Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective

Claude Lehmann*
claude.lehmann@zhaw.ch
Zurich University of Applied Sciences
Winterthur, Switzerland

Pavel Sulimov*
pavel.sulimov@zhaw.ch
Zurich University of Applied Sciences
Winterthur, Switzerland

Kurt Stockinger
kurt.stockinger@zhaw.ch
Zurich University of Applied Sciences
Winterthur, Switzerland

ABSTRACT

The current boom of learned query optimizers (LQO) can be explained not only by the general continuous improvement of deep learning (DL) methods but also by the straightforward formulation of a query optimization problem (QOP) as a machine learning (ML) one. The idea is often to replace dynamic programming approaches, widespread for solving QOP, with more powerful methods such as reinforcement learning. However, such a rapid "game change" in the field of QOP could not pass without consequences - other parts of the ML pipeline, except for predictive model development, have large improvement potential. For instance, different LQOs introduce their own restrictions on training data generation from queries, use an arbitrary train/validation approach, and evaluate on a voluntary split of benchmark queries.

In this paper, we attempt to standardize the ML pipeline for evaluating LQOs by introducing a new *end-to-end benchmarking framework*. Additionally, we guide the reader through each data science stage in the ML pipeline and provide novel insights from the machine learning perspective, considering the specifics of QOP. Finally, we perform a *rigorous evaluation of existing LQOs, showing that PostgreSQL outperforms these LQOs in almost all experiments depending on the train/test splits.*

PVLDB Reference Format:
Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. PVLDB, 17(7): 1565 - 1577, 2024.
doi:10.14778/3654621.3654625

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/edualc/lqo_ml_perspective.

1 INTRODUCTION

Over the last decade, machine learning (ML) approaches have heavily dominated classical query optimization methods. Having in total $O(n!)$ possible logical plans in the worst case for queries where the join graph is a clique with n tables, the problem is classified as NP-hard [37]. This implies that exhaustive methods cannot solve the problem for a higher order of joins[1], thus demanding the need for heuristical approaches.

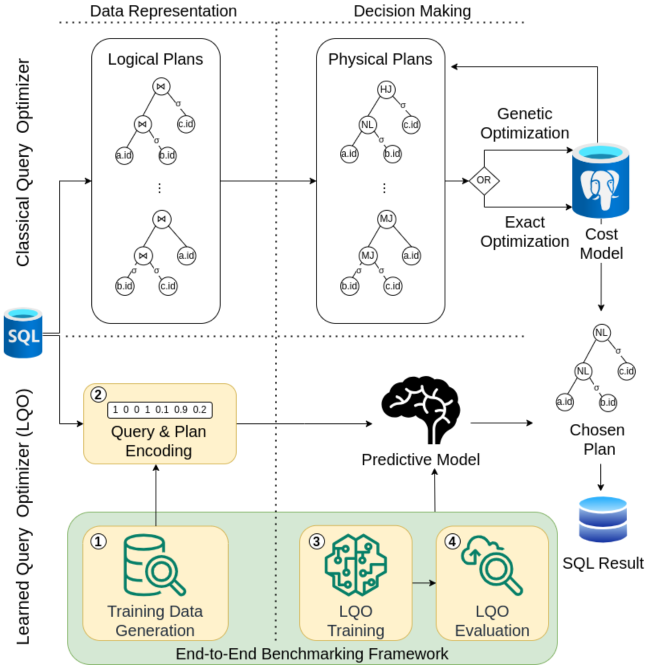


Figure 1: Comparison of classical and learned query optimizers (LQO) - see top and bottom halves, respectively. The stages (1) Training Data Generation, (3) LQO Training, and (4) LQO Evaluation are the primary components of our End-to-End Benchmarking Framework. Together with the (2) Query & Plan Encoding stage, they form the typical machine learning pipeline for a LQO.

In Figure 1, we compare typical pipelines for classical and learned query optimizers. The *classical approach*, implemented inside database management systems (DBMS), has the stages of query representation via logical and physical plans, with a follow-up search of an optimal plan using cardinality-based cost model estimations. In addition to dynamic programming-based methods, genetic algorithms [34] are also used since they are proven to be more efficient for queries with a high number of joins [28].

The bottom part of Figure 1 shows *learned query optimizers* (LQO), the most recent trend for end-to-end query optimization. These approaches require a more complicated pipeline because of

*Both authors contributed equally to this work.

Proceedings of the VLDB Endowment, Vol. 17, No. 7 ISSN 2150-8097.
doi:10.14778/3654621.3654625

[1] PostgreSQL abandons exhaustive methods for queries with 12 or more `FROM` items.

the use of ML methods. Looking at it from the *ML perspective*, the pipeline should consist of several stages: (1) training data generation, (2) query & plan encoding, (3) ML model training, and (4) ML model evaluation. The violation of theoretical ML principles [30] at each stage and the absence of a unified reproducible framework make it currently *impossible to fairly compare the results of LQOs.*

Let us briefly describe what can go wrong at each stage, i.e., the major challenges of the ML pipeline for LQOs from both a data science and an engineering perspective and how we solve them as contributions of this paper.

Training Data Generation.[2] *When no ready-to-use training data is provided for benchmarking, opportunities for biased data creation appear.* For LQOs, we observed that only the queries are given as SQL statements for popular benchmarks such as JOB [18]. The key problem is that these statements cannot be explicitly used as input for ML models without querying the databases (DB) and extracting metadata such as cardinalities or execution times. This implies a gap between the given benchmark data and the actual features used to train LQOs, which are strongly correlated with the parametric conditions when querying the database.

Contribution: *We discuss general limitations that could hamper the process of similar training data creation in Section 3.*

Query & Plan Encoding. *Encoding the queries such that the principle of invariance*[3] *is broken, leads to inconsistencies in the performance.* When different queries are encoded using column selectivities, it is possible that large sections of the encoding (or even the full vectors) are identical. This is because many filter combinations result in the same selectivity. Hence, the model would potentially suffer from the mismatch between features and target variables and will only perform well if this inconsistency is mitigated.

Contribution: *We diagnose the invariance issues in particular methods and give encoding recommendations in Section 4.*

LQO Training. *Contravening the basic training techniques and misapplying mathematical models makes your ML model behave unexpectedly.* Complicated DL models are hard to train, which makes hyperparameter tuning and validation procedures the cornerstone for gaining high predictive power. Moreover, injecting additional mathematical mechanisms can have adverse side effects that negatively impact the training itself and, in turn, the query performance.

Contribution: *We propose enhancements to make the training process of LQO methods more stable and reliable in Section 5.*

LQO Evaluation. *When your model is trained and then evaluated on a non-fixed train/test split, comparisons become data-centric rather than model-centric*, i.e., the choice of the train/test split strongly correlates with the model's performance. For example, the performance on two different train/test splits of the same type (such as randomly splitting queries) is not comparable. Explicit examples of this can be seen in Figures 4 and 5 of our experiments). Despite the existence of public query optimization benchmarks like the one in [18] and [22], it remains an open question which queries serve as the train data and which ones are the test data. The attempts to suggest the procedure of a unified evaluation were only made recently in [23], [39] and [43].

Contribution: *We unify data splitting for LQOs and introduce a procedure to test different levels of generalization in Section 6.*

Reproducibility. *Any ML method developed in academia has negligible practical value if it cannot be reproduced on arbitrary software and hardware.* With ML approaches finding widespread use in academic research, navigating the realm of learned query optimization presents challenges, as it requires proficiency in the core subject of database research and numerous related engineering fields. These approaches typically require complex programming code and (ML) models with inherent stochasticity. Hence, reproducibility is becoming a growing concern in academia.

Contribution: *We suggest an* ***End-to-End Benchmarking Framework*** *- a novel meta-benchmarking framework that is capable of equalizing the conditions under which the ML-based LQOs are trained and tested, guaranteeing consistency in comparisons in Section 7.*

Main contribution: We perform an extensive evaluation of existing LQOs using our end-to-end benchmarking framework in Section 8. **Our results demonstrate that current LQOs do not systematically perform better than PostgreSQL.** These findings indicate that *novel research is required to make LQOs competitive* with more traditional approaches - and not only in specific cases.

The paper is organized as follows: First, we briefly review recent LQO methods in Section 2. Then, we dissect the data science stages in the ML pipeline applied to query optimization, not only discussing potential hurdles that can occur while processing each stage but also suggesting ways of mitigating them via practical (see Sections 3 & 6) and theoretical (see Sections 4 & 5) recommendations. Based on the challenges in the reproducibility of ML approaches, we propose our End-to-End Benchmarking Framework in Section 7. Afterwards, we perform an elaborate experimental evaluation of recently released LQOs from an ML perspective in Section 8. Finally, we conclude the paper in Section 9.

2 RELATED WORK

Before end-to-end LQOs appeared, significant progress had been made toward using modern ML approaches for query optimization. For instance, DQ [15], ReJOIN [24, 25], and others [11, 16] apply reinforcement learning (RL) in an exploration-exploitation strategy with the goal of finding the optimal join order. These methods use a cost model to produce a "join score" reward for the learning agent.

The first end-to-end LQO Neo [23] uses a neural network (NN) to estimate the latencies of a full query plan given a sub-plan as an input. The optimal plan is predicted via a greedy tree search in the join and scan space and consecutive bottom-up plan construction.

RTOS [42] assumes that the join graph is built as a sequence of join operations between two tables, ignoring scans, and applies a graph NN to train an RL agent. The predicted query plan is built similarly to Neo, though it applies a depth-first search.

Bao [22] sits on top of the PostgreSQL query optimizer, controlling the execution flow by enabling or disabling a subset of join and scan operations. These subsets are referred to as hint sets, and Bao provides neither the full join order nor which scan types are used for which table but rather advises which operations not to use.

Balsa [39] is based on the same architecture as Neo. However, it introduces several modifications to the training pipeline: it pretrains using the cost model estimations of a DBMS instead of real

[2] The points about training data also apply to validation and test data.

[3] We formulate the principle of invariance [17] in data generation as follows: When given the same input, the data generating system should return the same output.

latencies, it uses timeouts during query executions, and it does not sample training data from the replay buffer but rather uses the data points produced by the most recent NN state.

Lero [44] formulates the problem as a learning-to-rank (LTR) task and generates various candidate query plans from the DBMS by changing the internal cardinality estimations. The plan comparator module selects the better of two generated candidate plans, similarly choosing the optimal plan during inference. LEON [4] is another LTR method. Unlike Lero, it brute-forces many possible physical plans in a dynamic programming manner and prunes them before training. Training happens only on the top chosen SQL/query plan-pairs, ranked by their latency and posterior uncertainty estimation obtained from a Bayesian NN.

LOGER [3] uses the conceptual ML model pipeline from RTOS, though extending the action space for join order recommendations by adding the join type. LOGER restricts the operation recommendation, i.e., which join type not to use, by applying ϵ-beam search for plan prediction.

HybridQO [41] uses a mix of cost and latency estimations, like some other methods, but in a different manner: it first gets the candidate plans from the DBMS via hints. Those hints are obtained from the top levels of the query plan tree explored by a Monte-Carlo Tree Search (MCTS) with an upper confidence bound and using the cost as a target (the cost is estimated with an NN from RTOS). Then, the same network architecture is used to predict the latency and uncertainty from the candidate plans. A multi-head performance estimator makes the final plan selection.

In the recent paper [43], the authors question the reasonability of training complicated and computationally costly LQOs. As an alternative, they suggest the combination of look-ahead information passing (LIP), in which adaptive semi-join techniques and adaptive join algorithms (AJA) are used. The latter checks whether a hash join should be replaced by a nested loop join at runtime.

In this paper, we introduce neither a new LQO nor a classical alternative. Instead, we provide *recommendations to improve LQOs* based on a *vast evaluation of existing LQOs from an ML perspective.*

3 TRAINING DATA GENERATION

Typically, ML problems have publicly available benchmarks with ready-to-use training data that is identical for all participants. QOP benchmarks differ regarding the provided data and only serve as a source for generating the training data, suitable as an input into ML models. This makes the whole ML pipeline vulnerable to inconsistencies in the data generation process, namely:

(1) Having training data generated under unreasonable restrictions reduces the domain of data points available for training and potentially decreases the generalization of the ML models. (2) The generated training data can result in cases where the same input leads to a different output (or target).

In this section, we first explain the choice of the benchmark and then discuss the issues around generating the training data from it.

3.1 Dataset Choice

We use the JOB [18] and STACK [22] benchmarks for all the experiments in this paper. Sourcing data from the IMDB and Stack-Exchange, respectively, both datasets reflect natural challenges in real-world workloads. Moreover, a recent paper [44] claims that the JOB benchmark is the most challenging one for LQOs, and the majority of current methods use JOB and/or STACK. We do not use the STATS-CEB benchmark suggested in [10], as it was originally developed for challenges in cardinality estimation as opposed to end-to-end query optimization, which is the focus of this paper. We also do not use the TPC benchmark family [35], as it has underlying assumptions of multivariate uniformity, which does not create reasonable challenges for LQO methods.

3.2 Reduced Complexity of Query Plans

During our evaluation of LQOs, we noticed that some authors suggest severely reducing the number of possible physical plans by, for example, disabling nested loop joins (as has been done in [18]). This might yield improvements for some queries but solves the query optimization challenge by using a data-dependent solution at the cost of reduced generalizability. In general, we observe from some queries that limitations such as disabling specific scan or join methods, non-exact optimization, or join tree types lead to a possible increase in the chances of finding a sub-optimal plan.

E.g. the PostgresPro Community [29] discussed that any of the join methods could have an advantage over others depending on the selectivity of subqueries. The authors of [22] show experimentally that disabling nested loop joins in PostgreSQL can improve the performance of query 16b or harm the performance of query 24b.

For bitmap and tid scans, the Genetic Query Optimizer (GEQO), and bushy trees, we provide extensive experiments producing the counter-examples in Sections 8.4, 8.5 and 8.7 respectively.

3.3 Invariant Training Data Generation

The data used as an input into LQO ML models, which all have either a reward or a prediction value, has a canonical view of (D, y)-pairs: D refers to the vector of *feature variables*, consisting of either an independent set of variables X for supervised methods, or $(s, a, s^{'})$ - a set of *state*, *action* and *next action*, respectively, for RL methods. y is a *target variable*, which is either the *query latency, cost*, or the *ranking* depending on the ML model used.

In this subsection, we discuss why both types of variables are subject to the absence of invariance during training data generation.

3.3.1 *Feature Variables: Dynamic Optimization.*

The vast majority of LQOs use the pg_hint_plan extension [27] to *force PostgreSQL to execute an explicit query plan* rather than using a plan predicted by the built-in query optimizer.

However, one should not expect that a plan with its hints is really executed. This is due to the dynamic updates of the plan during execution [1], referred to as *dynamic optimization.* All the LQOs we evaluated force the DBMS to execute their plans during the stage of plan encoding, hence potentially training on incorrect data.

Dynamic optimization could also be the reason for a possible discrepancy between the executed plan and the output provided by `EXPLAIN`. This means that LQOs, which rely on the cardinality estimations from `EXPLAIN`, potentially introduce significantly inaccurate estimations.

Recommendation: This could be mitigated via a *direct RL approach*, where the DBMS is treated as a "black box". The objective function is directly maximized via gradient descent without the need to learn transition probabilities (i.e., the stochastic behavior of the DBMS) and without the need to solve Bellman equations [8].

3.3.2 Dependent Variables: Cold vs. Hot Cache. If a query is executed several times, the executing time decreases due to reading pre-calculated information from previous runs (hot cache) instead of creating everything from scratch (cold cache). We want to create a situation that yields comparable and consistent results for every query. Hence, the cache status should be either fully cold or hot, i.e., when all potential caching has been performed. No intermediate "warm" cache should be allowed.

However, it is unreasonable to expect a full cold cache situation [19]. Moreover, it is an ethical question if it is fair enough to run queries with a cold cache, considering that it disables all the optimization techniques that the DBMS has based on cache buffers.

Recommendation: Taking into account potential correlations of queries inside workloads like JOB due to the use of base templates/patterns, we believe that *forcing a hot cache setting is fairer*, as it mitigates the influence of previously executed queries on the execution time of any particular query. The way of achieving a hot cache setup is discussed in detail in Section 7.3; conceptually, it is a consecutive run of the same query until the latency converges.

4 QUERY & PLAN ENCODING

In this section, we discuss which information can be extracted from SQL queries and their physical plans as input to the ML model. Moreover, we explain which principles should be followed so that LQO models are trained smoothly.

The recent LQOs, to the best of our knowledge, are all *query-driven methods* in contrast to data-driven methods used for cardinality estimation [12, 40]). In other words, LQOs use queries as an indispensable proxy to the data underneath the DBMS. It implies that the encoding schema for a query should be both expressive and robust. We will now discuss the *principles of encoding robustness and expressiveness* and how we can achieve them.

4.1 Encoding Robustness

Table 1 gives an overview of the main encoding components used by various LQOs. Note that we distinguish between *query encoding* (information that is independent of how the query is executed) and *plan encoding* (information based on the physical plan). For instance, the text attributes of the query can either be encoded based on their cardinality or by using e.g. word2vec to generate a vectorized form. Moreover, encodings can be aggregated using either stacking or pooling, sometimes with additional post-transformations.

We notice that Bao [22] and Lero [44] do not use query encoding but only plan encoding. For instance, Bao does not identify which table is used in a particular node of the query plan, using only table cardinalities and costs. Such a representation can benefit from more schema-agnosticism and easier re-training when the database schema changes, though it violates the *principle of invariance* [17].

Let us consider the following thought experiments. Applying different filters in a query can result in the same cardinality for the same table. Similarly, tables with the same cardinalities can have the same encoding. In an ideal setting, we would want a unique 1-to-1 mapping between the feature variables D and the latency or cost y of a query and its given plan. However, the query latency is volatile and differs between multiple executions so that the plan encoding will instead result in a 1-to-many mapping of (D, y)-pairs.

Moreover, even having the query encoding as an additional input cannot guarantee invariance under a single cardinality encoding of the attributes. As we have discussed in the example above, applying different filters for a given column can result in the same cardinality estimation, i.e., leads to the loss of invariance.

Recommendation: To avoid spoiling the training process by not having the 1-to-1 mappings for (D, y)-data pairs, one can *use the embeddings instead of single value representations*, e.g., embeddings for text attributes like in Neo (see Figures 12 and 13 in the original paper [23]), and explicit vectorization of filters like in RTOS [42].

4.2 Encoding Expressiveness

The final set of features should clearly reflect both the global and local context. In query optimization, the global context is the *query* (as it does not change throughout the physical plan space search), and the local context is the *query plan*. This concept comes from Graph CNNs [14]. The basic idea is that applying more rounds of convolutions in the neural architectures will result in a graph node embedding with more global graph context and less local context.

Continuing the idea of using graph NNs, graph transformers [5] are used in LOGER [3] in an adjacent context for query encoding aggregation. On the other hand, methods like Bao [22] and Lero [44] are missing the query encoding part, which increases the probability of converging to a local optimum [9].

Recommendation: We would suggest *using both the query and the plan encoding*, which will result in better convergence.

5 TRAINING LEARNED QUERY OPTIMIZERS

In this section, we discuss how the "brains" of LQOs work and what conditions should be met to make them work as expected. The key feature of recent LQOs is the possibility to learn the entire query optimizer process with the help of ML models. From Table 1, it is visible how different the training pipelines are among LQOs. For example, a query plan having a tree representation structure implies two possibilities when processing: some can treat it as an image and apply Tree Convolutions [26], others treat it as a sequence of node pairs (i.e., text) and apply a Tree-LSTM [33]. However, there is still no common ground, e.g., for the performance analysis during model training or the choice of the training method. In this section, we discuss the most widespread issues of LQOs at the training phase.

5.1 Avoiding ML Model Overfitting

Overfitting is a typical ML problem when the model performance improves on the training data and at the same time deteriorates on the validation data [38]. From the definition, it is clear that RL-based methods do not suffer from this problem because they learn an optimal policy by maximizing or minimizing a non-stationary objective function that depends on the action policy itself. However, RL methods might get stuck in a sub-optimal policy without enough exploration [32]. Contrarily, classical supervised methods are prone to converge to a suboptimal solution.

To avoid overfitting, commonly *hyperparameter tuning via cross-validation (CV)*, *early stopping* and *regularization* are applied. Regularizations like, e.g., dropout [36] are straightforward and simply

Table 1: Main encoding components of LQOs. We distinguish between *query encoding* and *plan encoding*. Both Bao and LOGER provide hints about what types of joins not to use. Bao also provides hints for scan types.

LQO	Query Encoding				Plan Encoding			Training Specifics					
	Adjacency Matrix[1]	Numerical Attributes[2]	Text Attributes	Encoding Aggregation*	Join Type	Scan Type	Table Identifier[3]	Data+[4]	ML Model*	Plan Processing	Model Output	Testing*	DBMS Integration
Neo [23]	✓	cardinality	word2vec	stacking	✓	✓	✓	-	Regression	Tree-CNN	Plan	Static	-
RTOS [42]	✓	filters	cardinality	FC + pooling	-	-	✓	-	Regression	Tree-LSTM	Plan	CV	-
Bao [22]	-	-	-	-	✓	✓	-	✓	Regression	Tree-CNN	Hint set	Time Series	✓
Balsa [39]	✓	cardinality	cardinality	stacking	✓	✓	✓	-	Regression	Tree-CNN	Plan	Static	-
Lero [44]	-	-	-	-	✓	✓	✓	✓	LTR	Tree-CNN	Plan	Static	✓
LEON [4]	✓	cardinality	cardinality	stacking	✓	✓	✓	-	LTR	Tree-CNN	Plan	Static	-
LOGER [3]	✓	filters	cardinality	FC + pooling + GT	✓	-	✓	-	Regression	Tree-LSTM	Hint	Static	-
HybridQO [41]	✓	cardinality	cardinality	stacking + FC	✓	✓	✓	✓	Regression	Tree-LSTM	Plan	Static	-

[1] One-hot-encoding of the join subgraph for a particular (sub)query

[2] Filters explicitly encode >, =, and < symbols with min-max scaled filter values

[3] One-hot-encoding of tables in the DBMS schema

[4] Whether the method uses additional queries (outside of the provided benchmark queries) for training data generation or not

* Stacking: assembling of several features or vectors into a single vector, Pooling: downsampling of the spatial dimensions of the input data, CV: Cross-validation on JOB, FC: Fully-connected layer in the neural network, GT: Graph transformation, LTR: Learning-to-rank, Static: Static split of JOB, Time Series: Sequential continuous testing on previously unseen queries

increase the number of hyperparameters that need to be tuned, though other techniques are harder to tweak. Among recent LQOs, only RTOS applies CV to measure final aggregated performance metrics, though this does not help choose the final model. Balsa uses early stopping with performance improvement on the non-fixed validation set. LEON is doing a similar early stopping procedure, though using accuracy as a target metric. Bao uses a continuous "time series" testing of the model on previously unseen queries.

Recommendation: For RL methods, one can still *use hyperparameter tuning* as it would also help to improve the general model performance. For the QOP, accuracy for both cost and latency is a suboptimal quality metric as we do not know the optimal plan in advance (at least for higher-order joins). Thus, using accuracy as an early-stopping or cross-validation criterion is undesirable. The holdout data should be fixed (not CV, not "time series"), as the measurement on it should be comparable [7].

5.2 Changing Target Variables On-the-Fly

Query optimization has interesting specifics regarding the target to be optimized, which could either be a cost or a latency. It results in finding a *trade-off between speed* (as costs could be quickly estimated by an arbitrary cost model) *and accuracy* (as latency gives the exact value for how long the query takes to execute). Some methods like HybridQO take advantage of both by first training the model that suggests plans based on cost and then training another model that chooses between candidates based on latencies. At the same time, methods like RTOS, Balsa, Lero, and LEON try to use a single predictive model that first pre-trains using costs and then continues training with latencies. A key issue of this approach is that latencies and costs have significantly different numerical properties, and any progress made in the pre-training phase is lost, as the model needs to adapt to an entirely new scale and variance (i.e. deviation from the mean) of the target values [2].

Recommendation: You can *exchange the cost and latency on-the-fly during the training when using learning-to-rank models*, since real values are transformed into relative rankings forming the target variable [20]. Another approach is to *use an architecture that chains the ML models* like in HybridQO, where different target variables are served to different models in the ML pipeline.

6 EVALUATING LEARNED QUERY OPTIMIZERS

In this section, we outline the importance of choosing the right test set, how this decision influences the model's measured performance, and the concept of covariate shift.

6.1 Test Set Choice

The *train/test split* is a cornerstone of any supervised method. This split is used to differentiate between which part of the data an ML model is allowed to see during training and which part is used to test its ability to perform on previously unseen data, measuring the generalization ability of the model.

The extended JOB workload introduced by Neo [23] was a first attempt to test the ability of models to deal with previously unseen queries that are distinct from the original JOB queries. The queries added in Ext-JOB exhibit additional operators that are not present in JOB (such as GROUP BY or ORDER BY). Due to the nature of merge joins [13], LQOs that prefer this join method tend to gain an advantage from including ORDER BY operators. As a result, the comparison between different methods is unfairly skewed.

Balsa introduced JOB-Slow, where the 19 slowest queries shape the test set, and all other queries are the training set. This intuitively simple-to-understand train/test split focuses on the queries that have the most impact on the overall execution time for a full workload. However, all the 19 queries of the JOB-Slow test set have 11 or fewer joins, while 11 queries have just 6 or fewer joins. Figure 2 shows a scatter plot of the execution time vs. the number of joins. We observe that queries having between 6 and 11 joins have the largest execution times and thus, the highest potential for being optimized. At the same time, this is the range where non-exhaustive optimizers are typically disabled (e.g., PostgreSQL's GEQO is by default only enabled for 12 or more tables). Hence, exhaustive methods can still fully explore the space of possible plans.

Another approach for splitting queries was introduced by [43], where the authors built train/test splits based on the number of joins. For example, all queries with 3 or 4 joins form the test set, and all others form the training set. From Figure 2 it is clear that the number of joins is an irrelevant proxy for execution time, according to a regression analysis with $R^2 = -0.11$. Thus, splitting queries as such forms groups that are not aligned with the true optimization target, i.e., the execution time.

Recommendation: We propose several *edge cases for train/test splits to cover different areas of generalization*, namely the generalization gap and sampling out-of-distribution (see Section 7.2).

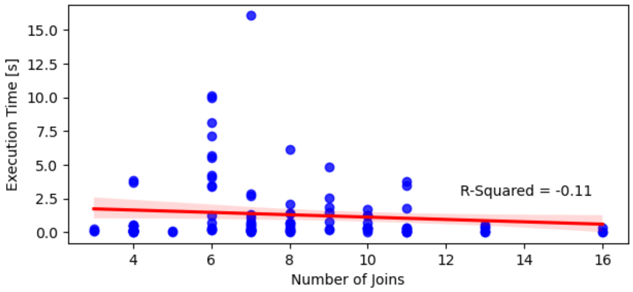


Figure 2: Scatter plot of the execution time per number of joins for all queries in JOB.

6.2 Covariate Shift

Another relevant topic for evaluating LQOs is the concept of *covariate shift*, i.e. a change in the database content away from how a method was trained. DBMSes tackle this challenge by continuously updating the internal statistics. For a LQO, however, a change in the database content affects how a query is encoded and thus its prediction. For example, a query about movies with a release date greater than 2022 will continuously increase its result set size, as newly released movies are added to the DBMS.

While this topic is often mentioned in aspirational future work, methods like Bao have started to think about designing their encoding to be able to deal with covariate shift by omitting tables and column identifiers in their encoding (see Section 4 for more details). However, as we show in an experiment in Section 8.3, updated cardinality estimates in the encoding are insufficient to keep up with changing database content.

Recommendation: We propose that future methods should *include a simple experiment to measure the ability to deal with covariate shift*, as we have performed in Section 8.3.

7 FRAMEWORK FOR BENCHMARKING LEARNED QUERY OPTIMIZERS

In this section, we introduce our benchmarking framework, aiming for a comprehensive evaluation of LQOs. The objective is to conduct benchmarking in a holistic manner, ensuring a fair comparison of methods in an end-to-end setting.

To do this, our benchmark assumes a reproducible setup, particularly regarding engineering, including but not limited to (a) the content of the database underlying a benchmark workload, (b) the full code base of the LQO, (c) the version of the programming language, such as Python, and all used libraries, (d) a detailed configuration of the DBMS (unless all parameters are left on default), as well as (e) all queries and their assignment into train/test splits.

7.1 DBMS Configuration & Database Tuning

For analyzing query execution times, both the used hardware and the DBMS configuration greatly impact the comparability of LQOs. We will now analyze the major parameter settings systematically.

Table 2 gives an overview of the different parameter settings used in various publications, compared to the default values of PostgreSQL, as well as the suggested setting for the Join Order Benchmark [18]. Note that the configurations for Neo [23] and HybridQO [41] are omitted from Table 2, as their code (Neo) and database configuration (HybridQO) are not publicly available. A further observation is that only Balsa and LEON published the full DBMS configuration file among their artifacts. We have categorized the parameters into the following groups:

Join Order: The join order is typically forced through libraries such as `pg_hint_plan` [27], though PostgreSQL can also be made to follow the explicit order given in the SQL statement by setting `join_collapse_limit` to 1. The genetic query optimization algorithm (GEQO) of PostgreSQL is used for queries with large number of joins, by default 12 or more. It can either be disabled by setting `geqo_threshold` to a value larger than the number of joins in a workload or disabled completely with the `geqo` parameter.

Working Memory: The default values for PostgreSQL's memory are small. Given the amount of RAM available today, increasing the working memory and buffer sizes is advisable. Balsa drastically increases the working memory (`work_mem`) from 4 MB to 4 GB, while Bao and Neo keep the default value, despite the proposed 2 GB by [18]. Similarly, for the `shared_buffers`, Balsa uses a much larger buffer at 32 GB compared to the 4 GB recommendation that Bao and Neo use. LOGER further increases the `shared_buffers` value to 64 GB, though their machine also has more RAM available.

Note that the amount of `work_mem` is available to all workers in parallel query execution, that means for N amount of workers, the `shared_buffers` should be at least $N\times$ `work_mem`. Furthermore, all methods use the default cache size (`effective_cache_size`) of 4 GB, ignoring the recommendation to increase it to 32 GB by [18]. Increasing its value in our configuration from 4 to 32 GB reduced the planning time for a handful of outlier queries significantly (from up to 3 seconds to below 100 milliseconds).

Parallelization: These parameters define the number of workers and processes used during query execution. To fully utilize a multi-core system, Balsa increases the number of worker processes `max_worker_ processes` to match `max_parallel_workers`. While increasing the number of parallel workers can speed up query execution, the amount of required compute resources also increases significantly. LOGER and Lero take a different approach, disabling any parallel query execution completely.

Scan Types: These parameters directly change the types of scans that are being used by PostgreSQL and significantly alter the toolset available for query execution. Only Balsa and LEON change these values by disabling both bitmap and tid scans, while neither paper offers an explanation for taking this approach.

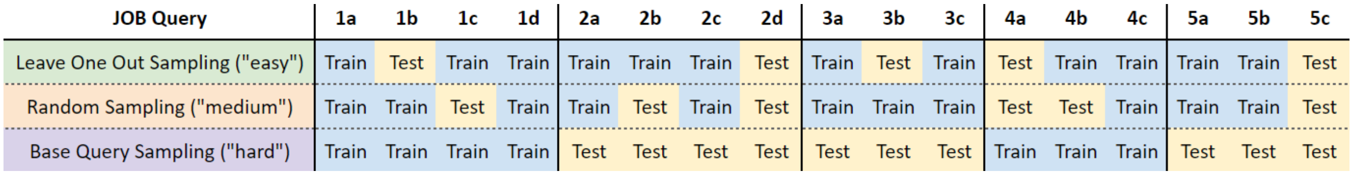
7.2 Dataset Split

The way the dataset is split into training and test sets has a significant impact on the performance of a trained model. While it is advisable that both sets contain data from a similar distribution,

Table 2: Overview of different PostgreSQL configurations (database tuning parameters) used in various papers of LQOs. *Deviations from PostgreSQL's default values are marked* in the respective columns. Note, that the values for Neo [23] and HybridQO [41] are missing from the table, as their configuration parameters are not publicly available.

PostgreSQL Config Parameter	**Default Values**	JOB [18]	Bao [22]	Balsa [39], LEON [4]	LOGER [3]	Lero [44]	**Our Framework**
Amount of RAM used by authors		64 GB	15 GB	64 GB	256 GB	512 GB	64 GB
Join Order							
geqo_threshold	12	18			2 or 1,024		
geqo	on			off	off	off	off[1]
Working Memory							
work_mem	4 MB	2 GB		4 GB			4 GB
shared_buffers	128 MB	4 GB	4 GB	32 GB	64 GB		32 GB
temp_buffers	8 MB			32 GB			32 GB
effective_cache_size	4 GB	32 GB					32 GB
Parallelization							
max_parallel_workers	8				1	0	
max_parallel_workers_per_gather	8				1	0	
max_worker_processes	2			8			8
Scan Types							
enable_bitmapscan	on			off			
enable_tidscan	on			off			

[1] GEQO is only turned on for Bao and when PostgreSQL fully controls the query execution.



JOB Query	1a	1b	1c	1d	2a	2b	2c	2d	3a	3b	3c	4a	4b	4c	5a	5b	5c
Leave One Out Sampling ("easy")	Train	Test	Train	Train	Train	Train	Train	Test	Train	Test	Train	Test	Train	Train	Train	Train	Test
Random Sampling ("medium")	Train	Train	Test	Train	Train	Test	Train	Test	Train	Train	Train	Test	Test	Train	Train	Train	Test
Base Query Sampling ("hard")	Train	Train	Train	Train	Test	Test	Test	Test	Test	Test	Test	Train	Train	Train	Test	Test	Test

Figure 3: Overview of different dataset split sampling types for JOB: Leave One Out Sampling (top), Random Sampling (middle), and Base Query Sampling (bottom). For instance, Base Query 1 has 4 variations: 1a, 1b, 1c and 1d.

we have to be careful to avoid leaking information from one set to the other. More specifically, the Join Order Benchmark queries are deduced from 33 different base queries (or templates), and the full 113 queries are made up of between 2 and 6 variations of each base query (denoted as 1a, 1b, 1c, ...). Variants of the same base query share the same tables and joins but differ in filter statements. These differences can be different filter values (e.g. production_year < 2000 vs. production_year = 2023) or applying filters on other columns (e.g. genre = 'horror' vs. name LIKE %an%). The queries in the STACK [22] dataset also follow the same pattern of 16 base queries across 6,191 queries (with 100 to 1,010 variations per query).

Generating queries from templates introduces a strong correlation in the structure of the optimal join plan for some, but not all, queries. To measure the effect of potential data leakage, we propose the following sampling techniques to generate dataset splits (see Figure 3 for a visual example of training and test set assignments):
(1) **Leave One Out Sampling** extracts exactly one variant of each base query into the test set. All other variants of the base query are contained in the training set. This split maximizes the amount of information that can potentially be leveraged from the training onto the test set. We expect this split to be the *easiest to learn.*
(2) **Random Sampling** distributes all queries randomly into train and test sets, ignoring any base query or template affiliations. This is a *medium difficulty* sampling, and it can be applied to any workload, as there is no requirement for the existence of *base query families.*
(3) **Base Query Sampling** keeps all queries of the same base query either in the training or the test set. This ensures that the intra-family similarity of the query structure does not leak from the training set into the test set. We believe this to be the *most difficult* split, as a model cannot apply the join structure learned from one variant of the same base query to another.

7.3 Measuring Query Executions

As LQOs are all evaluated by the runtime of queries in a workload, and some LQOs directly predict the execution time for a given physical plan, it is vital that runtime measurements are as consistent as possible. One of the primary reasons for high variance in executing the same query is caused by the buffer and cache states in the DBMS. For example, when the same query is executed twice one after another, the first run generally takes longer than the second one. As buffers and caches switch from cold to hot cache, runtimes become more consistent. In the ideal scenario, we could execute every query many times to achieve a robust measurement. However, every additional execution after the first one takes additional time that is not spent on executing other queries, costing valuable compute resources. We experimentally determined that executing queries *3 times* and taking the third execution gives the most stable results without incurring an unnecessary amount of execution overhead (see Section 8.6 for more details on the experiment).

8 EXPERIMENTS

In this section, we present our extensive evaluation of LQOs on the Join Order Benchmark (JOB) and STACK. First, we give an overview of the setup and hardware used; then, we discuss different approaches to generate train/test splits. Finally, we show the results of our experiments with a number of ablation studies.

8.1 General Setup

8.1.1 Software and Hardware. All our experiments were conducted using PostgreSQL version 12.5 by measuring the query execution time through `EXPLAIN ANALYZE` calls, using both execution and planning time. In addition, we also include the inference time for LQOs. Measurements are taken by executing the same query three times and taking the last query execution (hot cache).

Our instance of PostgreSQL is configured largely with default parameters in mind, closely following the configuration used by Balsa [39]. In comparison, we reenabled both bitmap and tid scans and increased the `effective_cache_size` from 4 to 32 GB. The main differences to PostgreSQL's default can be seen in Table 2 and primarily include changes to the memory configuration and an increased amount of parallel workers. In addition, we disabled the `AUTOVACUUM` feature, as the query workload is stable and `ANALYZE` is run once after loading all data into PostgreSQL, taking 3 minutes for IMDB (JOB) and 16.5 minutes for STACK.

We have decided to follow the configuration of the Balsa experiments, as they include memory settings that strongly follow the best practices guide proposed by PostgreSQL [31] and the suggestions of Leis et al. [18]. Furthermore, Balsa is the first method to increase the number of available workers processes from 2 to 8, given the typical machines with many CPU cores. We further change the `effective_cache_size` parameter in line with the best practices of PostgreSQL and re-enable both bitmap and tid scans.

For the Join Order Benchmark, the authors of Balsa added two additional indexes on the `subject_id` and `status_id` columns of the `complete_cast` table, compared to the indexes provided by [18]. We also include the additional indexes in our experiments. The experiments were run inside Docker containers, using a Tesla T4 GPU, 64 GB of RAM, and 16 CPU cores.

8.1.2 Query Workload. We evaluate various LQOs on the JOB and STACK workloads. Both workloads are highly relevant in recent literature and have been used in most of the evaluated LQO methods. For JOB [18], we use the 113 queries provided. The STACK [22] workload includes 6,191 queries across 16 base queries, which we down-sampled to 14 base queries[4] with 8 randomly sampled variations, each. This allows the methods to be trained and evaluated using a similar amount of data for JOB and STACK, leaving the models at a similar level of statistical power.

8.1.3 Dataset Split. For our experiments, we generated the train/test splits by uniformly sampling across all queries (Random Sampling), the base queries (Base Query Sampling), or the variants of each base query (Leave One Out Sampling). For the Random splits and Base Query splits, we used an 80-20 ratio between training and test sets. The dataset splits are sampled once and shared across all the evaluated methods. Detailed listings of the training and test sets for all splits can be found in our code repository[5], along with the hyperparameters of all methods.

8.1.4 Additional Noteworthy Changes. As we evaluate the LQO methods under our unified framework, there are differences to the experiments conducted by the authors of the methods (see previous sections). Hence, direct comparisons to prior results are impossible.

In addition, Bao was originally trained on 2,500 newly generated queries in the JOB workload style. In our experiments, Bao was only trained on the training set of the respective train/test splits and has seen the training queries multiple times. For LEON, we have limited the amount of real time spent on training to twice the time it took Balsa to finish training, i.e., 120 hours. This time budget likely reduces the performance of LEON, but as shown in Section 8.2.2, the inference time heavily dominates its overall runtime, not just the execution time.

8.2 Comparison of Current State-of-the-Art Learned Query Optimizers

In this section, we analyze the performance of current state-of-the-art methods for LQOs (namely Neo[6] [23], Bao [22], Balsa [39], LEON [4] and HybridQO [41]) compared to PostgreSQL as our baseline. We do not include RTOS [42], Lero [44] and LOGER [3] in our experiments because they are either (a) unavailable, (b) require to disable parallelization in query executions because multiple queries are run in parallel or, (c) require to invest an extensive amount of engineering to enable these methods to parse the EXPLAIN output.

8.2.1 End-to-End Performance. For all algorithms, we report a variety of time measurements defined as follows:
(1) **Inference Time:** This measure includes all time that an LQO spends to encode a query, iterate over variations of query plans, gather cost information to guide further decisions, and finally, use an ML model to generate predictions. After the inference time has passed, a given SQL query is ready to be sent to PostgreSQL with hints on which scan or join types to use and in which order.
(2) **Planning Time:** Once PostgreSQL receives a query, it spends an amount of time on planning the query before a final physical plan is generated and sent for execution. For LQOs with an extension running inside PostgreSQL, typically, the inference time is reported as part of the planning time.
(3) **Execution Time:** Encompasses the amount of time spent by PostgreSQL to execute the query and gather the result set.
(4) **End-to-end Execution Time:** A combination of the previous three-time measurements, measuring how long a method takes to devise a query plan to execute and how much time PostgreSQL spends to get the result from the database. We believe this measurement to be the primary objective for optimization.

We do not include the network latency in our inference, planning and execution times, as LQOs cannot influence it directly. While the network latency can be a significant amount of time (notably for fast queries), optimizing for it is beyond the scope of this evaluation.

[4] Templates 9 and 10 are removed in accordance with Balsa [39], where the authors report a limitation in the pg_hint_plan extension in dealing with views and subqueries.

[5] See our code here: https://github.com/edualc/lqo_ml_perspective

[6] Since the code of Neo is not publicly available, we have used the re-implementation in Balsa. The original authors of Neo have checked it and confirmed to us that it is suitable for benchmarking.

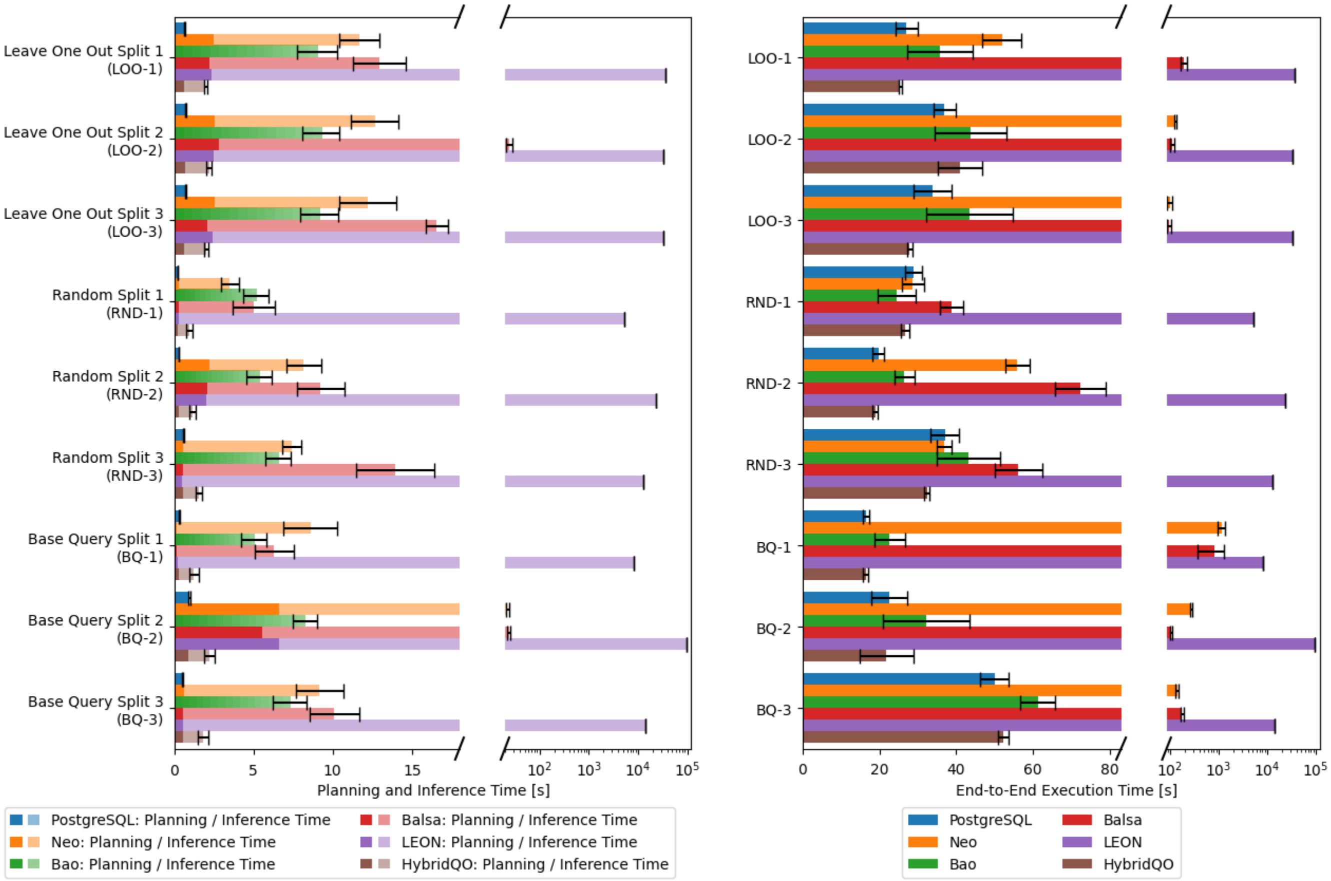


Figure 4: Comparative overview of each method's performance on the *test set* of various dataset splits on the Join Order Benchmark (JOB). The figure on the left depicts the planning time (darker colour) and inference time (lighter colour), respectively. Note that Bao runs inside PostgreSQL as an extension, and its inference time is directly added to the planning time. The figure on the right side shows the execution times on the same train/test splits. Please observe that the x-axis of both figures is divided.

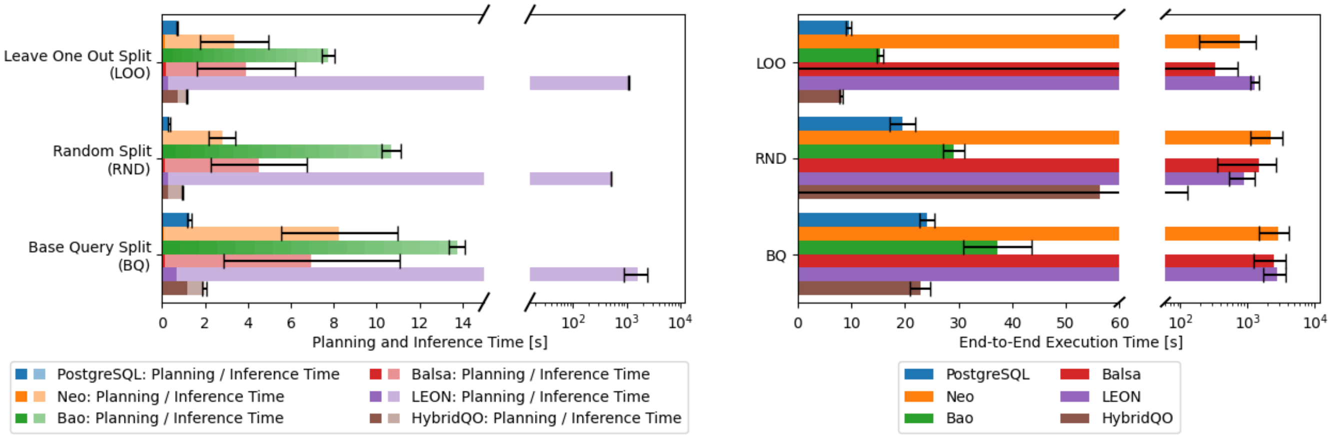


Figure 5: Comparative overview of each method's performance on the *test set* of various dataset splits on STACK.

We increase the difficulty iteratively across experiments, starting with the leave one out sampling, then the random sampling and finally, the base query sampling that generated the train/test splits. All queries were executed three times. The planning and execution times have been taken from the third execution.

Figures 4 and 5 present the performance on JOB and STACK across all three sampling methods and their individual train/test splits. In summary, **PostgreSQL generally performs best, followed by HybridQO, then Bao, Neo, Balsa, and finally LEON**. However, PostgreSQL fails to eclipse all methods on all splits by a statistically significant margin. In particular, HybridQO and Bao achieve comparable results on most train/test splits, with HybridQO outperforming PostgreSQL on the leave one out split of STACK.

Let us first take a look at the results on JOB. For the *leave one out sampling*, which we consider to be the easiest train/test split, PostgreSQL, Bao and HybridQO execute the test queries in around 30 seconds. However, Bao spends 8.5 seconds longer to plan queries, resulting in a 25% slower end-to-end execution time. Bao's larger confidence interval gives a first hint that it has found plans that are generally faster than PostgreSQL, but they are not speeding up the execution time enough to have an advantage. HybridQO, on the other hand, finds plans that are 2.5 seconds faster while only spending 1.4 seconds for inference, allowing it to outperform PostgreSQL significantly on the third split.

LEON is the fourth fastest method by execution time at 58 seconds, but its inference time is around 9.6 hours long, making its use impractical for interactive querying (with more complex queries requiring proportionally more inference time to complete).

The overall fourth fastest method is Neo at 93 seconds, followed by Balsa at 134 seconds with 286% and 411% slower end-to-end execution times compared to PostgreSQL, respectively.

For the *random sampling*, i.e., the medium difficulty train/test split, PostgreSQL and Bao remain competitive with each other, with Bao achieving even a lower execution time of 25 vs. PostgreSQL's 28 seconds. However, this is not a statistically significant difference. Including the inference and planning times as well, Bao is again at a slight disadvantage. Similar to the previous split, HybridQO matches PostgreSQL and even significantly outperforms it and Bao on the third split with 24 seconds. Compared to the leave one out sampling, both Neo and Balsa achieve 2-3× faster end-to-end execution times, reaching comparable results for PostgreSQL on 2 out of the 3 train/test splits using this sampling. LEON struggles with these queries and two queries timeout (26b and 32b) in two separate splits, leading to a drastic increase in the execution time. However, given the large inference time of on average of 3.8 hours, this has little impact on its overall ranking.

Finally, let us examine the *base query sampling*, i.e., the most difficult sampling technique. For the first time, Bao only achieves comparable results to PostgreSQL on 1, and HybridQO on all, out of the 3 train/test splits, confirming the increased difficulty. Neo and Balsa struggle particularly with base query split 1. 3 queries of the test set timeout for Neo, and 15 queries across both train and test set for Balsa. LEON, however, can largely match PostgreSQL's direct execution time if the method can overcome its inference time.

The results on the STACK dataset (see Figure 5) largely confirm the results of JOB. A major difference is the inference time of LEON, which is one order of magnitude lower given the generally lower number of joins in the STACK queries compared to JOB. Neo, Balsa and LEON all suffer greatly from significant amounts of timed out queries. Unlike in JOB, Bao is unable to match the performance of PostgreSQL on STACK due to more complicated SQL features in the STACK queries, leading to much longer inference times. HybridQO outperforms PostgreSQL on the leave one out split, is comparable on the base query split but also suffers from timed out queries on the random split, hinting at problems of robustness.

In summary, we see the *significant impact of the inference time on the overall end-to-end execution time.* While there are methods that, on some train/test splits, perform comparable to PostgreSQL or even slightly outperform it, these results show the *importance of how queries are split for training*. Furthermore, it is vital that evaluations include the inference time, as it strongly shapes the ranking between methods compared to the execution time alone.

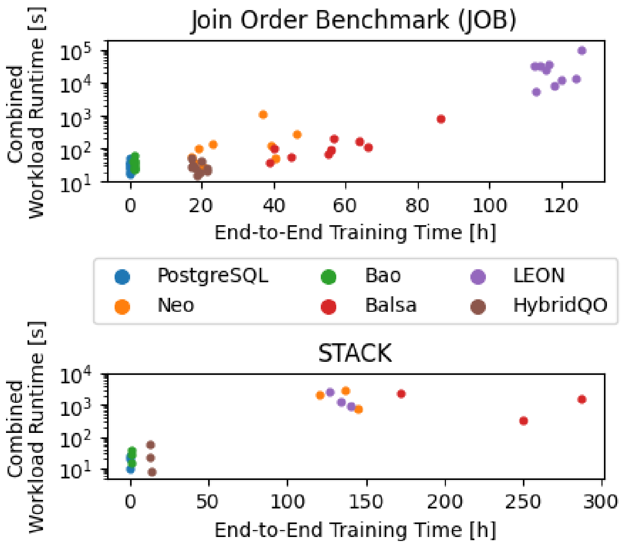


Figure 6: Comparison of the end-to-end training time against the combined workload runtime for both workloads, where each dot represents a model from a different split.

8.2.2 Training Time. After comparing the query performances, we also take a look at the amount of time to train a model. While the definition of training time is sometimes unclear, we intend to take a holistic look with an *end-to-end training time*, that is, including (a) the time spent collecting query results from the DBMS, (b) any time spent training the model, (c) the ongoing evaluation of the current model's performance, and (d) any pre- or postprocessing, initialization, and artifact generation. In short, the *full amount of time spent from starting the training procedure until it terminates.*

We make this distinction not to penalize additional logging or more frequent checks on the model performance but to get a fairer overall comparison. For example, a method might have a very quick training period but spend a lot of time querying training data from the DBMS, while another method needs fewer database queries but uses a more complex NN architecture that spends more time in weight updates. The overall amount of time spent also informs how often a model could be retrained given a time budget.

In Figure 6 we compare the end-to-end training times on the x-axis and the combined workload runtimes (the sum of the end-to-end execution times of all queries in the workload) on the y-axis. Each dot represents one train/test split. For example, one orange dot could be the Neo method on random split 2.

Since PostgreSQL's optimizer does not require any inherent training acting as a baseline, its end-to-end training time is set to zero. Among the evaluated methods, Bao requires the least amount of time on JOB at around 2 hours, HybridQO around 20 hours, Neo between 20 and 40 hours, Balsa between 40 and 85 hours, and LEON from 110 to 130 hours. For STACK, Bao trains for 2 hours, HybridQO for 12 to 14 hours, Neo and LEON between 120 and 140 hour, and Balsa between 170 and 290 hours.

As a naive assumption, one would expect to see a performance increase as more time is spent during training, but we observe exactly the opposite behaviour: Methods that have spent *more time to build and train their model reach inferior results* compared to methods that finish training more quickly. We primarily attribute this discrepancy between methods to the number of plans considered.

For example, during the training on JOB, Neo executed between 4,000 and 8,000 plans in PostgreSQL, Balsa between 19,000 and 21,000 plans. Even ignoring the quality of either methods' executed plans, it is obvious that 2-3× more plans also require more processing time. The authors of Balsa specifically tackled this challenge by allowing all required plans to be executed on multiple DBMS instances in parallel and by timing out long-running queries, which Neo does not. LEON does not fully execute the majority of its generated plans; However, it calls PostgreSQL to ask for cost estimates up to multiple tens of thousands of subplans, such that predicting just a plan for query 29a (the query with 17 aliased tables, the highest amount in all of JOB) takes around 6.5 hours[7].

8.3 Ablation Study: Covariate Shift

One of the challenges for query optimizers, in general, is their dependency on up-to-date statistics of the database content. In DBMS, statistics are regularly refreshed, but LQOs do not have the luxury to easily update trained model weights, with options to either train a new model from scratch or fine-tune and continue training, adapting to changes to the underlying database.

To show whether an encoding that represents the content of the database solely by cardinality (such as Bao) can deal with covariate shift, we conduct the following experiment. We generate a smaller copy of IMDB, referred to as IMDB-50%. As the name implies, we keep 50% of the rows in the `title` table using Bernoulli sampling, ensuring that the available data is halved, but the distribution of values remains comparable to the original version. The other 50% of rows are dropped using CASCADE to ensure referential integrity. We specifically choose to alter the contents of the `title` table since it is the only table in IMDB that is part of all JOB queries.

After sampling, we see a reduction of 50% for the number of records in all movie-related (title, movie_companies, movie_info, movie_info_idx, movie_keyword, and movie_link) and cast-related tables (cast_info and complete_cast). Our sampling on `title` leaves all other tables unaffected. After the changes had been made to IMDB-50%, the internal statistics of PostgreSQL were updated.

Our experiment aims to show that methods like Bao only using the cardinality in their encoding show a performance degradation when more data is added (simulating covariate shift). We train one Bao model on IMDB (referred to as Bao-Full) and a second Bao model on the reduced size IMDB-50% (referred to as Bao-50) using the same "base query split 1" train/test split.

Query 16b is a striking outlier, timing out in 1 of 4 Bao-50 models, while the other 3 generated plans that are 19 seconds slower than Bao-Full. In relative differences, query 31c is 24× slower using Bao-50 at 8.4 seconds compared to Bao-Full with 350 milliseconds. Query 17a is 4.5× slower at 12.2 seconds compared to Bao-Full with 2.7 seconds. On the other hand, the different cardinality regimes seen by Bao-50 also allow it to improve a few queries over Bao-Full by a factor of 1.9× for query 7c, 1.6× for 26c and 1.3× for 10c.

These results indicate that the *DBMS system updating the statistics (i.e., cardinality estimates) is insufficient to keep up with a newly trained model*. This performance degradation further points to difficulties in generalization, particularly when larger cardinality values have not been seen during the training process and are, hence, out of distribution. There is currently no solution to this problem other than re-training or fine-tuning the model with queries running against the new database state. Because of this, methods that can continuously be updated and re-trained are preferable.

8.4 Ablation Study: Bitmap and Tid Scans

We have observed multiple publications that disabled bitmap and tid scans, namely Balsa [39], LEON [4], and a recently published analysis [43], without giving a reason for doing so. This experiment aims to see if changing PostgreSQL's tool kit significantly impacts the query performance of the individual queries. For the comparison, we use the baseline PostgreSQL performance from the previous experiment in Section 8.2, and we have run the same 113 queries from JOB with bitmap and tid scans disabled.

The difference in execution time exceeds 250 milliseconds for 28 queries, 24 of which are statistically significant. For those 24 queries, disabling bitmap and tid scans **speeds up** queries 28a, 7c, and 30a relative to their original execution times by a factor of 5.5×, 2.0×, and 1.8×, respectively. In contrast, queries 30c, 28b, and 15c are **slowed down** by a factor of 2.4×, 1.9× and 1.5×, respectively.

These findings show that *allowing PostgreSQL to use bitmap and tid scans significantly impacts the query performance*, particularly for the query templates 7, 8, 28, and 30. An interesting observation here is that the same family that has the highest gain of disabling said scans (query 28a with a speedup of 5.5×) also features a large slowdown (query 28b with a slow down by 1.9×).

8.5 Ablation Study: Genetic Query Optimizer

Similar to the disabling of various scan types, there exist differences in using GEQO, i.e., PostgreSQL's genetic query optimizer, across recent publications. We have analyzed the impact of disabling GEQO on the execution time of queries from JOB. Our experiment revealed 5 queries, for which the difference is statistically significant. Disabling GEQO **speeds up** query 30a by a factor of 1.6×, while the

[7] LEON caches plan and subplan cost estimates, generating files on the hard disk as large as 1.7 GB for JOB and 120 MB for STACK, respectively.

other four queries are **slowed down** by a factor of 9.9× (24b), 2.2× (26c), 2.1× (28a) and 1.7× (28b). The large slow-down factor of query 24b is explained by its quick execution time of just 28 milliseconds versus 272 milliseconds with GEQO disabled. While the impact of GEQO is smaller than that of bitmap and tid scans, it remains significant in particular among the slowest query templates.

In summary, these results show that it is *paramount that PostgreSQL operates at full capacity* (i.e., with GEQO enabled) in particular when the LQO does not replace, but rather enhance or guide the existing optimizer (for example, through the use of hints).

8.6 Robustness of Query Execution Times

The goal of this experiment is to determine the choice of the number of repeated executions k for the same query to reach a consistent execution time in a hot cache scenario. For example, in LEON [4] the authors use the geometric mean with $k = 3$, while in [43] the authors execute queries $k = 5$ times and take the arithmetic mean.

To achieve a fair comparison, we executed all queries of JOB using `EXPLAIN ANALYZE` 50 times in succession and in order (i.e., 1a, 1a, 1a, ..., 1a, 1b, 1b, ...). The execution time is extracted from PostgreSQL's `EXPLAIN` response, removing the network latency to the database from our measurements. By evaluating the distribution of execution times for the k-th iteration empirically, we can propose a value of k that strikes the balance between costs and robustness.

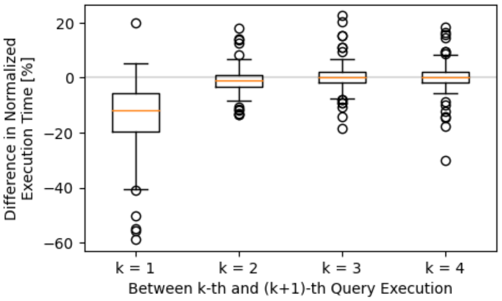


Figure 7: Difference in normalized execution time between successive query executions. For example, k=1 shows the difference between the 1st and 2nd query execution.

Figure 7 shows the normalized difference in query execution time (relative difference to the first executed query) when comparing pairs of the k-th and $(k + 1)$-th query execution. We observe, that the query execution time significantly shifts for the majority of executed queries at $k = 1$, with a mean reduction of 14.6% between the 1st and 2nd query execution, and another 1.03% from the 2nd to the 3rd. From then on, the fluctuations no longer show a trend that would benefit from more executions.

We see, thus empirically, that for robust measurements of execution times, it is important to at least execute a query twice. If time and costs allow, a third execution further improves the robustness, after which one can safely stop. Compared to the choice of $k = 5$ in [43], taking the third execution is 40% faster and more robust than averaging three measurements (for which the first query execution typically dominates as an outlier measurement).

8.7 Analysis of Query Plan Types

Given that there exists a larger number of bushy compared to left-deep and right-deep plans[8], it needs to be asked whether omitting *bushy plans* (as for example in RTOS, LOGER and HybridQO) is a reasonable choice. In [18], the slowdown for restricted tree shapes was measured in comparison to the optimal plan. The experiments' outcome shows that *left-deep trees are worse than bushy ones* but still result in a reasonable performance. It is worth noting that these experiments were executed by injecting *true cardinalities* to the cost model of the query optimizer. Moreover, some constraints on the join method selection according to [6] were applied.

By forcing all combinations, we analyzed all possible plans for JOB queries with ≤ 5 joins in the spirit of [18]. However, rather than using true cardinalities (which are considered the optimal case), we ran our experiments with the *DBMS's internal cardinality estimator*. Moreover, we allowed all join methods to be used.

As a result, *bushy plans perform on average like left-deep plans*. We confirm our results by obtaining the minimum of *p-value=0.285* for a two-side Mann-Whitney U-test [21][9] for the means of execution times. At the left tail of the combined distribution of execution times (i.e. among the fastest plans), however, bushy trees are significantly superior with a *p-value of 0.015* for the alternative hypothesis. That means, removing bushy trees from consideration drastically lowers the chance that a model finds the best plan.

9 CONCLUSION

In this paper, we outline the limitations of current LQO methods and put an emphasis on previously under-reported challenges. We provide a framework to equalize many parameters involved in benchmarking to yield increasingly robust results.

We perform an evaluation of current LQO methods on the Join Order Benchmark and show that consistently outperforming PostgreSQL is more difficult than expected, particularly when looking at the query optimization problem as an end-to-end process. We believe that our paper is a first step towards reproducible and consistent benchmark evaluations for LQOs and thus provides important novel insights into LQOs from an ML perspective.

ACKNOWLEDGMENTS

The project has received funding from the Swiss National Science Foundation under grant number 1921052. We also thank our colleagues from University of Konstanz, namely Michael Grossniklaus, Mehmet Aytimur and Silvan Reiner, and Dennis Gehrig from Zurich University of Applied Sciences, for valuable discussions.

[8] Left-deep and right-deep plans are hereafter only referred to as left-deep plans, without loss of generality.

[9] The selection of the non-parametric test over the T-test stems from the observed lack of normal distribution plausibility across distinct logical and physical plans.

REFERENCES

[1] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. *SIGMOD Rec.* 29, 2 (may 2000), 261–272. https://doi.org/10.1145/335191.335420

[2] Jason Brownlee. 2020. *Data preparation for machine learning: data cleaning, feature selection, and data transforms in Python.* Machine Learning Mastery.

[3] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A Learned Optimizer Towards Generating Efficient and Robust Query Execution Plans. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1777–1789.

[4] Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. 2023. LEON: A New Framework for ML-Aided Query Optimization. *Proc. VLDB Endow.* 16, 9 (2023), 2261–2273.

[5] Vijay Prakash Dwivedi and Xavier Bresson. 2021. A Generalization of Transformer Networks to Graphs. arXiv:2012.09699 [cs.LG]

[6] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2 ed.). Prentice Hall Press, USA.

[7] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press, Cambridge, MA, USA. http://www.deeplearningbook.org.

[8] Yang Guan, Shengbo Eben Li, Jingliang Duan, Jie Li, Yangang Ren, Qi Sun, and Bo Cheng. 2021. Direct and indirect reinforcement learning. arXiv:1912.10600 [cs.LG]

[9] Isabelle Guyon and André Elisseeff. 2006. *An Introduction to Feature Extraction.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/978-3-540-35488-8_1

[10] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Tan Wei Liang, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2022. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *VLDB* 15, 4 (2022).

[11] Jonas Heitz and Kurt Stockinger. 2019. Join query optimization with deep reinforcement learning algorithms. *arXiv preprint arXiv:1911.11689* (2019).

[12] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, Not from Queries! *Proc. VLDB Endow.* 13, 7 (mar 2020), 992–1005. https://doi.org/10.14778/3384345.3384349

[13] Mouna Kacimi and Thomas Neumann. 2009. *System R (R*) Optimizer.* Springer US, Boston, MA, 2900–2905. https://doi.org/10.1007/978-0-387-39940-9_384

[14] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations* (Palais des Congrès Neptune, Toulon, France) *(ICLR '17).* https://openreview.net/forum?id=SJU4ayYgl

[15] Sanjay Krishnan, Zongheng Yang, Kenneth Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. (08 2018).

[16] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[17] Joseph P La Salle. 1976. *The stability of dynamical systems.* SIAM.

[18] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[19] Justin J Levandoski, Per-Åke Larson, and Radu Stoica. 2013. Identifying hot and cold data in main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE).* IEEE, 26–37.

[20] Tie-Yan Liu. 2009. Learning to Rank for Information Retrieval. *Foundations and Trends® in Information Retrieval* 3, 3 (2009), 225–331. https://doi.org/10.1561/1500000016

[21] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60. http://www.jstor.org/stable/2236101

[22] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data.* 1275–1288.

[23] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019).

[24] 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

[25] Ryan Marcus and Olga Papaemmanouil. 2018. Towards a Hands-Free Query Optimizer through Deep Learning. (09 2018).

[26] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona) *(AAAI'16).* AAAI Press, 1287–1293.

[27] Nippon Telegraph and Telephone Corporation. 2012. pg_hint_plan Documentation. https://pghintplan.osdn.jp/pg_hint_plan.html. [Online; accessed August, 2023].

[28] Dušan Petković. 2011. Dynamic Programming Algorithm vs. Genetic Algorithm: Which is Faster?. In *Research and Development in Intelligent Systems XXVII*, Max Bramer, Miltos Petridis, and Adrian Hopgood (Eds.). Springer London, London, 483–488.

[29] Egor Rogov. 2022. Queries in PostgreSQL: Sort and Merge. https://postgrespro.com/blog/pgsql/5969770. [Online; accessed August, 2023].

[30] Stuart Russell and Peter Norvig. 2010. *Artificial Intelligence: A Modern Approach* (3 ed.). Prentice Hall.

[31] Greg Smith, Robert Treat, and Christopher Browne. 2021. Tuning Your PostgreSQL Server. https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server. [Online; accessed August, 2023].

[32] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press. http://incompleteideas.net/book/the-book-2nd.html

[33] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers).* Association for Computational Linguistics, Beijing, China, 1556–1566. https://doi.org/10.3115/v1/P15-1150

[34] The PostgreSQL Global Development Group. 2023. Genetic Query Optimization (GEQO) in PostgreSQL. https://www.postgresql.org/docs/current/geqo-pg-intro.html. [Online; accessed August, 2023].

[35] Transaction Processing Performance Council. 2023. TPC Benchmarks Overview. https://www.tpc.org/information/benchmarks5.asp. [Online; accessed August, 2023].

[36] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. 2013. Regularization of Neural Networks using DropConnect. In *Proceedings of the 30th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Sanjoy Dasgupta and David McAllester (Eds.), Vol. 28. PMLR, Atlanta, Georgia, USA, 1058–1066. https://proceedings.mlr.press/v28/wan13.html

[37] Chihping Wang and Ming-Syan Chen. 1996. On the complexity of distributed query optimization. *IEEE Transactions on Knowledge and Data Engineering* 8, 4 (1996), 650–662.

[38] Geoffrey I. Webb. 2010. *Overfitting.* Springer US, Boston, MA, 744–744. https://doi.org/10.1007/978-0-387-30164-8_623

[39] Zongheng Yang, Wei Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (6 2022), 931–944. https://doi.org/10.1145/3514221.3517885

[40] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (sep 2020), 61–73. https://doi.org/10.14778/3421424.3421432

[41] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-based or learning-based? A hybrid query optimizer for query plan selection. *Proceedings of the VLDB Endowment* 15, 13 (2022), 3924–3936.

[42] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE).* IEEE, 1297–1308.

[43] Zhang Yunjia, Chronis Yannis, Patel Jignesh M., and Rekatsinas Theodoros. 2023. Simple Adaptive Query Processing vs. Learned Query Optimizers: Observations and Analysis. *Proc. VLDB Endow.* 16, 9 (2023), 2962–2975.

[44] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *arXiv preprint arXiv:2302.06873* (2023).