GenJoin: Conditional Generative Plan-to-Plan Query Optimizer that Learns from Subplan Hints

PAVEL SULIMOV*, Zurich University of Applied Sciences, Switzerland
CLAUDE LEHMANN*, Zurich University of Applied Sciences, Switzerland
KURT STOCKINGER, Zurich University of Applied Sciences, Switzerland

Query optimization has become a research area where classical algorithms are being challenged by machine learning algorithms. At the same time, recent trends in learned query optimizers have shown that it is prudent to take advantage of decades of database research and augment classical query optimizers by shrinking the plan search space through different types of hints (e.g. by specifying the join type, scan type or the order of joins) rather than completely replacing the classical query optimizer with machine learning models. It is especially relevant for cases when classical optimizers cannot fully enumerate all logical and physical plans and, as an alternative, need to rely on less robust approaches like genetic algorithms. However, even symbiotically learned query optimizers are hampered by the need for vast amounts of training data, slow plan generation during inference and unstable results across various workload conditions. In this paper, we present GenJoin - a novel learned query optimizer that considers the query optimization problem as a generative task and is capable of learning from a random set of subplan hints to produce query plans that outperform classical optimizers. GenJoin is the first learned query optimizer that significantly and consistently outperforms PostgreSQL as well as state-of-the-art methods on two well-known real-world benchmarks across a variety of workloads using rigorous machine learning evaluations.

CCS Concepts: • **Information systems** → **Query optimization**.

Additional Key Words and Phrases: Query optimization, Generative AI

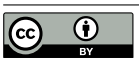
ACM Reference Format:
Pavel Sulimov, Claude Lehmann, and Kurt Stockinger. 2025. GenJoin: Conditional Generative Plan-to-Plan Query Optimizer that Learns from Subplan Hints. *Proc. ACM Manag. Data* 3, 4 (SIGMOD), Article 247 (September 2025), 25 pages. https://doi.org/10.1145/3749165

1 Introduction

Query optimization remains an active area of research for learned query optimizers (LQOs). In recent years, increasingly sophisticated methods have been developed for both cardinality estimation (CE) [13, 14, 16, 24, 31, 42, 43, 46, 49, 52] and join order selection (JOS) [2, 5, 6, 12, 17, 26–28, 41, 44, 45, 47, 48, 50]. CE approaches typically use statistical and machine learning (ML) models to approximate multivariate distributions over database table attributes [22]. The resulting cardinality estimates serve as an input to the cost models of query optimizers. At the same time, JOS models are considered to be the "brain" of query optimizers, whose outputs are logical and physical query plans [9].

*Both authors contributed equally to this work.

Authors' Contact Information: Pavel Sulimov, pavel.sulimov@zhaw.ch, Zurich University of Applied Sciences, Winterthur, Switzerland; Claude Lehmann, claude.lehmann@zhaw.ch, Zurich University of Applied Sciences, Winterthur, Switzerland; Kurt Stockinger, kurt.stockinger@zhaw.ch, Zurich University of Applied Sciences, Winterthur, Switzerland.



ACM 2836-6573/2025/9-ART247
https://doi.org/10.1145/3749165

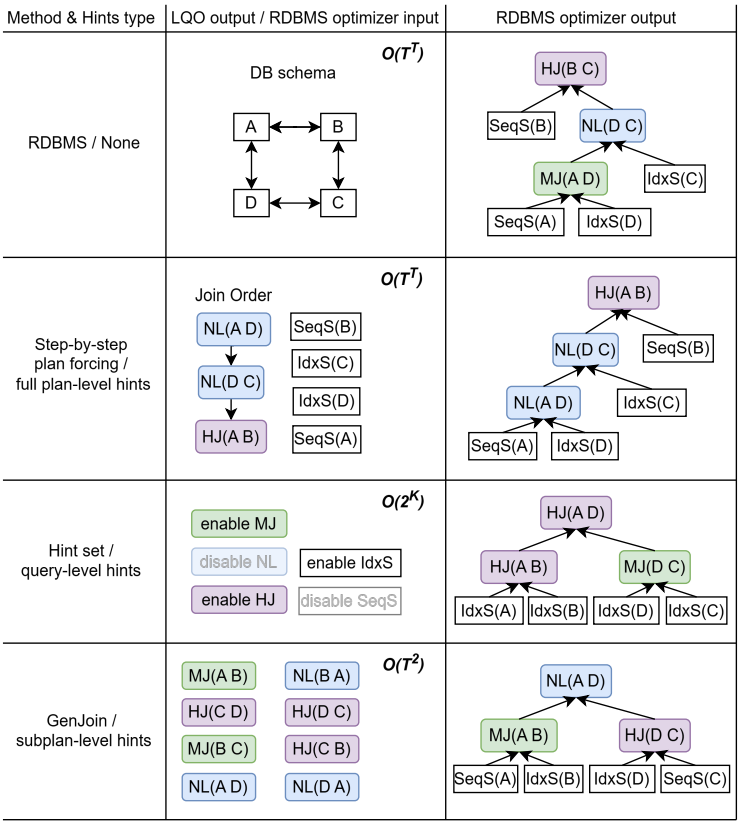


Fig. 1. Illustration of how GenJoin is different from two main streams of learned query optimizers for evaluating the query $A \bowtie B \bowtie C \bowtie D$: step-by-step plan generation and hint set methods. NL: nested loop join, HJ: hash join, MJ: merge join, SeqS: sequential scan, IdxS: index scan, K: number of hints, T: number of tables.

Starting from Cascades [11], JOS was mainly considered as dynamic programming (DP) [7] task, naturally assuming a join choice as a step in top-down or bottom-up plan construction. With the rise of deep learning and reinforcement learning (RL) [34] as a logical continuation of the DP ideas [23], classical query optimizers started being challenged by LQOs. Such step-by-step plan-building models control the full target plan specifications like the types of joins used, which scans to apply and in what order to join the tables, via a set of explicit hints for the RDBMS[1]. We call these approaches *full plan-level hint*[2] methods (see second row of Figure 1).

The trend of producing complete query plans as output was first questioned by Bao [26]. The idea is to use classical optimizers and empower them by just giving high-level hints like 'enable merge_join' rather than building complete query plans. In such a way, the exact join order choice is made by the built-in classical optimizer, though its search space is constrained through the provided *query-level hints*[3] (see third row of Figure 1).

[1]Using extensions such as pg_hint_plan: https://pg-hint-plan.readthedocs.io/en/latest/.
[2]The complexity of the potential prediction space is $O(T^T)$ [37] scaling with the number of tables T.
[3]Each hint can be turned on or off independently, resulting in a potential prediction space complexity of $O(2^K)$ scaling with the number of hints K.

Despite the progress made, LQOs have yet to yield significant improvements upon traditional approaches and consistently outperforming them continues to be an elusive goal [19]. We argue that the following weaknesses persist in today's LQOs:

(1) **Inefficient and limited RL.** Typical RL agents can take suboptimal steps and still end up with strong solutions due to the fact that either the game takes many steps to finish, and/or there is a possibility of stepping back. However, for query optimization and, in particular, bottom-up generation of join orders, every step is irreversible and has a significant impact on further steps with potentially fatal consequences in case of a bad choice.
(2) **Hint-based methods are double-edged swords.** Methods like Bao that give general hints (such as 'disable_hashjoin', which impact every single pairwise join in a query) limit the query plan subspace like in k-d tree search [30]. This reduces the chance of finding the optimal solution due to the coarse level of granularity.
(3) **Requirement for vast amounts of (training) data.** An important goal of query optimization is to learn the distributions and correlations of various attributes within and across tables to estimate the join cardinality. However, due to minute differences in query predicates, LQOs need to sample large amounts of query workloads to reach good generalization capabilities.
(4) **Finding optimal solutions with ML is time-intensive.** Certain ML algorithms do indeed find better query plans than traditional classical approaches. However, the time to find the solutions is often prohibitive due to the computational overhead for encoding queries or when using a model for inference which often greedily explores the plan space. Previous approaches often ignored these performance aspects and only focused on execution time as their only metric.

As a possible way to mitigate these LQOs' hurdles, we developed GenJoin - a novel, *generative plan-to-plan query optimizer that learns from subplan hints*[4] limited to join types such as nested loop, merge, and hash join. These three subplan hints are supported by the major database systems such as PostgreSQL, Oracle or SQL Server.

The output of the GenJoin model is a *set of two-way-join hints* (or subplan hints), e.g., use merge join on tables A and B, i.e. $MJ(A, B)$, hash join on tables C and B, i.e. $HJ(C, B)$, or nested loop join on tables D and A, i.e. $NL(D, A)$. **GenJoin does not specify exactly where in the join tree this particular join should be performed (if performed at all)** but leaves it up to the classical optimizer. Also, this way, GenJoin gives the classical optimizer the freedom to choose all kinds of plans, including bushy ones. Moreover, GenJoin enables the classical optimizer to discover parts of the query plan space that it would not explore itself. For instance, the classical optimizer wanted to do $HJ(D, A)$ initially, but GenJoin recommends doing $NL(D, A)$ instead. This way, the classical optimizer might decide not to join (D, A), which pushes it towards initially discarded alternatives.

Why does GenJoin recommend only the join type and neither the type of scans nor the join order? All the LQOs, including GenJoin, rely on internal RDBMS subquery cardinality estimations based on pre-calculated statistics. We never know if the RDBMS considers a sequential scan over an index scan due to high predicate selectivity. Moreover, we also do not know what the RDBMS has already cached, and LQOs typically ignore what is currently indexed and how. This applies that recommending just the join type gives the RDBMS the ability to solve those parts where it is more knowledgeable than we are: when performing a merge join, one table would require sorting, so it might be beneficial to perform an index scan.

We choose not to recommend the full join order but only the join type to avoid overfitting and to enable the LQOs to generalize better in a smaller search space.

[4]The complexity of the potential prediction space is $O(T^2)$ since all pairs of tables T can participate in a subplan hint.

Why is GenJoin a representative of the next generation of LQOs? In Table 1 we present a comparison of GenJoin's architectural components against the state-of-the-art LQOs, as well as their predecessor LQOs. We emphasize the novelty of our method in terms of the following three categories:

- **Plan Hints**: GenJoin suggests the "golden middle" between forcing exact query plans and giving a set of binary query-level hints (see bottom row of Figure 1).
- **ML Model**: GenJoin uses a conditional generative model with variational inference to create a region in the latent space from where an improved plan can be sampled.
- **Output Plan**: Sampling from a region implies the need to generate multiple solutions and to apply a filtering step. For GenJoin, it suffices to take a single sample, since the formation of the region already performs an *implicit pruning*.

Table 1. Main architectural components of LQOs. Methods follow the chronological top-down order from oldest to newest.

LQO	Plan Hints			ML Model				Output Plan		
	Full-plan	*Subplan*	*Query*	*Regression*	*Reinforcement Learning*	*Learning-to-Rank*	*Generative*	*Single*	*Multiple w/ Sampling*[1]	*Multiple w/ Pruning*[2]
Bao [26]			✓	✓	✓			✓		
Lero [50]	✓					✓				✓
HybridQO [47]	✓			✓	✓					✓
COOOL [44]			✓			✓				✓
FASTgres [41]			✓	✓				✓		
AutoSteer [2]			✓	✓	✓					✓
GenJoin (ours)		✓					✓		✓	

[1] The method defines a region in the latent space where multiple plans could be selected.
[2] The method produces a number of intermediate plans during inference and afterwards applies a pruning algorithm to choose a single output plan.

The **major contributions** of our paper are as follows:

- We introduce *GenJoin, a generative conditional query optimizer* that learns from only three types of subplan hints (i.e. nested loop join, merge join and hash join) using a generative model with variational inference-inspired architecture. GenJoin enhances query plans by *pruning the search space of join types* of the built-in optimizer to greatly boost their query execution performance.
- We show, for the first time, that a *learned query optimizer consistently outperforms* PostgreSQL as well as state-of-the-art methods on the two well-known real-world benchmarks JOB and STACK across a variety of workloads using rigorous ML evaluations.
- GenJoin is not only optimized for producing plans with low execution times but also *minimizes the required inference time of the ML models* during query execution.
- Inside GenJoin, we introduce a *new way of measuring the distance between query plans* via unnormalized difference of execution times and the corresponding p-value of a T-test for the means of plan execution time samples, which can be both used for training purposes and for calculating the confidence intervals of the query optimizers' differences.

2 GenJoin Overview

2.1 General Setup

The basic idea of our approach is to start from some random query plan (represented via a set of hints[5]) and train an ML model to improve the initial query plan using certain conditions (like the context of the query) - instead of a blind initial search. The research question we address is as follows: *"How can we modify a set of hints such that the resulting query plan executes faster than the initial?"*

In order to be confident that the generated query plan results in an execution time that is not only faster than a random query plan, but also faster than the one produced by a built-in optimizer, we might need to apply our model multiple times on its own output. Such an architecture would be prone to overfitting (similar to other LQO methods that use model chaining for candidate plan selection and pruning). Instead, our model is designed to directly generate a query plan that outperforms PostgreSQL, which, as for now, is still state-of-the-art [19]. For that, we need to incorporate additional auxiliary information, revealing the next question: *"By how much is the target plan faster than the one produced by PostgreSQL?"*.

Combining these two questions, we find the answers with GenJoin - a generative hint-to-hint method with the goal of providing a hint set (rather than fully specifying the join order), which leads to a faster query plan compared to a given plan and the plan produced by the baseline method, namely PostgreSQL.

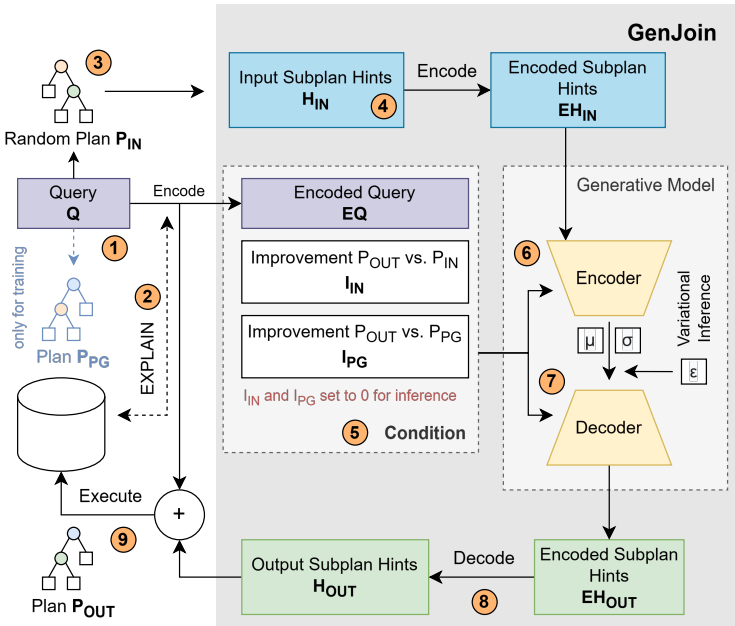


Fig. 2. GenJoin query processing: Illustration of GenJoin and how it interacts with the RDBMS when a query is executed. The model is based on the conditional generative architecture with variational inference and produces an improved set of subplan hints H_{OUT}, given a query Q and an initial set of subplan hints H_{IN}.

[5]Hereafter, we consider sets of subplan hints and query plans to be interchangeable, under the assumption that PostgreSQL turns a set of subplan hints deterministically into a query plan under static conditions.

Figure 2 depicts the individual components of GenJoin for executing a previously unseen query Q ①. If running without GenJoin, PostgreSQL would execute the query using plan P_{PG}. The query Q is encoded by querying the RDBMS for cardinality estimates of the filter predicates in Q by individually querying each table using EXPLAIN ②. Next, we randomly generate an input plan P_{IN} ③ based on the query ① and transform it into a set of subplan hints ④. The set of subplan hints H_{IN} is then encoded as EH_{IN} for the generative model ⑥. The condition of our model ⑤ is a combination of the encoded query EQ and two p-values that represent the improvement of the output plan P_{OUT} compared to the input plan P_{IN} ③ as I_{IN} and compared to the PostgreSQL plan P_{PG} ① as I_{PG}. During inference, the improvement values are set to zero. For additional information on the condition, see Section 2.4.1.

The encoded subplan hints EH_{IN} (namely, *nested loop join, merge join or hash join* of a given pair of tables) are concatenated with the condition ⑤ as the input to our generative model ⑥. With the output of the encoder, we perform variational inference and pass the output and the condition to the decoder ⑦. GenJoin predicts the improved subplan hints EH_{OUT} which are decoded to a set of subplan hints H_{OUT} ⑧, so they can be added to the original query as query hints using `pg_hint_plan` and executed as plan P_{OUT} ⑨.

During training, pairs of plans are used where we can measure the difference between the two plans and the improvement factors. This allows our model to learn the traits of a good plan and which operations are needed to significantly improve over an input plan.

2.2 GenJoin Encoding Scheme

A variety of query encodings has been used in LQOs [17, 26–28, 45, 48], with varying levels of complexity. A common practice is to split the encoding into two sections, the *query* and the *plan encoding*. The query encoding contains all information about which tables are involved in a query and what kind of filters are applied on which columns. The plan encoding specifies which tables are joined, what the order of joins is, and the types of join used. This type of encoding represents at each step of the join tree which tables have been joined so far and how, and what else is left to join.

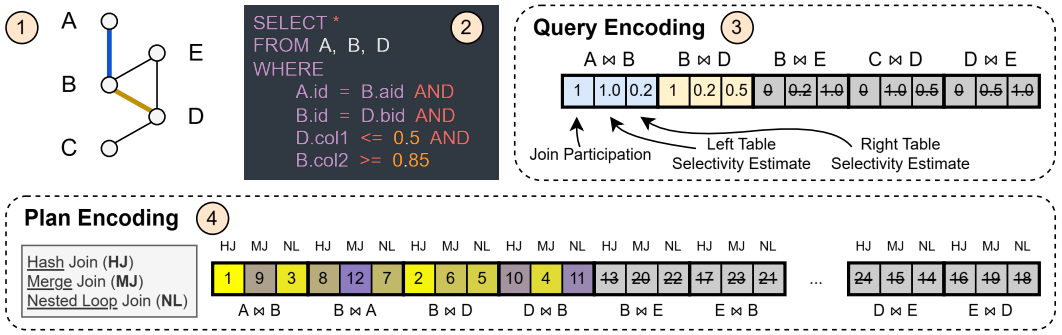


Fig. 3. GenJoin encoding: ① Shows a simple data model depicted as a graph with tables A through E. Edges denote potential join paths. The join paths for the query A ⋈ B ⋈ D are shown in blue and yellow. ② Depicts the SQL query with join and filter predicates. ③ Shows the *query encoding*. For each potential 2-way join, the encoding marks the join participation and the estimated selectivity per table. The participating edges A-B and B-D are marked in ① and colored accordingly in ③. ④ Shows the *plan encoding* which indicates which join to perform. GenJoin ranks the bi-directional join options. All greyed-out cells with strikethrough text belong to non-participating joins for the example query.

In case of GenJoin, the *query encoding* is similar to the one used in Neo [27] (and the majority of LQOs released after it). The *plan encoding* is new and specific for the given subquery hints, i.e., this

is the novelty coming along with the method itself. To illustrate our encoding scheme, we have prepared an example in Figure 3. Our proposed query encoding views a database as a graph (see ①), where each table is a node and every edge indicates a potential join path between two nodes. In the example, we can see that the five tables A through E have different ways to be joined, e.g., one can join tables A and B directly (indicated by the blue edge), but not B and C. As an example, we have prepared a query (see ② in Figure 3) joining tables A, B and D with predicates on tables B and D. Assuming the columns B.col2 and D.col1 are uniformly distributed between 0 and 1, they result in a selectivity of 50% and 15% for those two tables, respectively.

The *query encoding* (see ③ in Figure 3) contains one 3-sized cell for every edge in the graph of potential 2-way joins. In our example, that means there are five cells for the 2-way joins $A \bowtie B$, $B \bowtie D$, $B \bowtie E$, $C \bowtie D$ and $D \bowtie E$. Each cell contains one number to indicate whether this join participates in the query and two numbers corresponding to the estimated selectivity after all filter predicates are applied to this table. For example, the blue cell for $A \bowtie B$ contains $(1, 1.0, 0.2)$, since the join is part of the example query (see ② in Figure 3) resulting in a 1, table A has no filters applied resulting in 1.0 and table B with the filter $B.col2 >= 0.85$ resulting in an estimated selectivity of 0.2. Please note, that the selectivities of the encoding are extracted estimates (for example, from EXPLAIN calls to PostgreSQL) and not the true selectivity (hence 0.2 rather than the true 0.15 in the example).

The *plan encoding* (see ④ in Figure 3) uses the same idea of 3-sized cells like the query encoding, but encodes each edge in both directions (as there are significant differences in hinting $A \bowtie B$ versus $B \bowtie A$, e.g., depending on whether the tables are sorted or not). Unlike the query encoding, the plan encoding solely contains the information about *join types and rankings* thereof. Each cell contains three entries for ranking the hash, merge, and nested loop joins on this particular join. In our example, lower values mean a higher rank e.g., 1 being first rank. The cell for $A \bowtie B$ contains the ranks 1, 9 and 3 (ranked across all bi-directional join options), implying that GenJoin would force PostgreSQL to use a hash join if it were to join tables A and B in the order $A \bowtie B$.

One would assume from the example that PostgreSQL performed a hash join on $A \bowtie B$ (rank 1) and a hash join on $B \bowtie D$ (rank 2), since these are the highest ranking join types across all participating joins. However, GenJoin does neither fully specify the order in which the joins are executed nor in which order each pair of tables is to be joined. In our example, PostgreSQL could also decide to instead reverse either join as a nested loop join of $B \bowtie A$ (rank 7) and a merge join of $D \bowtie B$ (rank 4).

The idea behind the plan encoding is that GenJoin fully specifies all subplan hints provided to PostgreSQL by the highest ranking value in every cell. Moreover, individual rankings can also be understood as a confidence of the model in a particular subplan hint.

2.3 Model Architecture: Conditional Generative Model with Variational Inference

In this section, we describe the ML model architecture used in GenJoin. Our approach is inspired by conditional variational autoencoders (cVAE) [32]. We suggest an encoder-decoder architecture, include conditional information and apply variational inference as shown in Figure 2. We use a set of subplan hints as our input (the plan encoding) and use the query encoding as our condition. Likewise, the condition contains auxiliary information on the expected performance of the generated output (see Section 2.4.1 for more details). The output of our model is another set of subplan hints.

Until now, the potential of using autoencoders for query representation was presented as an idea [25], suggesting to optimize queries in the latent space. This way, the encoder is limited to only performing a dimensionality reduction by learning a *bijective* function that maps query plans into a latent space where semantic properties emerge. We make the encoder in GenJoin perform a different task instead: Our novelty is that we learn an *injective* projection of a *set of subplan*

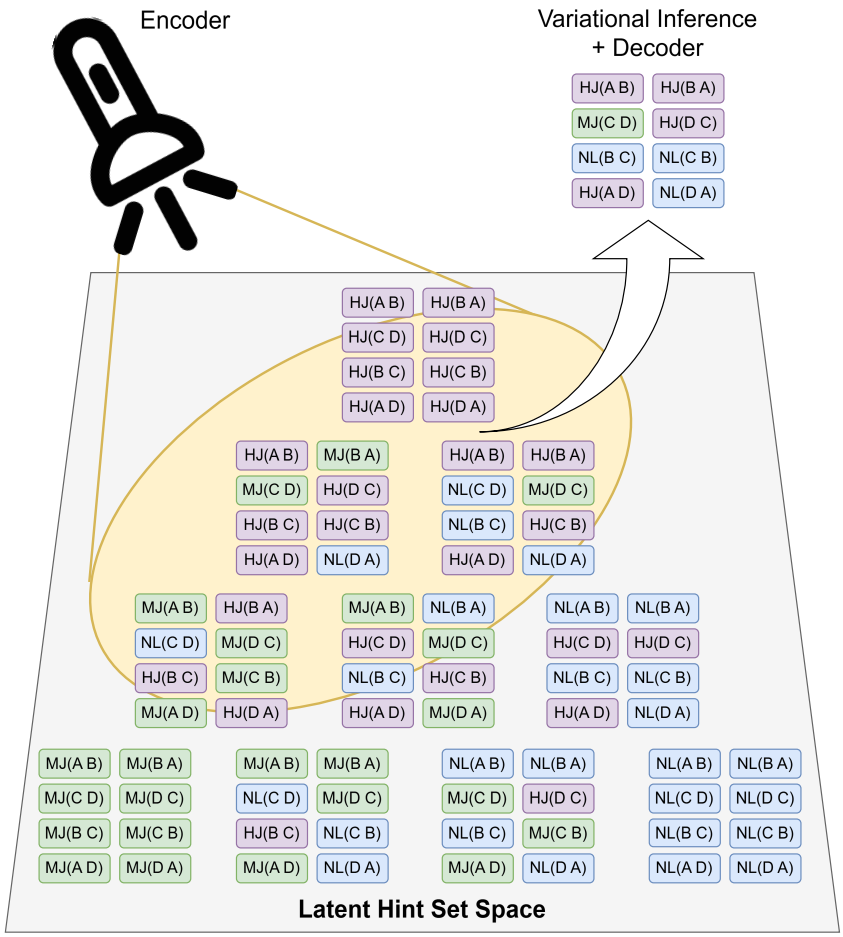


Fig. 4. GenJoin inferencing: The encoder makes a projection to the region in the latent space where hint sets leading to faster query plans are presumably located. A hint set is generated by sampling from the region using variational inference, followed by the decoder, which turns the latent representation into a hint set.

hints into a region of the latent space from which we can directly sample a superior set of subplan hints (see Figure 4) in a single shot without the need to apply additional latent space optimization algorithms, which could be both expensive and prone to error propagation. The advantage of our approach is that during inference, GenJoin guides the RDBMS to turn the input set of subplan hints into a faster-running physical query plan through latent space injection (which performs implicit pruning of the latent space to find the optimal region).

Why can we expect to find a faster set of subplan hints given a random plan? For any given input set of subplan hints there exist many sets of subplan hints (or consequently resulting query plans) that outperform the initial one, because the probability of randomly providing the fastest plan as input is low. These resulting sets of subplan hints form a clustered region in the latent space containing potentially faster query plans. Even if the optimal plan is provided as an input, GenJoin can provide sets of subplan hints that result in the same executed physical plan through dynamic optimization [3].

2.3.1 Model Design Details. For GenJoin, the encoder is designed to map the input (the query and plan encodings, as well as the conditioning information about the desired output plan) to a distribution in the lower-dimensional latent space, typically assumed to be a Gaussian distribution (as the inputs are not randomly projected, but rather constrained to lie within a restricted subspace). By sampling from this region, the decoder performs variational inference [8].

Training such a model is made possible by the "reparametrization trick" [10]. For example, if we decide to project our input into a standard normal latent space $N(0, 1)$, the encoder learns two parameter vectors μ and σ, aiming to keep the mean $\mu \approx 0$ and the standard deviation $\sigma \approx 1$. The decoder can randomly sample ϵ from $N(0, 1)$, add our μ, and multiply by our standard deviation σ:

$$z = \mu + \sigma \odot \epsilon \tag{1}$$

The result continues to follow the distribution $N(0, 1)$. With this trick, we have a differentiable Equation 1, allowing the gradient flow through the neural network since we delegate the random sampling to a noise vector, effectively decoupling it from the gradient flow.

2.4 Training Data Generation

Since GenJoin uses subplan hints both for its input and output, we have to generate training data specifically for our model in the form of pairs of sets of subplan hints, which implies our approach is data-centric rather than model-centric. The challenge is that, learning to predict a "better" set of subplan hints, the pairs need to exhibit a significant difference in execution time and are constructed such that the output set of hints always results in a faster query execution compared to the input set of hints.

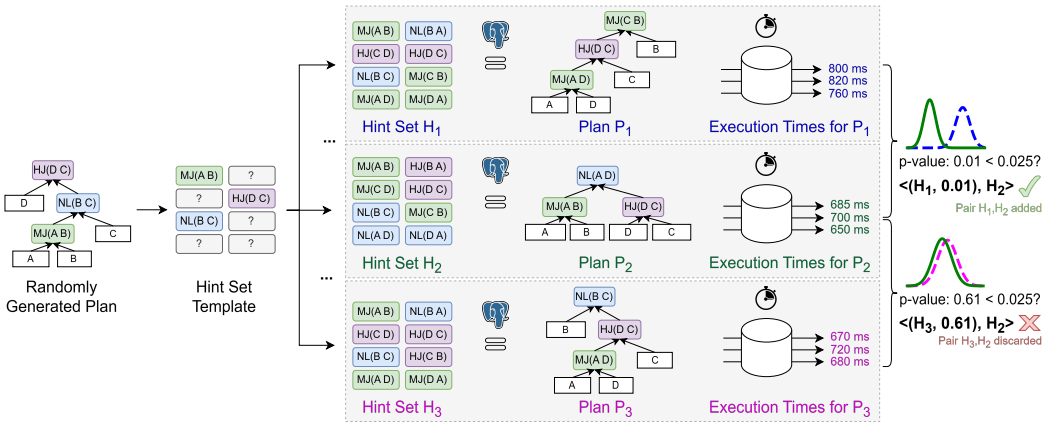


Fig. 5. GenJoin training data generation: We show three different sets of subplan hints H_1, H_2 and H_3 obtained from a randomly generated plan for query $A \bowtie B \bowtie C \bowtie D$. The sets are constructed from the physical operators of the given plan (such as hash join for $D \bowtie C$) as a hint set template. The remaining combinations of pairwise join types that are not present in the plan are chosen arbitrarily. The PostgreSQL optimizer, restricted by H_i, produces a physical plan P_i, that is executed 3 times. From these execution time measurements, we calculate the pairwise p-values of queries executed with hint sets H_i and H_j. Pairs with significant differences are selected for training. For example, we only keep the pair $< (H_1, condition), H_2 >$ where $(H_1, condition)$ is the input and H_2 the output of our model.

For every query in the training set, we randomly generate 200 sets of subplan hints H_i and pass them into PostgreSQL to measure the corresponding execution time e_i. It is worth noticing

that generating training data in such a way, i.e. treating PostgreSQL as a black-box environment, sending a set of hints, and receiving feedback in the form of a query plan, aligns with the concept of direct RL, which was advised in [19]. This approach exempts us from describing the environment, i.e. knowing the details of the internal PostgreSQL rules, but rather gives the possibility to learn how it reacts to a given set of subplan hints. See Figure 5 for an example with 3 sets of subplan hints for the query A ⋈ B ⋈ C ⋈ D.

We generate pairs of sets of subplan hints, which are combined with the global context of the query encoding and the condition, yielding tuples in the form of $< (H_i, condition), H_j >$. H_i represents the slower of the two sets of subplan hints and is fed to the model as input together with the condition (see Section 2.4.1 for more details), while H_j is the faster set of subplan hints representing the desired output. Since we are interested in learning a function that improves a given set of subplan hints, we only keep pairs for which there is a statistically significant difference between the execution times $e_i > e_j$. For this, we require a metric that provides us with a level of confidence for the difference between execution times and measures the distance between any two queries using sets of subplan hints. We conclude that the best choice is a p-value.

Why choose a p-value as a proxy for the distance between arbitrary query plans? First of all, since we know from [19] that the true execution time e_i cannot be exactly measured due to cache states and similar database properties, we use the mean estimation $\overline{e_i}$ calculated over a number of executions. Thus, measuring the unnormalized difference $\overline{e_i} - \overline{e_j}$ gives a distance metric, though it can produce values in an unbounded range from query to query. This may lead to learning inefficiencies and training instability for ML models. LQOs based on learning-to-rank models [6, 44, 50] utilize relative rankings instead, solving the ML issue mentioned above, though they lose the ability to measure how similar plans are.

One option to force the values of the unnormalized difference to be inter-query comparable is Studentization [15] - a division of a first-degree statistic derived from a sample by a sample-based estimate of a standard deviation - creating a scale-free metric. For the difference of means, such a normalization factor could be a pooled standard deviation s_p, i.e. a compound of standard deviations s_{e_i} and s_{e_j} with corresponding sample sizes n_i and n_j:

$$s_p = \frac{1}{\sqrt{\frac{{s_{e_i}}^2}{n_i} + \frac{{s_{e_j}}^2}{n_j}}} \tag{2}$$

The series of mathematical manipulations described above end up with nothing else but an empirical T-test [33] statistic for the means of the two independent query plan execution time samples. To achieve the property of distance significance, we compute the p-value associated with the created empirical T-test statistic:

$$\text{p-value}(\frac{\overline{e_i} - \overline{e_j}}{s_p}) \tag{3}$$

To ensure having only statistically significant differences, we keep only pairs with p-value $\leq$ 0.025, i.e. confidence of 5% for the one-tailed hypothesis. The p-value serves as a bounded metric for comparing the distance between queries. Being outlier-robust, the p-value enables ML models to learn from timeout queries.

How many samples should we choose for the p-value measurements? The p-value can only be measured between the distributions of query execution times, i.e. we need to collect ≥ 2 query samples from the RDBMS. Since we do not have a predefined minimal detectable effect nor a desired statistical power, we choose the number of query samples arbitrarily, aiming to have a

stable variance with the minimal executions-per-query required. Experimentally studying samples from standard normal distributions of different sizes and applying the elbow rule [36], we conclude that starting from *sample_size=3*, the change in variance is no longer significant.

This way, for each hint set pair, we compute the p-value of the T-test between the input and output execution times. We do the same between the output execution time e_{out} and the execution time of our baseline method (PostgreSQL) e_{pg}. Starting with the 3rd consecutive execution, we can trust the runtimes as an unbiased estimation of the true execution time [19]. Our empirical distribution is made up of the 3rd, 4th and 5th query execution.

How can the unnormalized difference be used for comparing LQOs? To make sure that one LQO has an advantage over the other in terms of execution time on a fixed workload of queries, we can utilize the unnormalized difference introduced above, using the pooled standard deviation s_p from Equation 2 for computing the confidence intervals aka error bars. For relative differences, it would be impossible to provide these kinds of error bars. One may ask, why using the approach for a single query difference can be extrapolated from a set of queries in a workload. We argue that as long as for the two normal distributions [21] X and Y, the following is true:

$$X \sim N(\mu_X, \sigma_X^2), Y \sim N(\mu_Y, \sigma_Y^2) \Rightarrow$$
$$Z = X + Y \sim N(\mu_X + \mu_Y, \sigma_X^2 + \sigma_Y^2)$$

and our execution time samples per query are assumed to be normally distributed, the methodology described in Section 2.4 above is generalizable for the sum of differences.

2.4.1 Conditional Information in GenJOIN. The conditional information allows us to specify what kind of output should be generated. To transform one set of subplan hints into another, we add supplementary contextual information through the query encoding (see Section 2.2), as well as information about the improvement in performance across the two hint sets. The level of improvement is captured with two distinct p-values: one approximates the difference between the performance of the query using the output set of subplan hints and the input set. The other p-value represents the difference in query performance between the output set of subplan hints and the set produced by PostgreSQL, answering the questions from Section 2.1, how much faster the target plan is, compared to both the input plan and the plan produced by PostgreSQL. During training, all plans and their execution times are available and thus all improvement p-values can be calculated. During inference, both p-values are set to the minimum value without the need to explicitly measure the performance of other plans.

2.5 Prediction of Subplan Hints

In the previous section, we outlined how the training data, and especially the pairs of input subplan hints H_{IN} and output subplan hints H_{OUT}, are generated. For the prediction, we also choose a randomly generated initial set of subplan hints as H_{IN} while our model predicts the subplan hints H_{OUT} (as shown after step 7 in Figure 2), which are then attached to the query and executed.

This leaves the question of what to assign to the p-values in the condition, namely for the confidence to surpass PostgreSQL and the confidence to surpass the input plan. Both of these values are set to zero, which corresponds to the theoretically best p-values under the hypothesis that the generated plan H_{OUT} performs better than both PostgreSQL and the input plan H_{IN}.

However, these p-values only produce a plan that is potentially better than the one initially given, contrary to the general aim of query optimization to find the best possible plan. Since the suggested approach considers the output H_{OUT} to be better than input H_{IN}, we can run the prediction in a

loop - a so-called "chain-of-subplan-hints" - assuming that every next output is faster than the previous input (see Section 3.5 for the results of this ablation study).

3 Experiments and Results

In this section, we provide our experimental evaluation of GenJoin on the Join Order Benchmark (JOB) and STACK. We start by describing the experimental setup, present our results of the experiments and finish with a number of ablation studies. Our code is available on Github[6].

3.1 General Setup

3.1.1 Software and Hardware. Planning and execution times are measured using `EXPLAIN ANALYZE`. The LQOs' inference time is measured as the wall time for e.g., preprocessing, encoding or inferencing. Measurements are taken in a hot cache setting by executing the same query three times and taking the last query execution.

PostgreSQL is configured as in [19], with `AUTOVACUUM` disabled to keep the sampled statistics consistent across the experiment. An `ANALYZE` call was run after setting up each workload. The experiments were run on machines with 64 GB of RAM, 16 CPU cores and Tesla T4 GPUs using a Docker environment.

We emphasize that a large number of publications are based on PostgreSQL versions 12 [6, 19, 26, 44, 45] and 13 [2, 50]. All our experiments were conducted using PostgreSQL version 16 to reflect the incremental changes added over time, further increasing the difficulty of outperforming the PostgreSQL baseline.

3.1.2 Query Workload. For our experiments, we evaluate all methods on two established query optimization workloads, namely the Join Order Benchmark (JOB) [20] and the StackExchange-based STACK [26]. These two real-world benchmarks give good insights into the overall performance of various methods, including many queries that are hard to optimize [20]. JOB has been the primary benchmark for query optimization in recent years, with STACK in the runner up position since its release in 2021. Both benchmarks show skewness in the data and violate the uniformity assumption.

Join Order Benchmark (JOB). JOB is comprised of queries around a snapshot from May 2013 of the popular Internet Movie Database (IMDB). It contains 21 tables stored in a relational database. The queries have, on average, 8 joins per query with a maximum of 16 joins. In total there are 113 queries that come from 33 base queries (or templates) that each have between two and six variations, where either filter predicates are changed or entirely new attributes are being filtered on. These predicates have a significant impact on the selectivity of the tables involved. Hence, the optimal physical plan might differ from variant to variant. These differences in execution time can span multiple orders of magnitude. However, well-performing plans often share partial subplans with other variants, creating a potential for data leakage.

STACK. Introduced alongside the Bao method [26], the STACK workload and dataset is based on data from the StackExchange websites. STACK features 10 tables and over 6,000 queries from 16 base queries (or templates). The queries follow a similar variation style as in JOB, meaning there exist many different variations in the filter predicates for every base query available. As per [19], we have downsampled the numbers of queries again to keep the comparison between methods at a similar statistical power. For this reason, we use 112 queries with 7 variants for each of the 16 base queries.

[6] https://github.com/edualc/genjoin

3.1.3 Training Data Generation. For our workloads with 112 and 113 queries and 200 randomly generated hint sets per query, we can potentially generate up to 4.5 million pairs[7]. Due to the limitation of only selecting them in order of execution time and with significant differences, the resulting number of usable training pairs is on the order of about 1 million. We subsample the actual amount of pairs and evaluate the ratio of pairs for training using the Optuna hyperparameter optimizer[8]. For JOB we use 30% of all pairs during training, for STACK it is 45%. These ratios serve as a surrogate for classical early stopping mechanisms because our training data volume regulation is aimed to reduce the generalization error [4]. In line with other LQOs, supporting a new database requires generating new training data and train a new model.

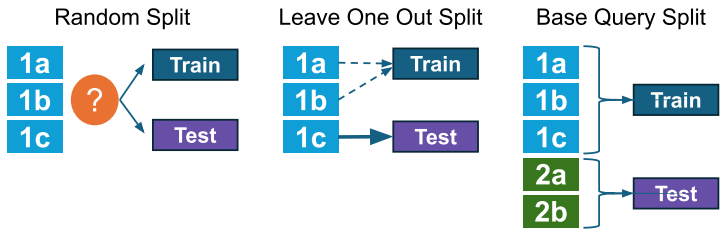


Fig. 6. Summary of the dataset split types used in order: Random Split, Leave One Out Split and Base Query Split.

3.1.4 Dataset Split. We continue using the different types of splits described in [19], namely the Random, Leave One Out and Base Query split types. These dataset splits take advantage of the structure of the query workloads, where queries can be clustered together as having different filter predicates, but an overall shared structure. These shared structures are called *base queries*, with the different variations of the filter predicates being called base query variants. Figure 6 gives a summary for how the queries 1a, 1b and 1c belonging to the same base query 1 would be assigned to the train and test set. In the *random split*, queries are randomly assigned to the train and test set. In the *leave one out split*, all variants of a base query except one are put into the train set. Finally, the *base query split* assigns all base query variants either to the train or test set, respectively. Hence, queries 1a, 1b and 1c end up in the train set together, while queries 2a and 2b are collectively put into the test set. While the *base query split* aims to reduce data leakage across shared query structures, the *leave one out split* forces a minimum level of shared information between the training and test sets.

The various splits emphasize different characteristics of the query workloads and give strong overall insights into the models' performances. We have made one adjustment compared to [19]: For every type of split, we distribute the data according to the split function into three separate folds in the style of k-fold cross-validation. The three splits are then called A, B and C, e.g., "Base Query Split A", "Base Query Split B", and "Base Query Split C", where the *test sets together form partitions of all queries*. Since the test sets of splits A, B and C form a congruent set of all queries, we list the performance of a model on that type of split as the combination of all three test sets. This allows us to make a more accurate, direct comparison across split types.

There are two special cases that need to be handled: First, if a group of the same base query has a number of variants that is not divisible by three, the remainder is randomly assigned between the

[7] $112 * 200 * 200 \approx 4.5$ million, though we are interested in an imposed order of their execution time, which leaves half that amount before checking significances.
[8] https://optuna.org/

three folds. Second, if there are fewer than three query variants, some variants will be assigned to multiple splits. In the latter case, the evaluation takes the worse of the two query's measurements.

For a number of ablation studies, we also use the *slow split*, first introduced by [45] for JOB as *JOB-Slow*. This takes the slowest N queries (for JOB-Slow in Balsa $N = 19$) into the test set and maximizes the absolute amount of time that can be optimized in query executions. The extension of the slow split for STACK follows the same notion and includes the 19 slowest queries in the test set.

3.2 Comparing GenJoin against Current State-of-the-Art Learned Query Optimizers

In this section, we compare the performance of GenJoin with two state-of-the-art LQO methods - whose source code is available, run with PostgreSQL Version 16, and the results are reproducible - namely, HybridQO [47] and AutoSteer [2]. We have chosen HybridQO since it outperformed Neo [27], Balsa [45], Bao [26] and LEON [6] in previous evaluations [19].

Bao has been extensively used for comparison in recent LQO publications, but because it does not natively run under PostgreSQL version 16, we needed an alternative for the query-level hint methods. While both AutoSteer and FASTgres [41] improve upon Bao, AutoSteer adds a dynamic exploration of tuning knobs in addition to a much larger number thereof, while FASTgres primarily uses a different type of model with the same number of knobs as Bao. Therefore, we have chosen AutoSteer for our evaluation, which has a more expressive optimization space over FASTgres and Bao.

We have also considered including COOOL [44], but the experiments could not be reproduced[9] - similar to other non-reproducible methods discussed in [19]. For additional details on other LQO methods, we refer to our Related Works in Section 4.

Figures 7 and 8 show the performance of GenJoin compared against HybridQO and AutoSteer executed on JOB and STACK, respectively. The methods are evaluated along the three types of splits outlined in the previous sections, in order of difficulty, starting with the leave one out split, then the random split and finally discussing the base query split. Please note that **the level of HybridQO and AutoSteer performance is different compared to the one stated in original papers** [2, 47] due to the usage of fair training and testing procedures justified in [19].

In Figures 7 and 8 we compare the methods against PostgreSQL since executing queries faster than the classical reference is the primary goal. The x-axis describes the difference in *planning and execution time* summed over all test sets of that split. This way of measurement is based on the unnormalized difference, introduced in Section 2.4. We argue that using other methodologies, like those comparing sums of workload queries side by side, are statistically inaccurate, as they do not construct any hypothesis with an empirical and theoretical test metric, i.e. cannot provide significance in terms of confidence intervals.

Note that the inference time of the ML models is not included in this first section, but we will analyze the results including the inference time in the following sections. The error bars indicate the statistical significance. Since a value of zero on the x-axis would correspond to an identical performance to PostgreSQL, any bar into the positive (right side) indicates that a method is faster than PostgreSQL. Conversely, bars into the negative (left side) of the x-axis indicate the opposite, i.e. the method is slower than PostgreSQL. Whenever the error bars cross over the zero position, the difference in performance is not statistically significant. In summary, whenever a bar is going

[9]We have contacted the authors of COOOL because their published code only includes model checkpoints and the code for inference, but lacks the required scripts for training a model. The authors did not provide the necessary training code, under the motivation that "posting the training code may confuse readers because the results in the paper cannot be obtained via re-training".

to the positive side with an error bar that does not overlap the zero position, then that method outperforms PostgreSQL by a statistically significant margin.

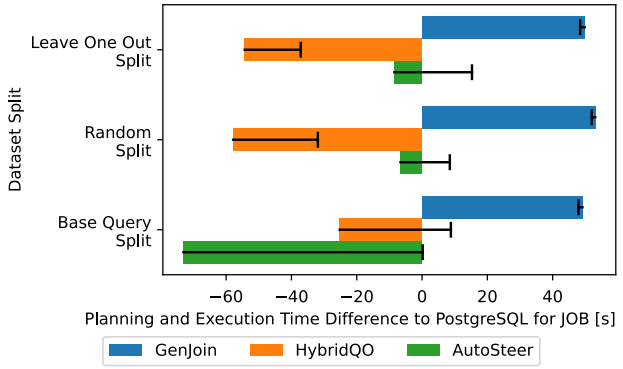


Fig. 7. Results on the Join Order Benchmark (JOB) without inference times. Methods are compared against the PostgreSQL baseline. If the bars extend into the positives, the methods are faster than PostgreSQL and vice versa.

We start by looking at the results on JOB shown in Figure 7. On the *leave one out split*, what is considered to be the easiest type of split, GenJoin is the only method that significantly outperforms PostgreSQL by 50 seconds taking 104 seconds on average, compared to 154 seconds for PostgreSQL. AutoSteer finishes the queries in 163 seconds, i.e. it is 9 seconds slower than PostgreSQL, while HybridQO requires 209 seconds, double the time of GenJoin.

The *random split* shows an almost identical performance for all methods compared to the previous split without drastically changing the amount of time spent on planning and executing queries. GenJoin is 53 seconds faster than PostgreSQL on average, AutoSteer is 7 seconds slower than PostgreSQL, and HybridQO is 58 seconds slower. While some fluctuations are natural, this shows that the way the queries are distributed between leaving one out and random does not have an impact on the models' performances.

Finally, for the *base query split*, there is less overlap between the seen training queries and the test queries. As for the previous splits, GenJoin demonstrates a stable lead over PostgreSQL being 49 seconds faster. However, HybridQO and AutoSteer have now switched ranks with HybridQO performing better than on other splits, albeit still 25 seconds slower than PostgreSQL. AutoSteer is most impacted by the different split method where queries are distributed to generate the adversarial case where most useful information between similar queries is hidden. AutoSteer now requires over 73 seconds more than PostgreSQL, although due to the large confidence intervals, AutoSteer is not statistically significantly slower, as there are large fluctuations in the three separate models trained.

Now that we have analyzed the results for JOB, we take a look at the same results for STACK in Figure 8. Compared to JOB, the queries in STACK are executed 50% faster on average, which leaves less room to be optimized.

We will first analyze the *leave one out split* for STACK, where both GenJoin and AutoSteer finish the queries faster than PostgreSQL. AutoSteer is 15 seconds and GenJoin 6 seconds faster than PostgreSQL, which took 81 seconds in total. HybridQO performs rather unstable with a massive confidence interval and takes 10 seconds longer than PostgreSQL.

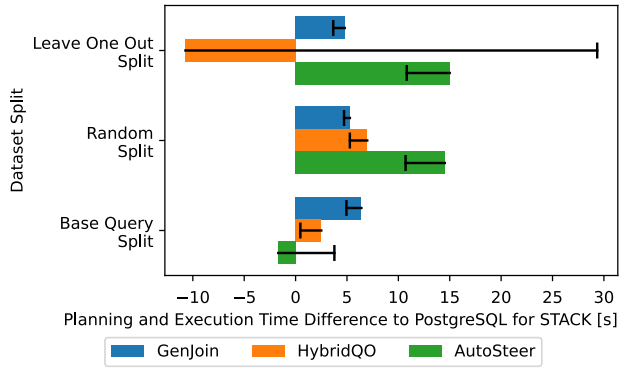


Fig. 8. Results on STACK without inference times.

On the *random split* for STACK, we observe a nearly identical performance for GenJoin and AutoSteer, with both methods showing no change in their behavior in the different split types. Furthermore, HybridQO also outperforms PostgreSQL by 7 seconds. In summary, for the random split all three methods outperform PostgreSQL by a statistically significant margin.

Finally, for the *base query split*, GenJoin cements its stable mode of operation, 7 seconds faster than PostgreSQL. HybridQO is in second place, being 3 seconds faster. Lastly, AutoSteer's performance once again suffers on this split, even though it was the best method on the previous two splits for STACK. Even so, it still reaches a comparable time to PostgreSQL being just 2 seconds slower.

Over both JOB and STACK, we have observed that GenJoin is the only method that consistently outperforms PostgreSQL on all splits by a statistically significant amount of time. While AutoSteer finds better plans on two out of three splits on STACK compared to GenJoin, it never outperforms PostgreSQL on JOB. Similarly, HybridQO also displays a much more unstable performance that is much worse on JOB and barely better than GenJoin on one split of STACK. These results not only show the strength of our method in finding better plans but especially highlight an even more important ability: stability in different environments. However, across all evaluated LQOs there exist queries with degraded performance when analyzed individually.

3.3 Measuring Query Processing Time

In the following paragraphs, we outline the different measurements in evaluating an *end-to-end execution time*, which includes:

(1) **Inference Time**: Any amount of time spent on other data processing that is not directly part of planning or executing a query (primarily for LQOs).
(2) **Planning Time**: The time investment by the RDBMS to analyze a query and optimize its query plans. Note, that LQOs do not skip the planning time, as the RDBMS optimizer applies further optimizations.
(3) **Execution Time**: How long the database engine takes to execute the physical plan and extract the result set.

The primary goal of evaluating query optimizers based on the end-to-end execution time is to include all significant parts of the overall time spent in query execution while at the same time focusing only on the aspects where classical and LQO methods can have a direct impact. As such, network latency (i.e. in establishing a connection and sending the query to the RDBMS) is not included.

3.3.1 Inference Time. With the introduction of LQO methods, there is now a need to also include inference time in the evaluation of the execution time, as learned methods can take a significant amount of time to produce their result[10] (e.g., encoding a query, using an ML model for inference or deciding what set of hints to pass to the RDBMS), before the RDBMS plans and executes a query. Inference time includes the prediction time of a model (e.g., neural network predictions), but also pre- and postprocessing steps (e.g., parsing the SQL statement, encoding a query into a vector representation for the model and turning it into an actionable hint set that can be prefixed to the original query) and additional EXPLAIN queries to fetch structural or statistical information from the classical optimizer (e.g., cardinality estimates for the encoding step). For a classical optimizer like the one used in PostgreSQL, this corresponds only to the planning time, as no additional processes are run. However, in the case of LQO methods, indicating this time as a separate entity allows us to compare the impact of the methods on the overall amount of time spent waiting for a query result. For this reason, we urge the community to focus not only on LQOs that produce good plans but also on methods that are efficient at inference.

With this new perspective, the goal of query optimization evaluations changed: While the execution time remains the primary metric to improve, it becomes equally important to find the plans, hint sets, or other LQO results in a reasonable amount of time. This begs the question: What is a *reasonable* amount of time?

When considering inference time as a variable amount of time with a lower bound (e.g., because the encoding requires an EXPLAIN call that is on the order of 50 milliseconds), then the LQO should improve the execution time by at least the same amount to not become slower than the classical optimizer.[11]

3.3.2 Planning Time. Arguably a smaller subset of the overall time spent executing queries is the planning time of the RDBMS. For LQOs that give hints or otherwise restrict the space of options, we could assume that less time is spent compared to the classical baseline. However, it has been shown that even in those cases, the planning time differs greatly and can grow by a factor of up to 5 depending on the method and workload [19]. We want to stress that just because an LQO produces a plan for the RDBMS to execute, does not mean that the planning time can be skipped.

3.3.3 Execution Time. Naturally, execution time is the primary optimization objective with the most impact for optimizers (classical and learned). We observe, in particular for the STACK workload, a high variance in execution times, which exacerbates the problem of achieving consistent measurements.

3.4 Revisiting Results under Inference Time

Having discussed the importance of including inference time, we now revisit the results by measuring the end-to-end execution time of all methods against PostgreSQL. The goal is to assess if the LQOs not only find a plan that executes faster but also that the methods do so efficiently such that the overall execution time is reduced.

We start with the Join Order Benchmark again, for which the results could be found in Figure 9 now using the end-to-end execution time (i.e., including inference time).

On the *leave one out split* GenJoin is still faster than PostgreSQL, but now only by 24 seconds. That means, half of the advantage vanishes in using GenJoin and finding an improved plan. Both HybridQO and AutoSteer are now significantly slower than PostgreSQL, with HybridQO needing

[10]We focus on the amount of time required to process a *previously unseen* query.

[11]A different perspective is to say, that the amount of time spent categorized as inference time would not have existed if an RDBMS did not use an LQO.

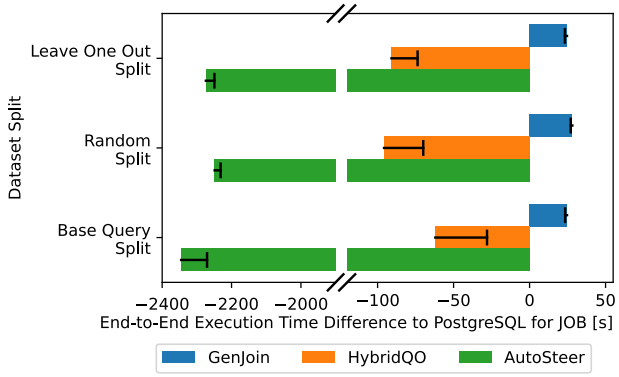


Fig. 9. Results on the JOB using end-to-end execution times (incl. inference). Note the break on the x-axis.

an additional 36 seconds for finding its solutions. Finally the biggest jump is seen in AutoSteer, where the inference takes on the order of 37 minutes. To some extent, this is not surprising, as AutoSteer tries out a multitude of plans through EXPLAIN calls to find the hint set configurations, which lead to different plans than the PostgreSQL default plan. Since it is required to gather a subset of plans that differ to predict on an unseen query, this time is included in its inference time (and represents the dominant factor).

Looking at the *random* and *base query splits* reveals nothing new: GenJoin consistently loses about half its benefit in inference time (around 25 seconds), but can outperform PostgreSQL on all splits. HybridQO stays slower than PostgreSQL by an additional 37 seconds of inference time across both splits. AutoSteer again uses about 37 minutes for inference remaining in the last spot.

Interestingly, these results indicate that the way the queries are split into train and test sets has no impact on the amount of time spent producing a result, showing these times can be seen as a constant across a workload.

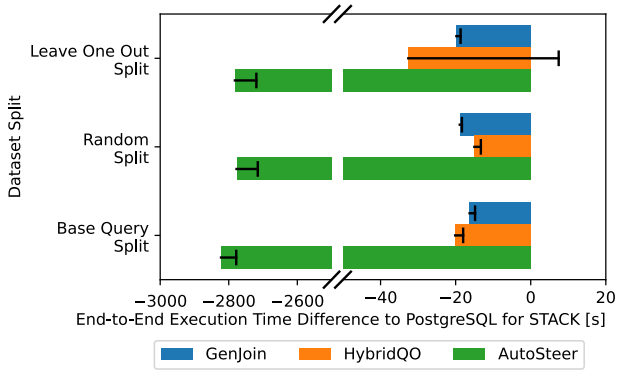


Fig. 10. Results on STACK using end-to-end execution times (incl. inference). Note the break on the x-axis.

Now we will analyze the results on STACK using Figure 10. As we have discussed in Section 3.2, the queries in the STACK workload leave less room for improvement in terms of absolute time spent. We can observe now that all methods, including GenJoin, are not able to beat PostgreSQL

anymore on any type of split, with GenJoin and HybridQO reaching similar end-to-end execution times overall. While GenJoin's inference time is very consistent at around 24 seconds, HybridQO was able to find its result more quickly for STACK in 22 seconds, compared to 36 seconds for JOB. AutoSteer needs even more time for inference at 47 minutes.

We attribute this drastic change to the composition of queries in STACK, which includes about 50% more queries than JOB that take less than a second. The queries impose a harsh limit on the amount of time that can be used for inference and require that LQOs massively improve over the execution time of PostgreSQL. Due to the way current LQOs encode queries using EXPLAIN calls (which is typically the dominant factor for inference time) to gather auxiliary information from PostgreSQL, it remains an open question of how to encode queries and plans more efficiently.

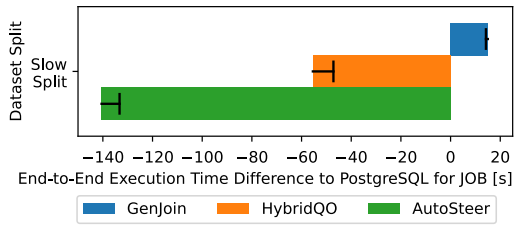


Fig. 11. Results on JOB using end-to-end execution times (incl. inference) showing the *slow split*.

Can a LQO outperform PostgreSQL on STACK? To answer this question, we can first look at the *slow split* with the 19 slowest queries from JOB, where the results are identical to the other splits (refer to Figure 9): GenJoin significantly outperforms PostgreSQL (see Figure 11). However, both HybridQO and AutoSteer do not outperform PostgreSQL due to their large inference time.

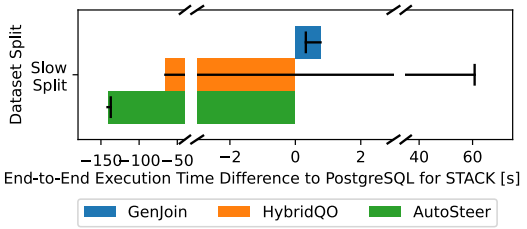


Fig. 12. Results on STACK using end-to-end execution times (incl. inference) showing the *slow split*. Note the two breaks on the x-axis to accommodate all orders of magnitude.

Figure 12 illustrates the performance on the slow split of STACK, which contains the 19 slowest queries in its test set. We see now that GenJoin can overcome the constraints on the inference time and significantly outperform PostgreSQL. In particular, the test queries take long enough such that it becomes worthwhile to use GenJoin. HybridQO demonstrates a large variance in its end-to-end execution time due to instabilities in its predictions but takes over 60 seconds on average longer than PostgreSQL. Finally, AutoSteer spends over two minutes longer than PostgreSQL.

The experiments have shown that while both HybridQO and AutoSteer are able to find good plans, their inference time becomes a bottleneck to match or outperform PostgreSQL. Additionally, the different split types have shown large fluctuations in these methods' performances, with AutoSteer especially struggling on the adversarially constructed base query split. While GenJoin demonstrates

a very stable performance across both workloads and various split types, it did not find the best plans on STACK. These results show the need for more efficient encoding approaches that do not require EXPLAIN calls, lowering the overhead cost of inference time for fast queries, posing the question of whether an LQO is even necessary for queries that are expected to finish in < 1 second.

3.5 Ablation Study: Chain of Subplan Hints

With a method that generates a set of subplan hints given a set of subplan hints, the question arises if applying the model multiple times on its own output might yield even better results. This approach referred to as "chain-of-thought prompting" [39] in the space of large language models (LLMs) such as OpenAI's ChatGPT 4o model[12], has shown to be rather effective in guiding the LLM to better responses[13]. We have conducted an experiment where we let GenJoin start with a random set of subplan hints H_0 to predict the first iteration H_1 and, from then on, always use the previous output H_{n-1} as the subplan hints input for H_n (up to $N = 25$).

The major drawback of the chain of subplan hints idea is that (a) it is not trivial to find the optimal number of iterations for each query and (b) with every additional iteration we spend more time making inference, which is already a limiting factor. However, the most costly aspect of GenJoin's inference time lies in the EXPLAIN queries to gather cardinality estimates from the RDBMS. To run additional iterations, however, the query encoding remains constant, and no additional EXPLAIN calls need to be executed, drastically speeding up any iterations after the first.

We have not observed a degradation in the execution time across 25 iterations, but rather minor fluctuations which could be explained by the natural swings in execution time. We also observed no significant improvement on the subplan hints generated after more than one iteration, showing that the additional time spent doing inference is not worth it for the overall end-to-end performance. Moreover, the results demonstrate the stability of the subplan hints generated by our method, confirming the validity of using randomly generated subplan hints over a pre-optimized set of hints, such as e.g., the set of hints contained in PostgreSQL's physical plan.

4 Related Work

In this section, we offer a short introduction to LQOs. We observe two distinct groups of methods that generate *full plan-level hints* and *query-level hints* (see Figure 1), a number of *meta-LQO* methods [1, 40], and *environments* for ML-driven query optimization [38, 51].

The first ML-based query optimizers that apply RL are DQ [17], ReJOIN [28, 29], and FOOP [12], predicting *full plan-level hints*. To find the optimal join order, an exploration-exploitation strategy was applied using a reward cost model.

One family of methods originated from Neo [27], which was the first end-to-end method using a neural network to predict the latencies of a full query plan combined with a greedy bottom-up plan creation. Balsa [45] followed the overall structure of Neo, but introduced a variety of modifications to the training procedure, most prominently timeouts for query executions.

Another family of LQOs started with RTOS [48], a method focusing on the sequence of two-way join operations disregarding join and scan types. Compared to Neo, RTOS uses a depth-first search for plan construction, also using an RL agent. LOGER [5] follows RTOS, extending the action space by including the types of joins.

A new paradigm of changing the space of RL-based latency predictors was the emergence of learning-to-rank methods, such as Lero [50] and LEON [6]. Lero generates a number of candidate plans by feeding alternate cardinality estimations into the PostgreSQL optimizer. A plan comparator

[12]https://openai.com/index/learning-to-reason-with-llms/
[13]E.g., the infamous "let's think step by step" prompt.

model then predicts, given a pair of plans, which one to execute. LEON generates candidate plans through brute-forcing and pruning before the model ranks the candidate plans based on their estimated latency and uncertainty. LEON can be seen as the learning-to-rank continuation of Balsa.

Finally, HybridQO [47], as the name implies, combines cost and latency predictions. It generates candidate plans through hinting with the help of a Monte Carlo tree search where a model based on RTOS predicts their costs. The same type of network is then used to predict both the latency and uncertainty to inform the final model in choosing which plan to execute.

Now we will focus on the LQO methods that generate *query-level hints*, emphasizing the main differences between GenJoin and other hints-based methods. In addition, we refer to [35] for a more detailed review of optimizer hinting.

Bao [26] was the first method that does not produce complete query plans from scratch but uses *hint sets* that prune the optimization space of a given plan. In total, Bao uses a set of 6 scan and join type hints (though they experimented with up to 2^6 combinations and chose the best-performing subset). However, the hints are on a query level, rather than on a subplan or table level.

COOOL [44] uses pre-defined hint sets as a model input (contrary to other hint sets-based methods like Bao for which the hint set is the output of ML model) along with the query and plan encodings, and applies a learning-to-rank model to choose the best hint set according to the estimated execution time.

FASTgres [41] uses hint set prediction like Bao (with the possibility of many more sets, essentially 2^6 as a result of multi-class classification), primarily modifying the ML model, choosing a regression model instead of a bandit optimizer.

AutoSteer [2] expands upon the idea of Bao, where the space of query optimization rules is explored to find the best set of hints that globally enables or disables parts of the optimizer. While Bao uses 6 different scan and join hints, AutoSteer extends these to 20 hints for PostgreSQL (resulting in potential search space of 2^{20} hint sets), massively increasing the combinatorial space to be explored. AutoSteer further experiments on a variety of RDBMS where the number of hints (also called knobs) varies by an order of magnitude across systems. Instead of checking all combinations, AutoSteer iteratively checks if the hints make the optimizer deviate from its initial optimized plan and further merges these hints if they do.

GenJoin considers a more limited amount of hints, in particular for the types of join, and applies them for each relation individually, rather than globally. GenJoin also does not need to explore which combinations of hints would make the optimizer deviate from its original plan, significantly reducing the amount of time spent optimizing an unseen query.

5 Discussion and Lessons Learned

Symbiosis instead of competition. While the trend of working in tandem with the built-in optimizer is not completely new, designing the GenJoin method to only restrict its operations through hints and leaving a high degree of freedom in the hand of the built-in optimizer has proven to work well. This allows the LQO to focus on a more narrow problem, in turn reducing the inference time and improving the generalization ability.

A new distance metric for comparing query plans. Not only has the T-test comparison solved the problem of stabilizing the range in which we compare plans in GenJoin's training data generation, but we could also show its effectiveness in evaluating LQO methods. Thanks to this, calculating the statistical significance has become straightforward. Moreover, this distance metric produces a bounded output that has no units, is robust to outliers, and allows for the interpretability of the p-values.

Tackling the complexity of the prediction space. Analyzing the methods based on the categorization in Figure 1, we identify three different classes of complexity for the space of potential

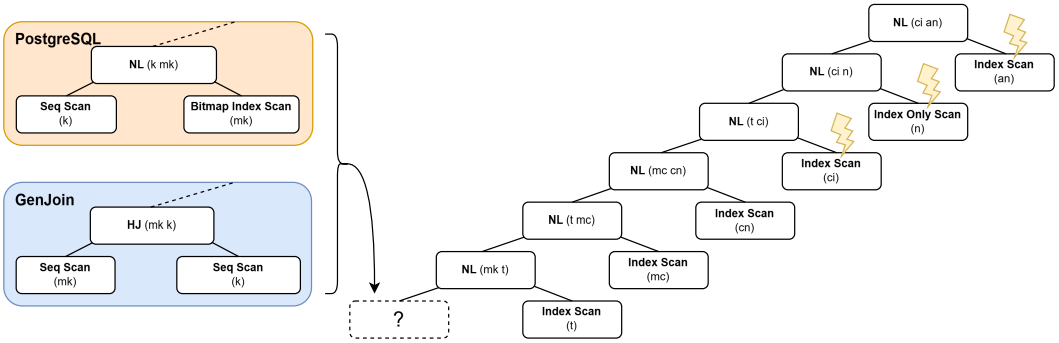


Fig. 13. Illustration of the difference between the plans selected by GenJoin and PostgreSQL for Query 16b from the Join Order Benchmark. Differences at the beginning of the join tree have significant impact on the final three joins, since the Bitmap Index Scan loses the indexed ordering required by downstream nested loop joins that rely on index conditions.

predictions: full plan-level hint methods with $O(T^T)$[14], query-level hint methods with $O(2^K)$ and GenJoin with $O(T^2)$ (T represents the number of tables and K the number of different hint sets). The first two classes require to process a potentially exponentially growing number of solutions. Because of this, they adopt a two-model approach, where one model prunes the search space and the other model finds the best solution in the pruned space (or vice versa).

GenJoin scales in the worst case polynomially with the number of tables, which does not require an explicit pruning step, allowing GenJoin to spend less time both for training and inference. In addition, having fewer components and trained parameters simplifies the overall architecture and leads to a more robust solution. Nevertheless, GenJoin follows the paradigm of a two-step approach explicitly by pruning the space of potential predictions during the training data generation, keeping only the training pairs for which the difference in execution time is statistically significant.

Cardinality estimations and `ANALYZE`. It is evident that the way the internal statistics of PostgreSQL are sampled has a profound effect on the performance of the built-in optimizer. LQOs are equally affected by changes in the internal statistics, for example, from explicit `ANALYZE` calls or through incidental updates via `AUTOVACUUM`, as their performance is strongly correlated with the internal state of PostgreSQL's statistics. An interesting consequence of this realization is that reproducibility is further challenged not only by the available hardware, the database configuration, and the random seed during the training of an ML model but also by the sampled values of PostgreSQL's internal statistics.

The nature of GenJoin's advantage. In the majority of the cases where GenJoin significantly outperforms PostgreSQL, there is a common pattern of query plan tree change. We can demonstrate it using as an example query 16b from JOB. This query behaves stably among the splits, resulting in roughly 13.2 seconds for PostgreSQL and 4.3 seconds for GenJoin. We have depicted the key differences of the query plan trees between PostgreSQL and GenJoin in Figure 13. The only deviating part is a first action at a left deep tree, where PostgreSQL chooses to perform $NL(k, mk)$ and GenJoin prefers starting with $HJ(mk, k)$. Such a tiny, at first glance, change led to drastic consequences, since (1) for the nested loop join PostgreSQL used a fast BitMap Index Scan over mk, while GenJoin had to switch and run a longer Seq Scan (as the join type defines the scan type as discussed in Section 1); (2) as a result, for PostgreSQL the mk lost the original indexing [18], and could not be

[14]There are $T!$ different permutations for T tables and for each permutation there are $C(T-1)$ possible binary trees, where $C()$ is the Catalan number. Simplifying, we end up with asymptotic complexity of $O(T^T)$.

used efficiently for further Index Scans by *ci*, *n* and *an* - for GenJoin it is not an issue. This way, we see a clear example of how classical greedy optimizers might win in subplan performance (local optimum), while the generative approach would outperform on the plan level (global optimum).

Lessons from competitor performance. Both HybridQO and AutoSteer consistently predict changes from the default PostgreSQL plan, even if the plan chosen by PostgreSQL is optimal. In fact, between 5 and 13% of queries (depending on the dataset split) are best executed without changing the default plan. These queries seem to drive the observed performance degradation on the base query split for AutoSteer. Furthermore, HybridQO always predicts a single `Leading(t1 t2)` hint specifying the first two-way join. Unlike AutoSteer, HybridQO can propose a hint that matches PostgreSQL's default plan, which happens in approximately 21% of cases.

6 Conclusion and Future Work

We have introduced GenJoin, the first generative LQO that consistently outperforms PostgreSQL in terms of execution time. GenJoin is able to compete with and outperform current state-of-the-art methods, in particular in terms of the amount of time spent for inference. We have demonstrated that GenJoin behaves very stably across a variety of workloads and train/test split characteristics.

The experimental results have highlighted a need for encoding strategies that are more efficient at inference. We plan to investigate how GenJoin, or LQOs in general, can produce query and plan encodings that conserve a similar level of expressiveness without the need to call the RDBMS for cardinality estimations or other auxiliary information through EXPLAIN queries. In addition, there remain many situations that are badly captured by current encodings, such as tables appearing multiple times in the same query through multiple aliases, multiple attributes on which to join the same two tables, or performing self-referential joins.

We acknowledge the recent trend in bringing LQOs into industrial applications [1, 2, 51] through the use of meta-LQO methods [40] and database environments [38] for faster iteration and reduced performance regression. With the novel architecture of GenJoin and its training data generation, we will explore how to further apply it in more challenging settings, as well as in other RDBMS that support subplan hints (e.g., Oracle or SQL Server).

Acknowledgments

The project received funding from the Swiss National Science Foundation under grant number 192105. This work was also partially supported by DataGEMS, funded by the European Union's Horizon Europe Research and Innovation Programme, under grant agreement number 101188416.

References

[1] Remmelt Ammerlaan, Gilbert Antonius, Marc Friedman, HM Sajjad Hossain, Alekh Jindal, Peter Orenberg, Hiren Patel, Shi Qiao, Vijay Ramani, Lucas Rosenblatt, et al. 2021. PerfGuard: deploying ML-for-systems without performance regressions, almost! *Proceedings of the VLDB Endowment* 14, 13 (2021), 3362–3375.

[2] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. Autosteer: Learned query optimization for any sql database. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3515–3527.

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

[4] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer-Verlag, Berlin, Heidelberg.

[5] 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.

[6] 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.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[8] Alp Kucukelbir David M. Blei and Jon D. McAuliffe. 2017. Variational Inference: A Review for Statisticians. *J. Amer. Statist. Assoc.* 112, 518 (2017), 859–877. doi:10.1080/01621459.2017.1285773

[9] Bolin Ding, Rong Zhu, and Jingren Zhou. 2024. Learned Query Optimizers. *Foundations and Trends® in Databases* 13, 4 (2024), 250–310. doi:10.1561/1900000082

[10] Michael C. Fu. 2006. Chapter 19 Gradient Estimation. In *Simulation*, Shane G. Henderson and Barry L. Nelson (Eds.). Handbooks in Operations Research and Management Science, Vol. 13. Elsevier, 575–616. doi:10.1016/S0927-0507(06)13019-4

[11] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data(base) Engineering Bulletin* 18 (1995), 19–29. https://api.semanticscholar.org/CorpusID:260706023

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

[13] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-shot cost models for out-of-the-box learned cost prediction. *arXiv preprint arXiv:2201.00561* (2022).

[14] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. Deepdb: Learn from data, not from queries! *arXiv preprint arXiv:1909.00607* (2019).

[15] M.G. Kendall and A. Stuart. 1973. *The Advanced Theory of Statistics. Vol. 2: Inference and: Relationsship.* Griffin. https://books.google.ch/books?id=elabQwAACAAJ

[16] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[17] 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).

[18] Tom Lane. 2005. Re: Bitmap indexes etc. https://www.postgresql.org/message-id/12553.1135634231@sss.pgh.pa.us. [Online; accessed April, 2025].

[19] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. *Proceedings of the VLDB Endowment* 17, 7 (2024), 1565–1577. https://dl.acm.org/doi/10.14778/3654621.3654625

[20] 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 (11 2015), 204–215. Issue 3. doi:10.14778/2850583.2850594

[21] D.S. Lemons and P. Langevin. 2002. *An Introduction to Stochastic Processes in Physics.* Johns Hopkins University Press. https://books.google.ch/books?id=Uw6YDkd_CXcC

[22] Beibin Li, Yao Lu, Chi Wang, and Srikanth Kandula. 2021. Cardinality Estimation: Is Machine Learning a Silver Bullet?. In *AIDB*. https://www.microsoft.com/en-us/research/publication/cardinality-estimation-is-machine-learning-a-silver-bullet/

[23] Shengbo Li. 2023. *Reinforcement Learning for Sequential Decision and Optimal Control.* Springer. doi:10.1007/978-981-19-7784-8

[24] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. 2021. Fauce: fast and accurate deep ensembles with uncertainty for cardinality estimation. *Proceedings of the VLDB Endowment* 14, 11 (2021), 1950–1963.

[25] Ryan Marcus. 2023. Learned Query Superoptimization. *arXiv preprint arXiv:2303.15308* (2023).

[26] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making Learned Query Optimization Practical. *ACM SIGMOD Record* 51 (6 2022), 6–13. Issue 1. doi:10.1145/3542700.3542703

[27] 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 (4 2019), 1705–1718. Issue 11. doi:10.14778/3342263.3342644

[28] 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.* 1–4.

[29] Ryan Marcus and Olga Papaemmanouil. 2018. Towards a Hands-Free Query Optimizer through Deep Learning. (2018). arXiv:1809.10212 [cs.DB] https://arxiv.org/abs/1809.10212

[30] Rina Panigrahy. 2008. An improved algorithm finding nearest neighbor using Kd-trees. In *Proceedings of the 8th Latin American Conference on Theoretical Informatics* (Búzios, Brazil) *(LATIN'08)*. Springer-Verlag, Berlin, Heidelberg, 387–398.

[31] Silvan Reiner and Michael Grossniklaus. 2023. Sample-Efficient Cardinality Estimation Using Geometric Deep Learning. *Proceedings of the VLDB Endowment* 17, 4 (2023), 740–752.

[32] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. 2015. Learning structured output representation using deep conditional generative models. *Advances in neural information processing systems* 28 (2015).

[33] Student. 1908. The probable error of a mean. *Biometrika* (1908), 1–25.

[34] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
[35] Jerome Thiessat, Dirk Habich, and Wolfgang Lehner. 2024. Steering the PostgreSQL query optimizer using hinting: State-Of-The-Art and open challenges. (06 2024).
[36] Robert L. Thorndike. 1953. Who belongs in the family? *Psychometrika* 18 (1953), 267–276. https://api.semanticscholar.org/CorpusID:120467216
[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] Junxiong Wang, Kaiwen Wang, Yueying Li, Nathan Kallus, Immanuel Trummer, and Wen Sun. 2024. JoinGym: An Efficient Query Optimization Environment for Reinforcement Learning. https://openreview.net/forum?id=aAEBTnTGo3
[39] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
[40] Lianggui Weng, Rong Zhu, Di Wu, Bolin Ding, Bolong Zheng, and Jingren Zhou. 2024. Eraser: Eliminating Performance Regression on Learned Query Optimizer. *Proc. VLDB Endow.* 17, 5 (May 2024), 926–938. doi:10.14778/3641204.3641205
[41] Lucas Woltmann, Jerome Thiessat, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2023. FASTgres: Making Learned Query Optimizer Hinting Effective. *Proc. VLDB Endow.* 16, 11 (July 2023), 3310–3322. doi:10.14778/3611479.3611528
[42] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: a new cardinality estimation framework for join queries. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.
[43] Ziniu Wu, Amir Shaikhha, Rong Zhu, Kai Zeng, Yuxing Han, and Jingren Zhou. 2020. Bayescard: Revitilizing bayesian frameworks for cardinality estimation. *arXiv preprint arXiv:2012.14743* (2020).
[44] Xianghong Xu, Zhibing Zhao, Tieying Zhang, Rong Kang, Luming Sun, and Jianjun Chen. 2023. COOOL: A Learning-To-Rank Approach for SQL Hint Recommendations. arXiv:2304.04407 [cs.DB] https://arxiv.org/abs/2304.04407
[45] 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. doi:10.1145/3514221.3517885
[46] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. Neurocard: One cardinality estimator for all tables. *arXiv preprint arXiv:2006.08109* (2020).
[47] 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.
[48] 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.
[49] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. Queryformer: A tree transformer model for query plan representation. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1658–1670.
[50] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A learning-to-rank query optimizer. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1466–1479.
[51] Rong Zhu, Lianggui Weng, Wenqing Wei, Di Wu, Jiazhen Peng, Yifan Wang, Bolin Ding, Defu Lian, Bolong Zheng, and Jingren Zhou. 2024. PilotScope: Steering Databases with Machine Learning Drivers. *Proc. VLDB Endow.* 17, 5 (May 2024), 980–993. doi:10.14778/3641204.3641209
[52] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2020. FLAT: fast, lightweight and accurate method for cardinality estimation. *arXiv preprint arXiv:2011.09022* (2020).

Received January 2025; revised April 2025; accepted May 2025