Generation of Adversarial Queries via Bayesian Optimization

Jeffrey Tao*
University of Pennsylvania
jefftao@seas.upenn.edu

Yimeng Zeng*
University of Pennsylvania
yimengz@seas.upenn.edu

Natalie Maus
Massachusetts Institute of Technology
nmaus@mit.edu

Haydn Jones
University of Pennsylvania
haydnj@seas.upenn.edu

Osbert Bastani
University of Pennsylvania
obastani@seas.upenn.edu

Jacob R. Gardner
University of Pennsylvania
jacobrg@seas.upenn.edu

Ryan Marcus
University of Pennsylvania
rcmarcus@seas.upenn.edu

ABSTRACT

Benchmarks determine where the database community will direct its optimization efforts and directly shape the behaviors of learned systems. However, existing benchmarks are constructed to be realistic or difficult, with no guarantee that systems can actually improve on their queries. We argue that benchmarks should instead target *headroom*, the gap between how well a query currently executes and how well it *could* execute. By identifying queries which have optimization potential, headroom provides a clear signal of where a system is underperforming. To this end, we introduce Gremlin, a system for generating adversarial benchmarks of queries with high headroom. To do so, we apply Bayesian optimization to search the joint space of queries and witness plans, seeking query-plan pairs that maximize headroom, where each witness plan provides concrete evidence of how a system could have selected a better join order. We show that Gremlin is system and dataset-agnostic, demonstrating that the generation technique is generalizable and reusable. Comparing against existing benchmarks, we find that Gremlin-generated benchmarks have multiple orders of magnitude more headroom, with the median query having 34.2× headroom to JOB + JOB-Complex's 1.4×, and summed headroom at the benchmark level of over 8 hours compared to 12.4 seconds.

VLDB Workshop Reference Format:
Jeffrey Tao, Yimeng Zeng, Natalie Maus, Haydn Jones, Osbert Bastani, Jacob R. Gardner, and Ryan Marcus. Generation of Adversarial Queries via Bayesian Optimization. VLDB 2026 Workshop: Applied AI for Database Systems and Applications (AIDB 2026).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Speculative/adversarial-queries.

*Equal contribution.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

1 INTRODUCTION

Database researchers and system developers depend upon benchmarks as a feedback mechanism that helps drive development [31]. The "performance surface area" targeted by a benchmark determines where optimization effort will be directed and what progress looks like; effort will be directed towards making the benchmark queries (and, by extension, queries with similar performance characteristics) run faster [35]. What gets measured gets managed: for example, before the introduction of the Join Order Benchmark [27], Leis et al. [29] notes that query optimization had been widely considered a "solved problem." The publication of the Join Order Benchmark, which demonstrated that existing query optimizers were far from optimal, re-ignited research in query optimization, especially in cardinality estimation [28].

This dynamic has intensified with the rise of learned database components [15, 19, 20, 60], query optimizers [33, 34, 59], and even whole systems [10, 24, 57]. Now, rather than merely being a systemic bias amongst the humans creating database systems, benchmark dependence directly shapes learned systems, since the behavior that such systems exhibit is instilled by their training data. In the learned setting, a benchmark's performance surface area determines which behaviors a system gets signal to learn, which emphasizes the need for better teaching material.

What makes a good benchmark? This dependence upon benchmarks as optimization signals has led database developers to seek out "good" benchmarks in order to guide their systems to suit the needs of real users. To serve this purpose, a "good" benchmark must simultaneously (1) *represent queries that users care about*, or must at least give developers confidence that improvements on the benchmark will correlate with improvements to user-facing performance, and (2) must *expose optimization opportunities* inside of even the most well-tuned DBMSes – that is, good benchmarks must have *headroom*, the gap between how fast a DBMS currently executes a certain query and how fast it *could* execute that query.

Existing benchmarks like TPC-H [47] and TPC-DS [38, 48] are hand-constructed to push systems to their limits, but such benchmarks are painfully difficult to construct (i.e., years of effort) and may not represent modern query workloads [21, 51, 52].[1] The high

[1]The synthetic data used by these benchmarks may also skew the evaluation of learned database components [9, 32].

cost (i.e., multiple years) of constructing and validating a high quality benchmark like TPC-DS also means that variants are limited, and thus practitioners often overfit to the benchmark (i.e., "benchmark specials" [31]).

Newer approaches, such as SQLStorm [43] and SQLBarber [25], use large language models to construct realistic user queries, but it is not clear that these queries push existing DBMSes to their limits (e.g., the majority of SQLStorm's StackOverflow queries execute in under 100ms). While these tools work well for their stated goals (e.g., identifying differences between systems, testing a wide range of capabilities), it is not clear that they are well-suited as optimization targets. More specifically, the *headroom* of these generated queries is unknown: how much better could an "optimal" DBMS really do? And for learned systems, what more do they need to learn?

Gremlin: directly targeting headroom. We propose that benchmarks could directly target headroom. Headroom is the right metric to target for benchmarks, as a difficult query is only a useful optimization signal if we have evidence that it could be executed more quickly. High headroom implies that the DBMS is underperforming on that query and other queries like it – such queries give us precise, concrete evidence for how the DBMS can be improved.

To this end, we introduce Gremlin (GeneRative Exploration Method for pLan INefficiencies), a method for generating high-headroom, *adversarial* queries. Given a particular database and schema, Gremlin generates SQL queries and witness plans for which the DBMS's optimizer can be shown to have bad performance by comparing the performance of the DBMS's plan to the performance of the witness plan (i.e., the DBMS could perform better on this query if its optimizer came up with the witness plan instead of the one that it actually selected). Put simply, Gremlin generates queries that answer the question "what is my query optimizer bad at?"

Generating benchmarks with high headroom presumes the existence of (1) an oracle that says how fast a particular query *could* be and (2) a way to enumerate queries with high headroom. Gremlin solves this problem using Bayesian optimization (BO) [44] over the joint space of queries and witness plans. For each proposed query-plan pair, it executes both the DBMS default plan and the witness plan, feeding the difference in execution latency back to the BO process as the optimization signal and optimizing to maximize the gap. Thus, as BO proceeds, it finds query-plan pairs with increasing headroom, with the witness plans' advantage serving as a lower bound for headroom.

Furthermore, as a benchmark generator, Gremlin is *automated*. Benchmark creation requires substantial effort and may be hampered by concerns of realism and customer privacy. As a system and dataset-agnostic benchmark generator, Gremlin can be run against arbitrary systems and datasets, allowing for the creation of adversarial queries targeted to a specific system and dataset without manual development effort.

A reasonable criticism of Gremlin is that the generated adversarial queries may not be representative of the "real" queries that users want to run. We mitigate this in two ways. First, we carefully designed Gremlin's search space so that every candidate query is a select-project-join-aggregate query consisting of only primary key-foreign key equijoins, with reasonable single-column predicates. This is at least a *plausible* class of queries. Second, we note that there is evidence that database customers adjust their workloads to what their system can already run effectively [35]. This suggests that the current paradigm of prioritizing benchmarks that exactly match what users are already doing may be over-indexing database optimization efforts on making fast what is already fast enough. As such, future database development needs benchmarks where current systems are underperforming, and queries with these characteristics are likely absent from current benchmarks – but can be generated by Gremlin!

We make the following contributions:

(1) We implement Gremlin, an adversarial query generator that finds queries and witness plans exhibiting high headroom by conducting Bayesian optimization over the space of possible queries and plans (§3)
(2) We show that Gremlin produces queries with multiple-orders-of-magnitude higher headroom compared to existing benchmarks and that it generalizes across datasets and systems (§4)

2 RELATED WORKS

Benchmarks. The database community has long engaged in building benchmarks in order to push database systems forward. TPC-H [47] and TPC-DS [38, 48] have become standard benchmarks for implementing database systems [5]. Both are benchmarks over synthetic data and have gaps in realism that make them unsuitable for comparing modern systems [51].

The publication of the Join Order Benchmark [27] kicked off a wave of interest in cardinality estimation and query optimization for analytical workloads. The JOB is a fixed set of 113 hand-written queries over the real IMDB dataset and was explicitly authored to be "realistic" and "challenging." JOB-Complex [56] similarly consists of 30 hand-written queries, but exercises a wider variety of query features such as non-PK-FK joins. Being hand-written, both benchmarks are limited in the number of queries and the space of query optimizer weakness exposed. The JOB authors, writing 10 years later, note that there is a danger of overfitting to JOB due to "its small dataset and limited query workload" [28].

In comparison, the Cardinality Estimation Benchmark [39] is an automated approach, explicitly designed to compensate for the JOB's number of queries being too small for supervised learning. The CEB generates queries using a fixed set of parameterized query templates with hand-written join graphs, with manually configured parameter distributions for varying the predicate columns and values. However, the CEB was designed to cover a broad range of different cardinality estimation cases, as opposed to highlighting specific cases where the optimizer struggles. The same is true for the DSB [9], which extends TPC-DS with more complex data distributions and parameterized query templates.

Stack [33] is a real database dump and set of queries from StackExchange. The queries are a mix of real queries from StackExchange analytics dashboards and hand-written queries intended to be challenging for query optimizers. The Stack benchmark is a one-time artifact that will be hard to replicate: its existence is the product of a long-lived and popular website, as well as the rare willingness to make both the data and queries public.

The rise of large language models has inspired new approaches to generating benchmarks. SQLStorm [43] provides automated generation of large and diverse benchmarks over a given database schema, relying on pretraining to give realistic queries. SQLBarber [25] uses an LLM in conjunction with Bayesian optimization to create customized, realistic benchmarks tailored to fit the aggregate statistics released by database vendors about their real workloads. Both seek to automate the creation of large, realistic benchmarks, automating away much of the problem of costly, manual benchmark creation. In this new era, we believe that targeted, challenging benchmarks are still relevant in pushing database systems forward: the JOB authors wrote that JOB was widely adopted because it was targeted (to evaluate cardinality estimation and join orderings) and easy to implement [28].

Avoiding cardinality estimation. In response to the problem of join orderings, database researchers have sought to mitigate or avoid their consequences. On the systems side, adaptive query processing detects bad plans and switches mid-execution [1]. Adaptive query processing can be competitive with learned query optimization in reducing overall workload runtimes [64].

On the theoretical side, worst-case optimal join algorithms avoid enumerating large intermediate results that binary joins may incur by instead performing multi-way joins per-attribute across relations [40] and have begun to show promising practical results [13, 55]. For acyclic conjunctive queries, there has been renewed interest in the Yannakakis algorithm [61], which avoids the problem of materializing large intermediate results incurred by conventional binary join algorithms by using semijoins to filter away input tuples that do not appear in the output. Though the original algorithm has a good theoretical worst-case bound, recent works have sought to reduce the constant factors to make its application practical in real systems [3, 4, 6, 45, 53, 65].

Better cardinality estimation. From the confluence of interest in cardinality estimation and improvements in machine learning, there has been a wave of works that propose learned cardinality estimators to replace the traditional heuristic optimizers that came before [17, 54]. These learned techniques fit into two main camps: those whose estimates are driven by query features [19, 39] and those that model the underlying data distributions [15, 60].

Recent theoretical works generate tight worst-case bounds for "pessimistic" cardinality estimation, which never underestimate the true cardinality [8, 63].

Learned query optimization. Other works have attempted to bypass cardinality estimation by replacing traditional query optimization wholesale with learned optimizers. Many of these works rely on reinforcement learning (e.g., [2, 7, 16, 34, 49, 58, 59, 62, 66]). Learned query optimizers are known to have regression problems [33, 58], which are one of the primary obstacles to their wider adoption. Recent work has reported that, when standardizing train/test splits across different learned optimizers, PostgreSQL's traditional optimizer matches or outperforms them [26], underscoring both the fragility of current LQOs and the degree to which reported gains hinge on the choice of evaluation workload. Gremlin can potentially be used as a regression finder: if queries adversarial to one system are not adversarial on a baseline, then the system has a demonstrated performance regression.

Synthesized data systems. Instance-optimized systems have previously been proposed as specialized systems that are tuned for maximum performance on a single workload [10]. Recent works have demonstrated this as a possible direction for data systems in the new LLM era of synthesized code, generating code per-query [24] or per-workload [57]. Gremlin could be useful for testing the limits of the code that these systems generate and help understand whether there are workloads for which these systems fail to generate performant code.

Bayesian optimization in databases. Gremlin implements adversarial benchmark generation using Bayesian optimization (BO). BO has previously been applied in the databases setting for tuning system configuration parameters [50].

This work builds upon our previous work applying BO to conduct offline query optimization [46], which combined BO over a structured space of query plans with real query plan executions in order to find highly optimized query plans. We apply a similar structured BO scheme here to search the joint space of queries and witness plans and optimize for headroom.

3 SYSTEM

Gremlin casts adversarial benchmark generation as an optimization problem. Over a constrained space $\mathcal{A}$ of (query, witness-plan) pairs, it searches for a pair (Q, P) whose *headroom*—the gap between the latency of the plan the DBMS chooses for Q and the latency of the witness plan P—is as large as possible. Measuring the headroom of a candidate requires *executing* it on the target system, so the objective is an expensive, noisy, black-box function of a discrete and highly structured input. These are exactly the conditions under which *Bayesian optimization* (BO) is effective. Gremlin therefore (i) uses continuous representations in which queries and witness plans are jointly represented, and (ii) runs BO over those representations, decoding each proposed latent point into a concrete query–plan pair, executing it, and feeding the measured headroom back to the optimizer. We first review BO (§3.1), then describe the joint latent space (§3.2), and finally the end-to-end generation pipeline (§3.3).

3.1 Bayesian Optimization

Bayesian optimization [14] is a sample-efficient method for optimizing black-box functions that are expensive to evaluate. Given an input space $\mathcal{X}$ and an unknown objective $f : \mathcal{X} \to \mathbb{R}$, BO seeks an input $\mathbf{x}^\star \in \mathcal{X}$ that maximizes f in as few evaluations as possible. It maintains a probabilistic *surrogate model* of f—in our case a Gaussian process (GP)—and uses the surrogate to decide which candidate to evaluate next. Each iteration fits the surrogate to the observations collected so far, applies an *acquisition function* to propose new candidates, evaluates those candidates with the true black-box objective, and updates the surrogate with the new observations. In Gremlin, the expensive black-box evaluation is query execution for the default plan and generated witness plan, and the observation is the measured headroom of a decoded query–plan pair.

Our objective is high-dimensional and defined over structured objects, so we use BO in a continuous latent space rather than directly over discrete SQL strings and join orders. Following latent-space BO [36], Gremlin decodes continuous vectors into structured candidates before evaluation and then fits the surrogate on the corresponding latent vectors and measured headroom. We use trust-region BO (TuRBO) [12] with Thompson sampling [42]: acquisition is restricted to a local region around strong candidates, and the region expands or contracts as the search succeeds or stalls. This is the same broad optimization setting as BayesQO [46], but here the search variables include both the query and its witness plan. As in BayesQO, trust region BO is a competitive *global* optimization scheme able to find solutions that are far from its initialization points [12].

3.2 A Joint Latent Space over Queries and Plans

To search queries and witness plans together, Gremlin represents each pair as a single continuous vector $\mathbf{z} = [\,\mathbf{z}_q\,;\,\mathbf{z}_p\,] \in \mathbb{R}^{320}$, the concatenation of a 256-dimensional query embedding $\mathbf{z}_q$ and a 64-dimensional plan embedding $\mathbf{z}_p$. BO operates on the full vector $\mathbf{z}$; to evaluate a point, we split it and decode each half independently. The two halves are constructed differently.

Queries: a frozen embedding with a learned decoder. We map a query into $\mathbf{z}_q$ using a *frozen*, pretrained text-embedding model (OpenAI `text-embedding-3-large`), reduced to 256 dimensions via its Matryoshka representation [22]. This direction—text to vector—requires no training. The reverse direction—vector back to a query—is *learned*: a small feed-forward network maps $\mathbf{z}_q$ to four *soft-prompt* token embeddings [30], which are prepended to a fine-tuned Qwen2.5-0.5B-Instruct language model [41] that then autoregressively emits the query.

We train the feed-forward network and the language model jointly with a reconstruction objective: given the embedding of a query, reproduce that query's text (a cross-entropy loss on the query tokens). Training over a large corpus of schema-valid queries makes the embedding space *decodable*, so that latent points proposed by BO—which need not coincide with the embedding of any previously seen query—decode into plausible queries. This query component uses the text embedding model as the encoder and the fine-tuned language model as the decoder. We do not require an exact encode/decode round trip: the surrogate is fit on (proposed $\mathbf{z}$, measured headroom) pairs, which absorbs any gap between a proposed $\mathbf{z}_q$ and the embedding of the query it decodes to.

These training queries are sampled uniformly from the grammar, not from a preexisting workload. As the feed-forward network is trained on queries from a particular grammar, it must be retrained for new grammars (e.g., to run Gremlin with a new database schema). In our experiments in §4, the same decoder and grammar are used across different systems on the same schema, yet the queries surfaced for each system use entirely disjoint combinations of joined tables. This is consistent with the search, rather than the decoder, determining which queries are surfaced.

Plans: a trained variational autoencoder. The plan half $\mathbf{z}_p$ is produced by a variational autoencoder (VAE) [18] carried over from BayesQO [46]: a Transformer encoder–decoder VAE with a 64-dimensional latent space over serialized join trees. The encoder maps a plan to a posterior distribution whose mean we take as $\mathbf{z}_p$; the decoder samples a plan (a join order) from a latent vector. During adversarial query generation, we apply $\mathbf{z}_p$ to the decoder to resolve the plan.

Though the plan encoding format is not novel to this work, we briefly explain it here due to its significance in the optimization process. We train the plan VAE with strings that correspond to join orders. Unlike in BayesQO [46], the plan language consists only of join orders without support for multiple table aliases or operator selection. We omit aliases as our query grammars only support queries with up to a single copy of each table, and we omit operator selection as DuckDB, the primary system targeted by our experiments, lacks multiple join operators.

The VAE decodes compressed latent space vectors into plan strings, which must be processed further to resolve a join order. We guarantee that all plan strings resolve to valid join orders for a given query by construction. In the plan encoding language, pairs of symbols refer to the left and right children of the next binary join to specify. The language has one symbol per table in the schema. Plan strings are processed in a pair-wise fashion from left to right. After a table is joined at least once, occurrences of its symbol later in the string instead refer to the largest join subtree that it is a child of.

Symbols in the decoded string may specify invalid joins: a decoded symbol may refer to a table not present in the query, or pairs of symbols may specify an invalid join such as the same subtree joined with itself. Our decoding process performs a deterministic modular/hashing repair step that maps any invalid symbol into the space of semantically valid symbols in that position.

Concatenating $\mathbf{z}_q$ and $\mathbf{z}_p$ yields the joint space that BO searches.

3.3 Adversarial Query Generation

Objective. Gremlin maximizes headroom over the constrained pair space $\mathcal{A}$. For a query Q and witness plan P, let $L_{\text{def}}(Q)$ be the latency of the plan the DBMS's own optimizer chooses for Q, and let $L_{\text{wit}}(Q,P)$ be the latency of Q executed under the witness plan P. We consider two complementary headroom objectives:

$$f_{\text{abs}}(Q,P) = L_{\text{def}}(Q) - L_{\text{wit}}(Q,P), \qquad f_{\text{rel}}(Q,P) = \frac{L_{\text{def}}(Q)}{L_{\text{wit}}(Q,P)}.$$

The *absolute* objective rewards wall-clock seconds saved and tends to surface expensive queries, whereas the *relative* objective rewards the speedup factor and surfaces queries the optimizer handles disproportionately badly irrespective of their absolute cost. A large value of either is concrete evidence that the system *could* have executed Q far faster.

The optimization loop. Combining §3.1 and §3.2, each Gremlin iteration (Figure 1):

(1) proposes a latent point $\mathbf{z}$ by trust-region Thompson sampling;
(2) splits $\mathbf{z} = [\,\mathbf{z}_q\,;\,\mathbf{z}_p\,]$ and decodes $\mathbf{z}_q$ into a query $\hat{Q}$ (soft-prompt network followed by the language model; see below)
(3) decodes $\mathbf{z}_p$ into a witness plan $\hat{P}$ (the plan VAE decoder);

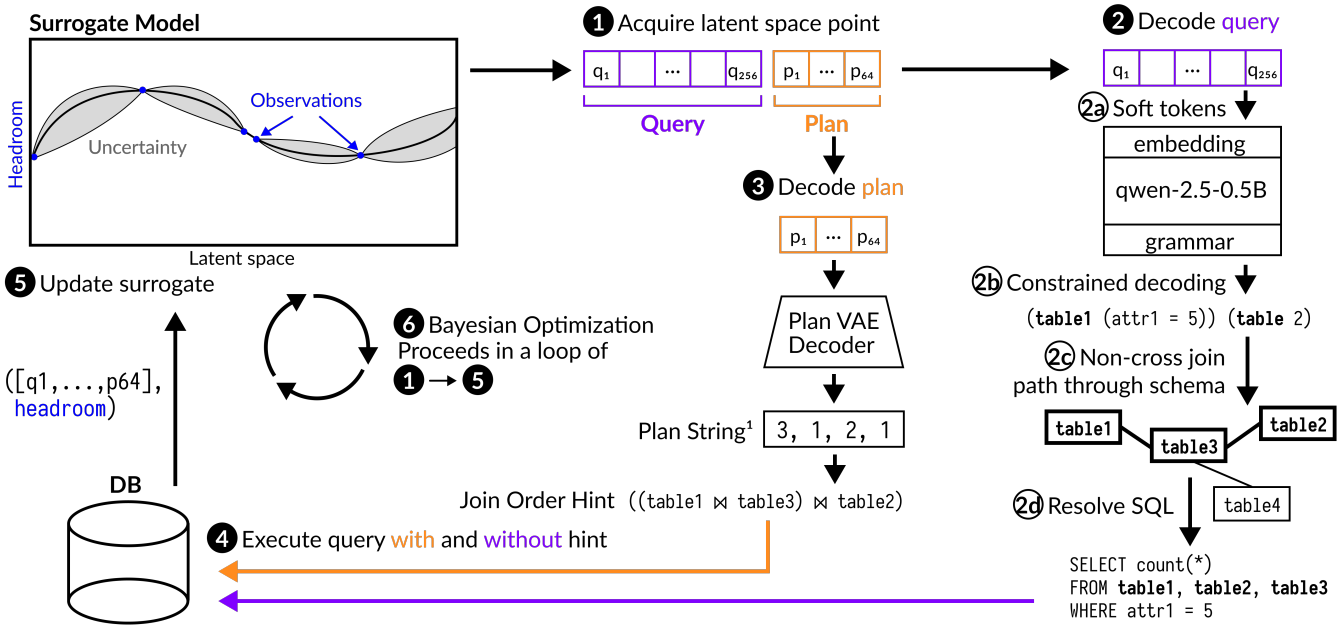

Figure 1: The Gremlin generation loop.

(4) executes $\hat{Q}$ twice—once under the DBMS's default plan and once forced onto $\hat{P}$—to measure L_{def} and L_{wit};
(5) scores the pair with f_{abs} or f_{rel} and updates the GP surrogate.

Iterating drives the search toward higher-headroom pairs; the output is a stream of benchmark entries $(\hat{Q}, \hat{P})$ ranked by their measured headroom.

Grammar-constrained decoding. A decoded query is useful only if it is valid for the target schema, so we enforce validity at decode time. The fine-tuned query model is served with vLLM [23]; the four soft-prompt tokens are injected directly as input embeddings, and generation is constrained by an XGrammar-style guided decoder [11] that, at each step, masks out any token that would violate a per-schema grammar. Every decoded string is therefore a syntactically valid query specification by construction, requiring no rejection sampling.

Search-space design. The grammar that constrains decoding also defines the space $\mathcal{A}$ that Gremlin searches, and is the primary knob for steering the generator toward plausible queries. In our IMDB experiments, we study three grammar variants that differ in how many tables they expose and how restrictive they are about predicates. These grammars let us vary the size and diversity of the generated query space without changing the BO machinery. To demonstrate that the method is not tied to a single schema, we also target Stack-style schemas with their own grammars and query decoders. Gremlin executes candidates on both DuckDB and PostgreSQL.

Rather than generate syntactically and semantically valid SQL strings, which would not be achievable with just a grammar, we design the query string language to specify only the tables to be joined and any filter predicates that apply to those tables. We implicitly infer join predicates by constructing the PK-FK join graph of the entire schema, in which nodes are tables and edges are PK-FK relationships. From this join graph, we take the subgraph consisting of the decoded set of tables and add the join predicates corresponding to all PK-FK edges present in the subgraph. The decoded set of tables may not necessarily be connected in the join graph; we use a Steiner tree to get the minimum additional tables to join in, adding their join predicates but no additional filter predicates.

For example, `(title (production_year > 0750)) (name (gender most_popular))` specifies a query over the title and name tables. Since these tables do not have a PK-FK relationship in the IMDB schema, the Steiner tree brings in the cast_info table to connect the join subgraph. The filter predicate on title specifies that the production year should be greater than the 75th percentile across all values in the column. The filter predicate on name specifies that the gender should be equal to the mode value of the column. Taken together, these form the query that could be read as "The cast members of the most commonly occurring gender in the quarter of all movies that are most recent."

As a final step, the resolved list of tables, join predicates, and filter predicates is formulated as a SQL query to be submitted to the target system. Queries have a `count(*)` aggregation as the top level projection to avoid transmitting large results.

Reusability. Retargeting Gremlin to a new schema or DBMS requires a schema grammar, a compatible query decoder, and a plan representation for measuring default and witness-plan latencies. Once those components are provided, the latent-space BO loop is unchanged. Gremlin is thus a reusable benchmark generator: it can be rerun with new datasets and against new or upgraded systems.

$$
\begin{aligned}
\textit{query} &\rightarrow \textit{title}^{*}\ \textit{name}^{*}\ \ldots \\
\textit{title} &\rightarrow \texttt{(title}\ \textit{title\_prod\_year}^{*}\ \textit{title\_season\_nr}^{*}\ \ldots\ \texttt{)} \\
\textit{title\_prod\_year} &\rightarrow \texttt{(production\_year}\ \textit{intop}\ \textit{intval}\ \texttt{)} \\
&\ldots \\
\textit{intop} &\rightarrow \texttt{>} \mid \texttt{<} \mid \texttt{=} \mid \texttt{!=} \\
\textit{intval} &\rightarrow \texttt{min} \mid \texttt{median} \mid \texttt{max} \mid \textit{int int int int} \\
\textit{strop} &\rightarrow \texttt{most\_popular} \mid \texttt{median}
\end{aligned}
$$

Figure 2: Example query grammar, expanded for one table and column predicate.

Grammars such as G1, which we use in the experiments below, can be inferred automatically from the database schema. As illustrated in Figure 2, from the database schema, we emit a context-free grammar: the start symbol consists of each table appearing at most once, and each table consists of its name and predicates over its columns appearing at most once. One table production rule is added for each table in the schema and one typed predicate production rule is added for each column of each table depending on the column type. Predicates are described abstractly as comparisons to column value aggregates: string columns can be equal to the most popular or median popularity value, numeric columns use a comparison operator and literal value corresponding to the min, median, max, or some specific percentile. In this way, simple conjunctive queries in the schema can be expressed, and all grammatical strings are guaranteed to correspond to well-formed queries.

Finally, while execution of default plans merely implies executing the resolved SQL, execution of witness plans requires a mechanism that allows us to specify the join order. Such a mechanism is present on both of the systems we perform experiments on (DuckDB and PostgreSQL) by specifying the join tree directly as a parenthesized S-expression of the tables in the `FROM` clause and preceding DBMS-specific statements to disable join reordering.

4 EXPERIMENTS

We evaluate Gremlin by measuring the *headroom* of the queries it generates—the gap between the latency of the plan the DBMS chooses by default and the latency of the witness plan Gremlin discovers—and comparing it against the headroom present in existing benchmarks. We report results for three IMDB grammars and one Stack grammar, executed on DuckDB and PostgreSQL. §4.1 describes the experimental setup; §4.2 reports the headroom of generated queries against existing benchmarks; §4.3 examines generalization across systems and schemas; §4.4 examines the effect of grammar design; §4.5 examines the potential source of the significant headroom found by Gremlin; §4.6 reports the reproducibility of the discovered headroom; and §4.7 examines the costs of generating adversarial benchmarks.

4.1 Setup

Workloads and grammars. We run Gremlin with the four grammars summarized in Table 1: three over the IMDB schema and one over a Stack schema. The IMDB grammars vary two design axes—the number of joinable tables and whether predicates are optional or a single predicate is mandatory—while the Stack grammar demonstrates that the method transfers to a different schema.

Table 1: Grammars used to generate adversarial queries.

Grammar	Schema	#Tables	Predicates	System(s)
G1	IMDB	21	Arbitrary	DuckDB, PostgreSQL
G2	IMDB	21	Single	DuckDB
G3	IMDB	8	Arbitrary	DuckDB
GStack	Stack	10	Arbitrary	DuckDB

G1 is our full, unrestricted grammar on the IMDB schema, which allows for any number of tables and any number of filter predicates. G2 is restricted to only a single filter predicate per-query. G3 allows arbitrary filter predicates, but narrows the set of tables to only those within 2 hops in the join graph from the main `title` table. GStack is an unrestricted G1-style grammar for the Stack schema. Note that, as described in §3.3, join predicates are automatically inferred from the schema's PK-FK join graph for the set of tables present in the decoded query string.

Optimization. Each Gremlin run initializes the surrogate with 1,000 randomly sampled query–plan pairs and then optimizes with trust-region Thompson sampling (§3.1). We run Gremlin under both the absolute and the relative headroom objective. Each optimization run on a specific grammar/system combination was run for approximately 5 days before being cut off.

Metrics. For a generated query Q with witness plan P, we report the *absolute advantage* $L_{\text{def}}(Q) - L_{\text{wit}}(Q, P)$ in seconds and the *relative advantage* $L_{\text{def}}(Q)/L_{\text{wit}}(Q, P)$. Both are *lower bounds* on the true headroom: the witness plan is one concrete better plan, not necessarily the optimal one, and because we cap execution at the validation timeout, the default-plan runtime (and the advantage between default-plan and generated witness plan) of the slowest queries is capped.

Validation. BO-time measurements are taken under a single execution and an adaptive timeout, which can lead to measurement-noise false positives. We therefore re-execute every discovered query–plan pair under a uniform protocol—5 repeated runs of both the default and witness plan at a 300 s timeout—and report mean latencies. We keep only pairs that reproduce a meaningful gap ($\geq$ 1 s absolute *and* $\geq 2\times$ relative), and for each configuration we report the top 143 pairs ranked by absolute advantage, so that all configurations are compared at the same cardinality.

Baseline: headroom in existing benchmarks. To quantify how much headroom *existing* benchmarks contain, we run a plan-level optimizer—BayesQO [46], which applies the same BO machinery to search plans for a *fixed* query—over the queries of JOB [27], JOB-Complex [56], and the Stack benchmark [33] (the SO_FUTURE split, as used in [46]). We run BayesQO for 4000 plan executions per-query, which is the same number used in the BayesQO paper. For each benchmark query, the baseline's advantage is the gap between its default plan and the best plan BayesQO finds over the entire search. This is a strong baseline: it measures the headroom that a state-of-the-art plan optimizer can extract from each existing

query, using the same Bayesian optimization technique Gremlin uses internally. Note that, similarly to the results for the queries generated with Gremlin, the advantage metrics for the baselines are *lower bounds* on the true headroom.

Implementation. DuckDB queries were executed using v1.4.3 of the Python library. Workers were configured with `memory_limit` set to 32GB. To execute witness plans, we prevented the DuckDB optimizer from reordering joins by setting `disabled_optimizers` to `join_order, build_side_probe_side`.

PostgreSQL queries were executed against v16.3 on the same hardware as DuckDB queries. We disable JIT and GEQO, configure `shared_buffers` to 32GB and `work_mem` to 16MB. To execute witness plans, we set `join_collapse_limit` to 1.

4.2 Headroom of Generated Queries

Table 2 compares the headroom of Gremlin-generated queries against the existing-benchmark baselines. The Gremlin-generated queries exhibit dramatically more headroom.

On IMDB, the median Gremlin query under Grammar 1 on DuckDB has a relative advantage of 34.2× and an absolute advantage of 231.9 seconds (when optimizing against a default plan execution timeout of 300 seconds, implying that the true advantage could be higher still on some queries). In comparison, the median advantage BayesQO can find on a JOB or JOB-Complex query is less than 1 second on both DuckDB and PostgreSQL; in aggregate, Grammar 1 surfaces 57.8× (PostgreSQL) to 2,526× (DuckDB) more total absolute headroom than the 143-query JOB+JOB-Complex benchmark. On Stack, Gremlin queries have a median relative advantage of 6.8×, a median absolute advantage of 30.9 seconds, and expose 117× more total headroom than the 200-query SO_FUTURE benchmark.

In short, queries generated by Gremlin have *multiple orders of magnitude* more headroom than the queries in existing benchmarks. Figure 3 shows that this holds across the whole distribution of generated queries, not only at the extreme tail.

In the rest of this section, we examine the queries generated by optimizing for absolute headroom. The results on these queries were more robust, as many relative headroom improvements made already-fast queries even faster, where measurement noise is a larger fraction of the measured latency.

4.3 Generalization across Systems and Schemas

Gremlin finds high-headroom queries against two very different execution engines. Run with the same grammar (G1), it surfaces a median absolute advantage of 16.8s against PostgreSQL's optimizer and 231.9s against DuckDB's. More dramatically, on both systems it finds queries that time out after 300s under the DBMS's own plan, yet execute in under a second using the witness plan. As we explore in §4.4, the two systems mis-optimize different queries, but Gremlin exposes large gaps in both.

The method also transfers across schemas: the GStack queries exhibit different characteristics in number of joins and predicates per query compared to G1 while still having headroom well above the baseline benchmarks, suggesting that Gremlin can be rerun against the same system with different datasets to generate meaningfully different benchmarks.

4.4 Characterizing Generated Queries

Differences between grammars. Here, we examine the differences between queries generated by our grammars. As visualized in Figure 3, the restrictions imposed on G2 and G3 do meaningfully reduce their headroom relative to G1. Still, all adversarial grammars generated sets of queries with 2-3 orders of magnitude higher headroom than the baseline benchmarks.

Next, we characterize the composition of the queries generated by our four grammars. As shown in Figure 4, G1, G1-PostgreSQL, and G3, which are defined to be able to generate arbitrary predicates, have similar distributions of filter predicates per query with a median of 1. G2, which is defined to use a maximum of 1 filter predicate per query, largely eschews having filter predicates whatsoever. GStack, which is executed on DuckDB using a different dataset from G1–G3, tends to have multiple predicates per query, ranging from 1–5. In comparison, most JOB queries have 2–10 filter predicates.

Despite this similarity in number of predicates per query, the different grammars are indeed distinct sets of queries and exhibit substantial variation within a single grammar's set of queries. We analyzed the sets of tables (ignoring predicates) used by queries generated by G1–3 on DuckDB and G1 on PostgreSQL. G1 (DuckDB) and G1 (PostgreSQL) contained no overlapping table sets. G1 and G2 on DuckDB had only 4 overlapping table sets, constituting 5 G1 queries and 5 G2 queries. Most table sets were unique or only repeated < 5 times within a grammar. For G1 PostgreSQL, one table set in particular contained 73 repetitions with different values for a certain predicate (constituting half of the benchmark). For comparison, the IMDB schema contains 95,136 distinct non-cross-join queries when only considering sets of joined tables (calculated by counting the number of distinct connected subgraphs contained in the PK-FK join graph).

Number of joined tables. Figure 5 characterizes the distribution of number of joined tables per-query in G1 vs. JOB. On the whole, G1 queries contain more joined tables than JOB queries. As we explore further in §4.5, the headroom in Gremlin-generated queries is likely coming from cardinality estimation errors, so more joins would tend to lead to higher error.

G4. We attempted another grammar, G4, which combined the single-predicate restriction of G2 with the restricted schema of G3. However, the resulting space of possible queries was too small to generate many queries with significant headroom, so we do not report its results here. This demonstrates how grammar design meaningfully impacts the optimization process by defining the space of possible queries.

4.5 Q-error

We hypothesized that the high headroom of the adversarial queries generated by Gremlin would mostly be explained via cardinality mis-estimation. To test this, we collected pre-aggregation estimated and true cardinalities to calculate Q-error [37] for the highest headroom queries generated with Gremlin G1 ($n = 143$) and JOB ($n = 113$) on DuckDB. The resulting Q-error distributions are shown in Figure 6.

Table 2: Headroom of GREMLIN-generated queries (validated top-143 per configuration) vs. the headroom of existing benchmarks (best plan found by BayesQO plan optimization per query). "Total abs." sums the absolute advantage over all queries. Absolute advantages are right-censored at the 300 s validation timeout.

			Absolute advantage (s)		Relative advantage (×)	
Configuration	n	Total abs. (s)	median	max	median	max
GREMLIN (generated)						
G1 (IMDB, PostgreSQL)	143	6,645	16.8	299.9	23.3	4457
G1 (IMDB, DuckDB)	143	31,333	231.9	299.6	34.2	809
G2 (IMDB, DuckDB)	143	16,758	88.8	292.2	14.7	48
G3 (IMDB, DuckDB)	143	706	3.5	19.2	8.1	40
GStack (Stack, DuckDB)	143	14,259	30.9	299.2	6.8	390
Existing benchmarks (BayesQO plan-optimization headroom)						
JOB + JOB-Complex (IMDB, DuckDB)	143	12.4	0.04	4.9	1.40	39
JOB (IMDB, PostgreSQL)	113	115	0.129	34.3	2.48	97
SO_FUTURE (Stack, DuckDB)	200	121.4	0.009	4.5	1.05	10
SO_FUTURE (Stack, PostgreSQL)	200	809	1.9	62.5	2.2	97

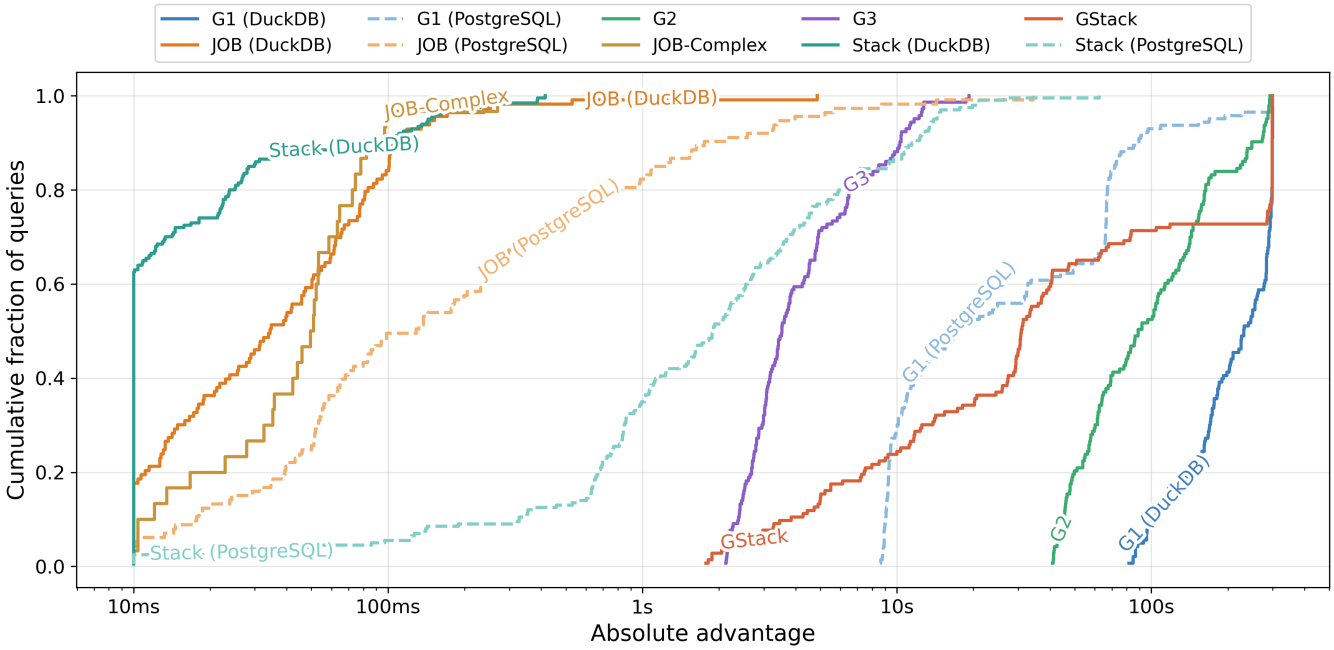

Figure 3: Cumulative distribution of headroom across generated queries and baselines. Towards the bottom right corner implies higher headroom on more queries. Note that the x axis is log-scaled. Due to our 300s cap on execution time, maximum advantage at ~300s is a lower bound. GStack refers to our adversarial grammar while Stack (DuckDB, PostgreSQL) refer to baselines.

GREMLIN G1 (p_{50}=105,625×, p_{90}=27,554,147×) had much higher Q-error than JOB (p_{50}=138×, p_{90}=4,437×). Though these are only measurements of Q-error at the top of the query plan (meaning errors from plan subtrees could compound or cancel out), the distributions differ by multiple orders of magnitude. As such, cardinality mis-estimation is likely the primary cause of the high headroom (though it may not be the *only* cause), with witness plans outperforming the DBMS query optimizer's plans by selecting join orders that better conform to the true cardinalities. We discuss further analysis as future work in §5.

4.6 Reproducibility of Discovered Headroom

The reported gaps are not measurement artifacts. Under the independent 5× re-execution protocol, 96.8%–100% of the discovered query–plan pairs reproduced a meaningful advantage across the IMDB grammars (e.g., 1,152 Grammar-1/PostgreSQL pairs revalidated with 96.8% reproducing the ≥ 1 s / ≥ 2× gap, and Grammar 3 reproducing on 100% of pairs with no regression beyond 100 ms). The small number of non-reproducing pairs is dominated by cases where the default plan timed out during BO and was found

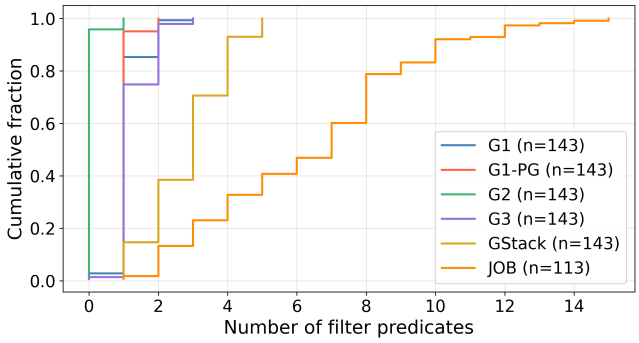

Figure 4: Cumulative distribution of number of filter predicates per query on all GREMLIN-generated benchmarks and JOB.

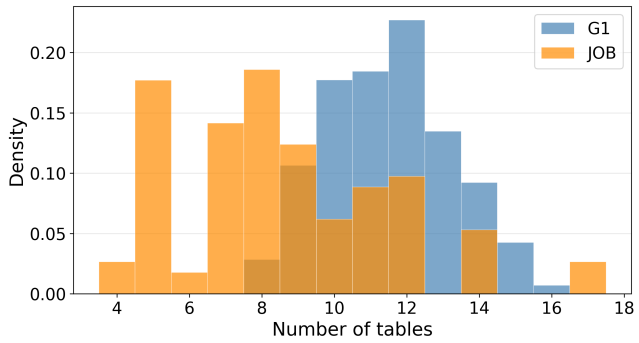

Figure 5: Distribution of number of tables joined together per query for G1 vs. JOB.

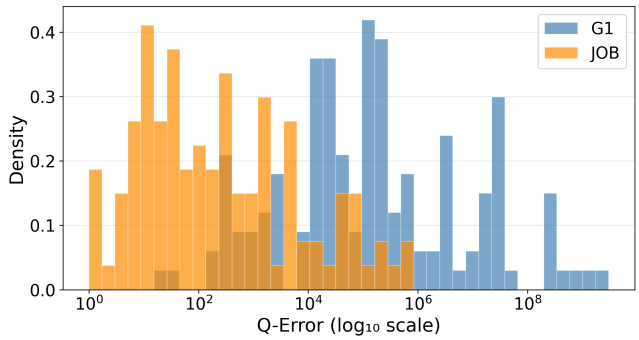

Figure 6: Distribution of Q-error for G1 queries vs. JOB. Note that the x axis is log-scaled.

to be acceptably fast under fresh measurement—a known artifact of single-shot, BO-time observations, which the validation pass is designed to filter out.

4.7 Generation Cost

All learned components were executed on a cluster with 5 NVIDIA RTX A6000 (Ampere) GPUs. Training of the query decoder and plan VAE took 50 GPU-hours each. Adversarial query generation runs were conducted for 5 days for each of 5 grammar/system runs and run on 4 GPUs, totalling ~ 2400 GPU-hours, though much of this time was spent waiting for database query executions. All experiments were run against a cluster of 20 database worker machines hosting read-only copies of the datasets.

5 DISCUSSION

The experiments in §4 demonstrate that GREMLIN successfully generates high-headroom queries and witness plans across different systems, datasets, and grammars. The witness plan generated during the GREMLIN optimization process provides an existence proof that improvement is possible, and the measured headroom gives the degree of potential improvement. Notably, our adversarial queries generated with GREMLIN have orders of magnitude higher headroom than the JOB and Stack benchmarks despite executing over the same datasets and despite these benchmarks emphasizing the difficulty of cardinality estimation. Even though cardinality estimation is already known to be a problem, the GREMLIN adversarial queries demonstrate that the problem is far more severe than the existing benchmarks expose.

We emphasize that GREMLIN is an adversarial query *generator*: it can be run on other systems and with other datasets. GREMLIN is best thought of as a reusable *bug finder* rather than a benchmark or workload generator: each high-headroom query/witness pair is concrete evidence that the query optimizer underperformed, with the witness plan demonstrating what it could have done instead.

For reproducibility and for use in future comparison and bug-finding efforts, we release both the query generator and the generated set of queries used in our experiments as part of the artifacts for this paper.

Limitations. It is experimentally unknown whether improving DBMS performance on the queries generated by GREMLIN will lead to better performance on other benchmarks or on real-world workloads. We have constructed the query grammars to match a common class of queries (SPJA, PK-FK joins, simple equality/inequality filter predicates), but the set of generated queries does not necessarily correspond to a realistic workload in aggregate. The realism of the generated queries could potentially be improved by changing the grammar to target different classes of queries or by incorporating an LLM in the optimization loop to edit queries to be more semantically meaningful. Finally, GREMLIN makes no guarantee of completeness in its bug-finding. The grammar bounds the space of reachable queries, and BO finds high-headroom queries within it—but they are not necessarily those with the highest possible headroom, nor a complete account of the ways the optimizer can underperform.

Future work. Whether this technique for stressing DBMS components works beyond cardinality estimation is an open question. A natural follow-up experiment would be to provide the DBMS query optimizer with true cardinalities when evaluating the DBMS plan quality during the GREMLIN optimization inner loop.

Furthermore, we are interested in evaluating if this technique will continue to find new adversarial queries if a system is changed to improve its performance on the first set of adversarial queries.

Doing so requires investigating the cause of the cardinality misestimation issues on one of the systems we targeted, fixing it, and then generating new adversarial queries and measuring their headroom.

6 CONCLUSION

We argued that benchmarks should target *headroom*, the gap between how fast a query executes and how fast it could execute, rather than realism or difficulty alone, as it provides a clearer signal about how systems (both traditional and learned) can be improved. We introduced Gremlin, which generates queries with high headroom by performing Bayesian optimization over the joint space of queries and witness plans, measuring headroom by comparing the execution latency of the DBMS's default plan to the latency of the witness plan. Across two systems (DuckDB, PostgreSQL) and two datasets (IMDB, Stack), Gremlin produces queries with multiple orders of magnitude higher headroom compared to existing benchmarks. We found that each of our grammars produces a diverse set of queries, varying in their table sets, number of joined tables, and number of filter predicates. We also analyzed query-level Q-error of our G1 benchmark vs. JOB, which gives evidence that cardinality estimation errors are likely the primary cause of the high headroom in our G1 queries. Gremlin's automated, dataset and system-agnostic design opens the door to future applications in finding regressions, comparing systems, and understanding the potential faults of learned and synthesized systems.

REFERENCES

[1] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Dallas, Texas, USA) *(SIGMOD '00)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/342009.335420

[2] 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

[3] Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. 2025. Instance-Optimal Acyclic Join Processing Without Regret: Engineering the Yannakakis Algorithm in Column Stores. *Proc. VLDB Endow.* 18, 8 (April 2025), 2413–2426. https://doi.org/10.14778/3742728.3742737

[4] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proc. VLDB Endow.* 17, 11 (July 2024), 3215–3228. https://doi.org/10.14778/3681954.3681995

[5] Peter Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Revised Selected Papers of the 5th TPC Technology Conference on Performance Characterization and Benchmarking - Volume 8391*. Springer-Verlag, Berlin, Heidelberg, 61–76. https://doi.org/10.1007/978-3-319-04936-6_5

[6] Daniela Böhm, Georg Gottlob, Matthias Lanzinger, Davide Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. 2025. Selective Use of Yannakakis' Algorithm to Improve Query Performance: Machine Learning to the Rescue. arXiv:2502.20233 [cs.DB] https://arxiv.org/abs/2502.20233

[7] 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

[8] Kyle B. Deeds, Dan Suciu, and Magdalena Balazinska. 2023. SafeBound: A Practical System for Generating Cardinality Bounds. *Proc. ACM Manag. Data* 1, 1, Article 53 (May 2023), 26 pages. https://doi.org/10.1145/3588907

[9] 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

[10] Jialin Ding, Ryan Marcus, Andreas Kipf, Vikram Nathan, Aniruddha Nrusimha, Kapil Vaidya, Alexander van Renen, and Tim Kraska. 2022. SageDB: An Instance-Optimized Data Analytics System. *Proc. VLDB Endow.* 15, 13 (Sept. 2022), 4062–4078. https://doi.org/10.14778/3565838.3565857

[11] Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. 2025. XGrammar: Flexible and Efficient Structured Generation Engine for Large Language Models. arXiv:2411.15100 [cs.CL] https://arxiv.org/abs/2411.15100

[12] David Eriksson, Michael Pearce, Jacob R Gardner, Ryan Turner, and Matthias Poloczek. 2019. *Scalable global optimization via local Bayesian optimization*. Curran Associates Inc., Red Hook, NY, USA.

[13] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.* 13, 12 (July 2020), 1891–1904. https://doi.org/10.14778/3407790.3407797

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

[15] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: learn from data, not from queries! *Proceedings of the VLDB Endowment* 13, 7 (March 2020), 992–1005. https://doi.org/10.14778/3384345.3384349

[16] 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

[17] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-depth Study. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1214–1227. https://doi.org/10.1145/3514221.3526154

[18] 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

[19] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. https://vldb.org/cidrdb/papers/2019/p101-kipf-cidr19.pdf

[20] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA. https://doi.org/10.1145/3183713.3196909

[21] Skander Krid, Mihail Stoian, and Andreas Kipf. 2025. Redbench: A Benchmark Reflecting Real Workloads. https://doi.org/10.1145/3735403.3735998 arXiv:2506.12488 [cs].

[22] Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, and Ali Farhadi. 2022. Matryoshka Representation Learning. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 30233–30249. https://proceedings.neurips.cc/paper_files/paper/2022/file/c32319f4868da7613d78af9993100e42-Paper-Conference.pdf

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[24] Jiale Lao and Immanuel Trummer. 2026. GenDB: The Next Generation of Query Processing – Synthesized, Not Engineered. arXiv:2603.02081 [cs.DB] https://arxiv.org/abs/2603.02081

[25] Jiale Lao and Immanuel Trummer. 2026. SQLBarber: A System Leveraging Large Language Models to Generate Customized and Realistic SQL Workloads. *Proc. ACM Manag. Data* 4, 1, Article 85 (April 2026), 27 pages. https://doi.org/10.1145/3786699

[26] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. *Proc. VLDB Endow.* 17, 7 (March 2024), 1565–1577. https://doi.org/10.14778/3654621.3654625

[27] 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

[28] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2025. Still Asking: How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 18, 12 (Aug. 2025), 5531–5536. https://doi.org/10.14778/3750601.3760521

[29] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *The VLDB Journal* 27, 5 (Oct. 2018), 643–668. https://doi.org/10.1007/s00778-017-0480-7

[30] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The Power of Scale for Parameter-Efficient Prompt Tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 3045–3059. https://doi.org/10.18653/v1/2021.emnlp-main.243

[31] Charles Levine, Jim Gray, Steve Kiss, and Walt Kohler. 1993. The Evolution of TPC Benchmarks: Why TPC-A and TPC-B are Obsolete. In *Open OLTP Report*, Vol. 4. Standish Group, Yarmouth, MA. https://jimgray.azurewebsites.net/papers/TPC_Evolution.pdf

[32] Ryan Marcus, Andreas Kipf, Alexander Van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *PVLDB* 14, 1 (Sept. 2020), 1–13. https://doi.org/10.14778/3421424.3421425

[33] 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* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1275–1288. https://doi.org/10.1145/3448016.3452838

[34] 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

[35] Ryan Marcus, Jeffrey Tao, Peizhi Wu, and Zijie Zhao. 2026. Survivorship Bias in Industrial Database Workloads. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. https://www.vldb.org/cidrdb/papers/2026/p22-marcus.pdf

[36] Natalie T. Maus, Haydn T. Jones, Juston S. Moore, Matt J. Kusner, John Bradshaw, and Jacob R. Gardner. 2022. 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.

[37] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *PVLDB* 2, 1 (2009), 982–993. https://doi.org/10.14778/1687627.1687738

[38] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 1049–1058.

[39] 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 (July 2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[40] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Scottsdale, Arizona, USA) *(PODS '12)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/2213556.2213565

[41] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

[42] Daniel J. Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. 2018. A Tutorial on Thompson Sampling. *Found. Trends Mach. Learn.* 11, 1 (July 2018), 1–96. https://doi.org/10.1561/2200000070

[43] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proc. VLDB Endow.* 18, 11 (July 2025), 4144–4157. https://doi.org/10.14778/3749646.3749683

[44] 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

[45] Mihail Stoian, Andreas Zimmerer, Skander Krid, Amadou Latyr Ngom, Jialin Ding, Tim Kraska, and Andreas Kipf. 2025. Parachute: Single-Pass Bi-Directional Information Passing. *Proc. VLDB Endow.* 18, 10 (June 2025), 3299–3311. https://doi.org/10.14778/3748191.3748196

[46] Jeffrey Tao, Natalie Maus, Haydn Jones, Yimeng Zeng, Jacob R. Gardner, and Ryan Marcus. 2025. Learned Offline Query Planning via Bayesian Optimization. *Proceedings of the ACM on Management of Data* 3, 3 (June 2025), 1–29. https://doi.org/10.1145/3725316

[47] Transaction Processing Performance Council. 2022. *TPC Benchmark H (TPC-H) Standard Specification.* Technical Report. Transaction Processing Performance Council. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf Revision 3.0.1.

[48] Transaction Processing Performance Council. 2024. *TPC Benchmark DS (TPC-DS) Standard Specification.* Technical Report. Transaction Processing Performance Council. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v4.0.0.pdf Version 4.0.0.

[49] 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

[50] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[51] Alexander Van Renen, Dominik Horn, Pascal Pfeil, Kapil 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* 17, 11 (July 2024), 3694–3706. https://doi.org/10.14778/3681954.3682031

[52] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1413–1425. https://doi.org/10.14778/3583140.3583156

[53] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees. *Proc. ACM Manag. Data* 3, 3, Article 235 (June 2025), 28 pages. https://doi.org/10.1145/3725423

[54] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are we ready for learned cardinality estimation? *Proc. VLDB Endow.* 14, 9 (May 2021), 1640–1654. https://doi.org/10.14778/3461535.3461552

[55] Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proc. ACM Manag. Data* 1, 2, Article 150 (June 2023), 23 pages. https://doi.org/10.1145/3589295

[56] Johannes Wehrstein, Timo Eckmann, Roman Heinrich, and Carsten Binnig. 2025. JOB-Complex: A Challenging Benchmark for Traditional & Learned Query Optimization. arXiv:2507.07471 [cs.DB] https://arxiv.org/abs/2507.07471

[57] Johannes Wehrstein, Timo Eckmann, Matthias Jasny, and Carsten Binnig. 2026. Bespoke OLAP: Synthesizing Workload-Specific One-size-fits-one Database Engines. arXiv:2603.02001 [cs.DB] https://arxiv.org/abs/2603.02001

[58] 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

[59] 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

[60] 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.

[61] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7* (Cannes, France) *(VLDB '81)*. VLDB Endowment, 82–94.

[62] 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

[63] Haozhe Zhang, Christoph Mayer, Mahmoud Abo Khamis, Dan Olteanu, and Dan Suciu. 2025. LpBound: Pessimistic Cardinality Estimation Using p-Norms of Degree Sequences. *Proc. ACM Manag. Data* 3, 3, Article 184 (June 2025), 27 pages. https://doi.org/10.1145/3725321

[64] Yunjia Zhang, Yannis Chronis, Jignesh M. Patel, and Theodoros Rekatsinas. 2023. Simple Adaptive Query Processing vs. Learned Query Optimizers: Observations and Analysis. *Proc. VLDB Endow.* 16, 11 (July 2023), 2962–2975. https://doi.org/10.14778/3611479.3611501

[65] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. *Proc. ACM Manag. Data* 3, 3, Article 146 (June 2025), 28 pages. https://doi.org/10.1145/3725283

[66] 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