PostgreSQL Query Optimization Using an Implementation of Lero

Sai Naga Viswa Chaitanya Basava, Shanmukha Bodapati, Rusheel Koushik Gollakota, Sree Harsha Koyi, Tarun Teja Obbina
The University of Texas at Dallas
{sxb220302, ssb180006, rxg220072, sxk230025, txo220011}@utdallas.edu

***Abstract*—**Advancements in Machine Learning and Deep Learning have led to a rise of their applications across the spectrum. In databases, Machine Learning based query optimization has become especially popular. We present our implementation of one such model: Lero, which is a pairwise ranking based query optimizer. Lero employs a binary classification task while pretraining on previous models to select the most efficient query execution plan among a diverse distribution of Candidate Plans. We show the effectiveness of Lero through experimentation and evaluation in which we conclude Lero performs about 65% faster than PostgreSQL's Native Query Optimizer. The code for our implementation can be found at: https://github.com/chaitanya-basava/lero.

I. Introduction

A. Research Issue

Query optimization, or efficiently finding an execution plan for a database query, has been a prominent research topic in both academia and industry. A properly optimized query can significantly reduce query execution speed, memory usage, disk reads, etc. Oftentimes the optimization process itself can be complex and computationally intensive. Therefore, a vast amount of research has gone into identifying algorithms and models which are able to generalize well across different query structures while requiring low resources. In this paper, we present and implement one such work.

We present our implementation with the goal of putting Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou's published work "Lero: A Learning-to-Rank Query Optimizer" [1] into practice. This research paper was presented at the 49th International Conference on Very Large Data Bases (PVLDB in February 2023). Precise latency values are the goal of traditional optimizers. However, this is a difficult and sometimes inaccurate operation. By emphasizing the relative ranking of execution plans, Lero presents a paradigm change that streamlines the procedure and might outperform the existing methods in terms of performance.

Given a set of query execution plans during optimization, the goal is to choose the most efficient plan. This was traditionally solved by evaluating the cost of each execution plan and selecting the one with the lowest cost. The work done by the original authors and our implementation deals with the high computational requirement issue by employing a pairwise-ranking model that rates each query rather than evaluating its cost [1]. Lero was shown to enhance query optimization quality quickly by experimenting with alternative optimizations and learning to rate them more accurately.

B. Contributions

Our group of 5 members worked over the course of 3 months to research, implement, evaluate, and present the Lero query optimizer. We divided the work so no one is overwhelmed, and we feel everyone contributed equally. Sai Naga Viswa Chaitanya Basava (sxb220302) worked on setting up the development environment and implementing the core algorithms. Sree Harsha Koyi worked with Chaitanya on the implementation and reviewed the code for identifying potential bugs. Rusheel Koushik Gollakota (rxg220072) led the literature review and gathered additional data sources for the implementation and experimentation phases. Tarun Teja Obbina (txo220011) worked on setting up the development environment scripts, helped write the project report, and carried out the experiments using the training model. Shanmukha Bodapati (ssb180006) conducted experiments using the training model and wrote the project report.

II. Related Work

Traditionally, query optimization revolves around heuristic-based models, or cost-based techniques. These traditional models aim to find a plan with the lowest estimated cost based on execution and resource complexities. [2] A cardinality estimator approximates the number of tuples in the output for each subquery. Then, a cost model estimates the overall cost for each viable execution plan, generally using the results from the cardinality estimator. Finally, the plan enumerator selects a viable plan with minimal cost. [2]

Heuristic-based models exist with extensive practical applications, are highly reliable with most standard query patterns, and require a very low computational overhead. However, current query styles can be highly complex and dynamic, and heuristic-based models fail to adapt accordingly. This is exacerbated by a high dependency on magic constant numbers which are used to align estimated costs with actual performance. [2]

As a result, recent query optimization algorithms adopt Machine Learning (ML) based models. These models utilize the power of ML frameworks to improve the generalization drawbacks of heuristic-based models. In June 2021, Marcus *et al.* presented the Bandit Optimizer (Bao) [3] which utilizes a combination of tree-based convolutional networks and a reinforcement learning algorithm known as Thompson sampling. With Thompson sampling [4], Bao adapts to the dynamic nature of modern queries and data distributions using contextual multi-armed bandits. Similarly, the Neural Optimizer (Neo) [5] also by Marcus *et al.* in July 2019 uses reinforcement learning to optimize the join order, operator selection, and index choices during query execution. Both Bao [3] and Neo [5] work well with generalization to different data distribution and have low tail execution latency. However, there is a very high and infeasible computational overhead due to their complex structure. Furthermore, they require extensive training data to be trained properly.

A rise in the research of Deep Learning (DL) frameworks led to the emergence of DL-based models. In July 2021, Liu *et. al.* introduced Fauce [6] which incorporates uncertainty information to capture correlations across all columns and tables in a database. Earlier in September 2020, Yang *et al.* introduced NeuroCard [7], an autoregressive model which uses joint sampling to build one neural density estimator over an entire data distribution. Another DL-based model was introduced in June 2019 again by Yang *et al.* [8] The NaRu, or Neural Relation Understanding, model approximates the joint data distribution to a somewhat high accuracy. Both NeuroCard [7] and NaRu [8] ignore all column independence assumptions which allows for a faster approach in determining and understanding the correlation between multiple tables. However, while comparatively more efficient than previous models, both NeuroCard [7] and NaRu [8] require a lot of intensive computational resources, even more than ML models. Also, reliability is limited since the models are still in the development phase.

Compared to traditional heuristic-based models and ML/DL-based models, our implementation of Lero is far more accommodating of computational constraints. Lero's unique method of ranking queries rather than estimating the runtime and cost of queries being compared significantly reduces training time and space requirements. [1] This combined with its pairwise, or binary classification, approach over two-plan embeddings with shared parameters, Lero excels at keeping the structural properties of the query. As we show in this paper, our model distinguishes itself with its adaptability and balanced approach to query optimization. Instead of creating a completely new optimizer, Lero also builds upon the existing decades of query optimization wisdom. [1]

III. Methodology

As mentioned earlier, Lero is unique in its approach to constructing a ranking-based model rather than predicting the exact cost of execution for every query plan. This relies on the hypothesis that while perfect latency predictions are ideal for accuracy, there is a heavy drawback through computational complexity. Instead, understanding the relative efficiency of different executions proves to be more practical and beneficial. Lero compares pairs of models to determine the optimal execution plan. This results in a lightweight model while not compromising on accuracy.

A. Lero Workflow

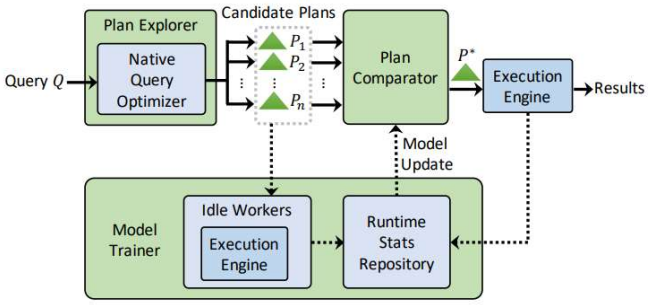

Fig. 1. Lero Workflow

As shown in *Figure 1*, Lero starts by sending a Query Q to the Plan Explorer. [1] The Plan Explorer utilizes the Native Query Optimizer to generate n Candidate Plans offering viable and diverse options to execute the query in the database. Here, the cardinality estimator is up/downscaled to generate a variety of plans. Variation in the execution plans is crucial to high adaptability and generalization of the model. The Candidate Plans are fed to the Plan Comparator model which works to pairwise compare two plans based on their latencies resulting in a binary classification output as shown in Equation 1.

$$\text{CmpPlan}(P_1, P_2) = \begin{cases} 0 \text{ if } \text{Cost}(P_1) < \text{Cost}(P_2) \\ 1 \text{ if } \text{Cost}(P_1) > \text{Cost}(P_2) \end{cases}, \quad (1)$$

The Model Trainer runs parallel in the background when resources are available and executes candidate plans to find latency costs which are stored in the Runtime Stats Repository. [1] The Model Trainer also periodically updates the Plan Comparator model for training and improving generalization and accuracy. This parallelized approach saves a large amount of computational resources while aiding in the training of the Plan Comparator Model. Therefore, Lero's primary workflow consists of generating candidate plans through the Plan Explorer and selecting optimal plans via the Plan Comparator model while executing and evaluating plans in parallel to train the Plan Comparator model. [1]

B. Plan Explorer Algorithm

The purpose of the Plan Explorer's creation of Candidate Plans is twofold: 1) to generate enough viable Plans so the Comparator model can select the execution plan with the minimum execution time, and 2) to diversify the distribution of Candidate Plans to improve training the Plan Comparator model and as a result the overall generalization of the model. [1] Unlike previous models, which use a basic random strategy to generate Candidate Plans like in Neo or a Boolean-flag set-based strategy like in Bao, Lero utilizes a cardinality estimator. As mentioned earlier, the cardinality estimates are scaled up or down to diversify the Candidate Plans. While traditionally a standard cardinality estimator is used to generate Plans, Lero uses modified (scaled) cardinality estimators to generate Candidate Plans. [1] In most cases, and for our implementation, only one cardinality estimator is used with tunable parameters to mock a unique cardinality estimator and a unique result.

For the second task, more sophistication is necessary. A brute-force diversification (or exploration as termed by the authors of Lero) algorithm tries all possible combinations for the parameters from a set of scaling factors to generate valid Plans(the query can be executed accurately and without error using the Plan). [1] However, this method is computationally intensive, would result in far more Plans than required, and may even lead to overfitting. Instead, Lero utilizes a priority-based heuristic method in which more importance is placed on conditions in which the Native Query Optimizer is likely to make a mistake. [1] More importance is placed on the mistakes in estimating cardinalities sub-queries of size k on k tables.

Algorithm PlanExplorer(*Query Q, α,* Δ):
1: PriorityQueue *candidatePlans* ← []
2: **for** each $\boldsymbol{f} \in \boldsymbol{F}_{\boldsymbol{\alpha}}^{\Delta}$ in the increasing order of $|\log f|$:
3: **for** k ← 1 to q:
4: **Native Query Optimizer:**
5: Let C be the default cardinality estimator
6: $\widetilde{\boldsymbol{C}}(\boldsymbol{Q}') \leftarrow \boldsymbol{f} \cdot \boldsymbol{C}(\boldsymbol{Q}')$ for sub-queries q of size k
7: $\widetilde{\boldsymbol{C}}(\boldsymbol{Q}') \leftarrow \boldsymbol{C}(\boldsymbol{Q}')$ for sub-queries q not of size k
8: feed $\widetilde{\boldsymbol{C}}$ into cost model to generate plan P
9: *candidatePlans* ← *candidatePlans* ∪ {*P*}
10: **return** *candidatePlans*

Fig. 2. Plan Explorer Algorithm

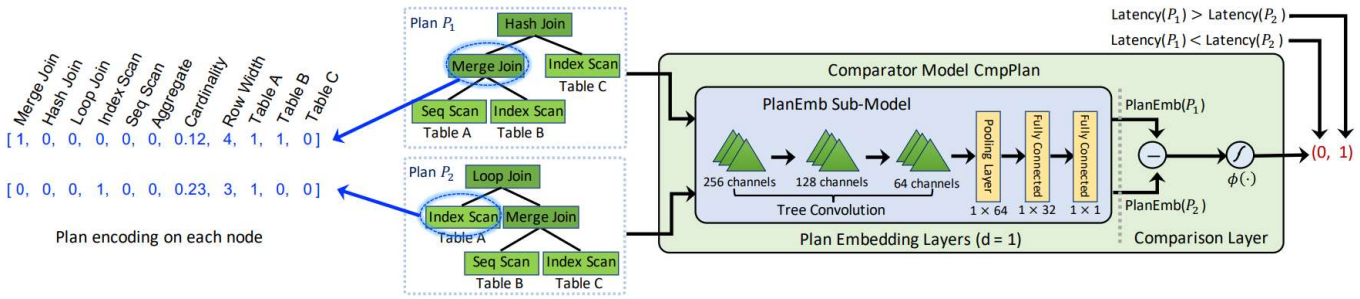

Fig. 3. Plan Comparator Algorithm

As shown in *Figure 2*, a priority queue data structure is maintained to keep track of Candidate Plans. [1] Then, a double-nested loop goes through a combination of scaling factors $\boldsymbol{f} \in \boldsymbol{F}_{\boldsymbol{\alpha}}^{\Delta}$ in $|\log f|$ increments and k, or number of tables, from 2 to q, the total number of tables in query Q. This restricts, or more accurately focuses, the scope of the Plans to those that are close enough to the original Plan but diverse enough that the Native Query Optimizer is likely to make a mistake. More specifically, within the inner loop, a default cardinality estimator is initialized and then scaled by scaling factor f for every subquery Q' of size k in Q. For the remaining sub-queries (*ie.* not of size k), the unscaled cardinality estimator is used. This is combined into a single cardinality estimator which is sent to be a cost model to generate Plan p which in turn is added to the priority queue. For further diversification, a mixture of different query tree shapes (*ie.* left-deep, right-deep, bushy, etc.) and join orders. These diversified Candidate Plans generated because of the Plan Explorer are sent to the Plan Comparator.

C. Plan Comparator Algorithm

In the Plan Comparator, the Candidate Plans are first converted into feature vectors represented through One-Hot Encoding of query operations. [1] Then, the vectors are normalized through Min-Max Normalization and are converted into Embeddings through a series of transformations to extract as much information from the Plans as possible. Embeddings are *1-dim* and are transformed through a tree-based convolutional neural network as shown in *Figure 3*. The network starts with 3 sets of convolutional layers with 256, 128, and 64 filtering channels respectively. Then, a 1x64 Pooling Layer is used, followed by two Fully Connected Layers of size 1x32 and 1x1. Notice how the dimensionality of the Embeddings is flattened gradually until the final *1-dim* vector remains, which is to be interpreted as ranking information. [1]

The final Comparison Layer involves two models: *PlanEmb* and *CmpPlan* which relies on *PlanEmb* used on two Plans. [1] This results in a pairwise comparison of the models through a binary classification task as follows:

$$\text{CmpPlan}(P_1, P_2) = \phi\big(\text{PlanEmb}(P_1) - \text{PlanEmb}(P_2)\big), \quad (2)$$

The difference of *PlanEmb* on the two Plans is fed to a logistic activation function (whose output is between 0 & 1), resulting in the output of *CmpPlan*. This result effectively translates to how likely one Plan is better than the other. More formally:

$$\text{CmpPlan}(P_1, P_2) = \begin{cases} 0 \text{ if } \text{PlanEmb}(P_1, Q) \ll \text{PlanEmb}(P_2, Q) \\ 1 \text{ if } \text{PlanEmb}(P_1, Q) \gg \text{PlanEmb}(P_2, Q)' \end{cases} \quad (3)$$

Through this binary classification task and passing all the Plans through CmpPlan, the Plan with the smallest value of PlanEmb is selected. [1]

D. Loss Function

Because this is an ML task, or more accurately a supervised learning task, there must be a Loss Function utilized in coordination with an error function. [1] We use Cross-Entropy Loss as a basis which is standard for classification tasks. Our overall goal is to maximize the likelihood that the correct Plan (the one with smaller value of *PlanEmb*) is selected between any two Plans. We let A be a randomized algorithm to select the best Plan out out P1 and P2 as follows:

$$\boldsymbol{A}(P_1, P_2) = \begin{cases} P_1 \text{ with probability } 1 - \text{CmpPlan}(\text{P}_\text{i}, \text{P}_\text{j}) \\ P_2 \text{ with probability } \quad \text{CmpPlan}(\text{P}_\text{i}, \text{P}_\text{j})' \end{cases} \quad (4)$$

After applying this to all Plans, the probability of making no mistake is:

$$\prod_{Q \in W} \Big(\prod_{P_i < P_j \in P(Q)} (1 - \text{CmpPlan}(P_i, P_j)) \cdot \prod_{P_i > P_j \in P(Q)} \big(\text{CmpPlan}(P_i, P_j)\big) \Big), \quad (5)$$

After transforming by *-log*, we reach:

$$\sum_{Q \in W} \Big(\sum_{P_i, P_j \in P(Q)} (\mathbb{I}_{P_i < P_j} \cdot log\big(1 - \text{CmpPlan}(P_i, P_j)\big) + \mathbb{I}_{P_i > P_j} \cdot log\big(\text{CmpPlan}(P_i, P_j)\big)\Big), \quad (6)$$

where $\mathbb{I}$ is the indicator function.

E. Overall Analysis

Lero's workflow and primary algorithms of the Plan Explorer and Plan Comparators are what sets it apart from other traditional and ML-based query optimization models. Its pretraining stage (done outside of our implementation) already gives it a competitive edge in learning from the pitfalls of previous models, but it's parallel execution and training of Plans with the Model Trainer and Model Comparator significantly reduce training time which set it apart from all other models. [1]

Using the cardinality estimator and native query optimizer with the Plan Explorer allows Lero to be platform independent and easier to implement since minimal configuration is needed. [1] Furthermore, while a brute force exploration strategy would result in $O(log_{\alpha}^{2^q}\Delta)$ Candidate Plans generated, its streamlined priority-based heuristic strategy generates only $O(q \cdot log_{\alpha}\Delta)$ Plans which are both valid and diversified. [1] This is a massive jump in the efficiency of space and time resources.

F. Implementation

For this paper, the Lero model is implemented using Python, bash, and C programming languages and a patched version of PostgreSQL (*v13.1*) [9] provided by the authors of Lero. [1] We utilize PyTorch [10] , an ML framework for ML/DL-based tasks such as binary classification, and NumPy [11], a mathematical Python library for computationally intensive array processing. All work is developed in the Jupyter Notebook, Visual Studio Code, and IntelliJ programming environments while all training and experimentations are conducted locally on a 2019 MacBook Pro. The algorithms, workflow, and strategies outlined in the sections above are replicated in our implementation. [1]

IV. Experimentation & Results

The primary goal of our experimentation and evaluation is to determine how well our implementation of Lero compares to the performance of Lero as implemented by the authors and how well it performs against PostgreSQL's Native Query Optimizer. [1, 10] In this section, we give an overview of our experimentation and evaluation processes and present their results.

A. Experimentation Conditions

All experiments were conducted locally on a 2019 MacBook Pro with an Intel chip. We utilized the TPC-H [12] benchmark dataset which contains 270 queries generated and augmented by the original authors through a uniform distribution. The queries are varied in structure but do not include highly simple queries (those that only require one or two tables). The queries are also not related to views or nested as the original Lero implementation is not compatible with those formats. [1] We ran the TPC-H benchmark dataset under the conditions listed above to extract the performance of our implementation of Lero and compare it against PostgreSQL's query optimizer. [12]

B. Key Findings

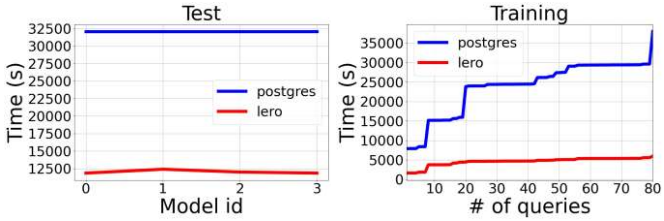

Fig. 4. Lero vs PostgreSQL query optimizer performance

As Figure *4* shows, Lero outperforms the standard PostgreSQL optimizer across all training queries. Furthermore, notice the performance is consistent across the spectrum as more queries are added. There are no spikes whereas the performance of the PostgreSQL query optimizer contains several spikes in the plot of elapsed time for query execution. This shows there are certain queries for which the PostgreSQL query optimizer takes longer than Lero to optimize and execute the query. This spike is most likely a result of an unusual, unique, or complex query structure. However, Lero does not

falter and behaves consistently even with a varied query style, further proving Lero's strength of generalization. This is also backed up by the consistency present in the test models with Lero-based optimization taking only about 65% of the time as the PostgreSQL optimizer. For comparison, the authors determined Lero completed its tasks in approximately 70% of time as the PostgreSQL optimizer. We can effectively conclude that our implementation is on par with the original authors, with differences attributed to processor conditions.

V. Conclusion

We describe our implementation of Lero, a query optimization algorithm through ranking the effectiveness of different plans in executing a query. [1] Lero diminishes the problems of generalization and computational overhead present in traditional heuristic-based methods and other ML/DL-based models. It is also pretrained against the pitfalls of earlier models to give an extra accuracy boost. [1]

Our implementation uses a pairwise, or binary classification, approach to rank generated execution Plans against each other to determine the Plan with the minimal cost. [1] Lero can generate valid and diverse Candidate Plans through our Plan Explorer Algorithm and compare/rank those Plans through the Plan Comparator model. Lero simultaneously executes generated plans to calculate their cost and provides training information to the Plan Comparator to enhance the model. [1]

Through our experimentation and evaluation process, we determined the strength of Lero in performing better than the Native Query Optimizer of PostgreSQL v13.1. [9] We found Lero was about 65% more efficient and generalized more consistently than PostgreSQL's query optimizer. We were able to carry out the core systems, algorithms, and experiments described in the paper and are satisfied with our results to determine that our implementation matches the benchmarks set by the original paper.

VI. References

[1] Zhu, R., Chen, W., Ding, B., Chen, X., Pfadler, A., Wu, Z. and Zhou, J., 2023. Lero: A Learning-to-Rank Query Optimizer. arXiv preprint arXiv:2302.06873.

[2] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. 1979. Access path selection in a relational database management system. In Proceedings of the 1979 ACM SIGMOD international conference on Management of data (SIGMOD '79). Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[3] Marcus, R., Negi, P., Mao, H., Tatbul, N., Alizadeh, M. and Kraska, T., 2021, June. Bao: Making learned query optimization practical. In Proceedings of the 2021 International Conference on Management of Data (pp. 1275-1288).

[4] Thompson, W.R., 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. Biometrika, 25(3-4), pp.285-294

[5] Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O. and Tatbul, N., 2019. Neo: A learned query optimizer. arXiv preprint arXiv:1904.03711

[6] Liu, J., Dong, W., Zhou, Q. and Li, D., 2021. Fauce: fast and accurate deep ensembles with uncertainty for cardinality estimation. Proceedings of the VLDB Endowment, 14(11), pp.1950-1963.

[7] Yang, Z., Kamsetty, A., Luan, S., Liang, E., Duan, Y., Chen, X. and Stoica, I., 2020. NeuroCard: one cardinality estimator for all tables. arXiv preprint arXiv:2006.08109

[8] Yang, Z., Liang, E., Kamsetty, A., Wu, C., Duan, Y., Chen, X., Abbeel, P., Hellerstein, J.M., Krishnan, S. and Stoica, I., 2019. Deep unsupervised cardinality estimation. arXiv preprint arXiv:1905.04278.

[9] E.13. release 13.1. PostgreSQL Documentation. (2023, November 9). https://www.postgresql.org/docs/13/release-13-1.html

[10] PYTORCH documentation¶. PyTorch documentation - PyTorch 2.1 documentation. (2023). https://pytorch.org/docs/stable/index.html

[11] Numpy reference#. NumPy reference - NumPy v1.26 Manual. (2023, September 16). https://numpy.org/doc/stable/reference/index.html

[12] Transaction Processing Performance Council(TPC). 2021. TPC-H Vesion 2 and Version 3. http://www.tpc.org/tpch/.