Learned Offline Query Planning via Bayesian Optimization

JEFFREY TAO, University of Pennsylvania, USA
NATALIE MAUS, University of Pennsylvania, USA
HAYDN JONES, University of Pennsylvania, USA
YIMENG ZENG, University of Pennsylvania, USA
JACOB R. GARDNER, University of Pennsylvania, USA
RYAN MARCUS, University of Pennsylvania, USA

Analytics database workloads often contain queries that are executed repeatedly. Existing optimization techniques generally prioritize keeping optimization cost low, normally well below the time it takes to execute a single instance of a query. If a given query is going to be executed thousands of times, could it be worth investing significantly more optimization time? In contrast to traditional online query optimizers, we propose an offline query optimizer that searches a wide variety of plans and incorporates query execution as a primitive. Our offline query optimizer combines variational auto-encoders with Bayesian optimization to find optimized plans for a given query. We compare our technique to the optimal plans possible with PostgreSQL and recent RL-based systems over several datasets, and show that our technique finds faster query plans.

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

Additional Key Words and Phrases: Query optimization; Bayesian optimization

ACM Reference Format:
Jeffrey Tao, Natalie Maus, Haydn Jones, Yimeng Zeng, Jacob R. Gardner, and Ryan Marcus. 2025. Learned Offline Query Planning via Bayesian Optimization. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 179 (June 2025), 29 pages. https://doi.org/10.1145/3725316

1 Introduction

Query optimization is a long-standing problem in the database community [3, 46, 52]. Recent advancements in *learned* query optimization (LQO) [8, 9, 39, 58, 66, 70, 95, 98, 99, 104, 106] have shown significant promise, often delivering 2-10x improvements in query runtime. However, deploying LQO is complicated due to two main challenges: (1) query regressions ("my query was fast yesterday, why is it slow today?") and (2) the tight integration of machine learning components into the core query processing pipeline (which are generally engineered with different levels of reliability in mind).

Despite various efforts to address these challenges [56, 92], most real-world deployments of LQO (such as at Meta [1], Microsoft [67, 100], and Alibaba [91, 105]) have separated learned query optimization two components, an offline component and an online component. The *offline component* tests new query plans and caches those plans that perform better than the plans produced

Authors' Contact Information: Jeffrey Tao, University of Pennsylvania, USA, jefftao@seas.upenn.edu; Natalie Maus, University of Pennsylvania, USA, nmaus@seas.upenn.edu; Haydn Jones, University of Pennsylvania, USA, haydnj@seas.upenn.edu; Yimeng Zeng, University of Pennsylvania, USA, yimengz@seas.upenn.edu; Jacob R. Gardner, University of Pennsylvania, USA, jacobrg@seas.upenn.edu; Ryan Marcus, University of Pennsylvania, USA, rcmarcus@seas.upenn.edu.

ACM 2836-6573/2025/6-ART179
https://doi.org/10.1145/3725316

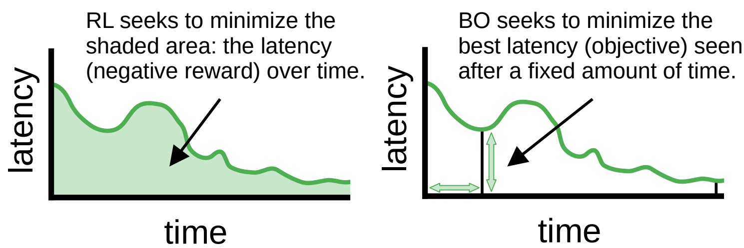


Fig. 1. A comparison of reinforcement learning (RL, left) and Bayesian optimization (BO, right). BO is a better match for the offline query optimization problem because we do not care about "regressions" in the offline search phase; we only care about the quality of the best-discovered plan.

by the traditional optimizer. The *online component* then checks this cache for a plan; if a cached plan is not found, it calls the traditional optimizer instead.

This compromise—spending additional resources offline to find good query plans for specific queries—solves issues with query regressions and avoids the need to put ML primitives into the query processing pipeline, but it is also motivated by the nature of analytic workloads. In many analytic systems, the majority of compute resources are spent executing repetitive report generation [55] or dashboarding queries [75], with some queries being executed hundreds of times per day [94] or hundreds of thousands of times per year [55]. Recent studies of Amazon Redshift showed that, for the median database, 60% of all queries executed were repeated queries (verbatim) [94] and that roughly 10% of all Redshift clusters have their entire workload consisting of queries that repeated within the last day [87].[1] Substantial repetition means that even a small improvement in query latency can be amplified many times over, making it worthwhile to invest additional optimization resources.

We call the goal of these offline components the **offline query optimization problem**: find the best query plan using as few offline resources (i.e., offline query time) as possible. Unlike traditional query optimizers, which generally seek to be so fast that optimization time amortizes to zero compared to execution time, an offline query optimizer is expected to take many times longer than a single query execution.

Learned query optimizers using the "offline/online" compromise are implicitly performing offline query optimization. Current systems must solve two fundamental problems: first, offline query optimizers must have a *search strategy* to decide which plans to test. Second, offline query optimizers must have a *timeout strategy* to deal with query plans that take too long to execute.

Search strategy Existing techniques either use a coarse-grained search of predefined alternative plans [1, 100], or adopt a fine-grained reinforcement learning procedure [91, 105]. The biggest drawback of the coarse-grained approach is that the set of plans explored is limited, and significant improvements may be outside of the search space. The drawback of the RL approach is more subtle.

Reinforcement learning (RL) is fundamentally the "wrong tool for the job" of offline query optimization. Specifically, the objective function of reinforcement learning is poorly aligned with offline query optimization. Consider the latency of the best plan found for a query over time as the query is optimized, as depicted on the left side of Figure 1. RL seeks to minimize the shaded area, representing latency (negative reward) over time. This corresponds to balancing exploration and exploitation [80]: the optimizer is penalized each time it chooses a bad plan, so the optimizer must frequently choose "lower risk" plans that lower the area under the curve, but might not be as informative as "higher risk" plans. This is perfectly aligned with the goals of *online* query

[1]Some unknown proportion of these verbatim repeats may involve staging tables or views, for which the underlying query may be changing.

optimization. However, in the context of *offline* query optimization, we actually care about the best query plan observed during the time-bounded optimization phase (that is, the minimum of the plotted function, not the area under the curve), as shown on the right side of Figure 1.

Trouble with timeouts An important, but often overlooked, dimension of learned query optimization is query timeouts: since some query plans are orders of magnitude worse than others [46], any learned query optimizer (either online or offline) has a non-zero chance of hitting a poor-performing plan. Current approaches solve this fundamental dilemma in ad-hoc ways, often by "timing out" (termination prior to completion) proposed query plans after a fixed threshold. These timed-out values are problematic because (1) they represented a large amount of wasted time (a poor query plan was selected [95]) and (2) no new information was gained (since updating an RL model with the timed-out value would cause the model to strictly underestimate the cost of the timed-out query [57]). Thus, a successful offline query optimizer must have a strategy for *selecting timeout values* and for *learning from timed-out queries*.

Bayesian optimization (BO) We propose an offline query optimizer, BayesQO, that closely mirrors recent work in computational drug discovery [64]: we use a learned encoder and decoder to translate query plans to and from vectors (called the *latent space*), such that similar query plans are mapped to nearby vectors. Then, off-the-shelf and well-studied Bayesian optimization techniques manipulate the encoded vectors in the latent space, using query execution as a reward signal. Existing coarse-grained techniques that generate a fixed set of plan variants for each query can be used to initialize the process, ensuring that the best-found plan is at least as good as the best plan in the initialization set.

We show that off-the-shelf BO techniques [76] can be easily adapted to the offline query optimization objective (that is, finding the fastest possible plan in the least amount of time). Furthermore, we show that the framework of Bayesian optimization enables robust *learning from timeouts* as well as *selecting timeouts* on a learned, plan-by-plan basis. In other words, we can meaningfully represent "query latency $> x$" within the learned model as a *censored observation*, and our model's confidence intervals provide a robust way of selecting timeouts to maximize information gain. Finally, we show how cross-query information can be incorporated into BayesQO by fine-tuning a language model to provide database-specific initialization points for the BO search process.

In our experiments, we show that offline optimization can yield 10-100x performance improvements over prior learned query optimization for some queries, and we demonstrate that our approach can find modest improvements for nearly every query in several benchmarks. Since our system targets repetitive analytic queries, even modest gains can be significantly amplified in practical settings. Our contributions include:

(1) We formalize the problem of offline query optimization,
(2) We implement BayesQO, an offline query optimizer that applies Bayesian optimization techniques,
(3) We show how recent developments in Bayesian optimization that accommodate high-dimensionality and censored observations can be integrated into BayesQO,
(4) We show how fine-tuning a language model can be used to incorporate cross-query information into BayesQO,
(5) We show that BayesQO can outperform online learned query optimization techniques in an offline setting.

2 Related work

Query optimization is a long-standing problem in the databases community. System R [3] proposed the heuristic query optimization scheme now used in most production databases [28], consisting of a cost model, cardinality estimates, and a dynamic programming search. Conventional query optimizers are designed according to the "query optimization contract" [7], which expects query optimizers to produce plans quickly (within hundreds of milliseconds, as the actual execution of the plan might be very quick); this work proposed that in order to improve query optimizers, we should consider "breaking" this contract in a number of ways, such as allowing the optimizer to intrusively examine the base data (as opposed to keeping cheap histograms), spend a long time on optimization, or even adaptively change the query plan during execution. We believe our work falls into this "breaking the contract" category. Older work [53] considered amortizing the cost of searching parts of the plan space across multiple executions, but did not consider offline execution. Query *reoptimization*, perhaps the first "contract breaker," is the task of proactively modifying or recreating a query plan during execution, based on information found during execution, with the overall goal of minimizing total latency [4, 51, 72].

A related concept from the compilers literature is "superoptimization" [61], in which a program compiler, which traditionally follows a similar "contract" as a query optimizer (i.e., fast compilation times), instead uses a large time budget to produce the best possible sequence of assembly instructions for a given program. Our work can be considered a sort of "superoptimization for query plans." GenesisDB [35] represents a similar effort, focusing on developing fast implementations over relational operators, instead of entire query plans (thus, GenesisDB is mostly orthogonal to the work presented here). Kepler [15] uses a genetic algorithm and exhaustive execution to map the plan space for parameterized queries, which can be viewed as a type of superoptimization. SlabCity [14] takes an approach similar to superoptimization by considering SQL-level semantic rewrites of queries to improve performance (e.g., query simplification). Finally, DataFarm [86] and HitTheGym [50] investigated how best to produce datasets for machine learning powered database components, including query optimizers.

In recent years, the databases community has been increasingly engaged in applying machine learning techniques to query optimization, including latency prediction [16, 17, 32, 59, 65, 94], cardinality estimation [26, 38, 41, 43, 49, 68, 71, 74, 96], and cost models [79]. Other works have attempted to either augment existing optimizers with learned components (e.g., [9, 10, 48, 56, 92, 98]) or entirely replace query optimizers with reinforcement learning (e.g., [5, 8, 39, 57, 85, 91, 95, 99, 102–105]). Most of these works are focused on the online optimization setting: they must complete quickly while avoiding performance regressions relative to traditional heuristic-based optimizers. Most of these works also employ reinforcement learning, seeking to manage regret from exploring alternatives instead of exploiting the current known-best plan. In comparison, we apply Bayesian optimization to the superoptimization problem because we are principally concerned with finding the query plan with the best possible latency, and ignore suboptimal plans. In the superoptimization setting, bad plans are only bad insofar as executing them until the timeout consumes part of the optimization time budget.

Bayesian optimization is not the only sample-efficient learning technique. For example, NeuroCARD [96] learns join distributions efficiently by uniformly sampling tuples from the full outer join of all tables in a schema. Reiner et al. [74] show how domain knowledge can be incorporated into learned models to improve sample efficiency via geometric deep learning. LlamaTune [40] uses database documentation to accelerate DBMS knob-tuning.

While our plan encoding was inspired by work in molecular dynamics (i.e, SELFIES [44] strings as used by Maus et al. [63]), a representation with similar goals for query plans was presented by Reiner et al. [74]. Our approaches mainly differ in what we are trying to represent: as our format only seeks to encode join orderings, it does not encode predicates. Furthermore, while Reiner et al. use invariances to give joins with the same cardinality the same representation, our encoding format may have multiple representations for the same join ordering. Further motivation for this design choice is given in Section 4.1. Other works have also looked at non-string representations of queries based on graphs [74], trees [57], and recurrences [81], typically by using neural network architectures which model these structures.

The random search heuristic we presented in Section 5 can be considered a modified version of QuickPick [89]. While we still sample random query plans, instead of using a cost model to evaluate their quality, we actually execute them. Such a suggestion would seem ludicrous in the original context of [89], but for offline optimization, executing terrible query plans is *not* off the table, if it eventually leads to a better plan!

Since Bayesian optimization is a relatively old technique, it may be reasonable to ask "why now?" Recent innovations in the machine learning community have made it practical to apply Bayesian optimization to *structured* (i.e. non-continuous) inputs with high dimensionality [21, 23, 64], which was previously impossible. The key innovation that enabled this advancement was attention transformer models [88], which allowed sequences to be efficiently and accurately mapped into vector spaces. In the databases literature, Bayesian optimization has been most frequently applied to tuning configuration knobs [6, 45, 101]. To our knowledge, this is the first work to apply BO directly to the optimization of individual queries.

Perhaps most similar to this work is LimeQO [97], a system that uses offline query execution to find the best query hint for each query in a workload. LimeQO can be viewed as a practical way of finding an optimal Bao [56] model for a given workload. LimeQO is arguably much simpler than the present work, requiring only linear methods (i.e., no VAE or Bayesian optimization). However, LimeQO only considers a finite set of query hints to apply to each query in a workload, whereas we fully construct query plans. As a result, the present work can potentially find better plans. Additionally, LimeQO focuses on optimizing an entire workload of queries at once (i.e., considering which queries are best to explore next), whereas we focus on optimizing only a single user-specified query.

3 System model & problem definition

We define the offline optimization problem for database query planning and show an approach to offline optimization based on Bayesian optimization. We implement this approach in BayesQO.

Challenges An obvious naïve strategy to perform offline optimization is to exhaustively enumerate the space of all possible query plans. This is computationally intractable even for relatively simple queries on few tables: the number of join orderings alone grows factorially with the number of joined tables: for a query joining n tables, considering only binary joins and ignoring physical operator selection, there are $n! \cdot C_n = \frac{2n!}{n!}$ distinct join orderings, where C_n is the n-th Catalan number [69].

A refinement of this strategy might be to consider query planning with "perfect cardinalities," since cardinality estimates are often hypothesized to be the main culprit for poor query plan performance [46]. Exhaustively computing cardinality estimates for even a simple query can take *months* [68], and adaptively measuring only the cardinality estimates used by the query planner leads to the infamous "fleeing from knowledge" problem, in which the optimizer repeatedly picks poor query plans due to underestimation from the independence assumption [60].

Furthermore, the plan space contains numerous bad plans which, on their own, are intolerable to execute to completion as they are many orders of magnitude slower than the optimal. Thus, an offline optimization method must efficiently explore the space of query plans while avoiding executing these bad plans to completion.

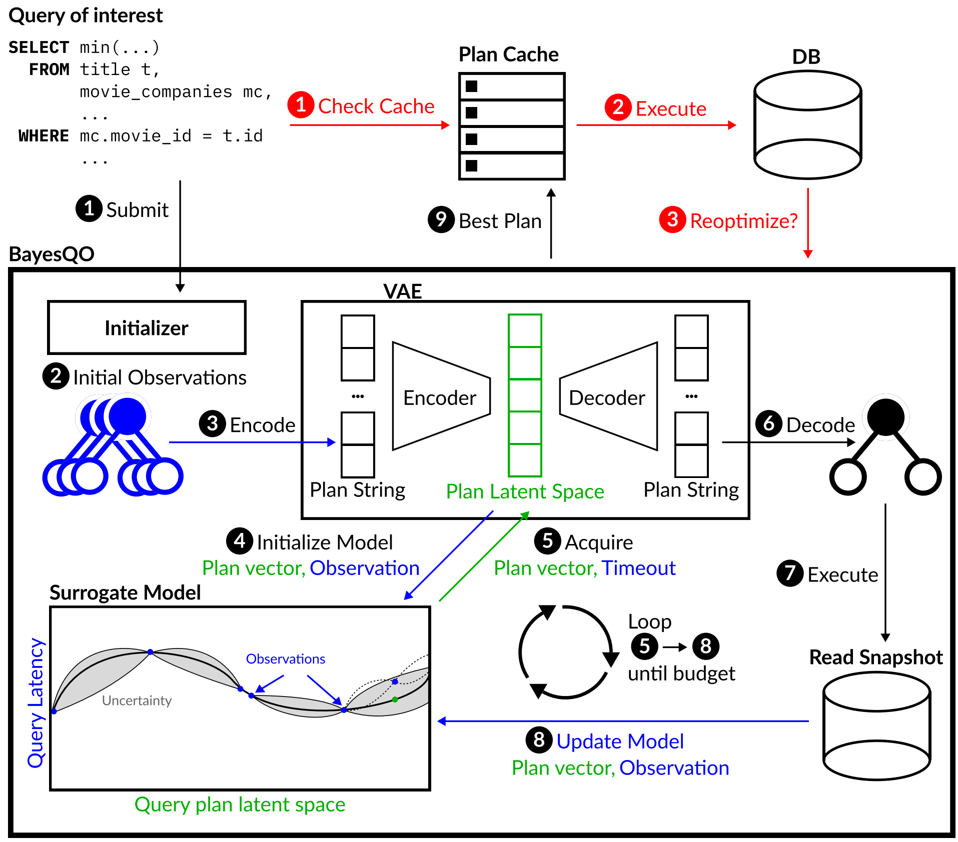


Fig. 2. BayesQO workflow

It is also crucial that the optimization process use information gained from executing candidate plans to inform its exploration. A necessary property of an offline optimization method is that it can be run for longer in order to obtain better results.

System model Figure 2 depicts the architecture of BayesQO. First, ❶ a user identifies a query Q that they wish to optimize offline. BayesQO will then begin searching for a fast plan for Q. To do so, BayesQO uses a *Bayesian optimization* loop. ❷ An initialization strategy is used to produce a set of plans (section 4.4), ❸ which are translated into strings and embedded into a vector space using a learned model (section 4.1). ❹ The embedded plans and their observed execution latencies are used to initialize the surrogate model. ❺ An acquisition function is used to select a point in the latent space (section 4.3), ❻ and then the latent space vector is decoded back to a query plan. ❼ This plan is then given a timeout value $TO(P)$, and executed against a read-only snapshot of the database. The query either executes successfully with latency $L(P) < TO(P)$, or times out. ❽ The Bayesian optimization algorithm uses the observed latency of the new query plan to improve its understanding of the query space. Then, the process repeats steps ❺ → ❽ until a time budget is exhausted or the user is satisfied with the achieved latency. ❾ Finally, the best seen plan goes into a cache.

When the query is being executed online, ❶ the user submits a query to the system. ❷ If the query was optimized offline, the cached plan is used. Otherwise, the DBMS' optimizer is used. ❸

Looking at the runtime statistics of the executed plan, BayesQO decides whether the query should be re-optimized.

Problem definition Our goal is to find a query plan P with low latency for a query Q using as few additional resources as possible.[2] Unlike traditional query optimizers, BayesQO will continuously test new query plans until terminated. We denote the sequence of queries produced by BayesQO at a given time t as $S_t = P_1, P_2, \ldots, P_n$. Let $\mathbb{I}_i$ be an indicator such that when $\mathbb{I}_i = 0$, the plan P_i completed successfully after $L(P_i)$, and when $\mathbb{I}_i = 1$, the plan P_i timed out after $TO(P_i)$. The cost a sequence S_t is given by:

$$Cost(S_t) = \sum_i \mathbb{I}_i \times TO(P_i) + (1 + -\mathbb{I}_i) \times L(P_i)$$

... and the best latency achieved within S_t is denoted as:

$$Latency(S_t) = \min_i \begin{cases} L(P_i) & \text{if } \mathbb{I}_i = 0 \\ \infty & \text{if } \mathbb{I}_i = 1 \end{cases}$$

Our goal is minimize latency while staying within a user-specified cost budget B:

$$\begin{aligned} \min_{S_t} \quad & Latency(S_t) \\ \text{subject to} \quad & Cost(S_t) < B \end{aligned}$$

Difference from prior query optimizers Traditional and previous learned query optimizers solve this problem for (very small) budgets B (e.g. for Neo [57], $B < 500ms$). Here, we consider B large enough to actually execute candidate query plans. As such, optimization time is assumed to be many times higher than the query latency.

Assumptions Our system model assumes the following about the DBMS and workload:

(1) There is a *default query optimizer* that produces reasonable but not globally optimal query plans for any given query.
(2) Queries can be executed against *read snapshots* of the database.
(3) The execution engine can *accept physical plans/hints* that specify join orders and physical join operators.
(4) Joins within queries are PK-FK equijoins.[3]

Why a read snapshot? BayesQO needs to execute many plans over the course of optimizing a query, some of which may be bad plans with very large intermediate results. So as not to disrupt the database's currently-running workload, we assume the ability to execute queries against read snapshots. Such read snapshots are common offerings from modern cloud providers (e.g., AWS Redshift [2]).

Example: a trivial offline optimizer The easiest way to implement an offline optimizer in our framework is with random plan search, similar to Quickpick [89] but ignoring cost estimates. Given a query Q, first measure the latency of the query plan P_d produced by the default query optimizer $L(P_d)$. Then, select a query plan P_1 for Q at random, and execute it with timeout equal to the latency of the best plan seen so far — initially, $L(P_d)$. Continue executing random plans up to the budget B. While the odds of finding a better plan than P_d are poor, note that you never exceed the

[2]Note that the constraint on time is required to make the problem non-trivial: if we have infinite computational resources, we can simply test every possible query plan and pick the fastest one.
[3]This is mostly a constraint of our current implementation; future work could straightforwardly extend this technique to support non-key or non-equijoin queries, since our core technique only needs to know which tables are involved in which joins.

budget B and your final plan is always at least as good as the default optimizer. We experimentally evaluate this simple baseline in Section 5.

Cross-query learning A hidden benefit to BayesQO is that each time a query is optimized, a large number of query plans are executed. These execution traces can be used as training data for cross-query models. BayesQO does this by using past execution traces to fine-tune an LLM to conditionally generate a query plan string (described in Section 4.1) for a given query. We show experimentally that these LLM-generated plans are oftentimes reasonable starting points for future optimizations. Thus, BayesQO creates a "virtuous cycle:" each time a query is optimized, additional training data is collected, which can be used to fine-tune an LLM. The LLM can then generate good initialization points for optimizing the next query, and so on. We describe our technique for cross-query learning in Section 4.4.

4 Bayesian Optimization for Query Plans

Bayesian optimization (BO) is a promising approach to efficiently explore the space of possible query plans while minimizing the cost of executing extra queries against the database. BO enables optimization of expensive-to-evaluate, black-box functions while requiring relatively few evaluations of the expensive function. However, BO techniques operate over continuous, real-valued domains, whereas query plans are discrete tree structures.

Prior work on Latent Space Bayesian Optimization (LSBO) has allowed Bayesian optimization to be applied over other discrete, combinatorial spaces by using a deep autoencoder model (DAE) to transform a discrete, structured search space $\mathcal{X}$ into a continuous, numerical one $\mathcal{Z}$ [12, 20, 27, 29, 37, 64, 84]. For example, Maus et al. [64] applied LSBO to problems in drug discovery using a DAE trained on string representations of molecules. Inspired by this work, we define a string encoding format for query plans (Section 4.1), train a variational autoencoder (VAE) on strings of this format (Section 4.2), and perform BO in the latent space of this VAE (Section 4.3), making novel contributions in the selection of timeouts (Section 4.3.1). Finally, we discuss different strategies for initializing the local BO process which impact BO performance (Section 4.4).

4.1 Query Plan String Format

Our first step towards optimizing query plans with Bayesian optimization is to represent query plans as strings. We design this language with particular properties that have been important in prior work to perform BO over other domains [64].

Desiderata for string representations Maus et al. [64] identify two essential properties that are key to success of a string language for LSBO. We translate those properties to query optimization, and adopt them as requirements for our query plan language:

(1) **Completeness:** Any valid query plan in X must be representable as a string using the language. If this is not true, high-quality plans that are not representable will not be discoverable by our optimization algorithm.
(2) **Decoding validity:** Any sequence of characters in the language must correspond to a valid query plan. If points in the latent space of the DAE decode to invalid query plans, optimization would need to be done under additional feasibility constraints, which adds unnecessary complexity to the optimization problem. Validity was a primary goal of the models in both Jin et al. [36] and Maus et al. [62].

The ideal string representation would also be *injective*, meaning that each unique string maps to a unique plan [74]. We were unable to find a string representation with all three of these properties, so we settle for a complete representation with decoding validity that is not injective, meaning that multiple strings may map to the same plan. Taking this tradeoff is motivated by prior work:

Maus et al. [63] showed that a string representation with only completeness and decoding validity (SELFIES [44]) outperformed an injective representation with completeness (SMILES [90]). We leave the investigation of alternative string representations to future work.

String language for query plans We design an encoding format for binary-tree-structured query plans that specifies join orders and operators (henceforth, a "join tree"). The join tree does not encode other aspects of the query such as selections, join and filter predicates, and aggregations. When the join tree is decoded to executable SQL, we translate the join tree to a hint string and prepend it to the original SQL text which contains these aspects of the query. We observe that a join tree can be unambiguously reconstructed if each non-leaf node's left and right children and its join operator are known; we simply need an unambiguous way to identify these subtrees.

In our encoding format, each join subtree is expressed as a 3 symbol sequence (left child, right child, operator). Each physical join operator (*h*ash, *m*erge, *n*ested loops) is given a unique symbol. The fully specified join is simply the concatenation of these sequences.

The leaves of a join tree are always the tables being joined, so we define a unique symbol for each base table in the schema. However, we cannot define symbols for each possible join subtree, as doing so would be tantamount to defining a unique symbol for every possible query plan. Instead, we observe that after a table symbol is used once to specify a join subtree, it will never be used again, as any table will only ever appear as a leaf once. Similarly, a particular non-leaf node only appears once, so to the right of a sequence specifying a particular join subtree, all symbols composing the join instead refer to the larger subtree.

The leftmost occurrence of a table symbol in a plan string always references the base table itself, but subsequent occurrences represent the largest subtree that the table is part of. For example, in a join between three tables A, B, and C, $(A \bowtie_{\text{hash}} (B \bowtie_{\text{merge}} C))$, the valid encoding strings are $(B, C, \bowtie_{\text{m}}, A, B, \bowtie_{\text{h}})$ and $(B, C, \bowtie_{\text{m}}, A, C, \bowtie_{\text{h}})$.

For multiple occurrences of the same table under different aliases within the same query, we rename such aliases to be numbered (e.g. `movies1`, `movies2`, ...), and we define a unique symbol in our language per table-number pair. This requires us to choose a maximum number of possible aliases of a single table. In our experiments we select the maximum occurrences of a particular table within the benchmark queries.

To fulfill our second requirement, decoding validity, we use a simple trick. We maintain state about the partially-specified join tree as we decode the string from left to right. If the decoder encounters a symbol that is syntactically invalid (e.g., a table in place of a join operator) or semantically invalid (e.g., a table that is not part of the join), the decoder deterministically resolves the symbol to a valid one by constructing a list of all valid symbols and using the invalid symbol's integer value as an index into the list.

While the choice of replacement symbol is arbitrary, this scheme for ensuring decoding validity is preferable to more obvious schemes such as refusing and resampling invalid strings or decoding them to some default plan. Rejecting strings would prevent the Bayesian surrogate model from learning about vast regions of the plan space. Decoding invalid strings to some default plan would make vast regions of the space undifferentiated in performance. Our technique ensures that all strings can be evaluated and that similar invalid strings are mapped to somewhat different query plans.

Limitations Our language does not represent subqueries and CTEs. When processing queries that contain such structures, they are left untouched, so the decoded query plan hint will not contain any reference to them or to tables only occurring within them.

4.2 Encoding & Decoding Query Plans

While BO can be applied to various search spaces, it is most straightforward in a continuous, real-valued domain. We train a deep autoencoder (DAE) model on these encoded strings encoded using the format defined in Section 4.1. This process generates a *latent space* – a continuous, real-valued domain that serves as a proxy for the discrete space of query plans, enabling application of BO techniques. Intuitively, our goal for the DAE is to construct a latent space in which similar query plans are mapped to similar vectors. This way, a search routine that finds a particularly good plan can look at the "neighbors" of that plan in the latent space for similar plans. The notions of "similar" and "neighbors" are both approximate: no actual neighborhoods or similarity scores are computed, but this property is *implicitly* created when training the DAE.

A DAE consists of an encoder $\Phi : \mathcal{X} \rightarrow \mathcal{Z}$ that maps from an input space $\mathcal{X}$ to a latent space $\mathcal{Z}$ (sometimes called a bottleneck [77], as it is often lower dimensional than $\mathcal{X}$ in order to force a degree of compression) and a decoder $\Gamma : \mathcal{Z} \rightarrow \mathcal{X}$ that maps from the latent space back to the input space. We use a type of DAE known as a variational autoencoder (VAE) [42], in which the encoder produces a distribution over latent points $\Phi(\mathcal{Z}|\mathcal{X})$, and the decoder produces a distribution over $\mathcal{X}$ given $\mathcal{Z}$, $\Gamma(\mathcal{X}|\mathcal{Z})$. The model is trained by maximizing the evidence lower bound (ELBO):

$$\mathbb{E}_{\Phi(\mathcal{Z}|\mathcal{X})}[\log \Gamma(\mathcal{X}|\mathcal{Z}) - \mathrm{KL}(\Phi(\mathcal{Z}|\mathcal{X})||p(\mathcal{Z})]$$

The learned decoder $\Gamma : Z \rightarrow X$ produces string-encoded query plans given points in the latent space. The VAE regularization (the KL term, representing relative entropy) makes the search space smooth, facilitating more effective optimization. These components allow us to perform BO: the surrogate model is defined over the latent space $\mathcal{Z}$, and we evaluate the black-box function f for points in the latent space by decoding the point to a string using the decoder Γ and executing them against the real database.

Training data In order to train the DAE, we compute a large set of encoded query plans (~1 million). Our goal is to have the DAE learn a smooth probability distribution of the space of query plans. Training strings for the DAE should be somewhat representative of the *family* of optimal query plans—the purpose of this is to create a space of plans in which points that are close to each other have similar performance characteristics. However, the space still contains points for *all possible* query plans.

To generate this set of plans, we sample random PK-FK equijoin queries from the "k-alias reference graph" of the database schema. This graph contain k nodes corresponding to each table and edges corresponding to all PK-FK references between tables. We choose k equal to the highest number of aliases of the same table used in any query in the workload. From this k-alias reference graph, we sample queries by selecting random connected subgraphs with varying numbers of vertices. Given a particular subgraph, we produce a query joining all table aliases (subgraph nodes) with equality join predicates for all present edges.

This process does not require any query execution, and can be done using only metadata from the DBMS. For each sampled query, we use the existing default query optimizer (e.g., PostgreSQL) to plan the query, encode the plan in our string encoding format, and add it to the VAE training set. In order to expand the diversity of plans used to train the VAE, we additionally produce encoded plans using hints [73] to the default query optimizer (e.g. disable nested loops, disable sequential scans).

Our training data generation process makes two key design choices: (1) sampling random queries from the database schema, and (2) generating query plans with the database's default optimizer. The first decision ensures that we have coverage for a wide variety of input queries. Our goal is to train the DAE once per schema, and then reuse the DAE for every query over the schema. The second decision ensures that the query plans we get are somewhat reasonable. For example, the underlying database optimizer is unlikely to pick a plan full of cross joins, and is likely to take advantage of index structures if applicable.

4.3 Background on Bayesian Optimization

Given a query plan language and a trained DAE to translate query plan strings into vectors in a latent space (and back), we can now optimize queries inside of the latent space using Bayesian Optimization (BO). Intuitively, BO in our application works by learning the relationship between the DAE's latent space and actual query plan latency. BO learns this relationship by repeatedly testing points sampled from the latent space. If the BO algorithm can get a good estimation of the relationship between the latent space and query latency, then good plans can be found. This section gives important background on the BO technique we use in this paper. Then, in Section 4.3.1, we explain some of the small changes we made to traditional algorithms to address query optimization specifically.

Bayesian optimization This section provides a brief overview of Bayesian optimization (BO). For readers unfamiliar with BO, we recommend the comprehensive book by Garnett [25]. Our methodology builds upon the approach developed by Eriksson et al. [22], with specific novel modifications tailored for optimizing query plans and execution latency in a DBMS.

Bayesian optimization is a method for optimizing black-box functions that are expensive to evaluate, aiming for *sample efficiency*. Given an input space $\mathcal{X}$ and an unknown objective function $f : \mathcal{X} \to \mathbb{R}$, BO seeks to find an input $\mathbf{x}^* \in \mathcal{X}$ that minimizes $f(\mathbf{x})$ in as few evaluations of f as possible. This is particularly useful when each evaluation of $f(\mathbf{x})$ is costly—for example, when $f(\mathbf{x})$ involves executing a query plan in a DBMS to measure its runtime.

BO operates by constructing a probabilistic surrogate model of the objective function, which is iteratively refined as new data is acquired. The general optimization procedure follows these steps:

(1) **Initialization**: Build a surrogate model of the objective f.
(2) **Acquisition Function Optimization**: Use an acquisition function to select the next point $\mathbf{x}_{\text{next}}$ to evaluate, balancing exploration and exploitation.
(3) **Evaluation**: Compute the true function value $f(\mathbf{x}_{\text{next}})$ by decoding and executing the query plan corresponding to $\mathbf{x}_{\text{next}}$.
(4) **Model Update**: Update the surrogate model with the new observation $(\mathbf{x}_{\text{next}}, f(\mathbf{x}_{\text{next}}))$, and repeat steps 2–4.

To efficiently navigate the search space, BO leverages the surrogate model along with the acquisition function to select promising candidate plans while minimizing the number of expensive evaluations. In BayesQO, we use *Thompson Sampling* [82] as the acquisition function.

Local BO Standard BO methods can struggle with high-dimensional or discrete optimization problems, such as those encountered in query plan optimization, due to the curse of dimensionality and the combinatorial explosion of the search space. To address this, we incorporate methods from the local BO literature, specifically *TuRBO* [22]. TuRBO maintains a hyper-rectangular "trust region" within the input space, which constrains the region from which points are sampled. By dynamically adjusting the size and location of these trust region based on the optimization success / failure, TuRBO can balance global exploration with local exploitation, allowing for efficient optimization in high-dimensional spaces.[4]

[4]Though called "local BO", this is a *global* optimization process that can produce results significantly different from the initialization points. Local BO methods are the most competitive methods in high-dimensional spaces, as established in Eriksson et al. [22].

Right-censored observations During typical Bayesian optimization, when we make an observation at a point $\mathbf{x}$, we obtain an associated objective value $f(\mathbf{x}) = y$. For a right-censored observation, when we observe at the point $\mathbf{x}$, we instead only learn that y was greater than some threshold τ. In our application, right-censored observations represent query timeouts: If a query $\mathbf{x}$ is observed to execute for τ seconds before timing out, then we know that the true latency of q is *at least* τ: $f(\mathbf{x}) \geq \tau$.

In the query optimization setting, using right-censored observations is particularly important. Obtaining true values for arbitrary plans in the space of possible plans can be infeasible, as bad plans may take days or even weeks. Thus, it is more efficient if we can *time out* plans that perform poorly and update the surrogate model with knowledge that the running time of $\mathbf{x}$ is "at least as bad as $\mathbf{y}$". Intuitively, for the purposes of finding optimal plans, it is not necessary to know exactly how bad a particular plan is—it suffices to know that plans like it should be avoided.

BO in the presence of censored observations was first explored by Hutter et al. [34], where an EM-like algorithm was used to impute the value of censored responses. They applied this method to algorithm configuration, terminating any runs that exceeded a constant factor of the shortest running time observed so far. Building on this, Eggensperger et al. [18] trained a neural network surrogate on a likelihood based on the Tobit model to directly model right-censored observations:

$$p(\mathbf{y}|\mathbf{f}) = \phi(\mathbf{z})^{1-\mathbf{I}}(1 - \Phi(\mathbf{z}))^{\mathbf{I}}$$

$$\mathbf{z} = \frac{\mathbf{f} - \mu}{\sigma^2}$$

$$I = \begin{cases} 0, & \text{if } \mathbf{y} \text{ is uncensored} \\ 1, & \text{if } \mathbf{y} \text{ is censored} \end{cases}$$

where ϕ and Φ denote the Gaussian density and cumulative density function respectively. In Eggensperger et al. [18], timeout thresholds were chosen as a fixed percentile of existing observations.

Approximate Gaussian processes Because the space of query plans is large, we anticipate needing to test a large number of query plans. As a result, we must select a surrogate model that (1) allows for *probabilistic inference*, that is, gives a probability distribution at each point instead of a simple point estimate, and (2) can scale to a large number of observations. Thus, we select an *approximate* Gaussian Process (GP) model.

Approximate GP models, such as the popular Scalable Variational Gaussian Process (SVGP) [31], use inducing point methods in combination with variational inference to allow approximate GP inference on large data sets [30, 83]. The standard evidence lower bound (ELBO) on the log-likelihood used to train a SVGP model is the following:

$$\log p(\mathbf{y}) \geq \mathbb{E}_{q(\mathbf{f})}[\log p(\mathbf{y} \mid \mathbf{f})] - \mathrm{KL}(q(\mathbf{u}) \,||\, p(\mathbf{u})) \tag{1}$$

4.3.1 Bayesian Optimization with Censored Observations. While previous work on Bayesian optimization with censored observations (censored BO) did not use approximate SVGP [31] models, we contribute a straightforward extension of SVGP [31] models to the censored BO setting. Starting from Equation (1) and using the Tobit likelihood given in Section 4.3, we derive the new ELBO:

$$\begin{aligned}
&\log p(\mathbf{y}) \geq \mathbb{E}_{q(\mathbf{f})}[\log p(\mathbf{y} \mid \mathbf{f})] - \mathrm{KL}(q(\mathbf{u}) \,||\, p(\mathbf{u})) \\
&= \mathbb{E}_{q(\mathbf{f})}[\log \phi(\mathbf{Z})^{1-\mathbf{I}}(1 - \Phi(\mathbf{Z}))^{\mathbf{I}}] - \mathrm{KL}(q(\mathbf{u}) \,||\, p(\mathbf{u})) \\
&= \mathbb{E}_{q(\mathbf{f})}[\log \phi(\mathbf{Z})^{1-\mathbf{I}} + \log(1 - \Phi(\mathbf{Z}))^{\mathbf{I}}] - \mathrm{KL}(q(\mathbf{u}) \,||\, p(\mathbf{u})) \\
&= \mathbb{E}_{q(\mathbf{f})}[\log \phi(\mathbf{Z_u})] + \mathbb{E}_{q(\mathbf{f})}[\log(1 - \Phi(\mathbf{Z_c}))] - \mathrm{KL}(q(\mathbf{u}) \,||\, p(\mathbf{u}))
\end{aligned}$$

Here, $\mathbf{Z}_u$ correspond to $\frac{\mathbf{f}-\mu}{\sigma^2}$ values for uncensored observations, and $\mathbf{Z}_c$ correspond to censored observations. The first term can be computed analytically as in standard SVGP models. The second term, $\mathbb{E}_{q(\mathbf{f})}[\log(1-\Phi(\mathbf{Z_c}))]$, can be computed using one dimensional numerical techniques like Gauss-Hermite quadrature.

During optimization, we select a threshold τ for each executed query plan $\mathbf{x}$, and cut off execution once the running time exceeds τ. This results in right-censored observations. Selecting the timeout for any given observation is crucial: selecting too low of a timeout deprives BO of important knowledge about the space of plans, whereas selecting too high of a timeout wastes time executing bad plans. Previous work in BO uses a constant multiplier over the best observation seen so far [33], or a fixed percentile across all observations [19]. Balsa [95] also uses a fixed multiplier in order to bound the impact of executing bad plans. We use an uncertainty-based method for selecting timeouts that, compared to prior work, ensures that the surrogate model will be sufficiently confident that a particular point is suboptimal before timing out.

Before evaluating a new candidate query plan $\mathbf{x}_t$ during step t of optimization, we dynamically set a new timeout threshold τ_t. We select thresholds so that, after conditioning on the right-censored observation $(\mathbf{x}_t, \tau_t)$, we are *confident* that the best query plan observed so far, $\mathbf{x}_t^*$, is still a better design than the candidate plan $\mathbf{x}_t$. Because we do not want to waste additional running time evaluating $f(\mathbf{x}_t)$, we ideally want the *smallest* such τ_t.

Selecting τ_t. The above discussion leads to the following optimization problem of finding the smallest threshold τ so that our incumbent is confidently better than $\mathbf{x}_t$ *after conditioning on* $(\mathbf{x}_t, \tau)$:

$$\begin{aligned} \tau_t^* &= \arg\min \tau \\ \text{s.t. } & y_t^* \leq \mu_t'(\tau) - \kappa\sigma_t'(\tau) \end{aligned}$$

On its surface, this optimization problem is challenging, as evaluating our constraint for a given τ involves updating the Gaussian process surrogate model with that value τ as the observed timeout. This is similar to other acquisition functions in the Bayesian optimization literature that use fantasization to do lookahead, e.g., knowledge gradient [24].

Because we use variational GPs, there are several inexpensive strategies that we can use to evaluate the constraint. For example, Maddox et al. [54] recently proposed an efficient routine for online updating sparse variational GPs, both with conjugate and non-conjugate likelihoods. Alternatively, a few additional iterations of SGD can be used to update the model in a less sophisticated way.

Finally, we note that the value of $\mu_t'(\tau) - \kappa\sigma_t'(\tau)$ should generally be monotonic in τ—fantasizing that $\mathbf{x}_t$ is cut off with a larger threshold should strictly increase the gap between our belief about y_t and y_t^*. Therefore, given a routine to cheaply evaluate the constraint, the constrained minimization problem over τ can, for example, be solved with binary search.

4.4 Initialization Strategies

The initial step of BO typically involves selecting points within the latent space using the acquisition function. As the surrogate is initialized with a random prior, this amounts to selecting random points within the latent space. Theoretically, given sufficient time for BO execution, this approach would yield optimal results. However, to improve the practicality of BO within high dimensional spaces, it is helpful to initialize the process with a small number of precomputed $(\mathbf{x}, f(\mathbf{x}))$ pairs representing high-quality plans. We explore multiple methods of generating these initialization points.

Hinted plans (Bao) We can leverage an existing traditional query optimizer that accepts hints, such as PostgreSQL, to generate the initialization points. We exhaust all of the combinations of join and scan hints (as in the hint sets used by Marcus et al.'s Bao [56] optimizer) to produce 49 initialization points for each query. These 49 initialization points are *guaranteed* to contain the best plan that could have possibly been chosen by Bao.

The default optimizer plan A simpler strategy would be to generate a single optimization point by using the DBMS' underlying optimizer. This approach has the advantage of simplicity, since the underlying DBMS almost surely has an optimizer. Unfortunately, we found that this approach does not work well in practice, mostly because initializing BO with a single initialization point seems to be suboptimal [64].

LLM Inspired by previous work demonstrating the effectiveness of large language models (LLMs) in optimizing program runtimes [78], we explore the use of fine-tuned LLMs for generating initialization points. We collected trajectories from 606 BayesQO runs, selecting the top-1 and top-5 query plans for each query to construct a fine-tuning dataset. Using this dataset, we fine-tuned GPT4o-mini for one epoch. For each new query, we use the fine-tuned model to sample 50 initialization points. This approach leverages the model's ability to learn patterns from previous optimization runs, potentially producing high-quality plans that outperform those generated by traditional query optimizers. Our evaluation (section 5.6) demonstrates that this LLM-based strategy can often produce the best query plan among all initialization strategies considered here.

Extensibility BayesQO simply admits sets of initialization pairs $(x, f(x))$, so any strategy can be used to generate these pairs. As such, our approach can incorporate future improvements in traditional or learned query optimization techniques.

4.5 Random plans

Though not related to BO, we implement random plan search, which can be thought of as a completely exploration-based algorithm. The intuition behind this method is that joins are commutative but that cross-joins are generally bad for performance. This strategy samples random plans from the space of all plans that do not contain any cross joins.

Given a particular query over a set of table aliases, we construct the subgraph of the schema's alias-k reference graph containing only the table aliases referenced in the query. From this query graph, we can construct a random join tree by constructing a spanning tree. Whenever an edge is added to the spanning tree, we add the join between the two newly connected components to the join tree. Physical join operators are selected uniformly randomly.

One potential benefit of utilizing this strategy is that its viability implies that it may be possible to perform offline optimization in the absence of a traditional query optimizer. As we show in Section 5, this strategy can be used on its own to perform offline optimization.

5 Experiments

We sought to answer the following research questions about BayesQO:

RQ1. How effectively does BayesQO reduce query latency for a workload given a certain time budget? (Section 5.2, Section 5.3)

RQ2. How important are our modifications to Bayesian optimization to the performance of BayesQO? (Section 5.4)

RQ3. How robust are plans generated by BayesQO to data drift? Can previously optimized plans help jump-start reoptimization? (Section 5.5)

RQ4. Can we train an LLM from offline optimization results to generalize to unseen queries? (Section 5.6)

5.1 Setup

We evaluate BayesQO over four sets of queries:

Name	Size on Disk	Queries	Median joins per query
JOB	8GB	113	7
CEB	8GB	234	10
Stack	64GB	200	6
DSB	89GB	90	5

Table 1. Characteristics for the four evaluation workloads.

(1) **JOB**: The entire Join Order Benchmark (introduced by Leis et al. [47]), which consists of 113 queries over the IMDB dataset.
(2) **CEB**: A subset of the Cardinality Estimation Benchmark (introduced by Negi et al. [68]), which consists of $\sim$ 3000 queries across 16 query templates over the IMDB dataset. We select the top 100 and bottom 100 queries by improvement of the optimal hint set vs. the PostgreSQL default plan, as well as the 100 queries with longest-running PostgreSQL default plans. There is some overlap between these categories, resulting in 234 total queries representing 13 templates.
(3) **Stack**: A subset of the StackOverflow benchmark (introduced by Marcus et al. in "Bao" [56]), which consists of $\sim$ 6000 queries divided across 16 query templates. We selected the longest-running queries from each template in equal proportion (excluding templates consisting entirely of queries that took less than 1 second), producing a list of 200 queries.
(4) **DSB**: 3 generated queries from each of 30 templates, based on TPC-DS but enhanced with more complex data distributions and query templates (introduced by Ding et al. [13]). We use generated queries from the "agg" and "spj" template sets. Following Wu et al. [93], we use a scale factor of 50.

Workload characteristics are summarized in Table 1.

The offline query planning setting has not received much attention in the query optimization literature. To our knowledge, this work is one of the first to demonstrate an offline optimization technique. As such, we compare plans generated by BayesQO to plans from PostgreSQL, Bao [56], Balsa [95], and the random non-cross-join plan generation technique described in section 4.5, which we refer to as Random. The Random strategy can be seen as representing the gains from performing offline optimization at all, and we use it to understand whether our technique brings further improvement when rethinking the query optimization contract [7].

Instead of actually running Bao, we instead execute all hint sets (49 total, comprised of all combinations of join and scan hints) and take the hint set with the fastest runtime. This is the best plan that Bao could ever produce, since it focuses on steering PostgreSQL's existing optimizer using hints. We choose this baseline as representing the best that traditional heuristic-based query optimizers can do in the offline query optimization setting. Unless stated otherwise, all BO runs in the following section are initialized using these 49 hinted "Bao" plans and runtimes.

We choose Balsa [95] as a baseline representing reinforcement learning-based systems, which are also not originally designed for the offline query optimization setting. We use the default configuration for Balsa, except that we set $S = 1.5$ (the multiplier on query timeout values), which we found to universally improve results in our experimental setup. We note that the comparison to Balsa is not entirely fair: as a reinforcement learning-powered query optimizer, Balsa seeks to minimize *regret*, whereas our BO-based approach seeks to minimize the *best latency found*. This distinction is illustrated in Figure 1, but the most important experimental consequence is that Balsa will occasionally repeat a query plan that it considers to be "good" in order to maximize reward, which is obviously suboptimal in an offline optimization scenario. In these experimental results, we use a plan cache to avoid actually executing any exactly-duplicated query plans.

Latent Dimension	Reconstruction Accuracy
128	97.93%
64	89.67%
32	58.71%
16	24.79%
8	8.49%

Table 2. VAE reconstruction accuracy on the validation set at different latent dimensions, higher is better.

Random can be thought of as an offline optimization technique that purely explores the space of possible plans. It learns from feedback only insofar as it decreases the time spent on bad plans by settings its timeout to the runtime of the best plan seen (as, unlike with BO, there is no point in executing plans worse than the best-seen). The initial timeout greatly affects how many plans can be tried within a given time budget, as a lower timeout results in more plans tried within a particular time period, but we do not know a priori what the runtime of the fastest possible plan is. We initialize the Random plan generation process with the PostgreSQL default plan runtime as the initial timeout.

All queries are executed against PostgreSQL version 16.3. For all workloads, we disable JIT and set `join_collapse_limit` to 1. Physical operator hints are specified using `pg_hint_plan`. We configure PostgreSQL with 32GB shared buffers and 16MB work memory. For the Stack queries, we disabled GEQO as it was causing high variability in plan performance.

We create indexes on all join keys on the IMDB, Stack, and DSB datasets. For Stack, some tables have compound join keys. We build all indexes such that for each set of join predicates between two tables present in all queries in the workload, an index exists containing all of the referenced columns on each side.

Our VAE is based on the transformer VAE architecture introduced in [64]. For each database, the set of training query plans were divided into an 80%/20% train-test split, over which the VAE was trained for 800, 000 steps. To determine an appropriate latent space size, VAEs were trained with varying latent dimensionality on the IMDB dataset to evaluate the trade-off between compression and reconstruction, results of which are shown in Table 2. A latent space of 64 dimensions was chosen as it represented a good balance between latent dimension and reconstruction accuracy. Each VAE was trained on 2 Ampere A6000s for ~ 24 hours. On the GPU cloud that we were using, this cost roughly $24 for each VAE.

5.2 Plan Optimization

In order to answer **RQ1.**, we executed each of our baseline optimization techniques for several hours for each query in each of our three workloads. Figure 3 visualizes the results at the end of optimization for BO, Balsa, and Random across our three workloads by comparing the cumulative distribution of queries that achieve at least a certain percentage improvement in plan runtime compared to Bao. "% improvement over Bao" refers to the percentage reduction in plan runtime for a certain query compared to the runtime of the optimal Bao plan (e.g. a reduction in runtime from 1 second to 200 milliseconds would be an improvement of 80%). The visual separation between the series for the different techniques trending towards the top right of each plot indicates gaps in performance in which one technique is finding plans for more queries with better performance than another technique.

We strove to make comparisons between techniques fair by giving each optimization technique the same optimization budget, since offline optimization techniques can hypothetically be executed for an indefinite amount of time to find potentially faster plans. We only considered time spent

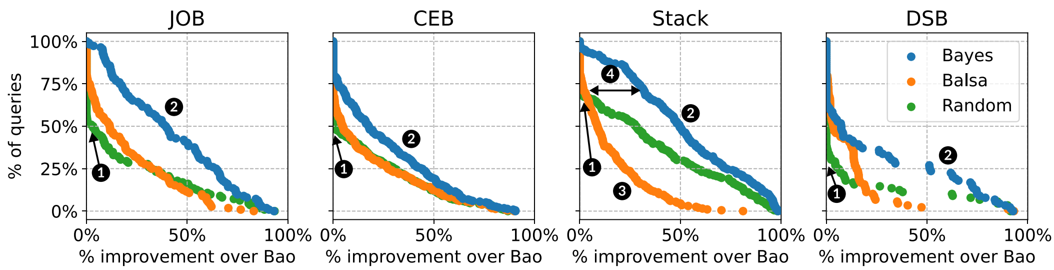
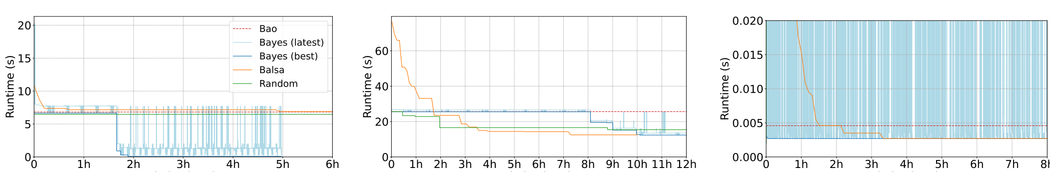


Fig. 3. Best plans found at the end of optimization with each technique for each workload. Towards the top right corner is better. ❶ On each workload, Random fails to find any improvement over the best Bao plan for 30-50% of queries. ❷ BayesQO always finds the most improvement compared to the other methods, with this difference being more pronounced on JOB and Stack than on CEB. On Stack, BayesQO finds over 2× improvement on 50% of queries. ❸ Balsa underperforms when used as an offline optimizer on Stack due to longer query runtimes limiting exploration by the RL algorithm. ❹ BayesQO finds improvement on the ~25% of Stack queries where the other two techniques find none.



(a) STACK_Q2-025: BO finds a better plan than any of the other techniques

(b) CEB_11A102: All techniques find good plans, though BO takes notably longer.

(c) JOB_1B: All techniques converge to the same (very small) runtime.

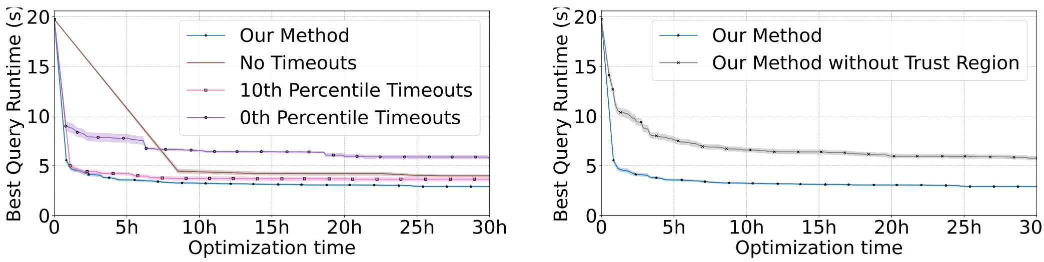
Fig. 4. Case studies showing optimization time vs. plan performance for individual queries. Lower is better.

executing proposed plans against the database as consuming budget and exclude the overhead of executing each technique (the overhead for BayesQO is analyzed in Section 5.7). Each optimization technique was executed for 4000 plan executions. We choose this because query runtimes span from tens of milliseconds to tens of seconds, and we did not want to optimize some queries with 1000× more observations than others.

These plots make obvious the fact that BayesQO is finding more plans with better performance compared to the baselines across all three workloads. JOB and Stack are highly differentiated, while all techniques perform similarly on CEB. DSB is notable in its proportion of queries for which no technique finds much improvement over Bao, but for those queries where improvement can be found, BayesQO is moderately differentiated from the baselines.

5.3 Case Studies

We select three queries from the optimization workloads to illustrate the different optimization outcomes for BayesQO. The optimization runs for these three queries are visualized in Figure 4. All three plots visualize the runtime of the best plan found so far over the course of an optimization run across all of the optimization techniques. Bao is visualized as a horizontal line showing the runtime of the best plan because it cannot improve after its hint sets are executed. The light blue "Bayes (latest)" line illustrates the runtime of the plan run most-recently by BO, hence its constant fluctuation as BO explores the space of possible plans. The x axis in each of these plots captures



(a) Optimization time vs. best plan runtime for different timeout schemes when running BO.

(b) Optimization time vs. best plan runtime with and without the local "Trust Region" optimization.

Fig. 5. Ablation study of our novel BO scheme.

cumulative execution time on the database and ignores time spent in the rest of the optimization algorithms such as plan proposal and model updates.

In Figure 4a, we highlight a case where the advantages of BO are most obvious. In the first hour, BO is exploring the space immediately around its initialization points, executing plans for slightly longer than the best Bao initialization due to uncertainty-based timeouts. Around 1.5 hours into the optimization run, it finds a plan within a trust region that has substantially better performance and rapidly exploits this new information by trying nearby plans. Note that in most of the queries made afterwards, timeouts are lowered to be closer to the new optimal as the BO surrogate model no longer needs to know if plans are much worse than the new optimal. We also note that neither Random nor Balsa manages to find a plan better than the Bao optimal.

In Figure 4b, all optimization strategies converge to the same best plan runtime after several hours of execution. BO takes notably longer than Random and Balsa to find this plan. That Random finds an optimized plan so quickly suggests that the space of possible plans contains many good plans that perform approximately this well, but they may be quite different from the initialization points given to BO. We also note that despite the fact that we give Balsa training examples including the Bao optimal plan, it begins its search with plans that are considerably worse before eventually passing the Bao optimal.

Figure 4c shows a query for which all techniques converge relatively quickly to the same plan runtime and do not make any progress afterwards. We note that this optimized plan executes in 2ms, which is short enough to be indistinguishable from noise in our experimental setup. We observe that Balsa takes the longest time to arrive at this plan whereas the other techniques find it nearly instantly, which we take as further evidence of the unsuitability of RL-based algorithms for offline optimization. The plateaus in Balsa's progress occur when it is exploiting its best known plan in order to minimize regret instead of trying to find a faster plan.

5.4 Timeout Ablation Study

To answer **RQ2.**, we justify our usage of a novel timeout strategy, as well as the choice to perform local BO based on trust regions, [5] via an ablation study visualized in Figure 5. As described in Section 4.3, we utilize timeouts in order to manage the impact of executing terrible plans that take many orders of magnitude longer to execute than the optimal plan, wasting optimization budget for little gain.

In Figure 5a, we show the results of using different timeout schemes when optimizing a single JOB query. Intuitively, using timeouts longer than the runtime of the current best-seen plan (i.e. the 0th percentile timeout) allows the surrogate model to learn more about regions of the space

[5] Which, despite the name, is a *global* optimization scheme: see Section 4.3.

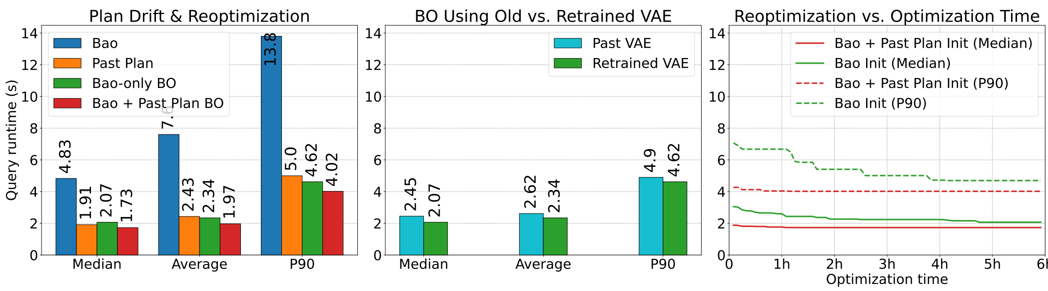


Fig. 6. Left: Plans from the past vs. plans optimized in the future. Middle: BO run results when using the VAE from the past vs. the VAE retrained in the future. Right: Optimization speed of BO initialized with Bao vs. those including the past plan.

of plans that it is less certain about. Since we use censored observations, two plans (and their surrounding regions of plan space) will look exactly the same if they both timed out after 1 second, even if one plan would have executed for 1.2 seconds and the other for 2 hours. By using longer timeouts, the surrogate model gains greater confidence in whether a particular region of the space is still promising to explore or if it is clearly terrible. As shown in the plot, our uncertainty-based method for determining the timeout threshold results in BO finding faster plans while consuming less optimization budget. In fact, using the best-seen runtime as the timeout causes BO to find the worst plan by the end of optimization, perhaps due to the artificially low timeout uniformly discouraging BO from exploring the space of plans.

We also justify the choice to use trust region-based local BO instead of global BO by performing another ablation, shown in Figure 5b. Our choice to represent the space of plans via the latent space of a VAE comes at the cost of high dimensionality (64). Though it is possible in principle that local BO will miss globally optimal plans, in our experiments we find that local BO initialized with plans derived from the PostgreSQL optimizer can find highly-optimized plans for many queries. In the ablation plots, we can see that even after many hours of optimization, global BO does not catch up to the quality of plans found by local BO due to the exponentially larger space of plans that it must explore.

5.5 Data Drift After Optimization

In order to model data drift, we modified the StackOverflow dataset by deleting all rows in all tables with timestamps after 2017, as well as the transitive closure of all rows whose foreign keys became invalidated as a result of deleting those rows. This reduced the overall dataset size by roughly 20%, with individual tables decreasing in row count between 0% and 28%. This deletion effectively restored the database to a snapshot from the end of 2017, while the original StackOverflow dataset snapshot was taken in late 2019. We present this two year shift as a worst-case scenario for data drift, expecting that if plan performance were to degrade due to data drift, it would degrade more over a longer period of time. For the rest of this section, we will refer to this 2017 snapshot as the "past" and the original 2019 snapshot as the "future".

We sought to answer three questions about the impact of data drift:

RQ3.1. How do the past plans perform in comparison to plans produced by reoptimizing from scratch on the future dataset?
RQ3.2. Is it important to retrain the VAE for the future dataset before performing BO?
RQ3.3. Does including the past plan as an initialization point in the future BO process speed up optimization?

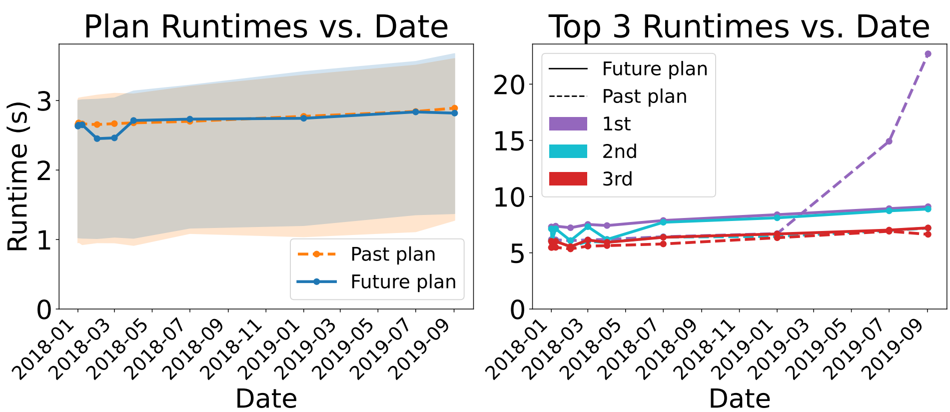


Fig. 7. Left: Performance of plans optimized in the past vs. plans optimized in the future executed on dates in between. Shaded regions show the 25th to 75th percentile runtimes. Right: The top 3 longest plan runtimes per date.

To answer **RQ3.1.**, we trained a VAE for the past dataset using the procedure described in Section 4.2, planning the sampled queries using PostgreSQL with the past dataset, akin to conducting the full BayesQO offline optimization process in the past. We then performed BO against the past dataset using this past version of the VAE and a reduced set of 50 (out of our original 200) queries. We then took the past query plans and executed them against the future dataset, effectively simulating the real-world use case of continuing to use previously optimized query plans well past when data drift may have rendered those plans suboptimal.

The results of executing these past plans against the future dataset are visualized in the left plot of Figure 6. We compare past plans against the Bao optimal plans for the future dataset and the future plans produced by BO from our initial experiment. We also compare them to the results of performing BO for a short time (about 1 hour) initialized with the past plan in addition to the Bao plans, discussed further below. Despite the substantial data drift, past plans perform about as well as if we had performed BO for the first time on the future dataset and continue to perform much better than the best Bao hint. This suggests that the optimality of most of the plans found by BO is not affected much data drift.

We also executed these past and future plans on dates in between the "past" and "future" endpoints, shown in Figure 7. For the vast majority of queries, there is not a significant difference in runtime between the past and future plans, but as shown by the dramatic increase in runtime of the longest-running past plan, it is indeed possible for data drift to render a plan suboptimal.

To answer **RQ3.2.**, we performed a set of BO runs against the future dataset using the VAE trained on the past dataset, simulating the real-world use case of attempting to perform offline optimization in the future without retraining the VAE. The results are visualized in the middle plot of Figure 6. We observe that keeping the VAE up-to-date has a non-negligible effect on the optimality of plans found by BO. Given the small cost of training the VAE compared to the rest of the BO process, it seems worthwhile to periodically retrain the VAE to account for data drift.

To answer **RQ3.3.**, we performed a set of BO runs against the future dataset initialized with both the Bao initialization and the optimized past plan, simulating reoptimization when a query had previously been optimized in the past. For these BO runs, we used the VAE trained on the future dataset. The aggregate results in red for reoptimization in the left plot of Figure 6 show not only that reoptimization to account for data drift is viable, but that it tends to produce the best plans across the entire workload. In the right side of the same figure, we observe that the BO runs

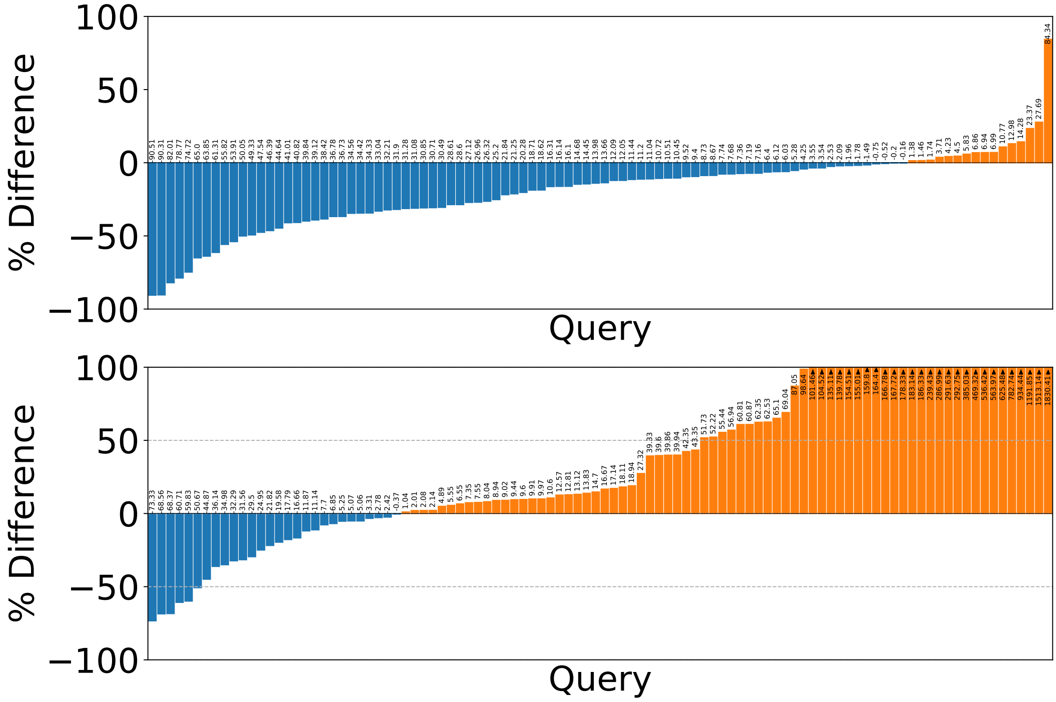


Fig. 8. Top: LLM trained on plans of the same template. Bottom: LLM trained on plans not of the same template. Lower (more blue) is better.

converge to optimized plans much more quickly than a run started from scratch. Reoptimizations ran for an average of 1.5 hours compared to from-scratch optimization runs, which ran for an average of 8.2 hours. This suggests that plans generated by BayesQO can be brought up-to-date to account for data drift while consuming much less optimization budget than the initial offline optimization run.

5.6 Few-Shot LLM from BO Results

As a byproduct of performing this experimental evaluation, we generated plans that, to our knowledge, are the best plans that can currently be found for the queries in our evaluation workloads. We hypothesized that a large language model fine-tuned using these high-quality plans could potentially be a better few-shot plan optimizer than existing techniques. Here, we evaluate the LLM's ability to generate initialization points for BayesQO and defer offline optimization initialized using the LLM outputs to future work.

In order to answer **RQ4.**, we performed two experiments. In the first experiment, visualized in the top plot of Figure 8, we fine-tuned GPT-4o mini on the fastest 10 optimized plans for each query from running BO on the CEB workload as described in Section 4.4. This training set included queries from all query templates present in the workload. We then compared the best runtime out of 50 plans (giving it as many plans as Bao) per-query generated by the LLM for a particular query against the runtime of the optimal Bao plan.

In the second experiment, we performed the same process, but withheld BO results from two query templates from the fine-tuning process. We then performed the same test, comparing the best runtime out of 50 plans per-query from the LLM against the optimal Bao plans. The results are

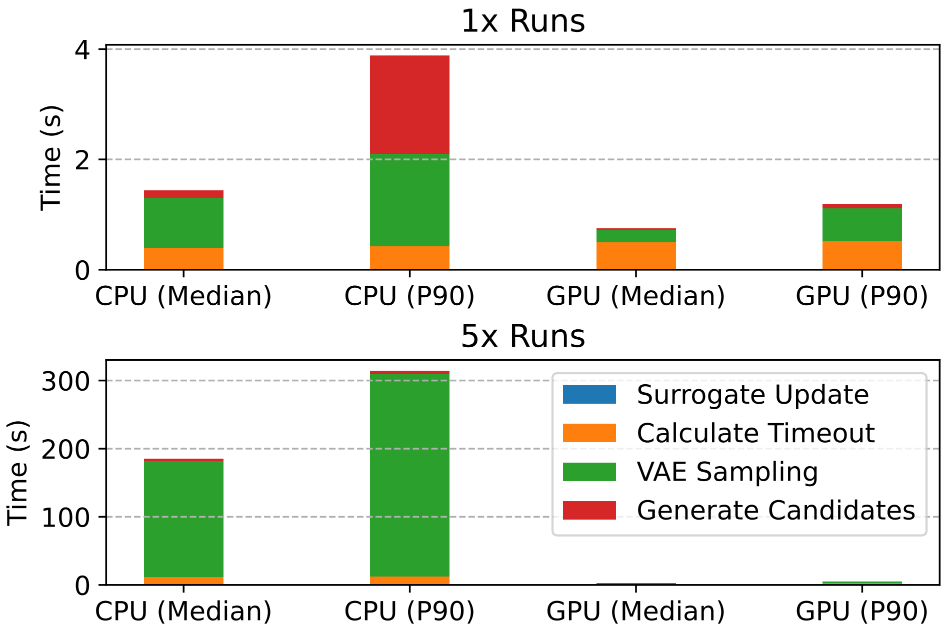


Fig. 9. Overheads per iteration of BO. Top: 1x simultaneous BO run. Bottom: 5x simultaneous BO runs.

visualized in the bottom plot of Figure 8. The LLM clearly does not perform as well when it has not been fine-tuned with results from the same query template.

5.7 BayesQO Overhead

We measured the overhead of running BayesQO under multiple conditions in order to better understand its resource requirements and to determine if a GPU is necessary to perform optimization. We recorded time spent in each part of the optimization loop when executing the BO components on both CPU and GPU while varying the number of simultaneous BO runs. The results are visualized in Figure 9. We find that with only one BO run, while overhead on a CPU is worse than on a GPU, the absolute time spent on overhead (i.e., everything but query execution) is in the single-digit seconds range. For sufficiently long-running queries, this CPU overhead may be tolerable. However, even with just 5 simultaneous runs, we note that VAE sampling scales significantly better on GPUs than CPUs – this is attributable to the GPU's hardware support for scaled dot product attention [11].

5.8 Comparison to LimeQO

LimeQO [97] is another work that performs offline optimization for repetitive workloads, but LimeQO chiefly differs from BayesQO in that its potential optimizations are limited to finding optimal hints from a small set, whereas BayesQO constructs entire join orders. LimeQO uses the hint sets from Bao [56], which BayesQO also uses to initiate the Bayesian optimization process. As shown in Figure 10, both techniques explore all of the Bao hints: once LimeQO has exhausted all of the hints, there are no remaining avenues for further optimization, whereas BayesQO continues exploring and finds better plans.

6 Conclusion

In this work, we presented BayesQO, an *offline* learned query optimizer. BayesQO uses modern Bayesian optimization techniques to search for fast query plans for important repeated queries.

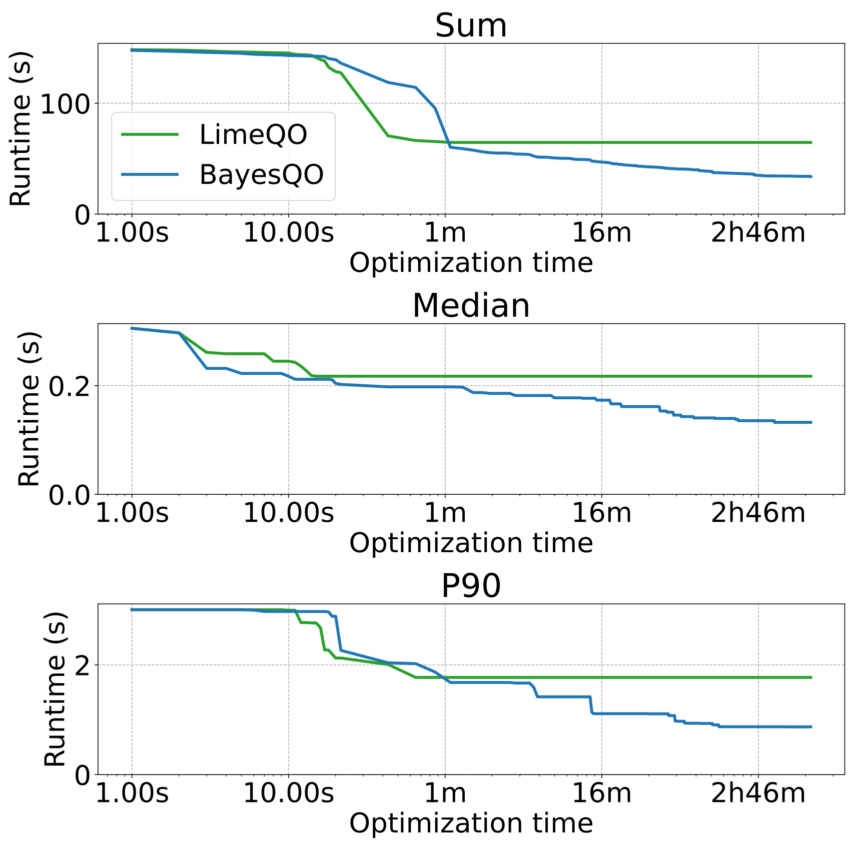


Fig. 10. Optimization using BayesQO vs. LimeQO across the entire JOB. Lower is better. The x-axis is log scale so both final performance and the initial improvement is visible.

Experimentally, we show that BayesQO was able to optimize nearly every query in a set of common, non-synthetic benchmarks, sometimes achieving multiple orders-of-magnitude improvements.

In the future, we plan to integrate BayesQO into a wider variety of database systems. Given the promising results from our LLM experiment, we plan to investigate how we can "close the loop," using the LLM to generate initialization data for future optimization runs. We are also excited to examine how Bayesian optimization can be applied to other databases problems such as automatic index selection, data layout, and benchmark curation.

Acknowledgments

N. Maus was supported by the National Science Foundation Graduate Research Fellowship; J. R. Gardner was supported by NSF awards IIS-2145644 and DBI-2400135.

References

[1] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithvi Pandian, Nikolay Leptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *PVLDB* 14, 1 (Aug. 2023). https://doi.org/10.14778/3611540.3611544

[2] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2205–2217. https://doi.org/10.1145/3514221.3526045

[3] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. 1976. System R: relational approach to database management. *ACM Trans. Database Syst.* 1, 2 (June 1976), 97–137. https://doi.org/10.1145/320455.320457

[4] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization with Rio. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 936–938. https://doi.org/10.1145/1066157.1066294

[5] Henriette Behr, Volker Markl, and Zoi Kaoudi. 2023. Learn What Really Matters: A Learning-to-Rank Approach for ML-based Query Optimization. In *Database Systems for Business, Technology, and the Web 2023 (BTW '23)*, Birgitta König-Ries, Stefanie Scherzinger, Wolfgang Lehner, and Gottfried Vossen (Eds.). Gesellschaft für Informatik e.V. https://doi.org/10.18420/BTW2023-25

[6] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. 2021. CGPTuner: a contextual gaussian process bandit approach for the automatic tuning of IT configurations under varying workload conditions. *Proc. VLDB Endow.* 14, 8 (April 2021), 1401–1413. https://doi.org/10.14778/3457390.3457404

[7] Surajit Chaudhuri. 2009. Query optimizers: time to rethink the contract?. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 961–968. https://doi.org/10.1145/1559845.1559955

[8] 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 (March 2023), 1777–1789. https://doi.org/10.14778/3587136.3587150

[9] 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 (May 2023), 2261–2273. https://doi.org/10.14778/3598581.3598597

[10] Guilherme Damasio, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Alex Mihaylov, Jaroslaw Szlichta, and Calisto Zuzarte. 2019. Guided automated learning for query workload re-optimization. *Proceedings of the VLDB Endowment* 12, 12 (Aug. 2019), 2010–2021. https://doi.org/10.14778/3352063.3352120

[11] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG] https://arxiv.org/abs/2205.14135

[12] Aryan Deshwal and Janardhan Rao Doppa. 2021. Combining Latent Space and Structured Kernels for Bayesian Optimization over Combinatorial Spaces. *CoRR* abs/2111.01186 (2021). arXiv:2111.01186 https://arxiv.org/abs/2111.01186

[13] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: a decision support benchmark for workload-driven and traditional database systems. *Proc. VLDB Endow.* 14, 13 (Sept. 2021), 3376–3388. https://doi.org/10.14778/3484224.3484234

[14] Rui Dong, Jie Liu, Yuxuan Zhu, Cong Yan, Barzan Mozafari, and Xinyu Wang. 2023. SlabCity: Whole-Query Optimization Using Program Synthesis. *Proceedings of the VLDB Endowment* 16, 11 (July 2023), 3151–3164. https://doi.org/10.14778/3611479.3611515

[15] Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altinbuken, Eugene Brevdo, and Campbell Fraser. 2023. Kepler: Robust Learning for Faster Parametric Query Optimization. *Proceedings of the 2023 ACM SIGMOD Conference* 1, 1 (May 2023), 109. https://doi.org/10.1145/3588963

[16] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance Prediction for Concurrent Database Workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. ACM, Athens, Greece, 337–348. https://doi.org/10.1145/1989323.1989359 tex.acmid= 1989359 tex.numpages= 12.

[17] Jennie Duggan, Olga Papaemmanouil, Ugur Cetintemel, and Eli Upfal. 2014. Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT '14)*. 109–120.

[18] Katharina Eggensperger, Kai Haase, Philipp Müller, Marius Lindauer, and Frank Hutter. 2020. Neural Model-based Optimization with Right-Censored Observations. arXiv:2009.13828 [cs.AI] https://arxiv.org/abs/2009.13828

[19] Katharina Eggensperger, Kai Haase, Philipp Müller, Marius Lindauer, and Frank Hutter. 2020. Neural Model-based Optimization with Right-Censored Observations. arXiv:2009.13828 [cs.AI] https://arxiv.org/abs/2009.13828

[20] Stephan Eissman, Daniel Levy, Rui Shu, Stefan Bartzsch, and Stefano Ermon. 2018. Bayesian optimization and attribute adjustment. In *Proc. 34th Conference on Uncertainty in Artificial Intelligence*.

[21] David Eriksson and Martin Jankowiak. 2021. High-dimensional Bayesian optimization with sparse axis-aligned subspaces. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*. PMLR, 493–503. https://proceedings.mlr.press/v161/eriksson21a.html ISSN: 2640-3498.

[22] David Eriksson, Michael Pearce, Jacob Gardner, Ryan D Turner, and Matthias Poloczek. 2019. Scalable Global Optimization via Local Bayesian Optimization. In *Advances in Neural Information Processing Systems*. 5496–5507. http://papers.nips.cc/paper/8788-scalable-global-optimization-via-local-bayesian-optimization.pdf

[23] David Eriksson, Michael Pearce, Jacob R Gardner, Ryan Turner, and Matthias Poloczek. 2019. Scalable global optimization via local Bayesian optimization. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS '19)*. Curran Associates Inc., Red Hook, NY, USA, 5496–5507.

[24] Peter Frazier, Warren Powell, and Savas Dayanik. 2009. The knowledge-gradient policy for correlated normal beliefs. *INFORMS journal on Computing* 21, 4 (2009), 599–613.

[25] Roman Garnett. 2023. *Bayesian Optimization*. Cambridge University Press.

[26] Damjan Gjurovski, Angjela Davitkova, and Sebastian Michel. 2024. Grid-AR: A Grid-based Booster for Learned Cardinality Estimation and Range Joins. https://doi.org/10.48550/arXiv.2410.07895 arXiv:2410.07895 [cs].

[27] Rafael Gómez-Bombarelli, Jennifer N Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. 2018. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science* 4, 2 (2018), 268–276.

[28] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[29] Antoine Grosnit, Rasul Tutunov, Alexandre Max Maraval, Ryan-Rhys Griffiths, Alexander Imani Cowen-Rivers, Lin Yang, Lin Zhu, Wenlong Lyu, Zhitang Chen, Jun Wang, Jan Peters, and Haitham Bou-Ammar. 2021. High-Dimensional Bayesian Optimisation with Variational Autoencoders and Deep Metric Learning. *CoRR* abs/2106.03609 (2021). arXiv:2106.03609

[30] James Hensman, Nicolò Fusi, and Neil D. Lawrence. 2013. Gaussian processes for Big data. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence* (Bellevue, WA) *(UAI'13)*. AUAI Press, Arlington, Virginia, USA, 282–290.

[31] James Hensman, Alex Matthews, and Zoubin Ghahramani. 2014. Scalable Variational Gaussian Process Classification. arXiv:1411.2005 [stat.ML] https://arxiv.org/abs/1411.2005

[32] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-shot cost models for out-of-the-box learned cost prediction. *Proceedings of the VLDB Endowment* 15, 11 (July 2022), 2361–2374. https://doi.org/10.14778/3551793.3551799

[33] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. 2013. Bayesian Optimization With Censored Response Data. arXiv:1310.1947 [cs.AI] https://arxiv.org/abs/1310.1947

[34] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Bayesian Optimization With Censored Response Data. *CoRR* abs/1310.1947 (2013). arXiv:1310.1947 http://arxiv.org/abs/1310.1947

[35] Immanuel Trummer. 2022. *GenesisDB: Synthesizing Customized SQL Execution Engines from Natural Language Instructions Using GPT-3 Codex.* Technical Report. Cornell, Ithaca. NY. https://rm.cab/genesisdb

[36] Wengong Jin, Regina Barzilay, and Tommi S. Jaakkola. 2018. Junction Tree Variational Autoencoder for Molecular Graph Generation. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 2328–2337. http://proceedings.mlr.press/v80/jin18a.html

[37] Hiroshi Kajino. 2019. Molecular hypergraph grammar with its application to molecular optimization. In *International Conference on Machine Learning*. PMLR, 3183–3191.

[38] Amin Kamali, Verena Kantere, Calisto Zuzarte, and Vincent Corvinelli. 2024. RobOpt: A Tool for Robust Workload Optimization Based on Uncertainty-Aware Machine Learning. In *Companion of the 2024 International Conference on Management of Data (SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 468–471. https://doi.org/10.1145/3626246.3654755

[39] Amin Kamali, Verena Kantere, Calisto Zuzarte, and Vincent Corvinelli. 2024. Roq: Robust Query Optimization Based on a Risk-aware Learned Cost Model. (2024). https://doi.org/10.48550/ARXIV.2401.15210

[40] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: sample-efficient DBMS configuration tuning. *Proc. VLDB Endow.* 15, 11 (July 2022), 2953–2965. https://doi.org/10.14778/3551793.3551844

[41] Kyoungmin Kim, Sangoh Lee, Injung Kim, and Wook-Shin Han. 2024. ASM: Harmonizing Autoregressive Model, Sampling, and Multi-dimensional Statistics Merging for Cardinality Estimation. *Proceedings of the ACM on Management of Data* 2, 1 (March 2024), 1–27. https://doi.org/10.1145/3639300

[42] Diederik P. Kingma and Max Welling. 2014. Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1312.6114

[43] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research (CIDR '19)*. http://arxiv.org/abs/1809.00677

[44] Mario Krenn, Florian Häse, AkshatKumar Nigam, Pascal Friederich, and Alán Aspuru-Guzik. 2020. Self-Referencing Embedded Strings (SELFIES): A 100% robust molecular string representation. https://doi.org/10.48550/arXiv.1905.13741 arXiv:1905.13741.

[45] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *Proc. VLDB Endow.* 17, 8 (May 2024), 1939–1952. https://doi.org/10.14778/3659437.3659449

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

[47] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[48] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. openGauss: an autonomous database system. *Proceedings of the VLDB Endowment* 14, 12 (July 2021), 3028–3042. https://doi.org/10.14778/3476311.3476380

[49] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. 2023. ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads. *Proc. VLDB Endow.* 17, 2 (Oct. 2023), 197–210. https://doi.org/10.14778/3626292.3626302

[50] Wan Shen Lim, Lin Ma, William Zhang, Matthew Butrovich, Samuel Arch, and Andrew Pavlo. 2024. Hit the Gym: Accelerating Query Execution to Efficiently Bootstrap Behavior Models for Self-Driving Database Management Systems. *Proceedings of the VLDB Endowment* 17, 11 (July 2024), 3680–3693. https://doi.org/10.14778/3681954.3682030

[51] Mengmeng Liu, Zachary G. Ives, and Boon Thau Loo. 2016. Enabling Incremental Query Re-Optimization. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1705–1720. https://doi.org/10.1145/2882903.2915212

[52] Guy Lohman. 2014. Is Query Optimization a "Solved" Problem?. In *ACM SIGMOD Blog (ACM Blog '14)*. https://wp.sigmod.org/?p=1075

[53] Hongjun Lu, K. Tan, and S. Dao. 1995. The Fittest Survives: An Adaptive Approach to Query Optimization *(VLDB '95)*. Zurich, Switzerland. https://www.semanticscholar.org/paper/The-Fittest-Survives%3A-An-Adaptive-Approach-to-Query-Lu-Tan/3d2625f15445adc9dd23324d4839c5dd364630fa

[54] Wesley J Maddox, Samuel Stanton, and Andrew G Wilson. 2021. Conditioning sparse variational gaussian processes for online decision-making. *Advances in Neural Information Processing Systems* 34 (2021), 6365–6379.

[55] Ryan Marcus. 2023. Learned Query Superoptimization. In *Joint Workshops at 49th International Conference on Very Large Data Bases (AIDB@VLDB '23)*. CEUR Workshop Proceedings, Vancouver, BC, Canada.

[56] 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 (SIGMOD '21)*. China. https://doi.org/10.1145/3448016.3452838 Award: 'best paper award'.

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

[58] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM @ SIGMOD '18)*. Houston, TX.

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

[60] V. Markl, P. J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T. M. Tran. 2007. Consistent selectivity estimation via maximum entropy. *The VLDB Journal* 16, 1 (Jan. 2007), 55–76. https://doi.org/10.1007/s00778-006-0030-1

[61] Alexia Massalin. 1987. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News* 15, 5 (Oct. 1987), 122–126. https://doi.org/10.1145/36177.36194

[62] Natalie Maus, Haydn Jones, Juston Moore, Matt J. Kusner, John Bradshaw, and Jacob R. Gardner. 2022. Local Latent Space Bayesian Optimization over Structured Inputs. In *Advances in Neural Information Processing Systems 35:*

Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/ded98d28f82342a39f371c013dfb3058-Abstract-Conference.html

[63] Natalie Maus, Haydn Thomas Jones, Juston Moore, Matt Kusner, John Bradshaw, and Jacob R. Gardner. 2022. Local Latent Space Bayesian Optimization over Structured Inputs *(NeurIPS '22)*. https://openreview.net/forum?id=nZRTRevUO-

[64] Natalie T. Maus, Haydn T. Jones, Juston S. Moore, Matt J. Kusner, John Bradshaw, and Jacob R. Gardner. 2024. Local latent space Bayesian optimization over structured inputs. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 2500, 14 pages.

[65] Mert Akdere and Ugur Cetintemel. 2012. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering (ICDE '12)*. IEEE, 390–401.

[66] Songsong Mo, Yile Chen, Hao Wang, Gao Cong, and Zhifeng Bao. 2023. Lemo: A Cache-Enhanced Learned Optimizer for Concurrent Queries. *Proc. ACM Manag. Data* 1, 4 (Dec. 2023), 247:1–247:26. https://doi.org/10.1145/3626734

[67] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc Friedman, and Alekh Jindal. 2021. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. ACM, Virtual Event China, 2557–2569. https://doi.org/10.1145/3448016.3457568 Award: 'best paper honorable mention'.

[68] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-loss: Learning cardinality estimates that matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[69] Kiyoshi Ono and Guy M. Lohman. 1990. Measuring the Complexity of Join Enumeration in Query Optimization. In *VLDB (VLDB '90)*. 314–325. http://dl.acm.org/citation.cfm?id=645916.671976

[70] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *2nd Workshop on Data Managmeent for End-to-End Machine Learning (DEEM '18)*. https://arxiv.org/abs/1803.08604

[71] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2018. QuickSel: Quick Selectivity Learning with Mixture Models. *arXiv:1812.10568 [cs]* (Dec. 2018). http://arxiv.org/abs/1812.10568

[72] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. 2019. How I Learned to Stop Worrying and Love Re-optimization. *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (April 2019), 1758–1761. https://doi.org/10.1109/ICDE.2019.00191 Conference Name: 2019 IEEE 35th International Conference on Data Engineering (ICDE) ISBN: 9781538674741 Place: Macao, Macao Publisher: IEEE.

[73] PostgreSQL Developers. 2024. PostgreSQL hints, https://www.postgresql.org/docs/current/runtime-config-query.html. https://www.postgresql.org/docs/current/runtime-config-query.html tex.key= 1.

[74] Silvan Reiner and Michael Grossniklaus. 2024. Sample-Efficient Cardinality Estimation Using Geometric Deep Learning. *Proc. VLDB Endow.* 17, 4 (March 2024), 740–752. https://doi.org/10.14778/3636218.3636229

[75] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. 2024. Predicate Caching: Query-Driven Secondary Indexing for Cloud Data Warehouses. In *Companion of the 2024 International Conference on Management of Data (SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 347–359. https://doi.org/10.1145/3626246.3653395

[76] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. 2016. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proc. IEEE* 104, 1 (Jan. 2016), 148–175. https://doi.org/10.1109/JPROC.2015.2494218

[77] Ohad Shamir, Sivan Sabato, and Naftali Tishby. 2010. Learning and generalization with the information bottleneck. *Theoretical Computer Science* 411, 29 (June 2010), 2696–2711. https://doi.org/10.1016/j.tcs.2010.04.006

[78] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. Learning Performance-Improving Code Edits. In *The Twelfth International Conference on Learning Representations (ICLR)*. https://openreview.net/pdf?id=ix7rLVHXyY

[79] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment* 13, 3 (Nov. 2019), 307–319. https://doi.org/10.14778/3368289.3368296

[80] Richard S. Sutton and Andrew G. Barto. 1998. *Introduction to Reinforcement Learning* (1st ed.). MIT Press, Cambridge, MA, USA.

[81] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *arXiv:1503.00075 [cs]* (Feb. 2015). http://arxiv.org/abs/1503.00075

[82] William R. Thompson. 1933. On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika* (1933).

[83] Michalis K. Titsias. 2009. Variational Learning of Inducing Variables in Sparse Gaussian Processes. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics, AISTATS 2009, Clearwater Beach, Florida, USA, April 16-18, 2009 (JMLR Proceedings, Vol. 5)*, David A. Van Dyk and Max Welling (Eds.). JMLR.org, 567–574. http://proceedings.mlr.press/v5/titsias09a.html

[84] Austin Tripp, Erik A. Daxberger, and José Miguel Hernández-Lobato. 2020. Sample-Efficient Optimization in the Latent Space of Deep Generative Models via Weighted Retraining. In *Advances in Neural Information Processing Systems 33*.

[85] Immanuel Trummer, Samuel Moseley, Deepak Maram, Saehan Jo, and Joseph Antonakakis. 2018. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *PVLDB* 11, 12 (2018), 2074–2077. https://doi.org/10.14778/3229863.3236263

[86] Robin Van De Water, Francesco Ventura, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2022. Farming Your ML-based Query Optimizer's Food. In *2022 IEEE 38th International Conference on Data Engineering (ICDE) (ICDE '22)*. 3186–3189. https://doi.org/10.1109/ICDE53745.2022.00294 ISSN: 2375-026X.

[87] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Eknath Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. *Proceedings of the VLDB Endowment* (2024). https://www.amazon.science/publications/why-tpc-is-not-enough-an-analysis-of-the-amazon-redshift-fleet

[88] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems (NeurIPS '17, Vol. 30)*. Curran Associates, Inc. https://papers.nips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[89] Florian Waas and Arjan Pellenkoft. 2000. Join Order Selection (Good Enough Is Easy). In *Advances in Databases (BNCD '00)*. Springer, Berlin, Heidelberg, 51–67. https://doi.org/10.1007/3-540-45033-5_5

[90] David Weininger. 1988. SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences* 28, 1 (1988), 31–36. https://doi.org/10.1021/ci00057a005 arXiv:https://doi.org/10.1021/ci00057a005

[91] Lianggui Weng, Rong Zhu, Di Wu, Bolin Ding, Bolong Zheng, and Jingren Zhou. 2024. Eraser: Eliminating Performance Regression on Learned Query Optimizer. *PVLDB* 17, 5 (2024), 926–938. https://doi.org/10.14778/3641204.3641205

[92] Lucas Woltmann, Jerome Thiessat, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2023. FASTgres: Making Learned Query Optimizer Hinting Effective. *Proceedings of the VLDB Endowment* 16, 11 (Aug. 2023), 3310–3322. https://doi.org/10.14778/3611479.3611528

[93] Peizhi Wu and Zachary G. Ives. 2024. Modeling Shifting Workloads for Learned Database Systems. *Proceedings of the ACM on Management of Data* 2, 1 (March 2024), 1–27. https://doi.org/10.1145/3639293

[94] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *Proceedings of the 2024 International Conference on Management of Data (SIGMOD '24) (SIGMOD '24)*. Santiago, Chile. https://doi.org/10.48550/arXiv.2403.02286

[95] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 931–944. https://doi.org/10.1145/3514221.3517885

[96] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *arXiv:2006.08109 [cs]* (June 2020). http://arxiv.org/abs/2006.08109 arXiv:2006.08109.

[97] Zixuan Yi, Yao Tian, Zachary G. Ives, and Ryan Marcus. 2025. Low Rank Learning for Offline Query Optimization. *Proceedings of the ACM on Management of Data* 3, 3 (June 2025). https://doi.org/10.1145/3725412

[98] 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 (Sept. 2022), 3924–3936. https://doi.org/10.14778/3565838.3565846

[99] 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 '20)*. 1297–1308. https://doi.org/10.1109/ICDE48307.2020.00116 ISSN: 2375-026X.

[100] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. 2022. Deploying a Steered Query Optimizer in Production at Microsoft. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. ACM, Philadelphia PA USA, 2299–2311. https://doi.org/10.1145/3514221.3526052

[101] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating database tuning with hyper-parameter optimization: a comprehensive experimental evaluation. *Proc. VLDB Endow.* 15, 9 (May 2022),

1808–1821. https://doi.org/10.14778/3538598.3538604

[102] 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 (April 2022), 1658–1670. https://doi.org/10.14778/3529337.3529349

[103] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A learned query rewrite system using Monte Carlo tree search. *Proceedings of the VLDB Endowment* 15, 1 (Sept. 2021), 46–58. https://doi.org/10.14778/3485450.3485456

[104] 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 (Feb. 2023), 1466–1479. https://doi.org/10.14778/3583140.3583160

[105] 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. *PVLDB* 17, 5 (2024), 980–993. https://doi.org/10.14778/3641204.3641209

[106] Sergey Zinchenko and Sergey Iazov. 2024. HERO: Hint-Based Efficient and Reliable Query Optimizer. https://doi.org/10.48550/arXiv.2412.02372 arXiv:2412.02372 [cs].

Received October 2024; revised January 2025; accepted February 2025