UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

VITÓRIA LENTZ

Otimizador de Consultas SQL através de Aprendizado por Reforço

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof.ª Dra. Renata de Matos Galante

Porto Alegre
2024

AGRADECIMENTOS

Agradeço profundamente ao meu irmão gêmeo, Lucas, que infelizmente faleceu em 2022, vítima de câncer. Foi o Lucas quem abriu para mim as portas da área da tecnologia. Ele me mostrou o mundo onde hoje construo a minha carreira.

Sinto sua falta todos os dias. Obrigada por acreditar em mim e por ser minha inspiração. Essa conquista também é sua.

RESUMO

Este trabalho apresenta uma solução para otimização de consultas SQL em Sistemas de Gerenciamento de Banco de Dados (SGBDs), utilizando aprendizado por reforço profundo, com o objetivo de aumentar a eficiência em comparação aos métodos tradicionais de otimização. A crescente complexidade e o aumento do volume de dados, impulsionados pela expansão da Internet e da Internet das Coisas (IoT), tornam a busca por otimização de consultas uma tarefa fundamental. A abordagem proposta visa superar as limitações dos otimizadores em consultas com junções complexas, utilizando o otimizador do PostgreSQL como referência, que se baseia em heurísticas e modelos de custo estatísticos. Com a aplicação dos algoritmos de Deep Q-Network (DQN) e Double Deep Q-Network (DDQN), utilizando de bibliotecas de suporte ao aprendizado de máquina, desenvolveu-se um modelo capaz de aprender estratégias de otimização eficientes sem a necessidade de um modelo de custo pré-definido, baseando-se apenas em recompensas. Para o desenvolvimento e avaliação do modelo, foram utilizados o banco de dados IMDb e um conjunto de consultas pré-definido, aplicando o *benchmark* de suporte à decisão JOB. A análise dos resultados demonstrou que o otimizador do PostgreSQL tem bom desempenho em consultas simples, mas apresenta limitações em cenários mais complexos. O DQN destacou-se por sua eficiência em tempo de otimização, especialmente em consultas com muitas junções, equilibrando rapidez e qualidade nas soluções geradas. Por outro lado, o DDQN mostrou-se superior em termos de profundidade das árvores de consulta, criando estruturas mais rasas e otimizadas. Contudo, essa eficiência estrutural veio acompanhada de tempos mais elevados de processamento em consultas complexas, evidenciando a necessidade de ajustes no algoritmo. Os resultados reforçam o potencial do aprendizado por reforço profundo para superar as abordagens tradicionais de otimização em SGBDs. No entanto, desafios permanecem, como a redução do tempo de processamento e a validação do impacto desses algoritmos em cenários reais de produção.

Palavras-chave: Otimização de consultas. Aprendizado por reforço profundo. Cláusula JOIN.

SQL Query Optimizer through Reinforcement Learning

ABSTRACT

This work presents a solution for optimizing SQL queries in Database Management Systems (DBMSs), using deep reinforcement learning to increase efficiency compared to traditional optimization methods. The increasing complexity and volume of data, driven by the expansion of the Internet and the Internet of Things (IoT), make the search for query optimization a fundamental task. The proposed approach aims to overcome the limitations of optimizers in queries with complex joins, using the PostgreSQL optimizer as a reference, which is based on heuristics and statistical cost models. Applying the Deep Q-Network (DQN) and Double Deep Q-Network (DDQN) algorithms, using machine learning support libraries, a model capable of learning efficient optimization strategies without the need for a predefined cost model was developed, based only on rewards. The IMDb database and a predefined set of queries were used to develop and evaluate the model, applying the JOB decision support benchmark. Data analysis demonstrated that the PostgreSQL optimizer performs well on simple queries, but has limitations in more complex scenarios. DQN stood out for its efficiency in optimization time, especially in queries with many joins, balancing speed and quality in the generated solutions. On the other hand, DDQN proved to be superior in terms of query tree depth, creating shallower and more optimized structures. However, this structural efficiency was accompanied by higher processing times for complex queries, highlighting the need for adjustments to the algorithm. The results reinforce the potential of deep reinforcement learning (DRL) to outperform traditional optimization approaches in DBMSs. However, challenges still need to be addressed, such as reducing processing time and validating the impact of these algorithms in real production scenarios.

Keywords: Query optimization. Deep reinforcement learning. JOIN operation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BD	Banco de Dados
DDQL	Double Deep Q-Lerning
DDQN	Double Deep Q-Network
DQL	Deep Q-Learning
DQN	Deep Q-Network
DRL	*Deep Reinforcement Learning*
IMDb	*Internet Movie Database*
IoT	*Internet of Things*
JOB	*Join Order Benchmark*
ML	*Machine Learning*
SGBD	Sistema de Gerenciamento de Banco de Dados
SQL	*Structured Query Language*

SUMÁRIO

1 INTRODUÇÃO

Ao longo da história da Internet, da invenção da Internet das Coisas (IoT), e da utilização cada vez mais massiva de bancos de dados por empresas para armazenar grandes volumes de informações, a otimização de cada etapa do serviço de dados tornou-se essencial. Conforme destacado por Silberschatz, Korth e Sudarshan (2020), os Sistemas de Gerenciamento de Banco de Dados (SGBDs) tem como um de seus principais objetivos a otimização tanto da economia de memória, através do uso eficiente do espaço físico de disco e dos *buffers*, quanto na redução da latência de resposta em consultas.

Apesar de haver diversas tecnologias de armazenamento, os bancos de dados relacionais ainda dominam o cenário global atual. De acordo com o DB-Engines Ranking, atualizado mensalmente, os quatro primeiros colocados são Oracle, MySQL, Microsoft SQL Server e PostgreSQL, todos SGBDs relacionais, amplamente utilizados em diversas aplicações. Essa predominância deve-se à confiança que os bancos de dados relacionais oferecem no armazenamento de informações.

A demora na resposta de consultas em bancos relacionais com grandes volumes de dados é uma preocupação significativa em termos de eficiência. A importância da otimização de consultas tem crescido exponencialmente devido ao aumento do volume de dados e da complexidade das consultas realizadas em ambientes de *big data* e aplicações empresariais.

Para enfrentar esses desafios, otimizadores de consultas têm sido propostos para determinar a forma de execução mais eficiente para uma consulta SQL, buscando minimizar o tempo de resposta e o uso de recursos do sistema de banco de dados. Os otimizadores amplamente utilizados hoje (Oracle, PostgreSQL, MySQL), tanto comerciais quanto de código aberto, como o PostgreSQL (postgresql.org), utilizam heurísticas e modelos de custo baseados em estatísticas para determinar a estratégia de consulta. No entanto, essas abordagens apresentam limitações, onde as estimativas podem ser imprecisas.

Nos últimos anos, o avanço da inteligência artificial tem estimulado novas pesquisas voltadas para a otimização de consultas SQL, por ainda ser uma tecnologia muito utilizada no mercado. Técnicas de aprendizado por reforço profundo[1] têm sido exploradas para aprender estratégias de otimização sem a necessidade de um modelo de custo preexistente. Estudos recentes mostram que esses métodos podem superar otimizadores tradicionais, oferecendo melhorias significativas na eficiência dos recursos do banco de dados.

[1] O aprendizado por reforço profundo também é conhecido como DRL - *deep reinforcement learning*.

Entre as soluções propostas, destacam-se otimizadores como Neo (MARCUS et al., 2019), Balsa (YANG et al., 2022) e RL_Qoptimizer (RAMADAN et al., 2022), que aplicam aprendizado por reforço para aprimorar a eficiência das operações de consulta. Essas abordagens não apenas demonstram um desempenho superior em comparação com otimizadores comerciais, como também abrem novas possibilidades para a otimização de consultas em ambientes de dados complexos e dinâmicos.

Este trabalho tem como objetivo apresentar uma solução para otimização de consultas SQL utilizando aprendizado por reforço profundo, visando explorar a eficiência dos algoritmos de Deep Q-Network (MNIH et al., 2015) e Double Deep Q-Network (VAN HASSELT et al., 2016), quando comparados ao otimizador padrão do PostgreSQL. A maioria das pesquisas citadas não disponibiliza o código de treinamento da rede neural de forma aberta, dificultando a identificação dos diferenciais que levaram ao melhor desempenho e a comparação entre cada otimizador proposto. Assim, este estudo visa realizar uma análise de melhorias do algoritmo de treinamento por reforço.

Os experimentos realizados com o *Join Order Benchmark* (JOB) e o *Internet Movie Database* (IMDb) avaliaram o desempenho dos algoritmos DQN e DDQN em comparação ao otimizador padrão do PostgreSQL. Os resultados demonstram que o DQN se destaca em termos de tempo de otimização para consultas complexas, enquanto o DDQN gera árvores de consulta mais rasas e eficientes, mas com maior custo de processamento.

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 apresenta a revisão bibliográfica, abordando os fundamentos de aprendizado de máquina e banco de dados, além dos algoritmos e tecnologias utilizadas para o desenvolvimento dos modelos propostos. No Capítulo 3, são discutidos os trabalhos relacionados, destacando pesquisas recentes na área de otimização de consultas com aprendizado por reforço. A metodologia aplicada é detalhada no Capítulo 4, cobrindo o ambiente de aprendizado desenvolvido, os algoritmos implementados e as configurações experimentais. O Capítulo 5 analisa os resultados obtidos com o experimento, fazendo uma análise comparativa de dados entre os três otimizadores. Por fim, o Capítulo 6 apresenta as conclusões e propostas para trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Este Capítulo aborda os conceitos e fundamentos do aprendizado de máquina e suas aplicações em otimização de consultas em bancos de dados. A Seção 2.1, apresenta-se uma introdução aos algoritmos de Deep Q-Network (MNIH et al., 2015) e o Double Deep Q-Network (VAN HASSELT et al., 2016). Em seguida, na Seção 2.2, é discutido o uso de ferramentas para criação de ambientes de aprendizado por reforço, aplicado ao contexto de bancos de dados. A Seção 2.3 explora o uso de bibliotecas para implementar e treinar agentes de aprendizado por reforço. Por fim, a Seção 2.4 detalha os fundamentos de banco de dados e *benchmarks* utilizados, destacando o banco IMDb e o *Join Order Benchmark* (JOB) como fontes para avaliação e validação de modelos.

2.1 Aprendizado de Máquina

O aprendizado de máquina[2] visa criar modelos a partir de históricos de dados para a realização de previsões ou decisões baseadas em novos dados, ajustando-se conforme necessário. De acordo com Mitchell (1997), "um computador aprende a partir de exemplos ou dados quando melhora seu desempenho em uma tarefa, com o tempo e com a experiência, sem ser explicitamente programado para isso".

O aprendizado por reforço (SUTTON et al., 2018), é uma área do aprendizado de máquina. Nessa abordagem, um agente aprende a tomar decisões sequenciais em um ambiente para maximizar uma recompensa acumulada. O agente não recebe diretamente as respostas corretas, mas sim, uma recompensa ou penalidade com base nas suas ações, permitindo que ele aprenda através de tentativa e erro. Esse tipo de aprendizado é amplamente utilizado em áreas como robótica, jogos, sistemas de recomendação e controle de processos industriais, onde decisões sucessivas transformam o resultado.

2.1.1 Deep Q-Network

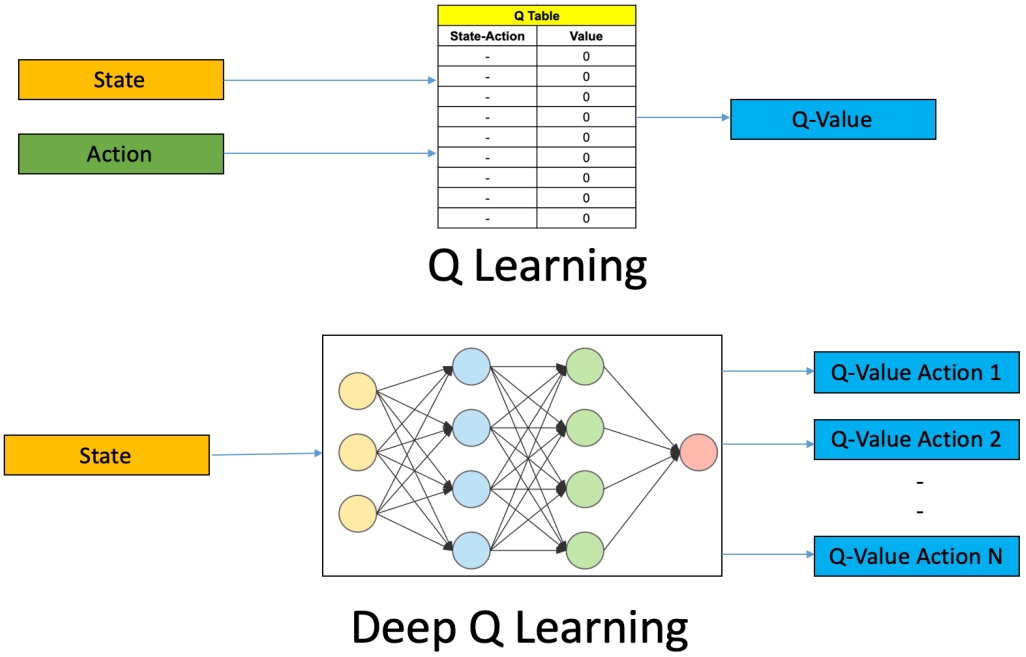
O Deep Q-Network (MNIH et al., 2015) é uma técnica de aprendizado por reforço que estende o algoritmo Q-Learning com o uso de redes neurais profundas. O Q-Learning é uma técnica clássica que cria uma tabela chamada Q-table, onde são armazenados os valores Q de

[2] O aprendizado de máquina também é conhecido como ML - *machine learning*.

cada combinação de estado e ação. Esse valor representa a recompensa esperada para uma ação em um estado específico. No entanto, essa abordagem torna-se inviável em problemas com grandes espaços de estado, como jogos complexos, onde armazenar todos os pares de estados e ações seria computacionalmente custoso.

Para contornar essa limitação, o Deep Q-Network usa uma rede neural profunda para aproximar a função, substituindo a Q-table. A rede neural recebe o estado como entrada e gera um valor Q para cada ação possível. Com isso, o DQN consegue lidar com problemas complexos, com muitas operações, e de grande escala. O algoritmo, assim, opera da forma mostrada na Figura 2.1.

Figura 2.1 – Comparação entre Q-Learning e Deep Q-Learning



Fonte: Karagiannakos (2018).

- Observação do Estado - o agente observa o estado atual do ambiente e o passa para a rede neural.
- Predição dos Valores - a rede neural gera valores para cada ação possível.
- Escolha da Ação - com base nos valores Q, o agente escolhe a ação com maior valor.
- Execução e Recompensa - a ação é executada, e o agente recebe uma recompensa do ambiente, além de observar o novo estado.
- Treinamento da Rede - a transição (estado, ação, recompensa, próximo estado) é armazenada em memória, que permite o treinamento da rede neural com amostras

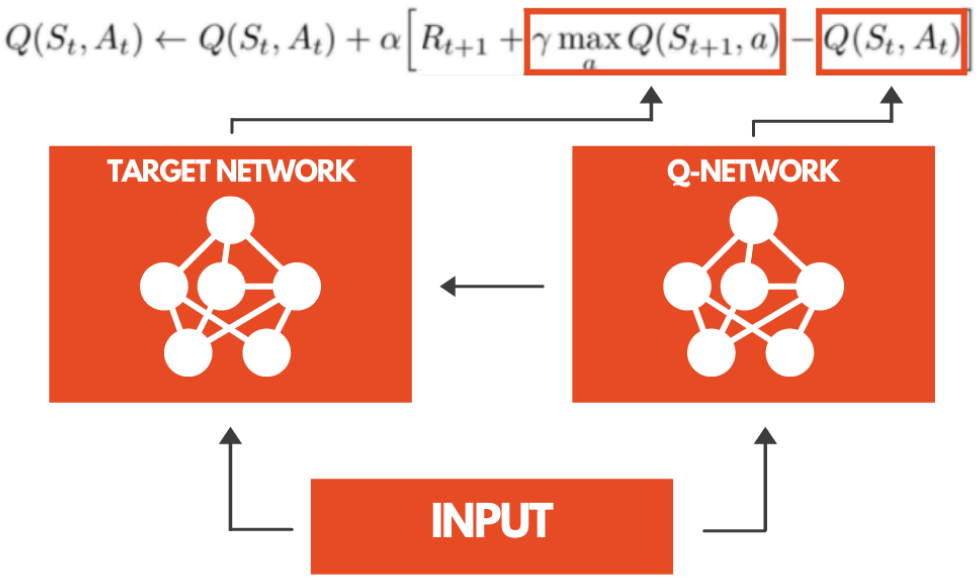
aleatórias de experiências passadas, ajudando a reduzir correlações entre as transições e melhorando o aprendizado.

2.1.2 Double Deep Q-Network

O Double Deep Q-Network (VAN HASSELT et al., 2016) é uma melhoria do Deep Q-Network (DQN) projetada para resolver problemas de viés de superestimação. Esse viés ocorre quando a rede neural tende a superestimar os valores Q de certas ações, o que pode levar o agente a fazer escolhas sub-ótimas.

O Double Deep Q-Network corrige essa questão utilizando duas redes neurais ao invés de uma: uma rede principal e uma rede-alvo. Essas redes desempenham papéis complementares, como visto na Figura 2.2.

Figura 2.2 – Fluxograma do Algoritmo de Double Deep Q-Network



Fonte: Zivkovic (2021).

- Escolha da Ação - a rede principal é usada para selecionar a ação com o maior valor no próximo estado.
- Cálculo do Valor Q - a rede-alvo calcula o valor Q para essa ação selecionada, e esse valor é utilizado como estimativa da recompensa futura.

A diferença fundamental é que a rede-alvo não é atualizada a cada passo: ela é sincronizada com a rede principal em intervalos regulares. Portanto, essa separação entre a escolha e a avaliação das ações reduz o viés de superestimação, pois a rede principal e a rede-alvo não compartilham os mesmos pesos em cada iteração.

2.2 Ambiente de Aprendizado

A biblioteca Gymnasium (gymnasium.farama.org) é uma ferramenta amplamente utilizada no campo de aprendizado por reforço para criar e simular ambientes onde agentes podem ser treinados. Ela é uma continuação da OpenAI Gym, com melhorias de suporte e atualizações. Com Gymnasium, modela-se ambientes complexos em que um agente pode aprender a tomar decisões com base em recompensas e punições.

Em um contexto de banco de dados, como no PostgreSQL, Gymnasium pode ser aplicada para desenvolver modelos que otimizem consultas SQL. O desempenho das consultas em grandes bases de dados é um aspecto crítico para muitas aplicações, e a escolha correta do plano de execução para essas consultas pode fazer uma grande diferença no tempo de resposta. Utilizando Gymnasium, pode-se criar um otimizador personalizado que busca superar a eficiência do otimizador nativo, aprendendo com *feedbacks* de performance.

Para utilizar Gymnasium na otimização de consultas SQL, primeiro é necessário definir um ambiente que represente o banco de dados e as operações de consulta:

- Estados - representam a configuração atual do banco de dados, incluindo índices, tabelas e métricas específicas de cada consulta.
- Ações - podem ser diferentes abordagens de execução para as consultas, como escolher certos índices, métodos de acesso (varredura completa de tabela, índices, etc.) ou mesmo estratégias de *cache*.
- Recompensas - medidas de desempenho, como o tempo de execução, cardinalidade ou o uso de recursos computacionais para cada consulta. A recompensa deve ser baixa para consultas lentas e alta para consultas rápidas.

Com o tempo, o agente aprende quais configurações e planos de execução resultam em tempos de resposta menores, desenvolvendo uma política otimizada para a execução de consultas específicas.

2.3 Projeto com TensorFlow

O TensorFlow (tensorflow.org) é uma biblioteca de código aberto desenvolvida pela Google que oferece ferramentas para construir e treinar modelos de aprendizado de máquina, incluindo redes neurais profundas. É amplamente usada em aplicações de aprendizado por

reforço e tem compatibilidade com outras ferramentas de aprendizado de máquina, facilitando a integração em *pipelines* de aprendizado de máquina. Por isso, foi escolhida para o treinamento de um agente que interage com o ambiente utilizando Gymnasium.

A biblioteca RLLib (docs.ray.io) fornece uma interface para conectar modelos do TensorFlow. Com ela, registram-se os modelos personalizados para usá-los nos algoritmos de treinamento do RLLib. Ele possui suporte tanto para Deep Q-Network quanto Double Deep Q-Network. É importante destacar que a biblioteca também possui suporte para *Proximal Policy Optimization* (PPO), um outro algoritmo que não será estudado aqui. Porém, há situações em que o uso de Deep Q-Network (DQN) ou Double Deep Q-Network (DDQN) pode ser mais adequado. Já pensando em trabalhos futuros, portanto, seria interessante estudar o funcionamento do PPO neste cenário.

2.4 Banco de Dados

Segundo Elmasri e Navathe (2015), "um banco de dados é uma coleção de dados relacionados logicamente que representa informações sobre um domínio específico", ou seja, é um sistema organizado para armazenar, gerenciar e recuperar informações de forma eficiente e segura. Ele permite que os dados sejam acessados e manipulados por meio de sistemas de gerenciamento de bancos de dados (SGBDs), como o MySQL, o PostgreSQL e o Oracle.

O PostgreSQL é um sistema de gerenciamento de banco de dados que foi projetado para ser extensível, com suporte a tipos de dados definidos pelo usuário e integração com diferentes linguagens de programação (STONEBRAKER; KEMNITZ, 1991). Ele foi escolhido para ser utilizado neste trabalho por ser um software *open-source* com capacidade de lidar com cargas de trabalho complexas, suporte extensivo a tipos de dados e com suporte a linguagem SQL.

2.4.1 Consultas em Bancos de Dados

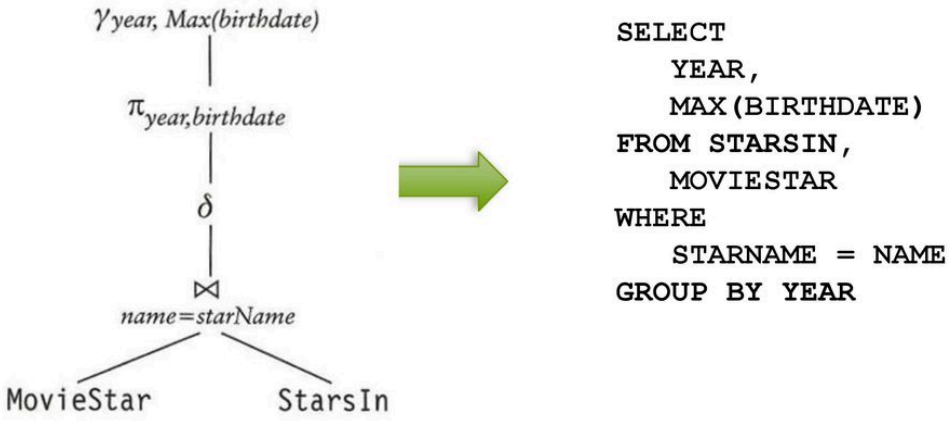
Uma consulta em um banco de dados é um pedido formal para acessar ou modificar dados em um banco, expressado em uma linguagem de consulta, como o SQL (*Structured Query Language*) (GARCIA-MOLINA et al., 2009). A instrução enviada para o SGBD pode buscar, inserir, atualizar ou deletar dados armazenados em uma tabela ou conjunto de tabelas.

Por exemplo, uma consulta de busca (`SELECT`) permite recuperar informações específicas, enquanto consultas de manipulação (`INSERT`,`UPDATE`,`DELETE`) alteram os dados armazenados.

Cardinalidade de uma consulta, por sua vez, é o número de instâncias de uma entidade que estão associadas a instâncias de outra entidade em um relacionamento (CORONEL; MORRIS, 2021). Em um contexto de banco de dados, ela descreve a quantidade de valores em uma tabela ou a quantidade de relações entre tabelas. Por exemplo, em um relacionamento 1:N (um para o valor N), uma entidade está associada a várias entidades em outra tabela.

Uma consulta SQL pode ser representada hierarquicamente por uma árvore de consulta, onde os nós representam operações ou instruções que serão realizadas no banco de dados, e as folhas da árvore representam as tabelas ou fontes de dados. Ela é utilizada pelos otimizadores de consultas em sistemas de gerenciamento de banco de dados (SGBDs) para analisar e otimizar sua execução. A Figura 2.3 a seguir traz um exemplo de árvore de consulta.

Figura 2.3 – Exemplo de Árvore de Consulta



Fonte: dos Santos (2018).

Neste exemplo, é possível ver como cada etapa é processada. Das tabelas `MovieStar` e `StarsIn`, é feita a junção dos dados onde `name` e `starName` são iguais. A partir disso, é selecionado somente os valores de `year` e `birthdate`. Por último, se coleta o maior valor de `birthdate` e o `year` correspondente.

2.4.1.1 Consultas com Joins

Uma consulta de junção (*join*) é uma operação que combina colunas entre tabelas com base em uma condição de correspondência (GARCIA-MOLINA et al., 2009). Existem diferentes tipos de *joins*, mas neste trabalho será usado apenas um em específico: o *join* implícito. Esse tipo de *join* utiliza o comando básico `SELECT` com uma cláusula `WHERE` para especificar a condição de correspondência entre as tabelas.

A cláusula `WHERE` é usada em consultas SQL para filtrar os registros retornados, limitando os resultados com base em uma ou mais condições. Ela permite selecionar apenas os dados que atendem aos critérios fornecidos.

A Figura 2.4 mostra um exemplo de consulta retirada do *Join Order Benchmark* (RAHN, 2019) que utiliza o *join* implícito. Nesse exemplo, as tabelas definidas após o comando `FROM` estão sendo unidas pelas cláusulas definidas pelo comando `WHERE`. O objetivo dessa consulta é retornar o nome do personagem e do filme que foi filmado com uma produção americana após o ano de 1990 e, para isso, utiliza informações que estão espalhadas em diferentes tabelas.

Figura 2.4 – Exemplo de Consulta com *Join* Implícito

```
1   SELECT MIN(chn.name) AS character,
2          MIN(t.title) AS movie_with_american_producer
3   FROM char_name AS chn,
4        cast_info AS ci,
5        company_name AS cn,
6        company_type AS ct,
7        movie_companies AS mc,
8        role_type AS rt,
9        title AS t
10  WHERE ci.note LIKE '%(producer)%'
11    AND cn.country_code = '[us]'
12    AND t.production_year > 1990
13    AND t.id = mc.movie_id
14    AND t.id = ci.movie_id
15    AND ci.movie_id = mc.movie_id
16    AND chn.id = ci.person_role_id
17    AND rt.id = ci.role_id
18    AND cn.id = mc.company_id
19    AND ct.id = mc.company_type_id;
```

Fonte: Rahn (2019).

2.4.2 Otimização em Bancos de Dados

Como apontado por Garcia-Molina, Ullman e Widom (2009), "a otimização de consultas é o processo de escolher a maneira mais eficiente de executar uma consulta SQL, considerando os recursos do sistema e as características do banco de dados". Isso inclui o uso eficiente de índices, normalização de tabelas, consultas bem estruturadas, uso de memória *cache* e ajuste de configurações no SGBD. O objetivo final é reduzir o tempo de resposta, uso de recursos e melhorar a experiência do usuário.

Um exemplo dado por Garcia-Molina, Ullman e Widom (2009) é o processo de dividir tabelas grandes e complexas em tabelas menores e mais simples para evitar a redundância de dados. Isso pode melhorar a consistência do banco e reduzir a duplicação, mas também pode resultar em consultas mais lentas devido a maior necessidade de junções (*joins*), que é conhecidamente uma operação custosa.

Outra solução usada em muitos sistemas de gerenciamento de banco de dados (SGBDs) é o uso de memória *cache* de consultas, que armazena os resultados de consultas frequentemente executadas. Isso pode melhorar o desempenho em casos onde as mesmas consultas são repetidamente executadas.

Quando a quantidade de dados é muito grande, o particionamento de tabelas pode ser uma técnica de otimização muito útil. Divide-se uma tabela grande em várias partes menores, as partições, com base em critérios como de identificação. Isso traz uma melhoria no desempenho de consultas ao reduzir o número de registros que precisam ser acessados.

O plano de execução de consultas, portanto, é muito importante para a otimização em bancos de dados. A análise da cardinalidade ajuda a identificar gargalos de desempenho e indicar onde as consultas podem ser otimizadas. Ajustar as consultas com base no plano de execução pode ajudar a melhorar o desempenho.

2.5 Dados e Consultas de Desenvolvimento

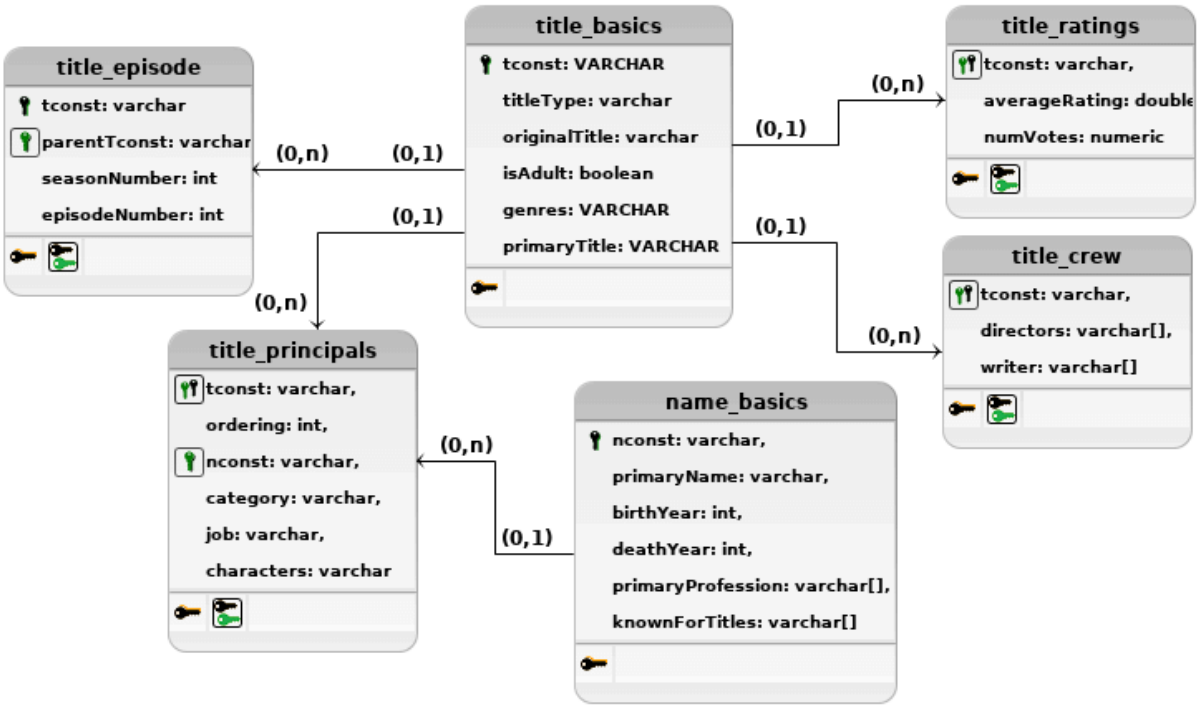
Para garantir um ambiente eficaz de treinamento e teste, é essencial dispor de uma ampla variedade de dados, permitindo que os algoritmos de aprendizado por reforço treinem, aprendam e validem suas descobertas. Assim, foram selecionados o banco de dados IMDb e o *benchmark* de consultas JOB, que oferecem os recursos necessários para atender a esses objetivos.

2.5.1 Internet Movie Database

Para o desenvolvimento, foi utilizado o banco de dados IMDb (*Internet Movie Database*) (developer.imdb.com), pois se trata de um dos maiores e mais populares bancos de dados. Amplamente utilizado na área de otimização de consultas, foi o banco de dados utilizado por todos os trabalhos de referência citados no Capítulo 3 deste trabalho.

A Figura 2.5 abaixo mostra como o banco está estruturado em tabelas relacionadas.

Figura 2.5 – Modelo Lógico de Dados do Banco IMDb



Fonte: Schreiner et al. (2019).

Esse banco possui informações sobre filmes, séries, produções televisivas, e outros conteúdos audiovisuais. Ele fornece detalhes como o elenco, a equipe de produção, as sinopses, as classificações e resenhas de usuários. O banco de dados do IMDb pode ser baixado e consultado por desenvolvedores, o que permite que esses dados sejam explorados e analisados de diversas maneiras, incluindo consultas complexas que integram várias tabelas.

2.5.2 Join Order Benchmark

O *Join Order Benchmark* (JOB) (RAHN, 2019) é um conjunto de testes projetado para avaliar a eficiência e desempenho de sistemas de gerenciamento de banco de dados na execução de consultas complexas, especificamente aquelas que envolvem apenas operações

join implícitas. Disponível em código aberto, ele tem como base de consulta o banco de dados do IMDb, destacado no Capítulo 2.5.1.

O JOB contém 113 consultas com diferentes níveis de complexidade e tamanhos de tabelas, criando cenários de variação que impõem diferentes tipos de cargas e padrões de consulta. Esse *benchmark* é amplamente utilizado para testar, otimizar e comparar diferentes mecanismos de execução de consultas em bancos de dados. Por isso, também foi utilizado em grande parte dos trabalhos citados como referência na área de otimização de consultas.

3 TRABALHOS RELACIONADOS

A seguir, são apresentados alguns dos principais estudos relacionados à otimização de consultas na atualidade. Cada otimizador proposto possui um diferencial em seu estudo, tanto na forma de treinamento, como no algoritmo de *learning* implementado, quanto na forma de recompensa e aprendizado escolhida pelos autores, como a cardinalidade ou o tempo de execução da consulta.

No entanto, todos os estudos abaixo compartilham o mesmo objetivo: criar um otimizador de consultas que supere os otimizadores atuais utilizando o aprendizado de máquina. Alguns conseguiram ultrapassar a eficiência de otimizadores comerciais, como os da Microsoft e Oracle, enquanto outros se equipararam a eles e superaram o otimizador *open-source* do PostgreSQL.

A principal dificuldade em compará-los, no entanto, é a falta de acesso aos algoritmos de aprendizado por reforço da maioria dos trabalhos, uma vez que os desenvolvedores geralmente não disponibilizam o otimizador ao público. Dessa forma, não é possível fazer uma comparação profunda entre os projetos, mas apenas entre os projetos e os otimizadores atualmente utilizados no mercado, como apresentado em cada artigo a seguir.

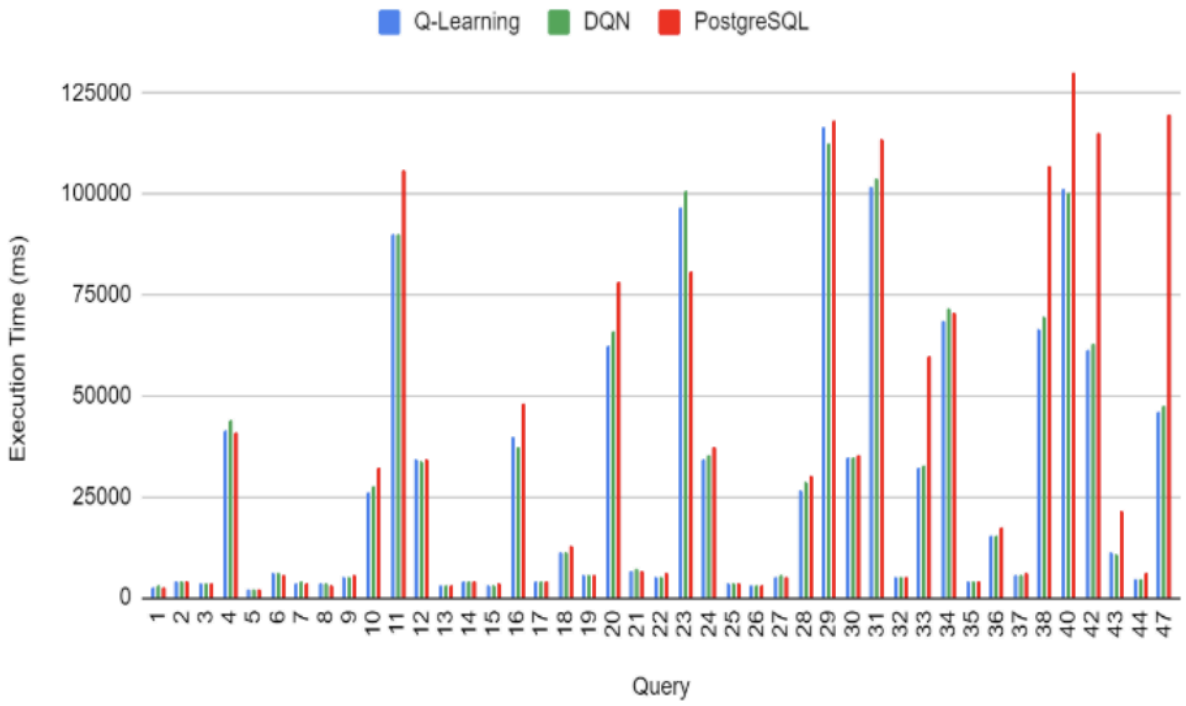
3.1 RL_QOptimizer: A Reinforcement Learning Based Query Optimizer

Neste estudo, conduzido na Universidade de Cairo, os autores Ramadan et al. (2022) comparam duas versões de um otimizador de consultas com o otimizador padrão do PostgreSQL, amplamente utilizado em bancos de dados. O otimizador proposto apresenta uma limitação, restringindo-se a buscas utilizando exclusivamente a função `JOIN`, visto que a junção de tabelas continua a ser um processo extremamente custoso e ainda pouco eficiente na atualidade.

A primeira versão do otimizador emprega o método de Q-Learning para aprender a forma mais eficiente de realizar uma consulta, utilizando um sistema de recompensas e transições de estado. A segunda versão adota o Deep Q-Network, que se baseia no aprendizado profundo (*deep learning)* para atribuir recompensas a cada estado. Dessa maneira, o otimizador é capaz de calcular a melhor busca não apenas para consultas conhecidas, através de seu treinamento, mas também para consultas novas que ainda não foram aprendidas.

O estudo revelou que ambos os otimizadores, Q-Learning e Deep Q-Network, apresentam eficiências bastante semelhantes quando testados com o mesmo conjunto de consultas, enquanto o otimizador do PostgreSQL demonstrou ser consideravelmente menos eficiente. A Figura 3.1 ilustra a comparação entre os otimizadores utilizando o banco de dados IMDb.

Figura 3.1 – Comparação entre Diferentes Otimizadores



Fonte: Ramadan et al. (2022).

3.2 Learning State Representations for Query Optimization with Deep Reinforcement Learning

Os autores Ortiz et al. (2018) apresentam um modelo de otimizador de consultas que utiliza *deep reinforcement learning*. O algoritmo atua dividindo a consulta em subconsultas e aprendendo as representações de estado de forma incremental através de redes neurais. As consultas utilizadas contêm cláusulas de seleção e conjunção, tanto no treinamento quanto nas comparações finais.

Um importante diferencial a ser destacado é que o treinamento foi realizado visando melhorar a cardinalidade da consulta, e não se baseou no tempo de execução da consulta.

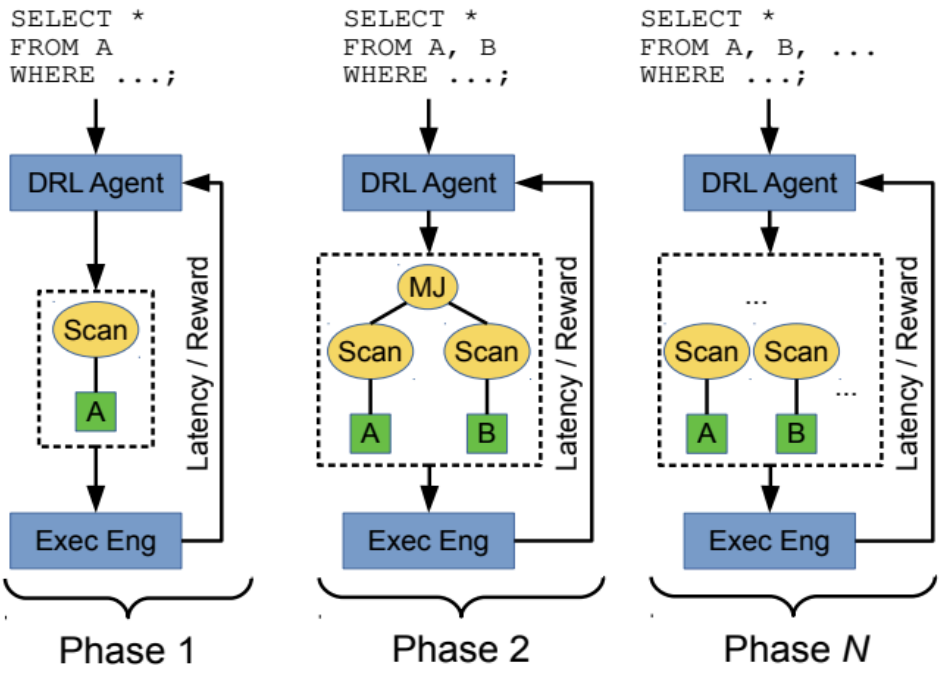
Quando comparado a um otimizador comercial, cujo nome é mantido anônimo, os resultados demonstraram que o modelo proposto apresenta desempenho semelhante. No entanto, quando o número de subconsultas em uma consulta aumenta, o otimizador desenvolvido acaba se mostrando mais lento que o otimizador comercial.

3.3 Towards a Hands-Free Query Optimizer through Deep Learning

Neste estudo de caso, os autores Marcus e Papaemmanouil (2018) analisam otimizadores de consulta já implementados e propõem três novos modelos utilizando *deep reinforcement learning* que não necessitam de *inputs* do programador para serem treinados. Embora o estudo não tenha desenvolvido os otimizadores, impossibilitando a comparação direta entre a eficiência de cada um e a de otimizadores atuais, o artigo destaca pontos relevantes. Entre eles, a influência da escolha da recompensa no treinamento do modelo sobre sua eficiência. A recompensa pode ser baseada na melhor cardinalidade, no menor tempo de acesso à memória (latência) ou na melhor utilização do buffer, cada um desses critérios apresentando suas respectivas vantagens e desvantagens.

Todos os modelos propostos pelo artigo utilizam um otimizador já existente na primeira fase para aprender a melhor forma de transformar as consultas.

Figura 3.2 – Projeto de Otimizador com Pipeline Incremental



Fonte: Marcus e Papaemmanouil (2018).

O primeiro modelo utiliza o *pipeline-based incremental learning*, visto na Figura 3.2, que separa as subconsultas e as processa em um pipeline predefinido de operações. O segundo modelo adota o *incremental relations*, aumentando progressivamente uma relação nas consultas utilizadas em cada fase de treinamento (começando com duas relações, depois três, e assim por diante). O terceiro modelo é uma combinação das duas abordagens anteriores.

Esta abordagem demonstra como diferentes estratégias de aprendizado podem ser aplicadas para otimizar consultas de forma eficiente, sem a necessidade de intervenção direta do programador.

3.4 LEO – DB2's LEarning Optimizer

LEO (STILLGER et al., 2001) é um otimizador de consultas capaz de aprender durante o processo de busca e ajustar seu resultado com base em uma série de informações fornecidas durante a execução. Essa informação é dada pelo esqueleto de cardinalidade da consulta.

O estudo também aborda tópicos avançados como a re-otimização de consultas e a adaptação de outros parâmetros do sistema de gerenciamento de banco de dados (DBMS) para um ambiente dinâmico.

Os autores destacam que o LEO proporciona uma melhoria significativa na qualidade da otimização de consultas e na redução da necessidade de *tuning* de consultas problemáticas, contribuindo para a redução do custo de propriedade.

3.5 Neo: A Learned Query Optimizer

O artigo propõe um novo otimizador de consultas baseado em aprendizado de máquina, chamado Neo (MARCUS et al., 2019). Este sistema utiliza uma abordagem de aprendizado *end-to-end* para otimização de consultas, incluindo a ordenação de junções, seleção de índices e operadores.

Neo substitui todos os componentes de um otimizador de consultas tradicional por modelos de aprendizado de máquina. Isso inclui a representação de consultas através de *features*, um modelo de custo baseado em redes neurais profundas (DNN), e uma estratégia de busca guiada por DNN. Após o treinamento com um conjunto de dados e consultas, Neo é

capaz de generalizar seu modelo para novas consultas e atingir um desempenho comparável aos otimizadores de consultas comerciais.

Na Figura 3.3, o autor compara o Neo treinado com diferentes otimizadores padrão (PostgreSQL, SQLite, SQL Server e Oracle) e diferentes *benchmarks* (JOB, TCP-H e um banco de dados corporativo privado). É possível ver como ele se sobressai em relação a latência padrão.

Figura 3.3 – Análise do Otimizador Neo Aplicado

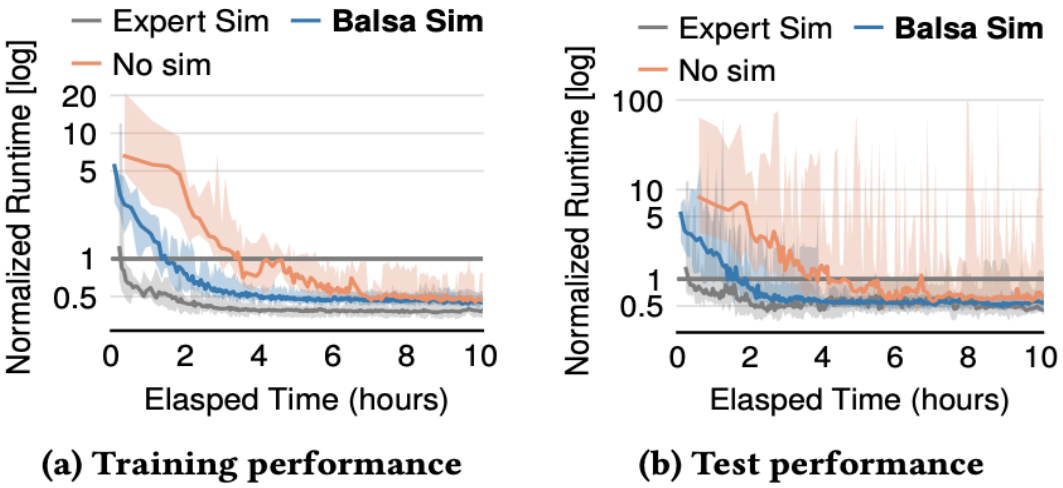
Fonte: Marcus et al. (2019).

3.6 Balsa: Learning a Query Optimizer Without Expert Demonstrations

O artigo apresenta o Balsa (YANG et al., 2022), um otimizador de consultas construído utilizando *deep reinforcement learning*. O Balsa aprende a otimizar consultas sem a necessidade de um otimizador preexistente, ao contrário da maioria dos otimizadores propostos em outros estudos. Essa abordagem também elimina a necessidade de um modelo de custo especializado, diferentemente dos métodos anteriores que dependem disso.

O Balsa é capaz de superar o otimizador de consultas do PostgreSQL e outros otimizadores comerciais após algumas horas de treinamento, como é visto na Figura 3.4. Ele demonstra que é possível otimizar consultas de maneira eficiente e automática sem a necessidade de otimizadores especializados pré-existentes. Isso abre novas possibilidades para a otimização automática em pesquisas futuras, onde não há otimizadores projetados por especialistas.

Figura 3.4 – Comparação entre Otimizadores Existentes e o Balsa



Fonte: Yang et al. (2022).

3.7 Comparação de Otimizadores

Cada otimizador apresentado anteriormente busca superar os métodos tradicionais, destacando-se pela eficiência e escalabilidade, especialmente em operações de alto custo computacional, como em junções. A Tabela 3.1 a seguir sintetiza as principais características de cada um e compara, ao final, com o otimizador desenvolvido neste trabalho.

Tabela 3.1 - Comparação de Otimizadores de Consultas

Otimizador	Metodologia	Precisa de um otimizador padrão para ser treinado?	Limitado a otimizações com cláusula de *join*?
RL_QOptimizer	Q-Learning e Deep Q-Network	Sim	Sim
Ortiz et al. (2018)	Representação incremental de estado; foca na cardinalidade das consultas.	Sim	Sim
Hands-Free Optimizer	Pipeline incremental e incremento de relações de consultas.	Não	Não
LeoDB2	Aprendizado adaptativo	Sim	Não
Neo	Deep Neural Network	Sim	Não
Balsa	Deep Reinforcement Learning	Não	Não
a autora (2024)	Deep Q-Network e Double Deep Q-Network	Sim	Sim

Fonte: a autora (2024).

Vale ressaltar que as informações aqui descritas refletem o que foi disponibilizado pelos autores nos artigos. Além disso, o único otimizador que possui código aberto (*open-source)* é o Balsa, o que impossibilita uma análise mais detalhada e comparativa das implementações.

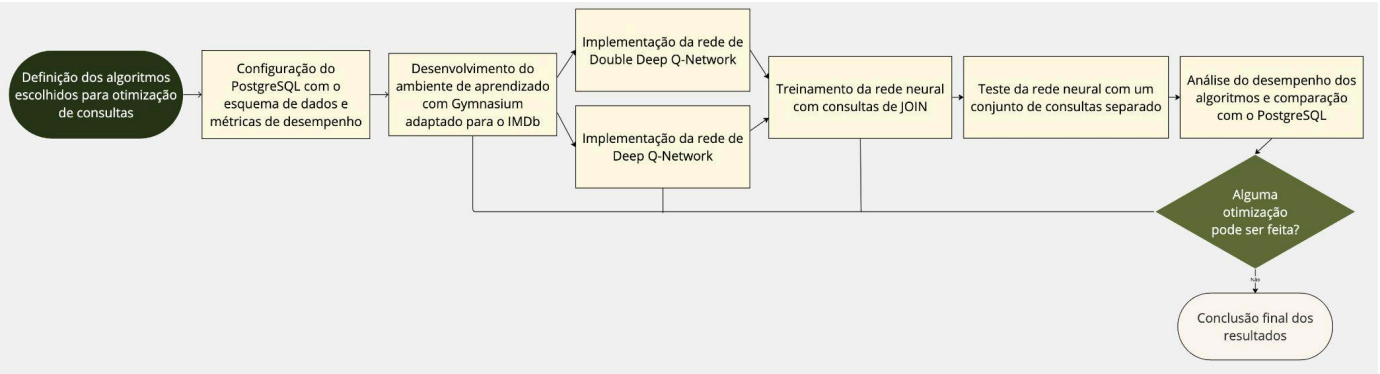
Com isso, pode-se observar que a principal diferença entre o trabalho presente e o Balsa é a utilização de algoritmos diferentes de treinamento da rede neural. O Balsa utiliza `pytorch`, mas não faz uso de Deep Q-Network e Double Deep Q-Network. O pytorch é uma outra biblioteca de redes neurais que não será utilizada neste trabalho.

4 METODOLOGIA

O objetivo final deste trabalho é implementar um otimizador de consultas utilizando *deep reinforcement learning*. Muitos dos estudos citados na revisão bibliográfica compartilham esse mesmo objetivo, buscando superar o desempenho dos otimizadores existentes. Neste trabalho, no entanto, são explorados dois algoritmos diferentes de aprendizado por reforço, o Deep Q-Network e o Double Deep Q-Network.

A Figura 4.1 apresenta o fluxograma que detalha as etapas de desenvolvimento seguidas neste trabalho.

Figura 4.1 – Fluxograma de Desenvolvimento do Otimizador



Fonte: a autora (2024).

De acordo com esse fluxo, este trabalho foi projetado para explorar o desempenho de modelos de aprendizado baseados em Deep Q-Network e Double Deep Q-Network. Inicialmente, foi necessário configurar o banco de dados, que utiliza as tabelas do *Internet Movie Database* (IMDb). O banco foi construído localmente usando o PostgreSQL, o que simplificou conexões e aumentou a eficiência do processo.

Em seguida, foi desenvolvido o ambiente de treinamento e teste, além dos algoritmos de aprendizado, utilizando bibliotecas em Python que facilitam a implementação, como TensorFlow, RLLib e Gymnasium. Após a configuração do ambiente, selecionaram-se consultas SQL do *Join Order Benchmark* (JOB), um *benchmark* que reúne consultas com a cláusula `JOIN`, conhecida por ser de alto custo computacional em sistemas de bancos de dados relacionais. O modelo foi, portanto, treinado e focado exclusivamente em otimizar consultas envolvendo *joins*, dada sua relevância para o desempenho de consultas SQL.

Por fim, foi realizada uma análise crítica dos resultados finais, comparando o desempenho da rede ao otimizador padrão do PostgreSQL. Essa análise buscou identificar

melhorias potenciais e validar a eficácia dos modelos de aprendizado em relação ao método tradicional através de visualizações gráficas feitas com o auxílio da biblioteca `matplotlib`.

4.1 Configuração do Banco de Dados

O *Internet Movie Database* (IMDb), utilizado para o treinamento e teste do modelo, é um banco de dados gratuito e aberto, disponível no site oficial (developer.imdb.com). Esse conjunto de dados também é amplamente utilizado em trabalhos anteriores, como os de Ramadan et al. (2022) e Yang et al. (2022).

A avaliação da capacidade do otimizador para tomar decisões é feita através do *benchmark* de suporte à decisão: *Join Order Benchmark* (JOB). O otimizador desenvolvido, portanto, se limita a resolver problemas com consultas usando o *join* implícito.

Para preparar o ambiente, foi instalado o PostgreSQL 12 a partir da ferramenta de gerenciamento de pacotes `apt` do próprio Linux. Assim, para criar e iniciar um *cluster* do PostgreSQL, basta usar o comando `pg_ctl` que aponta para o diretório o qual é utilizado pelo servidor do banco de dados. Em seguida, utilizando o comando `psql`, é feita a inicialização e a cópia dos dados baixados do IMDb para dentro do servidor local, que foi nomeado `imdb_`.

Figura 4.2 – Tabelas Contidas no Banco de Dados

```
  _imdb=# SELECT * FROM pg_catalog.pg_tables WHERE schemaname='public';
 schemaname |    tablename    | tableowner | tablespace | hasindexes | hasrules | hastriggers | rowsecurity
------------+-----------------+------------+------------+------------+----------+-------------+-------------
 public     | cast_info       | vlentz     |            | t          | f        | t           | f
 public     | aka_title       | vlentz     |            | t          | f        | f           | f
 public     | char_name       | vlentz     |            | t          | f        | t           | f
 public     | complete_cast   | vlentz     |            | t          | f        | t           | f
 public     | company_name    | vlentz     |            | t          | f        | f           | f
 public     | company_type    | vlentz     |            | t          | f        | f           | f
 public     | link_type       | vlentz     |            | t          | f        | t           | f
 public     | info_type       | vlentz     |            | t          | f        | t           | f
 public     | kind_type       | vlentz     |            | t          | f        | t           | f
 public     | aka_name        | vlentz     |            | t          | f        | t           | f
 public     | comp_cast_type  | vlentz     |            | t          | f        | t           | f
 public     | movie_companies | vlentz     |            | t          | f        | t           | f
 public     | keyword         | vlentz     |            | t          | f        | t           | f
 public     | movie_link      | vlentz     |            | t          | f        | t           | f
 public     | movie_info      | vlentz     |            | t          | f        | t           | f
 public     | name            | vlentz     |            | t          | f        | t           | f
 public     | movie_info_idx  | vlentz     |            | t          | f        | t           | f
 public     | person_info     | vlentz     |            | t          | f        | t           | f
 public     | title           | vlentz     |            | t          | f        | t           | f
 public     | role_type       | vlentz     |            | t          | f        | t           | f
 public     | movie_keyword   | vlentz     |            | t          | f        | t           | f
(21 rows)
```

Fonte: a autora (2024).

Após isso, o servidor local do PostgreSQL está configurado e pronto para ser utilizado no treinamento e testes do otimizador. A tabela na Figura 4.2 dá uma ideia de como o banco local com as informações do IMDb é disposto.

Para integrar o código desenvolvido em Python ao banco de dados, foi utilizada a biblioteca `psycopg2`, que fornece suporte para conexões e operações no PostgreSQL. Essa biblioteca simplifica a execução de consultas e manipulações no banco, facilitando o treinamento e avaliação do otimizador desenvolvido.

4.2 Implementação do Ambiente

A linguagem escolhida para o desenvolvimento foi Python, devido à sua ampla variedade de bibliotecas com suporte ao aprendizado de máquina. Essa característica facilita a implementação de algoritmos e contribui significativamente para a qualidade do trabalho realizado.

O framework Gymnasium foi utilizado para modelar ambientes que representam a execução de consultas SQL em bancos de dados, permitindo que um agente explore diferentes estratégias de otimização. Nesse contexto, o ambiente desenvolvido neste estudo é projetado para capturar informações sobre cardinalidades, que representam o número estimado de registros resultantes de uma consulta SQL. Essas informações são fundamentais para que o agente possa avaliar as decisões tomadas no processo de otimização.

A classe `LearningJob` foi criada com o objetivo de realizar consultas no banco de dados e calcular a cardinalidade das junções, retornando esses valores como insumos para o modelo de aprendizado por reforço. A partir do custo estimado da consulta, o ambiente de aprendizado é configurado para refletir as escolhas do agente durante a execução.

Primeiramente, a classe é iniciada definindo o esquema do banco de dados, as chaves primárias de cada tabela e estabelecendo a conexão com o banco de dados local, utilizando a biblioteca `psycopg2`. O espaço de treinamento é definido pela biblioteca Gymnasium, sendo estruturado de forma a representar uma caixa com diversos passos disponíveis, correspondendo às escolhas que o agente pode realizar ao longo do processo.

É importante destacar que a maior parte do ambiente foi inspirada no projeto Balsa (YANG et al., 2022). Por ser um código *open-source*, que fornece uma boa base para o desenvolvimento do ambiente, foram retiradas desse projeto informações de definição de

recompensa por cardinalidade, definição de ações e transformações de consultas em valores numéricos para a rede neural.

A Figura 4.3 contém a função de inicialização, onde é possível ver como o esquema do banco de dados da Figura 4.2 é formado.

Figura 4.3 – Inicialização do Ambiente

```
def __init__(self):
  self.schema = {"cast_info": ["id", "person_id", "movie_id", "person_role_id", "note", "nr_o
                 "aka_title": ["id", "movie_id", "title", "imdb_index", "kind_id", "productio
                 "char_name": ["id", "name", "imdb_index", "imdb_id", "name_pcode_nf", "surna
                 "complete_cast": ["id", "movie_id", "subject_id", "status_id"],
                 "company_name": ["id", "name", "country_code", "imdb_id", "name_pcode_nf", "
                 "company_type": ["id", "kind"],
                 "link_type": ["id", "link"],
                 "info_type": ["id", "info"],
                 "kind_type": ["id", "kind"],
                 "aka_name": ["id", "person_id", "name", "imdb_index", "name_pcode_cf", "name
                 "comp_cast_type": ["id", "kind"],
                 "movie_companies": ["id", "movie_id", "company_id", "company_type_id", "note
                 "keyword": ["id", "keyword", "phonetic_code"],
                 "movie_link": ["id", "movie_id", "linked_movie_id", "link_type_id"],
                 "movie_info": ["id", "movie_id", "info_type_id", "info", "note"],
                 "name": ["id", "name", "imdb_index", "imdb_id", "gender", "name_pcode_cf", "
                 "movie_info_idx": ["id", "movie_id", "info_type_id", "info", "note"],
                 "person_info": ["id", "person_id", "info_type_id", "info", "note"],
                 "title": ["id", "title", "imdb_index", "kind_id", "production_year", "imdb_i
                 "role_type": ["id", "role"],
                 "movie_keyword": ["id", "movie_id", "keyword_id"]}
  self.cursor = psycopg2.connect(host="localhost", database="_imdb", user="vlentz", password=
  columns = sum(len(x) for x in self.schema.values())
  tables = len(self.schema)
  self.space = spaces.Box(0, 1, shape=(tables*columns), dtype=np.float32)
  self.action = spaces.Discrete(columns*(tables-1))
```

Fonte: a autora (2024).

Para obter o plano de execução de uma consulta e, consequentemente, calcular sua cardinalidade, o comando `EXPLAIN` é utilizado. Esse comando retorna o custo estimado da consulta, que serve como métrica para a tomada de decisão do agente.

Figura 4.4 – Função de Cálculo de Cardinalidade

```
def estimate_cardinality(self,query):
    explain_query = f"EXPLAIN {query}"
    self.cursor.execute(explain_query)
    rows = self.cursor.fetchall()
    num_rows = rows[0][0].split("(cost=")[1].split(' ')[1].replace("rows=", "")
    cardinality = self.parse_cardinality(query,num_rows)
    return float(cardinality)
```

Fonte: a autora (2024).

No código implementado na Figura 4.4, o cursor é criado a partir da conexão estabelecida com o banco de dados, permitindo a execução da consulta SQL. A partir do plano de execução retornado pelo comando `EXPLAIN`, é extraída a informação sobre a cardinalidade.

Ao iniciar uma consulta, o ambiente é reinicializado, configurando um novo episódio para o agente. Nesse contexto, são definidos o estado inicial da consulta e o conjunto de ações e subconsultas possíveis. Para garantir a aleatoriedade do ambiente a cada nova consulta, foi utilizada a função de *seeding* do próprio Gymnasium.

Figura 4.5 – Transformação de Consultas em Ações do Ambiente

```
def get_actions(self,query,ids,masks,schema):
    actions = []
    joins = {}
    conditions = []
    q = query.split('FROM')[1]
    if 'WHERE' in q:
        relations = q.split('WHERE')[0].replace(" ", "").split(',')
    else:
        relations = q.replace(" ", "")
    for r in relations:
        r = r.replace("AS"," AS ")
        rs = r.split(" AS ")
        actions.append(Relation(r,ids,masks[rs[0]],schema[rs[0]]))
    if 'WHERE' in query:
        q = query.split('WHERE')[1]
        if 'AND' in q:
            conditions = q.split('AND')
    for c in conditions:
        if "=" in c:
            element = []
            clauses = c.split("=")
            for clause in clauses:
                element.append(clause.replace("\n", "").replace(" ", "").split("."))
```

Fonte: a autora (2024).

Adicionalmente, a função auxiliar da Figura 4.5 foi desenvolvida para interpretar e estruturar as consultas em um formato manipulável, adaptando-as para o ambiente de aprendizado. Essa função constrói o espaço de ações e define as condições de junção com base na consulta fornecida como entrada. Para isso, as condições de junção são extraídas a partir da cláusula `WHERE`, permitindo separar as junções e combiná-las em diferentes subconsultas. Esse processo possibilita testar diversos cenários e identificar combinações de subconsultas que se destacam em termos de eficiência.

A função de `step` da Figura 4.6 é responsável por executar ações. Ela recebe como entrada uma ação tomada pelo agente e retorna o novo estado resultante, a recompensa obtida

e, possivelmente, um sinal indicando o término do episódio caso a consulta tenha sido concluída.

Conforme a ação tomada, o número de subconsultas na consulta que está em execução pode diminuir, refletindo as escolhas do agente. Para determinar quais ações podem ser realizadas a cada passo, o agente explora as subconsultas disponíveis que ainda não foram processadas, que são consideradas válidas, permitindo a exploração de novos estados no ambiente.

Figura 4.6 – Função de Tomada de Ações

```
def step(self, action):
  costs = 0
  count_q = 0
  action = self.action_list[action]
  action_left = action[0]
  action_right = action[1]
  if (type(self.queries[action_left]) is not EmptyQuery) and (type(self.queries[action_right]) is not E
    new_action = []
    for q in self.queries:
      if q is self.queries[action_left]:
        new_action.append(Query(self.queries[action_left], self.queries[action_right]))
      if q not in (self.queries[action_left], self.queries[action_right]):
        new_action.append(q)
      else:
        new_action.append(EmptyQuery(list(np.zeros(len(self.obj_mask[0]), dtype=int))))
    self.queries = new_action
    for q in self.queries:
      if not((type(q) is Relation) or (type(q) is Query)):
        count_q += 1

  costs = 0
  if count_q is len(self.queries)-1:
    for q in self.queries:
      if (type(q) is Relation) or (type(q) is Query):
        try:
          costs = -1 * ((sqrt(calculate_cost(q, self.cursor) - self.cost['min'])) / (sqrt(self.cost['ma
        except:
            costs = 0
            pass
```

Fonte: a autora (2024).

Por fim, a função `close` da classe foi implementada para limpar os recursos utilizados e encerrar conexões ao final da execução. Essa prática garante que nenhum recurso permaneça ocupado desnecessariamente na máquina, promovendo eficiência e evitando possíveis problemas de alocação de memória ou desempenho.

Com essa configuração, o ambiente e o agente trabalham da seguinte forma:

- O estado atual do agente é a configuração atual da consulta durante o processo de otimização com as relações já unidas.

- As ações disponíveis são as possíveis junções que podem ser escolhidas entre as relações restantes.
- A recompensa é inversa ao custo estimado da árvore de consulta, atribuída sempre ao final do episódio, após a geração completa da árvore de consulta.

4.3 Implementação das Redes

Para construir e treinar os modelos de rede neural, utilizou-se a biblioteca TensorFlow, que permite criar arquiteturas para os algoritmos de Deep Q-Network (DQN) e Double Deep Q-Network (DDQN). Paralelamente, a biblioteca RLLib é usada para implementar redes totalmente conectadas por meio da função `FullyConnectedNetwork`, além de aplicar uma máscara de ações para evitar escolhas inválidas pelo agente.

As configurações de hiperparâmetros, como taxas de aprendizado, tamanhos de *batch* e número de episódios, diferem entre DQN e DDQN. A principal diferença é o parâmetro `double_q`, que é ativado somente para o treinamento em DDQN. Além disso, o DDQN utiliza o *replay buffer* como parte da arquitetura, permitindo que o agente armazene e utilize experiências passadas durante o treinamento. Isso não apenas melhora a eficiência dos dados, mas também reduz a correlação entre as amostras consecutivas, ajudando a estabilizar o aprendizado e a minimizar o viés nas atualizações de Q-values.

A Figura 4.7 mostra como foi feita a configuração de parâmetros. As configurações do algoritmo de DQN são exportadas da biblioteca RLLib, onde há um exemplo com configurações padrões, e a configuração de DDQN é adaptada a partir dela.

Figura 4.7 – Configurações de DQN e DDQN

```
dqn_config = DQNConfig()

ddqn_config = dqn_config.copy()
ddqn_config["schedule_max_timesteps"] = 200000
ddqn_config["exploration_final_eps"] = 0.00
ddqn_config["learning_starts"] = 150000
ddqn_config['double_q']=True
ddqn_config['prioritized_replay']=True
```

Fonte: a autora (2024).

Após o treinamento, os resultados são avaliados para identificar ajustes que melhorem o desempenho do otimizador e previnam o *overfitting*, onde a rede acaba decorando as

melhores respostas para o conjunto de treinamento e perdendo capacidade. A recompensa atribuída ao agente durante a execução das consultas é fundamental para o sucesso da rede.

Neste trabalho, optou-se por usar a cardinalidade como métrica porque esse parâmetro reflete melhor o tamanho da consulta e é independente de variáveis externas, como a configuração de hardware ou a presença de dados em *cache*, fatores que podem influenciar significativamente o tempo de execução. A cardinalidade, por sua vez, é intrínseca à consulta e fornece uma medida consistente.

4.4 Treinamento e Desempenho

O *Join Order Benchmark* (JOB), utilizado para treinamento e teste dos algoritmos neste trabalho, é composto por 113 consultas SQL disponíveis em código aberto (RAHN, 2019). Essas consultas apresentam diferentes níveis de complexidade e variações na formação de *joins*, sendo projetadas para avaliar a eficiência e a rapidez de sistemas de gerenciamento de bancos de dados.

No contexto deste estudo, as consultas foram organizadas da seguinte maneira: as primeiras 80 consultas foram destinadas ao treinamento do modelo, enquanto as 33 restantes foram reservadas exclusivamente para o teste. Essa separação garante que o agente seja avaliado em dados não vistos durante o treinamento, permitindo uma análise confiável de sua capacidade de generalização.

É importante que as consultas de treinamento e de teste estejam equilibradas em nível de complexidade, isto é, tenham consultas com poucas e muitas junções em ambos os conjuntos. Será visto no Capítulo 5, Experimentos e Resultados, na Figura 5.3, que as consultas estão bem distribuídas nesse sentido.

Após o treinamento, então, avaliações do modelo de aprendizado por reforço foram realizadas com base em *checkpoints* armazenados. A partir de um *checkpoint* específico, o sistema restaura o estado do agente treinado, incluindo a configuração do modelo e os hiperparâmetros associados. A Figura 4.8 a seguir exemplifica a saída de um *checkpoint*, destacando informações agregadas sobre recompensa, exploração, métricas de tempo e posições coletadas ao longo dos episódios de treinamento.

É importante destacar que os *checkpoints* de DQN e DDQN carregam as mesmas informações. O que muda de um para o outro, no entanto, é a forma da rede de treinamento.

Figura 4.8 – Saída de um Checkpoint de DQN

DQN > DQN_learning_job_mask-v0_0_2024-11-17_12-11-23hglqld41 > {} result.json > ...


```
{"episode_reward_max": -0.011812649489164943, "episode_reward_min": -1.7188920922579758, "episode_reward_m
{"episode_reward_max": -0.008123040354348762, "episode_reward_min": -2.1165183967650774, "episode_reward_m
{"episode_reward_max": -0.007144318939138238, "episode_reward_min": -1.465472043750692, "episode_reward_me
{"episode_reward_max": -0.009913407780756594, "episode_reward_min": -2.4281232470470377, "episode_reward_m
{"episode_reward_max": -0.010211044335744638, "episode_reward_min": -0.8282895011445874, "episode_reward_m
{"episode_reward_max": -0.009707159905085283, "episode_reward_min": -1.0838085295383557, "episode_reward_m
{"episode_reward_max": -0.01048135654320964, "episode_reward_min": -0.776521404928769, "episode_reward_mea
{"episode_reward_max": -0.012174497970012564, "episode_reward_min": -0.7422810913515997, "episode_reward_m
{"episode_reward_max": -0.007804321244717122, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
{"episode_reward_max": -0.012599576733959493, "episode_reward_min": -0.6557048111268361, "episode_reward_m
```


Fonte: a autora (2024).

A Figura 4.9 a seguir ilustra como o agente computa cada ação e calcula a recompensa total.

Figura 4.9 – Cálculo de Recompensa Total

```python
total_reward = 0.0
while not done and steps < (num_steps or steps + 1):
    obs = {_DUMMY_AGENT_ID: observe}
    action_dict = {}
    for agent_id, o in obs.items():
        if o is not None:
            policy_id = mapping_cache.setdefault(agent_id, policy_agent_mapping(agent_id))
            a_action = agent.compute_action(o,prev_action=prev_a[agent_id],prev_reward=prev_r[agent_id]
            action_dict[agent_id] = a_action
            prev_a[agent_id] = a_action
    action = action_dict
    next_o, reward, done = env.step(action)
    total_reward += reward
    steps += 1
```

Fonte: a autora (2024).

O teste de desempenho utilizou as 33 consultas separadas e distintas das usadas para o treinamento. Durante essas avaliações, o agente é executado por um número específico de etapas no ambiente, armazenando os resultados relacionados a etapas, recompensas e custos de cada ação tomada. Esses dados são posteriormente analisados para avaliar o desempenho global do modelo.

4.5 Melhorias de Código

Com o objetivo de aprimorar a eficiência do modelo de aprendizado por reforço, algumas análises foram realizadas a partir do código original. Identificou-se uma melhoria relevante: a eliminação de junções cruzadas durante a execução de ações no ambiente. Junções cruzadas ocorrem quando duas tabelas são combinadas sem uma condição explícita de junção, resultando em um produto cartesiano. Esse tipo de operação é geralmente ineficiente e indesejável em otimizações de consultas devido ao alto custo computacional. O código na Figura 4.10 foi inserido a fim de resolver esse problema. Ele é executado cada vez que o agente realiza uma ação no ambiente.

O processo de validação, portanto, consiste em extrair os nomes das tabelas associadas à ação atual e verificar se o identificador da tabela, que pode ser um alias ou o nome original, está presente no conjunto de condições de junções pré-definidas. Caso a condição de junção não exista, isso indica que a ação levará a um estado inválido por se tratar de uma junção cruzada.

Figura 4.10 – Eliminação de Junções de Tabelas Cruzadas

```
right_join = self.queries[self.action_list[i][1]].name
left_join = self.queries[self.action_list[i][0]].name
if " AS " in right_join:
    right_list = [right_join.split(" AS ")[1]]
else:
    right_list = right_join.split('_')
if " AS " in left_join:
    left_list = [left_join.split(" AS ")[1]]
else:
    left_list = left_join.split('_')
q = '_'.join(sorted(left_list + right_list))
if q in join_conditions:
  self.actions.append(i)
```

Fonte: a autora (2024).

Ao descartar pares de tabelas em consultas e subconsultas sem condições de junção definidas, o agente é protegido contra combinações ineficientes. Essa abordagem não só melhora o desempenho do modelo de aprendizado por reforço, mas também garante que apenas junções válidas e relevantes sejam consideradas durante o processo de otimização.

4.6 Teste de Confiabilidade

Embora a confiabilidade de um otimizador de consultas não seja o foco principal deste trabalho, ela é um aspecto extremamente relevante, especialmente quando se trata de um otimizador voltado para aplicações comerciais. Nesse contexto, a confiabilidade refere-se à corretude da saída em relação ao que foi solicitado na consulta.

Neste estudo, não foram realizados testes extensivos para avaliar a confiabilidade e corretude, mas essa análise é fortemente recomendada como uma etapa importante para pesquisas futuras.

Para verificar se a rede neural gera saídas corretas para as consultas SQL propostas, foram comparados os resultados obtidos pelos algoritmos Deep Q-Network, Double Deep Q-Network e o otimizador padrão do PostgreSQL no conjunto de consultas de teste. Nenhum problema foi identificado nas respostas durante os experimentos. Contudo, não se pode descartar a necessidade de realizar testes mais amplos e rigorosos para validar a confiabilidade das redes neurais de maneira mais abrangente.

5 EXPERIMENTOS E RESULTADOS

Neste capítulo, são apresentados os experimentos e as análises realizadas nos modelos de algoritmo de aprendizado por reforço propostos e seus comparativos de desempenho. A Seção 5.1 contém as informações de como as redes foram treinadas e como os dados resultantes foram manipulados para os testes. Na Seção 5.2, são apresentados os gráficos e as análises dos resultados obtidos a partir dos testes.

5.1 Configuração dos Experimentos

O desempenho das redes neurais no treinamento é influenciado pelas características do hardware e pela capacidade de processamento disponível. Embora todos os testes tenham sido realizados na mesma máquina, é importante destacar algumas especificações do ambiente utilizado, para maior clareza e transparência dos resultados.

O trabalho foi desenvolvido em um computador com *Windows Subsystem for Linux* (WSL), rodando o Ubuntu 20.04, uma distribuição do sistema operacional Linux. As configurações da máquina são descritas na tabela abaixo.

Tabela 5.1 - Informações de Hardware do Computador

Processador	AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx 2.30 GHz
RAM instalada	12,0 GB (utilizável: 9,94 GB)
Tipo de sistema	Sistema operacional de 64 bits, processador baseado em x64
Placa de vídeo	AMD Radeon (TM) RX Vega 10 graphics

Fonte: a autora (2024).

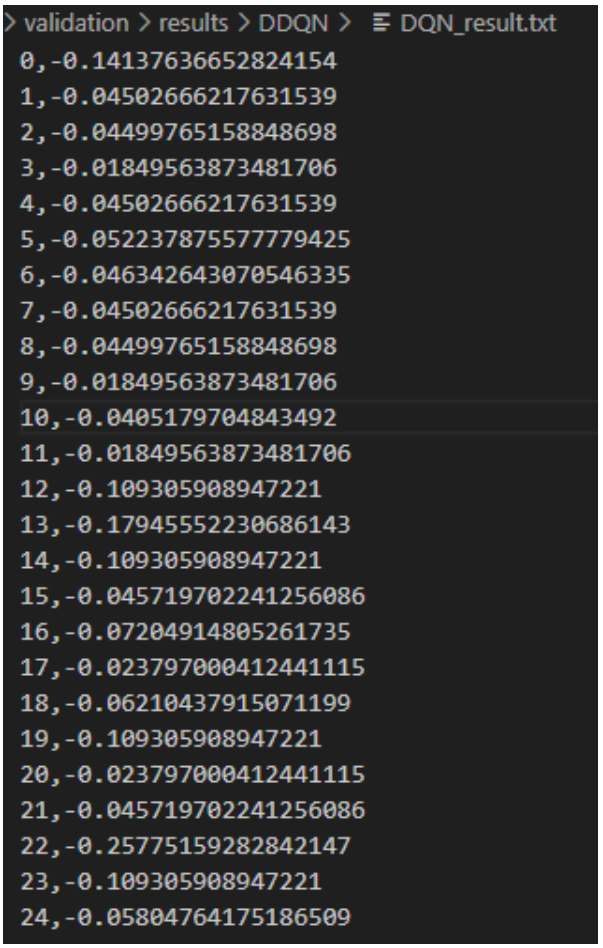
Os resultados dos experimentos foram comparados ao desempenho do otimizador padrão do PostgreSQL, com o suporte de visualizações gráficas criadas pela biblioteca `matplotlib`. Os scripts de análise desenvolvidos têm como objetivo processar e interpretar os dados gerados, permitindo avaliar o desempenho dos modelos treinados e as estratégias adotadas para a otimização de consultas.

Os resultados finais dos testes dos algoritmos são armazenados em arquivos de texto contendo informações do custo de cada consulta executada. Isso inclui tanto os resultados dos algoritmos Deep Q-Network (DQN) e Double Deep Q-Network (DDQN) quanto os do

método padrão utilizado pelo PostgreSQL. Uma função genérica para o cálculo de cardinalidade foi implementada, vista na Figura 4.4, utilizando o comando `EXPLAIN` do PostgreSQL, que fornece estimativas do custo e cardinalidade das consultas.

A Figura 5.1 a seguir apresenta um exemplo de um arquivo de resultados gerado, ilustrando os custos das consultas executadas.

Figura 5.1 – Arquivo Final de Recompensas



Fonte: a autora (2024).

Esses dados são utilizados em análises estatísticas que incluem a avaliação do custo médio por consulta, a comparação de desempenho entre os otimizadores e o acompanhamento da evolução das recompensas ao longo dos episódios de treinamento. Essas comparações são fundamentais para identificar padrões de desempenho e serão apresentadas nas seções a seguir.

Para assegurar comparações consistentes entre os diferentes experimentos, os custos são normalizados com o uso da fórmula ilustrada na Figura 5.2 , baseando-se no maior e menor valor de custo.

Figura 5.2 – Fórmula de Normalização de Custo

```
def normalize_df(df_list):
    return (df_list-df_list.min())/(df_list.max()-df_list.min())
```

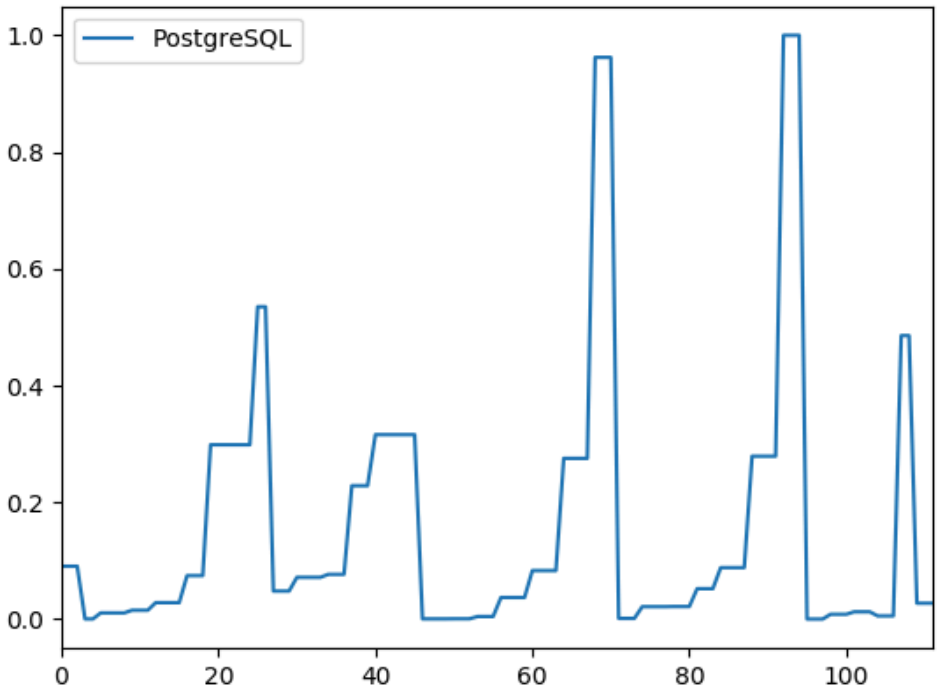
Fonte: a autora (2024).

5.2 Experimentos

Aqui são apresentados os resultados dos experimentos. A Seção 5.2.1 ilustra a complexidade das consultas de *benchmark* utilizadas. Já as Seções 5.2.2 e 5.2.3 apresentam os resultados de cada otimizador em relação a cardinalidade da consulta, que possui relação direta com a árvore de consulta gerada e o custo total durante o processo de otimização. Na Seção 5.2.4, a análise é feita utilizando o tempo de resposta da consulta. Essa é uma métrica que não foi usada no treinamento da rede, mas que deve ser considerada nos resultados a fim de comparar a eficiência de otimização de cada algoritmo proposto.

5.2.1 Número de Joins por Consulta

Figura 5.3 – Gráfico de Número de *Joins* por Consulta



Fonte: a autora (2024).

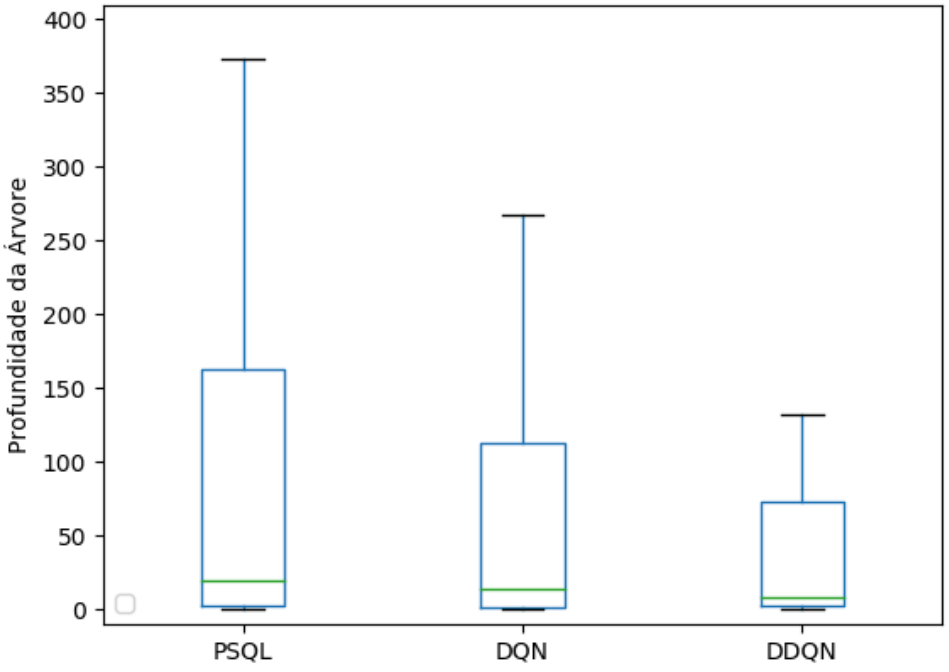
O gráfico de linha ilustrado na Figura 5.3 mostra o número de *joins* por consulta presentes no *Join Order Benchmark* (JOB) de forma normalizada, executados no otimizador padrão do PostgreSQL. Isso mostra como o *benchmark* contém 113 consultas com diferentes níveis de complexidade de *join*. As 33 últimas consultas (81 a 113) são usadas exclusivamente para os testes de desempenho.

Nesse gráfico, é possível ver como as consultas de treinamento e teste encontram-se bem distribuídas a nível de complexidade, ou seja, número de *joins* por consulta. As consultas de 81 a 113 possuem complexidades grandes e pequenas, assim como as consultas de 0 a 80. Isso garante que o treinamento das redes e os testes estejam equilibrados.

5.2.2 Profundidade da Árvore de Consulta

O gráfico da Figura 5.4 descreve a profundidade das árvores de consultas geradas por três otimizadores: o otimizador padrão do PostgreSQL e os otimizadores desenvolvidos neste trabalho, baseados em Deep Q-Network (DQN) e Double Deep Q-Network (DDQN). O *boxplot*, utilizado aqui, resume a distribuição dos dados e destaca seus principais estatísticos descritivos.

Figura 5.4 – Gráfico de Profundidade da Árvore de Consultas



Fonte: a autora (2024).

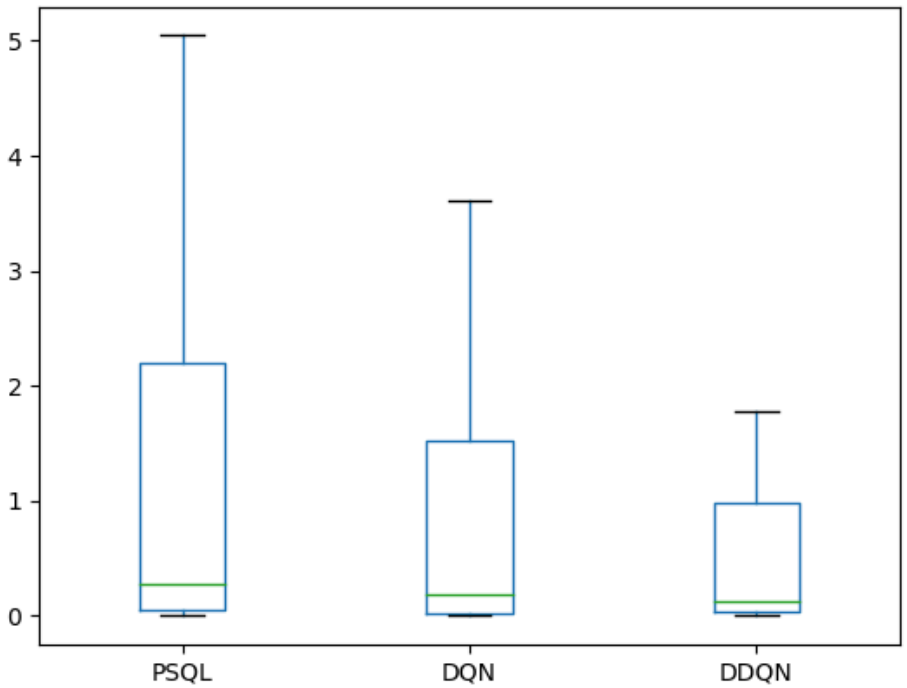
A linha central da caixa representa a mediana, ou seja, o valor central dos dados. Os limites inferior e superior da caixa correspondem ao 1º quartil (25%) e ao 3º quartil (75%), respectivamente. As linhas de extensão indicam a amplitude dos dados dentro de um intervalo aceitável, até 1,5 vezes o intervalo interquartil. Valores fora desse intervalo são considerados *outliers*.

Com base nessa análise, o otimizador padrão do PostgreSQL apresenta a maior variabilidade, com profundidades que chegam a valores significativamente altos. Sua mediana é visivelmente superior às dos demais métodos. O Deep Q-Network (DQN), por sua vez, exibe uma profundidade moderada, com menor variabilidade do que o PostgreSQL, mas ainda com maior amplitude em comparação ao Double Deep Q-Network (DDQN).

Em relação a média dos valores, todos os otimizadores se aproximam do mesmo valor, mas tem variabilidades bem diferentes. Conclui-se, portanto, que o Double Deep Q-Network (DDQN) é o mais eficiente em manter árvores com menor profundidade. Isso é vantajoso, pois árvores mais rasas tendem a melhorar o desempenho computacional, especialmente em consultas complexas.

5.2.3 Custo Total das Consultas

Figura 5.5 – Gráfico de Custo de Consultas



Fonte: a autora (2024).

O gráfico da Figura 5.5 apresenta a comparação do custo das consultas de teste para os três otimizadores avaliados: o otimizador padrão do PostgreSQL, o Deep Q-Network (DQN) e o Double Deep Q-Network (DDQN). Para todos os algoritmos, o custo de uma consulta é representado pela profundidade da árvore resultante, conforme mostrado no gráfico da Seção 5.2. Por esse motivo, os gráficos das Figuras 5.4 e 5.5 são proporcionais.

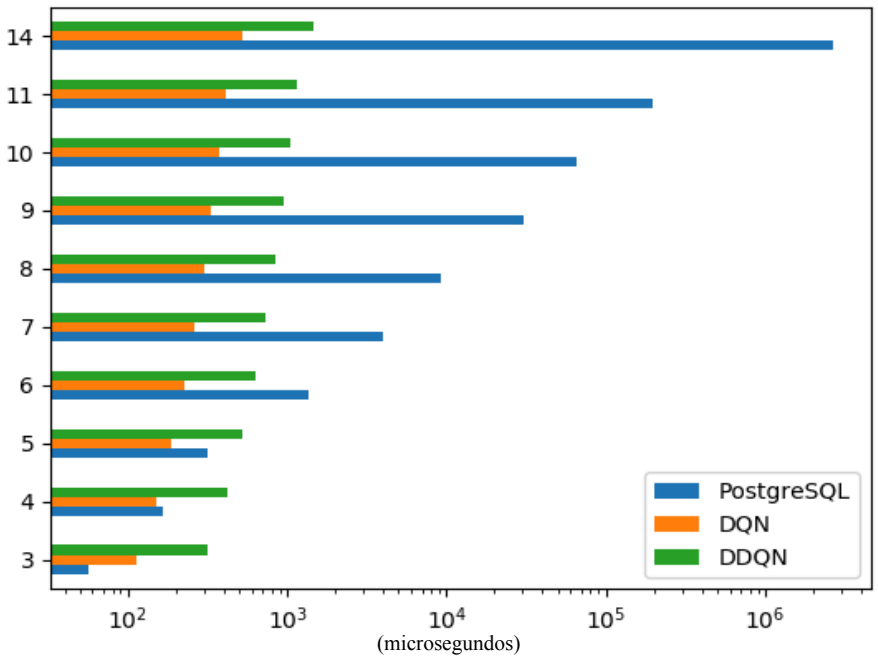
Com isso, é possível ver que o otimizador padrão do PostgreSQL representa a maior variabilidade de custo das consultas, ou seja, tem a maior amplitude no gráfico. Também tem valores extremos em relação aos outros, vistos pelas linhas de extensão inferior e superior.

A análise, portanto, revela que o otimizador padrão do PostgreSQL apresenta o pior valor de custo das consultas. Já o otimizador baseado em Deep Q-Network (DQN) apresenta uma variabilidade menor em relação ao PostgreSQL padrão, indicando um custo médio de consulta menor. Por fim, o otimizador baseado em Double Deep Q-Network (DDQN) destaca-se por apresentar o menor custo total e a menor variabilidade, indicando maior eficiência em comparação com os demais métodos.

Em conclusão, o DDQN é o método mais eficiente em termos de custo médio das consultas, que está relacionado com a profundidade da árvore de consulta.

5.2.4 Número de Joins por Tempo de Otimização

Figura 5.6 – Gráfico do Tempo de Consultas



Fonte: a autora (2024).

O gráfico de barras da Figura 5.6, mostra o tempo de otimização em milissegundos (ms) para diferentes números de *joins* por consulta. Os mesmos três otimizadores são comparados: PostgreSQL, Deep Q-Network (DQN) e Double Deep Q-Network (DDQN).

Foi realizado um agrupamento de consultas por número de *joins* presentes, com isso, tem-se a complexidade de cada grupo de consultas. Realizando a média do tempo de execução de cada consulta por agrupamento, é possível ter uma ideia do processamento de cada algoritmo baseado na complexidade.

O otimizador padrão do PostgreSQL apresenta tempos de otimização consistentemente mais altos em consultas com números maiores que 6 *joins*, alcançando tempos superiores a 10^6 ms. No entanto, para consultas mais simples, com até 5 *joins*, ele mostra desempenho semelhante, e até superior, aos algoritmos baseados em *deep reinforcement learning*, o que é justificável, dado que o otimizador do PostgreSQL é baseado em estatísticas bem calibradas.

O Double Deep Q-Network (DDQN), embora tenha tempos de otimização menores em relação ao PostgreSQL, ainda apresenta valores elevados quando comparado ao Deep Q-Network (DQN). Isso sugere que ele consome mais tempo para calcular as melhores estratégias de junção (*joins*), dado que, como observado em gráficos anteriores, como o 5.4, ele é o mais eficiente na construção de árvores de menor profundidade. O principal desafio desse algoritmo é, portanto, otimizar o tempo de processamento de consultas, sem comprometer sua eficácia em gerar árvores otimizadas.

O Deep Q-Network (DQN), por sua vez, se destaca como o mais eficiente em termos de tempo de otimização, especialmente em consultas com mais de 5 *joins*. Sua eficiência é evidente em cenários de alta complexidade, onde ele supera os outros dois métodos em velocidade. Isso indica que, ao contrário do DDQN, o DQN não investe tanto tempo na busca pela solução ideal, equilibrando melhor o tempo de computação e a qualidade da otimização.

Com isso, nota-se que o DQN demonstra superioridade em consultas complexas, reforçando sua eficiência. O DDQN, embora robusto, precisa de melhorias para reduzir o tempo de computação, e o PostgreSQL, apesar de confiável, torna-se ineficiente em cenários complexos devido à sua abordagem estatística limitada, evidenciando a necessidade de avanços no seu otimizador.

5.3 Considerações Gerais

A análise comparativa dos três otimizadores – PostgreSQL, Deep Q-Network (DQN) e Double Deep Q-Network (DDQN) – evidencia diferenças marcantes em desempenho, eficiência e consistência, especialmente em cenários de consultas complexas.

O PostgreSQL, apesar de ter um bom desempenho em consultas simples, apresenta dificuldades de eficiência em consultas com muitos *joins*. Seu alto custo total, grandes profundidades de árvores geradas e tempos de otimização elevados demonstram que sua abordagem estatística não retorna o melhor resultado quando comparado a otimizadores que usam aprendizado por reforço. A ampla variabilidade observada reforça a necessidade de aprimoramentos em seu otimizador.

O Deep Q-Network (DQN) se destaca pela sua eficiência, principalmente em consultas complexas. Ele consegue reduzir o tempo de otimização sem sacrificar significativamente a qualidade das árvores geradas, mantendo-se competitivo em termos de custo e profundidade. Essa característica faz do DQN uma solução promissora, equilibrando desempenho e velocidade de processamento.

Por outro lado, o Double Deep Q-Network (DDQN), embora apresente os melhores resultados em termos de profundidade de árvores e custo total das consultas, sofre com tempos de otimização mais elevados, especialmente em consultas complexas. Esse comportamento sugere que o DDQN investe mais tempo na busca por soluções ideais, gerando árvores mais otimizadas, mas com um custo computacional elevado. Assim, seu principal desafio reside na redução do tempo de processamento para torná-lo mais aplicável em cenários práticos.

6 CONCLUSÃO

Este trabalho apresentou uma solução para a otimização de junções de consultas SQL em sistemas de gerenciamento de bancos de dados (SGBDs), evidenciando avanços significativos no uso de algoritmos de aprendizado por reforço profundo quando comparados ao otimizador padrão do PostgreSQL. A implementação dos algoritmos de Deep Q-Network (DQN) e Double Deep Q-Network (DDQN) demonstrou que é possível superar as limitações dos otimizadores convencionais que são baseados em heurísticas e modelos estatísticos.

Os experimentos revelaram que, enquanto o PostgreSQL apresenta bom desempenho em consultas com menos junções (*joins)*, devido à sua abordagem baseada em estatísticas, ele enfrenta dificuldades em consultas mais complexas, com um número maior de *joins*. Nesses casos, o Deep Q-Network (DQN) destacou-se como o algoritmo mais eficiente em termos de tempo de otimização.

Por sua vez, o Double Deep Q-Network (DDQN) foi superior na construção de árvores de consulta com menor profundidade, o que potencialmente melhora o desempenho computacional em operações com muitos *joins*. No entanto, o algoritmo enfrenta limitações relacionadas ao elevado tempo de computação da otimização para consultas complexas, indicando a necessidade de melhorias nesse aspecto.

Os resultados obtidos reforçam o potencial dos algoritmos baseados em *deep reinforcement learning* para superar as limitações dos otimizadores tradicionais, especialmente em cenários de alta complexidade. Apesar disso, desafios como a redução do tempo de processamento de otimização e a avaliação do impacto desses algoritmos em ambientes de produção permanecem em aberto.

Futuras pesquisas podem explorar a aplicação desses algoritmos em outros cenários e sistemas, como a utilização de bancos de dados produtivos e consultas reais, já que este trabalho só utilizou um único banco de dados não-comercial e consultas de *benchmark*. Além disso, é necessário investigar estratégias para reduzir o tempo de processamento sem comprometer a qualidade das soluções, chegando a um equilíbrio entre entregar uma árvore de consulta ótima e um tempo de processamento rápido.

Outro aspecto essencial, que não foi abordado neste estudo e que é necessário para que os algoritmos de aprendizado por reforço possam ser implementados a nível comercial, é estudar a confiabilidade das árvores de consulta geradas. A consistência e precisão dessas árvores são fundamentais para o uso em ambientes de banco de dados comerciais.

Para o futuro, um outro algoritmo da biblioteca RLLib pode ser estudado, o *Proximal Policy Optimization* (PPO). Ele utiliza aprendizado baseado em políticas e tem um limite de *clipping* para regular atualizações, o que melhora a convergência do aprendizado. Para otimizações de consultas futuras, o PPO pode fornecer mais desempenho e facilidade de ajuste em comparação aos métodos de DQN e o DDQN, que são baseados no Q-learning.

Ainda pensando em avanços futuros, também é encorajado o uso de Deep Q-Network e Double Deep Q-Network para estudos que otimizem outras consultas que vão além da operação de junção. Pode-se, também, utilizar esses algoritmos de forma híbrida com otimizadores estatísticos, como o do PostgreSQL, a fim de utilizar o melhor desempenho em todos os cenários, como em consultas com poucos e com muitos *joins*. Assim, a otimização pode se estender às demais cláusulas de uma consulta.

REFERÊNCIAS

CHATGPT. **ChatGPT by OpenAI.** Disponível em: <https://chatgpt.com>. Acesso em: 9 dez. 2024.

CORONEL, C.; MORRIS, S. **Database Systems: Design, Implementation, & Management.** Cengage Learning, 2021.

ELMASRI, R.; NAVATHE, S. B. **Fundamentals of Database Systems.** Pearson, 2015.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. **Database Systems: The Complete Book.** Pearson, 2009.

GYMNASIUM. **Farama Gymnasium Documentation.** Disponível em: <https://gymnasium.farama.org/index.html>. Acesso em: 18 jul. 2024.

IMDB. **Non-commercial Datasets.** Disponível em: <https://developer.imdb.com/non-commercial-datasets/>. Acesso em: 9 maio 2024.

JARKE, M., & KOCH, J. (1984). **Query optimization in database systems**. *ACM Computing surveys (CsUR)*, *16*(2), 111-152.

KARAGIANNAKOS, **S**. (2018). **Deep Q-Learning Explained: A Beginner's Guide to Deep Reinforcement Learning.** Disponível em: <https://theaisummer.com/Deep_Q_Learning/>. Acesso em: 9 dez. 2024.

MARCUS, R., NEGI, P., MAO, H., ZHANG, C., ALIZADEH, M., KRASKA, T., ... & TATBUL, N. (2019). **Neo: A learned query optimizer**. *arXiv preprint arXiv:1904.03711*.

MARCUS, R., & PAPAEMMANOUIL, O. (2018). **Towards a hands-free query optimizer through deep learning**. *arXiv preprint arXiv:1809.10212*.

MITCHELL, T. M. **Machine Learning.** McGraw-Hill, 1997.

MNIH, Volodymyr et al. **Human-level control through deep reinforcement learning.** *Nature*, v. 518, n. 7540, p. 529–533, 2015.

ORTIZ, J., BALAZINSKA, M., GEHRKE, J., & KEERTHI, S. S. (2018, June). **Learning state representations for query optimization with deep reinforcement learning**. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning* (pp. 1-4).

POSTGRESQL. **PostgreSQL: The World's Most Advanced Open Source Relational Database.** Disponível em: <https://www.postgresql.org>. Acesso em: 9 maio 2024.

PYTHON. **Python: Programming Language.** Disponível em: <https://www.python.org>. Acesso em: 15 maio 2024.

RAHN, G. (2019). **join-order-benchmark**. Disponível em: <https://github.com/gregrahn/join-order-benchmark>. Acesso em: 16 jun. 2024.

RAMADAN, M., EL-KILANY, A., MOKHTAR, H. M., & SOBH, I. (2022). **RL_QOptimizer: A Reinforcement Learning Based Query Optimizer.** *IEEE Access*, *10*, 70502-70515.

RLLib. **RLlib: Industry-Grade, Scalable Reinforcement Learning.** Disponível em: <https://docs.ray.io/en/latest/rllib/rllib-algorithms.htm>. Acesso em: 23 jun. 2024.

SCHREINER, G. A., DUARTE, D., & MELLO, R. D. S. (2019). **When relational-based applications go to NoSQL databases: A survey.** Information, 10(7), 241. Disponível em: <https://doi.org/10.3390/info10070241>. Acesso em: 17 jul. 2024.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Database System Concepts**. 7. ed. New York: McGraw-Hill Education, 2020.

STILLGER, M.; LOHMAN, G.; MARKL, V.; KANDIL, M.. (2001) **LEO -DB2's LEarning Optimizer**. Disponível em: <https://www.vldb.org/conf/2001/P019.pdf>. Acesso em: 8 maio 2024.

STONEBRAKER, M.; KEMNITZ, G. **The Postgres Next-Generation Database Management System.** Communications of the ACM, 1991.

SUTTON, Richard S.; BARTO, Andrew G. **Reinforcement Learning: An Introduction**. 2. ed. Cambridge: MIT Press, 2018.

TENSORFLOW. **TensorFlow: An end-to-end open source machine learning platform.** Disponível em: <https://www.tensorflow.org>. Acesso em: 15 maio 2024.

VAN HASSELT, Hado; GUEZ, Arthur; SILVER, David. **Deep reinforcement learning with double Q-learning.** In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*. Phoenix, AZ: AAAI Press, 2016. p. 2094–2100.

YANG, Z., CHIANG, W. L., LUAN, S., MITTAL, G., LUO, M., & STOICA, I. (2022, June). **Balsa: Learning a query optimizer without expert demonstrations**. In *Proceedings of the 2022 International Conference on Management of Data* (pp. 931-944).

ZIVKOVIC, N. **Introduction to Double Q-Learning.** Disponível em: <https://rubikscode.net/2021/07/20/introduction-to-double-q-learning/>. Acesso em: 9 dez. 2024.