الجمهورية الجزائرية الديمقراطية الشعبية

People's Democratic Republic of Algeria

وزارة التعليم العالي و البحث العلمي

MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

جامعة عمار ثليجي بالأغواط

AMAR TELIDJI LAGHOUAT UNIVERSITY

كلية العلوم

FACULTY OF SCIENCE

DEPARTMENT OF MATHEMATICS AND INFORMATICS

Master Thesis

Domain : Mathematics and Computer science
Field : Computer science
Option : Information and Decision Systems

Submitted By :

Fatma Zahra CHELLAMA

THEME

Query Optimization using Machine Learning Techniques

Defended publicly on 24/06/2024, before the jury composed of:

Pr. Mostapha BOUAKKAZ	Professor	à l'UATL	Président
Pr. Younes GUELLOUMA	Professor	à l'UATL	Examiner
Dr. Lakhdar KECHNA	MAA	à l'UATL	Examiner
Dr. Laradj CHELLAMA	MAA	à l'UATL	Supervisor
Dr. Saida Sara BOUDOUH	MAA	à l'UATL	Assistant Supervisor

Academic Year 2023/2024

Acknowledgments

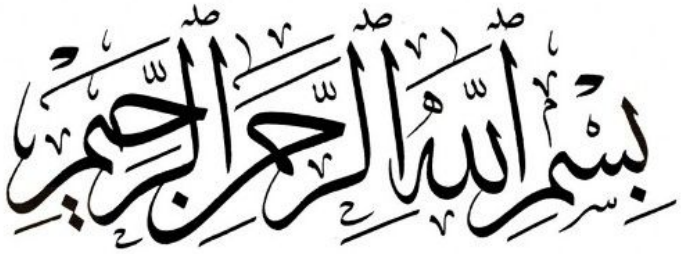
We are with great pleasure that I reserve these few lines as a sign of gratitude to all those who have contributed directly or indirectly to the development of this work.

First of all, I would like to address my most sincere thanks to my supervisor **M. Laradj CHELLAMA** for his availability, patience, and precious follow-up throughout the realization of this work. I would also like to thank the members of the jury for having devoted part of their time to reading this thesis and for their interest in this work.
A special thanks to **Miss. Sara Saida Boudouh** for her exceptional guidance and support. Your wealth of experience has been instrumental in my accomplishments.

My thanks extend to all our teachers in the Computer Science department of Amar Telidji University of Laghouat. Finally, I would like to thank all the people who contributed directly or indirectly to the accomplishment of this work.

Contents

List of Figures


بسم الله الرحمن الرحيم

ملخص

يعد تحسين الاستعلام جانبًا مهمًا في تصميم أنظمة إدارة قواعد البيانات العلائقية (DBMS) ، بهدف إيجاد خطة تنفيذ مثالية عن طريق تقليل وقت التنفيذ الكلي للاستفسارات. مع وضع ذلك في الاعتبار، يتضمن عملنا استخدام نموذج جديد مثل التعلم المعزز العميق (Deep RL) هو مجال فرعي للتعلم الآلي يجمع بين التعلم المعزز (RL) والتعلم العميق لتحسين مناهج تحسين الاستعلام وهي مشكلة NP كاملة. من خلال هذه المهمة، نهدف إلى إعادة تنفيذ وتكييف خوارزميات DRL لإثبات أدائها. نحن نستخدم خوارزمية تحسين السياسة القريبة (PPO) كخالية من النماذج، وتقريبات الدالة الشاملة (UVFA) مع اعادة تجربة الإدراك المتأخر (HER) . سمحت لنا مجموعة الإختبارات المتواضعة على مجموعة بيانات IMDB بمراقبة الأداء التدريجي من خلال تغييرات على اعدادات بارامترات PPO مثل وظيفة التنشيط والاختلاف الطفيف لصالح UVFA .

الكلمات المفتاحية : تحسين الإستعلام ، التعزيز العميق، تدريب الخوارزميات ، PPO ، UVFA ، HER

Abstract

Query optimization is an important aspect in the design of relational database management systems (DBMS), aiming to find an optimal execution plan by minimizing the total execution time of queries. With this in mind, our work involves using a new paradigm such as deep reinforcement learning (Deep RL) is a sub-domain of machine learning that combines reinforcement learning (RL) and deep learning to improve query optimization approaches which is a complete NP problem. Through this task, we aim to reimplemente and adapt DRL algorithms to prove their performance. We use the Proximal Policy Optimization (PPO) algorithm as a model-Free , and the Universal Value Function Approximators (UVFA) with Hindsight Experience Replay. The tests of our modest expreinces on the IMDB dataset allowed us to observe a gradual performance by playing on the hyperparametres of PPO such as the activation function and a slight difference in favor of UVFA with HER.

Keywords: Query Optimization,Deep Learning,PPO,UVFA,IMDB dataset.

Resume

L'optimisation des requêtes est un aspect important dans la conception de systèmes de gestion de base de données (SGBD) relationnels, visant à trouver un plan d'exécution optimal en minimisant le temps total d'exécution des requêtes. Dans cet optique, notre travail consiste à utiliser un nouveau paradigme tel que l'apprentissage par renforcement profond (Deep RL) est un sous-domaine de l'apprentissage automatique qui combine l'apprentissage par renforcement (RL) et l'apprentissage profond pour apporter une amélioration aux approches d'optimisation des requêtes qui est un problème NP complet. A travers cette tâche, nous visons à réimplementer et adapter des algorithmes DRL pour prouver leur performance . Nous avons utiliser l'algorithme Proximal Policy Optimization (PPO) en tant que modéle-Free , et l'algorithme Universal Value Function Approximators (UVFA) avec Hindsight Experience Replay. Les essais de nos modeste expreinces sur le dataset IMDB nous a permis de constater une performance graduelle en jouant sur les hyperparametres de PPO telq que le fonction d'activation et une légere difference en faveur de UVFA.

Mots Clée: Optimisation des requêtes, l'apprentissage en profondeur , PPO, UVFA, IMDB dataset

Chapter 1

Introduction

The goal of the query optimizer is to automatically identify the most efficient execution strategies for executing the declarative sql queries submited by users. the query optimization process produces a query execution plan (qep) which represents an execution strategy for the query. Due to the complexity of the underlying query optimizer, comprehension of a qep demands that a student is knowledgeable of implementation-specific issues related to the rdbms.
Existing database management systems (DBMSs) still choose poor execution plans for some queries to reduce the effect of this problem we want to use and simulate a new approach in the case of deep reinforcement learning for query optimization by building queries incrementally via encoding feautirization of queries using a learned representation.
Several works related to our subject have been cited in the literature that can be subdivided into traditional optimization and that based on learning.

Traditional query optimization

[23] describes how System R chooses access paths for both simple (single relation) and complex queries (such as joins),introduced a number of optimization methods, including the use of dynamic programming for bottom-up join tree creation. It was also the first instance of SQL implementation.
[4] use the traditional cost model to determine the best plan for a given query by generating different strategies using cardinality estimations,These estimations are based on assumptions and database statistics that might or might not be accurate. Poor execution plans result from incorrect cardinality estimation computations or invalid assumptions.
[3] propose a Parametric query optimization (PQO) attempts to solve optimization problem by exhaustively determining the optimal plans at each point of the parameter space

at compile time.
Most DBMSs use histogram-based techniques as part of their cost model to summarize the data of tables to perform efficient selectivity estimations [19],

Learning-Based Query Optimization

Recently, several deep reinforcement learning (DRL) based approaches propose using neural networks to construct a query plan. They demonstrate that efficient query plans can be found without exhaustively enumerating the search space [8].
[27] proposed an execution plan recommendation system based on similarity identification between SQL queries.
Learning Optimizer [24], is introduced as a comprehensive way to repair incorrect statistics and cardinality estimates of a query execution plan by monitoring previously executed queries and computes adjustments to cost estimates and statistics that may be used during future query optimizations.
A novel cardinality estimation approach [9]with the use of machine learning methods that is using query execution statistics of the previously executed queries to improve cardinality estimations and increases the DBMS performance for some queries by several times or by several dozens of times.
ReJOIN model [17] focuses on the Join order selection problem by applying deep reinforcement learning techniques. In this model, the agent learns to maximize the reward through continuous feedback with the help of an artificial neural network.
Neo (Neural Optimizer) presented in [16], uses a supervised learning model to guide a search algorithm through a large and complex space. Neo assumes the existence of a sample workload which consists of a set of queries that is considered a representative of the total workload.
BAO (the Bandit optimizer) presented in [15], is a learned component that sits on the top of an existing query optimizer in order to enhance query optimization rather than discarding the traditional query optimizer. Bao component learns to map the query to the best execution strategy for the query. Then, upon receiving a query, the query optimizer generates multiple plans according to different strategies where the learned model is expected to choose the best query plan given the possible strategies.
the objective of this modest work is to bring (or propose) an improvement for the optimization problem via the use of machine learninge techniques in particular deep learning
.
Our work consists first of all in implementing and modifying some hyperparameters of the PPO algorithm, and secondly, in order to use an extension of the DQN algorithm in this case Universal Value Function Approximators (UVFA) with Hindsight Experience Replay. to achieve the goal, we have subdivided our thesis as follows:

Thesis Structure

- ✔ Chapter 2 delves into the domaine of query optimization, exploring their historical development and reviewing relevant approaches.
- ✔ Chapter 3 serves as an introduction to machine learning and reinforcement learning.
- ✔ Chapter 4 is to describe the methodology used in our approach through the application of deep reinforcement learning. This chapter also covers the problem formulation and reinforcement learning algorithms used.Also we present the findings of our experiments, analyzing the performance and examining their impact on query optimization problem.
- ✔ Finally, Chapter 5 concludes the thesis by summarizing the key discoveries, and proposing potential future research.

Chapter 2

Query Optimization

2.1 What is query optimization?

The process of determining the most effective way to perform a query given the available resources, data, and constraints is known as query optimization. Query optimization involves analyzing the query, data, schema, indexes, statistics, and database system configuration, as well as generating alternative plans for query execution. The database engine then compares the estimated costs and benefits of each plan and chooses the one that minimizes resource consumption and optimizes query performance.

The query optimizer is an essential part of any database management system (DBMS) when it comes to the query evaluation process. By estimating the costs of each project within the range of feasible execution plans, the query optimizer is in charge of identifying the most cost-effective execution plan for any given SQL query. The task of the query optimizer is not trivial, since the algebraic representation of a SQL query can be converted into a set of equivalent expressions [7]. Having said that, as illustrated in Figure 2.1 , every query that needs to be answered must go through a particular DBMS traversal. The functions of the individual modules are as follows, as shown in the above diagram:

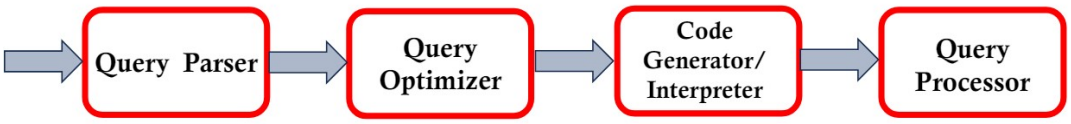


Figure 2.1: Query Flow [7]

- **Query Parser:** validates the query and then transforms it into an equivalent internal form, using relational calculus expressions.
- **Query Optimizer:** generates an efficient execution plan by examining all the

equivalent algebraic expressions produced by the query parser and choosing the optimal one.

- **Code Generator:** converts the optimal execution plan into a set of the appropriate calls to the query processor.
- **Query Processor:** executes the query and produces the final output. Since the core of this project mainly concerns the query optimization process, the following sections will be focused on the modular architecture and functionality of a typical query optimizer

2.2 How to generate an execution plan?

An execution plan is a graphical or textual representation of how the database engine executes a query. It shows the steps, operations, and algorithms that the database engine uses to retrieve, join, sort, filter, aggregate, and generate the data. It also displays the estimated costs, cardinalities and line sizes of each step, as well as the indexes, partitions and statistics used or updated by the query. An execution plan helps you understand how the database engine interprets and optimizes your query, as well as potential performance issues. The figure 2.2 describe that the original query in DML syntax is parsed into a logical expression tree over a logical algebra that is easily manipulated by later stages. This internal logical form of the query then passes to the Query Optimizer, which is responsible for transforming the logical query into a physical plan that will be executed against the physical data structure holding the data. Two kinds of transformations will be performed: **Logical transformations** which create alternative logical forms of the query, such as commuting the left and right children of the tree, and **physical transformations** which choose a particular physical algorithm to implement a logical operator, such as sort-merge join for join. This process generates a large number of plans that implement the query tree. Finding the optimal plan is the main concern of the query optimizer.
Once an optimal (or near optimal) physical plan for the query is selected, it is passed to the query execution engine. The query execution engine executes the plan using the stored database as input, and produces the result of the query as output

2.3 Query Plan Space Enumeration

Using an enumeration approach similar to bottom-up dynamic programming, join order optimization begins with the enumeration of subsets of join orders in the search space. The join procedures are represented as internal nodes in binary trees, or query trees, whose leaf nodes are the relations found in a query's FROM clause. Query trees come in various varieties:

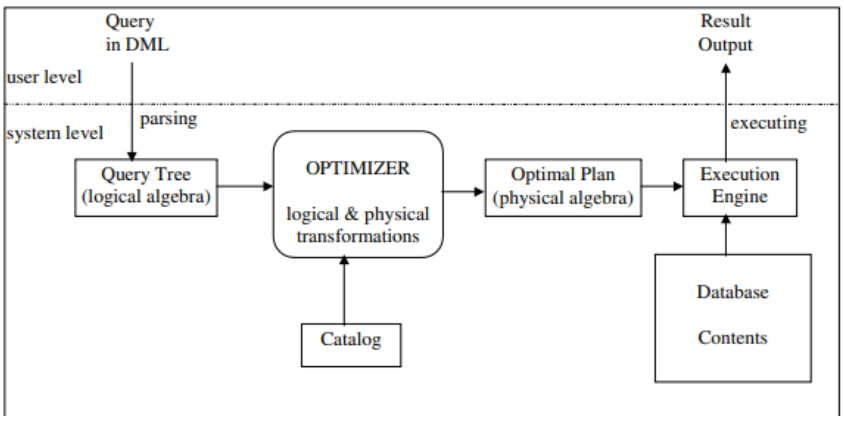


Figure 2.2: Query processing

a **Left-deep trees :** these involve using the join's outcome as an external input for a subsequent join.

b **Right-deep trees:** in these cases, the inner input for the subsequent join is the join result.

c **Bushy trees:** executes a join depending on the outcome of two other joins.

c **Zig-Zag trees:** a blend of deep trees on the left and right.

The problem of finding an optimal join order is same as finding an optimal binary tree whose leaf nodes are the relations in the FROM clause of a query.
Dynamic programming can have the worst time complexity and does not necessarily enumerate all possible tree shapes. While the classical System-R optimizer restricts the search space to only left-deep join trees, systems like PostgreSQL do not consider bushy trees at all.

Optimizers of modern systems or systems that follow System-R architecture, completely ignore join orders with cross products. These restrictions and heuristics significantly reduce the complexity of the search problem. Based on the principle of optimality, dynamic programming enumeration strategies proceed by incrementally constructing optimal subplans. These strategies provide an efficient exploration of the search space as long as the cost of a join is linear in the size of the input relations.
However, in many cases, high intermediate result sizes that are larger than memory and cause partitioning can cause non-linearity in join costs, making it possible that the heuristics based solely on left-deep tree consideration will not work.
In order to regulate the plan alternatives by either restricting them to certain sets or configuring the system to halt the enumeration at a specific threshold, a Database Administrator (DBA) may need to intervene in many commercial systems that employ

additional heuristics.

2.4 complexity:

In a database system, the purpose of a query optimizer is to convert declarative query writing into an effective query execution plan. The task at hand is determining the optimal **QEP** for a given query.

A query optimizer must also take into account aspects of physical query optimization. The simplest way to solve this problem is to take into account all possible permutations of those relations that are involved in the join. This means that every query with **N** relations can be expressed with **N!** possibilities.

Consequently, query optimization becomes increasingly complicated. The number of all possible binary tree forms for a j-way-join-query is $\mathbf{(2j)!\ \ (j!\ \ (j\ +\ 1)!)}$, where a j-way-join has n relations $\mathbf{(j\ =\ n{+}1)}$, in accordance with the Catalan Numbers principle [18].

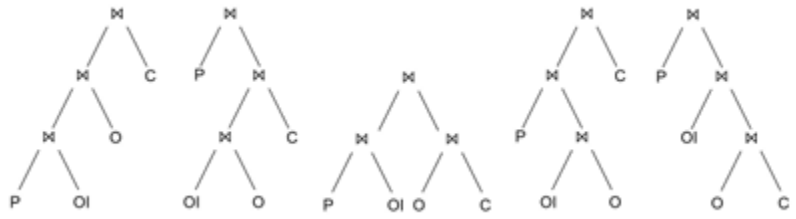


Figure 2.3: Some possible binary-tree shapes for a three-way join-query (P refers to product, OI to orderItem, O to order and C to customer).

2.5 Challanges of query optimizer

Optimization time: In query optimization, determining the best join order is an NP-hard task. As a result, the query optimizer frequently needs to make drastic cuts in the search area and decide on a strategy that is, hopefully, close to the theoretical optimum. One must choose between optimization time and plan performance due to the impossibility of exhaustive search. A difficult engineering task is determining the "sweet spot" between optimization time/resources and plan performance while also fine-tuning the various heuristics. New optimizations usually expand the search space and present new possibilities, necessitating the adjustment of these trade-off choices [6].
Cardinality estimation: A factor that complicates the validation of execution plan optimality is the reliance of the query optimizer on cardinality estimation. Query optimizers mainly rely on statistical information to make cardinality estimates, which is inherently inexact and it has known limitations as data and query patterns become more

complex. Moreover, there are query constructs and data patterns that are not covered by the mathematical model used to estimate cardinalities. In such cases, query optimizers make crude estimations or resort to simple heuristics. While in the early days of SQL Server the majority of workloads consisted of prepared, single query-block statements, at this time query generator interfaces are very common, producing complex ad-hoc queries with characteristics that make cardinality estimation very challenging. Inevitably, test ing the plan selection functionality of the query optimizer depends on the accuracy of the cardinality estimation. Improvements in the estimation model, such as increasing the amount of detail captured by statistics and en hancing the cardinality estimation algorithms, increase the quality of the plan selection process. However, such enhancements typically come with additional CPU cost and increased memory consumption [6].

Cost estimation: Just like cardinality estimation models, cost models utilized by query optimizers are imprecise and incomplete. The query optimizer does not model every aspect of the hardware, runtime environment, or physical data layout. These design decisions can undoubtedly result in dependability issues, but they frequently include fair trade-offs made to avoid extremely complicated designs or to meet optimization time and memory requirements [6].

Chapter 3

Machine Learning: An Overview

Reinforcement learning (RL) is a form of machine learning in which an agent learns to interact with its environment by taking actions that maximize a cumulative reward signal. The goal of reinforcement learning is to train an agent to make decisions by selecting the best possible actions to achieve a specific goal [25]. this Agents learn by receiving feedback from the environment in the form of positive or negative rewards. This type of training helps us a lot when it comes to autonomous driving because we know that autonomous cars live in an uncertain environment and they have to make decisions and actions based on their perception of the world, such as deciding when Brake or accelerate based on current speed and distance to other vehicles. Reinforcement learning allows an agent to learn from its own experience through trial and error. So this helps us a lot when we talk about the scalability of the world and thousands of scenarios where you can't learn from expert data.

3.1 Artificial Neural Networks

Powerful machine learning algorithms belong to the class of artificial neural networks. They consist of numerous interconnected layers of nodes that can learn intricate relationships between input and output variables. The feed forward neural network is the neural network architecture most frequently used to solve regression problems. The building blocks of artificial neural networks (ANNs) are node layers, which have an input layer, an output layer, and one or more hidden layers.
Every node, or artificial neuron, has a weight and threshold that are connected to one another. A node is activated and sends data to the following layer of the network if its output exceeds a given threshold value. If not, that node does not forward any data to the following tier of the network.
As shown in Figure 3.1, one single neuron may be represented. According to this theory, "a single neuron is just the total of all of the inputs times their respective weights, fed

through an activation function."

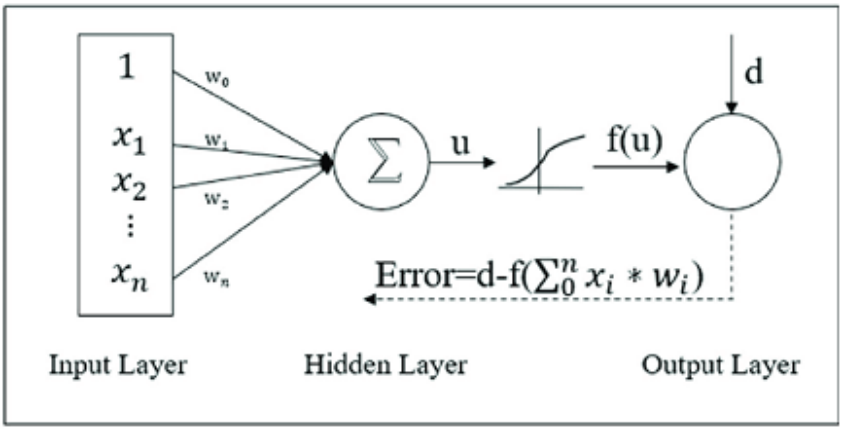


Figure 3.1: General layout of an artificial neural network single neuron

An activation function simulates whether a neuron fires or not. A simple example is the stepper function, where if at some point a threshold is exceeded, the neuron fires a 1, otherwise it fires a 0. It communicates with the next hidden layer. In other words, it sends a 0 or 1 signal to the next neuron, which multiplies the weights.

- ■ Let X be the input data layer of the neuron with **n** input features. Each input feature is multiplied by a weight w_i , resulting in **n** weighted inputs

$$Weighted - input = \sum(X_i * w_i), i = (1..n) \tag{3.1}$$

- The weighted inputs are then aggregated to produce the neuron's total input **u**, given by:

$$u = \sum(X_i * w_i) + b, b \ is \ the \ bias \ term \tag{3.2}$$

- The total input is then passed through an activation function *f(u)*, which introduces non-linearity into the neuron's output. For example, sigmoid activation functions are given by:

$$f(x) = \frac{1}{(1 + exp(-u))} \tag{3.3}$$

- The output of the function f(u) will always be a value between 0 and 1, which can be interpreted as a probability or confidence score for a given input value. There are more Common activation functions including the **Rectified Linear Unit (ReLU)** function, which is half rectified (from the bottom), all the negative values become zero immediately, so it takes max(0, u).

- Then, the output y of the neuron is compared to the expected output using *an error function or loss function*. The error function quantifies the difference between the

neuron's output and the expected output and is used to adjust the weights and bias of the neuron during the training process. For example, *Mean Squared Error (MSE)* is defined as :

$$MSE = \frac{\sum_{i=0}^{N}(y_i - y_i')^2}{N} \tag{3.4}$$

where N is the number of samples, y_i is the actual value, y_i' is the predicted value.The main goal of the optimizer is to minimize the loss function or error function of the network by iteratively updating parameters based on the gradient of the loss function with respect to the weights. The optimizer accomplishes this by using various optimization algorithms, such as **RMSprop and Adam**, to determine how the weights should be updated. Update.

- This process of computing the weighted inputs, aggregating the inputs, applying the activation function, and computing the outputs is repeated for each neuron in the neural network, leading to the final output of the network.

3.1.1 Deep neural networks

Essentially, deep neural networks have multiple layers, and a typical **deep neural network (DNN)** has at least three hidden layers. Furthermore, if there are more hidden layers, the model learns nonlinear relationships between the input and output layers. to be learned. In other words, a deep neural network can have many layers and millions of parameters. This makes them suitable for processing many different types of data. For example, deep neural networks are often used for image recognition, speech recognition, natural language processing, and time series prediction.

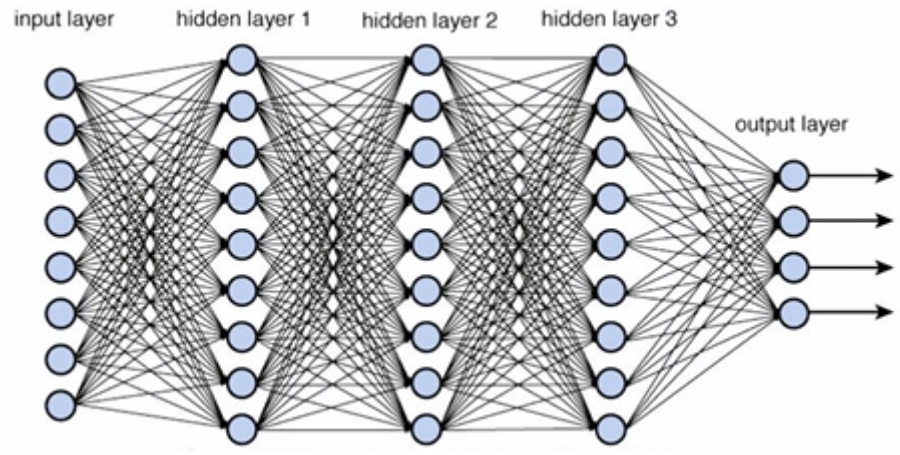


Figure 3.2: deep network architecture with multiple layers[[1]

3.1.2 Dense networks

A high-density neural network (also called a fully connected neural network) consists of multiple layers of interconnected nodes. Each node receives inputs from all nodes in the previous layer and produces outputs that are passed to all nodes in the next layer. High-density neural networks are often used for classification and regression problems such as image classification, language translation, and speech recognition. In autonomous driving Dense neural networks can be used for object detection, road segmentation, and obstacle avoidance.

3.1.3 Convolutional networks

Convolutional neural networks (CNNs) are designed to process data with a lattice topology, such as images and videos. Using a convolutional layer to extract features from the input and a pooling layer to reduce the spatial dimension of the feature map. CNNs achieve state-of-the-art performance in classification, object detection, and segmentation. In autonomous driving, CNNs are used for tasks such as They are used for tasks such as lane detection, object detection, and pedestrian detection.

3.1.4 Recurrent networks

Recurrent neural networks (RNNs) are designed to process sequential data, such as time series data and natural language. Designed to process natural language, RNNs use recurrent connections between nodes to maintain internal state. It remembers previous inputs and produces outputs that depend on the context of the sequence. RNNs are commonly used for language modeling, speech recognition, machine translation Autonomous driving, RNNs can be used for tasks such as trajectory prediction and action prediction.

3.2 Data Pre-processing Techniques

Data preprocessing involves converting raw input data into an appropriate format that enhances the performance and effectiveness of the neural network. The principle of data preprocessing is that a neural network is only as good as the quality and relevance of the input data used to train it. The main techniques used in data pre-processing are :

1. Data cleaning: This is the removal or correction of inconsistencies or missing values in the data set. This prevents potential biases or misleading patterns during training.

2. Data Normalization: Normalization is the process of scaling the characteristics of a data set to a specific range, usually between 0 and 1. This prevents features with larger scales from dominating the learning process and speeds up the convergence

of the neural network. Hastens the convergence of the neural network.

3. Feature scaling: Similar to normalization, feature scaling scales features similarly to avoid bias in the learning process.avoidance. Common scaling methods include standardization and min-max scaling.

4. Handling Categorical Data: Since neural networks typically deal with numeric data, categorical variables need to be properly encoded. This can be done using techniques such as one-hot encoding, where each category is represented as a binary vector.

5. Handling of outliers: Outliers can have a significant impact on the learning process and model performance. Outliers can be handled by removing them or replacing them with more appropriate values.

3.3 Reinforcement learning

As discussed in the previous section, reinforcement learning (RL) is a type of machine learning. Agents learn to interact with their environment by taking actions that maximize their cumulative reward signal. The RL problem can be formally defined as a Markov decision process (MDP), as shown in Figure 3.3.

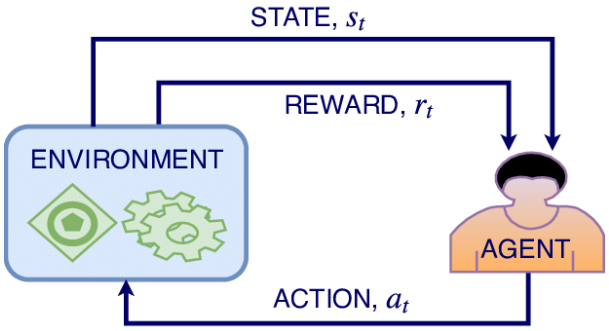


Figure 3.3: Reinforcement learning problem

3.3.1 Markov Decision Process (MDP)

Since the RL problem is a decision-making problem in a stochastic environment, it can be defined as an MDP. The MDP consists of a tuple **(S, A, R, P,** γ**)**, where :

- S is the set of states,
- A is the set of possible actions
- $R(r_t|s_t, a_t)$ is the distribution of current rewards given a (state, action) pair,
- $P(s_{t+1}|s_t, a_t)$is the split between two states (state, action) across the next state
- γ is the discount percentage, which reduces the degree to which future rewards are taken into account in the current action choice.

The MDP has the Markov property, which means that the future is independent of the past given the present. This can be expressed as :

$$P(s_{t+1}|s_t) = (s_{t+1}|s_1, ..., s_t) \tag{3.5}$$

Reinforcement learning involves the agent interacting with the environment by choosing an action **a** based on a policy π that defines the behavior of the learning agent at a given time **t** and maps states to actions.
At the beginning of each episode, the agent starts from an initial state S_0 that is sampled from a distribution $P(s_0)$. The agent then chooses an action according to policy π and receives a reward r_t and the next state s_{t+1},which are sampled from distributions **R** and

P, respectively. Note that reward defines what are the good and bad events for the agent. The agent uses the received reward and the next state to update its policy and improve its decision-making over time. The process is repeated in the form of episodes until the end of the rollout of the policy[20].

3.3.2 Exploration and Exploitation

Collecting the reward signals from as many state-action pairs as you can is the goal of exploration. This entails experimenting with various actions in various states in order to find desired actions that are not offered by any previous body of knowledge. However, since many of the novel actions might not produce a valuable reward, the entire reward may have to be forfeited during the exploration phase. The second task in reinforcement learning is called exploitation, and it entails selecting the optimal course of action in a given state among all attempted courses of action in order to make the greatest use of the knowledge gained during exploration. Exploitation, however, ignores uncharted territory that might hold greater riches.

These two tasks complement each other, but they also form a trade-off, and exploration and exploitation need to be balanced. This balance can be achieved via ε − greedy exploration, where all possible actions with non-zero probabilities are tried out, and the probability ε by which an action is chosen at random is decreased over time to promote exploration at first and then moves towards exploitation as the known action-reward pairs increase in number.[11]

3.4 Reinforcement Learning Algorithms

3.4.1 Bellman Optimality and Q-Learning

First, we have to understand what value functions are, how to deal with future rewards, and what the Bellman optimality principle.

$$V\pi(s_t) = E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2}... \mid s_t, \pi] = E[\sum_{k=0}^{n} \gamma^k r_{t+k}] \tag{3.6}$$

The predicted cumulative discounted reward for taking action a_t in state s_t and then adhering to the policy is represented by the action-value function $Q\pi$ at state s_t and action a_t.

$$\pi : Q\pi(s_t, a_t) = E[\sum_{k>=0} \gamma^k r_{t+k} | s_t, a_t, \pi] \tag{3.7}$$

The action-value function differs from the state-value function only by additionally conditioning on the action. The discount factor γ works the same way as for the state-value function [5].
The optimal state-value function ($V^*(s_t)$ is the best $V\pi(s_t)$ over all policies π:

$$V^*(s_t) = max_\pi V^\pi(s_t) \tag{3.8}$$

Intuitively, it tells us the maximal value we can obtain. Similarly, the optimal action-value function $Q^*(s_t, a_t)$ is the best $Q\pi(s_t, a_t)$ over all policies π:

$$Q^*(s_t, a_t) = max_\pi Q^\pi(s_t, a_t) \tag{3.9}$$

It tells us what we can achieve in the best case if we start at st and have conducted at. These optimal functions tell us what the best possible performance in the MDP would be. In most cases, finding them is computationally intractable since we would have to search the space of all possible policies. Assuming we had Q^* , we would immediately have access to the optimal policy, since :

$$\pi^*(s_t) = argmax_{a' \in A} Q^*(s_t, a') \tag{3.10}$$

To determine Q^* is hard. Since we cannot search over all possible policies. So, it is not as easy as doing a complete search for the optimal behavior in the environment. We can use The Bellman Optimality Equation (BOE) by Richard Ernest Bellmann from the 1953 approach to determine Q*

$$Q^*(s_t, a_t) = E[r_t + \gamma max'_a Q^*(s_{t+1}, a')|s_t, a_t] \tag{3.11}$$

3.4.2 A Taxonomy of RL Algorithms

Whether the agent has access to (or learns) a model of the environment is one of the most significant branching points in an RL algorithm. A function that forecasts rewards and state transitions is what we refer to as an environment model[2].
The primary benefit of having a model is that it enables the agent to plan by anticipating outcomes, examining potential courses of action, and making explicit decisions between options. Agents can then create a learnt policy by combining the outcomes of their advance preparation. One particularly well-known application of this strategy is **AlphaZero**. When this is successful, sample efficiency can be significantly increased compared to approaches without a model.
The primary drawback is that the agent is typically unable to access a ground-truth model of the environment. There are several difficulties if an agent wishes to utilize a model in this situation since it must learn the model only through experience. The main

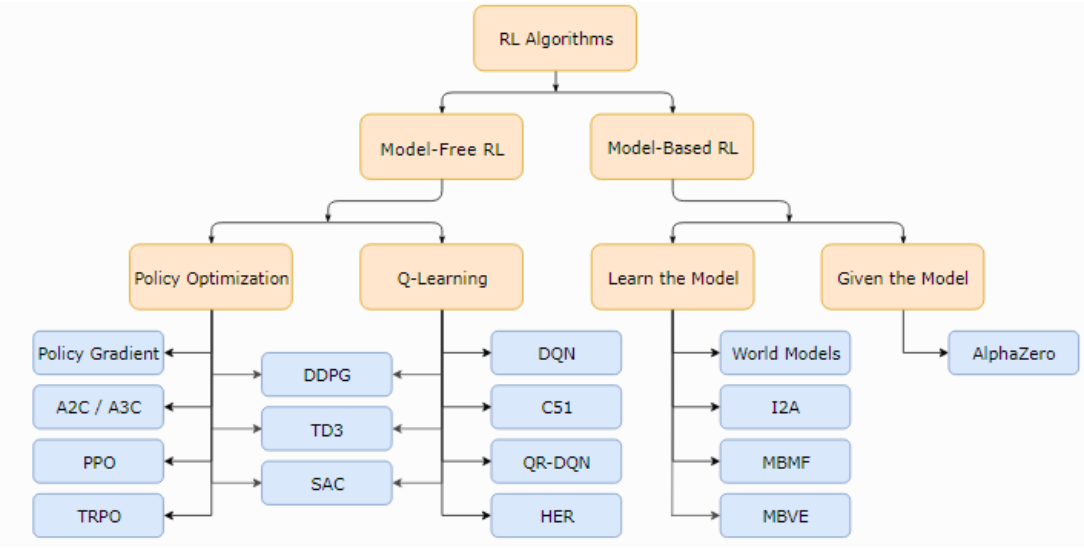


Figure 3.4: Taxonomy of algorithms in contemporary RL

problem is that the agent may be able to take advantage of bias in the model, which might lead to an agent that performs well when compared to the taught model but acts poorly—or extremely poorly—in real life. Due to the basic difficulty of model-learning, even great effort—that is, being prepared to invest a lot of time and computational power—may not be successful.

Model-based algorithms are referred to as **model-based methods**, while model-free algorithms are the opposite. Model-free approaches lack the potential sample efficiency improvements associated with utilizing a model, but they are typically simpler to set up and adjust.

Learn in Model-Free RL

Model-free RL can be used to represent and train agents in two primary ways. [2]:

1. **Policy Optimization:**
   This set of methods explicitly represents a policy as $\pi_\theta(a|s)$. Either directly, by using gradient ascent on the performance goal $J(\pi_\theta)$, or indirectly, by maximizing local approximations of $J(\pi_\theta)$, they optimize the parameters θ. Since this optimization is nearly always carried out on-policy, each update only makes use of data gathered while operating in accordance with the most recent iteration of the policy. In order to determine how to update the policy, policy optimization typically entails learning an approximator $V_\phi(s)$ for the on-policy value function $V^\pi(s)$.Here are a few instances of policy optimization techniques: In order to maximize performance directly, **A2C / A3C** uses gradient ascent; in contrast, **PPO** optimizes performance indirectly through updates that maximize a surrogate objective function that pro-

vides a conservative estimate of the amount that $J(\pi_\theta)$ will change.

2. **Q-Learning**: This family of methods learns an approximation for the optimal action-value function, $Q^*(s, a)$, which is $Q_\theta(s, a)$. The goal function they typically employ is predicated on the **Bellman equation** 3.6. Since this optimization is nearly always done off-policy, every update can make use of data gathered at any time during training, independent of the agent's mode of exploration at the time the data was gathered. The link between Q^* and π^* yields the corresponding policy, and the actions performed by the Q-learning agent are provided by:

$$a(s) = \arg\max_a Q_\theta(s, a).$$

DQN, a classic that significantly contributed to the development of deep reinforcement learning, and **C51**, a variant that learns a distribution over return with an expectation of Q^*, are two examples of Q-learning techniques.

3. **Trade-offs Between Policy Optimization and Q-Learning**: The main advantage of principled policy optimization techniques is that they optimize directly for the desired outcome. They become more steady and dependable as a result. Through training Q_θ to fulfill a self-consistency equation, Q-learning techniques, on the other hand, only indirectly optimize for agent performance. This type of learning is less reliable since there are numerous ways for it to fail.However, because Q-learning approaches may reuse data more efficiently than policy optimization techniques, they have the added benefit of being significantly more sample efficient when they do function.

4. **Interpolating Between Policy Optimization and Q-Learning**: By coincidence, policy optimization and Q-learning are not mutually exclusive—in fact, in some cases, they might be complementary—and a variety of algorithms exist that fall between the two extremes. On this spectrum, algorithms can carefully balance the advantages and disadvantages of either side. Examples are **DDPG**, an algorithm that learns a deterministic policy and a Q-function simultaneously by using each to improve the other, and **SAC**, a variant that stabilizes learning and outperforms **DDPG** on common benchmarks using stochastic policies, entropy regularization, and a few other techniques.

Learn in Model-Based RL

Model-based RL has various orthogonal ways of using models, unlike model-free RL, which has a limited number of easily defined clusters of approaches.

1. **Background:** Just preparation. The simplest method selects actions using pure planning techniques such as model-predictive control (MPC) and never explicitly specifies the policy. Every time the agent observes the environment in MPC, it

computes an optimal plan according to the model, which outlines all the activities to be taken across a predetermined window of time after the present. (The planning algorithm may take into account rewards that are further in the future by using a learnt value function.) After carrying out the plan's initial action, the agent quickly discards the remaining actions. To prevent using an action from a plan with a shorter-than-desired planning horizon, it computes a new plan every time it gets ready to interact with the environment.
The MBMF work investigates MPC on a few common deep learning benchmark tasks using learnt environment models.

2. **Expert Iteration.** Using and learning an explicit representation of the policy, $\pi_\theta(a|s)$, is a simple step beyond pure planning. In the model, the agent use a planning method (similar to Monte Carlo Tree Search) to generate potential plan actions by taking samples from its existing policy. The planning algorithm is a "expert" in relation to the policy since it generates an action that is superior to what the policy alone could have created. After then, the policy is modified to yield an outcome that more closely resembles the planning algorithm's output..

   - The ExIt algorithm trains deep neural networks to play Hex using this method.
   - AlphaZero is another illustration of this methodology.

3. **Data Enrichment** for Approaches Without a Model. Train a policy or Q-function using a model-free RL method, and then either 1) update the agent using only fictional experiences, or 2) update it using only real experiences.
   MBVE for an illustration of adding fictional experiences to actual ones, and **World Models** for an illustration of training an agent only through fictional encounters—a technique they refer to as "training in the dream."

4. **Embedding Planning Loops into Policies** Another method trains the policy's output using any common model-free algorithm while directly embedding the planning process as a subroutine into the policy, turning completed plans into side information. The main idea is that the policy can learn in this framework to decide when and how to employ the plans. Because the policy can simply learn to ignore the model's poor performance for planning in some states, model bias is less of an issue. **I2A** for an illustration of an agent possessing this kind of creativity.

3.4.3 Value-based learning

Q-learning is a popular value-based algorithm used to approximate the optimal Q-values and learn the optimal policy in a model-free manner. Q-Learning can converge to the optimal Q-values under certain conditions and has been widely used in various

applications of reinforcement learning.

Where Q(s, a) is the estimated value (or Q-value) of taking action in state s, and is the learning rate. Basically determines how much the Q-value for a state-action pair is updated based on the difference between the observed reward and the expected reward. A high learning rate means that new information is given more weight, leading to faster learning but also greater instability, while a low learning rate means that old information is given more weight, leading to slower learning but more stability.

3.4.4 Policy-based learning

Policy-based methods are a class of reinforcement learning algorithms that directly parameterize the policy function, which maps states to actions. The policy gradient method is based on the concept of gradient ascent, where the objective is to maximize an expected return or a performance measure, let‘s consider a parameterized policy function $\pi\theta$ that maps states to actions, where θ represents the parameters of the policy. The goal is to find the optimal set of parameters θ that maximizes the expected return $\tau(\theta)$ of the policy based on the total [10].

$$J(\theta) = E\pi_\theta[r(\tau)]$$

The parameters are then tuned based on the gradient of the cost function:

$$\theta_{t+1} = \theta_t + \alpha\Delta\tau(\theta_t)$$

An advantage of using policy-based methods is the possibility of mapping environments with huge, even continuous action spaces. Environments with stochasticity can also be solved with them. Examples of popular deep RL policy-based methods include DDPG(Deep Deterministic Policy Gradient)[13], PPO (Proximal Policy Optimization)[22], A2C (advantage actor-critic)[14]. amsthm algorithm algpseudocode

Chapter 4

Methodology and Experiments

An overview of the setup needed to carry out the experiments indicated by our research questions is given in the following sections. With this chapter, we hope to offer information that will help ensure the reproducibility of our evaluation. We begin by describing the database system and dataset used

4.1 Dataset Generation and Pre-processing

Creating a training and test dataset is the first step in any machine learning task. Given that JOB, which is based on the IMDB database, is being used for benchmarking, it is imperative that our training dataset accurately represents the test dataset. For training and testing, we exclusively used JOB workload in our early trials. we opted to use a new IMDB dataset proposed by Kaiwen Wang in his work [26], this dataset contains 3300 queries consisting of 100 queries for each of the 33 Join Order Benchmark (JOB) models witch is composed of 28 tables and 143 attributes,to produce a big dataset. We tested this dataset by executing the different requests via the PostgreSQl 16.1 DBMS on windows 10 to confirm and display the estimate of cardinalities and intermediate result cardinalities using the EXPLAIN statement with JSON format. All the information obtained for the requests has been saved as a Json file to facilitate their reading. For each template, we randomly selected 60 % queries for training (3300*0.60=1980 queries), 20 % for validation (3300*0.20=660 queries) and 20 % for testing (3300*0.20=660 queries).

4.2 Query Encoding

RL learning algorithms need an environment in which they can interact with the MDP as explained in section 3.3.1,we describe all the components of the agent environment that are necessary to solve a QO problem with reinforcement learning.

4.2.1 Observation State

An observation represents the state in which the agent currently is. Since we want to provide as much information as possible to our learning algorithms, we formulate query optimization as a fully observed RL problem, the database and the queries be encoded in such a way that the representation can be learned by the RL algorithms. The encoded observation serves as input for a neural network (NN) in order to learn from the observations.
We adopted Krishnan and all [12] technique for the encoding, which uses each database field as a separate feature. Every digit in a binary **1-hot vector** that represents a state corresponds to a database column. The vector's size is equal to the total number of columns in all tables.In our case, the observation vector's size is equal to concatenation of query encoding (28 tables) and current join columns (143 columns).

$$\textbf{obs\_size} = \textbf{2*nbre\_columns} + \textbf{nbre\_base\_table}$$

4.2.2 Actions

The agents take actions from all base tables,to construct the action space we take all combinations of all base tables. In our case, the size is equal to 28 tables.

4.2.3 Reward

A reward is defined as the negative costs of a query plan based on the cardinalities et intermediate results.

4.3 Experimental Setup

The IMDB database is created and stored using the PostgreSQL V.16.1 server, and queries are run on the PostgreSQL database using psycopg2 [1] to obtain true cardinalities and PostgreSQL's estimate.
We use Anaconda [2] to set up virtual environments and create an ecosystem of libraries and modules needed for our task on our local PC with an NVIDIA GeForce GTX 1050 Ti GPU. In order to visualize the tensorflow summaries in our experiments, we use Tensorflow V1.14, Cuda 10.0,Spyder 5.4.3 [3] with Python V.3.11, pandas, numPy, matplotlib, and seaborn[4]to pre/post process datasets and results.

[1] https://pypi.org/project/psycopg2/
[2] https://www.anaconda.com/
[3] https://www.spyder-ide.org/
[4] https://seaborn.pydata.org/

4.4 Evaluation of RL algorithms

4.4.1 Proximal Policy Optimization (PPO)

Proximal policy optimization (PPO) developed by John Schulman is an algorithm in the field of reinforcement learning that trains a computer agent's decision function to accomplish difficult tasks.The main advantages of PPO are simplicity, stability, and sample efficiency.

PPO is classified as **a policy gradient method** for training an agent's policy network (function that the agent uses to make decisions). Essentially, to train the right policy network, PPO takes a small policy update, so the agent can reliably reach the optimal solution.

A too-big step may direct policy in the false direction, thus having little possibility of recovery; a too-small step lowers overall efficiency. Consequently, PPO implements a clip function that constrains the policy update of an agent from being too large or too small. There are two primary variants of PPO: PPO-Penalty and PPO-Clip.

PPO-Penalty approximately solves a KL-constrained update like TRPO, but penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient over the course of training so that it's scaled appropriately.

PPO-Clip doesn't have a KL-divergence term in the objective and doesn't have a constraint at all. Instead relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy.

Key Equations:

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{E} \left[L\left(s, s, \theta_k, \theta\right)\right] \tag{4.1}$$

$$L\left(s, s, \theta_k, \theta\right) = min\left(\frac{\pi_\theta\left(a|s\right)}{\pi_{\theta_k}\left(a|s\right)} A^{\pi_{\theta_k}}\left(s, a\right), Clip\left(\frac{\pi_\theta\left(a|s\right)}{\pi_{\theta_k}\left(a|s\right)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}\left(s, a\right)\right) \tag{4.2}$$

in which ϵ is a (small) hyperparameter which roughly says how far away the new policy is allowed to go from the old.

Advantage is positive: Suppose the advantage for that state-action pair is positive, in which case its contribution to the objective reduces to :

$$L\left(s, s, \theta_k, \theta\right) = min\left(\frac{\pi_\theta\left(s|a\right)}{\pi_{\theta_k}\left(s|a\right)}, \left(1+\epsilon\right)\right) A^{\pi_{\theta_k}}\left(s, a\right) \tag{4.3}$$

Advantage is negative: Suppose the advantage for that state-action pair is negative, in

which case its contribution to the objective reduces to

$$L\left(s, s, \theta_k, \theta\right) = max\left(\frac{\pi_\theta\left(s|a\right)}{\pi_{\theta_k}\left(s|a\right)}, (1-\epsilon)\right) A^{\pi_{\theta_k}}\left(s, a\right) \tag{4.4}$$

4.4.2 Universal Value Function Approximators (UVFA)

Universal Value Function Approximators (UVFA) is an extension of DQN to the setup where there is more than one goal we may try to achieve. Let $\mathbf{G}$ be the space of possible goals.Every goal $g \in \mathcal{G}$ corresponds to some reward function $r_g : \mathcal{S} \times \mathcal{A} \longrightarrow \mathcal{R}$.Every episode starts with sampling a state-goal pair from some distribution $p(s_0, g)$. The goal stays fixed for the whole episode. At every timestep the agent gets as input not only the current state but also the current goal :
$\pi : \mathcal{S} \times \mathcal{G} \longrightarrow \mathcal{A}$ and gets the reward $r_t = r_g(s_t, a_t)$. The Q-function now depends not only on a state-action pair but also on a goal : $\mathrm{Q}_\pi\left(s_t, a_t, g\right) = \mathrm{E}\left[R_t|S_t, a_t, g\right]$. [21] show that in this setup it is possible to train an approximator to the Q-function using direct bootstrapping from the Bellman equation (just like in case of DQN) and that a greedy policy derived from it can generalize to previously unseen state-action pairs.

4.4.3 Hindsight Experience Replay

Recently, hindsight experience replay (HER) was proposed to improve the learning performance of Model-free algorithms like DQN,DDPG,SAC,TD3,..., HER is a sample-efficient experience replay method that enhances the performance of off-policy DRL algorithms by allowing the DRL agent to learn from both failures and successes.we propose a DRL algorithm called Universal Value Function Approximators (UVFA) with hindsight experience replay (HER).
The detailed process of HER algorithm 1 is as follows. First, we initialize an off-policy DRL algorithm $\mathcal{A}$ and empty the replay buffer$\mathcal{D}$. In each episode, an initial state s_0 and a goal } are uniformly sampled from the state space $\mathcal{S}$ and the goal space $\mathcal{G}$, respectively. Then, the DRL algorithm $\mathcal{A}$ interacts with the environment during environment steps t = 1,2,...,T to obtain the transition tuple $(s_t, a_t, r_t, s_{t+1}, g)$. After executing the environment steps, HER has knowledge regarding the visited states $\xi = \{s_0, s_1, ..., s_T\}$. Based on this knowledge,HER stores every transition tuple (s_t, a_t, r_t, s_{t+1}) together with the original goal g in the replay buffer $\mathcal{D}$. HER then stores extra transition tuples $(s_t, a_t, \acute{r}_t, s_{t+1})$ together with $\acute{g} \in \phi$, where $\phi = \{\acute{g}_1, \acute{g}_2, ..., \acute{g}_m\}$ is a set of m additional goals uniformly sampled from the visited states ξ.
Through this process, HER provides supplementary rewards $\acute{r} = r\left(s_t, a_t, \acute{g}\right)$ to the DRL agent even if the goal g is not achieved. This process enhances DRL algorithms in terms of learning speed and the success rate of reaching the goal.

Algorithm 1 Hindsight Experience Replay (HER)

Require:

- an off-policy RL algorithm A ▷ e.g.: UVFA
- a strategy S for sampling goals for replay,
- a reward function

Initialize $\mathcal{A}$
Initialize replay buffer $\mathcal{D}$
for *episode* $\leftarrow 1$ to M **do**
  Sample a goal g and an initial state s0
  for $t \leftarrow 0$ to $T-1$ **do**
    Sample an action at using the behavioral policy from $\mathcal{A}$:
    $a_t \leftarrow \pi_b\left(s_t||g\right)$ ▷ ||: Concatenation
    Execute the action a_t and observe a new state s_{t+1}
  end for
  for $t \leftarrow 0$ to $T-1$ **do**
    $\mathrm{r}_t := r(s_t, a_t, g)$
    Store the transition $(s_t||g, a_t, r_t, s_{t+1}||g)$ in $\mathcal{D}$ ▷ standard experience replay
    Sample a set of additional goals for replay G := S(current episode)
    for $\acute{g} \in \mathcal{G}$ **do**
      $\acute{r} := r(s_t, a_t, \acute{g})$
      Store the transition $(s_t||\acute{g}, a_t, r_t, s_{t+1}||\acute{g})$ in $\mathcal{D}$ ▷ HER
    end for
  end for
  for $t \leftarrow 1$ to N **do**
    Sample a minibatch B from the replay buffer $\mathcal{D}$
    Perform one step of optimization using $\mathcal{A}$ and minibatch $\mathcal{B}$
  end for
end for

4.5 Training and Evaluation

4.5.1 Training process: PPO algorithm

PPO makes use of the vectorized architecture, an effective paradigm that has a single learner that gathers data and picks up knowledge from multiple environments.
PPO initializes a vectorized environment **envs=4** that runs **envs** environments either sequentially or in parallel by leveraging multi-processes. The vectorized architecture loops two phases **the rollout phase** and **the learning phase**, for the first one, The agent samples actions for the (envs=4) environments and continue to step them for a fixed number of steps **(num_steps)**. During these steps, the agent continues to append relevant data in an empty list, in the Learning phase,the agent learns from the collected data in the previous phase of length **envs*num_steps**.PPO can estimate value for the next observation **next_obs** and calculate the advantage **advantages** and the return **returns**, both of which also has length **envs*num_steps**.
The weights of hidden layers use orthogonal initialization of weights with scaling np.sqrt(2), and the biases are set to 0,see figure 4.1. For our model architecture , we use

```
def layer_init(layer, std=np.sqrt(2), bias_const=0.0):
    torch.nn.init.orthogonal_(layer.weight, std)
    torch.nn.init.constant_(layer.bias, bias_const)
    return layer
```

Figure 4.1: Layer Initialisation

Actor-Critic separate architecture which is a type of reinforcement learning algorithm that combines aspects of both policy-based methods (Actor) and value-based methods (Critic). This hybrid approach is designed to address the limitations of each method when used individually.
PPO uses a simple multilayer perceptron (MLP) network. The critic network architecture consisting of three layers of 512 neurons and Using GELU as the activation function, the below pseudocode 4.2 describe this architecture

```python
def get_critic(observation_space, output_dim=1):
    input_dim = np.array(observation_space.shape).prod()
    return nn.Sequential(
        layer_init(nn.Linear(input_dim, 512), std=0.1),  # Adjusted weight initialization
        nn.GELU(),  # Using GELU activation function
        nn.Dropout(0.5),  # Added dropout layer with probability 0.5 for regularization
        layer_init(nn.Linear(512, 256), std=0.1),  # Adjusted weight initialization std
        nn.GELU(),  # Using GELU activation function
        nn.Dropout(0.5),  # Added dropout layer with probability 0.5 for regularization
        layer_init(nn.Linear(256, output_dim), std=0.1),  # Adjusted weight initialization
    )
```

Figure 4.2: Critic architecture

We use Adam Optimizer 4.3 with epsilon parameter equal to **1e-5** and learning rate equal to **8e-4**

```python
agent = Agent(envs).to(device)
optimizer = optim.AdamW(agent.parameters(), lr=8e-4, eps=1e-5)
```

Figure 4.3: Adam Optimzer

4.5.2 Training Performance: PPO algorithm

Figure 4.4: Best return of test evaluation

Figure 4.5: Best return of train evaluation

Discussion :

In order to test the performance of the approach of our model we have chosen the following metrics:

Figure 4.6: Best return of validation evaluation

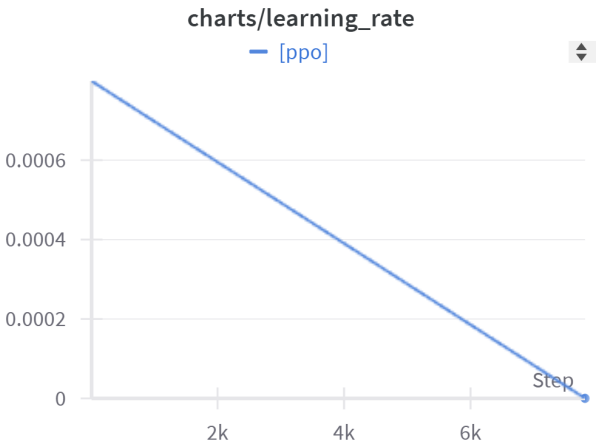


Figure 4.7: Learning rate

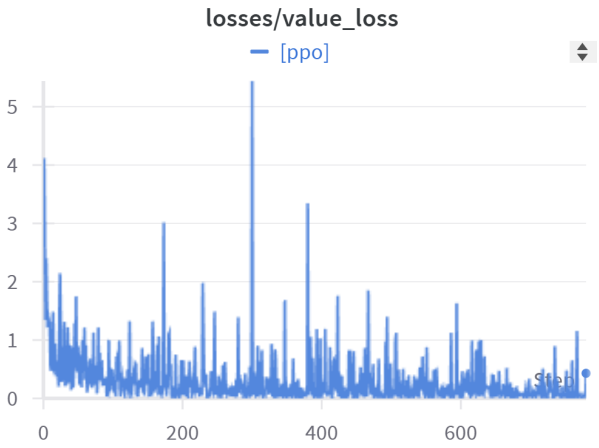


Figure 4.8: Value Loss

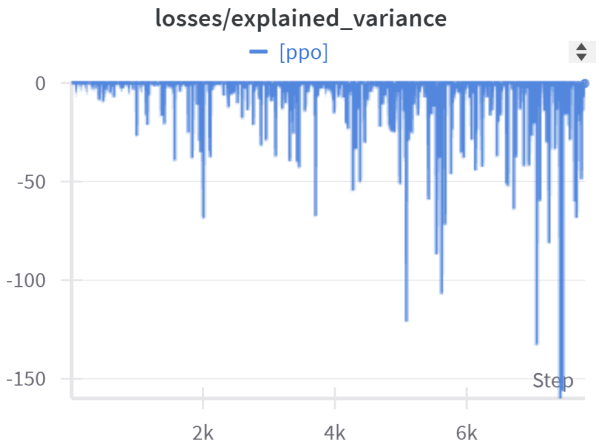


Figure 4.9: Explained variance

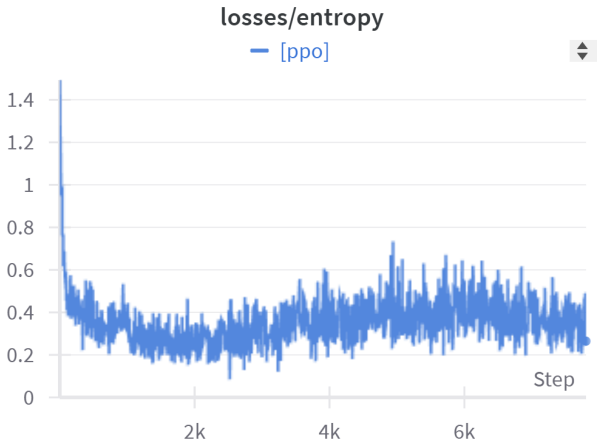


Figure 4.10: Entropy

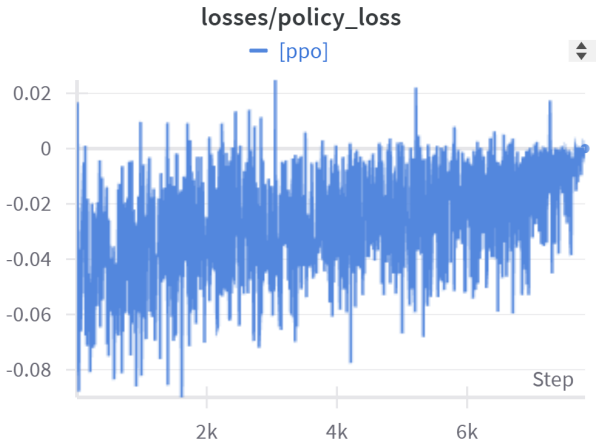


Figure 4.11: Policy Loss

1. **Reward:**
   As shown in figures 4.4,4.5 and 4.6 we find that the model is well formed because the curve of the three graphs are similar and converge.

2. **Learning Rate:**
   This rate indicates the extent to which the training algorithm expands when it seeks optimal policy and is expected to decrease over time.The figure 4.7 shows a rate that decreases over time.

3. **Value Loss:**
   It corresponds to the model's ability to predict the value of each state,the figure 4.8 The figure shows an increase as the agent learns, then decreases once the reward stabilizes.

4. **Policy Loss:**
   It correlates to how much the policy (process for deciding actions) is changing. The figure 4.11 indicates a decrease magnitude this shows a good learning experience

5. **Entropy:**
   Since the model's decisions are random, from figure 4.10, it can be seen that it decreases slowly during a training process, indicating successful learning.

4.5.3 Training process: Hindsight Experience Replay (HER)

The experiments are conducted in the same simple training environment of **PPO**,Training is performed using the Universal Value Function Approximators (UVFA) algorithm, the values of all hyperparameters :

learning rate of the optimizer	3×10^{-4}
the discount factor γ	0.999
Minibatch size	256
Replay memory buffer size	0.25×10^{6}
Activation function	GELU

4.5.4 Training Performance: Hindsight Experience Replay (HER)

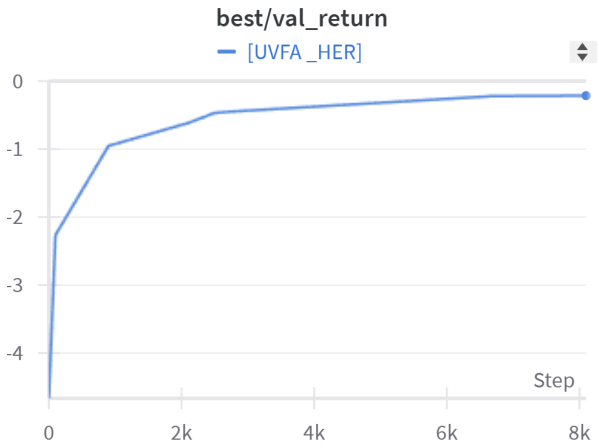


Figure 4.12: best return value- Validation -

Figure 4.13: best return value- Test -

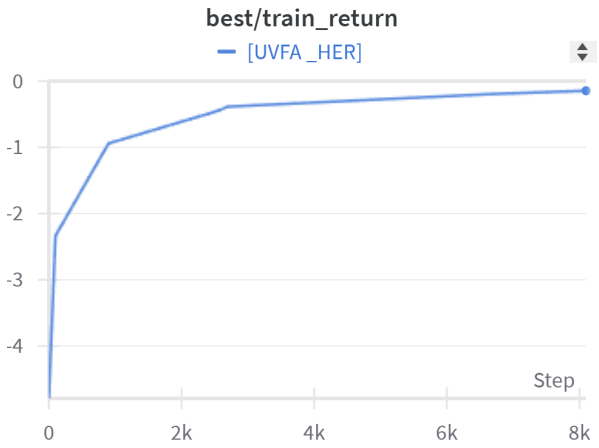


Figure 4.14: best return value- Training -

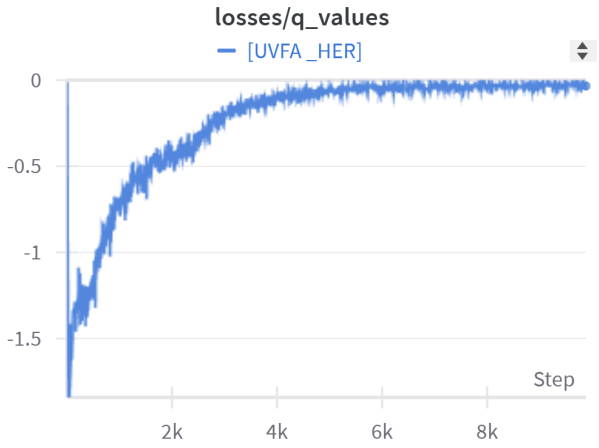


Figure 4.15: Q-values losses

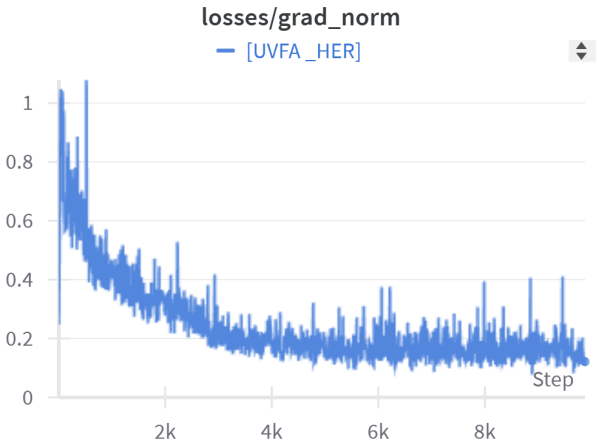


Figure 4.16: Evolution Gradient norm

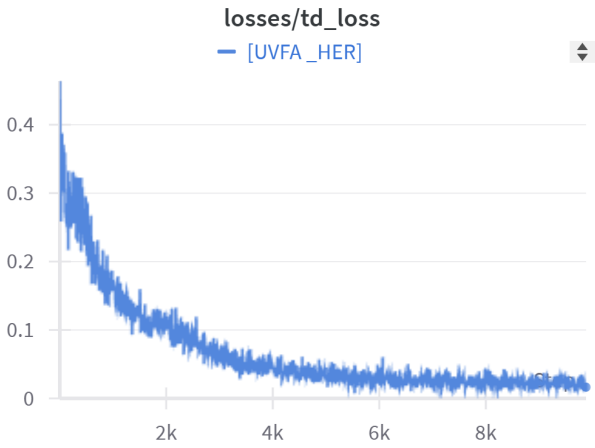


Figure 4.17: TD-Loss

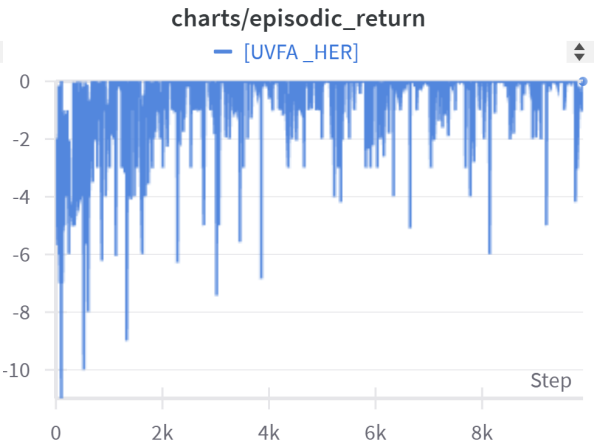


Figure 4.18: Evolution of Episodic return

Discussion:

4.12 - 4.13 - 4.14 :represents the evolution of the best return value (best/val_return) during a training process. This measure captures the value of the best cumulative reward obtained up to a certain point in the learning process. We can observe that the curve starts at very negative values, around -3.5, then increases in a quasi-linear way to reach values close to 0 at the end of the process. This indicates that the agent has been able to learn significantly and gradually improve its performance over time.The shape of the curve, with steady and continuous growth, suggests that the learning process has been relatively stable and effective. This positive evolution of the best return value shows that the agent was able to converge towards increasingly effective strategies, which is an encouraging sign of the effectiveness of the reinforcement learning approach used in this context.

4.15 : The graph represents a series of values called "losses/q_values" according to the "Step" (step). The values appear to fluctuate significantly, with peaks of about -0.5 and troughs of about -1.5. This significant variability likely indicates that the learning model or strategy is undergoing significant changes during the various stages. It seems to show the evolution of some metrics related to the training of a model or agent in a dynamic environment.

4.16 : This graph shows the evolution of the "losses/grad_norm" metric according to the "Step" (step) for the "[UVFA_HER]" model. Great variability can be observed in the values of this metric, with peaks reaching nearly 1 and troughs around 0.2. This large fluctuation indicates that the model undergoes significant changes during training, resulting in significant variations in the gradients used to update the model parameters. It can be said that this metric allows to follow the important

modifications of the model during the training stages.

4.17 : This graph represents the evolution of the "losses/td_loss" metric according to the "Step" (step) for the "[UVFA_HER]" model. This metric measures the loss (or error) of the model during its training. It can be observed that the values of this metric start at a high level, around 0.4, then gradually decrease over the training stages. This downward trend indicates that the model learns and becomes more and more efficient to minimize this specific loss. However, there is also great variability in values, with frequent peaks and troughs. This suggests that the model drive is not linear and has phases of progress and setback, probably related to hyperparameter adjustments. Overall, this graph allows to follow the evolution of the model performance during its training, focusing on the minimization of the loss "td_loss".

4.18 : shows the evolution of the "charts/episodic_return" metric over the steps ("Step") for the "[UVFA_HER]" model. This metric represents the average reward the model achieves for each episode of the learning environment. It can be observed that the values fluctuate significantly, with peaks reaching about 0 and troughs up to -8. This high variability indicates that the performance of the model varies greatly during the different episodes, due to the dynamic nature of the learning environment. Despite these fluctuations, there is a general trend towards improved performance, with episodic returns gradually approaching 0 over the course of the stages. This suggests that the model learns to better navigate this environment and get better average rewards.

Chapter 5

Conclusion

In this thesis, we addressed the subject of relationship query optimisation, which consists in selecting an optimal execution plan. This problem is NP-complete and for this we proposed the integration of machine learning techniques and in particular deep reinforcement learning. In our work, we based ourselves on a new IMDB dataset which was generated from the IMDBJOB dataset, this choice is justified by the size in number of queries and the diversification of the joins that make it up. The taxonomy of RL is very rich in term of algorithm, and which is subdivided into two families to know Model-Free and Model-based, we tried in first place to reimplement the PPO algorithm of the first class by modifying some parameters such as the architecture of model used by increasing the number 128 and the number of the layers as well as the function of activation, in second place, we tried to implement the Universal Value Function Approximators (UVFA) an extension of DQN with Hindsight Experience Replay (HER) The result of the experimentation showed a slight performance for the PPO which can be argued by the influence of the two modified parameters, for the UVFA algorithm, the results are considered very important and the fast convergences of loss parameters indicates the performance of this algorithm in the field of query optimization. In conclusion, we are aiming for the following future work :

- using other datasets to confirm these results,
- Consider integrating other types of queries such as OLAP queries.
- Implement an algorithm of the model-based methodes like the Imagination-Augmented Agents (I2A) algorithm of the second category.

Bibliography

[1] Deep neural network architecture. https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964. Accessed: 2024-04-05.

[2] Taxonomy of rl algorithms. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below. Accessed: 2024-03-26.

[3] P. Bizarro, Nicolas Bruno, and David J. DeWitt. Progressive parametric query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 21:582–594, 2009.

[4] Nicolas Bruno and Surajit Chaudhuri. Exploiting statistics on query expressions for optimization. In *ACM SIGMOD Conference*, 2002.

[5] dr. Matthias Hartwig. Self-driving and cooperative cars, 2020. The Institute for Climate Protection, Energy and Mobility (IKEM).

[6] Leo Giakoumakis and César A. Galindo-Legaria. Testing sql server's query optimizer: Challenges, techniques and experiences. *IEEE Data Eng. Bull.*, 31:36–43, 2008.

[7] Theodoros Goulas. Query optimization with deep learning architectures. 2022.

[8] Runsheng Benson Guo and Khuzaima S. Daudjee. Research challenges in deep reinforcement learning-based join query optimization. *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 2020.

[9] Oleg Ivanov and Sergey Bartunov. Adaptive cardinality estimation. *ArXiv*, abs/1711.08330, 2017.

[10] Philipp Moritz Michael I. Jordan Pieter Abbeel John Schulman, Sergey Levine. "trust region policy optimization".

[11] Littman M. L. Moore A. W. Kaelbling, L. P. Reinforcement learning: A survey. journal of artificial intelligence research, 4, 237-285., 1996.

[12] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *ArXiv*, abs/1808.03196, 2018.

[13] Hunt J. J. Pritzel A. Heess N. Erez T. Tassa Y. ... Lillicrap, T. P. and D Wierstra. Continuous control with deep reinforcement learning. arxiv preprint arxiv:1509.02971, (2016).

[14] Wu Y. Tamar A. Harb J. Abbeel O. P. Lowe, R. and I Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. in advances in neural information processing systems, (2017).

[15] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. *Proceedings of the 2021 International Conference on Management of Data*, 2020.

[16] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12:1705–1718, 2019.

[17] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 2018.

[18] Luka Mernik. An efficient representation for solving catalan number related problems. 2009.

[19] B. John Oommen and Luis Rueda. The efficiency of histogram-like techniques for database query optimization. *Comput. J.*, 45:494–510, 2002.

[20] M. L Puterman. Markov decision processes: Discrete stochastic dynamic programming (2nd ed.), 2014.

[21] Tom Schaul, Dan Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *International Conference on Machine Learning*, 2015.

[22] Wolski F. Dhariwal P. Radford A. Schulman, J. and O Klimov. Proximal policy optimization algorithms. arxiv preprint arxiv:1707.06347, (2017).

[23] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, page 23–34, New York, NY, USA, 1979. Association for Computing Machinery.

[24] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. Leo - db2's learning optimizer. In *Very Large Data Bases Conference*, 2001.

[25] Richard S. Sutton and Andrew G. Barto. Reinforcement learning : An introduction, 2014.

[26] Kaiwen Wang*, Junxiong Wang*, Yueying Li, Nathan Kallus, Immanuel Trummer, and Wen Sun. Joingym: An efficient query optimization environment for reinforcement learning. *arXiv preprint arXiv:2307.11704*, 2023.

[27] Jihad Zahir and Abderrahim El Qadi. A recommendation system for execution plans using machine learning. *Mathematical & Computational Applications*, 21:23, 2016.