Learn What Really Matters: A Learning-to-Rank Approach for ML-based Query Optimization

Henriette Behr[1], Volker Markl[2], Zoi Kaoudi[3]

Abstract: Query optimization is crucial for any data management system to achieve good performance. Recent advancements in Machine Learning (ML) have led to several efforts in the database research community that aim at improving query optimization with the help of ML. In particular, many works propose replacing the cost model used during plan enumeration with an ML model. The goal of these works is to learn a regression model from previously executed query plans that estimates the runtime of a given plan. Interestingly, it is well-known that what really matters in query optimization is the relative order of the query plan alternatives and not their actual cost or runtime. We thus take a learning-to-rank approach and propose a novel neural network model architecture that can predict the rank of a plan. It considers a plan in comparison with alternative plans of the same query and together with a loss function that incorporates ranking metrics into the learning process we highlight the learning-to-rank objective.To enable training, we first extract features from query plans by adapting a state-of-the-art deep learning approach so that all features are independent of the input dataset schema. Second, we devise two score functions that map the runtime of plans to scores which are then used as labels during training. We integrate the trained model into an adapted bottom-up plan enumeration algorithm that finds the best possible execution plan for a given query. We evaluate our approach against two state-of-the-art ML models and the highly tuned cost model of a commercial database and measure the runtime of the plans chosen in each case when executed in the database. We show that our approach achieves up to an order of magnitude better query performance than the comparison models and is able to either match (for short and medium-running queries) or outperform the commercial database (up to 5× for long-running queries).

Keywords: query optimization; learning-to-rank; cost model

1 Introduction

Recent advancements in machine/deep learning (ML)[§] have shed light to many open data management problems, ranging from indexing and query optimization to database tuning. As query optimization is at the core of any database system, there are several efforts towards using ML in various (or all) steps of the process, e.g., for cardinality estimation [Ya19, Ne21, Ki19], join ordering [Yu20, MP18], and cost modeling [Ma19, Ma21, MP19]. In particular for cost modeling, the main idea is to completely replace the cost model and statistics used in a query optimizer with an ML model. Following the term cost-based query optimization, we

[1] TU Berlin, Germany, henriette.behr.1@gmail.com. Work done while conducting master thesis in TU Berlin.
[2] TU Berlin, Germany, volker.markl@tu-berlin.de
[3] IT University of Copenhagen, Copenhagen, Denmark, zoka@itu.dk. Work done while at TU Berlin.
[§] We use the term ML to also refer to deep learning.

doi:10.18420/BTW2023-25

call such an optimization *learning-based or ML-based query optimization*. The benefit of ML-based query optimization compared to cost-based is twofold. First, the optimizer can learn more complex models that are not necessarily linear functions and can thus better depict the database's performance. Second, it mitigates the tedious and time-consuming task of manually tuning the cost model.

Recent works that follow the learning-based paradigm, such as [Ma19, Ma21, Ka20], build a regression model based on previously executed query plans (training data). Such models provide an estimation of a plan's runtime which is then used for plan enumeration and pruning. Even though current approaches focus on estimating the real cost of query plans, in the end *what really matters* for the query optimizer is their relative order: Is plan A faster than plan B? By considering just the order of the plans the optimizer can still prune inefficient ones, keep more promising ones, and select one that performs well. Based on this observation, we argue that it suffices to learn the order (rank) of query plans instead of estimating their runtimes. We, thus, devise an ML-based learning-to-rank (LTR) approach that ranks execution plans without estimating their execution time. Our approach scores execution plans of a given query and ranks them based on their scores. Although researchers have extensively used LTR models for information retrieval or recommendation [Li11], to our knowledge, an LTR approach has not been used for query optimization yet.

To reach this goal, we have to overcome some challenges. First, there is a wide spectrum of LTR approaches (pointwise, pairwise or listwise), loss functions, and neural network architectures that one has to choose from. We, thus, need to make design decisions in all these directions so that our solution performs at least as good as a highly-tuned cost-based optimizer, if not better. Second, although one could use the estimated runtimes to get a ranking over the plans, these are not only hard to estimate but also do not always preserve the ordering of the plans. To train an ML model that can rank execution plans, we need to devise a score function that uniformly ranks the different execution plans. Third, we need to study whether and how existing enumeration algorithms can be used with LTR models.

We tackle these challenges and make the following main contributions:
(1) We devise a listwise LTR approach that is more suitable for ML-based query optimization. We consider several loss functions and a neural network architecture tailored to the query optimization problem (Section 3.1).

(2) We propose two score functions, one local, which assigns scores to plans per query, and one global, which assigns scores across the plans of all training queries (Section 3.2).

(3) We describe our featurization scheme and adapt a well-known bottom up plan enumeration algorithm to work with our listwise LTR model (Sections 3.3 and 3.4).

(4) We extensively evaluate our LTR approach against other ML models as well as SQL server which includes a well-tuned cost model. Our results show that our LTR approach matches the performance of a commercial database's optimizer for short-running queries and even outperforms it (up to 5×) for long-running queries. Our approach consistently outperforms state-of-the-art ML-based baselines: it chooses plans with up to an order of magnitude better runtime performance.

We conclude the paper with related work and a discussion on future directions.

2 Preliminaries

Learning-to-rank (LTR) algorithms are a special type of supervised machine learning – the task of ranking is neither a classification nor a regression task. Instead, the goal of LTR algorithms is to predict a score for a set of items with the goal of sorting them by their predicted score such that a ranking can be given as a result [Li11]. An LTR algorithm considers an item with a high score to be more relevant to a given query than an item with a low score. Consequently, the goal of LTR tasks is different from classification or regression tasks and the algorithm itself does not regard the predicted score but the correct order of the input items for calculating the loss and improving the model. Similarly, the metrics for measuring predictive performance also differ from the "standard" metrics used for classification or regression tasks.

Generally, there are three main approaches for LTR algorithms: pointwise, pairwise, and listwise [Li11]. Pointwise algorithms handle each item separately as input and calculate its relevance score, which in turn is used to sort all items and get the final ranking. However, these algorithms do not consider the inter-dependency between items as the loss is calculated separately for each item. Unlike pointwise approaches, in a pairwise approach two items serve as input for one prediction which is given by three distinct values $\{-1, 0, 1\}$. The model outputs -1 when it predicts that the first input sample has a lower rank than the second input sample; otherwise, it outputs 1. The result is 0 if the two inputs are the same or if the model considers them to have the same relevance. In the listwise algorithms, the entire list of items belonging to the same query is used as input and the output is a corresponding sorted list. Both pairwise and listwise algorithms belonging can often be further categorized based on their underlying ML algorithms, such as Support Vector Machines (SVM) or Neural Networks (NN). As we will detail in the next section, we chose a listwise approach, and in particular the LambdaLoss [Wa18], which uses the idea that the scores define a distribution:

$$L(y, s) = - \sum_{y_i > y_j} \log_2 \sum_{\pi} \left(\frac{1}{1 + e^{-\sigma(s_j - s_i)}}\right)^{\delta_{ij} |G_i - G_j|} H(\pi | s) \tag{1}$$

where s_i is the true score for item i, and y_i describes the weight, i.e., the relevance grade the model predicts, $H(\pi|s)$ is a "hard assignment distribution" [Wa18] with $H(\hat{\pi}|s) = 1$ and $H(\pi|s) = 0$ for all permutations of the items with $\pi \neq \hat{\pi}$. Thereby, $\hat{\pi}$ is the permutation in which all documents are sorted correctly. δ_{ij} a tunable parameter and G is the gain function $2^y - 1$. Furthermore, the researchers extended this loss function to a top-k approach, similar to ListNet, with k being a tunable parameter. The authors define this extended loss such that

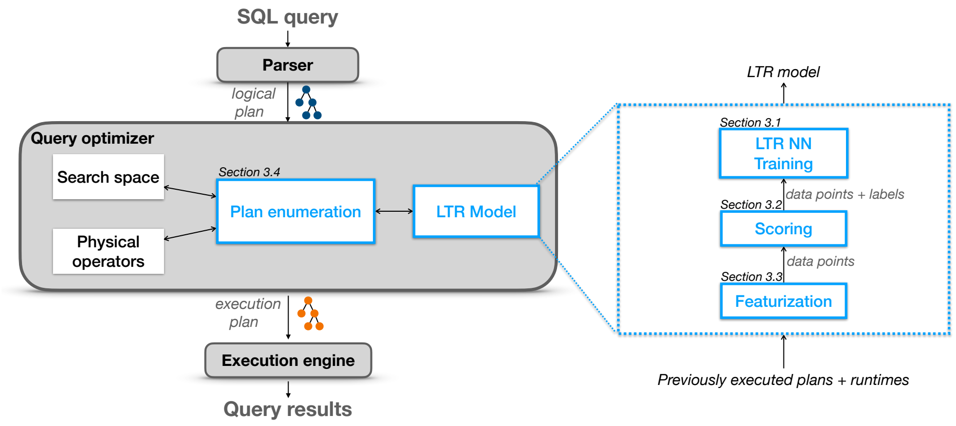


Abb. 1: Overview of proposed approach and contributions (in blue).

only those elements with $i \leq k$ or $j \leq k$ regarded, i.e., if one of the elements belongs to the true top-k elements.

3 Learning-to-rank for Query Optimization

Our goal is to alleviate database administrators from the burden of tuning the cost model of a database and potentially improve query performance by replacing the cost model and statistics of a query optimizer with an LTR model. Figure 1 shows an overview of our proposed ML-based query optimization process that uses an LTR model. The LTR model is crucial for evaluating and ranking the execution plans enumerated during the plan enumeration phase. To train the LTR model we first need to determine the LTR approach we will use (point-wise, pairwise, or list-wise) and a neural network architecture (Section 3.1). Then, to train the model, we need to determine our training data (data points and labels). Given a dataset composed of previously executed plans and their runtimes, we first need to determine how to featurize (i.e., extract a numeric vector) the execution plans to construct the data points (Section 3.3) and how to extract relevance scores based on the provided runtimes to construct the labels for our training data (Section 3.2). Once the LTR model is built, the plan enumeration uses it to prune the search space and determine the (near-)optimal execution plan. In Section 3.4, we elaborate on how we adapted an existing bottom-up plan enumeration to be able to use the LTR model.

3.1 LTR model architecture

First, we need to decide on the LTR approach we will use. As discussed in Section 2, most LTR algorithms can be categorized into one of the three following groups: pointwise

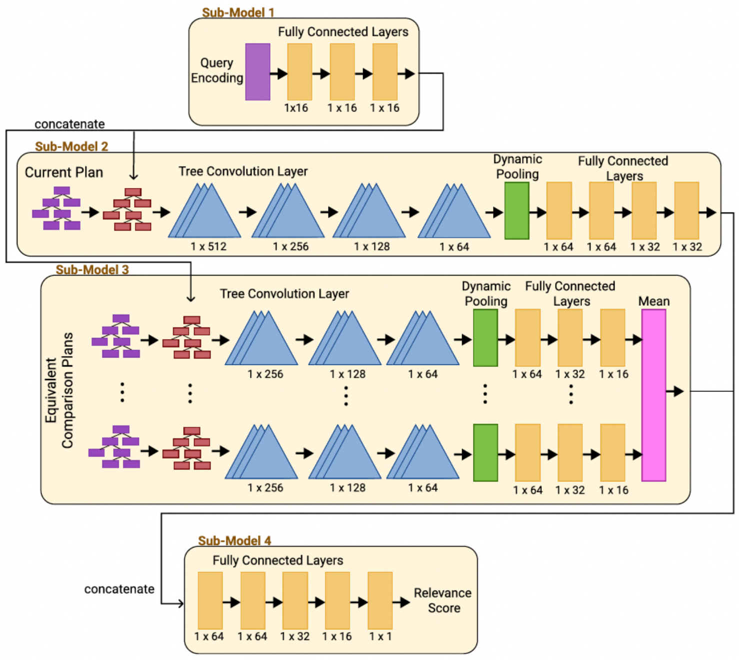


Abb. 2: Neural network architecture of our LTR model.

approaches, pairwise approaches, and listwise approaches [Li11]. Pointwise algorithms do "not consider the inter-dependency between items" [Li11] as the loss is calculated separately for each item. In our case, this means that the rank of an execution plan would be solely based on its (estimated) runtime and would not consider any comparison metric among equivalent plans. This would defeat the purpose of using an LTR model in the first place. For this reason, we discard a pointwise approach. Because traditional plan enumeration algorithms prune the search space by comparing two subplans at a time, it is intuitive to use a pairwise approach for our LTR model. However, such an approach (i) uses an objective loss for minimizing errors in classification of pairs rather than minimizing errors in ranking of items, (ii) requires a large number of pairs to train on, which can be computationally costly to generate, and, (iii) assumes that the pairs are generated i.i.d. which is not true in the case of equivalent execution plans. *We, thus, aim for a listwise approach.* Our experimental results shown in Section 4.3 demonstrate how our proposed listwise solution outperforms a pairwise approach.

Once we settle for the LTR approach, the next problem that we need to tackle deals with the neural network architecture that we shall use to build an LTR model. We get inspired by a popular listwise LTR architecture called FATE [PGH18], and the architecture of [Ma19] that uses a neural network for estimating the runtime of execution plans. Figure 2 depicts the neural network architecture of our LTR model. Similarly to FATE, our model uses equivalent execution plans as additional information to the current plan, for which the model has to predict a relevance score. Consequently, the model expects multiple equivalent plans as input and estimates the relevance score for each one of them. The model considers every plan of the inserted list individually as the current plan and utilizes every other plan in the list as a comparison plan.

For a given query (either during prediction or training), we first calculate the query encoding and plan encoding, both depicted with violet color in Figure 2. The query encoding forms a vector while the plan encoding is tree-shaped (see Section 3.3). Then, a first sub-model (`Sub-Model 1`), consisting of several fully-connected layers, takes the query encoding as input and outputs a vector of length 16. Subsequently, the model concatenates the output of `Sub-Model 1` with the plan encodings of all plans, i.e., the current plan and the comparison plans. The extended plan encoding of the current plan serves as input for a second sub-model (`Sub-Model 2`) while all extended comparison plans are inserted into a third sub-model (`Sub-Model 3`) separately. Both mentioned sub-models have a similar architecture consisting of multiple tree convolution layers, a dynamic pooling layer, and several fully-connected layers. The difference between the two sub-models is the number of layers as (`Sub-Model 2`) consists of more layers. The reason for this is that we want to put more focus on the current plan than on the comparison plans. Sub-Model 3 consists of a final layer that calculates the mean of the outputs of the comparison plans. Before the model transfers this vector to the last sub-model (`Sub-Model 4`), it concatenates this resulting mean vector and the vector produced by (`Sub-Model 2`) so that the final model considers each plan in comparison with the rest. Finally, it inserts the result into (`Sub-Model 4`), consisting of five fully-connected layers. The last fully-connected layer outputs the relevance score.

Note that the model architecture itself already implies a listwise comparison strategy regardless of the loss functions used. To further enforce the listwise approach we use the LambdaLoss [Wa18] function, which incorporates a ranking metric, as described in Section 2. Note that our main focus is not the implementation of a new loss function, which is why we choose to use an existing one.

3.2 Scoring of execution plans

The required training data to build an LTR model include execution plans (data points after featurization) and their rank (label). Yet, the data we usually have available from execution logs is execution plans and their runtimes.[¶]. One may think that it suffices to simply sort the plans and based on their runtime and convert it to a rank. However, this is not ideal because we lose information on how the runtimes differ among the plans. For example, in the case of OLAP queries, if plan A has a runtime of $200ms$ and plan B has a runtime of $205ms$,

[¶]The way to acquire such data is orthogonal to our approach and we thus, do not discuss it in the paper. For our implementation we have used DataFarm [Ve21].

we would like to give them the same score as their difference in runtime is insignificant. Conversely, in the case of OLTP workloads we would like to assign a different score to the above two plans. In few words, the goal of using a score function is is to take into account the relative performance among the plans.

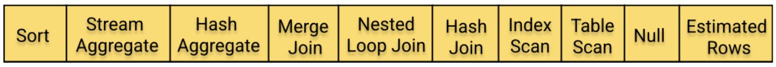
To enable the model to learn relevance scores to rank different plans, we need to devise a score function for scoring the training plans. The intuition behind devising such a score function is that, based on the runtimes of the plans, we want to teach the model which plans are good and bad, but also which plans have a similar performance. The best plan for a specific query should receive the highest relevance score, while the worst plan should receive a low score. Additionally, we want to be able to map plans with a similar runtime to the same score. However, the definition of "plans with a similar performance" can vary from query to query when looking only at execution times. Depending on the query, similar plans can differ, for example, in their run time by only $1ms$ or by $2000ms$. Utilizing a score function ensures that the plans follow a uniform scoring scheme. We devise two different score functions: a linear score function and an advanced global score function utilizing agglomerative clustering. For both functions, we use a hyperparameter s_{max} that determines the maximum score of an execution plan. Thus, by tuning s_{max} we can adjust the granularity of the score assignment, depending on our scenario (i.e., OLAP vs OLTP cases).

3.2.1 Linear score function

Our first proposed score function is the *linear score function*, which is a straightforward approach for transforming the increasing run times into decreasing relevance scores. The function uses a decreasing linear function $f : \mathbb{N} \rightarrow \mathbb{R}$ for calculating the scores such that the larger the execution time for a specific plan, the smaller the relevance score becomes. To calculate a score, the function maps an execution time in ms to a real-valued score $[0, s_{max}]$ with respect to other equivalent plans for the same query and the hyper-parameter s_{max} defining the maximum score. The parameter s_{max} influences the upper bound beyond which the values are receiving a score of 0, which we explain in the following.

Considering one query at a time, the linear score function calculates a linear function between the minimum run time t_{min} of a set of equivalent plans and $s_{max} \times t_{min}$. This minimum value t_{min} receives a score of s_{max}, and the value $s_{max} \times t_{min}$ gets mapped to a value of 0. Between those two values, the score decreases linearly. Additionally, the function sets every execution time higher than $s_{max} \times t_{min}$ to 0. Equation 2 expresses this formula.

$$f_{s_{max}}(x_i) = \begin{cases} s_{max}, & \text{if } x_i = t_{min} \\ \frac{-s_{max}}{(s_{max}-1)}(\frac{x_i}{t_{min}} - s_{max}), & \text{if } t_{min} \leq x_i \leq t_{min} \times s_{max} \\ 0, & \text{if } x_i \geq t_{min} \times s_{max} \end{cases} \tag{2}$$



Sort	Stream Aggregate	Hash Aggregate	Merge Join	Nested Loop Join	Hash Join	Index Scan	Table Scan	Null	Estimated Rows

Abb. 3: Features used in the plan encoding.

To clarify this score function, we consider the following example for the linear score function with $s_{max} = 5$. For a query, the best plan of all possible plans, i.e., the plan with the shortest run time, needs 10s for its execution. Therefore, the function sets $t_{min} = 10$ and maps this plan to a score of 5. Every plan with a run time higher than or equal to 50 s ($s_{max} \times t_{min} = 5 \times 10$) receives a score of 0. Between these two runtimes, the score decreases linearly, e.g., a query with a run time of 25s receives a score of 2.5 and a run time of 20s receives a score of 3.75.

3.2.2 Global agglomerative score function

Besides the linear score function, we devise a second score function, the *global agglomerative score function*. This function utilizes the unsupervised clustering algorithm *agglomerative clustering* [SB13]. Unlike the linear score function, it considers all available plans with their execution time at once, independently of the query they belong to. To make a global scoring feasible, the score function scales the query plans at first. For each query separately, it takes the minimum execution time and divides every equivalent plan for this respective query by the minimum execution time of this query. This results in factors in $[1, +\infty)$ describing how many times slower the execution of this plan is w.r.t. the minimum execution time for this query. During calculation, all factors for all queries of the training set are stored in an array, and on this array, we apply agglomerative clustering. However, before the clustering step, we need to remove outliers. This is because a huge outlier, i.e., a plan with long execution time, could be clustered as its own cluster while plans with small values can be all assigned into one cluster due to the nature of the agglomerative clustering algorithm. To achieve this, the function calculates the $border$-th percentile of factors, where $border$ is a hyperparameter. The algorithm sets every factor below a percentile $border$ to the value of $border$.

After execution of the agglomerative clustering, we sort the resulting clusters by the minimum value for each cluster. Finally, all query plans within the cluster with the smallest value receive a value of s_{max}, and for every next cluster in the sorted version, the scores attached decrements by 1. We show the explained procedure in Algorithm 1.

3.3 Featurization of execution plans

As described in the previous subsection, our model architecture requires two different inputs: the query encoding and the plan encoding. The query encoding is a simple vector

```
**Input:** $s_{max} \in \mathbb{N}$, $border \in (0, 1]$ and execution times $T = \{T_{11}, \dots T_{nq}\}$ with $q$ being the query
        number and $n$ being its index inside query $q$
**Output:** Scores $S = \{S_{11}, \dots S_{nq}\}$
Create empty array $factors$
**for** $i \in [1, q]$ **do**
    minimum = min($\{T_{1i}, \dots, T_{ni}\}$)
    **for** $t$ *in* $\{T_{1i}, \dots, T_{ni}\}$ **do**
        Append $\frac{t}{minimum}$ to factors
    **end**
**end**
percentile = get $border$th-percentile of factors
**for** $f$ *in factors where* $f \geq$*percentile* **do**
    $f$ = percentile
**end**
AgglomerativeClustering(factors, numberClusters=$s_{max}$)
Sort clusters by their minimum value
Give first cluster a score of $s_{max}$ and decrement $s_{max}$ for succeeding clusters
```

Algorithm 1: Calculation of the global agglomerative score

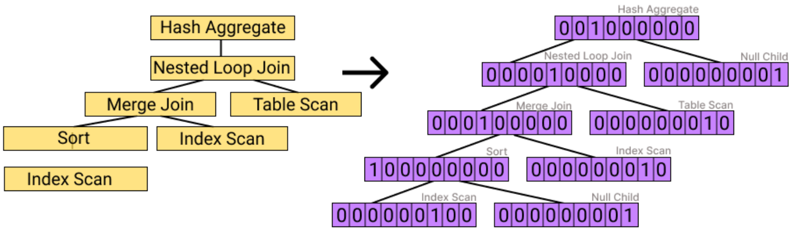


Abb. 4: Example featurization of an execution plan (plan encoding). For the sake of simplicity, we omit the last feature, which is the estimated intermediate cardinalities.

describing several different query features, such as whether a GroupBy exists or how many joins the query contains. The plan encoding consists of multiple vectors arranged in a tree shape describing the utilized operators of the execution plan at each position by using a one-hot-encoding strategy. The features of the plan encoding are similar to the features used in [Ma21]; we encode every operator to a one-hot-encoded vector for every important node in the plan. Thereby, we consider the different join and aggregate physical operators as well as two scan operators – index scan and table scan – and the sort operator. Figure 3 shows the features we use for the one-hot-encoding of each operator in the plan encoding.

The tree-shaped plan encoding considers binary trees which is the most popular type of query plans. As the tree convolution layer requires each inner node of the tree to have two children, we need to "fill" the children of unary operators, such as `sort` or `aggregate`. For this, we use a feature vector "Null" to indicate "null-children" and add a second "null-child"

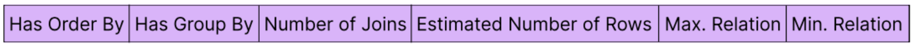


Abb. 5: Example featurization of an execution plan (plan encoding).

to unary operators. In addition to the one-hot-encoding of the operator, we extract the estimated resulting rows from the database optimizer and append this estimation to our feature vector. Figure 4 illustrates an example featurization of an execution plan. The plan in this example consists of two joins, three scans, a sort, and an aggregate. Moreover, we can see the use of null-children, which are inserted as the right child for the sort and the aggregate node. In the figure, we omit the estimated rows feature for simplicity.

In addition to the plan encoding, we have to encode the query into a feature vector as illustrated in Figure 5. In contrast to [Ma21], we keep this vector simple and independent of the relations used, i.e., there is no one-hot-encoded information about the utilized columns or relations. This allows our featurization to be used out of the box for any dataset. Instead, it includes information about whether the query utilizes a GroupBy and an OrderBy as well as the number of joins and the result size of the query estimated by the database. Furthermore, we append information about the maximum and the minimum number of rows of the relations used.

Before training, we normalize all features of the plan and query encoding. Therefore, for every feature used, we extract its maximum and minimum value and normalize all the features using the formula: $norm(x) = \frac{(x-min)}{(min-max)}$.

3.4 LTR-based Plan Enumeration

Bottom-up enumeration algorithms are very popular with existing database systems. In a learning-based query optimizer, these algorithms are responsible for calling the model and receiving its cost estimations (see Figure 1). However, they utilize a pairwise comparison of plans during the enumeration. This can work out of the box with an ML model that simply outputs the estimated runtime of a plan or with a pairwise LTR model. However, our model is listwise, i.e., it ouputs a ranked *list* of plans. One could use it with the default plan enumeration if the list consists of only two plans. However, this would be inefficient. For this reason we need to adapt the state-of-the-art enumeration algorithms to work with ranked lists of plans and not with the cost of pairs of plans.

After considering different bottom-up enumeration algorithms, we decide to use the DPccp algorithm by Moerkotte and Neumann [MN06]. This algorithm has the benefit of using graph algorithms, making it a better fit to transform the graph into the tree-structured plan-level encoding, which is needed for the LTR model to estimate the relevance score. To make DPccp work with our model we need to adjust the part of the algorithm that compares two plans at a time w.r.t. their cost. As a result, the decision for the best sub-plan in a set of equivalent sub-plans is made shortly before the sub-plan is needed for joining purposes. Furthermore, we extended the algorithm to get a third input, parameter k, next to the query graph G and model $model$. Parameter k is tunable and decides to how many execution plans the model reduces the list of possible plans during its prediction. For example, for $k = 10$, at every prediction of the model, the model returns the top $k = 10$ plans with the highest predicted relevance score. Consequently, the enumeration algorithm calls the model whenever the number of possible equivalent plans exceeds k.

```
Input: A connected query graph $G$ with relations $R = \{R_0, \ldots, R_{n-1}\}$, LTR model $model$ and
       parameter $k$
Output: an optimal bushy join tree
for all $R_i \in R$ do
    PossiblePlans($\{R_i\}$) = getScan($R_i$);
end
for every $S_1$ in $EnumerateCsg(G)$ do do
    for every $S_2$ in $EnumerateCmp(G, S_1)$ with $S = S_1 \cup S_2$ do
        if length of PossiblePlans($S_1$) > $k$ then
            $p_1$ = $model$.predictBestK(PossiblePlans($S_1$), k=$k$)
        end
        else
            $p_1$ = PossiblePlans($S_1$)
        end
        if length of PossiblePlans($S_2$) > $k$ then
            $p_2$ = $model$.predictBestK(PossiblePlans($S_2$), k=$k$)
        end
        else
            $p_2$ = PossiblePlans($S_2$)
        end
        Append plans from CreateJoinTrees($p_1$,$p_2$) to PossiblePlans($S$)
    end
end
return $model$.predictBestK(PossiblePlans($R$), k=1)
```
Algorithm 2: Adjusted DPccp algorithm based on [MN06]

Algorithm 2 shows the pseudocode of our adapted enumeration algorithm. In the beginning, the algorithm starts a for-loop for receiving all possible scans for each node in G, similar to the first loop in the original DPccp. After it has calculated every possible scan and saved it in a data structure *PossiblePlans*, it starts a second and third for-loop for calling the sub-graphs with two functions – *EnumerateCsg* and *EnumerateCmp* – that we extracted from the original DPccp algorithm and left unchanged. Inside the inner for-loop, for both sub-graphs S_1 and S_2, the algorithm tests if the number of possible plans for calculating the respective subgraph exceeds k. If it does, the algorithm calls the LTR model, reducing the number of possible plans to k plans for which it predicts the highest relevance scores. Afterward, it stores the remaining k plans in variable p_1 or p_2. If the number of possible plans is already smaller or equal to k, the algorithm directly stores all plans in p_1 or p_2 without further reduction. After p_1 and p_2 are created, the function *CreateJoinTrees* calculates all possible joins for each pair in p_1 and p_2, i.e., it considers every join operation. The algorithm stores the resulting sub-execution plans for S with $S = S_1 \cup S_2$ in our data structure $PossiblePlans(S)$. Eventually, when both for-loops have finished, the model predicts the best plan from $PossiblePlans(R)$, i.e., for the entire graph, with $k = 1$.

4 Experimental Evaluation

We evaluate our proposed solution against the cost-based query optimizer of a commercial database system and two state-of-the-art learning-based query optimizers. Specifically, we evaluate whether the query performance for a well-known query workload improves with our learning-to-rank approach compared to other baselines for both seen and unseen data.

4.1 Setup

We first describe our experimental setup, which includes our utilized hardware/software, baselines, and characterization of the utilized training and test data.

Hardware and Software Specification. We used a computer with an AMD Ryzen 7 1700X Eight-Core processor at 3.77 GHz with 32 GB main memory. We implemented all our components in Python version 3.9.7 and used PyTorch 1.10.0 for building both ours and the comparison ML models. For fair comparison, we use a commercial database to execute the plans chosen in of the cases we study. To automate the execution of our execution plans in the database system, we used the Python library `pyodbc` version 4.0.32.

Baselines. We use a commercial database (denoted as `DB X`) and three different comparison models as baselines. The latter are plugged in our implementation that uses the adapted enumeration algorithm as described in Section 3.4. We compare against two state-of-the-art models of learning-based optimizers – `Bao` [Ma21] and `Neo` [Ma19][‖] – as well as a simple baseline comparison model. This `baseline model` is a pairwise LTR approach and has a similar architecture to Bao's model. It uses the linear score function with $s_{max} = 50$ and RankNet [Bu10] as a loss function for training. For enumerating through the plans, we use our adapted DPccp enumeration algorithm for every case and set $k = 10$. For our approach (denoted as `LTR model`), we have ran extensive experiments to find the best combination of score functions and hyperparameters as well as loss functions. However, due to space limitations, we report the performance of our best model which uses the global agglomerative score function with $s_{max} = 50$ and $border = 97$ and the LambdaLoss function with the top-k extension (see Section 2).

Training Data. For training, testing and evaluating our solution, we decide to mainly use the database schema from the TPC-H benchmark[**]. Due to the lack of already labeled training data fitting TPC-H's database scheme, we generate and label appropriate training data

[‖] In the future, we plan to experiment with zero-shot cost models [HB22] as well.

[**] See: `https://www.tpc.org/tpch/`

Tab. 1: Number of generated testing queries based on their number of joins and runtime on 1GB data.

Query Length	1 Join	2 Joins	3 Joins	4 Joins	5 Joins	6 Joins	7 Joins
Short Queries (< 2sec)	8	8	8	8	8	8	8
Medium Queries (≥ 2sec & < 30sec)	2	8	8	8	8	8	8
Long Queries ≥ 30sec	0	2	4	7	8	8	5

using DataFarm [Ve21], a training data generation tool for ML-based query optimization. DataFarm uses as input a small amount of SQL queries and information about the different tables to create further similar queries. The initial version of the tool was implemented for Flink and, thus, we had to adapt some parts for our purpose. Specifically, we use six different TPC-H queries serve as input (namely, Q1, Q3, Q11, Q13, Q17, and Q21). These queries contain operators that our featurization process supports. Based on these six queries, we let DataFarm generate 2,000 SQL queries with maximum seven joins and five variations each. However, to get our training data we need runtimes for specific execution plans and not only for SQL queries. Thus, we insert each query into the plan enumeration algorithm and receive different valid execution plans. Because the amount of possible plans increases exponentially with the amount of relations in the query, we decided to implement a limit of 100 different execution plans per query. Thereby, the enumeration algorithm prunes the sub-plans randomly such that the resulting execution plans differ for different queries. Lastly, we need the runtime of these execution plans. Therefore, we let DB X run the different execution plans on 1GB TPC-H, forcing the usage of a specific plan. Due to time limits, we are not able to label all generated plans, resulting in a training set consisting of 25,067 plans. Furthermore, we set the timeout for all queries to six hours, which results in the problem that not all execution plans have an execution time in the end. For our LTR model, this does not pose a problem, as plans with a bad execution time are always assigned a relevance score of 0. However, for the two comparison models, Bao and Neo, this circumstance is a problem, as these models need the execution time of the plans for their estimations. To solve this problem, we set these estimations to 6 hours.

Testing Data. For evaluating the results of the different models, we decide to use several different queries. The first set of testing queries consists of 136 queries which we generated during the training data generation process but did not use for training purposes. We distinguish these queries based on their runtime: *short* queries with a run time of less than 2sec, *medium* queries with a run time of less than 30sec but more than 2sec, and *long* queries needing more than 30sec. For every model tested, we let it predict the best execution plan for each query with the help of our plan enumeration algorithm. Afterward, we insert this plan into the commercial database to measure its real execution time. In Table 1, we show the exact number of queries split by their number of joins for each category.

To test whether the performance of our model's suggested execution plan remains similar when increasing the size of the database, we increase the size of the data to 5GB and let our model predict the same test queries but using the larger dataset. However, due to time limitations, we had to reduce the number of queries from 136 to 95. As many queries have longer execution times on the larger dataset, we do not use the same definition of short, Tab. 2: Number of generated testing queries based on their number of joins and runtime on 5GB data.

Query Length	1 Join	2 Joins	3 Joins	4 Joins	5 Joins	6 Joins	7 Joins
Short Queries	8	8	7	7	6	2	0
Medium Queries	2	5	4	7	4	2	3
Long Queries	0	5	6	4	9	3	3

medium and long queries. Every query with a runtime with 15 seconds or less is regarded as short query. Medium queries have an execution time between 15 and 60 seconds, and we consider every query with more than a minute execution time as long query. Table 2 shows the corresponding queries distribution.

Lastly, to evaluate the performance of our models on *unseen data*, we use the IMDB dataset in combination with queries from the Join Order Benchmark (JOB)[††]. Out of these queries, we use 26 different queries all of them having between 3 and 11 joins for our experiments. Note that the TPC-H data we use for training has a maximum of 7 joins, and, using this dataset, we can test our model's performance for queries with a larger number of joins.

Evaluation Metrics. For each query q we measure the execution time of the models' best predicted plan when inserted in `DB X`. Furthermore, we record the time the commercial database, `DB X`, needs for running q. To ensure a fair comparison, we let `DB X` at first ouput the execution plan for q and then insert it again into `DB X` as we do for the other models and measure its runtime. For our evaluation, `DB X`'s time serves as a baseline as it contains a highly tuned cost-based optimizer. We, thus, use not only the exact execution times but also the ratio of the execution time of the models' predicted plans to `DB X` baseline, i.e., we divide for every query the time of our models' plans by the time that `DB X` needs.

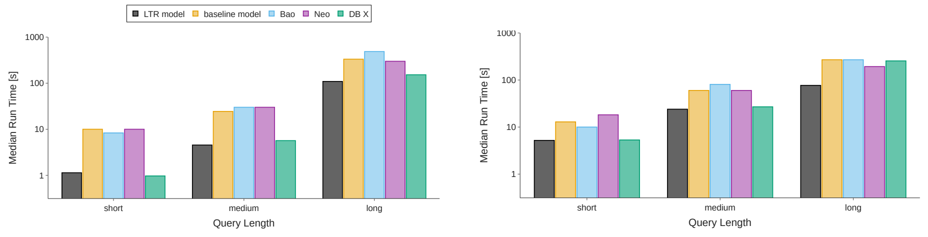
4.2 Performance against State-of-the-Art

We first evaluate how the models perform on our generated test queries.

4.2.1 Absolute performance

For a first overview of the model's performance, we plot the median execution time for the predicted plans categorized by the query type in Figure 6a. Note that the $y - axis$ is in logarithmic scale. While our `LTR model` seems to have a good performance which is able to match `DB X`'s performance, the plans of the comparison models have an extremely poor performance. To investigate the reasons for their bad performance, we analyze some of their predicted execution plans. In examining some of the plans, we note that `Neo` often predicts merge and nested loop joins for queries where the other models predict mainly hash joins.

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



(a) 1GB data – used in training data generation– (b) 5GB data – not used in training data generation–

Abb. 6: Median runtime of our `LTR model`, the three comparison models, and `DB X`. Our `LTR model` matches the performance of `DB X` and exceeds it for long-running queries while it always outperforms `Neo`, `Bao`, and the `baseline model`.

Most of these predicted merge joins require a preceding sort. We hypothesize that this is the reason why the plans predicted by `Neo` often throw a timeout. When analyzing the plans of the `baseline model` and `Bao`, we observe that also these models often predict merge joins with a preceding sort even though not as often as `Neo` does. Similar to `Neo`, we assume that these join operators in combination with sorts result in worse plans.

Note that we trained our model on execution plans with runtimes retrieved over a database with size of 1GB. Therefore, it is possible that our model has a different performance on queries for larger datasets. To test this, we run another set of experiments on 5GB of data for our model, the three comparison models and `DB X`. Here, we again use the same generated queries as before even though we had to reduce the number of test queries to 95 queries (see Table 2). We show the median runtimes of the different query types in Figure 6b. Similar to the prediction on the smaller dataset, our model significantly outperforms the comparison models. For short and medium queries, it matches SQL Server's performance, and it is also able to outperform SQL Server for long queries.

4.2.2 Relative performance

To gain more insights on the results of the predicted plans, we plot the relative performance of the models' chosen plans in relation to `DB X`. Here, we divide for every predicted plan its time by the time `DB X` needed. Figure 7 shows the resulting boxplots when the 1GB TPC-H dataset is loaded in `DB X`. A value below the red line signals that the corresponding model predicted a plan with a runtime lower than the `DB X` suggestion. In the boxplots, we can see clearly that our model has the best performance of all four models. Furthermore, except for some outliers, it is often able to match or outperform `DB X`, especially for medium and long queries. For short queries, it is worse than `DB X`. However, since short queries need only 2sec for execution on `DB X`, we can see that for most queries our model's plan is only

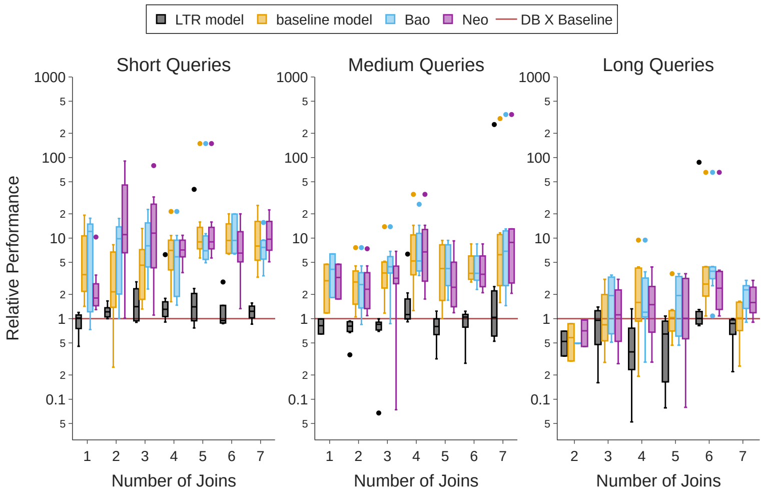


Abb. 7: Relative performance of our `LTR model` and the comparison models in relation to `DB X`. Values > 1 denote worse performance compared to `DB X` while values < 1 denote better performance.

slightly slower, i.e., the time difference lays within a second, while for the long queries our model can be 5 hours faster (4,015sec for `LTR model` compared to 22,292sec for `DB X`).

Similarly, Figure 8 shows the boxplots for 5GB of TPC-H data, while using 1GB for training. These plots emphasize that for most long-running queries and for many medium ones, our model is able to outperform `DB X`. The two queries for which our model's plans are significantly faster than `DB X`'s plans are a query with performance difference of 4,857sec and 1,467sec, respectively. For most queries where our model's plan is faster, `DB X` utilizes a stream aggregate with a pre-appended sort which slow down the query runtime.

4.3 Performance on Unseen Data

The goal of our solution is also to work with unseen data. To validate this, we load the IMDB dataset into `DB X` and execute 26 different queries, extracted from the same dataset. Again, we evaluate our model against `Bao`, the `baseline model` and `DB X`. Due to the fact that Neo's query featurization is tailored to the TPC-H dataset, we cannot use it for comparison. For the other models and `DB X`, we measure the plans runtime shown in Figure 9.

We observe that our `LTR model` and `DB X` have very similar performance for most queries, such as *14a* or *3a*. This is an astonishing result per se as our LTR model requires no manual tuning compared to the hours of human-engineered cost model of `DB X`. For some queries, such as 17a and 31a, our LTR model even manages to outperform `DB X` and achieve up to 5× better query performance. When analyzing our model's predicted plans, we observed

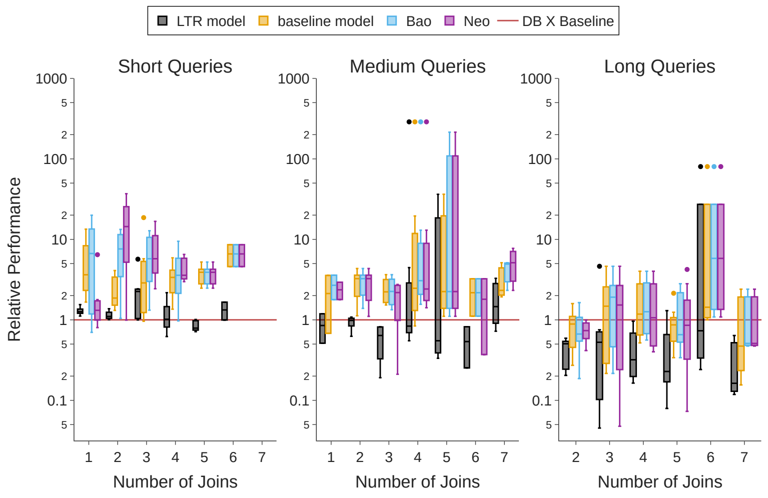


Abb. 8: Relative performance of queries on 5GB of data for our model and the comparison models to `DB X`. Our `LTR model` performs the best among all four models.

that our model predicts only hash joins. We assume that this might result from the nature of the IMDB data. In contrast to the data from the TPC-H benchmark, this dataset does not contain any indexes referencing to other tables which can be used to sort the table while scanning. Therefore, a merge join would always require at least one sort. We hypothesize that these sorts let the model predict a worse relevance score for queries with a merge join. Nevertheless, it seems that our model overfits slightly in favor of hash joins because it also does not predict any nested loop joins. However, in some cases `DB X` outperforms our model, e.g., for queries *20a* and *26a*. An analysis of the corresponding plans shows that `DB X` utilizes adaptive joins for these queries, a join algorithm we did not use while training because we were not able to produce valid plans with this join type. We, thus, consider the integration of adaptive joins future work. In addition, we observe that LTR performed slightly better than the baseline for most queries and for some of them (e.g., queries 17a, 5a, 6a) it was significantly better. Interestingly, we also observed that `Bao` predicts plans with a bad performance for almost every query. Looking into its predicted execution plans for queries where `Bao` timeouts, we realized that it always predicts bushy join trees with nested loop joins as well as many merge joins with preceding sorts for both relations. For most queries, this seems to be a bad combination.

In conclusion, our model's performance on unseen data is good even though it only predicts hash joins. We assume that including data of queries for other datasets as well as adaptive joins might solve its overfitting issue. Nevertheless, our model is able to predict proper execution plans on unknown data, and it predicts some better plans than `DB X` resulting to a performance increase up to 5×, while it does not require manual tuning.

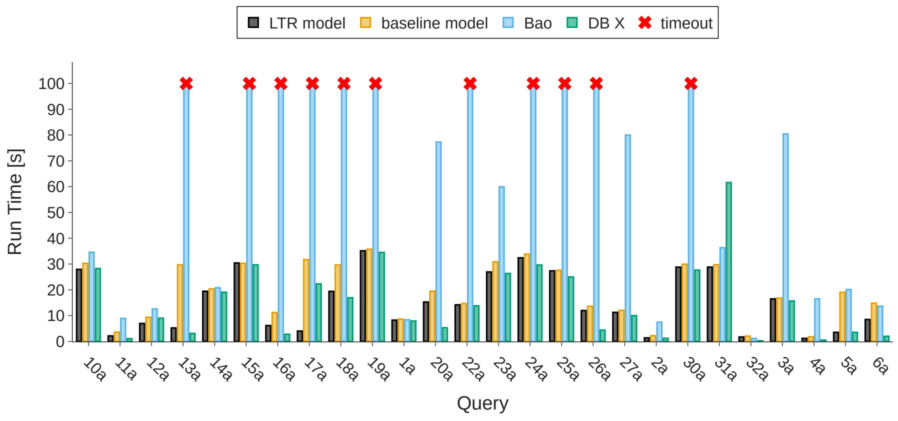


Abb. 9: Rutimes of chosen plans using the unseen IMDB dataset and JOB query benchmar.

5 Related Work

To our knowledge there is no work that leverages an LTR approach for query optimization. Works aiming at replacing a cost model with an ML model focus on estimating the execution time of plans, typically using regression models. Similarly, there are several efforts using ML for query performance prediction.

Early works that incorporated the use of ML for query performance prediction, mainly use a set of models, e.g., one for each operator, and aggregate them into a single cost. For example, [Ak12] uses different granularities for encoding and estimating plans, such as on plan-level, operator-level, and a hybrid approach. Compared to our solution, they not only use a different encoding strategy, but also different models for the plan-level and operator-level and not one model for the whole process.

A more recent work [MP19] for predicting query performance proposes the use of a modular principle to combine NNs with two hidden layers. Depending on the operator, the model utilizes a different neural network unit, whereas every unit produces an output with a similar structure to ensure consistency. The resulting deep NN represents the tree structure of the plan. The main difference to our solution is that the tree structure lies inside the model while we use it in the feature vector. Furthermore, the model building requires the execution time of every operator on its own, which leads to a more elaborated training data collection.

Neo [Ma19], a learned query optimizer, includes a model architecture for estimating the execution time of plans with NNs and uses a reinforcement approach for the join ordering. Their NN also uses tree convolution layers combined with a query vector, similar to our model approach, even though they do not use equivalent execution plans for their estimation. Another difference to our approach is that they do not inspect other important operators like aggregation or sort, only joins and scans and their featurization scheme is tight to the input dataset schema. Bao [Ma21], another learned query optimizer, uses a NN combined with reinforcement learning to predict the cost of an execution plan. However, the estimated cost is not used for selecting the best plan but for providing query hints to the database. While the NN layers are similar to our neural network, the featurization and the architecture of the model are different. In their features, they use no query encoding, which is also depicted in the architecture. Importantly, they do not use comparison queries during prediction.

[Ka20] proposes the use of ML, in particular a regression model, in the plan enumeration for predicting plan performance for the cross-platform system, Apache Wayang (incubating) (former Rheem) This work differs from ours not only because they use a vector representation and not a tree-based representation for the featurization but also because their focus is on performing the plan enumeration using entirely vectors.

6 Conclusions

We presented a learning-to-rank (LTR) approach for ML-based query optimization. Our proposal consists of (i) a novel listwise neural network architecture that considers not only one plan at a time but also its equivalent plans and a ranking-based loss function, (ii) two score functions that transform the runtime of the execution plans to a ranking score used as label, (iii) a featurization scheme for the execution plans that covers a wide range of operators and is oblivious to the dataset, and (iv) an adaptation of a bottom-up enumeration algorithm to work with our listwise LTR model. We evaluated our approach with a commercial database that includes a cost-based optimizer and two state-of-the-art regression-based ML approaches. Our results show that our LTR approach was able to match the performance of the commercial database for short and medium queries, while it outperformed it for long-running queries (up to 5×, leading to a runtime decrease of up to 5h). In addition, our approach systematically chooses better plans than state-of-the-art learning-based approaches, resulting in a performance speedup of over an order of magnitude.

Acknowledgments

This work was funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A).

Literaturverzeichnis

[Ak12] Akdere, Mert; Çetintemel, Ugur; Riondato, Matteo; Upfal, Eli; Zdonik, Stanley B.: Learning-based Query Performance Modeling and Prediction. In: ICDE. S. 390–401, 2012.

[Bu10] Burges, Christopher J.C.: From RankNet to LambdaRank toLambdaMART: An Overview. Learning, 2010.

[HB22] Hilprecht, Benjamin; Binnig, Carsten: Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. Proc. VLDB Endow., 15(11):2361–2374, 2022.

[Ka20] Kaoudi, Zoi; Quiané-Ruiz, Jorge-Arnulfo; Contreras-Rojas, Bertty; Pardo-Meza, Rodrigo; Troudi, Anis; Chawla, Sanjay: ML-based Cross-Plattform Query Optimization. S. 1489–1500, 2020.

[Ki19] Kipf, Andreas; Kipf, Thomas; Radke, Bernhard; Leis, Viktor; Boncz, Peter; Kemper, Alfons: Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In: CIDR. 2019.

[Li11] Liu, Tie-Yan: Learning to Rank for Information Retrieval. Springer-Verlag, 2011.

[Ma19] Marcus, Ryan; Negi, Parimarjan; Mao, Hongzi; Zhang, Chi; andTim Kraska, Mohammad Alizadeh; Papaemmanouil, Olga; Tatbu, Nesime: Neo: A Learned Query Optimizer. In: PVLDB. Jgg. 12, S. 1705–1718, 2019.

[Ma21] Marcus, Ryan; Negi, Parimarjan; Mao, Hongzi; Tatbul, Nesime; Alizadeh, Mohammad; Kraska, Tim: Bao: Making Learned Query Optimization Practical. In: SIGMOD. S. 1275–1288, 2021.

[MN06] Moerkotte, Guido; Neumann, Thomas: Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In: VLDB. S. 930–941, 2006.

[MP18] Marcus, Ryan; Papaemmanouil, Olga: Deep Reinforcement Learning for Join Order Enumeration. In: Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management. S. 1–4, 2018.

[MP19] Marcus, Ryan; Papaemmanouil, Olga: Plan-Structured Deep Learning Models for Query Performance Prediction. PVLDB, 12(11):1733–1746, 2019.

[Ne21] Negi, Parimarjan; Marcus, Ryan; Kipf, Andreas; Mao, Hongzi; Tatbul, Nesime; Kraska, Tim; Alizadeh, Mohammad: Flow-Loss: Learning Cardinality Estimates That Matter. PVLDB, 14(11):2019–2032, 2021.

[PGH18] Pfannschmidt, Karlson; Gupta, Pritha; Hüllermeier, Eyke: Deep Architectures for Learning Context-dependent Ranking Functions. arXiv preprint arXiv:1803.05796, 2018.

[SB13] Sasirekha, K.; Baby, P.: Agglomerative Hierarchical Clustering Algorithm - A Review. International Journal of Scientific and Research Publications, 3(3), 2013.

[Ve21] Ventura, Francesco; Kaoudi, Zoi; Quiané-Ruiz, Jorge-Arnulfo; Markl, Volker: Expand your Training Limits! Generating Training Data for ML-based Data Management. In: SIGMOD. S. 1865–1878, 2021.

[Wa18] Wang, Xuanhui; Li, Cheng; Golbandi, Nadav; Bendersky, Michael; Najork, Marc: The LambdaLoss Framework for Ranking Metric Optimization. In: Proceedings of the 27th ACM international conference on information and knowledge management. 2018.

[Ya19] Yang, Zongheng; Liang, Eric; Kamsetty, Amog; Wu, Chenggang; Duan, Yan; Chen, Xi; Abbeel, Pieter; Hellerstein, Joseph M.; Krishnan, Sanjay; Stoica, Ion: Deep Unsupervised Cardinality Estimation. 13(3):279–292, 2019.

[Yu20] Yu, Xiang; Li, Guoliang; Chai, Chengliang; Tang, Nan: Reinforcement Learning with Tree-LSTM for Join Order Selection. In: ICDE. S. 1297–1308, 2020.