results of all the runs. The vertical axis corresponds to the mean episode reward, whereas the horizontal axis corresponds to the wall time in hours relative to the first data point since the training run started (applies to all the graphs after this point.)
Table 5.5 provides a mapping between the runs presented in Table 5.4 and the graphs shown in Figure 5.13. It is important to note that all Runs were able to converge to a mean rewards of 100, hence achieving the optimal policy for the provided training set.

As predicted in our hypothesis Section 5.2.2.2, we conclude that $vf_{coeff}$ has a significant impact on the training times. According to Table 5.5, the Runs that set this hyperparameter to 0 had training times of circa 1m 10s in comparison to the rest of the Runs which around 20 or 30 minutes.

The $vf_{coeff}$ impacts the total loss of the agent as follows:

$$total\_loss = p_{loss} + vf_{coeff} * v_{loss}$$

(5.1)

where the $p_{loss}$ corresponds to the policy loss and and $v_{loss}$ to the value loss. This explains why the we saw a significant difference when the $vf_{coeff}$ was set to 0 and other values.

Furthermore, there is approximately an 8 minutes difference between the two batch modes, complete episodes taking the longest. Unfortunately, here is not an explanation for this behavior, in theory truncate episodes should have a lower performance than complete episodes considering the possible error that could be introduced to the value error during the last episode of the batch, taking it longer to converge. After discussing this issue with Ray’s team, they suggested that there might be an issue in their implementation of Marwil that under-performs when complete episodes is set.
5. Evaluation and Results

Beta 0.5

Figure 5.14 in combination with Table 5.6 depict the results obtained for the experiments when beta was set to 0.5. As with the previous value, all the Runs were able to converge to a 100 after certain time.

Figure 5.14: Episodes reward mean for beta 0.5.

<table>
<thead>
<tr>
<th>Colour</th>
<th>Run</th>
<th>vf_coeff</th>
<th>batch_mode</th>
<th>Conver.</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orange</td>
<td>7</td>
<td>0</td>
<td>complete</td>
<td>1m 23s</td>
<td>208.5k</td>
</tr>
<tr>
<td>Light blue</td>
<td>8</td>
<td>0</td>
<td>truncate</td>
<td>1m 6s</td>
<td>9.498k</td>
</tr>
<tr>
<td>Blue</td>
<td>9</td>
<td>0.5</td>
<td>complete</td>
<td>30m 34s</td>
<td>4.086M</td>
</tr>
<tr>
<td>Pink</td>
<td>10</td>
<td>0.5</td>
<td>truncate</td>
<td>22m 53s</td>
<td>89.03k</td>
</tr>
<tr>
<td>Red</td>
<td>11</td>
<td>1</td>
<td>complete</td>
<td>30m 56s</td>
<td>4.083M</td>
</tr>
<tr>
<td>Green</td>
<td>12</td>
<td>1</td>
<td>truncate</td>
<td>22m 57s</td>
<td>165.5k</td>
</tr>
</tbody>
</table>

Table 5.6: Colour-Mapping of graphs and runs and training times for beta 0.5.

Once more, when vf_coeff is set to 0, the training times are around 1 minute, and the there is also an 8 minute difference between truncate and complete episodes for vf_coeff values different than 0. We could not appreciate a significant difference on the impact that beta 0.5 has compared to 0, in terms of training times and convergence.

Beta 1.0

Figure 5.15 and Table 5.7 present the results obtained for Runs 13 to 18 which correspond to beta 1.0. Once again all Runs were able to converge to the optimal policy.
5.2. Ray

The most significant observation is that when running Marwil with $\beta$ 1.0, all the Runs finished in under 1m 30s regardless of the values of the other hyperparameters, compared to previous runs where PPO and A3C took between 20 and 30 minutes depending on the value of $\text{batch\_mode}$.

5.2.2.4 Online Training from offline experiences

In order to measure the impact that learning from demonstration can have on the training of an agent, we measured the performance of the following policy gradients agents with and without the help of offline training: PG, PPO and A3C. We have chosen policy based methods, because they are the only ones (at least in Ray’s implementation) that are able to restore previously stored checkpoints, in our cased created with Marwil. Our hypothesis here is that PPO and A3C would perform better than PG, given that they are more advanced policy optimization methods.

First, we set our baseline. We trained all the agents with the same configuration, using the same workloads as before, with the caveat that we trained with the training and testing workloads combined 6. Next, we trained the agents, having their training process

6We should note that we also trained these policy optimization agents with the 10 training workloads (instead of what we report about the 10 training + 3 testing workloads), but we did not find sufficient differences that made this additional study worthwhile of inclusion in our experimental results.

Table 5.7: Colour-Mapping of graphs and runs and training times for beta 1.0.

<table>
<thead>
<tr>
<th>Colour</th>
<th>Run</th>
<th>vf_coeff</th>
<th>batch_mode</th>
<th>Convergence</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orange</td>
<td>13</td>
<td>0</td>
<td>complete</td>
<td>1m 30s</td>
<td>229.4k</td>
</tr>
<tr>
<td>Light blue</td>
<td>14</td>
<td>0</td>
<td>truncate</td>
<td>1m 5s</td>
<td>9.818k</td>
</tr>
<tr>
<td>Blue</td>
<td>15</td>
<td>0.5</td>
<td>complete</td>
<td>1m 32s</td>
<td>226.5k</td>
</tr>
<tr>
<td>Pink</td>
<td>16</td>
<td>0.5</td>
<td>truncate</td>
<td>1m 6s</td>
<td>9.687k</td>
</tr>
<tr>
<td>Red</td>
<td>17</td>
<td>1</td>
<td>complete</td>
<td>1m 48s</td>
<td>273.3k</td>
</tr>
<tr>
<td>Green</td>
<td>18</td>
<td>1</td>
<td>truncate</td>
<td>1m 33s</td>
<td>13.11k</td>
</tr>
</tbody>
</table>

Figure 5.15: Episodes reward mean for beta 1.0.
use the results from the offline experiments as a starting point. This is possible, due to Ray’s ability to save and restore checkpoints\footnote{https://ray.readthedocs.io/en/latest/tune-usage.html#save-and-restore} for a given model.

Checkpoints allow to save the state of a model throughout and at the end of the training. Checkpoints for all the Runs were saved and the ones belonging to the two best Runs were selected as the basis for online training. We then compared the results obtained with the baseline.

**Baseline**

Table 5.8 presents the different configurations that were used for all the experiments (Baseline and restoring from checkpoints). These Runs were performed without action clipping and using 20 workers for parallel sampling.

Figure 5.16 and Table 5.7 present the results obtained as Baseline for the corresponding agents and their corresponding configurations. The vertical axis in Figure 5.16 correspond to the reward and the horizontal axis corresponds to the wall time in hours relative to the first data point since the training run started.

<table>
<thead>
<tr>
<th>Run</th>
<th>Agent</th>
<th>Batch Mode</th>
<th>Evaluation Interval</th>
<th>Timesteps per Iteration</th>
<th>Evaluation episodes</th>
<th>Training Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run 1</td>
<td>PG</td>
<td>complete</td>
<td>50</td>
<td>1000</td>
<td>20</td>
<td>2000</td>
</tr>
<tr>
<td>Run 2</td>
<td>PG</td>
<td>truncate</td>
<td>50</td>
<td>1000</td>
<td>20</td>
<td>2000</td>
</tr>
<tr>
<td>Run 3</td>
<td>PP0</td>
<td>complete</td>
<td>50</td>
<td>1000</td>
<td>20</td>
<td>2000</td>
</tr>
<tr>
<td>Run 4</td>
<td>PP0</td>
<td>truncate</td>
<td>50</td>
<td>1000</td>
<td>20</td>
<td>2000</td>
</tr>
<tr>
<td>Run 5</td>
<td>A3C</td>
<td>complete</td>
<td>50</td>
<td>1000</td>
<td>20</td>
<td>2000</td>
</tr>
<tr>
<td>Run 6</td>
<td>A3C</td>
<td>truncate</td>
<td>50</td>
<td>1000</td>
<td>20</td>
<td>2000</td>
</tr>
</tbody>
</table>

Table 5.8: Values of the hyperparameters
As depicted in Figure 5.16, PPO and A3C confirmed our hypothesis that they would perform better than PG and they even performed better than DQN and Rainbow (on even a bigger set of workloads). However, whereas PPO and A3C converge to the optimal policy, PG converges to a suboptimal policy with a reward averaging 45 regardless of the batch mode.

It is important to note, that the different batch modes, which had a huge impact for creating offline agents, were not as important here.

Table 5.9 provides further details of the different experiments together with the mapping of the different colour lines in the graph with their corresponding Run.

<table>
<thead>
<tr>
<th>Colour</th>
<th>Run</th>
<th>Agent</th>
<th>Mode</th>
<th>Reward</th>
<th>Conver.</th>
<th>Time to finish</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orange</td>
<td>1</td>
<td>PG</td>
<td>complete</td>
<td>46</td>
<td>12m 33s</td>
<td>2h 13m 54s</td>
<td>8.119M</td>
</tr>
<tr>
<td>Blue</td>
<td>2</td>
<td>PG</td>
<td>truncate</td>
<td>45</td>
<td>26m 16s</td>
<td>2h 16m 4s</td>
<td>8M</td>
</tr>
<tr>
<td>Pink</td>
<td>3</td>
<td>PPO</td>
<td>complete</td>
<td>103</td>
<td>8m 6s</td>
<td>4h 34m 51s</td>
<td>8.201M</td>
</tr>
<tr>
<td>Light Blue</td>
<td>4</td>
<td>PPO</td>
<td>truncate</td>
<td>103</td>
<td>8m 42s</td>
<td>3h 23m 40s</td>
<td>8M</td>
</tr>
<tr>
<td>Red</td>
<td>5</td>
<td>A3C</td>
<td>complete</td>
<td>102</td>
<td>40m 43s</td>
<td>2h 56m 30s</td>
<td>17.64M</td>
</tr>
<tr>
<td>Green</td>
<td>6</td>
<td>A3C</td>
<td>truncate</td>
<td>102</td>
<td>55m 58s</td>
<td>3h 2m 58s</td>
<td>17.79M</td>
</tr>
</tbody>
</table>

Table 5.9: Colour-Mapping of the graphs and the Runs.

In the next section, we cover how the different agents behave after restoring from previously stored checkpoints, with a \( v_f\_coeff \) of 0.5 given that we expected it to be a trade-off value. Overall, the results obtained were very similar.

**Restoring from first checkpoint**

The first checkpoint correspond to running Marwil with \( beta \) 1.0, \( v_f\_coeff \) 0.5 and complete episodes. The experiments consisted on running the exact same agents and configurations as shown in Table 5.8 but taking advantage of the offline training.
5. Evaluation and Results

Figure 5.17: Episodes reward mean for PG, PPO and A3C - Checkpoint 1.

<table>
<thead>
<tr>
<th>Colour</th>
<th>Run</th>
<th>Agent</th>
<th>Mode</th>
<th>Reward</th>
<th>Conver.</th>
<th>Time to finish</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>1</td>
<td>PG</td>
<td>complete</td>
<td>97</td>
<td>8m 20</td>
<td>11m 29s</td>
<td>1.046M</td>
</tr>
<tr>
<td>Blue</td>
<td>2</td>
<td>PG</td>
<td>truncate</td>
<td>98</td>
<td>8m 21s</td>
<td>11m 6s</td>
<td>1.031M</td>
</tr>
<tr>
<td>Pink</td>
<td>3</td>
<td>PPO</td>
<td>complete</td>
<td>102</td>
<td>6m 39s</td>
<td>21m 5s</td>
<td>1.049M</td>
</tr>
<tr>
<td>Light Blue</td>
<td>4</td>
<td>PPO</td>
<td>truncate</td>
<td>102</td>
<td>6m 31s</td>
<td>20m 34s</td>
<td>1.031M</td>
</tr>
<tr>
<td>Orange</td>
<td>5</td>
<td>A3C</td>
<td>complete</td>
<td>98</td>
<td>18m 49s</td>
<td>18m 49s</td>
<td>1.992M</td>
</tr>
<tr>
<td>Green</td>
<td>6</td>
<td>A3C</td>
<td>truncate</td>
<td>94</td>
<td>18m 41s</td>
<td>18m 41s</td>
<td>1.899M</td>
</tr>
</tbody>
</table>

Table 5.10: Colour-Mapping of the graphs and the Runs.

Figure 5.17 and Table 5.10 represent the results obtained by the different experiments. The first observation is that the time to finish the experiments was significantly reduced compared to baseline, changing from 2 or 3 hours to 6 to 18 minutes.

PPO and A3C, behaved similar to baseline reaching the optimal policy in the end. The most significant result was the difference presented by PG, which benefited the most from the online training. It was able to find the optimal policy for both batch modes, whereas it only reached a reward of 45 during baseline.

Also note that A3C is the only method that uses the value function, and hence is the one that shows more impact from loading from the checkpoint, since the checkpoint might not display adequate information on the value. In future work we could perhaps investigate if $vf\_coeff$ set to 0 would produce a worst or better A3C agent.

Restoring from second checkpoint

The second checkpoint correspond to running Marwil with $beta$ 1.0, $vf\_coeff$ 0.5 and $truncate\_episodes$ which showed one of the best results during offline training. The experiments consisted on running the exact same agents and configurations as shown in Table 5.8 but taking advantage of the offline training.
Figure 5.18: Episodes reward mean for PG, PPO and A3C - Checkpoint 2.

<table>
<thead>
<tr>
<th>Colour</th>
<th>Run</th>
<th>Agent</th>
<th>Mode</th>
<th>Reward</th>
<th>Converg.</th>
<th>Time to finish</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue</td>
<td>1</td>
<td>PG</td>
<td>complete</td>
<td>97</td>
<td>11m 39s</td>
<td>11m 39s</td>
<td>845.8K</td>
</tr>
<tr>
<td>Orange</td>
<td>2</td>
<td>PG</td>
<td>truncate</td>
<td>97</td>
<td>11m 15s</td>
<td>11m 15s</td>
<td>831K</td>
</tr>
<tr>
<td>Light Blue</td>
<td>3</td>
<td>PPO</td>
<td>complete</td>
<td>102</td>
<td>6m 14s</td>
<td>21m 2s</td>
<td>848.7K</td>
</tr>
<tr>
<td>Red</td>
<td>4</td>
<td>PPO</td>
<td>truncate</td>
<td>101</td>
<td>6m 22s</td>
<td>21m 11s</td>
<td>831K</td>
</tr>
<tr>
<td>Green</td>
<td>6</td>
<td>A3C</td>
<td>complete</td>
<td>92</td>
<td>18m 41s</td>
<td>18m 41s</td>
<td>1.783M</td>
</tr>
<tr>
<td>Pink</td>
<td>5</td>
<td>A3C</td>
<td>truncate</td>
<td>95</td>
<td>18m 39s</td>
<td>18m 39s</td>
<td>1.585M</td>
</tr>
</tbody>
</table>

Table 5.11: Colour-Mapping of the graphs and the Runs.

Figure 5.18 and Table 5.11 present the results obtained by the different experiments. Similar to the previous case, restoring the model that was trained offline showed a significant improvement in the training time.

PPO and A3C, behaved similar to baseline by finding the optimal policy in the end. The most significant result was the difference presented by PG, which again was able to find the optimal policy by taking advantage of learning from demonstrations.

5.2.2.5 Conclusions

From the results presented above, it is clear the impact that learning from demonstrations can have in improving the training times of PG, PPO and A3C from hours to minutes, but the most significant observation is how it helped PG to converge to the optimal policy.

5.3 Summary

In this this chapter we presented all the experiments conducted as part of our research and an analysis of the corresponding observations, presenting answers to all our research questions (See Section 3.1).
First, we evaluated different online agents using state-of-the-art Reinforcement Learning frameworks for solving the Index Selection Problem. We tested with DQN and Rainbow using Dopamine framework and DQN with Ray. Rainbow’s DQN was not able to converge and had a lot of variability across runs. On the other hand Rainbow, and Ray’s DQN (which has a setup similar to Rainbow) was able to converge to the optimal solution with less oscillation during training. However, none of the approaches performed well during the evaluation phase.

Then we conducted several experiments with different configurations using Learning from Demonstrations in order to find the best combination of hyperparameters and the impact they have on training. All experiments were able to converge to the optimal solution, regardless of the value of $\beta$. However, the higher the value (0 being imitation learning and 1 full on Marwil), the lower the training times were. Furthermore, we analyzed the impact that $\text{batch\_mode}$ and $\text{vf\_coeff}$ had on training. The higher the $\text{vf\_coeff}$ value, the longer it took the experiments to run and converge, due to the impact it has in the total loss.

Next, we trained three different policy optimization agents and saw how Learning from Demonstrations can help during training. First, we defined our baseline and trained PG, PPO and A3C with a combination of our training and testing workloads. As a result, only PPO and A3C were able to converge to the optimal policy, whereas PG converged to a suboptimal policy. Then we trained the same agents, but starting from checkpoints that were previously generated by Marwil runs. To our surprise, the training times were reduced from a couple of hours to a range between 10 and 20 minutes. Furthermore, all the agents were able to converge to the optimal policy, including PG which failed to do so during the Baseline.

In the next chapter we present work involving the use of Deep Reinforcement Learning in Database related tasks, that is related our research.
6. Conclusion and Future work

In this chapter, we summarize the main contributions of our research and provide some possible directions for future work that we would like to explore, but due to time constraints we could not conduct.

6.1 Conclusions

In this thesis we evaluated DRL applications for the Index Selection problem. To this end we started by carrying out a focused literature review that sought to present to the reader the necessary concepts to understand our experimental design and study. We reviewed recent work applying machine learning to the index selection task, the core concepts of reinforcement learning, alongside deep reinforcement models such as DQN, Rainbow, policy gradient, PPO and A3C. Based on this, we described the main approaches for learning from demonstrations: imitation learning, DLFD and MARWIL. We also presented in detail the 2 DRL frameworks which we used for our study. To conclude our review we discussed two further areas of application of DRL in data management, seeking to show to the reader that there are additional areas of application related to our research.

For our evaluation we designed synthetic workloads over the TPC-H lineitem table, in a manner similar to the described in related work [SSD18, WSY19, SKE+18, MNM+19], for which we selected 10 workload consisting of 25 different queries each for our training, and 3 held-out workloads of 25 different queries each for our testing. Based on this we created an environment, designing the stopping criteria, rewarding scheme, action and observation spaces, in order to model the index selection task as a DRL problem. We specifically decided on using the PostreSQL optimizer estimates and the HypoPG tool, for creating hypothetical indexes, to accelerate our training. Based on this we developed a complete system that used Ray and Dopamine as DRL frameworks. We also created logs the interaction with an index advisor, Dexter, for training from demonstrations, and created a component able to map the logs to the Ray format.
In our experiments, first we analyzed whether two different online agents DQN and Rainbow were suitable for the Index Selection Problem. We found Rainbow to be better for modelling the distribution of the problem, given that it was able to learn the optimal policy during training and provide on average 50 percent of the improvements recommended by the index advisor during evaluation. In contrast, plain DQN was not able to find the optimal policy during training. We believe that this was due to requiring more training time and more fine-tuning of the hyperparameters. On another implementation of DQN with double DQN and a dueling architecture, we were able to find better results for the online training. In terms of time we observed comparable training and inference times across frameworks, this is understandable since the interaction with the query optimizer serves as a common overhead for the performance.

Second, we explored different configurations for a Marwil agent that trained from previously generated experiences. The results showed that all the configurations were able to find the optimal policy with relatively low training times, ranging from one minute to half an hour. This method has proven to perform extremely well and with training times so low, fine tuning hyperparameters becomes a simple task.

Finally, we investigated the impact offline training can have in improving online training. We stored checkpoints for the runs that presented the best results. Then, we trained several online agents with the same configurations and used those results as baseline. The same online agents were trained but starting from the previously stored checkpoints. The most significant outcomes were the reduction in training times and the increase in PG performance. As part of the baseline, it was not able to find the optimal policy, in particular its performance was around 50 percent. However, with the addition of offline training it was able to reach the optimal policy in a limited number of steps. Loading from checkpoints also served as a strong starting points for PPO and policy gradient agents.

Similar than the case of loading from checkpoints in the evaluated agents, we hope that our experimental work could serve as a starting point for research that seeks to use offline learning to improve their adoption of DRL agents.

6.2 Future work

As part of this research, we investigated the possibility of using DRL in order to solve the Index Selection Problem. We explored different solutions, ranging from online to offline, and how the latter could improve training times for online solutions.

In future work, we would like to improve our contributions by addressing some of the items listed below:

- Improve existing tools. We found that both Dopamine and Ray are adequate for simple problems or for typical DRL benchmarking problems. But, as soon
as a certain deviation from the basic implementation is required, we found the tools to be too rigid and we were forced to edit their source code in order to implement some of our solutions. Ray’s team was very responsive to our questions in their Slack channel, and they even implemented some of the changes that we needed for our implementation. During our research, we encountered several issues in the implementation of Marwil that required us not only to repeat several experiments but also to be able to draw proper conclusions out of the experiment results. Hence, in the future, we propose working closely with Ray’s team in order to improve the framework and provide better abstractions for some of their Agents’ implementations (Marwil in particular) in order to make the framework suitable to a wider variety of problems.

- Investigate the possibility of using actual execution times for learning instead of using statistics, i.e. using `EXPLAIN ANALYZE SELECT ..` instead of `EXPLAIN SELECT ..` in order to get more accurate results. During our initial trials, after running one of our experiments using execution times, we realized that it would have taken up to 6 years to run all the experiments that we wanted. We propose to try running the experiments in the same machine but with an SSD instead of a rotating disk which should have a great impact in PostgreSQL performance.

- Extend our training and testing datasets to use real world data and not just use the TPC-H benchmark. We would like to test our solution with actual data from different databases in order to see how it adapts to different types of queries and schemas.

- Further extend our solution to be able to recommend multi-column indexes. One of the restrictions that we had during our research was limiting the recommendation to single column indexes. This compromise was due to the complexity of the problem at hand, and the fact that the feasibility of a solution for the simplest Index Selection Problem was unknown. Furthermore, the fact that the index advisor that we used as reference only consider single column indexes, only limited us further.

- Hyper-parameters fine-tuning. During our experiments we realised the impact that different hyper-parameters can have in terms of training time, convergence and discovery of optimal policies. In particular, as shown in Section 5.2.2 learning from demonstration has proven to be most affected by it. We believe that by further refining the different hyper-parameters and experimenting with different Neural Networks configurations we could achieve better results.

- Experiment with different evaluation techniques for MARWIL. In Section 5.2.2 we used simulation for evaluating the policy while reading offline experiences. Ray provide two more techniques: Weighted Step-Wise Importance Sampling Estimator (wis) and Step-Wise Importance Sampling Estimator (is). In future work, we would like to see how the agent behaves with these two techniques.
6. Conclusion and Future work
Bibliography


