Toulouse: Learning Join Order Optimization Policies for Rule-based Data Engines

Antonios Karvelas[1], Yannis Foufoulas[1,2], Alkis Simitsis[2] and Yannis Ioannidis[1,2]

[1] *University of Athens, Athens, Greece*
[2] *Athena Research Center, Athens, Greece*

Abstract

In recent times, several research works have explored the idea of leveraging machine learning techniques to improve or even replace core components of traditional database architectures, such as the query optimizer and selectivity and cardinality cost estimators. These efforts often rely on existing, cost-based optimizers and cost models to avoid a cold-start, and build on top of the optimizer's decisions. In this paper, we investigate whether learning could also be beneficial in rule-based optimizers for known and unknown workloads alike. As a proof of concept, we use MonetDB, an open-source, column-store analytics data engine, and explore whether a learning model based on Graph Neural Networks that is trained on a cost-based engine, such as PostgreSQL, could improve MonetDB optimizer's decisions. Our experimental results reveal deficiencies in MonetDB's query execution plans, especially for queries with long chains of join operators, and potential opportunities in exploiting learning techniques.

Keywords

Query processing, Query optimization, Join ordering, Machine learning, Graph neural networks, Reinforcement learning, Proximal policy optimization, Cost-based optimization, Rule-based optimization

1. Introduction

Query optimization has been a long-standing challenge from the very early days of data management systems. Traditional optimization techniques to search the space of alternative query execution plans and find the most efficient one include various heuristic and cost-based approaches, rule-based strategies, randomized algorithms, and so on [1, 2].

Cost-based optimizers depend on accurate cardinality and selectivity estimates, and especially for intermediate result, to produce reasonable plans. In general, these techniques are successful given specific assumptions such as attribute value independence, uniformity, data independence, etc. When these assumptions cannot be met the optimizers typically fall back to an educated guess. In practice, getting such accurate estimates is a significant challenge. And the challenge is more evident in queries having a long chain of join operators, where one should decide in which order the joins should be computed, forming the so-called join-ordering problem. On the other hand, rule-based techniques rely on a set of rules to produce query execution plans, usually with a single pass of the SQL statement. The rules are well-crafted based on experience, theory, and common practice, but they often miss optimization opportunities as they tend to overlook data properties.

A recent line of research exploits the advances made recently in machine learning (ML) technology and explores the potential of the so-called, learning query optimizers, which aim at learning the behavior of query operators and query patterns over time and tend to learn also from their previous decisions (i.e., execution plans). In the context of learning query optimization, earlier research efforts focus mainly on the problems of join ordering and end-to-end optimization. However, the vast majority of past work focuses on cost-based optimizers, relying on their choices and respective cost models to avoid a cold-start and also train the proposed models.

In this work, we present *Toulouse*, our optimization approach to rule-based data engines, and investigate the feasibility and potential benefit of applying learning techniques to a rule-based query optimizer. Our initial focus is on the join ordering optimization problem. Generally speaking, without having a cost model and cost estimates assigned to query plans and query operators, the problem formulation should rely on optimization patterns for the query workloads at hand.

As a proof of concept, we used MonetDB [3] in our investigation. A query in MonetDB undergoes an optimizer pipeline, which examines and applies a

DataPlat'23: 2nd International Workshop on Data Platform Design, Management, and Optimization, March 28, 2023, Ioannina, Greece
sdi1600060@di.uoa.gr (A. Karvelas); johnfouf@di.uoa.gr (Y. Foufoulas); alkis@athenarc.gr (A. Simitsis); yannis@di.uoa.gr (Y. Ioannidis)
© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

series of optimizer steps [4]. Our preliminary investigation showed that a featurization scheme based solely on the optimizer steps is not sufficient to produce useful learning patterns. Subsequently, we tried to leverage a model trained on a cost-based optimizer and obtained three interesting results. First, MonetDB's optimizer presents significant limitations when it comes to queries having long chains of join operators and thus, it seems that does not handle very effectively the join ordering problem. Second, that a model trained on a cost-based optimizer following a supervised approach behaves reasonably well when used in MonetDB, without requiring significant fine-tuning or massaging of MonetDB's execution runtime and optimizers pipeline. We find this result very interesting, as the alternative would be to develop a brand new rule-based optimizer step and add it to MonetDB's optimizer pipeline. Third, the approach can be further extended to support query and workload agnostic optimization using a more elaborate, reinforcement learning approach.

To the best of our knowledge, our work is the first attempt to employ learning in rule-based database optimization. Our contributions can be summarized as follows:

- We argue that training a model using a set of query optimization rules does not seem as effective as relying on a cost model.
- We show that applying a model trained on a cost-based optimizer to a rule-based system does seem to be effective, especially, if the underlying system is not optimized for join ordering.
- We present a simple, supervised learning approach able to transfer effective schema and workload specific optimization policies to a rule-based system.
- We present a more elaborate, reinforcement learning approach that transfers effective schema and workload agnostic optimization policies to a rule-based system.

Outline. Section 2 presents related efforts on learning query optimization. Section 3 describes the main principles and components of Toulouse. Section 4 presents our experimental evaluation. Finally, Section 5 summarizes our findings.

2. Related Work

There are several approaches to learning query optimization [5]. These include system frameworks, e.g., SageDB [6], that aim at replacing core database components, such as data structures, indices, and query execution with learned components.

Other approaches focus on the join ordering problem. DQ [7] and ReJOIN [8] combine reinforcement learning (RL) with a human-engineered cost model to automatically learn search strategies to navigate the space of possible join orderings. RTOS [9] extends these by employing a Tree-LSTM to compute query cost. SkinnerDB [10] uses RL to learn optimal join orders during query execution and enables fast join order switching. AlphaJoin [11] uses Monte Carlo Tree Search (MCTS) for join ordering and Adaptive Decision Network (ADN) to choose between plans produced by AlphaJoin (for long queries) and PostgreSQL (for short queries).

There are also attempts to end-to-end learning optimizers. Neo [12] replaces most traditional optimizer components with ML models and deep neural networks. Bao [13] follows a schema and data agnostic approach to learning optimization. It runs a query with a predefined set of hints and keeps the plan ranked first by a tree convolutional neural network. Microlearner [14] divides a complex cloud workload into a hierarchy of subsets, using them to learn micro-models independently and in parallel.

These methods focus on cost-based optimizers. Although several of these techniques could also be used in our work, we consider them complementary to our goal: explore whether a learning optimization approach could be beneficial for rule-based systems.

3. The Toulouse approach

In this section, we present our attempts towards learning optimization for rule-based query optimization. First, we report on our investigation toward building a learning model directly from query optimization rules. Next, we present a different direction: employ a simple, supervised approach to train a model on a cost-based system and apply it to a rule-based system. Finally, we present a more elaborate, reinforcement learning approach that serves reasonably well previously unseen (i.e., not included in the model training) query workloads.

3.1. Learning optimization rules

Our first attempt was to learn directly from query optimization rules. Interestingly, this attempt did not produce very positive results.

Under the hood, MonetDB uses a low level intermediate language called MAL (MonetDB Assembly Language) that encodes the execution complexity of a SQL statement. Roughly speaking, a MAL pro-

gram is equivalent to a query plan. Instead of using a single optimizer, MonetDB employs a collection of 20+ query optimizer transformers, each responsible for a single task such as removing scalar expression or aliases, avoid multiple calculations of the same operator, range propagation, code parallelization, optimize resource usage, garbage collection, operator rewriting, caching aggregation results, dead code removal, etc. [15]. There is also an optimizer that deals with join paths, i.e., looking for join operators and cascading them into multiple join paths, and considers strategies such as materialization of common subpaths. Additional optimizers can be implemented and added as needed.

Hence, query optimization in MonetDB is realized as a linear sequence of MAL transformations (optimizers) forming an optimizer pipeline. An optimizer may be called multiple times in the same optimizer pipeline. MonetDB employs a cost model that to date does not use any statistics on the actual data, but relies on the size of rows provided as an input to an optimizer. We found that this feature is not very mature yet and it does not seem to expose the costs to the runtime.

Inspired by Bao [13] that gets training data by running a query with various hint sets, our first attempt was to explore alternative variations of optimizer pipelines and employ both the optimizers and various common optimizer pipeline patterns as features in our featurization scheme. This can be realized easily, as MonetDB offers the functionality to enable or disable optimizers via a 'set optimizer' command. On the other hand, there are a few constraints to consider. First, there are dependencies among the optimizers; e.g., static evaluation may happen only after constants propagation. In addition, there is a notion of the minimal optimizer pipeline, which specifies a number of absolutely necessary optimizers to run. These constraints limit significantly the degrees of freedom we can consider.

Our modeling and design efforts compared to the out-of-the-box MonetDB optimization strategies, revealed limited benefits in generic optimizations for simple SPJ queries including aggregation and sort operations, but the results on join ordering were rather disappointing as the improvements seemed to be sporadic and random, which renders this path neither robust nor practical.

3.2. Employ a cost-based learning model

We explored next the idea of developing a model based on a cost-based query optimizer and investigating whether this would be effective on a rule-based system. This could be seen as a type of a transfer learning approach. In our current implementation, we used PostgreSQL as the reference cost-based system and MonetDB as the target rule-based system. Hence, our approach could be summarized as 'develop (train) a model that learns how the PostgreSQL optimizer works (with its successes and mistakes) and employ it (inference) on MonetDB bypassing its optimization choices'.

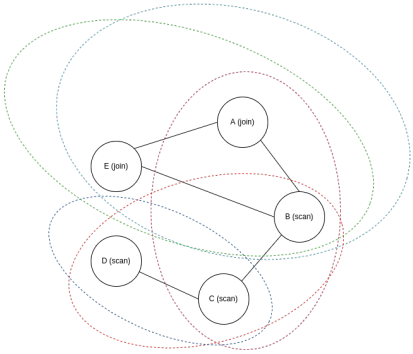


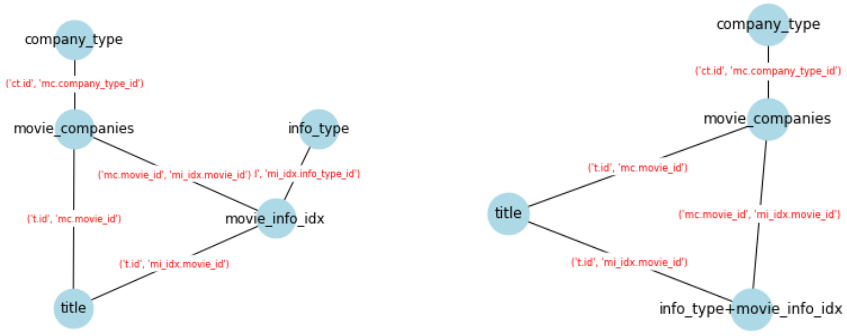
Figure 1: Example message passing network

Several available methods in the literature are bound to a specific schema and fail to generalize well and/or require expensive training. Still, previous approaches following a graph neural network (GNN) modeling (without this being the only good solution) provide promising results [5]. Inspired by this observation, Toulouse employs a GNN approach to collect structural information about all possible join decisions and learn how to pick the best one in each case, or at least try to approximate it and learn more about the problem through this process.

Next, we present the two techniques developed in Toulouse, a supervised approach and a reinforcement learning approach. The first relies heavily on the training data, and thus it is amenable to specific schema and query workloads. The later aims at providing a schema and query agnostic solution.

3.2.1. Preliminaries: GNN to the rescue

Graph Neural Networks (GNN) is a relatively new class of deep learning techniques that aim at solving the problem of having structural data of variable complexity, by aggregating data from their immediate neighbors. GNNs have been effective for problems that can be described as graphs, entities that have variable size and structure and where the interactions between them are as important as their features. They can be best understood as message passing networks, where every nodes exchanges data with its directly connected nodes (see also Figure 1).



(a) An example query graph

(b) Applying a join decision

Figure 2: Toulouse graph representation

With every added GNN layer, every node gets information from a larger neighborhood. We gather messages from all neighboring nodes and then reduce them with a sequence invariable function (sum, mean, max etc). Formally, the result of the k-th layer for node v would be:

$$h_k^v = \sigma(W_k \sum \frac{h_u^{k-1}}{|N(v)|} + B_k h_v^k - 1)$$

where the sum calculates the means of the neighboring values (embeddings), W_k and B_k are trainable parameters (neural networks), and σ is a non-linear function like ReLU.

Graph Convolutional Networks (GCN) are one of the most widely used architecture of GNNs. Every level results in:

$$X' = \hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}X\Theta$$

where X is the result of the previous level, $\hat{A} = A+I$ is the adjacency matrix with inserted self-loops (the $+I$), $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ is the diagonal degree matrix, and Θ is the layer's trainable weights.

GCNII is a continuation of GCN with the added features of initial residual connections and identity mapping, which help tremendously with the problem of oversmoothing (i.e., the more levels in a GNN, the less expressive are the later layers). To support this functionality, we change the GCN formulation as follows:

$$X' = ((1-a)\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}X + aX^{(0)})((1-\beta)I + \beta\Theta)$$

where $X^{(0)}$ represents the initial features that get added to each layer.

3.2.2. Join optimization as an MDP

Similar to earlier efforts (e.g., [7]), we formulate the join-ordering optimization problem as a Markov Decision Process (MDP).

We formulate each query as an undirected graph $G = (V, E)$, where every node $v \in V$ is a table (i.e., a table scan or the result of previously joined tables) and every edge $e \in E$ is an implicit join predicate. At every step of the process, we choose an edge $e = (v_1, v_2)$ and *join* the two connected nodes v_1 and v_2. We remove those two nodes from the graph, add a new node v_1v_2 and reconnect it with all the nodes previously connected to either v_1 or v_2.

Figures 2a and 2b illustrate an example query graph inspired by the Join Order Benchmark (JOB) [16]. Figure 2b is produced by Figure 2a after considering a join between $info\_type$ and $movie\_info\_idx$. In doing so, we remove those two nodes, replace them with the new $info\_type + movie\_info\_idx$ node, and reconnect it with all the nodes previously connected to either $info\_type$ or $movie\_info\_idx$.

Every such a step has a cost, and as we describe it as an MDP, we can assume that it follows the Bellman's Principle of Optimality, thus we can approximate a function, or else a policy, that selects the best action at each step, assuming that all proceeding steps are also chosen optimally.

Therefore, the problem we deal with is to find a sequence of joins in the query graph that minimizes the total cost, until after considering all joins needed there is only one node left. Formally, in a query graph G, with a cost model J, we search for the sequence of joins $c_1 \circ c_2 ... \circ c_T$ with the minimum cost $min_{c_1,...,c_T} \sum_{i=1}^{T} J(c_i)$.

3.2.3. A supervised approach

As a first approach to the problem, Toulouse employs a supervised approach using a simple GNN architecture. We first present the featurization scheme we use and then the architecture of the solution.

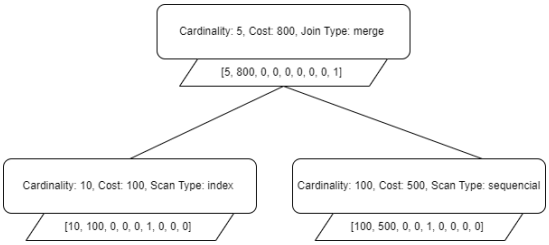


Figure 3: Featurization example: cardinality, cost, and node type

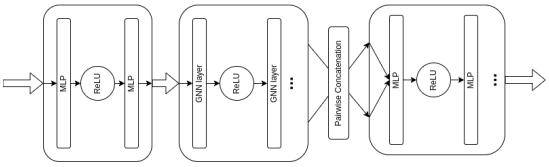


Figure 4: Neural network architecture of Toulouse's supervised approach

Featurization. Every node carries an embedding with one-hot encoded join types and scan types (e.g., merge-join, index scan, etc.), along with cardinality and cost estimates. Figure 3 illustrates an example of our featurization scheme with node (join and scan) types, cardinality, and cost estimates. To account for the numeric values of the estimates in our featurization, we scale the corresponding cardinality and cost slots in the one-hot vector to the values that they correspond to. For example, the first slot that corresponds to cardinality of the one-hot vector for the top node in Figure 3 reads 5 to account for the cardinality of that node. Obviously, our encoding scheme could be enriched with additional features as well.

Architecture. Using the pair embeddings from multiple layers of GNNs, each join has information about itself and about every other potential join. In the join ordering problem, direct neighbors are the most important, but general information about the rest of the graph provides insight about the future consequences of a join decision.

Figure 4 presents the neural network structure of the supervised approach in Toulouse. It employs two Multilayer Perceptron (MLP) encoding layers, three GNN layers that learn the query structure as described previously, and four MLPs that process the concatenated embeddings of the pair of nodes passing through the GNNs.

Our hyperparameter tuning analysis indicated that 2 and 4 MLPs, respectively, are sufficient for the scenarios we tested. We chose 3 GNNs to match the typical diameter (i.e., the length of the longest shortest path between any two graph nodes) in realistic query graphs, which is also a value that seems to balance nicely the trade-off between over-smoothing and performance boosting.

3.2.4. A reinforcement learning approach

Although our supervised approach provides a simple and fast solution, still our investigation shows that it does not capture effectively the case of unknown query workloads, i.e., queries and schemas that have not been seen by the model before. Related efforts have dealt with the problem of cost estimation for unseen queries considering transfer learning inspired techniques, such as zero-shot learning (e.g., [17]). Although past work has considered zero-shot learning for learned cost prediction, we believe that the basic principles could also apply to the join-ordering problem as well.

Towards this end, we investigate a more complex architecture that employs a Graph Convolutional Neural Network (GCN) - Reinforcement Learning (RL) architecture, aiming at providing a more practical solution to the join-ordering problem, developing a model that could also cope with *new* query workloads.

Featurization. As we aim at a more query and schema agnostic approach, we choose to use as general features as possible. Hence, for each node, we consider:

- cardinality of an operator (here, scan or join)
- width; the estimated average width of the result (in bytes)
- selectivity; $total\_tuples/cardinality$ for scan operators, and the average of the two subtrees for the join operators (multiplying them is prohibited due to vanishing gradient issues)
- total tuples; can be set to 0 for joins or the averages, see selectivity above.
- total pages
- $isJoin$; true for joins, false otherwise
- $usedInAgg$; whether the node contributes in an aggregation.

We also consider a number of (boolean) features indicating whether the query contains specific predicate expressions, such as filterEQ, filterNEQ, filterLike, filterIs, filterIn, filterBetween, filterGTELTE, and so on.

Finally, we complement our featurization scheme with features related to the edges of the graph, essentially capturing information about the join predicate they represent, and use statistics for the columns to be joined, such as:

- index; whether an index exists for the column(s)
- null_frac; the fraction of nulls in the column(s)
- n_distinct; the number of distinct elements in the column(s)
- correlation; a PostgreSQL metric showing the statistical correlation between physical row ordering and logical ordering of the column values, which is useful to determine whether for example an index scan can be efficient (similar info is captured in most modern databases).

Currently, Toulouse employs the PostgreSQL optimizer as a source of the cost/statistics estimates, but another approach could also be used e.g., leveraging Bayesian Optimization to build performance models [18].

Architecture. In this approach, Toulouse employs a GCN architecture in combination with the popular Proximal Policy Optimization (PPO) algorithm. In particular, it uses multiple levels of GCNII, combines them and passed them through two MLP heads (actor and value) that can be used for PPO. The value head represents the cost of query so far and the actor head represents the probability of choosing a certain join at each step.

For each step, we maintain two graphs, the one described above and its line graph. A line graph of an undirected graph G is another graph $L(G)$ that is created as follows: for each edge in G, we make a vertex in $L(G)$; for every two edges in G that have a vertex in common, we make an edge between their corresponding vertices in $L(G)$ [19]. In other words, we convert every edge of G to a node in $L(G)$ and every node of G to an edge in $L(G)$. Therefore, we end up with every node carrying its own features (of the edge turned to node) and of the features of the corresponding adjacent nodes that in the line graph were turned to edges. We also concatenate global features such as the number of nodes, the diameter of the graph, etc. that help enrich the graph representation. In addition, we calculate positional encoding with random walks [20] to encode positional information of graph nodes and make them more expressive. Nearby nodes have similar positional features and distant nodes have dissimilar positional features. Still, we keep the position encoding separately from the features. Finally, we pass each node through the model shown in Figure 5.

The x features are passed through the layers of GCNII. We pool the sum of each separate layer and concatenate them at the end to compute the value head. The positional features, *pos*, are passed through separate GCNII layers and we keep only the last one, which we concatenate with the layer results (a.k.a. tensors). The layer results are used directly in the actor head and the pooled ones in the value head. The actor heads ends up in a categorical distribution. The categorical distribution is used to sample join actions during training and choose the best action with argmax during inference.

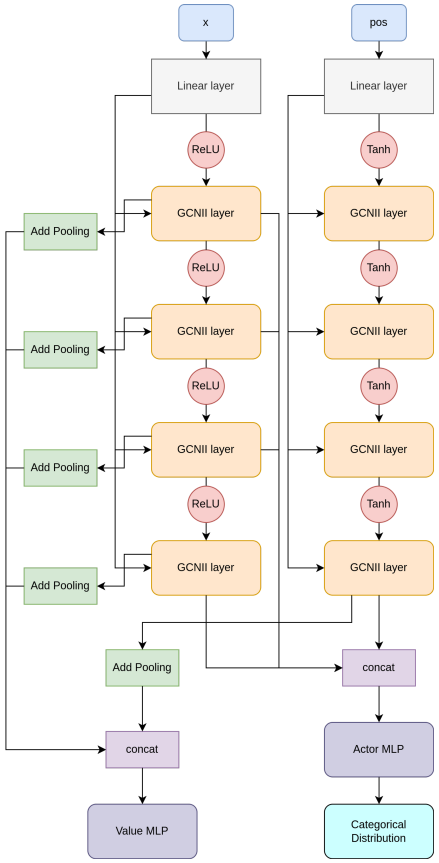


Figure 5: Network architecture of Toulouse RL approach

Reward function. An important part of any reinforcement learning implementation is the reward function. For our model, we work as follows. For the training of our model we use a multiplicity of datasets (databases) and query workloads to achieve as broad and generic learning as possible. (We explain this in more detail in the next section.) Then, we choose random queries from all the available workloads (evenly distributed to avoid any bias toward a specific workload), create the graphs with the tables of all the schemata, add all implicit connections (join predicates), add also the join predicates that we get from equality propagation, and finally,

we proceed to use the model for choosing the next best join. Hence, at each step we compute the reward as follows:

```
cut_parameter = 10.0
for database, query in queries:
  base_cost = postgres_cost(query)
  while number_of_nodes > 1:
    choose_join(query)
    cost = make_join(query)
    if cost > base_cost * cut_parameter:
      reward = - 10.0 * cut_parameter
      done = True
    else:
      reward = log(cost)
      reward = - (reward / reward_running_std)
```

The cut_parameter is set at a starting value experimentally configured (e.g., 10-20) and then we slowly reduce it as the training progresses. This helps avoid bad plans from diluting our training data, as instead of continuing when we consider a join than costs much more than what the query optimizer (e.g., PostgreSQL's optimizer) can do by default, we penalize it appropriately using the cut_parameter and continue with the next query. Otherwise, we *log* the cost and use running statistics (Welford algorithm) to normalize it. Another approach would be to penalize more the initial steps of a query, rather the later ones; e.g., a non-performant join in the first or second steps is worse than a suboptimal join toward the tail of the chain of join operators.

It is worth mentioning two advantages we find in favor of the reinforcement learning approach. First, it shows a potential to alleviate a limiting data generation performance challenge we faced with the supervised approach; that is, queries with many joins (e.g., more than 10 joins) result into a huge number of potential join paths, which in turn increases significantly the training time. In addition, the reinforcement learning approach is amenable to online improvement, as we could fine-tune the produced model at runtime using real query running times and potentially have a system that continuously learns as query workloads run. We consider this interesting optimization opportunity as future work.

3.2.5. Application to the rule-based system

To close the loop, the model that has been trained on a cost-based system is then used (inference) on a rule-based system. Given a new query to run on the rule-based system, Toulouse proposes a plan with a specific join ordering, which in turn executes on the rule-based system bypassing the system's optimizer.

4. Evaluation

4.1. Setup and implementation details

In our experiments, we used the following setup.

Hardware. Several rule-based systems are used either as server databases deployed on server machines or embedded databases running on lightweight configurations. We investigated both scenarios using two setups: (S1) 20-core Intel(R) Xeon(R) CPU E5-2630 v4 2.20GHz (3.1GHz max), 142GB memory; and (S2) 4-core AMD Ryzen 5 2500U 2GHz (3.6GHz max), 8GB memory.

Software. We used Pytorch (1.13.0) with Pytorch-Geometric (2.1.0) for the GNNs, GATv2 for the graph layer (although we also successfully tried SageConv and GINConv), Python for SQL query rewriting, PostgreSQL (13.3), and MonetDB 11.41.11.

Data and preparation. We tested two use case scenarios: (C1) training and inference using two workloads on the same schema, and (C2) training on a multiplicity of schemas and workloads, and inference on a new schema and a new workload. C1 resembles a common use case, which is typically seen in the experimental analysis of other learning optimization approaches. C2 represents a more challenging, but clearly more interesting case scenario.

Use case C1. Data generation, training, and testing was performed on PostgreSQL on S1, which was tuned for performance. For the graph layer, 20 epochs were enough, with early stopping to the best approximate result. We used two workloads: (W1) The Join Order Benchmark (JOB)[16] queries and data, which is based on the IMDB dataset and comprises 113 queries containing a varying number of joins, ranging from joining 5 to 17 tables; and (W2) We created a workload comprising 1200 queries generated by randomizing various parameters and predicates of the JOB queries, changing effectively several properties such as operator selectivity, size of the intermediate results, etc. For the evaluation, we used W2 and did not consider any of the queries used in the training phase.

Use case C2. We employed a variety of relational datasets and workloads as described in the DBGen benchmark [21][1]. This collection comprises 21 datasets, including IMDB and IMDB_full that are the base of the JOB benchmark. Initially, we trained the model on PostgreSQL as described in (C1) on all datasets except the two IMDB ones. Then, we tested the model on both PostgreSQL and MonetDB using the JOB workload (IMDB dataset). In this case, the JOB workload is unknown to the model

[1]The DBGen benchmark can be found here: https://github.com/DataManagementLab/zero-shot-cost-estimation

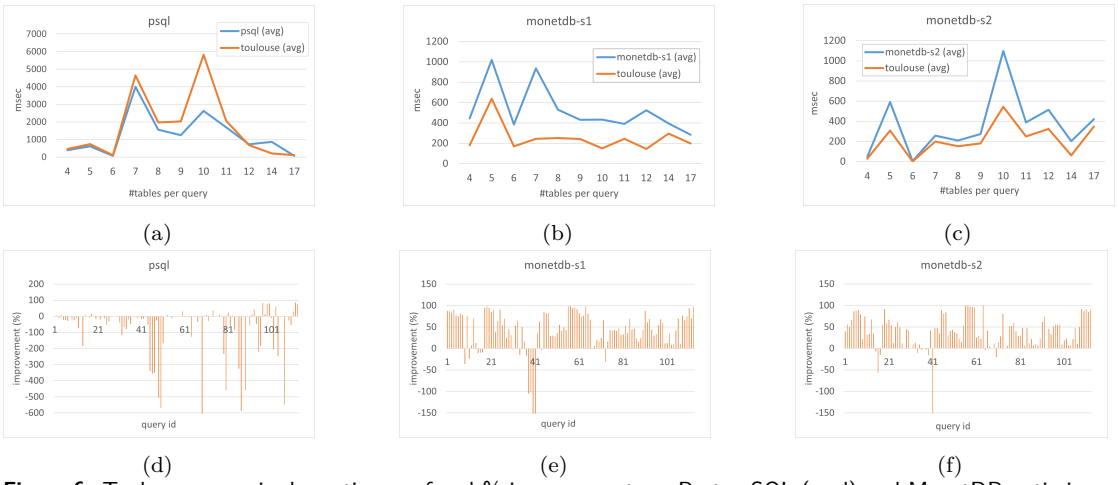


Figure 6: Toulouse supervised: runtime perf and % improvement vs. PostgreSQL (psql) and MonetDB optimizers

trained –an experiment designed to test whether our approach could potentially serve as a schema and workload agnostic solution.

An illustration of the scenarios tested is as follows:

	workload	database	hardware
C1: supervised	□	□ ■	□ ■
C2: RL	■	□ ■	□ ■

□: same with training ■: different than training

4.2. Model creation and evaluation

Evaluating the supervised approach. First, we present the results of our experimental analysis for Toulouse's supervised approach on the C1 scenario.

For training, we used 200,000 query graphs created by searching recursively in a subset of the JOB queries for valid join combination that each creates a different query graph, hence forming scenarios that require join decisions. Every potential join has an approximate best query runtime –should the join happens– which can be used directly for regression. In our case, we set the minimum join value as 1 and everything else as 0 for binary classification. Next, we fed this training data to our GNN structure, using in our implementation the Adam optimizer with Binary Cross Entropy loss.

Evaluation on PostgreSQL (psql). We tested the queries with explicit join orders produced by Toulouse vs. queries optimized by the psql optimizer using the W2 workload. We ran each query 6 times and report the average results. Figure 6a shows the avg runtime performance of queries grouped by the *#tables* they involve. Figure 6d shows the % of improvement per query pattern. The results show that although Toulouse has learnt the trend of the psql optimizer's decisions, still it does not outperform psql. This is normal, as we did not try to beat psql by fine-tuning the model and/or using additional features, thus avoiding the risk of over-fitting our model.

Evaluation on MonetDB (mdb). Toulouse has learnt efficient join paths from the psql optimizer, so next, we investigated whether it could leverage those on MonetDB and on two setups, S1 (same as psql) and S2 (new). Figures 6b and 6e show the avg runtime performance with varying *#joins* and the % of improvement per query pattern, respectively, for the S1 setup (Figures 6c and 6f show the same analysis on S2). Note that the more challenging queries involve 7-10 tables. These run better on the S1 setup that has more memory and cpus. The less resource demanding queries involving 4-7 tables run better on the S2 setup, whose resources suffice for these queries, as it has more modern and faster cpus. The plans specified by Toulouse (with a few exceptions) improve the mdb performance by an average 42% on S1 and 36% on S2. Hence, join path knowledge seems to be effectively transferable. Looking deeper into the mdb plans, we realized that the mdb optimizer does not make optimal decisions for complex queries with many joins. This is an interesting finding towards future research in query optimization: an ML based technique, trained on a different engine and setup, can effectively complement a traditional optimizer. The alternative, improving the optimizer codebase, would potentially be orders of magnitude more challenging to implement.

The supervised approach however does not perform equally well on the use case C2 (results not shown here), which also adds a different workload to the equation. This motivated us to examine a more complex architecture, with a richer feature set, and a different design to cope with the challenge of optimizing a previously unseen workload.

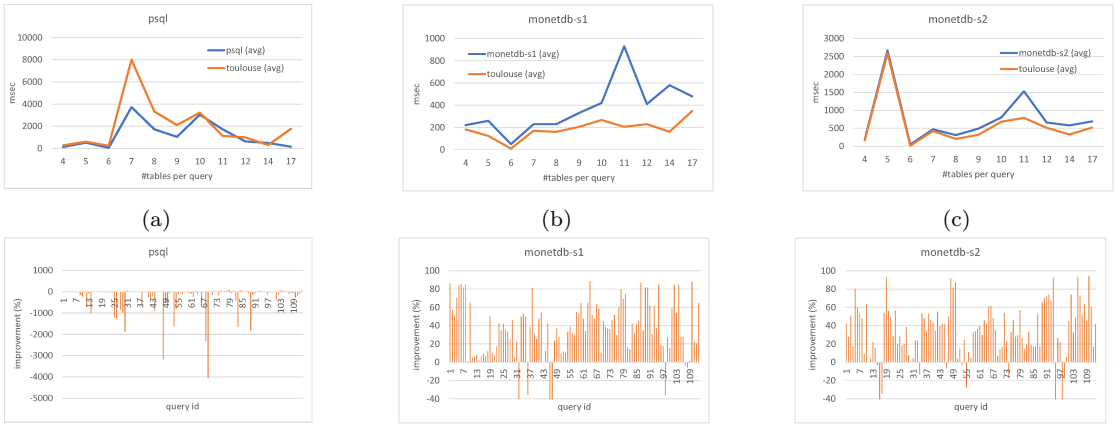


Figure 7: Toulouse RL: runtime perf and % improvement vs. PostgreSQL (psql) and MonetDB optimizers

Evaluating the reinforcement learning approach. Motivated by the positive results of re-using a cost-based learning model on a rule-based engine, next we increased the challenge to work with an unknown schema and workload. Hence, our second line of experiments focuses on the effectiveness of our reinforcement learning approach. As this approach performs similarly to the supervised one on the C1 scenario, we omit the presentation of results for this case and focus on the more complex, C2 scenario.

For starters, we trained the model on psql using a similar process as before. In more detail, we altered the query generation code of DBGen [21] to output implicit joins and used 18 different workloads from different database schemas, leaving out the two IMDB schemas completely during training and only performed inference on these two schemas. This way, the learning happens in a schema-agnostic manner and allows to test whether it could be transferable to different databases. In each epoch we collect 6400 steps and train in batches of 64 for 3 mini-epochs.

Evaluation on PostgreSQL (psql). We tested the queries produced by Toulouse vs. queries optimized by the psql optimizer using the 113 queries of the JOB benchmark. We ran each query 6 times and report the average results. Figure 7a shows the avg runtime performance of queries grouped by the *#tables* they involve. Figure 7d shows the % of improvement per query id. The results show that Toulouse manages to follow the general trend of the psql optimizer's decisions. Observe, that for some queries, Toulouse RL performs worst than the supervised approach (compare Figures 7d and 6d). This is explained by the fact that in this scenario Toulouse deals with queries and schema that has never seen before. Still, it seems to be able to apply effective optimization strategies that has learnt from the different workloads used in the training phase.

Evaluation on MonetDB. Next, we evaluate how Toulouse could cope with an unknown workload on two different hardware setups S1 (same as psql) and S2 (new). Figures 7b and 7e show the avg runtime performance with varying *#joins* and the % of improvement per query id, respectively, for the S1 setup (Figures 7c and 7f show the same analysis on S2). Notably, Toulouse manages to optimize almost all queries by a significant fraction despite the fact that the inference is performed with a new workload, on a new database, and on a new hardware setup (S2). Figure 8 shows the average improvement in MonetDB, on both hardware setups, for a varying number of tables (i.e., *#joins*) per query. On average, Toulouse improves query performance in MonetDB by 45.68% on S1 (server machine) and 24.4% on S2 (lower end machine).

Discussion. We measured the robustness of Toulouse's plans. The standard deviation of query performance is rather low for the entire workload, with the slight deviations effectively caused by typical runtime variability. Looking at the plans, we verified that Toulouse consistently produced the same join paths for each query having all the other parameters of the experiment unchanged.

Caveats. Although PostgreSQL is the predominant system used in the majority of related work both for model training and as a baseline, clearly there are more advanced optimizers in other database systems. We believe that using a more advanced optimizer to train our models would only make our approach perform better, but we leave this as a future work. The JOB benchmark (i.e., IMDB schema) has been used extensively in the literature as an analytics workload comprising queries with varying complexity of join chains. We find this as

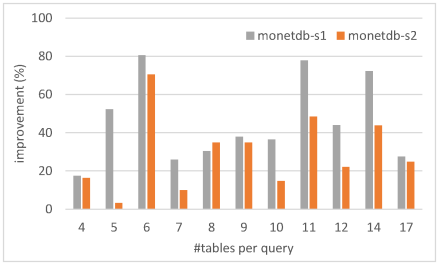


Figure 8: Toulouse RL: % improvement in MonetDB running on two hw setups S1 (server) and S2 (low end)

a useful first step for our analysis, still, we plan to investigate more real-world workloads in the continuation of our work. Finally, our results reported here are based on our experimentation with MonetDB; an excellent open source, analytics database system that comes with one of the most popular and successful rule-based optimizers. However, we plan to extend our investigation to other rule-based engines as well, in an effort to generalize our findings to rule-based systems beyond MonetDB.

5. Conclusions

We presented an investigation on whether learning could improve query optimization and in particular, the join ordering, in rule-based systems. Our initial attempt to use rules and optimization strategies as features was not very successful. Still, as we show in this paper, applying a model trained on a cost-based optimizer to a rule-based system for known and unknown workloads alike shows potential and deserves additional research. Finally, our work shows that ML techniques could be a reasonable supplement to potential shortcomings of traditional optimizers without requiring changing their codebase.

Acknowledgements. This work has been partially supported by EU's Horizon 2020 projects INODE and HBP-SGA3 with grant agreement numbers 863410 and 945539, respectively.

References

[1] L. M. Haas, J. C. Freytag, G. M. Lohman, H. Pirahesh, Extensible query processing in starburst, in: SIGMOD, 1989.

[2] G. Graefe, W. J. McKenna, The volcano optimizer generator: Extensibility and efficient search, in: ICDE, 1993.

[3] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, M. L. Kersten, MonetDB: Two decades of research in column-oriented database architectures, IEEE Data Eng. Bull. 35 (2012) 40–45.

[4] MonetDB Docs, Optimizer pipelines, 2022. https://www.monetdb.org/documentation-Sep2022/admin-guide/performance-tips/optimizer-pipelines/.

[5] D. Tsesmelis, A. Simitsis, Database optimizers in the era of learning, in: ICDE, 2022.

[6] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, V. Nathan, Sagedb: A learned database system, in: CIDR, 2019.

[7] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, I. Stoica, Learning to optimize join queries with deep reinforcement learning, 2018. CoRR:abs/1808.03196.

[8] R. Marcus, O. Papaemmanouil, Deep reinforcement learning for join order enumeration, in: aiDM@SIGMOD, 2018.

[9] X. Yu, G. Li, C. Chai, N. Tang, Reinforcement learning with tree-lstm for join order selection, in: ICDE, 2020.

[10] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, J. Antonakakis, Skinnerdb: Regret-bounded query evaluation via reinforcement learning, in: SIGMOD, 2019.

[11] J. Zhang, Alphajoin: Join order selection à la alphago, in: PVLDB-PhD, 2020.

[12] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, N. Tatbul, Neo: A learned query optimizer, in: PVLDB, 2019.

[13] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, T. Kraska, Bao: Making learned query optimization practical, in: SIGMOD, 2021.

[14] A. Jindal, S. Qiao, R. Sen, H. Patel, Microlearner: A fine-grained learning optimizer for big data workloads at microsoft, in: ICDE, 2021.

[15] MonetDB Docs, Mal optimizers, 2022. https://www.monetdb.org/documentation-Sep2022/dev-guide/monetdb-internals/mal-optimizers/.

[16] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, T. Neumann, How good are query optimizers, really?, Proc. VLDB Endow. 9 (2015) 204–215.

[17] B. Hilprecht, C. Binnig, One model to rule them all: Towards zero-shot learning for databases, in: CIDR, 2022.

[18] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, M. Zhang, CherryPick: Adaptively unearthing the best cloud configurations for big data analytics, in: USENIX NSDI, 2017.

[19] F. Harary, R. Z. Norman, Some properties of line digraphs, Rendiconti del Circolo Matematico di Palermo 9 (1960) 161—-168.

[20] V. P. Dwivedi, A. T. Luu, T. Laurent, Y. Bengio, X. Bresson, Graph neural networks with learnable structural and positional representations, in: ICLR, OpenReview.net, 2022.

[21] B. Hilprecht, C. Binnig, Zero-shot cost models for out-of-the-box learned cost prediction, Proc. VLDB Endow. 15 (2022) 2361–2374.