École Polytechnique Fédérale de Lausanne

Learning to generate query plans through Deep Reinforcement Learning

by Murat Topak

Semester Project Report

Approved by the Examining Committee:

Prof. Anastasia Ailamaki
Project Advisor

Dr. Srinivas Karthik Venkatesh
Project Supervisor

EPFL IC IINFCOM DIAS
BC 222 (Bâtiment BC)
STATION 14
CH-1015 Lausanne

January 10, 2020

Abstract

To find the optimal plan for a sequence of joins, it is normally avoided to go through each possibility since the search space is very large. Instead, the query optimizers use cost models and heuristics to narrow down the search. However, with the emerging applications of machine learning in the other fields, the database community also wondered whether it was possible to replace the human engineered optimizers with the data-driven trained ones. In our work, we follow the recent advancements in this field, and give a proof-of-concept deep reinforcement learning based query optimizer that learns the Postgres optimizer with an approximation factor. We demonstrate how such a system is designed and implemented and share our results and ideas for the future.

Contents

Chapter 1

Introduction

Two key components of an SQL database system are the execution engine and the query optimizer. The former implements a set of physical operators such as sequential scan, index scan, nested loop, and merge join. An operator takes one or more input data stream and produces an output stream. The latter then, knowing the available operators, is responsible for generating an efficient query plan for the given SQL query. The plan generated by the optimizer becomes the input to the execution engine. The task of an optimizer is nontrivial because there are many plans yielding the correct results, yet fewer are efficient. The considered search space is large because many join orderings are possible for a given query, and for each of them several access and join methods at the operator level are made available.

The query optimizers are not perfect after all, since they are built on top of heuristics and cost models. In the traditional architecture, cardinality estimates are used as the principal input to the cost models. These estimates are computed using simplifying assumptions on the data such as uniformity and independence, but in reality these assumptions are frequently wrong [4]. Thus, there have been many attempts to improve the query optimization in the past. Leo [9], for example, provides a mechanism to adjust its cardinality estimates and incorrect statistics with a feedback loop. More recent work in the field is the applications of deep reinforcement learning to the problem of query optimization. ReJOIN [6] is one such work to show join orderings can be learned with DRL. Their results match or outperform PostgreSQL optimizer in terms of join enumeration. In DQ [3], the authors proposed a general way of featurizing the queries so that a a deep-Q-network could be trained to find an efficient plan. In addition to join orderings, join operators were also studied in that work so as to highlight the extensibility of their featurization process. There is also Neo [5], the first to show that an entire query optimizer could be learned. In their work, not only they covered the indexes besides the join enumeration and operators, but also they introduced a variety of ideas like tree convolution and the use of word embeddings to replace the histograms.

In our work, we intend to follow the recent advancements mentioned in the above paragraph. This means design and implementation of one such system like DQ and Neo from the scratch, evaluating its performance, and considering possible extensibilities for the future. We will first provide a formal description of the query optimization problem. Hopefully, this will make our motivations clear about using deep reinforcement learning. Later, we will touch on the design and implementation issues. This includes the generation of training data, the neural network training itself, and the plan searching phase, followed by hinting. The evaluation will then be based upon how successful the training is and how the system generalizes to the test data.

Chapter 2

Background

Let us first have a set of relations and a sample SQL query to be used in the rest of the paper:

Sailors(*sid*, *sname*, *rating*), Boats(*bid*, *bname*, *color*), Reserves(*sid*,*bid*,*day*)

```
SELECT *
FROM Sailors as S, Boats as B, Reserves AS R
WHERE S.sid = R.sid
AND B.bid =  R.bid
AND S.rating > 5
```

The query above can be executed in many different ways depending on the join orderings and physical operator selections. Examples are $(S \bowtie_L R) \bowtie_L B$, and $S \bowtie_H (R \bowtie_H B)$. The subscript on the join operator denotes whether it is a hash join or a loop join. We omit different types of scans from the notation not to uglify the text. However, note that all the relations could use the the scan operators made available to them such as sequential scan, bitmap scan, and index scan.

The nature of the query optimization problem makes us think about the Markov Decision Processes. AN MDP is a 4-tuple $(\mathbf{S}, \mathbf{A}, \mathbf{P_a}, \mathbf{R_a})$ with the sets of states, actions, transition probabilities to the other states and rewards. Basically, a valid set of actions is available for each state and different amounts of rewards are given for different state,action pairs. Taking an action makes the agent transition into a new state with some probability and the process continues until a terminal state is arrived. Since the state transitions of MDPs satisfy the Markov property, the next state upon taking an action is completely dependent on the current state and the taken action and nothing else. The objective of an MDP is to find a decision policy $\pi : \mathbf{S} \mapsto \mathbf{A}$, a function mapping states to the actions, with the maximum expected reward:

$$\begin{aligned} \underset{\pi}{\text{argmax}} \quad & \mathbf{E}[\sum_{t=0}^{T-1} R(s_t, a_t) \\ \text{subject to} \quad & s_{t+1} = P(s_t, a_t), a_t = \pi(s_t) \end{aligned}$$

Trying to fit our problem into an MDP, now let us borrow the definition of a query graph, and the join optimization problem from [3].

Definition 1. A query graph G is an undirected graph, where each relation R is a vertex and each join predicate ρ defines an edge between vertices. Let κ_G denote the number of connected components of G.

Suppose we have the query graph $G = (V, E)$. Applying a join $c = (v_i, v_j)$ to G defines a new graph with vertices v_i, v_j removed, and a vertex $v_i + v_j$ added to V. The edges of the new vertex is set to be the union of the edges of the v_i, v_j. Each join reduces the number of vertices by one, and each plan can be described by a sequence of joins $c_1 \circ c_2 \circ ... \circ c_T$ until $|V| = \kappa_G$. This modelling of the joins is great in that it can be extended to cover operator selections, e.g. $c = (v_i, v_j, \text{LoopJoin})$ would mean relations are joined with a nested loop, or $c = (v_i^{\text{Seq}}, v_j^{\text{Bmap}}, \text{HashJoin})$, would even further specify the scan methods sequential scan and bitmap scan for the participating relations. Now that the notation is ready, the join optimization problem could be defined as the following:

Problem 1. Let G define a query graph and J define a cost model. Find a sequence $c_1 \circ c_2 ... \circ c_T$ terminating in $|V| = \kappa_G$ such that

$$\begin{aligned} \min_{c_1,\ldots,c_T} \quad & \sum_{i=1}^{T} J(c_i) \\ \text{subject to} \quad & G_{i+1} = c(G_i) \end{aligned}$$

It is clear now this problem defines an MDP, with the query graph G as state, c as an action, vertex merging process as the transition function that maps states to states, and the cost model J as the reward. Note that the greedy solution to the optimization problem is sub-optimal. The greedy algorithm is the one that picks the action with the lowest join cost every time. The problem with this method is that it does not take the future into account. What really matters is not actually the local costs but the final cost of the whole plan. The optimal algorithm to the problem would consider the effects of its actions on the long time reward instead. For this purpose, suppose now there is the function $Q(G, c)$ telling us the long time reward of an action c in some state G. Existence of such a function would solve the optimization problem since at each step taking the action with the lowest Q-value would lead to the optimal plan.

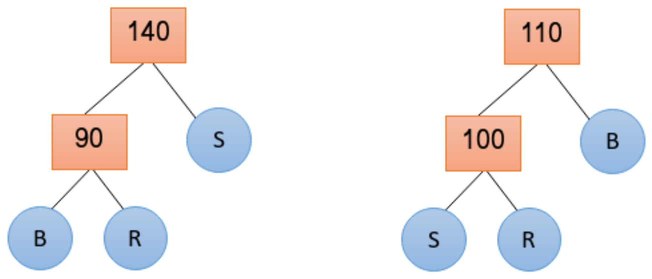

Figure 2.1: Two sample plan trees for the same query with their cumulative costs.

The Figure 2.1 is illustrating the problem with the greedy algorithm on the local costs. In the rectangles, the cumulative costs of the join operations are written. Without a consideration for the future and taking the minimum at each step, the left tree would be preferred over the right since 90 < 100. However, it turns out in the end the chosen plan is not the optimal one. Instead, the Q values must be substituted in for each node. This is visualized in Figure 2.2

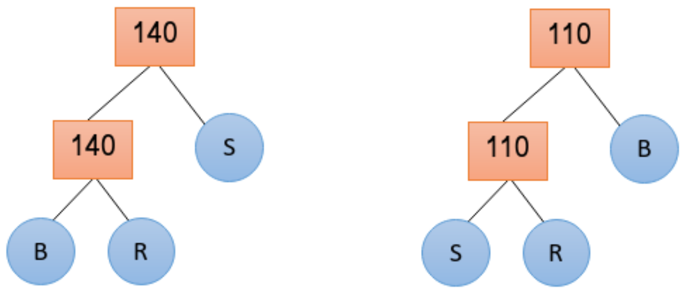

Figure 2.2: The same plan trees with their Q values in each node.

The Q values denote the long time reward of an action. Now running a greedy algorithm picks out the optimal plan as the node values reflect about the future. Indeed, there is no Q available so we have to approximate it. Use of neural networks in learning the Q-value is called deep reinforcement learning (RL). It has proved its use in robotics, video games, finance and health care [8]. Here in this work, we will use a deep network too.

Chapter 3

Design

It should be clear by now what the problem is and how we plan approaching it. We want to come up with a system so that for a given query it selects the optimal plan with respect to the join orderings and the physical operators. We decided to use a neural network to approximate the Q function. With such a roadplan, the design of the system is illustrated in Figure 3.1. The design starts with a sample workload, and an expert optimizer such as Postgres. The expert optimizer is used to extract the query execution plans (QEPs) for the given workload. For each QEP, there is a reward associated with it – the final cost of the whole tree suggested by the expert.

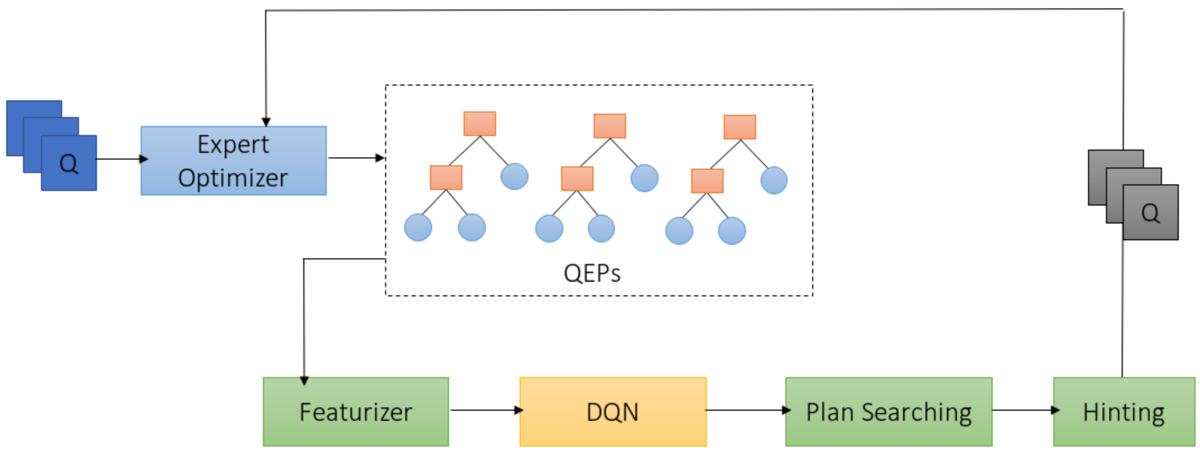

Figure 3.1: General view of the system.

The featurizer module takes the tree structured QEPs as its principal input. Here, any other related information about the query can also be provided with the plan tree. Examples are the existence of predicates and histograms of the database columns. The output of the featurizer are the regressors. These are used by the DQN module to train a deep network for approximating the Q function. Upon termination of the training, plans are searched in a bottom up fashion guided by the trained network. The hinting module takes care of forcing the target system follow the generated plan. At this point, there is a set of hinted queries for which we do not know about their performances yet. These are evaluated at the optimizer so as to see how well they are performing. The set of QEPs is updated with these new plan trees, and another iteration begins with the updated workload. This way the system learns by making mistakes and gets better by adjusting its approximator (DQN) accordingly at each iteration.

The design of the featurizer allows an efficient generation of the training data. From one sample QEP, we can extract many other QEPs as illustrated in Figure 3.2. This is because all the trees at the right hand side have the same Q value, which is equivalent to the cost of the final tree at the left.

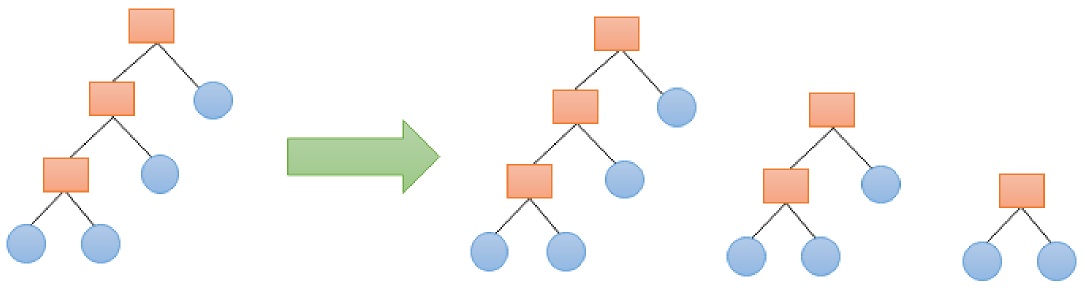

Figure 3.2: From one tree, many others can be extracted for the training.

Perhaps, for a better understanding, we can write down the objective of the deep network. Suppose full plan trees P_f along with their rewards $R(P_f)$ exist. Let us call the set of all full trees E. The tree at the left hand side of the above figure is an example to P_f. The model M is trained to approximate for all $P_i \subseteq P_f$,

$$M(P_i) \approx \min\{R(P_f) : P_i \subseteq P_f \wedge P_f \in E\}$$

Chapter 4

Implementation

First, the expert optimizer is a third party software. In the implementation Postgres 12.0 is used to retrieve QEPs. The EXPLAIN command of Postgres reveals the expert plan trees and may output in JSON as well. Indeed, not everything is important in the retrieved plan tree as it contains a lot of information. Thus, we convert the JSON to an intermediate readable form to be fed into the featurizer.

The featurizer implementation is critical as it prepares the input for the DQN. Here we follow a combination of featurization ideas proposed in [3] and [5]. Figure 4.1 visualizes the encoding of the tree nodes.

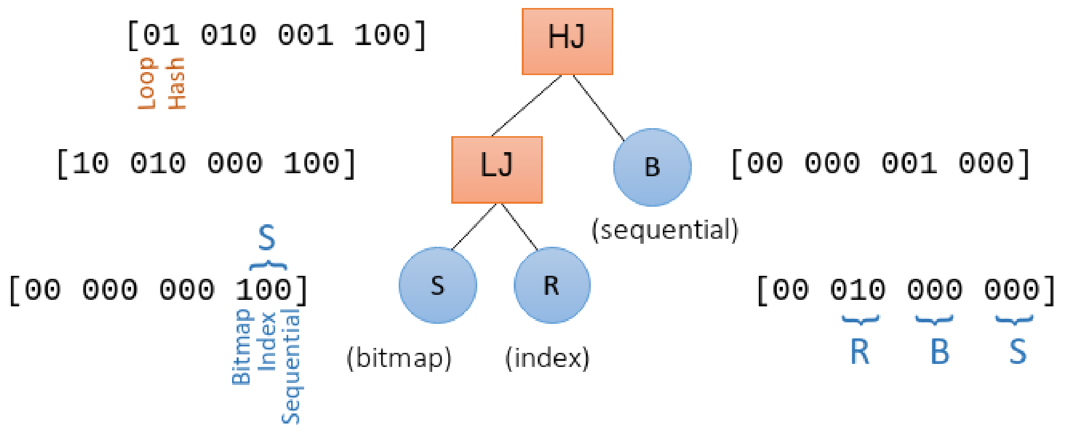

Figure 4.1: Node encodings of the trees.

As a first step, the features must contain enough information about the join types and scan types. In the figure, a possible plan tree is given for our sample query from Chapter 1. Let us have $|JT|$ different join operators, and $|ST|$ different scan operators available for each operation. Further, assume there are $|Rels|$ relations in the database in total. For each node of the tree, we construct a vector of size $|JT| + |Rels| \times |ST|$. In the example, $|JT|, |Rels|, |ST|$ are 2,3,3 respectively. Thus, the first two values of the vectors denote the join types. After that we have blocks of zeros and ones separated by spaces. Each block here denotes a relation from the database. In the example, the order selected for the blocks is R, B, S as can be seen in the figure. Within each block we specify scan operator used in retrieving the particular table. Again, from the example, S has its bitmap field set while R has its index field set. When joining two nodes, the resulting encoding is the union of the scan blocks of the node's children prepended by a one hot vector of length $|JT|$. For instance, the loop join node in the figure unions its children vector's last 9 numbers and prepends it with 10. Note that in the example it happened to be the case that all the relations of the database participated in the query. For this reason, you may have thought that the encoding is variable length and only includes the participating relations. This is not true. The encoding is fixed-sized and considers all the relations of the total workload/database. Another plan tree may only involve the relations S, R but we would still have the B in the encoding.

Having encoded the plan tree into above form, the last step is to flatten the tree to make it ready for the neural network. Recall we explained in the design chapter a full tree contains more than 1 feature thanks to its subtrees. Here we make use of this idea. For each join node of the full tree, the node's vector is concatenated with its left child's and right child's vectors. For example, we can extract two features from the tree in Figure 4.1. One is for the hash join and the other is for the loop join. The concatenations would give:

1. [01 010 001 100] + [10 010 000 100] + [00 000 001 000]
2. [10 010 000 100] + [00 000 000 100] + [00 010 000 000]

Fundamentally, this encoding is descriptive. We can identify between many different trees. For example, if this was a right deep tree instead of a left deep one, the encodings could have made the distinction. Again, if the joining order were to be B, R, S instead of S, R, B in Figure 4.1, the resulting vectors would be different than what we have now. However, to have even more enriched features, we append two more vectors to the each concatenated vector above. One is the participation vector, and the other is the predicate vector. The first is of length $|Rels|$. It is a zero one vector indicating which tables from the database participate in the joins clauses of the query. For instance, in the example the participation vector should be 111 since all three relations of the database happens to be joined. For another query where we join only R, B, it would be 110 for the database relations ordered as R, B, S. The second vector is as long as the number of columns in the database. Referring to our example database, it would be of size 9 since there are 3 relations each with 3 attributes. In the example query, we have a single predicate after the joins. This is identified by putting a 1 at the index of that attribute in the predicate vector.

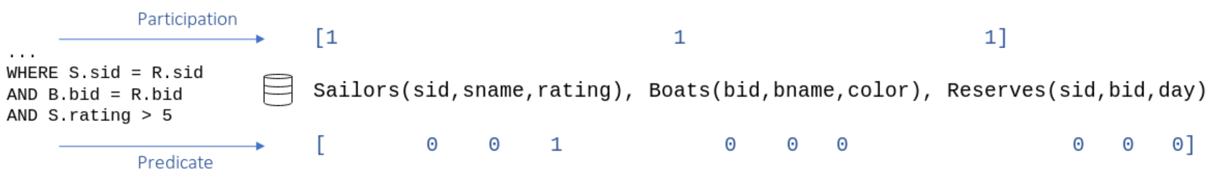

Figure 4.2: Construction of participation and predicate vectors.

Figure 4.2 illustrates the construction of participation and predicate vectors for a given query on a database. With these additions, two features we extract from the tree in Figure 4.1 becomes

1. [01 010 001 100] + [10 010 000 100] + [00 000 001 000] + [111] + [001 000 000]
2. [10 010 000 100] + [00 000 000 100] + [00 010 000 000] + [111] + [001 000 000]

The motivation to add these vectors is purely based on the fact that they are parameters of the join optimization problem. After all, a real world optimizer makes use of the predicates, histograms, the total number of joins, the name of the joined tables... Thus we aim to cover most of them too.

After we have such vectors, we use PyTorch to train a multi layer network with Leaky ReLU activations for the hidden layers and no activation for the output. The SGD optimizer is utilized throughout the training. Because the rewards in our case are the final costs, they tend to vary from as little as 500 to as much as 2 millions. When this is the case, it is problematic for both the SGD and the network itself. The previous work for Deep RLs for example limited their rewards to (-1,1) when teaching an agent how to play Atari [8]. Here, we use the log scaled cost for the final output of the network. This helps for the training.

The plan searcher implementation considers the left deep, right deep, and bushy trees. That is, we have the initial children which are the valid joining of any 2 tables. Note that the join types and scan types determine the number of children along with the number of tables. Then, as the plan is built up, the plan searcher may select a single table to construct a left/right deep plan with the partial plan from the previous step, or it may select 2 tables, join them first with each other, and join their results with the partial tree afterwards to construct a bushy tree for the next step. This procedure continues until there is no more table left to join to the partial tree being constructed.

Finally, after the plan searching phase, the hinting takes place. This is for the query to follow our generated plan. We use pg_hint_plan plugin [1] of Postgres to hint queries. The plan searcher specifices a complete plan but it is up to the hinting module that forces the Postgres to apply the given plan.

Chapter 5

Evaluation

We use the Join Order Benchmark (JOB)[2] dataset to evaluate the performance of the system. It is a set of 113 queries over Internet Movie Database (IMDB) specifically designed to test query optimizers. We randomly sampled 90 of them for the training and left the other 23 for testing. Let us present the training history first in Figure 5.1. The first subfigure demonstrates a classical

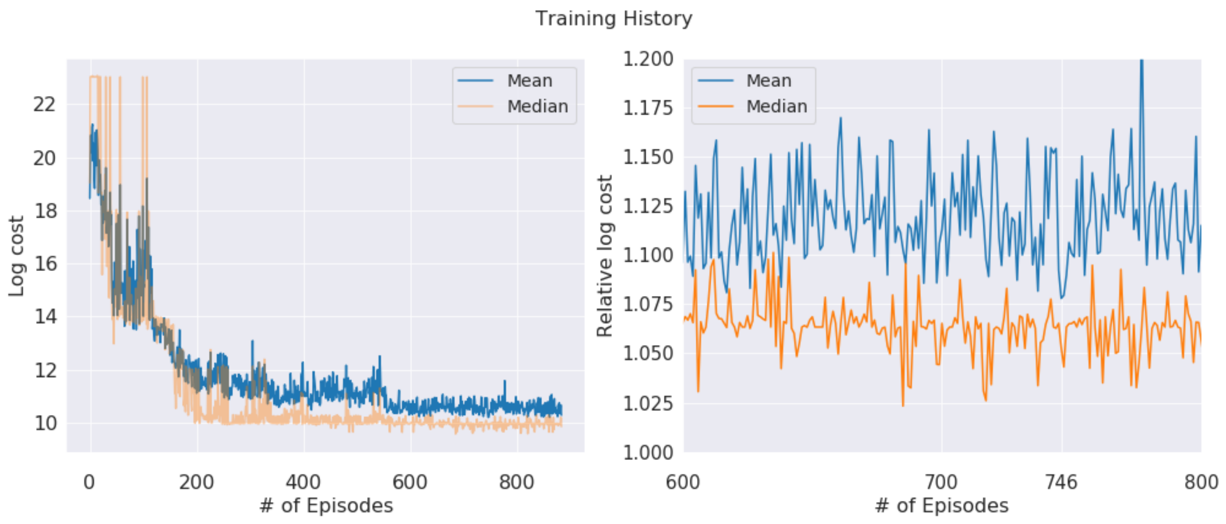

Figure 5.1: The training curves of the deep network.

reinforcement learning curve for our training. In the figure, the x-axis is the number of episodes, and the mean/median costs of the plans generated by our system for 90 training queries is indicated in the y-axis in log scale. Median line is included to show the mean can be deceptive in that it is affected by the outliers. Perhaps a closer look with a comparison to Postgres is more explaining. That is the second subfigure in Figure 5.1. In that, the episodes are limited to cover from 600 to 800, noting the best mean happened at 746. The relative log cost states the log scaled costs of our plans over the log scaled costs of the Postgres. Thus, the lower the better.

For example, the value 1.025 reveals the generated plans follow a function of $y = x^{1.025}$ where x is the Postgres cost value (not log scaled). This is not bad since that function is almost linear. You may see the scatterplot of 90 training queries below. The scatterplot starts almost perfectly

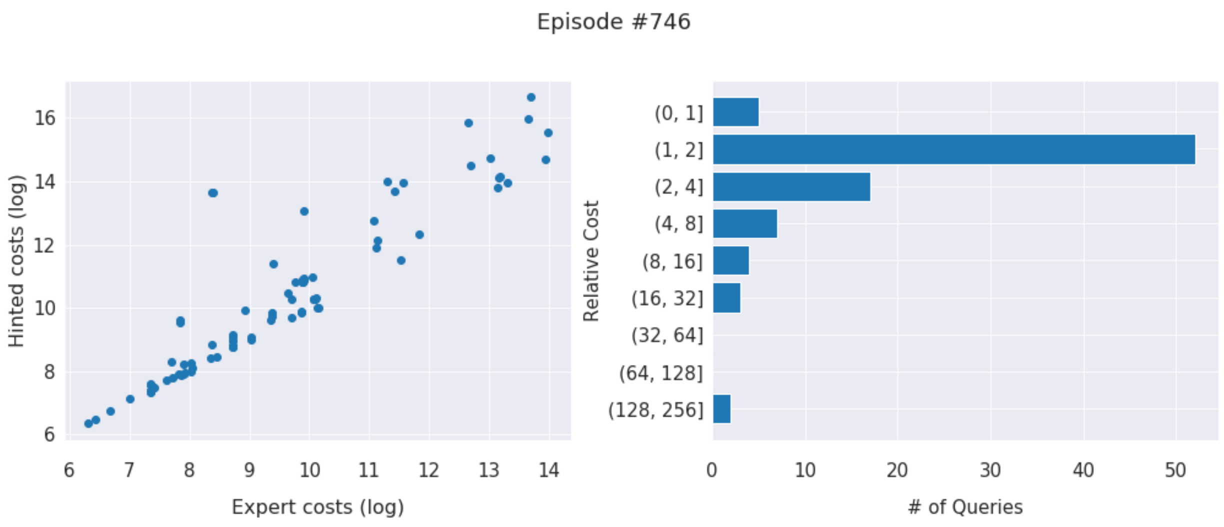

Figure 5.2: Comparison of 90 training queries between our system and Postgres.

linear but then as the expert cost increases we see deviations from the original. Next to this subfigure, a histogram of the relative costs (not log scaled) of our plans for the training queries is given. This demonstrates the approximation's quality in terms of a multiplicative constant instead of an exponential one from the previous figure. Thus now it means for example most of the generated plans follow kx where k is between 1 and 2, and x being the original cost from Postgres. To investigate the outliers, let us take the relative cost values and put them in a scatter plot against the number of join clauses.

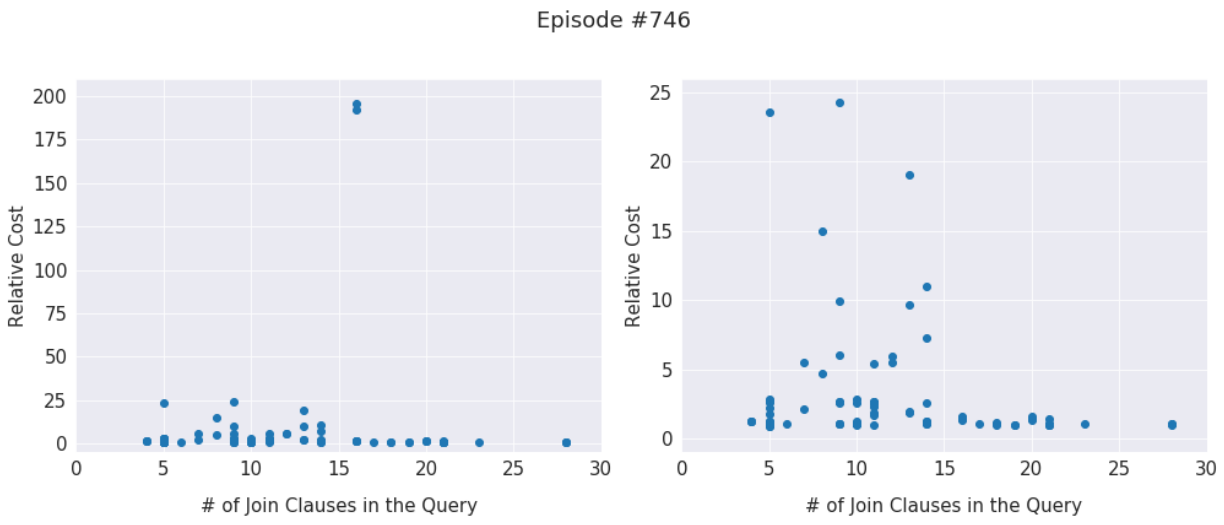

Figure 5.3: Performance of 90 training queries as a function of the number of join clauses.

The figure 5.3 shows shows all 90 queries on a scatterplot against the number of join clauses in their queries. The subfigure next to that is the same but zooomed in to ignore the outliers. First of all, we do not see any correlation between the number of joins and the performance but the points look more spread on the left side. However, this actually improves as the model keeps training. As we said before, reinforcement learning is noisy and taking a single wrong step may screw up the whole plan easily. Dealing with these outliers require either more training, or a stronger featurization. We will talk about the limitations of our featurization in the conclusion part. We end this section by saying our best model achieved to learn, through a deep network model, to build plans compared to Postgres, 1.4x worse in median on the training set, and 3x worse in median on the test set.

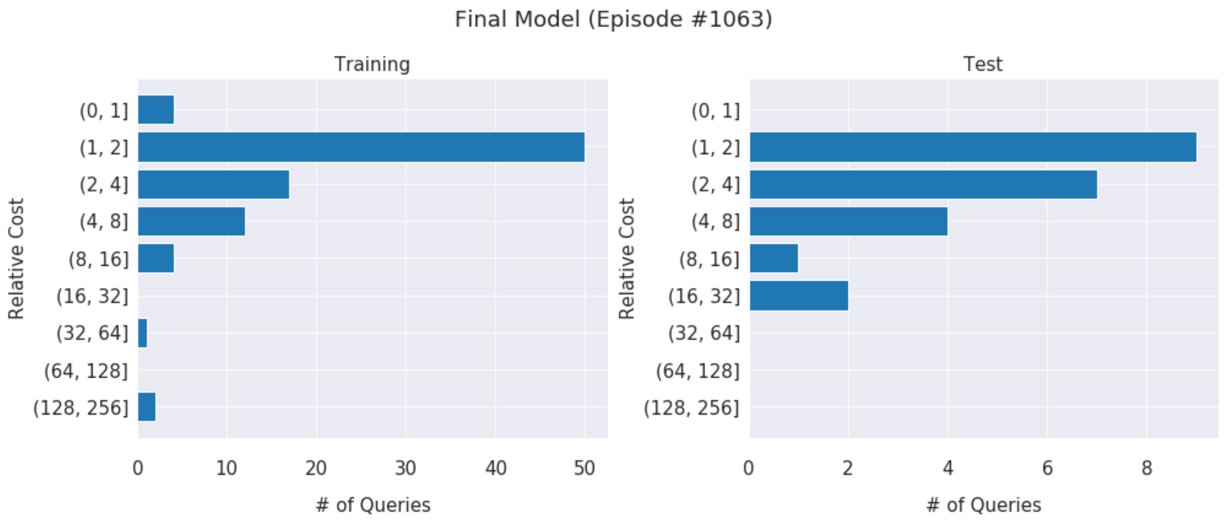

Figure 5.4: Final relative cost values on the training and test set.

Chapter 6

Related Work

ReJOIN [6] is a deep RL solution to the join ordering problem. They report 20 percent improvements on the Postgres optimizer. However, dealing only with the join orderings is not as complex as deciding also the scan types and join types. The same team from ReJOIN eventually builds Neo, the first end-to-end query optimizer as they claim. They introduce several ideas such as tree convolution and the use of row vectors to have at least 25 percent improvement (in terms of latency) compared to Postgres. However, we have to note they are working with the real feedback from the target executor, i.e. the latency value. Working with latencies has been proved to be useful in fine tuning such systems. It is well-noted by the same team in their recent paper that starting with the costs is fine because this is just like the "training wheels of a bicycle" [7]. Then they argue a conversion from the costs to latencies should be made eventually to make the system work with the real feedback. DQ [3] is another system using Deep RL to come up with plans including join & scan decisions, plus the join orders. They start with the costs as well and fine tune their system with the latencies. They report improvement on the Postgres optimizer for specific JOB queries. In DQ, histograms proved themselves useful and the exploration/exploitation trade off was also controlled with a parameter to select random actions instead of the best one during the plan searching to encourage the exploration.

Chapter 7

Conclusion

In this work, we demonstrated a query optimizer can be learned to construct plans for the given queries. We showed the use of deep networks comes in useful for learning the Q values for the plans and presented our results in comparison to Postgres optimizer. We may have not beaten the recent systems but this is fine given the improvability of our featurization. For the future, it makes sense to extend the featurization to cover more information about the queries. For example, we are only using children in the featurization but using grandchildren also is an idea one may consider. Likewise, we do not make use of histograms but it was noted explicitly in [3] histograms decrease the final loss of the network significantly. The last idea is to cover indexes also, i.e. we may create a binary vector to state which indexes on which columns have been used in the plan. In our work we only hinted the index type like the sequential scan, index scan, bitmap scan, but not the particular name of the index along with this type. As these are more complex systems than ours, our intention was to show that a basic featurization is more than enough to learn the Postgres optimizer.

Bibliography

[1] URL: `https://pghintplan.osdn.jp/pg_hint_plan.html`.

[2] URL: `https://github.com/gregrahn/join-order-benchmark`.

[3] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. "Learning to Optimize Join Queries With Deep Reinforcement Learning". In: *arXiv e-prints*, arXiv:1808.03196 (Aug. 2018), arXiv:1808.03196. arXiv: `1808.03196 [cs.DB]`.

[4] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. "How Good Are Query Optimizers, Really?" In: *Proc. VLDB Endow.* 9.3 (Nov. 2015), pp. 204–215. ISSN: 2150-8097. DOI: `10.14778/2850583.2850594`. URL: `https://doi.org/10.14778/2850583.2850594`.

[5] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. "Neo: A Learned Query Optimizer". In: *Proc. VLDB Endow.* 12.11 (July 2019), pp. 1705–1718. ISSN: 2150-8097. DOI: `10.14778/3342263.3342644`. URL: `https://doi.org/10.14778/3342263.3342644`.

[6] Ryan Marcus and Olga Papaemmanouil. "Deep Reinforcement Learning for Join Order Enumeration". In: *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management.* aiDM'18. Houston, TX, USA: Association for Computing Machinery, 2018. ISBN: 9781450358514. DOI: `10.1145/3211954.3211957`. URL: `https://doi.org/10.1145/3211954.3211957`.

[7] Ryan Marcus and Olga Papaemmanouil. "Towards a Hands-Free Query Optimizer through Deep Learning". In: *arXiv e-prints*, arXiv:1809.10212 (Sept. 2018), arXiv:1809.10212. arXiv: `1809.10212 [cs.DB]`.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing Atari with Deep Reinforcement Learning". In: *arXiv e-prints*, arXiv:1312.5602 (Dec. 2013), arXiv:1312.5602. arXiv: `1312.5602 [cs.LG]`.

[9] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. "LEO - DB2's LEarning Optimizer". In: *Proceedings of the 27th International Conference on Very Large Data Bases.* VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 19–28. ISBN: 1558608044.