big data and cognitive computing

MDPI

Article

ReJOOSp: Reinforcement Learning for Join Order Optimization in SPARQL

Benjamin Warnke [1], Kevin Martens [1], Tobias Winker [1], Sven Groppe [1,*], Jinghua Groppe [1], Prasad Adhiyaman [2], Sruthi Srinivasan [2] and Shridevi Krishnakumar [2]

[1] Institute of Information Systems, University of Lübeck, 23562 Lübeck, Germany; jinghua.groppe@uni-luebeck.de (J.G.)
[2] Centre for Advanced Data Science, Vellore Institute of Technology, Chennai 600127, India
* Correspondence: sven.groppe@uni-luebeck.de

Abstract: The choice of a good join order plays an important role in the query performance of databases. However, determining the best join order is known to be an NP-hard problem with exponential growth with the number of joins. Because of this, nonlearning approaches to join order optimization have a longer optimization and execution time. In comparison, the models of machine learning, once trained, can construct optimized query plans very quickly. Several efforts have applied machine learning to optimize join order for SQL queries outperforming traditional approaches. In this work, we suggest a reinforcement learning technique for join optimization for SPARQL queries, ReJOOSp. SPARQL queries typically contain a much higher number of joins than SQL queries and so are more difficult to optimize. To evaluate ReJOOSp, we further develop a join order optimizer based on ReJOOSp and integrate it into the Semantic Web DBMS Luposdate3000. The evaluation of ReJOOSp shows its capability to significantly enhance query performance by achieving high-quality execution plans for a substantial portion of queries across synthetic and real-world datasets.

Keywords: Semantic Web database; SPARQL query optimization; join order optimization; deep learning; reinforcement learning

check for updates

Citation: Warnke, B.; Martens, K.; Winker, T.; Groppe, S; Groppe, J; Adhiyaman, P.; Srinivasan, S.; Krishnakumar, S. ReJOOSp: Reinforcement Learning for Join Order Optimization in SPARQL. *Big Data Cogn. Comput.* **2024**, *8*, 71. https://doi.org/10.3390/bdcc8070071

Academic Editors: Amna Dridi, Edlira Kalemi Vakaj and Mohamed Medhat Gaber

Received: 28 April 2024
Revised: 1 June 2024
Accepted: 11 June 2024
Published: 27 June 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Joins in a database query can be performed in many orders and the order of joins will significantly affect query performance. Finding optimal join ordering is, however, NP-hard [1]: For joining n inputs there are $\frac{(2 \cdot n-2)!}{(n-1)!}$ different possibilities (often called join trees). Join order optimization for finding optimal join execution orders efficiently is an active research field. Different strategies for join order optimization have been developed.

Greedy algorithms for join order optimization [2,3] select the next join candidate with the estimated smallest costs (i.e., typically the one with the smallest intermediate result) until the whole join order is determined. This strategy neglects join trees, where first joins with a slightly larger intermediate result are processed before those joins significantly reducing the number of intermediate results, or where a faster join algorithm can be chosen for later joins.

Whenever the optimal solution (i.e., the complete join tree for join ordering) can be constructed from the optimal solutions of the subproblems (i.e., the subjoins), dynamic programming [2,3] might be used to solve the problem (i.e., for finding the optimal join tree). In this way, the search space of possible join trees does not need to be enumerated completely, only a smaller subset of it without missing the optimal solution.

Another problem with join order optimization is that it does not only take time to execute the query plan but also takes more time to optimize the query [4]. Therefore, we need to balance the time used for optimization and the execution time of the query plan. Greedy algorithms are typically fast and take a short optimization time but they usually

generate query plans with longer execution time. In contrast, the execution time of query plans generated by dynamic programming is often small but the optimization time is quite long. For queries (like SPARQL queries) that contain a higher number of joins, the higher optimization time of dynamic programming becomes critical as shown in the evaluation part (see Section 5.2).

There have been research efforts to apply machine learning to join order optimization [5–12]. Training machine learning models may need a longer time, but once a model has learned, it can perform join optimization very quickly. Furthermore, these studies also show that the estimations delivered by ML models are usually quite good. Most contributions so far focus on relational database management systems [5–8]. Other contributions deal with estimating the execution time of a query and not with optimizing the query itself [9–12].

In this paper, we propose to utilize machine learning for optimizing the join order in SPARQL queries. Relational database management systems usually administer only a small number of tables and, hence, queries typically contain only a small number of joins [13,14]. In comparison, SPARQL queries typically require a much higher number of joins to retrieve triples in RDF. For example, some contributions like [4,15] report practical settings with up to more than 50 joins in SPARQL queries. Empirical studies [16] of real-world SPARQL queries have shown that joins are one of the most frequently used operations in SPARQL queries.

In this work we develop ReJOOSp, a reinforcement-learning-based join order optimizer for SPARQL DBMS. By using machine learning, join trees can be optimized with a speed of up to 200 queries per second. On synthetic datasets, our ReJOOSp generates good join trees for 80% of the queries with a much higher speed in comparison to nonlearning approaches. We qualify a join-tree as good, when it generates at most twice as many intermediate results compared to the best-case join order.

The remainder of this paper is organized as follows: In Section 2, we discuss related work, especially machine learning approaches dealing with join trees. In Section 3, we extend the Semantic Web DBMS Luposdate3000 [17] with our proposed join order optimizer which we developed for conducting experiments and evaluation. We present our reinforcement learning approach for join order optimization in Section 4. In Section 5 we experimentally evaluate the quality of the learned models. Finally, we discuss open research questions and conclude our work in Section 6.

2. Related Work

In Section 2.1, we introduce several existing machine learning approaches for SQL queries, and in Section 2.2, we introduce existing machine learning approaches for SPARQL queries. In Section 2.3, we introduce various join order optimization algorithms as used in existing Semantic Web databases.

2.1. Machine Learning Approaches for Optimizing SQL Queries

The major drawback of existing machine learning approaches to optimizing SQL queries is that the sizes of their vectors and matrices used in their calculations scale with the number of tables and columns. This hinders the direct use of these approaches for SPARQL optimization.

ReJOIN [5] iteratively returns the next best join from join candidates based on the already determined join tree using reinforcement learning and the Proximal Policy Optimization (PPO) algorithm to enumerate the join trees. The training dataset of ReJOIN was the IMDB dataset containing 103 queries for training and 10 for analysis (https://github.com/gregrahn/join-order-benchmark.git, accessed on 26 April 2024). In ReJOIN, each binary subtree is encoded as a row vector of size n, where n is the total number of relations in the database. When using SPARQL predicates as tables in a relational database, this becomes impractical because this vector would have the size of many thousands and would contain many zeros. To hold critical information about join predicates, ReJOIN internally

uses a square matrix of the size of the total number of relations for each episode, which would exceed the main memory for large datasets.

The paper [7] presents RTOS, a novel database query optimizer leveraging deep reinforcement learning (DRL) and tree-structured long short-term memory (Tree-LSTM) to improve join order selection (JOS). RTOS uses graph neural networks (GNNs) to capture the structural information of join trees, unlike existing methods that rely on fixed-length feature vectors. This approach not only supports modifications in database schema and multi-alias table names but also aims to generate plans with lower execution costs and latency. Extensive experiments demonstrate that RTOS outperforms traditional and existing DRL-based optimizers in both cost and latency, showcasing its potential as a more efficient solution for JOS. Similar to the ReJOIN approach, they use vectors and quadratic matrices, both of sizes of the number of tables, again being problematic for SPARQL query optimization.

The Proximal Policy Optimization Algorithm is the basis of the contribution in [8], proposing the Fully Observed Optimizer (FOOP). To avoid enumeration of join orders, this work uses a data-adaptive learning query optimizer to beat the performance of dynamic programming algorithms. The proposed approach uses bit vectors containing as many bits as there are columns in every table. This bit vector is one dimension of a state matrix, where the other dimension is scaled with the number of joins in a given query. While the memory consumption is much better in comparison to the previously discussed approaches, it still would scale with the number of predicates for SPARQL queries.

Recent approaches [18,19] propose using quantum computers as hardware accelerators for applying quantum machine learning to optimize SQL queries. Their proposed encoding scheme is tailored to quantum computing and cannot be applied in classical machine learning. The contributions [9–12] describe approaches for predicting the execution times of queries based on the query logs of previously executed queries. In these proposed approaches, feature vectors encode the queries, and by determining similar feature vectors of previous executions with already known execution times, the execution time of the given query is predicted.

Other approaches predict the cardinalities of queries of relational databases with traditional approaches like using sampling [20–23], the Selinger model [24] and join histograms [25,26], data-driven [27–32] and query-driven [33–36] learning of cardinalities, and hybrid approaches combining traditional and learning approaches (e.g., bound-based ones [37–39] and factor join [40]). Approaches like [11,12] convert queries into feature vectors for predicting the utilization of the number of CPUs together with the execution times, but do not calculate the join trees.

2.2. Machine Learning Approaches for Optimizing SPARQL Queries

The methodology [41] employs an RL-based optimizer to iteratively construct optimized join plans for SPARQL queries on RDF graphs. It selects joins minimizing estimated execution time, utilizing a neural network trained to predict query time. Join plans are represented as bottom-up trees, with leaf nodes as triple pattern result sets and internal nodes as join result sets. Vector representations of predicates are generated using RDF2Vec [42], while N-ary Tree-LSTM and Child-Sum Tree-LSTM [7] process intermediate and root nodes, respectively. To enhance efficiency, time-outs and experience replay are employed, and feature encoding replaces one-hot encoding for scalability. Open challenges for the approach presented in [41] include effectively encoding different join types between triple patterns, extending optimization to complex SPARQL operations, and developing variable representations for improved triple pattern encoding. In contrast to [41], our approach encodes any possible join type between triple patterns and is integrated into our Luposdate3000 query optimizer in such a way that more complex SPARQL queries containing other operations than joins can also be handled.

The SPARQL join query optimizer [6] utilizes RL-based algorithms to enhance storage structure selection, index selection, and query optimization for RDF graphs. For storage, it uses deep Q-learning (DQN) to split and merge graph tables based on specific patterns,

optimizing the storage for query efficiency. Index selection is handled with Q-learning to manage indices on columns, aiming to maximize efficiency within space constraints. For query optimization, the system focuses on ordering join operations for basic graph patterns (BGPs) to minimize execution time. This involves using deep convolutional neural networks to predict and select the optimal join order, with training employing double DQN to improve accuracy and ensure convergence. The overall approach ensures efficient and scalable query processing by leveraging advanced RL techniques and neural network architectures. However, this approach is limited to reordering basic graph patterns (BGPs) not supporting general bushy join trees as query execution plans. In contrast, our proposed ReJOOSp optimizer generates optimized general bushy join trees and can handle more complex SPARQL query operations.

The paper [43] addresses the optimization of multi-join queries specifically in the context of SPARQL queries, which are used to query RDF data. The study examines the challenges inherent in optimizing such queries, particularly in terms of exploring the search space and producing accurate cost estimates. The proposed solution leverages Ant Colony Optimization (ACO) and Q-Learning to effectively order joins and improve query execution plans. The approach in [43] is designed for a distributed setting, but our proposed ReJOOSp approach optimizes the join orders in SPARQL queries for a local setting.

The authors in [44] extended their previous approach for the collaboration between routing and distributed SPARQL processing by using machine learning to reduce the number of intermediate results. In contrast to their distributed setting, we present a machine learning approach for join order optimization in a centralized Semantic Web database.

2.3. Optimizing SPARQL Queries in Open Source and Commercial Semantic Web Databases and Triple Stores

In this section, we introduce those third-party Semantic Web databases and triple stores, which we use in the evaluation of this paper to compare different approaches to optimization. However, none of these DBMSs integrate machine learning for optimizing the join order.

The RDF3X DBMS [45] (https://gitlab.db.in.tum.de/dbtools/rdf3x, accessed on 26 April 2024) creates bushy join trees based on the RDF3X indexing scheme, which is applied in many other Semantic Web DBMSs. The Apache Jena DBMS (https://jena.apache.org/, accessed on 26 April 2024) is a fully open-source DBMS supporting all SPARQL functions, but generates only left-deep join trees during query optimization. The commercial Ontotext GraphDB DBMS (https://graphdb.ontotext.com/documentation, accessed on 26 April 2024) also only creates left-deep join trees, but the number of intermediate results of their optimized join orders is, in experiments, significantly lower compared to Jena.

3. Luposdate3000 Join Order Optimizer

To experimentally evaluate our machine learning technique to join order optimization for SPARQL queries, we develop a join order optimizer and integrate it into the Semantic Web DBMS Luposdate3000 [17].

Luposdate3000 [17] is designed for the semantic Internet of Things and for easily integrating extensions of new features. Coding the DBMS in Kotlin supports compilations to various targets to be executed on most of the widely used platforms on servers, desktop computers, smartphones, and browsers. Kotlin supports not only the Java virtual machine as a target but also binaries (with improved memory footprint) and JavaScript (e.g., for running native Web applications). Figure 1 depicts the overall query processing pipeline. For generating the training data, we extended Luposdate3000 to return the number of intermediate results for each executed query instead of the actual results (see the box "Executable for Intermediates" in Figure 1). In another extension, we integrated an interface to enforce a given join tree for executing the join. Please see the red box in Figure 1 for the extension to integrate the join order optimizer utilizing machine learning. The machine learning approach is realized with the Python programming language, the most widely-

used programming language in the machine learning community, and the basis of relevant frameworks for machine learning.

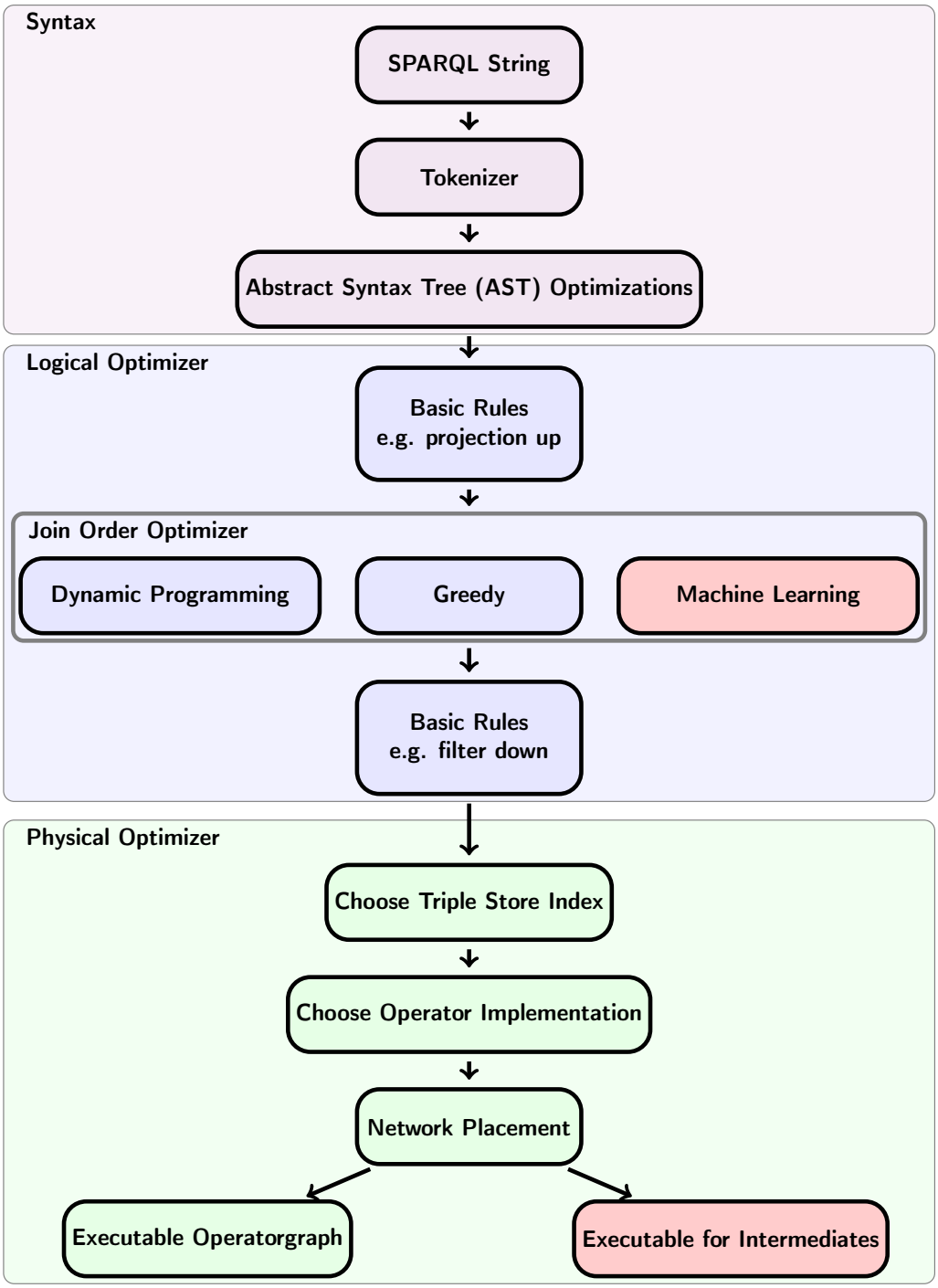


Figure 1. The pipeline for join order optimization in Luposdate3000 with the proposed extensions in the boxes with red background color. This figure was adapted from [44].

The two built-in join order optimizers of Luposdate3000 are a greedy optimizer and a dynamic programming one. Both optimizers do not rely on machine learning and can be used for comparing the quality of machine learning approaches with traditional approaches. Basic experiments using these traditional optimizers are crucial as they establish a performance baseline and provide insights into the strengths and limitations of conventional methods. The greedy optimizer, which is based on minimalistic histograms, is advantageous due to its simplicity and speed, offering linear execution time relative to the number

of inputs to join. On the other hand, the dynamic programming approach, although more computationally intensive with a runtime of $O(N \cdot 2^R)$ [46], generates higher-quality join trees where R is the number of inputs to join and N is the number of join orders.

4. Our Contribution

Our work consists of several contributions. First, we introduce our algorithm to generate the training and evaluation queries in Section 4.2. Then, we show and explain ReJOOSp in Section 4.3. The source code is freely available online (https://github.com/luposdate3000/luposdate3000, accessed on 26 April 2024).

4.1. Considerations for the Machine Learning Strategy

For machine learning approaches to optimize the join order, it does not make sense to label good or bad join orders manually. Hence, we designed a reinforcement learning approach and integrated the automatic improvement of its model on its own by exploration. While reinforcement learning algorithms belong to online learning approaches [47], we distinguish several variations of offline learning [47].

We propose to stop training when the model's performance on join order optimization is well enough for the following three reasons: Firstly, we can save CPU and GPU computations and, hence, time and energy for updating the model. Secondly, the data containing the statistics are not needed anymore after training, freeing up the occupied storage space for it. For retraining the model after many data updates or retrieving other types of queries, the statistics must be recomputed anyway because the data of the previous training are outdated. Thirdly, we follow the contribution in [48] by using different details for training and evaluation, such that the number of intermediate results retrieved for training in a time-consuming process is not required for evaluating queries and can be deleted after training.

4.2. Generating Queries

For training the machine learning models and evaluating them, we want to use SPARQL [49] queries, which contain exactly a certain number of joins to easily interpret the result. However, it is difficult to obtain enough queries for real-world queries to use for training. If applications pose queries with many joins, then these queries often do not vary much, such that it is difficult to obtain enough queries for machine learning approaches. Existing benchmarks originally designed for performance comparisons and not for machine learning purposes often also do not contain enough queries, or the queries are generated from query templates, resulting in many similar queries. Hence, we decided to generate SPARQL queries containing exactly n triple patterns based on given data. During our experiments, we learned the importance of queries with nonempty results for machine learning approaches: If the result of our generated queries is often empty, then machine learning approaches tend to prefer join trees that quickly detect the empty result. In real-world queries, almost every SPARQL-SELECT query returns a nonempty result [50].

The RDF graph represents the triples of RDF data as a directed graph, where the subjects and objects of triples are the nodes and the predicates of triples are directed labeled edges from the subjects to the object nodes (see Figure 2). For generating SPARQL queries, we choose a random (but connected) subgraph with n distinct edges in the RDF graph. For our example and the RDF graph in Figure 2, we choose the bold edges AB, BC, and CD with their predicates a, b, and c, respectively. Note that we propose a randomized algorithm, i.e., the chosen subgraphs are of any shape, including paths, star patterns, and circles, as well as any combination of those. By design, generated queries follow the given structures in the data, i.e., real-world queries for the same data probably will contain shapes similar to the generated queries.

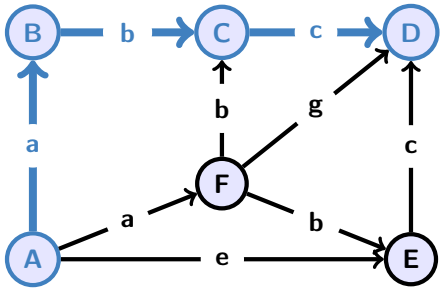


Figure 2. RDF graph for the example about generating queries, where we use the blue nodes and edges.

For generating the SPARQL queries, we replace all node labels with variable names. Figure 3 contains the generated query for this example. By design, the result of the generated query contains at least one result, but because RDF graphs often contain many similar subgraphs, the result size is likely much larger. In our experiments, some of the generated queries returned one result, some contained up to 20 results, and the results of some queries were huge. We observed similar issues for real-world queries: The intended result size of some queries is small, like "Who published x?"; other queries have a bigger but limited number of results like "Which articles are published by x?"; and other queries are expected to return many results like "Which articles are published in journal x?".

```
SELECT * WHERE {
?A a ?B .
?B b ?C .
?C c ?D .
}
```

Figure 3. Example of a generated query.

4.3. Our Approach ReJOOSp

RDF data consists of a set of triples such that this basic component, i.e., the triple, represents less information than a relational table with many columns. Because of this issue, SPARQL queries usually contain more joins than SQL queries. This dramatically increases the input size for machine learning making it harder for the model to learn what is important.

The simple example in Figure 4 retrieves the authors of an article along with the title and volume of the journal in which the article appears. Assuming the number of articles to be higher than the number of journals, a reasonable join order would first combine the triple patterns of lines 4 and 5 parallel to a second star-join formed by the triple patterns in lines 2 and 3. The last operation would finally join the results of both star-joins.

```
1  SELECT * WHERE {
2    ?article dc:creator ?author .
3    ?article swrc:journal ?journal .
4    ?journal dc:title ?title .
5    ?journal swrc:volume ?volume .
6  }
```

Figure 4. Example of an SPARQL query.

In our implementation, we apply the Stable Baselines 3 [51] framework supported by a large number of open-source contributors including a community addon [52] for integrating masking in the built-in PPO model. We identified the masking of actions as one of the key features for generating good join trees. We use this model with its default parameters. These are are shown in Table 1. Furthermore the algorithm is set to normalize the advantage. The later described reward, observation, and possible actions

are implemented as a gym [53] environment. The model obtains its information from this environment and alters its state by giving back the next action. Our architecture consists of the default MLP policy with 2 layers and 64 nodes, which is a common architecture in other approaches to join order optimization as well (e.g., [54]), and we also obtained a good performance with this architecture.

Table 1. The hyperparameters of the Proximal Policy Optimization (PPO) algorithm.

Parameter	Value
Learning rate	3×10^{-4}
Discount factor	0.99
Clipping parameter	0.2
Number of epochs when optimizing surrogate loss	10
Value function coefficient for the loss calculation	0.5
Maximum value for the gradient clipping	0.5
Entropy coefficient for the loss calculation	0
Trade-off between bias and variance for the Generalized Advantage Estimator	0.95

The reward of our machine learning approach is designed to reflect the quality of the join tree compared to other previously run join trees. Consequently, our proposed algorithm needs to know the previous best and worst cases. Calculating the reward for joins with up to five inputs is not time-consuming by just traversing the small number of all the join trees and determining the best and the worst join trees. For a larger number of joins, like 20 joins as considered in our experiments, this approach is too time-consuming and, hence, impractical. To handle a larger number of queries, we refined the pipeline of our machine learning approach to the one shown in Figure 5. Firstly, we generate queries as described in Section 4.2. Then, we run the default optimizer for collecting the total number of intermediate results of each SPARQL query, but not the corresponding join trees, avoiding the fact that the model just learns the generated join trees of the default optimizer. Furthermore, we add the number of intermediate results of a new current join tree whenever we calculate a new reward during training. We store all statistics about pairs of queries and their total number of intermediate results in a separate relational database. In this way, the statistics are continuously improved during training.

Figure 6 contains the function to calculate the reward. The variables v_{max} and v_{min} refer to the min and max of the number of intermediate results for the given query, while $v_{current}$ represents the number of intermediate results for the current join tree. Even if the real best and worst cases are not known, this estimation is good enough to train the model. We designed our reward function to reward good join trees and penalize those with a high number of intermediate results or where the DBMS aborts our benchmark because of a time-out. Hence, we use a logarithmic reward function, additionally considering a minimum to compensate for high variations potentially occurring for different join trees. Figure 7 depicts an example run containing the internal states of the important variables while constructing the optimal join tree, where each row in the figure shows a time step.

The observation matrix is first initialized according to the SPARQL query to optimize. For this purpose, the literals of the SPARQL query are replaced with numbers from a global dictionary, which assigns a unique positive number to each literal, starting at 1. Since the same dictionary is used for every query, the model can learn the cardinalities of the constants in the queries. To be able to distinguish between literals and variables, the variables in the given SPARQL query are transformed into negative numbers, starting with -2. Because the variables are always renamed to a sequence of negative numbers, the model is also able to learn patterns of variables in the queries to calculate the best join order. For example, the SPARQL query of Figure 4 is transformed into a sequence of triples containing numbers (representing the triple patterns) presented as follows: $[(-2,\ 1,\ -3),\ (-2,\ 2,\ -4),\ (-4,\ 3,\ -5),\ (-4,\ 4,\ -6)]$.

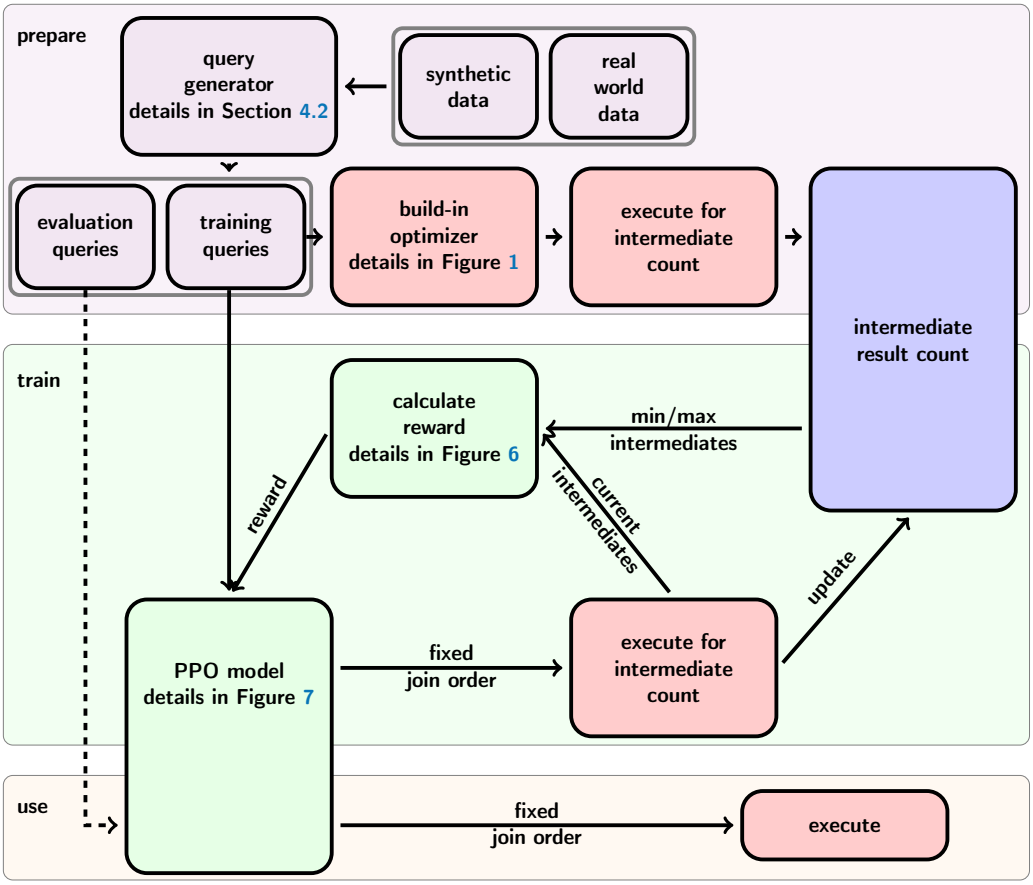


Figure 5. Our architecture for using machine learning for join order optimization: The blue component stores its data in an external SQL DBMS. The red components are executed within Luposdate3000. The other components are implemented in Python. This figure was adapted from [44].

$$r = \begin{cases} 0 & \text{if } v_{max} = v_{min} \\ -10 & \text{if } v_{current} = null \\ min\left(10, -\log\left(\frac{v_{current} - v_{min}}{v_{max} - v_{min}}\right)\right) & \text{otherwise} \end{cases}$$

Figure 6. Our reward function uses v_{min} and v_{max} from the statistics, which refer to the currently known best and worst case number of intermediate results. $v_{current}$ may receive a *null* value when the DBMS runs into a timeout. *null* values are not considered for calculating the known worst case.

The observation matrix is determined by transforming this list of triples into the diagonal of a square matrix. This square matrix represents the state of already determined joins and is quadratic in the allowed maximum number of joins. Hence, we can also use the same model for optimizing queries with a smaller number of joins than the allowed maximum number of joins. All other entries of the rows of the diagonal are filled with $(-1, -1, -1)$, and so far, not-used rows are filled with $(0, 0, 0)$ entries. The purpose of the matrix is that each join is mapped to a matrix of the same size.

In the matrix, a list of pairs of rows, which could be joined, is defined as possible actions. Furthermore, the difference between the left and right input of a join operator will be ignored, and this aims to reduce the search space of machine learning. To understand this better, we use Figures 4 and 7 as an illustration. Figure 4 describes an SPARQL query consisting of four triple patterns, and possible actions from these triple patterns are defined in Figure 7. When an action leads to rows consisting of only 0 values, the subsequent actions on these rows will not make sense, and such actions are masked out using the gray color in the figure.

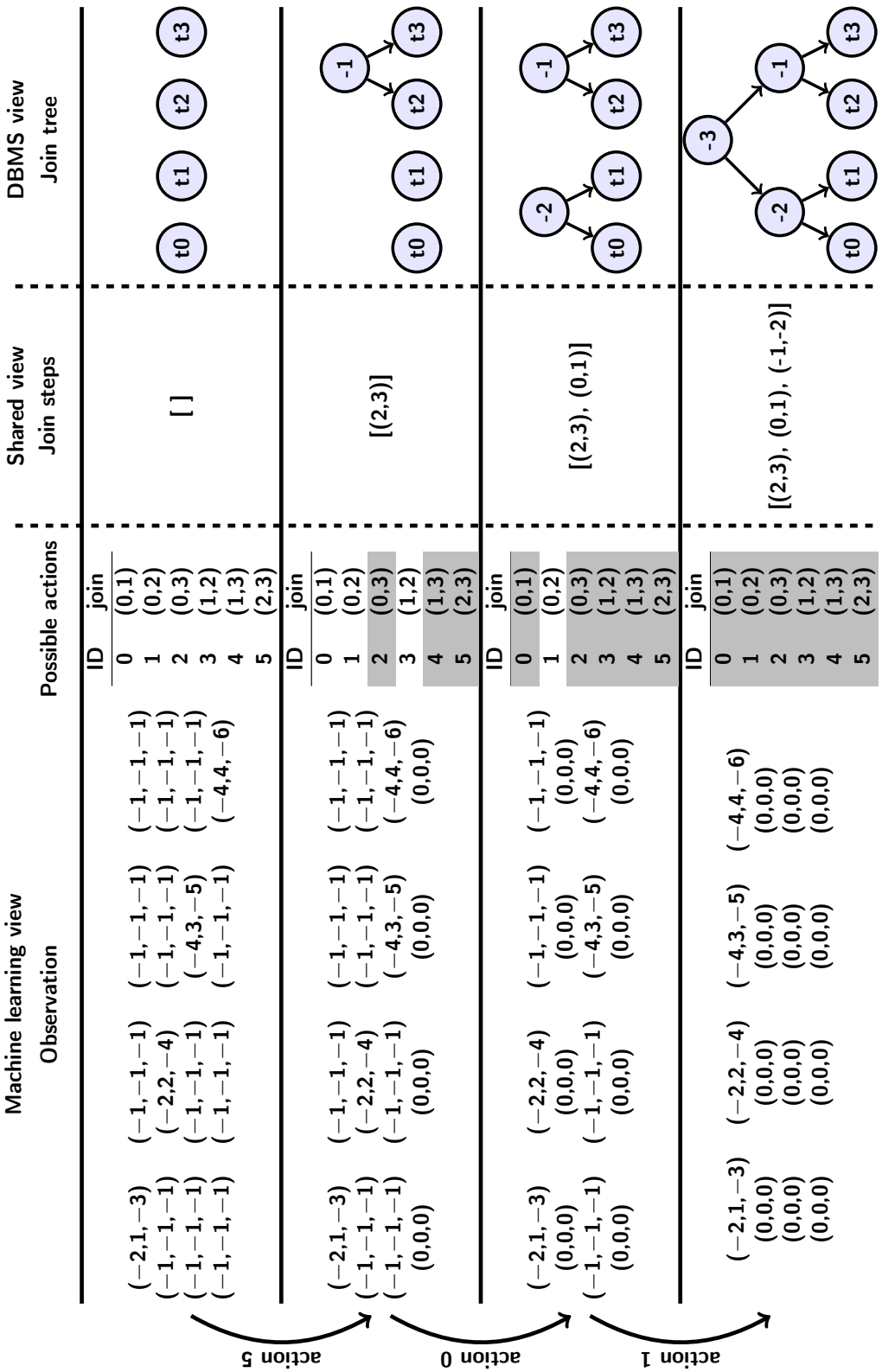


Figure 7. Overview of join order optimization with our machine learning approach. This figure was adapted from [44]. Possible actions with a gray background are invalid and masked out.

Given the matrix and an action list, the machine learning model selects a possible action and this triggers a step function. The step function first determines the two inputs to be joined by looking up the action ID in the action list, then updates the observation matrix by copying the values from the matrix row of the right join input to the matrix row of the left join input, but skipping all fields with values of $(-1, -1, -1)$ to avoid overwriting existing data. Finally, the entire row of the right join input is set to $(0, 0, 0)$.

The final observation matrix contains only the value $(0, 0, 0)$ except for the first row, which is the final matrix for all the chosen join orders. Hence, to be able to build the join tree in the final step, our approach has to log all selected join steps of the machine learning model, which is presented in the right column of Figure 7.

5. Evaluation

In this chapter, we first introduce the evaluation environment. Then, we evaluate the time needed to generate the query plans. Then, we evaluate the quality of the generated query plans and compare them to other optimizer approaches that do not use machine learning. Finally, we show that the model indeed learns something useful.

5.1. Environment

We ran our benchmarks on an INTEL i9-10900K 5.1GHz CPU with 128 GB of memory. However, the benchmarks only required about 32 GB. We use Ubuntu 21.10 as the operating system. When training multiple models in parallel, we encountered a memory bandwidth bottleneck. Therefore, we used an NVIDIA GeForce RTX 2080 Rev. A with 8 GB of memory to train several models in parallel. To train only a single model, the GPU is not needed. To evaluate the queries, we modified the Luposdate3000 DBMS in such a way that it outputs the number of intermediate results instead of the actual result of a query. To compile Luposdate3000, we used the Kotlin compiler 1.7.255 and its Java target with Java 16 to run it. For our machine learning, we used the stable baselines3 1.5.0 with the modifications of the maskable PPO model [52] running on Python 3.9.7.

We used the synthetic SP^2B dataset [55] generator to produce a 2^{20} triples dataset. Additionally, we used the WordNet real-world dataset (http://WordNet-rdf.princeton.edu/static/WordNet.nt.gz, accessed on 31 August 2022) containing 2,637,168 triples. The queries were generated according to Section 4.2. All the queries contain only basic triple patterns, which should be joined together.

Depending on the number of triple patterns to join, the training phase took between 1 h and 20 h. At the beginning of the training, the execution time is dominated by the timeout for a single query, because the more joins there are to be optimized, the higher the probability of choosing a very bad join order. When dealing with many joins, bad decisions during the join order optimization yield execution plans that will never terminate. Every learning approach that needs to execute some sample queries during training will suffer from this same problem; therefore, we do not consider this as a too-long training time. The size of the triple store also influences the time requirement of the training, mainly because slightly bad join orders become very bad due to the increased amount of data for every join iterator.

First, we compare the time needed to optimize the join tree in Figure 8. The greedy optimizer is the fastest one generating bad join trees according to our evaluation in the following sections. The dynamic programming optimizer needs much more time to optimize join trees. At approximately 11 join inputs, our machine-learning-based optimizer is faster than the dynamic programming optimizer during the generation of the join tree. The next sections will focus on the quality of the generated join orders.

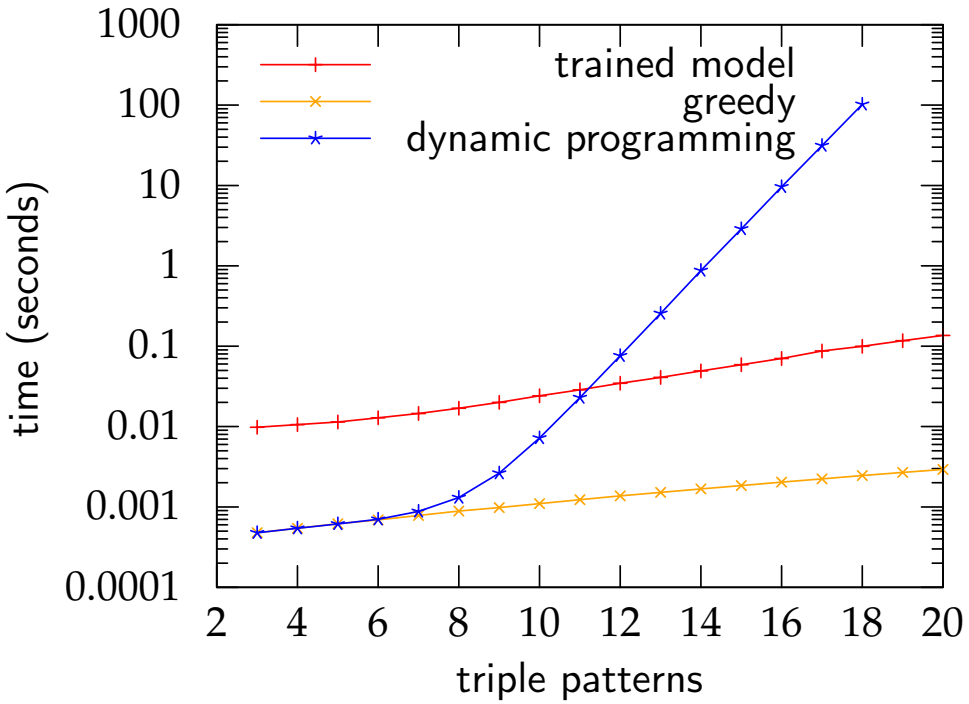


Figure 8. Time needed to construct an optimized join tree with a given number of inputs to join in the Luposdate3000 DBMS. The optimization of queries was repeated until 60 s were spent for each join size, then the average time per query was calculated. The benchmarks were executed on the hardware described in Section 5.

5.2. Evaluating Different Numbers of Triple Patterns

The *x*-axis in Figure 9 represents the various models used for comparison (where the numbers following "model trained on" represent the number of joins used during training), and the *y*-axis shows the number of joins when evaluating the models. In the left part of Figure 9, we used the explain function of other DBMSs to also compare their generated join trees with the one of our machine learning approach. GraphDB and Jena only generate deep left join trees. The Jena DBMS produces the worst query plans in the context of synthetic data, while RDF3X produces the worst results in the context of real data, as shown in Figure 10. Despite this, the RDF3X DBMS uses bushy join trees, which could allow parallel processing of independent subtrees. However, this is insufficient in the WordNet dataset, as the generated join orders are extremely bad compared to all other optimizers. Next, we investigate the results of our built-in join order optimizers in Luposdate3000. First, a greedy join order optimizer always chooses the two entries where the estimated result is the smallest. Second, in Luposdate3000 we implemented a join order optimizer for dynamic programming. Here, the quality of the join order is better, but its performance drops drastically with a higher number of joins (see Figure 8).

Overall, none of the deterministic synthetic data approaches can produce reproducible good join orders for a very large number of triple patterns. On real data, none of the join order optimizers tested achieved good results for more than eight joins. We can see that each trained model can optimize joins with up to six entries relatively well. On the right side of the diagram, we see different machine learning optimizers trained with different numbers of join inputs. As expected, these trained models provide plausible results over a wider range of inputs when used on synthetic datasets. In the context of real data, the mixed models show no improvement over other models. However, these results are inferior to specially trained models in both datasets, meaning that the user is probably better off training multiple models, each for the desired purpose. Furthermore, the figure shows that the models perform better for the number of join inputs they were trained on. While this result is fairly obvious, it can also be seen that each model can optimize join trees of

comparable size in the range from $n-3$ to $n+3$ very well. In the case of synthetic data, this can be seen in the graph. With real data, the measurements show a similar result, but this is much less noticeable in the figure because the training effect is weaker overall.

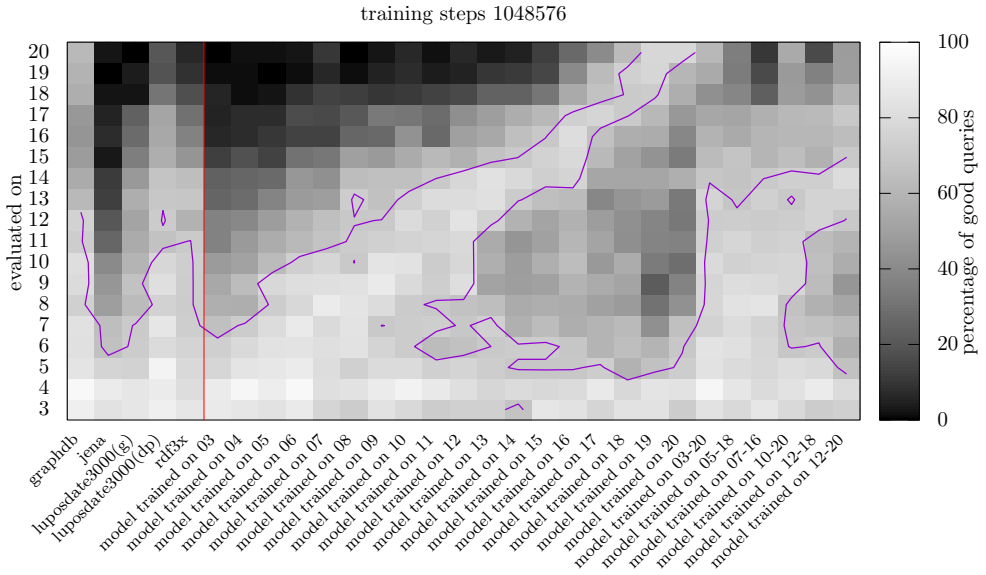


Figure 9. SPARQL evaluation on SP^{2}B-Dataset. The chart shows the percentage of queries that require at most twice as many intermediate results as the known best case. The magenta line surrounds the area, where 70% of the queries receive good join trees. The results of baselines are presented left from the red line and the results of our proposed approach are right from the red line.

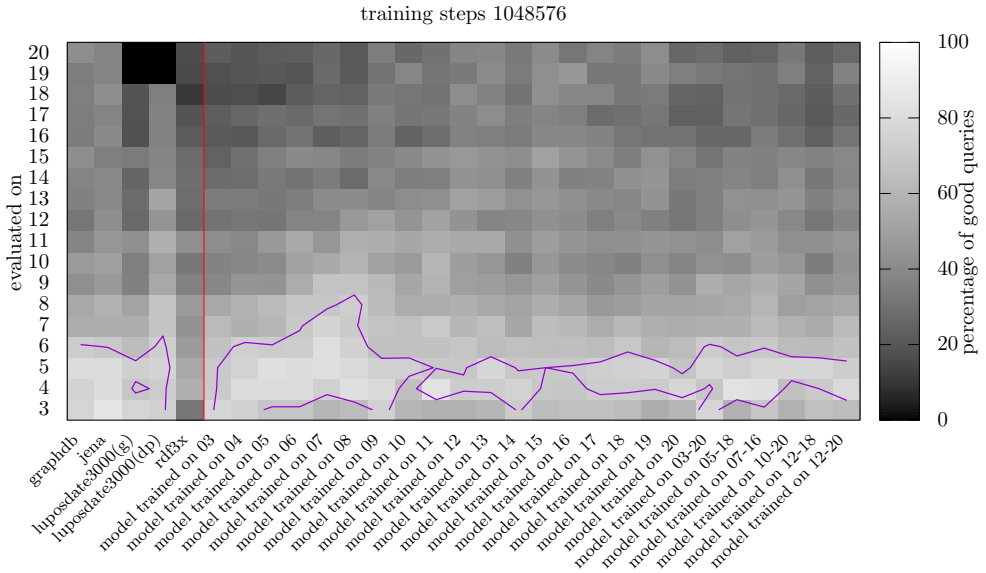


Figure 10. SPARQL evaluation on WordNet-Dataset. The chart shows the percentage of queries that require at most twice as many intermediate results as the known best case. The magenta line surrounds the area, where 70% of the queries receive good join trees. The results of baselines are presented left from the red line and the results of our proposed approach are right from the red line.

5.3. Evaluating Different Numbers of Training Steps

In Figure 11, we trained multiple models for queries with different numbers of joins. All models were tested with queries containing 16 triple patterns. The *y*-axis shows how long each model was trained. This should make it clear how long the training must last to achieve good results. On the left, we present the results of some join order optimizers without machine learning to show that machine learning on the synthetic dataset produces

better results than traditional query optimizers. For the real-world dataset, the quality of the join orders generated by machine learning is slightly worse. In addition, in our experiments, there is currently no optimizer that can reliably generate good join orders (see Figure 12).

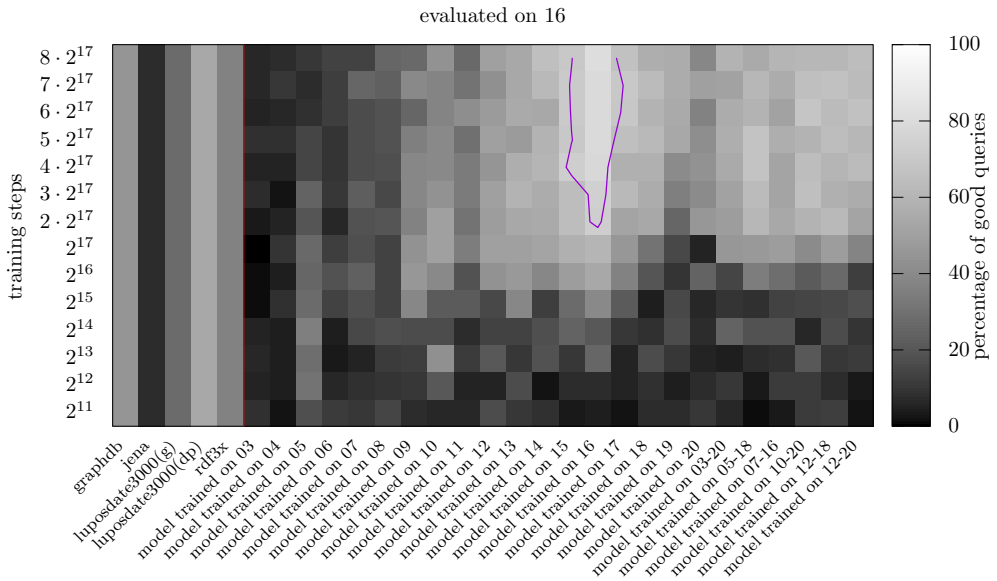


Figure 11. Different number of training steps on SP^{2}B-Dataset. The chart shows the percentage of queries that require at most twice as many intermediate results as the known best case. The magenta line surrounds the area, where 70% of the queries receive good join trees. The results of baselines are presented left from the red line and the results of our proposed approach are right from the red line.

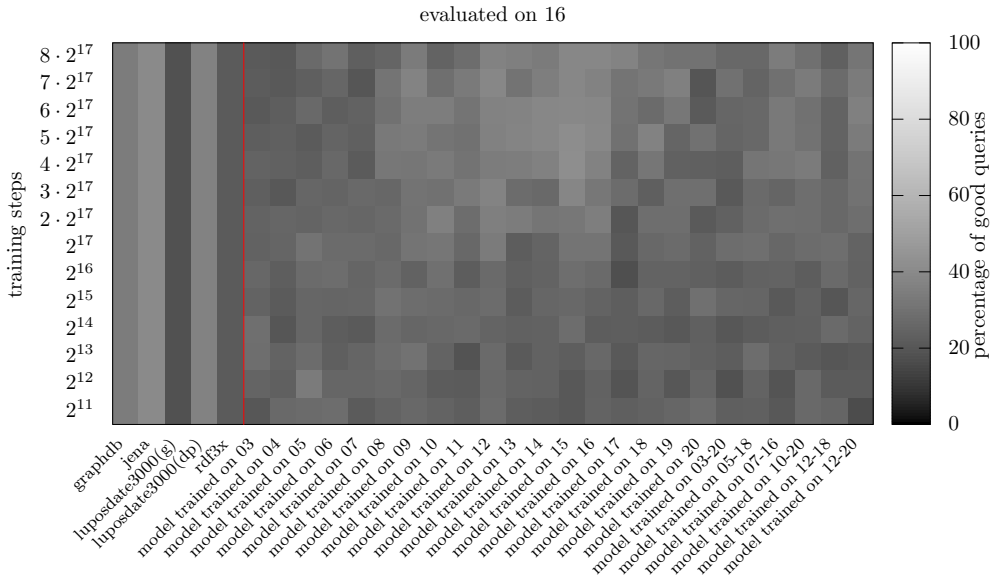


Figure 12. Different number of training steps on WordNet-Dataset. The chart shows the percentage of queries that require at most twice as many intermediate results as the known best case. The results of baselines are presented left from the red line and the results of our proposed approach are right from the red line.

The synthetic dataset shows that the models need to be trained at least 2^{17} steps for good results. The quality distribution for the WordNet dataset is similar, but the overall quality is lower. Furthermore, one can see that only the models trained with a similar number of joins can produce good join trees. In the diagrams on the right, we trained

models with mixed numbers of triples. The idea is that these models should be usable for a wider range of queries. However, the results show that these models for synthetic data can only be used for larger joins after a longer training time. With real-world data, these mixed models do not work at all. We conclude that it is better to explicitly train a new model for each desired join tree size.

5.4. Evaluating What the Model Has Learned

Synthetic datasets often receive good join orders, so it is unlikely that the quality of the join order is purely random. To prevent the model from learning one join order per number of joins, we took a closer look at what join order is used. For three and four triple patterns, we often find that half of the possible join orders are used. The unused join orders simply swap the left and right operands while keeping the same structure. When five or more triple patterns should be joined, no join tree is used in more than 1.5% of the queries. Among the join orders for 20 input triples, no join order is used in more than 0.03%. From this, we can conclude that either the model learns the cardinality of the predicates or the model learns to recognize patterns in the query structure and avoids Cartesian products.

5.5. Limitations

The results show that ReJooSp provides significantly better join orders than conventional join order optimizers such as GraphDB, Jena, Luposdate3000, and RDF3X on synthetic data. From Section 5.4, we also know that the model does not randomize join orders or learn the Cartesian products, making it a really good join order optimizer for synthetic data. However, it does suffer through some limitations, such as the following: 1. Performance of the model on real-world data does not scale up as well as on synthetic data. One possible reason for this might be the heterogeneity and diversity of real-world data; 2. ReJOOSp must be retrained for any new database and its typical queries; however, this issue is not specific to ReJOOSp and is inherited from the applied concept of machine learning.

6. Conclusions and Future Work

SPARQL queries typically contain a large number of joins and finding good join ordering is so time-consuming for SPARQL queries. In this work, we propose ReJOOSp, a novel reinforcement-learning-based approach tailored for SPARQL query optimization, and develop a join order optimizer that integrates ReJOOSp into the Semantic Web DBMS Luposdate3000 for the evaluation of ReJOOSp. We evaluate ReJOOSp and compare it with nonlearning approaches (Jena, GraphDB, RDF3X) on synthetic datasets and real-world datasets. The evaluation of ReJOOSp highlights its capability to significantly enhance query performance by achieving high-quality execution plans for a substantial portion of queries across synthetic and real-world datasets.

In the future, we plan to perform data augmentation and collect more real-world data, and evaluate ReJOOSp performance on these extended datasets. Also, in the future, we plan to train a generalized join order optimizer that would optimize queries from any database. Furthermore, there is a need for broader metric evaluations including network traffic implications. We also plan to explore these avenues to enhance the robustness and applicability of ReJOOSp in diverse operational contexts.

Author Contributions: Conceptualization, B.W., K.M., S.G., J.G., P.A., S.S., and S.K.; methodology, B.W., K.M., S.G., J.G., P.A., S.S., and S.K.; software, B.W., K.M., T.W., P.A., and S.S.; validation, B.W., K.M., P.A., and S.S.; formal analysis, B.W., K.M., P.A., and S.S.; data curation, B.W., K.M., P.A., and S.S.; writing—original draft, B.W., K.M., P.A., and S.S.; writing—review and editing, B.W., K.M., T.W., S.G., J.G., P.A., S.S., and S.K.; supervision, S.G., J.G., and S.K.; funding acquisition, S.G. All authors have read and agreed to the published version of the manuscript.

Funding: This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)— Project-ID 422053062 and the German Federal Ministry of Education and Research within the funding program quantum technologies—from basic research to market—contract number 13N16090.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The datasets analyzed during the current study are publicly available from the following links (accessed on 31 August 2022): http://wordnet-rdf.princeton.edu/static/WordNet.nt.gz; https://dbis.informatik.uni-freiburg.de/forschung/projekte/SP2B/.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Scheufele, W.; Moerkotte, G. On the complexity of generating optimal plans with cross products. In Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, AZ, USA, 12–14 May 1997; pp. 238–248.
2. Allam, J.R. Evaluation of a Greedy Join-Order Optimization Approach Using the IMDB Dataset. Ph.D. Thesis, University of Magdeburg, Magdeburg, Germany, 2018.
3. Lan, H.; Bao, Z.; Peng, Y. A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. *Data Sci. Eng.* **2021**, *6*, 86–101. [CrossRef]
4. Gubichev, A.; Neumann, T. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT, Proceedings of the International Conference on Extending Database Technology, Athens, Greece, 24–28 March 2014*; Amer-Yahia, S., Christophides, V., Kementsietsidis, A., Garofalakis, M.N., Idreos, S., Leroy, V., Eds.; Open Proceedings: Konstanz, Germany, 2014; pp. 439–450.
5. Marcus, R.; Papaemmanouil, O. Deep Reinforcement Learning for Join Order Enumeration. In Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, New York, NY, USA, 10 June 2018. [CrossRef]
6. Wang, H.; Qi, Z.; Zheng, L.; Feng, Y.; Ouyang, J.; Zhang, H.; Zhang, X.; Shen, Z.; Liu, S. April: An Automatic Graph Data Management System Based on Reinforcement Learning. In Proceedings of the 29th ACM International Conference on Information and Knowledge Management, Online, 19–23 October 2020; Volume 20, pp. 3465–3468.
7. Yu, X.; Li, G.; Chai, C.; Tang, N. Reinforcement Learning with Tree-LSTM for Join Order Selection. In Proceedings of the 2020 IEEE 36th International Conference on Data Engineering (ICDE), Dallas, TX, USA, 20–24 April 2020; pp. 1297–1308.
8. Heitz, J.; Stockinger, K. Join Query Optimization with Deep Reinforcement Learning Algorithms. *arXiv* **2019**, arXiv:1911.11689.
9. Hasan, R.; Gandon, F. A Machine Learning Approach to SPARQL Query Performance Prediction. In Proceedings of the 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), Washington, DC, USA, 11–14 August 2014; pp. 266–273. [CrossRef]
10. Ganapathi, A.; Kuno, H.; Dayal, U.; Wiener, J.L.; Fox, A.; Jordan, M.; Patterson, D. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In Proceedings of the 2009 IEEE 25th International Conference on Data Engineering, Shanghai, China, 29 March–2 April 2009; pp. 592–603. [CrossRef]
11. Gupta, C.; Mehta, A.; Dayal, U. PQR: Predicting Query Execution Times for Autonomous Workload Management. In Proceedings of the 2008 International Conference on Autonomic Computing, Chicago, IL, USA, 2–6 June 2008; pp. 13–22.
12. Zhang, W.E.; Sheng, Q.Z.; Qin, Y.; Taylor, K.; Yao, L. Learning-based SPARQL query performance modeling and prediction. *World Wide Web* **2017**, *21*, 1015–1035. [CrossRef]
13. Lu, H.; Chan, H.C.; Wei, K.K. A survey on usage of SQL. In Proceedings of the ACM SIGMOD Record, Washington, DC, USA, 25–28 May 1993; Volume 22, pp. 60–65. [CrossRef]
14. Zolaktaf, Z.; Milani, M.; Pottinger, R. Facilitating SQL query composition and analysis. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 14–19 June 2020; pp. 209–224.
15. Paasche, S.; Groppe, S. Enhancing Data Quality and Process Optimization for Smart Manufacturing Lines in Industry 4.0 Scenarios. In Proceedings of the International Workshop on Big Data in Emergent Distributed Environments (BiDEDE'22), Philadelphia, PA, USA, 12–17 June 2022.
16. Arias, M.; Fernández, J.D.; Martínez-Prieto, M.A.; de la Fuente, P. An Empirical Study of Real-World SPARQL Queries. *arXiv* **2011**, arXiv:1103.5043.
17. Warnke, B.; Rehan, M.W.; Fischer, S.; Groppe, S. Flexible data partitioning schemes for parallel merge joins in semantic web queries. In Proceedings of the Datenbanksysteme für Business, Technologie und Web (BTW), Dresden, Germany, 13–17 September 2021. [CrossRef]
18. Winker, T.; Groppe, S.; Uotila, V.; Yan, Z.; Lu, J.; Franz, M.; Mauerer, W. Quantum Machine Learning: Foundation, New Techniques, and Opportunities for Database Research. In Proceedings of the ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD), Washington, DC, USA, 18–23 June 2023.

19. Çalıkyılmaz, U.; Groppe, S.; Groppe, J.; Winker, T.; Prestel, S.; Shagieva, F.; Arya, D.; Preis, F.; Gruenwald, L. Opportunities for Quantum Acceleration of Databases: Optimization of Queries and Transaction Schedules. *Proc. VLDB Endow.* **2023**, *16*, 2344–2353. [CrossRef]
20. Leis, V.; Radke, B.; Gubichev, A.; Kemper, A.; Neumann, T. Cardinality Estimation Done Right: Index-Based Join Sampling. In Proceedings of the 8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, 8–11 January 2017.
21. Li, F.; Wu, B.; Yi, K.; Zhao, Z. Wander Join: Online Aggregation via Random Walks. In Proceedings of the 2016 International Conference on Management of Data, New York, NY, USA, 26 June–1 July 2016. [CrossRef]
22. Lipton, R.J.; Naughton, J.F.; Schneider, D.A. Practical selectivity estimation through adaptive sampling. In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 23–25 May 1990. [CrossRef]
23. Lipton, R.J.; Naughton, J.F.; Schneider, D.A. Practical selectivity estimation through adaptive sampling. *SIGMOD Rec.* **1990**, *19*, 1–11. [CrossRef]
24. Selinger, P.G.; Astrahan, M.M.; Chamberlin, D.D.; Lorie, R.A.; Price, T.G. Access Path Selection in a Relational Database Management System. In Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, MA, USA, 30 May–1 June 1979; pp. 23–34.
25. Ioannidis, Y.E. The History of Histograms (abridged). In Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, 9–12 September 2003; Freytag, J.C., Lockemann, P.C., Abiteboul, S., Carey, M.J., Selinger, P.G., Heuer, A., Eds.; Morgan Kaufmann: Burlington, MA, USA, 2003; pp. 19–30.
26. Ioannidis, Y.E.; Christodoulakis, S. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Trans. Database Syst.* **1993**, *18*, 709–748. [CrossRef]
27. Getoor, L.; Taskar, B.; Koller, D. Selectivity estimation using probabilistic models. In Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 21–24 May 2001; pp. 461–472. [CrossRef]
28. Tzoumas, K.; Deshpande, A.; Jensen, C.S. Lightweight graphical models for selectivity estimation without independence assumptions. *Proc. VLDB Endow.* **2011**, *4*, 852–863. [CrossRef]
29. Tzoumas, K.; Deshpande, A.; Jensen, C.S. Efficiently adapting graphical models for selectivity estimation. *VLDB J.* **2012**, *22*, 3–27. [CrossRef]
30. Wang, J.; Chai, C.; Liu, J.; Li, G. FACE: A normalizing flow based cardinality estimator. *Proc. VLDB Endow.* **2021**, *15*, 72–84. [CrossRef]
31. Yang, Z.; Kamsetty, A.; Luan, S.; Liang, E.; Duan, Y.; Chen, X.; Stoica, I. NeuroCard: One cardinality estimator for all tables. *Proc. VLDB Endow.* **2020**, *14*, 61–73. [CrossRef]
32. Zhu, R.; Wu, Z.; Han, Y.; Zeng, K.; Pfadler, A.; Qian, Z.; Zhou, J.; Cui, B. FLAT: Fast, lightweight and accurate method for cardinality estimation. *Proc. VLDB Endow.* **2021**, *14*, 1489–1502. [CrossRef]
33. Kipf, A.; Kipf, T.; Radke, B.; Leis, V.; Boncz, P.A.; Kemper, A. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In Proceedings of the 9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, 13–16 January 2019.
34. Liu, J.; Dong, W.; Zhou, Q.; Li, D. Fauce: Fast and accurate deep ensembles with uncertainty for cardinality estimation. *Proc. VLDB Endow.* **2021**, *14*, 1950–1963. [CrossRef]
35. Sun, J.; Li, G. An end-to-end learning-based cost estimator. *Proc. VLDB Endow.* **2019**, *13*, 307–319. [CrossRef]
36. Negi, P.; Marcus, R.; Kipf, A.; Mao, H.; Tatbul, N.; Kraska, T.; Alizadeh, M. Flow-loss: Learning cardinality estimates that matter. *Proc. VLDB Endow.* **2021**, *14*, 2019–2032. [CrossRef]
37. Atserias, A.; Grohe, M.; Marx, D. Size Bounds and Query Plans for Relational Joins. In Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, Philadelphia, PA, USA, 25–28 October 2008; pp. 739–748.
38. Cai, W.; Balazinska, M.; Suciu, D. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In Proceedings of the 2019 International Conference on Management of Data, Amsterdam, The Netherlands, 30 June–5 July 2019. [CrossRef]
39. Habich, D.; Krause, A.; Pietrzyk, J.; Faerber, C.; Lehner, W. Simplicity done right for SIMDified query processing on CPU and FPGA. In Proceedings of the 1st Workshop on Simplicity in Management of Data, SiMoD@SIGMOD 2023, Bellevue, WA, USA, 23 June 2023; Porobic, D., Wang, T., Eds.; ACM: New York, NY, USA, 2023; pp. 1–5.
40. Wu, Z.; Negi, P.; Alizadeh, M.; Kraska, T.; Madden, S. FactorJoin: A New Cardinality Estimation Framework for Join Queries. *Proc. ACM Manag. Data* **2023**, *1*, 41. [CrossRef]
41. Eschauzier, R.; Taelman, R.; Morren, M.; Verborgh, R. Reinforcement Learning-Based SPARQL Join Ordering Optimizer. In Proceedings of the Semantic Web: ESWC 2023 Satellite Events: Hersonissos, Crete, Greece, 28 May–1 June 2023; pp. 43–47. [CrossRef]
42. Ristoski P, P.H. Rdf2vec: Rdf graph embeddings for data mining. In Proceedings of the Semantic Web–ISWC 2016: 15th International Semantic Web Conference Proceedings, Part I 15, Kobe, Japan, 17–21 October 2016.
43. P Karthikeyan, M.; Krishnaveni, K.; Le, D.N. Analysis of Multi-Join Query Optimization Using ACO and Q-Learning. *Int. J. Comput. Digit. Syst.* **2024**, *15*, 1–11.
44. Warnke, B.; Fischer, S.; Groppe, S. Using Machine Learning and Routing Protocols for Optimizing Distributed SPARQL Queries in Collaboration. *Computers* **2023**, *12*, 210. [CrossRef]

45. Neumann, T.; Weikum, G. The RDF3X engine for scalable management of RDF data. *Vldb J. VLDB* **2010**, *19*, 91–113. [CrossRef]
46. Iker, B.R.; Swami, A.N. Method for Optimizing Processing of Join Queries by Determining Optimal Processing Order and Assigning Optimal Join Methods to Each of the Join Operations. U.S. Patent US5345585A, 2 December 1991.
47. Levine, S.; Kumar, A.; Tucker, G.; Fu, J. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. *arXiv* **2020**, arXiv:2005.01643.
48. Lample, G.; Chaplot, D.S. Playing FPS Games with Deep Reinforcement Learning. In Proceedings of the AAAI Conference on Artificial Intelligence, San Francisco, CA, USA, 4–9 February 2017; Volume 31. [CrossRef]
49. Seaborne, A.; Harris, S. SPARQL 1.1 Query Language. Technical Report. 2013. Available online: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/ (accessed on 14 June 2024).
50. Bonifati, A.; Martens, W.; Timm, T. An Analytical Study of Large SPARQL Query Logs. *Proc. VLDB Endow.* **2017**, *11*, 149–161. [CrossRef]
51. Hill, A.; Raffin, A.; Ernestus, M.; Gleave, A.; Kanervisto, A.; Traore, R.; Dhariwal, P.; Hesse, C.; Klimov, O.; Nichol, A.; et al. Stable Baselines. 2018. Available online: https://github.com/hill-a/stable-baselines (accessed on 14 June 2024).
52. Huang, S.; Ontañón, S. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. In Proceedings of the International FLAIRS Conference Proceedings, Clearwater Beach, FL, USA, 14–17 May 2023; Volume 35. [CrossRef]
53. Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; Zaremba, W. Openai gym. *arXiv* **2016**, arXiv:1606.01540.
54. Krishnan, S.; Yang, Z.; Goldberg, K.; Hellerstein, J.M.; Stoica, I. Learning to Optimize Join Queries With Deep Reinforcement Learning. *arXiv* **2018**, arXiv:1808.03196.
55. Schmidt, M.; Hornung, T.; Lausen, G.; Pinkel, C. SP2Bench: A SPARQL Performance Benchmark. *arXiv* **2008**, arXiv:0806.4627.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.