AutoSteer: Learned Query Optimization for Any SQL Database

Christoph Anneser[1]
Technical University of Munich
anneser@in.tum.de

Nesime Tatbul
Intel Labs and MIT
tatbul@csail.mit.edu

David Cohen
Intel
david.e.cohen@intel.com

Zhenggang Xu
Meta
zhenggang@fb.com

Prithviraj Pandian
Meta
prithvip@fb.com

Nikolay Laptev
Meta
nlaptev@fb.com

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

ABSTRACT

This paper presents AutoSteer, a learning-based solution that automatically drives query optimization in any SQL database that exposes tunable optimizer knobs. AutoSteer builds on the Bandit optimizer (Bao) and extends it with new capabilities (e.g., automated hint-set discovery) to minimize integration effort and facilitate usability in both monolithic and disaggregated SQL systems. We successfully applied AutoSteer on PostgreSQL, PrestoDB, SparkSQL, MySQL, and DuckDB – five popular open-source database engines with diverse query optimizers. We then conducted a detailed experimental evaluation with public benchmarks (JOB, Stackoverflow, TPC-DS) and a production workload from Meta's PrestoDB deployments. Our evaluation shows that AutoSteer can not only outperform these engines' native query optimizers (e.g., up to 40% improvements for PrestoDB) but can also match the performance of Bao-for-PostgreSQL with reduced human supervision and increased adaptivity, as it replaces Bao's static, expert-picked hint-sets with those that are automatically discovered. We also provide an open-source implementation of AutoSteer together with a visual tool for interactive use by query optimization experts.

PVLDB Reference Format:
Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. AutoSteer: Learned Query Optimization for Any SQL Database. PVLDB, 16(12): 3515 - 3527, 2023.
doi:10.14778/3611540.3611544

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/IntelLabs/Auto-Steer.

1 INTRODUCTION

Our research community has been making rapid strides in applying modern machine learning (ML) techniques to tackle longstanding problems in databases [6, 24, 48]. Learned query optimization lies at the forefront of this progress [51]. Various techniques from query-driven and data-driven to their combinations have been proposed [19, 20, 23] – not only to improve core query optimization tasks such as cardinality estimation [22, 23, 31, 32, 37, 39, 43], join order enumeration [29], or query rewriting [50], but also to build end-to-end query optimizers replacing [28, 42] or enhancing [27, 30, 44, 47] traditional ones. The practicality and robustness of these techniques are critical when applying them in industrial settings [47].

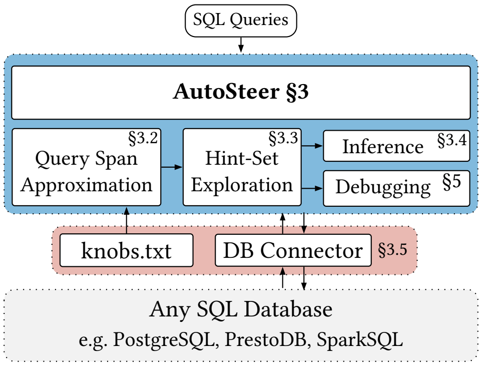


Figure 1: AutoSteer is a framework for steering query optimizers of SQL databases autonomously. For each query, we search for effective rewrite rules and store them in the query span. Then, we use a greedy algorithm to explore alternative query plans efficiently. The results can be used to train predictive models or to debug existing query optimizers.

The so-called "steering approach" of Bao (Bandit optimizer) has been a successful example of a practical solution due to its emphasis on shortening training times, adaptivity to dynamic workloads, and ability to integrate with traditional optimizers [27]. Given a pre-determined collection of "*hint-sets*" (a hint-set indicates which query rewrite rules (RRs) should be considered in query optimization), Bao learns to steer an already existing query optimizer by helping it choose the right hint-set to use for every incoming query. This way, potential planning mistakes of traditional query optimizers can be avoided. As Bao's initial success continues to drive wider adoption in increasingly more sophisticated deployment and workload settings [3, 47], it also brings new challenges to the surface. We tackle two such challenges in this paper:

Integration effort: Adopting Bao to a new database system requires coming up with the right collection of hint-sets. In the original approach developed for PostgreSQL [1], a static collection of 48 hint-sets is manually selected based on deep knowledge of the underlying PostgreSQL optimizer [5], after which Bao independently learns to choose among these hint-sets on a per-query basis. Unfortunately, manually engineering feature hint-sets can be quite

[1] Work done while at Intel.

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611544

challenging, as first noted by Negi et al. [30]. It is especially hard to hand-select hint-sets for those systems with a high number of possible hints to explore (e.g., Microsoft's query engine SCOPE has 256 rewrite rules, leading to 2^{256} possibilities to consider [30]). While it is possible to handcraft effective hint-sets for almost every database system [7–10], the knobs available in a particular system are not only different from all the other systems but are also implemented at different granularities. For example, PostgreSQL, PrestoDB, and MySQL expose these knobs at a session level [7–9], while SQLServer extends the structured query language to embed the knobs directly within the queries [10]. Therefore, integrating Bao into a new system requires deep insights and a solid understanding of the query optimizer to determine a promising collection of hint-sets. This impedes a generic application of Bao to new database systems and their optimizers. Instead, we need a more systematic approach that, given a SQL database, automates the hint-set selection process as well as minimizing the overall expert knowledge and manual engineering effort involved in integrating Bao.

Use in disaggregated settings: Database systems have been evolving away from their traditional monolithic architectures (e.g., federated, connector-based, coordinator/worker-style, data lake, and lakehouse querying systems [4, 13, 26, 34, 35, 40, 45, 49]). In such disaggregated settings with loosely-coupled database components, query optimizers must operate in complex and dynamic environments, often with limited access to accurate statistics and metadata [16, 21, 26]. Therefore, they often lack reliable cost models and rely on rules or heuristics for optimization. As such, an ML-based approach that can be easily integrated and automatically self-adapts could bring significant improvements to query performance [2, 41].

In this paper, we present *AutoSteer*, a new "plug-and-play" query optimization solution that builds on and extends Bao with new capabilities so it can be easily integrated and used with any SQL database system that exposes tunable optimization knobs. As illustrated in Figure 1, given a list of knobs as input (*knobs.txt*) and interacting with the database through SQL and explain statements (*DB Connector*), our solution uses a greedy algorithm to systematically explore promising hint-sets. This approach takes advantage of the notion of "query spans" [30] together with the compositional structure of advantageous hint combinations. This is in contrast to manually generated static hint-sets [27] and other previous approaches that rely on leveraging cost models or random sampling [30, 47]. Furthermore, AutoSteer generates hint-sets dynamically on a per-query basis, which maximizes workload adaptivity.

Furthermore, AutoSteer also provides an interactive usage mode to support human database experts in debugging and improving existing optimizers. For example, AutoSteer automatically discovers new hint-sets, generates alternative query plans, and evaluates the performance of the generated plans. This approach can assist query optimizer experts in gaining a deeper understanding of the cases where specific rewrite rules have a negative impact.

Overall, our extensions to Bao significantly expand the practical applicability of steering optimizers [27, 30, 47]. We provide experimental evidence from AutoSteer's use in five open-source databases. We further tested our solution using a real PrestoDB workload deployed at Meta, showing that it can also be effectively used in large-scale industrial settings.

Contributions. The key contributions of this paper include:

- We introduce AutoSteer, a practical learning-based framework to steer existing Cascades-style query optimizers.
- AutoSteer builds on and extends Bao with a novel hint-set discovery approach, which helps to generalize the technology behind Bao for a wide variety of SQL systems.
- We provide an open-source prototype implementation of AutoSteer[2] together with an interactive tool[3] to help human experts better understand AutoSteer's results. Our prototype has "plug and play" support for PrestoDB, PostgreSQL, SparkSQL, MySQL, and DuckDB and is extensible to other SQL engines.
- We demonstrate AutoSteer's practicality, generality, and effectiveness in query performance improvement via an extensive experimental evaluation based on several public benchmarks and a production workload tested inside Meta.
- Based on the experience gained along the course of this project, we share a few key insights which we believe can inform future work in query optimizer development.

2 RELATED WORK

Traditional Query Optimization. Our work primarily focuses on improving query optimization in SQL databases with traditional, Cascades-style query optimizers. First proposed in the 1990s [17], Cascades is an extensible query optimization framework that has been widely used in many industrial-scale as well as open-source database systems (e.g., PrestoDB [35], SCOPE [49], SparkSQL/Catalyst [13], Greenplum/Orca [36], Apache Calcite [14]). The Cascades framework follows a unified approach to logical/physical query planning by supporting both rule- and cost-based optimization, which is achieved by a set of transformation (logical) and implementation (physical) rules that are applied to the query plan. While rewrite rules drive logical planning, physical planning requires a reliable model to estimate the costs of query plan alternatives. Cost models, in turn, rely on the availability of accurate, up-to-date statistics and cardinality estimates [18]. Since mistakes happen, most industrial systems provide various workarounds to minimize the impact of such mistakes in their production deployments. For example, most of these systems support query hinting mechanisms as a tool to guide the optimizer's choices in exploring the plan search space more effectively [7–10, 15, 33]. Furthermore, in big data systems with federated architectures, such as PrestoDB [35], SCOPE [49], or SparkSQL [13], rule-based optimization is more heavily used in lack of the required statistics and cost models, which are harder to maintain in their larger scale and more heterogeneous production environments [16, 26].

Learned Query Optimization. Unsolved challenges of traditional query optimization have been investigated by several novel approaches that leverage recent advances in ML [51]. While there are too many to enumerate here [6], we believe it is sufficient to give a few representative examples. ML has been applied to improve both key components of a query optimizer such as the cardinality estimator [22, 23, 31, 32, 37, 39, 43] and the query planner [29, 44, 50], as well as the query optimizer itself as a whole [28, 42]. As an example of the use of ML in query planning, the LearnedRewrite approach

[2] https://github.com/IntelLabs/Auto-Steer

[3] https://github.com/christophanneser/QO-Insight

recently proposed by Zhou et al. uses Monte Carlo tree search to find better orders in which rewrite rules should be applied, reducing both query optimization and execution time in PostgreSQL's query optimizer [50]. As another example, HybridQO proposed by Yu et al. can produce better join orders by combining cost- and learning-based optimization and leveraging an optimizer's hint functionality in candidate plan generation [44]. In contrast, end-to-end learned query optimizers such as Neo [28] and Balsa [42] are designed as more performant drop-in replacements for traditional ones. While Neo bootstraps itself by learning from an expert optimizer (e.g., PostgreSQL's query optimizer), Balsa leverages a simulation-based approach. Both approaches have been shown to outperform open-source and commercial query optimizers, but only under certain workload assumptions (e.g., static datasets and schemas) and at the expense of long training times, which prohibits their frequent retraining. This motivated the more practical approach taken in Bao [27], which aims at steering traditional optimizers towards making better plan choices instead of entirely replacing them.

Steering Query Optimizers with Bao. The **Ba**ndit **o**ptimizer (Bao) learns to assist an already existing optimizer by providing it with *hints*, indicating which rewrite rules (RRs) should be turned off during query optimization [27]. Providing hints to a database system does not involve intrusive changes, as most database systems already expose optimizer knobs or flags that can be configured. Database experts and administrators can use these knobs to enable or disable specific RRs.

Bao was first applied to PostgreSQL, where it leveraged $n = 48$ different hint-sets to generate n (not necessarily different) query execution plans (QEP). Each hint-set disables a subset of the rewrite rules and can be seen as a *simpler* version of the default PostgreSQL query optimizer. In the second step, a tree convolutional neural network (TCNN) predicts the cost (e.g., the query latency or the CPU time) of each QEP. Based on the generated plan alternatives and their predicted costs, Bao decides which QEP should be executed. Instead of always choosing the plan with the best-predicted performance, Bao uses Thompson sampling to balance the exploration of new, alternative QEPs and the exploitation of plans already known to be efficient. Next, PostgreSQL executes the selected plan and records the execution time. Finally, both plan and execution times are added to Bao's experience, which is used to periodically re-train the model. After PostgreSQL, Bao has also been successfully applied to several commercial and open-source database systems, including Vertica, Microsoft Azure Synapse (SQL Server), and Amazon RedShift [3].

To assess Bao's industrial promise, Negi et al. explored how to apply Bao at the scale of Microsoft's SCOPE workloads [30]. SCOPE is Microsoft's internal query processing system for big data workloads, which primarily uses a rule-based query optimizer, though it can also support cost-based optimization through its cost model [49]. Negi et al.'s work on "*Bao-for-SCOPE*" introduced a number of key concepts which we also leverage in our work: *rule categories*, *rule configurations*, *rule signatures*, and *job/query spans*. There are four rule categories: required, off-by-default, on-by-default, and implementation. While required rules must be turned on to generate valid query plans, only rules from the other categories can be turned off to generate new query plans. A rule configuration is a bit vector specifying which rules are turned on and off during query optimization. Rule signatures track which rules have effectively contributed to the final query plan during the optimization. In addition to the rule signature, a job/query span contains only non-required rules. Based on these definitions, Negi et al. introduced a randomized configuration search to generate M rule configurations that produce possibly yet unknown QEPs.

In follow-up work, Zhang et al. built QO-Advisor to prepare Bao-for-SCOPE for actual production deployments at Microsoft [47]. This required adding support to deal with various operational challenges, such as the steering overhead, unexpected performance regressions, and the need for debugging. To reduce steering overhead, expensive tasks such as query span approximation and alternative plan exploration are done offline, as well as utilizing the cost model provided by SCOPE where possible.

All previous adoptions of Bao described above were done as custom integrations, each time targeting a particular system considering its specific architecture and workloads. Our approach fundamentally differs from these due to its *focus on generality*, i.e., making Bao more easily applicable in any SQL database system. The key enabler for this has been our *automated hint-set discovery approach*, which not only removes the need for manually designing system-specific hint-sets, but also makes them more flexible to use under changing workload conditions. Unlike previous Bao extensions [30, 47], AutoSteer is publicly available[2] to enable use across a wide range of SQL systems.

3 AUTOSTEER

In this paper, we present AutoSteer – a practical framework for adding Bao-style, steering-based learned query optimization capability to any SQL database that: (i) has a Cascades-style, rule-based query optimizer and (ii) exposes binary knobs to configure its rules.

Given a database system (DBMS) with an existing rule-based query optimizer, we aim to find semantically equivalent query plan alternatives that execute faster than the default plan generated by that system's native query optimizer. We follow the same general learned query optimization framework as in Bao [27]: Several hand-selected static hint-sets define which rewrite rules of the DBMS are turned on and off during optimization. Bao then leverages these different hint-sets to generate alternative query plans and picks the cheapest plan for execution using its TCNN-based neural prediction model. This approach is shown to be effective in finding query plans that are better than the ones that the underlying query optimizer can find, but there are two fundamental limitations:

(1) Database experts must manually identify a good collection of static hint-sets from scratch for each system Bao is to be integrated. Such an approach requires a deep understanding of that system and its optimizer.
(2) Scaling the number of hint-sets comes at the cost of additional optimization overhead. Bao always considers the same predetermined collection of hint-sets, since they are chosen in advance and independent of the actual query workloads, whether they impact a query's optimization or not.

In the following, we introduce AutoSteer as a new approach that overcomes these limitations. AutoSteer's key focus is on practicality. It builds on and extends Bao to make it more easily and adaptively applicable in SQL databases, no matter how simple or sophisticated

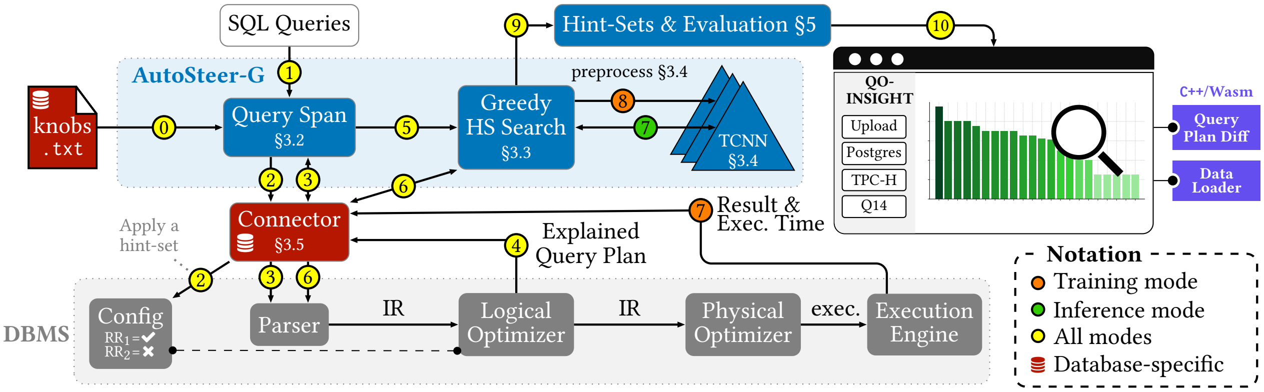


Figure 2: This figure illustrates AutoSteer-G, which communicates with the DBMS through a connector that is completely external to the DBMS. DBMS components are gray, and AutoSteer's components are blue. AutoSteer has two different execution modes: (1) generate training data and build the learned model (Training), (2) optimize queries at run time using the model (Inference). AutoSteer's results can also be interactively explored in QO-Insight [12] for debugging and analysis.

their optimizers are. As such, it generalizes recent industrial efforts on integrating Bao to specific systems [3, 30, 47], thereby facilitating broader adoption of this novel technology in a larger number and variety of DBMSs in the ecosystem. To maximize practicality, AutoSteer has been designed to support several usage options in terms of its: (i) DBMS integration level (custom vs. generic), (ii) execution mode (training vs. inference), and (iii) interaction mode (steering vs. debugging). We will describe these options in detail as part of the following subsections.

3.1 Architectural Overview

In Figure 3, we sketch two alternative ways of integrating AutoSteer into an existing DBMS: **(1) AutoSteer-Generic** leverages an *external connector* whose communication is purely based on SQL and explain statements; **(2) AutoSteer-Custom** implements a *connector* which is directly integrated into the DBMS optimizer. In this subsection, we assume AutoSteer-G, and provide further details on these two integration options in Section 3.5.

In Figure 2, we illustrate a typical SQL query optimization pipeline in a DBMS and show our AutoSteer-G solution in action. First, we provide a text file containing the knobs exposed by the optimizer to AutoSteer ⓪. Based on these knobs, AutoSteer will automatically explore and discover hint-sets without additional user input. Furthermore, instead of sending them to the DBMS, users and applications submit all queries to AutoSteer ①.

For each query, AutoSteer approximates a 'query span' by turning off rewrite rules (RR) systematically ②. Query spans track those RRs that *actually rewrite* the query plan. E.g., when such RRs are turned off, the optimizer would generate an alternative plan [30] (cf. to Section 3.2 for more details). As we are assuming AutoSteer-G in Figure 2, we use an external connector to approximate query spans: First, it configures the DBMS's RRs through its exposed knobs and then lets the DBMS explain the query plan (③ & ④). We call the RRs *effective* if the plan changes and add them to the query span.

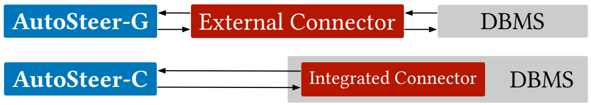


Figure 3: Integration Options: AutoSteer-G and -C.

Based on the query span ⑤, AutoSteer searches for alternative query plans using a greedy hint-set exploration strategy, which is explained in more detail in Section 3.3. For a query span with n effective RRs, the algorithm first creates n hint-sets, with each hint-set disabling *one* of the query span's RRs. In AutoSteer-G, as a next step, we send the query and *each* hint-set to the external connector ⑥ and let the DBMS explain the resulting plans ④.

For the following steps, we differentiate two execution modes: **(1) Training mode:** In the training mode, we *execute* the query plans from step ⑥ and track their execution times ⑦. Then, the aforementioned greedy search explores the search space of the beneficial hint-sets[4] by iteratively combining smaller beneficial hint-sets to create larger hint-sets, which might be beneficial as well. Later, we leverage the query execution plans (QEPs) and their run times to train a tree convolutional deep neural network ⑧.
(2) Inference mode: In contrast to the training mode, where QEPs have been *executed* to find out their *actual* run times, in the inference mode, we leverage the pre-trained tree convolutional neural network (TCNN)[5] to *predict* plan execution times ⑦. As before, when a query is submitted, we use the greedy search to find promising hint-sets more efficiently by pruning those hint-sets expected to perform poorly. Once the greedy search finishes, we sort all explored hint-sets by their predicted execution times and use contextual bandit to pick one hint-set to steer the query.

Finally, AutoSteer supports two user interaction modes:
(1) Steering mode: AutoSteer steers query execution at run time and uses the pre-trained TCNN to predict the execution time of the hint-sets (steps ① – ⑦, as described above).

[4] We call a hint-set beneficial iff it reduces the execution time wrt. the default plan.
[5] The DBMS could also be leveraged here if it provides a reliable cost model. In our experiments with PostgreSQL, however, we observed that the learned model leads to choosing better hint-sets and QEPs than the cost model.

(2) Debugging mode: AutoSteer exports the generated and evaluated hint-sets alongside the queries ⑨. These results can then be interactively explored in QO-Insight [12] ⑩.

3.2 Query Spans

In Bao, hint-sets define which rewrite rules (RRs) are turned on and off, and they are used to generate alternative query plans [27]. We cannot consider all the exponentially many hint-sets as database systems usually implement several tens to hundreds of RRs. However, creating a fixed number of *valuable* hint-sets, as suggested in [27], limits the search space and requires a deep understanding of the system's query optimizer and the workloads. Furthermore, Bao considers the same hint-sets for all queries regardless of whether the hint-sets turn off rules impacting the plan.

Instead, Negi et al. consider *effective* rules only (rules that *actually* rewrite the plan) and therefore introduce the concept of query spans [30]. A query span belongs to exactly one query and contains all the *non-required* rules that can *potentially* modify the query plan during its optimization. A rule r is non-required if the optimizer can generate a valid query plan without r. Calculating the *true* query span is challenging, as rules might have unknown dependencies on other rules. For example, turning off a set of rules could result in a different intermediate query plan that causes other alternative rules to become active.

Batch Approximation. Negi et al. [30] use a heuristics-based approximation of query spans instead. They leverage the SCOPE system for their work, whose query optimizer already tracks the *effective* RRs. Then, they turn off *all* effective RRs in one *batch*, and the process repeats until it does not detect other alternative rules.

Iterative Approximation. Alternatively, we use a more fine-grained, *iterative* approach: We iteratively turn off *one* effective rule (and its dependencies) at a time and check if other rules become effective. While this approach requires the query optimizer to run more often, it tracks rule dependencies more accurately. Later, we can utilize these dependencies during the exploration of hint-sets.

AutoSteer's *integration level* also impacts the query span approximation and the detected rules. When a connector is directly integrated into the query optimizer, it can track all rules programmatically in *one pass*. However, an external connector will not detect those RRs that change the query plan at an algorithmic level because such changes are usually not included in the explained query plan. Of course, the *more effective RRs* AutoSteer finds the *more* and potentially better *hint-sets* it can generate later. We evaluate the impact of the integration level for PrestoDB in Section 4.4.

While PrestoDB's optimizer implements 170 RRs, our experiments with AutoSteer-C and the iterative query span approximation for the 137 JOB queries show that only a few rules ($\leq$ 20) effectively contribute to the query plans. This observation reduces the theoretical search space of hint-sets from 2^{170} to 2^{20}. As 2^{20} configurations are still too many to explore, we propose a greedy exploration approach, as described next.

3.3 Dynamic Exploration of Hint-Sets

For a given SQL query, AutoSteer's goal is to find the most beneficial hint-sets that steer the optimizer toward better query plans. Although there are 2^n potential hint-sets for a query span with n non-required effective RRs, in our experiments in Section 4 with several different systems and workloads, we observed that only a few hint-sets are *beneficial* in practice. AutoSteer aims to find those few beneficial hint-sets as efficiently as possible.

Negi et al. propose an algorithm that *randomly* generates M hint-sets first and then filters for the most promising query plans according to SCOPE's cost model [30]. However, this approach requires an accurate cost model. Otherwise, we must *execute* the plans to determine whether they are beneficial. Furthermore, our experimental findings in Section 4.5 *suggest*:

(1) Most beneficial hint-sets *consist of smaller, beneficial hint-sets.*
(2) Most beneficial hint-sets are *small* (fewer than four knobs).

Based on these two empirical observations, we introduce a more structured and efficient way to explore the hint-sets. Our proposed algorithm, outlined in pseudocode in Listing 1, utilizes a greedy approach consisting of three building blocks:

🟩 First, we use the empty hint-set **{}** to execute the *default plan*, serving us as a baseline (i.e., the native optimizer's optimized plan) in line 4. In `results`, we map the hint-sets to their resulting query plans and execution times. The function `exec` executes the given query and hint-set, and returns the query plan and the execution time. When `infer=true`, AutoSteer does not execute the plan but instead uses a pre-trained model to predict its run time.

🟦 In the second block starting in line 7, we leverage the query span's effective RRs to generate *singleton* hint-sets, which turn off *exactly one* rule. Then, we let the DBMS execute these and track their resulting query plans and execution times in lines 11 and 12 if they perform better than the default plan. Please note that the greedy search is easily extendable. E.g., we could consider only those hint-sets whose improvements exceed a certain threshold.

```
1  def explore_hint_sets(query, query_span, infer):
```

Baseline

```
2    results = dict() # Hint-set → (QP, exec. time)
3    # 1. Execute baseline ({} is the empty hint-set)
4    results[{}] = exec(query, {}, infer)
5    singleton_hint_sets = []
```

Gen. Singletons

```
6    # 2. Run query with one rule turned off at a time
7    for rule in query_span.effective_rules:
8      QP, exec_time = exec(query, {rule}, infer)
9      # Keep track of beneficial hint-sets only
10     results[{rule}] = {QP, exec_time}
11     if exec_time < results[{}].exec_time: # Beneficial?
12       singleton_hint_sets.push({rule})
```

Greedy Exploration

```
13   # 3. Run a bottom-up greedy search
14   hint_sets = copy(singleton_hint_sets)
15   while not hint_sets.is_empty():
16     hs = hint_sets.pop()
17     # Generate larger hint-sets
18     combined_hs = combine(hs, singleton_hint_sets)
19     for new_hs in combined_hs:
20       QP, exec_time = exec(query, new_hs, infer)
21       results[new_hs] = {QP, exec_time}
22       if exec_time < results[{}].exec_time: # Beneficial?
23         hint_sets.push(new_hs)
24   return results
```

Listing 1: Pseudocode of AutoSteer's greedy hint-set search.

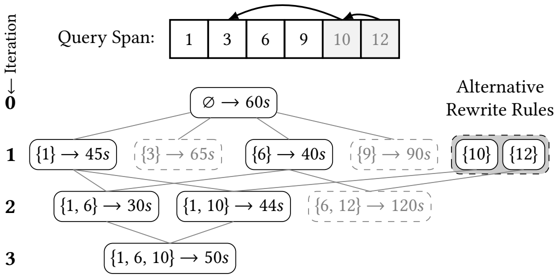


Figure 4: Example for AutoSteer's greedy exploration.

🟥 Third, the bottom-up greedy hint-set exploration loops over the previously seen beneficial hint-sets (line 15). We extract one hint-set at a time from the queue of beneficial hint-sets (line 16) and generate all other combinations with other singleton hint-sets in line 18. Note that we do not show the handling of alternative rules and tracking of the best-performing hint-sets (which is necessary for the inference mode) due to space limitations.

Figure 4 visualizes the algorithm for an example query span with effective RRs 1, 3, 6, and 9, and alternative RRs 10 and 12. Outgoing arrows denote rule dependencies of alternative rules. E.g., if rule c has been identified as an alternative to rule a, there is an edge $c \longrightarrow a$. We use the empty hint-set in the first iteration to execute the default plan. We then generate alternative plans using the hint-sets 1, 3, 6, and 9 and track their execution times. Hint-sets resulting in query plans with execution times exceeding the default plan execution time (3 and 9) are discarded and not considered in subsequent iterations. We also consider alternative rules in the following iterations while generating larger hint-sets.

3.4 Inference Mode using TCNNs

When AutoSteer executes in its inference mode (cf. step ⑦), it uses a learned model to make predictions about plan execution times. We borrow this part from Bao which uses a tree convolutional neural network (TCNN) for its predictive model [27]. Before query plans can be used with TCNNs, we must preprocess and featurize them. However, this step will slightly differ between DBMSs as databases have their custom query plan formats and operator types (e.g., PostgreSQL supports index scans, but PrestoDB does not). In addition to the preprocessing of PostgreSQL plans in [27], we implemented the preprocessing of PrestoDB's query plans and made the code publicly available.[2] We use, however, the same configuration for training as in [27]. We refer the reader to [28] for a deeper investigation into tree convolution applied to query plans.

3.5 Generic vs. Custom Integration

Generic Integration. As was already illustrated in Section 3.1, AutoSteer-G leverages an *external connector* whose communication is purely based on SQL and explain statements. This option is appealing due to its low programming effort. We thus far implemented external connectors for PostgreSQL, PrestoDB, SparkSQL, MySQL, and DuckDB, in less than 100 lines of code each. We show an example implementation of such an external database connector for PostgreSQL and PrestoDB in Listing 2. Since databases have their own custom APIs for exposing knobs or explaining query plans,

```
class PostgreSQLConnector(ExternalDBConnector):
  def __init__(url: str):
    self.conn = ... # setup PostgreSQL connection
  def set_knob(knob: str, enable: bool) -> void:
    self.conn.exec(f"SET {knob} TO \
        {'ON' if enable else 'OFF'}")
  def explain(query: str) -> dict:
    return self.execute(f'EXPLAIN {query}')
class PrestoDBConnector(ExternalDBConnector):
  def __init__(url: str):
    self.conn = ... # setup PrestoDB connection
  def set_knob(knob: str, enable: bool) -> void:
    self.conn.exec(f'SET SESSION {knob} = {enable}')
  def explain(query: str) -> dict:
    return self.execute(f'EXPLAIN JSON {query}')
```

Listing 2: External connectors for PostgreSQL and PrestoDB.

```
1  class QueryOptimizer:
2    def optimize(root_node) -> QuerySpan:
3      query_span = QuerySpan()
4      for rewrite_rule in self.rewrite_rules:
5        rewrite_rule.apply(root_node, query_span)
6      return query_span
7  class RewriteRule:
8    def apply(node, query_span):
9      if self.condition(node):
10       query_span.add(rule_id)
11       rewrite(node)
12     for child in node.child_nodes:
13       apply(child, query_span)
```

Listing 3: Tracking query spans in AutoSteer-C for PrestoDB.

these connectors implement functions to toggle knobs, explaining, and executing queries. As can be seen in this example, our external connectors only slightly vary in syntax from one system to another.

Custom Integration. As an alternative, AutoSteer-C's connector is directly integrated into the database's optimizer (i.e., similar to previous DBMS-specific applications of Bao [3, 30, 47]). While the implementation of an integrated connector involves more programming effort and requires a deeper understanding of the DBMS's optimizer, it can also make AutoSteer more efficient to execute. For example, AutoSteer-C would allow tracking effective RRs in a *single pass* and find RRs that cannot be detected by comparing the explained query plans. In contrast, AutoSteer-G would run *multiple* explain statements in the number of exposed knobs. Furthermore, AutoSteer-C reduces the run time overhead of optimizing queries in the inference mode by more efficiently interacting with the DBMS.

Listing 3 sketches how PrestoDB's query optimizer can be extended for AutoSteer-C to track query spans during optimization directly. First, PrestoDB parses the SQL statement into a logical query plan and then invokes the query optimizer on the root node in line 2. The query optimizer sequentially executes the RRs' apply function and passes references of the root node and the query span in lines 4 and 5. The modified RR directly adds itself to the query span once its conditions is fulfilled and it is applied to the query plan. Then, similar to explain statements, we would extend SQL's grammar to run the in-database query span approximation. Consequently, the custom integration avoids AutoSteer-G's overhead of running multiple explain statements.

Table 1: Benchmarks and workloads.

Benchmark	Dataset Size	Number of Queries
JOB [25]	7.2 GB	137
Stack [27]	100 GB	100
TPC-DS [38]	1/10/100 GB	100
Meta	>1 PB	>3000

Table 2: List of experiments.

Section	Experiment	Workload	Setup
§4.2	AutoSteer-C for PrestoDB	JOB, Stack	1
§4.3	AutoSteer-C for PrestoDB	Meta	2
§4.4	AutoSteer-C vs. -G for PrestoDB	JOB	1
§4.5	AutoSteer-G vs. Bao for PostgreSQL	JOB	3
§4.6	AutoSteer-G for SparkSQL	TPC-DS	4
§4.7	AutoSteer Coding Effort	N/A	N/A

We implemented both an external (see Listing 2) and an integrated connector for PrestoDB, and provide an empirical comparison in Section 4.4 and a more general discussion on coding effort in Section 4.7. In general, both of these integration options will be useful in practice. For example, we envision AutoSteer-G to be used for rapid proof-of-concept prototyping to show the feasibility and to approximate potential performance gains on a DBMS, after which AutoSteer-C is implemented for use in production deployments where its run time efficiency would matter more.

4 EVALUATION

To evaluate AutoSteer, we applied it to five different SQL databases: PrestoDB, PostgreSQL, SparkSQL, MySQL, and DuckDB. We report our experimental findings with the former three in this section and provide a summary of our experience with the latter two as part of Section 5. The high-level goals of our experimental study are:

- Show AutoSteer's generality and practicality by testing its effectiveness on a variety of open-source systems and benchmarks commonly used by database researchers and practitioners.
- Validate AutoSteer's effectiveness when applied to real-world workloads from large-scale deployments in industrial settings.
- Evaluate how AutoSteer's automatically generated hint-sets fare against the expert-selected hint-sets of original Bao and the randomized approach used in its SCOPE adaptation [30].
- Quantify the productivity and performance tradeoffs of using AutoSteer in its custom and generic integration levels.

4.1 Experimental Setup

Benchmarks and Workloads. Table 1 shows the workloads we used in our experiments. Three of these are public benchmarks heavily used by the database and query optimization communities (JOB w/o FK indexes [25], Stack [27], and TPC-DS [38]), and the fourth is a real-world workload from large-scale PrestoDB deployments at Meta. These workloads cover a range of scales regarding dataset sizes (GBs-PBs) and the number of queries (100s-1000s).

Hardware and Software Setups. We used multiple different hardware/software setups for our experiments:

Setup 1: As sketched in Figure 5, we deploy PrestoDB on a 5-node Kubernetes cluster. All nodes have a dual-socket Intel® Xeon® Platinum 8280 CPU with 2×28 cores at 2.7 GHz, 256 GB memory, and an Intel® DC S3500 SSD attached, which stores a copy of the datasets to reduce transfer times between nodes. The compute nodes are connected with 1000 MbE. We run all queries in isolation and with warm caches.

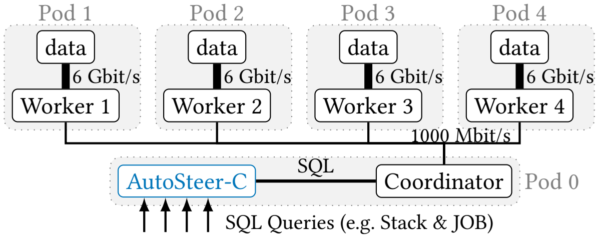


Figure 5: In Setup 1, we run PrestoDB on a 5-pod K8s cluster. AutoSteer intercepts queries and automatically explores alternative plans. Datasets are cached on worker-local SSDs.

Setup 2: This setup corresponds to our real-world workload experiments with PrestoDB conducted at Meta. This involved executing a large interactive dashboarding workload scanning petabytes of data against a large PrestoDB cluster with hundreds of nodes. We tested more than 3000 queries that run every day at Meta.

Setup 3: We run PostgreSQL 13 on a 16-core AMD Ryzen 3950X@3.5 GHz machine with 96GB DDR4-2667 memory. We only execute hint-sets yielding new query plans and use warm caches.

Setup 4: We configured SparkSQL v3.2.2 as it is internally used at Intel and deployed it on a single machine equipped with a dual-socket Intel® Xeon® Platinum 8280 CPU with 2×28 cores at 2.7 GHz and 256 GB memory. All datasets were stored in memory.

To account for runtime variances in Setups 1, 3, and 4, we executed the query plans generated by each hint-set multiple times and compared their median execution times.

Overview of Experiments. Table 2 provides an overview of the conducted experiments together with the experimental workloads and setups used for each. In terms of its usage options, we explicitly state if AutoSteer was used in the custom (*AutoSteer-C*) vs. generic (*AutoSteer-G*) integration level in each of the following subsections. For each experiment, we state whether we used the *Training* or the *Inference* execution mode. We set the interaction mode to *Steering* for all of our experiments.

4.2 AutoSteer-C for PrestoDB

Does AutoSteer-C find better plans than PrestoDB? In Figure 6, we compare AutoSteer-C to PrestoDB. We executed all 137 JOB queries on the PrestoDB cluster (Setup 1), and we show the relative performance changes for a uniform sample. Then, we sort the queries by their relative performance improvements achieved by the best known[6] query plan generated by AutoSteer-C's training mode and plot them in ascending order. Here, we consider only *alternative* plans that differ from the default query plan. Our approach finds a better alternative execution plan for most queries (green bars). For this selection, there are only four queries for which the best known alternative plan performed worse than the default plan generated by PrestoDB (28a, 5c, 25a, 21a). However, those queries have short execution times of ≤ 4 seconds. In contrast, by

[6] The term "best known hint-set" refers to the hint-set that leads to the fastest execution plan among *all plans explored* by either AutoSteer or its competitors.

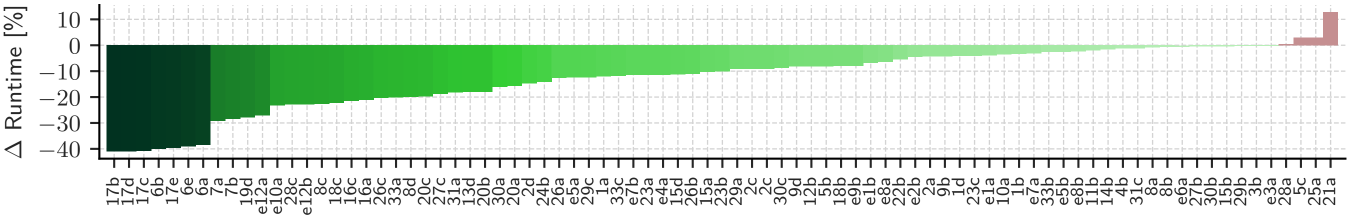


Figure 6: The relative run time changes (lower is better) of the *best known alternative (non-default) query plan* found by AutoSteer-C's training mode compared to PrestoDB's default plan. For space reasons, we consider a uniform sample of the JOB queries executed in Setup 1. We used the greedy algorithm described in Section 3 to explore beneficial hint-sets.

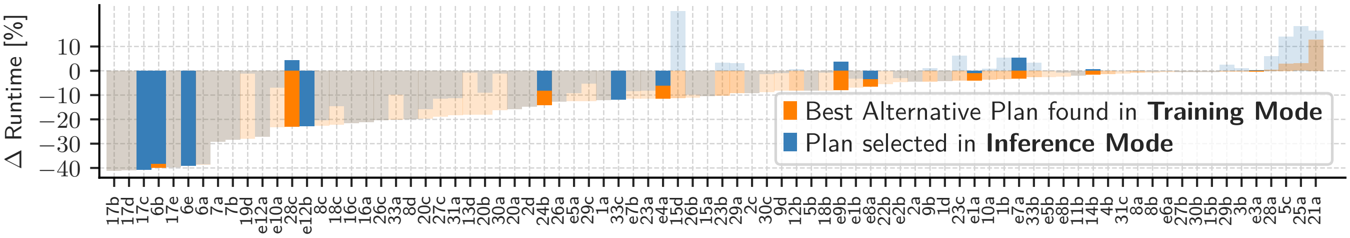


Figure 7: The relative run time savings of the best known plan (🟧) found by AutoSteer-C's training mode compared to PrestoDB. For the inference mode, a pre-trained TCNN predicts plan run times and selects the plan with the best-predicted performance (🟦). Solid colors represent unseen queries from the test set, and transparent colors represent those from the training set.

Table 3: The top-5 hint-sets that yield the largest performance gains wrt. to PrestoDB's default plan. We ran JOB queries in isolation (Setup 1). The last column shows the number of JOB queries for which this hint-set produced the fastest plan.

	Run Time Changes [%]		
Hint-Set	**Average**	**Worst Case**	**# Best HS**
HashGenOptimizer	-30.35%	+12.82	75
HashGenOptimizer, UnaliasSymbolRefs	-38.29%	+0.07	25
PickTabLayoutForPred	-5.75%	+0.03	13
UnaliasSymbolRefs	-8.99%	-0.44	9
PruneTabScanCols	-8.63%	+3.01	8

turning off the *HashGenOptimizer*, the execution time of query 17b decreases by more than 40% (127 − 75 = 52 seconds).

What are the top hint-sets AutoSteer-C generates? Table 3 presents the top-5 hint-sets discovered by AutoSteer-C for PrestoDB. The second column shows the average run time reduction achieved by each hint-set when it generated the fastest query plan. Out of all 137 JOB queries, the third column shows the impact on the performance in the worst case. The last column shows the number of queries for which the hint-set produced the best known plan. The top hint-set disables *HashGenOptimizer* and yields the fastest execution plan for 75 JOB queries. As described in [35], HashGenOptimizer adds local projections to compute hash codes early during execution, increasing the cost of downstream shuffles and filling up buffer memory. Moreover, as our experiments show, these shuffles' overhead will outweigh the parallelism's performance gains. While most JOB queries (75/137) benefit from disabling HashGenOptimizer, a few will regress by up to 12.8%. In Section 5, we discuss how experts could leverage AutoSteer's insights to improve a specific rule conceptually.

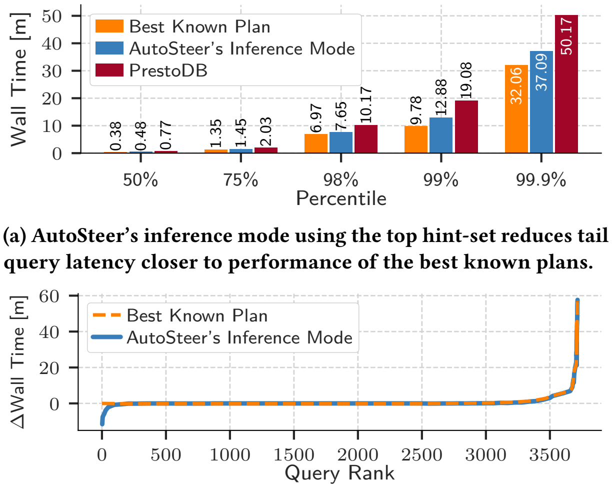
Table 4: AutoSteer's relative and absolute run time improvements compared to PrestoDB's default plans on average using Setup 1. The results belong to the queries from the test set.

	JOB	Stack
Rel. Improv. (Best Known Hint-Set)	30.25%	**42.38%**
Rel. Improv. (Inference Mode)	27.93%	**31.54%**
Abs. Improv. (Best Known Hint-Set)	**305s**	284s
Abs. Improv. (Inference Mode)	**282s**	211s

Can AutoSteer-C's Inference Mode improve PrestoDB? To answer that question, we fit a tree convolutional neural network (TCNN) that we later use to infer the query plans' execution times. First, we use AutoSteer's training mode to explore hint-sets for the JOB and Stack benchmarks in Setup 1. We split the 237 (100 Stack and 137 JOB) queries into training and test sets at an 80/20 ratio. Next, we train the TCNN in a supervised fashion on the training set and choose the same configurations as suggested in [27]. Then, the TCNN predicts the run time for each query plan.

Figure 7 compares the best-performing plans found in training mode to those selected in AutoSteer's inference mode for JOB. The orange bars show the relative improvements of the best known plan wrt. PrestoDBs default query plan. Blue bars show the relative improvements of the plans selected by AutoSteer's inference mode. For most queries, the TCNN chooses a hint-set that improves the execution time compared to PrestoDB's default plan. However, for a few queries, such as e9b and 15d, the TCNN chooses hint-sets that negatively impact the execution time. Overall, the inference mode generalizes well to unseen queries from the test set.

Table 4 considers AutoSteer's overall impact on the JOB and Stack benchmarks in Setup 1. We compare the average relative and absolute performance improvements of the hint-sets found



(a) AutoSteer's inference mode using the top hint-set reduces tail query latency closer to performance of the best known plans.

(b) Distribution of raw query time changes for AutoSteer with the top hint-set. A few queries regress, but tail latency improves.

Figure 8: Experiments with a production workload at Meta.

by AutoSteer-C when choosing either the best hint-sets known from the training mode or the selected hint-sets in the inference mode. For Setup 1 and the Stack queries, the best plans discovered in AutoSteer's training mode reduce the run time by up to 42%. AutoSteer's inference mode reduces the relative run time by up to 31%. The absolute, overall JOB execution time decreases by 305 seconds when selecting the best known hint-set. AutoSteer's inference mode reduces the absolute run time by 282 seconds.

RESULT SUMMARY. *AutoSteer-C can generate better query plans than PrestoDB's native query optimizer in both of its execution modes (Training and Inference), with zero help from a human expert in hint-set selection. It finds "HashGenOptimizer" to be the top hint-set.*

4.3 AutoSteer-C for PrestoDB: Meta Workload

To validate AutoSteer's effectiveness in real-world scenarios, we tested our approach on a large-scale dashboard application deployed at Meta (Setup 2). The dashboard application runs on PrestoDB and executes thousands of queries every day over petabytes of data. Since dashboard views commonly consist of many widgets (queries) and the dashboard is only helpful once a large portion of queries are completed, we focus our analysis on tail latency.

We first run a selection of hint-sets generated by AutoSteer on the workload. To minimize computation time, we leverage the most promising hint-set known from the experiments described in Section 4.2 (i.e., disabling *HashGenOptimizer* as shown in Table 3). First, we run AutoSteer's training mode to generate alternative query plans and track their execution times. Then, we use that training data to fit a TCNN, which is later used by AutoSteer's inference mode. Figure 8a shows the tail latencies achieved from an "optimal oracle" that always chooses the best query plan known from AutoSteer's training mode, best-predicted plan from AutoSteer's inference mode, and the default plan from Meta's production PrestoDB configuration, respectively. With the single hint-set discovered by our approach, we observe a noticeable reduction in the tail latency.

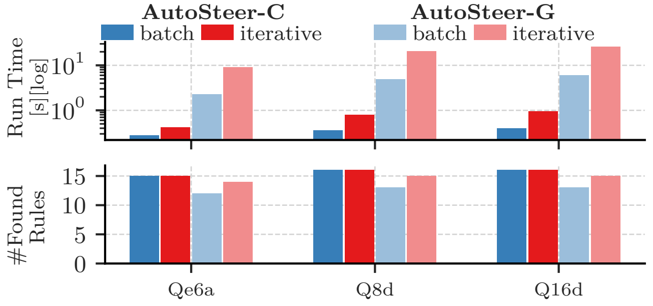


Figure 9: Time elapsed and number of rules found for different query span approximation variants in AutoSteer for PrestoDB. We consider three random queries from JOB [25].

While our predicted approach does not quite reach the performance of the best known plan, it comes close in most cases.

Any change to a query optimizer might result in a regression. We plot the query performance changes for our approach (AutoSteer) and the best known plan in Figure 8b. The "long tail" is significantly decreased by our approach, although a few regressions (in the order of 10 minutes) do occur. Since the slowest queries determine this application's performance, these regressions are negligible, and the overall performance is improved.

RESULT SUMMARY. *AutoSteer can help reduce tail query latency of a PB-scale interactive dashboard application running on a PrestoDB cluster at Meta. Even with one top hint-set discovered by AutoSteer ("HashGenOptimizer"), about 20% reduction in 99% tail latency can be achieved over PrestoDB's native query optimizer.*

4.4 Approximating Query Spans in PrestoDB

In this section, we evaluate the impacts of AutoSteer's integration level (Section 3.5) as well as its two approximation heuristics *batch* and *iterative* (Section 3.2) on query span approximation performance. We measure the performance considering two dimensions: the execution time and the number of effective RRs found.

How long does query span approximation take for the different variants? In the upper part of Figure 9, we compare the execution times of the query span approximations based on AutoSteer-C and AutoSteer-G for PrestoDB on three selected JOB queries. We further differentiate between the *batch* and the *iterative* approximation heuristics. AutoSteer-C, whose connector is *directly integrated* into PrestoDB's optimizer, tracks effective RRs during the optimization phase, which allows AutoSteer-C to achieve better performance and makes it almost an order of magnitude faster than AutoSteer-G's external connector. In contrast, the external connector runs one explain statement for each of the exposed knobs (170).

The *batch* approximation requires the query optimizer to run the fewest iterations, therefore, completes faster than the *iterative* approximation, which additionally tracks dependencies of the alternative RRs. The rule dependencies, however, help reduce the search space in the following hint-set exploration (cf. Section 3.3).

Given the significant overhead of AutoSteer-G using the iterative heuristic, this approach is not suitable for short-running and transactional queries, but it might amortize for long-running and analytical queries. Furthermore, it could help database experts in quickly implementing a first proof-of-concept revealing the potential performance improvements, where the query span approximation time would not be as critical.

How many rules do the different query span approximation variants detect? In the lower part of Figure 9, we plot the number of detected RRs for each approach. The experiments show that AutoSteer-C for PrestoDB finds the same number of RRs independent of the used approximation heuristic. AutoSteer-G detects fewer rules (especially in the batch mode) because some of the RRs change operator details at an algorithmic level which are *not* included in the explained query plan that is used by the generic integration level. For instance, the rules *SetFlatteningOptimizer*, *PickTableLayoutWithoutPredicate*, and *ApplyConnectorOptimization* affect the execution plan for JOB query 8d, but their changes are transparent in the explained plans. When using the iterative heuristic, the degradation in the number of RRs is not as noticeable.

RESULT SUMMARY. *AutoSteer-C with batch heuristic is most efficient in finding the most number of rules during query span approximation. Despite being slower, AutoSteer-G can also find a significant majority of the rules and, as such, can be a suitable option to use for workloads with long-running queries and initial prototyping.*

4.5 AutoSteer-G for PostgreSQL

We use Setup 3 to compare AutoSteer-G to (1) the original Bao-for-PostgreSQL [27] and (2) the randomized hint-set search used in its SCOPE adaptation [30, 47]. We analyze what hint-sets these approaches explore and their impact on query performance.

Does AutoSteer-G find better hints than Bao-for-PostgreSQL? Remember that the key difference between AutoSteer and Bao-for-PostgreSQL is which and how hint-sets are selected. In contrast to Bao's 48 static hint-sets *chosen manually* by an expert, AutoSteer generates them *automatically*. Considering the 137 JOB queries, both approaches found the best known hint-set[6] in 99 cases. AutoSteer-G, however, discovered better hint-sets for 23 queries. In the remaining 15 cases, Bao-for-PostgreSQL found at least one hint-set, which performed better than all hints-sets generated by AutoSteer-G. However, for those queries where AutoSteer-G did not find the best hint-set but Bao-for-PostgreSQL did, we missed performance improvements of 2% on average. Bao-for-PostgreSQL decreases the overall JOB run time by 33.1%, whereas AutoSteer-G saves an additional 0.4%, which results in a relative improvement of 33.5%. However, further experiments showed that the explored hint-sets and their performance implications also depend on the PostgreSQL configuration. *In other words, AutoSteer-G matches Bao-for-PostgreSQL's performance improvements, even though its hint-sets are automatically explored and not pre-selected by human experts.*

What are the top hint-sets found by AutoSteer-G? The top hint-set AutoSteer-G's training mode found is turning off *Nested Loop-Joins*. That hint-set improves query performance the most for 29 JOB queries, reducing the run time by 30.7% on average. The second-best hint-set turns off *index scans*: in many cases, PostgreSQL overestimates the selectivity of complex predicates and, therefore, index lookups yield substantial overhead compared to a sequential scan. These two top hints are also part of Bao-for-PostgreSQL. However, AutoSteer-G also discovered brand-new hint-sets, e.g., disabling *Parallel Hashing* and *GatherMerge*, which improved selected queries by up to 38.5% and 91.1%, respectively.

Is the greedy hint-set exploration approach effective? In Section 3, we hypothesized that beneficial hint-sets are often composed of smaller beneficial hint-sets and argued for using a greedy hint-set exploration approach. In Figure 10, we experimentally evaluate this assumption and visually compare AutoSteer-G's hint-set exploration with Bao-for-PostgreSQL's 48 handcrafted hint-sets for the JOB queries 10b, 12b, and e2a: The y-axis shows the hint-set size. Some hint-sets (■) defined by Bao-for-PostgreSQL result in duplicated execution plans. Contrary, AutoSteer-G tracks already-seen query plans and does not execute duplicates. The colors encode the query plan's relative performance wrt. the best and the worst plans known from both approaches (on a logarithmic scale). Light colors indicate better plans, and vice versa. Compared to PostgreSQL's default plan (the bottom-most square), most hint-sets result in worse plans, but only a few lead to better plans. Squares with black edges were discovered by AutoSteer-G's greedy training mode. Here, AutoSteer finds the best hint-sets (★) for each query.

How does AutoSteer-G's greedy hint-set exploration compare to randomized approaches? The hint-set exploration approaches used in Bao-for-SCOPE [30, 47] first uniformly draw hint-sets and then use SCOPE's cost model to select the ten hint-sets generating the cheapest plans. As discussed in Section 1, cost models may not always be available or sufficiently accurate. Therefore, AutoSteer-G's greedy hint-set exploration strategy does not rely on cost models. To compare these two strategies on fairgrounds, we tested both variants (w/ and w/o using the DBMS cost model) under Setup 3. We use eight representative JOB queries {10a,…,17a}, execute all query plans (for this experiment, we include duplicates, as some changes might not be exposed in the QEP) seven times to get robust measurements and we do *not* limit the execution time.

Greedy vs. Randomized for a Single JOB Query (Figure 11): We first compare greedy to randomized exploration without using the PostreSQL cost model. We consider the *expected query performance improvements* $\mathbb{E}(k) = (\sum_{x \in \mathcal{X}_k} \max(x))/|\mathcal{X}_k|$ for randomly drawing k hint-sets. Here, $\mathcal{X}_k$ is the set of all combinations with k hint-sets and $\max(x)$ returns the relative improvement of the top hint-set in x, or 0, if there are no improvements. We show the expected query performance improvements for JOB query 10a in Figure 11, for which we executed 250 randomly selected hint-sets. Greedy quickly gains the expected improvements in iteration 1 and stops after the second iteration 2 after exploring eight hint-sets. In contrast, the randomized exploration has to search significantly more hint-sets to achieve similar improvements (e.g., 95%/99% of greedy's improvements after exploring 32/43 hint-sets). These findings indicate that our greedy approach reaches higher query performance improvements faster than the randomized approach.

Greedy vs. Randomized for Multiple JOB Queries (Table 5): Next, we compare the two approaches for JOB queries {10a, …, 17a} (first w/o, then w/ using the cost model). As summarized in Table 5,

Table 5: Greedy vs. Randomized Hint-Set Exploration

Approach	Query Perf. Imp.	Time (min)
Greedy	48.68%	90.45
Randomized	50.22%	4597.10
Greedy w/ cost model	24.85%	1.58
Randomized w/ cost model	24.87%	10.77

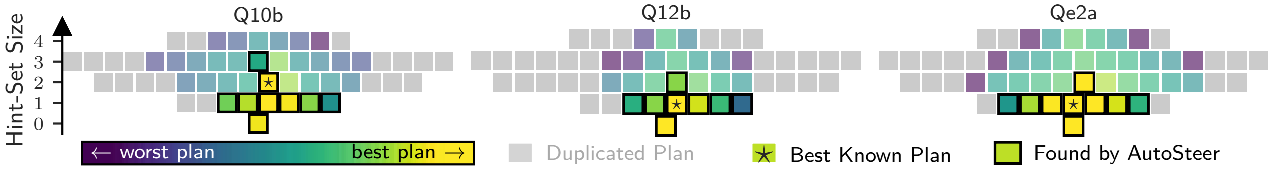


Figure 10: We compare AutoSteer-G's training mode with the 48 hint-sets defined in Bao-for-PostgreSQL [27] for three JOB queries. AutoSteer finds the best known hint-set for each of the three queries while it aggressively prunes the search space.

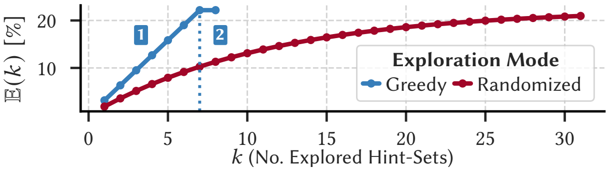


Figure 11: Expected performance improvements for greedy and randomized explorations for JOB Q10a and PostgreSQL.

greedy explores 86 hint-sets in about 90 minutes to reach a performance gain of 48.68%, while randomized explores 2000 hint-sets in about 4597 minutes to reach 50.22%. Thus, greedy reaches a similar level of performance improvement in about 2% of the time it takes for the randomized to do so. Finally, we compare variants of the two approaches that leverage the PostgreSQL cost model to limit their exploration to the ten cheapest QEPs. This significantly reduces the run times of both approaches but also degrades how much they can improve query performance to less than 25% since the PostgreSQL cost model overestimated the execution times of the most beneficial hint-sets. Overall, we observe that greedy strikes a good tradeoff between query performance improvement and exploration overhead while not requiring a reliable DBMS cost model.

RESULT SUMMARY. *AutoSteer-G's automated approach can find plans as good as those found based on Bao-for-PostgreSQL's expert-selected hint-sets, while also exploring the plan search space more efficiently. Furthermore, AutoSteer-G's greedy hint-set exploration strategy makes a better tradeoff between plan quality and exploration overhead than randomized alternatives.*

4.6 AutoSteer-G for SparkSQL

To further demonstrate our solution's general applicability, we also evaluated it with another widely used SQL engine, SparkSQL.

Can AutoSteer-G improve SparkSQL's query performance? Inspired by Intel's use of TPC benchmarks for its internal projects around SparkSQL, we experimented with different TPC-DS workloads and compared AutoSteer-G to SparkSQL's native optimizer for scale factors 1, 10, and 100. AutoSteer-G reduces the overall run time by up to 44.3% for SF1, 31.0% for SF10, and 22.0% for SF100.

Does the benchmark's scale factor impact AutoSteer-G's hint-set selection decisions? We interestingly observe that AutoSteer-G explored and selected a different collection of beneficial hint-sets at different scale factors. For smaller scale factors, the performance improvements primarily come from turning off expensive RRs such as *ConstantFolding*, which do not amortize for most short-running queries. SparkSQL primarily focuses on large-scale data processing. As performance is mainly dominated by query execution and not by its optimization, the RRs' efficiency was probably not a priority during development. With larger scale factors 10 and 100, performance improvements can be attributed to hint-sets improving the query plans on a structural level (e.g., ReorderJoins and CombineUnions).

RESULT SUMMARY. *AutoSteer-G can generate better query plans than SparkSQL's native query optimizer. Furthermore, due to its more adaptive approach, it can use different hint-sets at different scale factors – something that Bao would not be able to do due to its static approach to hint-set selection.*

4.7 AutoSteer-G vs. AutoSteer-C: Coding Effort

We implemented AutoSteer-G connectors for five well-known open-source database systems using Python3. We use the metric *lines of code* (LOC) to approximate the connectors' code complexity and exclude all comments, empty lines, import, and log statements. The connector for DuckDB has the fewest lines of code (34), followed by PrestoDB (53), PostgreSQL (49), MySQL (55), and SparkSQL's connector with 68 lines. For SparkSQL, we had to implement a post-processing step to remove random identifiers from the explained query plans. In contrast, implementing AutoSteer-C's custom integration for PrestoDB was significantly more complex, as we had to modify 1757 lines of code. The LOC metric indicates how simple the connectors are and prototyping a new connector for AutoSteer-G can be done in a few hours. All connectors are publicly available.[2]

5 LESSONS LEARNED AND FUTURE WORK

Applying AutoSteer in a Production Environment. We encountered several additional difficulties when applying AutoSteer to a large-scale production environment at Meta. First, PrestoDB's optimizer is cache-oblivious, meaning that the optimizer selects query plans without using information about the caches of a particular compute node. Cache hits or misses can significantly affect query performance. Thus, a change to a query plan that appears positive or negative may result from caching. Past works on learned query optimization dealt with this issue by simply assuming a warm or cold cache (e.g., [28, 42]), but in reality, the cache is rarely entirely warm or cold. An entirely warm or cold cache will often impact query performance more than many plan changes. Of course, the best solution to this problem would be to take the state of the cache into account as a feature (e.g., [46]), but this is easier said than done in large distributed environments: measuring the contents of Meta's PrestoDB deployment would take significantly longer than most queries. It is thus beneficial to examine many executions for a single query plan, preferably across a wide time range, to ensure different cache states are observed and accounted for statistically.

Second, many academic assumptions about query performance do not match the needs of some large organizations: (1) A few regressions are inevitable and acceptable with any optimizer change. Therefore, we use tail metrics – like P90/P95/P99 – to evaluate cluster performance and to accept or reject optimizer changes. (2) Changes in relative query performance are less important than changes in absolute query performance. For example, a 50ms query becoming a 200ms query looks like a 4x regression but is likely irrelevant in analytics. However, a 60s query becoming a 50s query, which "only" looks like a 15% improvement, is a desirable change. Thus, many past works using geometric means or relative latency metrics might be misleading. Metrics such as the geometric mean make an optimizer update that induces both previously described changes resemble a regression. However, it would actually be a significant upgrade. We suggest future evaluations of optimizers to include statistics about absolute changes in query latency.

Optimization Goals. In this work, we focus on minimizing query latencies. However, in industrial settings, there are different parameters that one would like to optimize for, including network transfers, I/O, and memory footprint, amongst others. For example, in Meta's PrestoDB deployments, memory is at a premium: increasing the concurrency of a particular query might improve its run time, but if the query's memory footprint increases substantially, other queries on the cluster might run out of resources, spilling to disk and causing general chaos. As a general rule of thumb, a 10% decrease in a query's memory footprint is as desirable as a 30% decrease in query latency (there are many exceptions to this rule, especially for queries with tight deadlines). Similarly, the CPU usage of a query is relevant to overall data center costs. Trading decreased latency for an overall increase in CPU time (e.g., again from parallelism) might be undesirable if the query was not time-critical.

Fortunately, AutoSteer can be easily extended to support arbitrary optimization functions. By changing the reward signal to whatever combination of measurable performance metrics is desired, AutoSteer can adapt to many different performance requirements. Unfortunately, real-world performance requirements often do not fit in single-query performance metrics. For example, a particular optimization might increase the memory footprint of one query by 60MB but decrease the footprint of two others by 25MB each. If these three queries run concurrently, this nets 10MB savings. However, such query-to-query tradeoffs are not expressible as a function of a single query's performance, so AutoSteer cannot yet handle them. We leave considerations for multi-query – and perhaps even entire workload – optimization to future work.

AutoSteer as a Tool for Human Experts. Query optimizers are highly complex software systems, as we observed firsthand in our collaboration with the PrestoDB team at Meta. Developed by over 130 software engineers, it has accrued almost 200 rewrite rules (RRs) [11]. While developers strive to make each rule applicable in general, this is impossible in practice; a rule that is helpful in one context may be harmful in another. For example, the *HashGenOptimizer* rule in PrestoDB enforces parallel generation of hash values for all joins, which improves the joining of large tables. However, the rule's overhead outweighs its performance gains for smaller tables. To address this issue, we crafted a heuristic that turned the HashGenOptimizer on or off based on the predicted input size, which resolved most of the regressions we observed in Meta's dashboard workload. It was AutoSteer that helped us improve PrestoDB's query optimizer by discovering this new heuristic, which is now being considered as a contribution to PrestoDB's upstream.

Furthermore, some rewrite rules can seem like "no-brainers" that ought to improve query performance wherever they are applied, but will surprisingly regress some queries. Since AutoSteer can automatically identify these kinds of surprising interactions like in the PrestoDB example above, it could serve as an invaluable tool for human experts in improving the design and implementation of their optimizers. On the other hand, investigating and debugging rule-based query optimizers for larger and more complex query plans that comprise tens to hundreds of relations would get increasingly more challenging. To simplify this process, one can use QO-Insight [12] – a visual tool that lets database experts explore AutoSteer's results interactively (i.e., by setting its interaction mode to *Debugging*) and supports a query- and rule-centric exploration mode. The former enables experts to analyze the potential improvements of benchmarks or individual queries, while the latter groups the performance results by hint-sets.

Integrating AutoSteer-G into other DBMSs. AutoSteer-G is easily applicable to other SQL databases. For MySQL and DuckDB, it took us less than an hour to implement the external connector. The main task is to figure out the database's syntax for toggling optimizer knobs. DuckDB has the session property `disabled_optimizers`, which is a string containing the list of disabled rules. MySQL exposes one session property per knob. For MySQL, AutoSteer found only around three effective rules because most of their changes were not exposed in the explained plan. For DuckDB, which exposes 14 knobs, we could improve the execution time of all JOB queries by 1.66% on average, but its tail is significantly enhanced, with a few queries improving by more than 10%.

To generalize, AutoSteer works best when the underlying DBMS (1) exposes a clean interface to modify the optimizer's configuration and (2) provides sufficiently detailed query plans to observe changes. Therefore, we argue that exposing more detailed optimizer statistics will help AutoSteer to detect better query plans. For example, database systems could have an option such as `EXPLAIN OPTIMIZATION <query>` returning a list or statistics describing the effective RRs. Such a feature would also help database developers see which RRs directly contribute to the final query plan.

6 CONCLUSIONS

In this paper, we introduced AutoSteer, a generic, learning-based query optimization framework that automatically steers traditional query optimizers of SQL databases. AutoSteer achieves this by extending Bao with automatically generated dynamic hint-sets, which can easily adapt to different query workloads and optimizers and lead to better plans than the manually selected static hint-sets in Bao. We have shown that our solution can be easily applied to several SQL databases and improve their query performance by up to 40% on well-known benchmarks. Furthermore, we tested AutoSteer on a real-world PrestoDB workload at Meta, where it achieved more than 20% reduction in 99% tail latency. Query optimization experts can also use AutoSteer as an interactive tool to generate insights that can be leveraged to improve existing RRs.

REFERENCES

[1] 2020. Bao for PostgreSQL. https://github.com/learnedsystems/BaoForPostgreSQL [Last Accessed: 2023/08/02].

[2] 2020. Solving Query Optimization in Presto. https://www.infoworld.com/article/3587781/solving-query-optimization-in-presto.html [Last Accessed: 2023/08/02].

[3] 2021. Applying Bao to Distributed Systems. https://rmarcus.info/blog/2021/06/17/bao-distributed.html [Last Accessed: 2023/08/02].

[4] 2021. Presto-on-Spark. https://prestodb.io/blog/2021/10/26/Scaling-with-Presto-on-Spark [Last Accessed: 2023/08/02].

[5] 2022. Bao Online Appendix. https://rm.cab/bao_appendix [Last Accessed: 2023/08/02].

[6] 2022. ML for Systems Papers. http://dsg.csail.mit.edu/mlforsystems/papers/ [Last accessed: 2023/08/02].

[7] 2022. MySQL Hints. https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_optimizer_switch [Last Accessed: 2023/08/02].

[8] 2022. PostgreSQL Hints. https://www.postgresql.org/docs/current/runtime-config-query.html [Last Accessed: 2023/08/02].

[9] 2022. PrestoDB Hints. https://prestodb.io/docs/current/optimizer/cost-based-optimizations.html [Last Accessed: 2023/08/02].

[10] 2022. SQLServer Hints. https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query [Last Accessed: 2023/08/02].

[11] 2023. PrestoDB on GitHub. https://github.com/prestodb/presto [Last Accessed: 2023/08/02].

[12] Christoph Anneser, Mario Petruccelli, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, Ryan Marcus, and Alfons Kemper. 2023. QO-Insight: Inspecting Steered Query Optimizers. *Proc. VLDB Endow.* 16, 12 (2023), 3922 – 3925.

[13] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark.. In *SIGMOD Conference.* ACM, 1383–1394.

[14] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources.. In *SIGMOD Conference.* ACM, 221–230.

[15] Nicolas Bruno, Surajit Chaudhuri, and Ravishankar Ramamurthy. 2009. Power Hints for Query Optimization.. In *ICDE.* IEEE Computer Society, 469–480.

[16] Amol Deshpande and Joseph M. Hellerstein. 2002. Decoupled Query Optimization for Federated Database Systems.. In *ICDE.* IEEE Computer Society, 716–727.

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

[18] Peter J. Haas, Ihab F. Ilyas, Guy M. Lohman, and Volker Markl. 2009. Discovering and Exploiting Statistical Properties for Query Optimization in Relational Databases: A Survey. *Stat. Anal. Data Min.* 1, 4 (2009), 223–250.

[19] Benjamin Hilprecht and Carsten Binnig. 2022. One Model to Rule them All: Towards Zero-Shot Learning for Databases. In *CIDR.* www.cidrdb.org.

[20] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005.

[21] Holger Kache, Wook-Shin Han, Volker Markl, Vijayshankar Raman, and Stephan Ewen. 2006. POP/FED: Progressive Query Optimization for Federated Queries in DB2. In *VLDB.* ACM, 1175–1178.

[22] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-depth Study.. In *SIGMOD Conference.* ACM, 1214–1227.

[23] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning.. In *CIDR.* www.cidrdb.org.

[24] Tim Kraska, Umar Farooq Minhas, Thomas Neumann, Olga Papaemmanouil, Jignesh M. Patel, Christopher Ré, and Michael Stonebraker. 2021. ML-In-Databases: Assessment and Prognosis. *IEEE Data Eng. Bull.* 44, 1 (2021), 3–10.

[25] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[26] Zhenxiao Luo, Lu Niu, Venki Korukanti, Yutian Sun, Masha Basmanova, Yi He, Beinan Wang, Devesh Agrawal, Hao Luo, Chunxu Tang, Ashish Singh, Yao Li, Peng Du, Girish Baliga, and Maosong Fu. 2022. From Batch Processing to Real Time Analytics: Running Presto® at Scale. In *ICDE.* IEEE, 1598–1609.

[27] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical.. In *SIGMOD Conference.* ACM, 1275–1288.

[28] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *CoRR* abs/1904.03711 (2019).

[29] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration.. In *aiDM@SIGMOD.* ACM, 3:1–3:4.

[30] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc T. Friedman, and Alekh Jindal. 2021. Steering Query Optimizers: A Practical Take on Big Data Workloads.. In *SIGMOD Conference.* ACM, 2557–2569.

[31] Parimarjan Negi, Ryan Marcus, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2020. Cost-Guided Cardinality Estimation: Focus Where it Matters.. In *ICDE Workshops.* IEEE, 154–157.

[32] Parimarjan Negi, Ryan C. 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.

[33] Fatma Ozcan, Sena Nural, Pinar Koksal, Mehmet Altinel, and Asuman Dogac. 1995. A Region Based Query Optimizer Through Cascades Query Optimizer Framework. *IEEE Data Eng. Bull.* 18, 3 (1995), 30–40.

[34] Zhifei Pang, Sai Wu, Haichao Huang, Zhouzhenyan Hong, and Yuqing Xie. 2021. AQUA+: Query Optimization for Hybrid Database-MapReduce System. *Knowl. Inf. Syst.* 63, 4 (2021), 905–938.

[35] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything.. In *ICDE.* IEEE, 1802–1813.

[36] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: A Modular Query Optimizer Architecture for Big Data.. In *SIGMOD Conference.* ACM, 337–348.

[37] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation. *Proc. VLDB Endow.* 15, 1 (2021), 85–97.

[38] TPC-DS Benchmark 2022. TPC-DS Benchmark. https://www.tpc.org/tpcds [Last Accessed: 2022/11/27].

[39] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (2021), 1640–1654.

[40] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. 2011. Query optimization for massively parallel data processing.. In *SoCC.* ACM, 12.

[41] Liqi Xu, Richard L. Cole, and Daniel Ting. 2019. Learning to Optimize Federated Queries.. In *aiDM@SIGMOD.* ACM, 2:1–2:7.

[42] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations.. In *SIGMOD Conference.* ACM, 931–944.

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

[44] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection. *Proc. VLDB Endow.* 15, 13 (2022), 3924–3936.

[45] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics.. In *CIDR.* www.cidrdb.org.

[46] Chi Zhang, Ryan C. Marcus, Anat Kleiman, and Olga Papaemmanouil. 2020. Buffer Pool Aware Query Scheduling via Deep Reinforcement Learning. In *AIDB@VLDB.*

[47] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc T. Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. 2022. Deploying a Steered Query Optimizer in Production at Microsoft.. In *SIGMOD Conference.* ACM, 2299–2311.

[48] 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 (2022), 1808–1821.

[49] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *VLDB J.* 21, 5 (2012), 611–636.

[50] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (2021), 46–58.

[51] Rong Zhu, Ziniu Wu, Chengliang Chai, Andreas Pfadler, Bolin Ding, Guoliang Li, and Jingren Zhou. 2022. Learned Query Optimizer: At the Forefront of AI-Driven Databases.. In *EDBT.* OpenProceedings.org, 1–4.