Research Challenges in Deep Reinforcement Learning-based Join Query Optimization

Runsheng Benson Guo, Khuzaima Daudjee
Cheriton School of Computer Science, University of Waterloo
{r9guo,kdaudjee}@uwaterloo.ca

ABSTRACT

The order in which relations are joined and the physical join operators used are two aspects of query plans which have a significant impact on the execution latency of join queries. However, the set of valid query plans grows exponentially with the number of relations to be joined. Hence, it becomes computationally expensive to enumerate all such plans for a complex join query. Recently, several deep reinforcement learning (DRL) based approaches propose using neural networks to construct a query plan. They demonstrate that efficient query plans can be found without exhaustively enumerating the search space. We integrated our implementation of a DRL-based solution to optimize join order and operators into the PostgreSQL query optimizer. In practice, we found limitations in the quality of the query plans chosen which are not addressed in existing approaches. In this paper we highlight some of these limitations and propose future research challenges along with potential solutions.

ACM Reference Format:
Runsheng Benson Guo, Khuzaima Daudjee. 2020. Research Challenges in Deep Reinforcement Learning-based Join Query Optimization. In *Third Workshop in Exploiting AI Techniques for Data Management (aiDM'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3401071.3401657

1 INTRODUCTION

A join query specifies a series of base relations $R_1, \ldots, R_n$ that are to be combined when they satisfy predicates $p_1, \ldots, p_m$. A join operator takes as input two relations and produces an aggregate relation based on the predicates specified between the two relations. The resulting relation can then be joined with other relations until the final result, consisting of all the initial base relations joined together, is produced.

The query optimizer of a database system is responsible for selecting a query plan that can be efficiently executed. For join queries, execution time is largely dependent on the order in which relations are joined and how each join is physically executed [6]. Analytical queries are popular workloads [4] but they can take hours to complete. Hence, improving the query optimizer can lead to significant time and monetary savings. The cost of each join is influenced by several factors including: (1) the data distribution of the input relations, (2) the join predicates between the input relations, and (3) the physical join operator (e.g., merge join, hash join). Consequently, the optimal query plan changes with not only the query but also with the data stored in the database system.

Recently, there has been growing interest in applying deep reinforcement learning (DRL) to query optimization [8, 11, 12]. These approaches are appealing because they can replace hand-tuned components of query optimizers and are adaptable to different data distributions and workloads [13]. We studied these DRL-based query optimizers to gain a better understanding of their performance limitations. Despite selecting high quality query plans efficiently a majority of the time, we found that these approaches have shortcomings because their underlying models (1) occasionally make poor estimates, and (2) lack a generalizable feature encoding. Since we were unable to obtain source code for existing work, we conducted our study using DQ+, our implementation of DQ [8]. We integrated DQ+ into the PostgreSQL query optimizer. In Section 2, we go over the traditional approach to query optimization as well as recently proposed DRL-based query optimization methods. We describe the architecture of DQ+ in Section 3, followed by our experimental results in Section 4 and an analysis of the limitations we encountered in Section 5. We propose that resolving these limitations will lead to further performance improvements for DRL-based query optimization.

aiDM'20, June 14–19, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8029-4/20/06…$15.00
https://doi.org/10.1145/3401071.3401657

2 RELATED WORK

Popular database systems such as PostgreSQL [1] and SQL Server [3] employ a cost-based approach [15] for query optimization. A cost model is used to estimate the cost of executing a query plan. The query optimizer searches for the plan with the least estimated cost. In this paper, we refer to the plan with the least estimated cost as the optimal plan. It is important to keep in mind the optimal plan is not necessarily the plan with the lowest execution latency. This is because cost models are based on cardinality estimates that in practice can be inaccurate and lead to greatly over or under estimating the cost of a query plan. Dynamic programming and genetic algorithm are two methods that PostgreSQL uses to search for the optimal plan.

2.1 Dynamic Programming

Dynamic programming (DP) can be used to exhaustively enumerate all query plans and was first proposed for query optimization in System R [15]. It is based on the observation that an optimal query plan for a join query is a join between two relations that were produced by optimal plans themselves. In DP, optimal plans are memoized in a lookup table and incrementally built. The optimal plan joining k relations is found by considering joins between optimal plans of 1 to $k-1$ relations. This is repeated until the optimal plan that joins all relations in the query is constructed. The number of plans that need to be considered grows exponentially with the number of relations to join. Consequently, even though DP will find the query plan with the lowest estimated cost, it can be computationally prohibitive to do so for large join queries.

2.2 Genetic Algorithm

The genetic algorithm can be used to cheaply search for query plans, even for large join queries [2]. It is a randomized search inspired by the process of natural selection. A group of *individuals*, each representing a query plan is initially randomly generated. During each iteration, a new query plan is generated by randomly recombining the join orderings of two query plans selected from the population. This is analogous to crossover in sexual reproduction. The algorithm maintains a fixed size *population* of the best query plans found so far by removing the most expensive query plans estimated by the cost model. The total time complexity is linear with respect to the number of base relations, so the algorithm is much faster than dynamic programming for complex join queries. However, there is no guarantee that the algorithm will produce an optimal plan.

2.3 Deep Learning for Query Optimization

Next, we describe several approaches that use deep learning to select a query plan. DQ [8] and ReJoin [12] formulate join order selection as a reinforcement learning problem and apply deep reinforcement learning (DRL) algorithms. In particular, DQ uses deep Q-learning. Join order selection is modeled as a Markov decision process. Each state is a query graph which encodes the relations that have been joined together as well as the relations that remain to be joined. An action corresponds to a join between two relations. Let s be a state, a be an action applied to state s, and let s' be the resulting state. The Q-function, $Q(s,a)$ returns the cost of the cheapest query plan possible that could be constructed by joining the remaining relations in s' in an optimal order. With the Q-function, the join ordering can be built by greedily selecting the join action minimizing Q until all relations have been joined. However, the Q-function is not known, so DQ approximates the Q-function with a neural network. This neural network is then used by the query optimizer to greedily select join orderings.

ReJoin is similar to DQ but uses the proximal policy optimization algorithm to find a join ordering instead of deep Q-learning. Both approaches build the join ordering based on the output of a neural network, avoiding exhaustive enumeration. Neo [11] is also similar to DQ and uses value iteration to train a neural network predicting the latency of an optimal query plan that could be built from a partial query plan. Neo determines not only the join order of a query plan but also the physical join operators and table access paths. The neural network searches for query plans through best first search instead of a greedy approach. Best first search still has an exponential runtime but in practice Neo caps the search time to 250ms which is enough to find good plans for a variety of workloads.

Another advantage of DRL-based query optimization approaches is that they are not limited by inaccurate estimates from a cost model. DQ and Neo initially rely on a cost model to train their underlying neural networks. However, these networks can be fine-tuned to reflect real latencies by learning from the latencies of queries executed in the database. Thus, DRL-based query optimizers have the potential to not only reduce optimization time but also improve the quality of query plans produced.

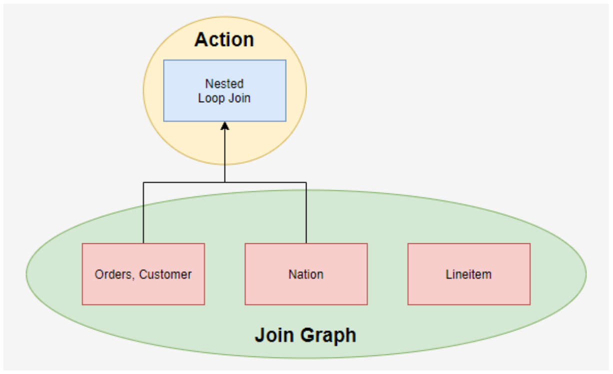

Figure 1: Feature Encoding Example

3 ARCHITECTURE

We implemented a variant of the neural network used in DQ [8] and integrated it into the PostgreSQL query optimizer. We call this DQ+. It was extended to select the physical join operators in addition to the join order, and draws ideas from both ReJoin [12] and Neo [11] to encode input for the neural network approximation of the Q-function. The Q-value of a state-action pair (s,a) is the log of PostgreSQL estimated cost of the cheapest query plan possible that could be constructed by joining the remaining relations after join a has been applied to query graph q. Since the Q-value of every action-state pair can be computed in this context, the neural network is trained to regress Q-values directly rather than using the standard deep Q-learning training process. The log of the cost is regressed rather than the cost to avoid exploding gradient issues during training. As in [8], state-action pairs and their corresponding Q-values are obtained from the DP table constructed when executing the training queries with the PostgreSQL DP optimizer.

Consider the following join query on a database containing the base relations *customer*, *nation*, *orders*, and *lineitem*:

```
SELECT *
FROM   customers as c,
       nation as n,
       orders as o,
       lineitem as l,
WHERE  c.customer_key = o.customer_key
       AND c.nation_key = n.nation_key
       AND l.order_key = o.order_key
```

One way to execute this query is to first join the *orders* and *customer* relations. This results in the join graph visualized in Figure 1. In particular, it contains the newly constructed *orders, customers* relation. From here, one possible action is to perform a nested loop join of *orders, customer* and *nation.* To evaluate the Q-value of performing this action in the current state, information about the action and state must be encoded as a fixed length vector which the trained neural network model expects as input.

The action and query in this example would be encoded for our model as :

$$\begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ c_L \\ c_R \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} L-Orders \\ L-Customer \\ L-Nation \\ L-Lineitem \\ R-Orders \\ R-Customer \\ R-Nation \\ R-Lineitem \\ Mergejoin \\ Hashjoin \\ NestedLoopJoin \\ L-CardinalityEstimate \\ R-CardinalityEstimate \\ Q-Orders \\ Q-Customer \\ Q-Nation \\ Q-Lineitem \\ Q-EquijoinOrders-Customer \\ Q-EquijoinOrders-Nation \\ Q-EquijoinOrders-Lineitem \\ Q-EquijoinCustomer-Nation \\ Q-EquijoinCustomer-Lineitem \\ Q-EquijoinNation-Lineitem \end{bmatrix}$$

(1) The teal rows 1-hot encode the tables belonging to the left and right input relations in the join action. The first four rows correspond to the left relation and the next four correspond to the right relation. *orders* and *customer* are in the left relation, so the first two rows are 1 while the third and fourth are 0.
(2) The orange rows 1-hot encode the physical join operator used for the join action, which is one of mergejoin, hashjoin, and nested loop join.
(3) The black rows encode the cardinality estimates for the left and right input relations, which are computed by the PostgreSQL optimizer
(4) The red rows 1-hot encode the relations belonging in the final query result.
(5) The six gray rows 1-hot encode whether there is an equijoin predicate between each of the six pairs of tables. Three of these rows are non zero, corresponding to the three equijoin predicates in the query.

4 EXPERIMENTAL RESULTS

We evaluated DQ+ by comparing the quality of query plans produced versus dynamic programming (DP) and genetic algorithm (GA) for the Join Order Benchmark (JOB) [9]. JOB consists of a set of challenging join queries on the real IMDB dataset. It features joins of up to 17 relations between correlated relations. DQ+ is trained on 70% of the queries in JOB and evaluated on the remaining 30%. The neural network took about 30 minutes to train.

Table 1: Query Plan Quality Comparison

Model	Median Ratio	95th Percentile Ratio	Mean Latency
DQ+	7.40x	138564.65x	37.52ms
DP	1.00x	1.00x	210.44ms
GA	1.00x	1.03x	65.95ms

4.1 Query Plan Quality

We evaluate the quality of a query plan by comparing the PostgreSQL estimated cost of the query plan to the lowest cost plan possible, which would be computed by the PostgreSQL DP optimizer. Table 1 provides the median and 95th percentile *cost ratio*, which is the ratio between the cost of the query plan produced by the optimizer and the optimal plan.

While DQ+ has the lowest mean latency, in the median case it produces plans 7.40 times more expensive than the cheapest plan. DP always finds the optimal plan because it enumerates through all possible query plans, and the nature inspired GA optimizer is also almost always able to produce near optimal plans. A big concern is that DQ+ occasionally produces plans that are extremely expensive. 5% of query plans produced by DQ+ are more than 10^5 times the cost of the optimal plan. We observed that some of the query plans selected by DQ+ took hours to execute or even caused the database system to crash because they required more disk space than the 2TB available to store intermediate results. The poor results suggest that the neural network in DQ+ is not able to adequately approximate the Q-function. It was surprising to us that the genetic optimizer was able to consistently build close to optimal query plans with comparable latency to DQ+. One disadvantage of the genetic optimizer is that it still relies on a hand designed cost model, which may not accurately reflect real execution latencies. DRL-based optimizers like Neo [11] do not have this limitation, however, we were unable to make a direct comparison because the source code for Neo has not yet been released.

4.2 Optimization Latency

Figure 2 compares the optimization latency, which is the time required to produce a query plan, for DQ+ and DP. Each point corresponds to one of the queries in the JOB and points falling above the black curve correspond to the queries for which DQ+ had a lower optimization time than DP. The points are color coded by the number of relations joined in the query. Even though DQ+ avoids the exponential search cost of DP, it is still slightly slower than DP for small join queries due to the overhead of making calls to the neural network. As the number of relations to join increases, the exponential search cost overshadows the neural network overhead and the optimization latency of DQ+ becomes relatively faster. This is observed for joins of 11 or more relations.

5 RESEARCH CHALLENGES

It is evident from the experimental results that there is significant room for improvement for the quality of queries produced from DQ+, especially at the tail. DQ+ and similar DRL-based optimizers [8, 11, 12] construct query plans entirely based on the output of their underlying neural network models. Consequently, improving

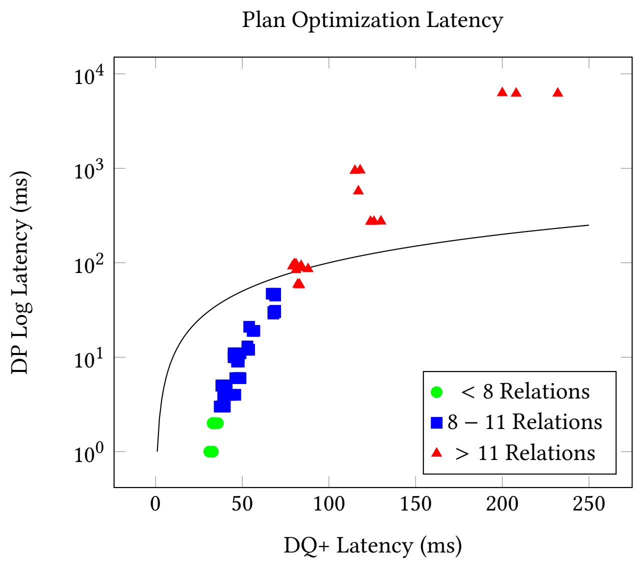

Figure 2: Plan Optimization Latency, DQ+ vs DP

the performance of DRL-based optimizers begins with improving their underlying models. In this section we outline two directions in which models of state-of-the-art DRL-based optimizers can be improved, along with their challenges and potential solutions. The architecture of the neural network used in DQ+ is primarily based on the design in DQ and ignores the tree structure of a query plan in the encoding process. Since then, there has been work on plan-structured neural networks and tree convolution on how to capture and learn from the tree structure of a query plan [11, 14]. While we do not explore all of the proposed model architectures in DQ+, the research directions we outline are still relevant to other architectures, and the solutions we propose can be used in conjunction with more complex architectures like tree convolution.

5.1 Model Robustness

In the 95th percentile scenario, DQ+ produces plans that are over 10^5 times more costly than the cheapest plan. It is important that a DRL-based optimizer does not produce any terrible query plans, as the orders of magnitude extra time taken to execute these queries is far greater than the time saved from avoiding full enumeration.

There are two approaches towards preventing terrible query plans from being generated. The first approach is to improve the model itself. Mean squared error (MSE) and mean absolute error (MAE) are two common loss functions used to train a neural network. We observed that models trained with MSE produced extremely expensive query plans less frequently than models trained with MAE. MSE and MAE are computed by the following equations, where e_i is the prediction error of the ith training example:

$$MSE = \frac{1}{n}\sum_{i=1}^{n} e_i^2 \qquad MAE = \frac{1}{n}\sum_{i=1}^{n} |e_i|$$

Since the error in MSE is squared, it penalizes large prediction errors relatively more than small errors. The network in DQ+ predicts the log of cost estimates, hence the prediction error is a measure of the relative error of the model's cost estimate. Consequently, MSE will train the model in a way that prioritizes avoiding large relative prediction errors. Large relative prediction errors can lead to higher cost plans being chosen over cheaper plans, so an MSE loss is preferred.

Even with a more robust model, inaccurate predictions may still occur from time to time. Hence, another approach to prevent poor query plans from being built is to add robustness around the model to account for inaccurate predictions. Neo [11] explores query plans greedily but does not stop after finding the first query plan. It continues exploring plans until a time threshold is reached. This is one way to build robustness around the query plan, however the time threshold may need to be adjusted depending on the number of relations joined in the query.

We explored a different approach to make DQ+ robust to model inaccuracies. We modified DQ such that it uses the Q-value estimates of the model to greedily build the top β cheapest query plans through beam search. Beam search is a breadth-first search that expands only the β best states at each level [5]. When $\beta = 1$, beam search is equivalent to the greedy algorithm in DQ. Out of the β plans generated, the one with the lowest cost estimated by the PostgreSQL cost model is chosen. We call this method *Beam DQ+*. β is a tuneable parameter called the beam width, and there is a trade-off between search quality and search time as β is increased. To illustrate this trade-off, Table 2 shows the median and 95th percentile plan cost ratios as well as the mean optimization latency when β is varied. Notice that the mean latency increases sub-linearly with β. This is because some query plans in the top β share the same sub-plans, allowing for computation to be shared. Moreover, neural network evaluations can be efficiently batched. When β is at least 64, the median plan cost is near optimal, and the 95th percentile plan cost drops dramatically to an acceptable level.

Table 2: Query Plan Quality & Search Time vs Beam Width

β	Median Ratio	95th Percentile Ratio	Mean Latency
1	7.40x	138564.65x	37.52ms
2	5.63x	92308.16x	49.38ms
4	4.72x	18.276.94x	71.28ms
8	1.58x	2894.39x	114.00ms
16	1.15x	393.84x	197.38ms
32	1.15x	288.93x	365.45ms
64	1.03x	28.36x	676.45ms
128	1.02x	19.25x	1208.26ms

Training a neural network to accurately approximate the Q-function currently remains a challenge for DRL-based optimizers. Two promising directions that deserve further exploration include improving the prediction accuracy of the model, and building robustness around the model.

5.2 Feature Encoding

Any neural network used to make query optimization decisions requires an adequate way to encode the features of a query and query plan as the network's input. DQ, ReJOIN, and Neo each have their own method of encoding these features. Deep learning

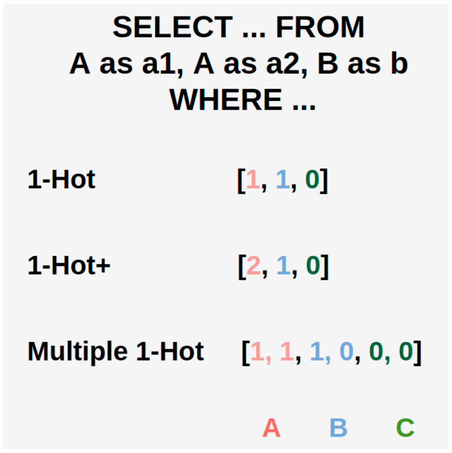

Figure 3: Self-Join Encoding Strategies

Table 3: Self-Join (SJ) Encoding Strategies Comparison

Dataset	JOB (No SJ)	JOB (SJ)	JOB (All)
1-Hot	3.88x	173.57x	5.96x
1-Hot+	5.27x	125.61x	15.05x
Multiple 1-Hot	**3.02x**	81.96x	**4.71x**
Multiple 1-Hot + DA	7.87x	**15.12x**	13.26x

Lowest cost ratios are bolded.

approaches for cardinality estimation [7, 10, 17] and query latency prediction [14] also propose their own methods of encoding query plans. However, all existing encoding strategies assume a fixed schema. In particular, all methods proposed so far 1-hot encode tables present in a query plan or query. Some methods also 1-hot encode the columns across all tables. This is problematic for three reasons. First, adding a column or table would change the encoding, requiring the neural network to be retrained. Second, an increase in the encoding would increase the parameters in the neural network. This leads to an increase in both the training and evaluation time of the network. This is unacceptable for commercial databases with a large number of tables. Finally, the 1-hot table encoding is not able to distinguish between multiple instances of a table appearing in a join (such as in a self-join), and is not defined for joins with sub-queries.

We explored two methods of extending the 1-hot encoding to explicitly represent self-joins. The first method, which we call *1-Hot+*, is simply encoding the number of times that a table appears, rather than a 1-hot encoding to indicate that the table appears. The second method, which we call *Multiple 1-Hot*, assumes the maximum number of times a table appears in a join query is known. If this number were c (for the JOB, $c = 2$), then *Multiple 1-Hot* reserves c entries to 1-hot encode up to c instances of the same table present in a join query. Figure 3 exemplifies the three encoding schemes for a query on a database with 3 tables A, B, and C. Table 3 compares the median query plan cost ratio of the encoding strategies on the subset of JOB queries without self-joins, the subset with self-joins, and the entire set of queries.

1-Hot+ and *Multiple 1-Hot* are able to encode multiple references to the same table, and consequently both outperform *1-Hot* on queries containing self-joins. Overall, *Multiple 1-Hot* produces the cheapest query plans. However, it still performs relatively poorly on the subset of queries containing self-joins, suggesting there is much room for improvement. *Multiple 1-Hot* needs to learn that each of the c fields corresponding to each table are logically equivalent and have the same properties. There are likely not enough training examples to do so, however, more training examples could be generated through data augmentation to help the model learn this relationship. Data augmentation is a technique for enhancing the size and diversity of a dataset without collecting new training examples by modifying existing training examples [16]. It is commonly used for image datasets, and for *Multiple 1-Hot*, new training examples can be generated by swapping the c locations used to 1-hot encode instances of a table. As shown in the last row of Table 3, when *Multiple 1-Hot* is trained with data augmentation (DA), plan costs for queries with self-joins are significantly reduced. However, this technique is not perfect as plan costs for queries without self-joins slightly increase.

From these results, it is evident that extending existing encoding schemes to work for self-joins is non-trivial. Moreover, sub-query joins are still unaccounted for. The underlying cause for both of these issues is the dependence on a fixed database schema for encoding. It is worthwhile to explore new encoding schemes that are invariant to the database schema. Such an encoding would avoid both expensive retraining costs and increase to the model size when tables or columns are added.

Very recently, RTOS [18] proposed an alternative DRL method using a Tree-LSTM network. Their architecture can represent self-joins and allows new columns or tables to be added to the database without retraining the entire network. We construe that this work brings us one step closer to achieving better encoding schemes.

6 CONCLUSION

DRL-based query optimizers [8, 11, 12] are attractive because they avoid the exponential search time of traditional optimizers and can be automatically tuned for the database and underlying hardware. They are already capable of matching state of the art query optimizers without relying on hand designed cost models [11]. However, there is still plenty of room for improvement. Existing approaches rely on a neural network to accurately predict which partial query plans lead to optimal complete query plans. It was shown in this paper that the model in DQ+ is susceptible to prediction errors which can lead to query plans with a very high cost relative to the optimal plan. We proposed two approaches to improve the robustness of the model. Our approaches are not limited to the architecture of DQ+ and can be combined with more complex architectures like tree convolution and plan-structured networks. Current models also rely on an inflexible feature encoding scheme; they assume a fixed schema and are not able to represent queries joining sub-queries or queries containing self-joins. We believe exploring new methods of improving model robustness and more flexible encoding schemes will pave the way to the next generation of DRL-based query optimizers.

REFERENCES

[1] 2020. *PostgreSQL database*. http://www.postgresql.org/

[2] 2020. *PostgreSQL: Genetic Query Optimizer*. https://www.postgresql.org/docs/12/static/geqo.html

[3] 2020. *SQL Server 2019*. https://www.microsoft.com/en-ca/sql-server/sql-server-2019

[4] Iqbal Alvi. 2019. *Transactional vs. Analytical Databases: How Does OLTP Differ from OLAP*. https://datawarehouseinfo.com/how-does-oltp-differ-from-olap-database/

[5] R. Bisiani. 1987. Beam Search. In *Encyclopedia of Artificial Intelligence*, S. Shapiro (Ed.). Wiley & Sons, 56–58.

[6] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, Washington, USA) *(PODS '98)*. Association for Computing Machinery, New York, NY, USA, 34–43. https://doi.org/10.1145/275487.275492

[7] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

[8] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR* abs/1808.03196 (2018). arXiv:1808.03196 http://arxiv.org/abs/1808.03196

[9] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[10] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality Estimation Using Neural Networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering* (Markham, Canada) *(CASCON '15)*. IBM Corp., USA, 53–59.

[11] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[12] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Houston, TX, USA) *(aiDM'18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 4 pages. https://doi.org/10.1145/3211954.3211957

[13] Ryan Marcus and Olga Papaemmanouil. 2018. Towards a Hands-Free Query Optimizer through Deep Learning. *CoRR* abs/1809.10212 (2018). arXiv:1809.10212 http://arxiv.org/abs/1809.10212

[14] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (July 2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[15] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[16] Connor Shorten and Taghi Khoshgoftaar. 2019. A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data* 6 (12 2019). https://doi.org/10.1186/s40537-019-0197-0

[17] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds. *Proc. VLDB Endow.* 12, 3 (Nov. 2018), 210–222. https://doi.org/10.14778/3291264.3291267

[18] Chengliang Chai Nan Tang Xiang Yu, Guoliang Li. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, USA, April 20-124, 2020*. IEEE, 1297–1308.