FASTgres: Making Learned Query Optimizer Hinting Effective

Lucas Woltmann
Technische Universität Dresden,
Dresden Database Research Group
Dresden, Germany
lucas.woltmann@tu-dresden.de

Kira Thiessat
Technische Universität Dresden,
Dresden Database Research Group
Dresden, Germany
kira.thiessat@tu-dresden.de

Claudio Hartmann
Technische Universität Dresden,
Dresden Database Research Group
Dresden, Germany
claudio.hartmann@tu-dresden.de

Dirk Habich
Technische Universität Dresden,
Dresden Database Research Group
Dresden, Germany
dirk.habich@tu-dresden.de

Wolfgang Lehner
Technische Universität Dresden,
Dresden Database Research Group
Dresden, Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT

The traditional and well-established cost-based query optimizer approach enumerates different execution plans for each query, assesses each plan with costs, and selects the plan that promises the lowest costs for execution. However, the optimal execution plan is not always selected. To steer the optimizer in the right direction, many query optimizers provide configuration parameters called query optimizer hints. These hints can be set for every single query separately. To show the great potential of these hints for the optimization of analytical queries, we present results of a comprehensive and in-depth evaluation using three benchmarks and two different versions of the open-source database system PostgreSQL. In particular, we highlight that query optimizer hinting is a non-trivial challenge. To solve this challenge, we propose *FASTgres*, a learning-based context-aware classification strategy for hint set prediction. Compared to related work, *FASTgres* provides transparent and direct hint set predictions with consistent performance improvements. In our end-to-end evaluation, we demonstrate that *FASTgres* effectively reduces benchmark runtimes by a factor of up to 3.25x with only steering the cost-based optimizer.

PVLDB Reference Format:
Lucas Woltmann, Kira Thiessat, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. FASTgres: Making Learned Query Optimizer Hinting Effective. PVLDB, 16(11): 3310 - 3322, 2023.
doi:10.14778/3611479.3611528

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/db-tu-dresden/FASTgres-PVLDBv16.

1 INTRODUCTION

With the continuously increasing amount of data managed by database systems (DBSs), efficient analytical query processing still poses a critical challenge [4]. To tackle this optimization task, every DBS features a query compiler that converts each incoming SQL query into a query execution plan (QEP). The most important component of such a query compiler is the *query optimizer* whose task is to determine the most efficient QEP [2]. Despite decades of research activities, query optimization is still far from being solved and, thus, the most efficient plan is not always executed [3, 15]. According to [2], the most challenging issues for the optimization of analytical queries consisting of complex joins and filter predicates are: (i) finding a proper join order and (ii) selecting the best-fitting physical join implementation for each join within the chosen join order. To solve these challenges, *traditional and over decades developed query optimizers* use three components: (i) an enumerator, which spans the search space of all possible QEPs, (ii) a cost model to assess the cost of any given QEP prior to its execution, and (iii) a cardinality estimator, which delivers the size of intermediate results and base tables as the most crucial input to the cost model.

Interestingly, such traditional query optimizers in many systems, e.g., PostgreSQL (PG) [25], MySQL [22], Oracle [24], or SQL Server [26], feature a rich set of configuration parameters to influence the characteristics of the selected QEP. These configuration parameters are also known as query optimizer hints since they can be specified separately as annotations for each query. For example, the evaluation of hash-join execution plan types can be enabled or disabled in PG [25]. Enabling or disabling such a hint for a query results in the underlying query compiler to either consider or strictly avoid using the hash-join plan types. By doing so, the optimizer can be hinted externally on a single query basis. Even though [18] has shown that hints can be used in two different DBSs for query optimization, still a clear and deep understanding of the optimization potential of hinting remains to be shown. Therefore, and in line with recent work [1, 7, 8, 13, 18], we conducted a comprehensive experimental analysis using PG for a profound understanding. As shown in [8], the physical operator selection matters for query optimization and, thus, we mainly focus our evaluation on the six primary Boolean PG hints for scan (sequential, index, index-only) and join (hash-join, merge-join, nested-loop) operations as suggested in [18]. Moreover, we use the *StackExchange (Stack)* [18], the *Join-Order-Benchmark (JOB)* [13], and the TPC-H benchmark [6] (scale factor of 10) for two different PG versions (v12.4 and v14.6). The PG versions are so different that they actually behave like two different systems.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611528

Table 1: Benchmark response times with PG default and optimal hint set configuration per query. The speedup compares the default and the optimal hint set configuration.

Benchmark & Configuration		PG 12.4	PG 14.6
Stack	default setting	19,384s	9,087s
	optimal hint set (**speedup**)	7,234s (**2.6x**)	3,561s (**2.5x**)
JOB	default setting	204s	185s
	optimal hint set (**speedup**)	87s (**2.3x**)	79s (**2.3x**)
TPC-H	default setting	445s	143s
	optimal hint set (**speedup**)	119s (**3.7x**)	110s (**1.3x**)

To analyze the optimization potential, we execute the following steps per benchmark and PG version: (i) determine the response time for every query with the optimizer using the default configuration, (ii) systematically execute every query with every possible hint combination – with six hints, $2^6 = 64$ combinations are possible –, and (iii) extract the optimal hint combination with the *lowest response time per query*. Then, we execute all benchmark queries by steering the optimizer with the optimal hint combination for every query. As illustrated in Table 1, query optimizer hinting has a decisive impact as the overall benchmark response times **are accelerated by a factor of 1.3x to 3.7x** compared to an execution run with the default optimizer configuration. This potential is considerable since we did not make any internal changes to the optimizer itself but steered it properly with existing hints.

Core Contribution: To utilize this potential for the optimization of complex analytical queries, we present *FASTgres*. *FASTgres* is a lightweight, learning-based context-aware classification strategy for predicting hint combinations directly for incoming queries. *FASTgres* is explicitly designed to augment any relational cost-based query optimizer that can be steered by hints. To show the impact on different systems, we focus on different versions of the open-source DBS PG. The input of *FASTgres* is a pure SQL query, while the output is a set of predicted proper hints for this query. Subsequently, the traditional cost-based query optimizer is invoked using the query and the predicted hints. Thus, *FASTgres* considers the underlying query optimizer a black box and does not require a deep integration into the optimizer, facilitating broad applicability. Unlike other learning approaches, we do not learn one global model but a distinct model per well-defined context to make fine-grained predictions of hint combinations. The idea behind our approach is that each set of joined tables represents a self-contained context since the queries per context are reasonably homogeneous with respect to the joins and differ only in the filter predicates. The context-aware models can distinguish fine-grained hint combination distinctions, which would be otherwise overshadowed by queries of completely different contexts. Additional advantages are (i) the sufficiency of simpler models, (ii) less training effort per context, and (iii) low runtime maintenance effort within each context. Especially, (iii) is useful to render on-line model updates a feasible option.

Contributions in Detail and Outline: To present *FASTgres*, we make the following detailed contributions:

- We start with a problem statement using an experimental analysis to investigate the potential of query optimizer hinting in Section 2. Moreover, we highlight the associated challenges for effective query hinting. In addition, we show that recent related work in this area is not able to achieve the potential to the full extent.
- In Section 3, we introduce the overall architecture of our *learning-based context-aware classification strategy* called *FASTgres* to accelerate analytical query executions by steering the optimizer with learned hints.
- Subsequently, Section 4 describes details of *FASTgres*, while the runtime model maintenance for *FASTgres* to cover changing conditions is presented in Section 5.
- In Section 6, we evaluate *FASTgres* in an end-to-end fashion by highlighting further insights on hinting. To show the effectiveness and applicability of *FASTgres*, we also provide a comparison to recent machine learning (ML) approaches.

After discussing related work in Section 7, we briefly summarize the paper in Section 8.

2 THE MERITS OF QUERY HINTING

Query optimizers in many database systems (e.g., Oracle [24], PostgreSQL (PG) [25], MySQL [22], or SQL Server [26]) feature a rich set of configuration parameters to influence the determination of the optimal QEP. The query optimization potential of these configuration parameters has become the subject of current research [18] as it is an important topic to investigate. To show this potential, we present selective results of an exhaustive experimental evaluation using three different benchmarks: the *StackExchange-Benchmark (Stack)* [18], the *Join-Order-Benchmark (JOB)* [13], and the TPC-H benchmark (scale factor of 10) [6]. In the following, we will focus primarily on Stack and JOB since the results for TPC-H do not differ. While Stack contains data from over 18 million questions and answers from different StackExchange websites with more than 6,000 analytical queries, JOB comprises 113 analytical queries over the Internet Movie Database with a total size of 10GiB.

In our experimental analysis, we mainly focus on the open-source DBS PG as a primary representative featuring a traditional cost-based query optimizer that has evolved over decades. The available configuration parameters are manifold [25]. Without loss of generality, we restrict ourselves to the following *six* Boolean configuration parameters with regard to scans and joins to force the optimizer to choose a different plan [25] as suggested in [18]:

- **hash join:** Enables or disables hash-join plan types.
- **merge join:** Enables or disables merge-join plan types.
- **nested loop join:** Enables or disables nested-loop join plans. Disabling discourages the optimizer from using this join operator if other methods are available.
- **index scan:** Enables or disables index-scan plan types.
- **sequential scan:** Enables or disables sequential scan plan types. Disabling discourages the optimizer from using sequential scans if other methods are available.
- **index-only scan:** Enables or disables index-only scan plans.

In the default PG setting, all six configuration parameters are enabled to span the largest possible plan search space. A one-elementary configuration is referred to as a *hint*, a multi-elementary configuration as a *hint set*, and the process of configuring and applying hints or hint sets as *query optimizer hinting* or just *hinting*. Based on the six considered Boolean hints, $2^6 = 64$ possible hint sets exist. For now, assume the hint order as introduced above to

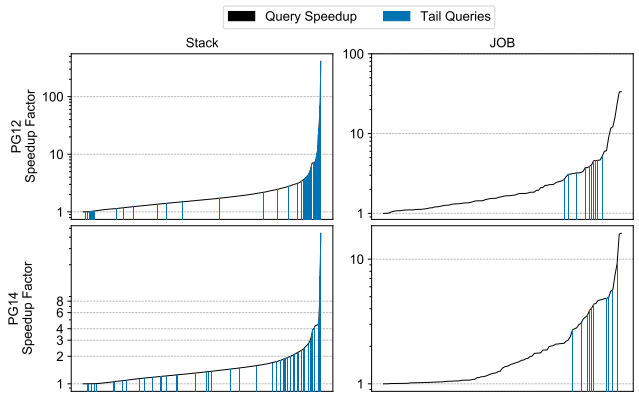


Figure 1: Speedup potential through optimal hinting for different benchmarks and PG versions.

be subsequently maintained and that every hint set is numbered as an integer according to the binary representation, e.g., the hint set with only hash joins enabled is represented as $100000_2 = 32_{10}$.

To systematically evaluate the effect of *query hinting*, we compare the query response times for all benchmark queries using (a) the default optimizer configuration and (b) all possible hint sets. Based on an exhaustive evaluation, we are able to determine the QEP with the lowest response time for every single query, including the optimal hint set per query. Unless described otherwise, we ran our experimental evaluation on a 64-bit Linux machine with an Intel Xeon Gold 6216 CPU (Skylake architecture) with 12 cores, 92 GiB of main memory, and 1.8 TiB HDD storage. This hardware environment is further denoted as Skylake machine. Moreover, we used two different PG versions (v12.4 and v14.6).

2.1 Results

Table 1 comprises the cumulative query response times for three considered benchmarks and both PG versions using the corresponding optimizer with the default configuration and the optimal hint set per query. As clearly visible in Table 1, a proper *hinting* of the optimizer leads to an enormous speedup. Regarding Table 1, Figure 1 displays the relative speedup per query for Stack and JOB. The individual queries are sorted according to the relative speedup. The diagrams in Figure 1 indicate that almost all queries for both shown workloads and both PG versions can be accelerated by proper hinting. In general, the achieved maximum speedup gain per query is much higher in PG v12.4 than in v14.6, but the overall achieved speedup for both versions is equal, as shown in Table 1.

From this analysis, we can conclude that hinting offers great optimization potential. To further emphasize the importance, a deeper investigation of queries benefiting the most from hinting seems appropriate. To answer that question, we colored the slowest 100 for Stack and 10 for JOB queries according to the default configuration of the optimizer per benchmark – the queries are usually called tail latency queries – in Figure 1. On average, these tail queries experience the highest relative speedup and thus profit the most from effective query hinting. Thus, hinting is beneficial in general and specifically for tail latency queries.

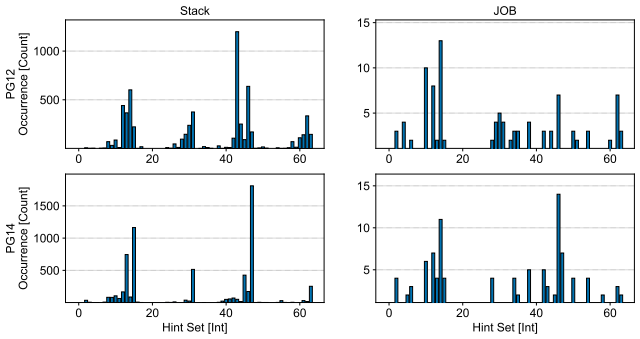


Figure 2: Distribution of optimal hint set occurrences for different benchmarks and PG versions.

However, this still leaves the question of which hint sets get the most out of the optimization potential. To answer that question, Figure 2 shows how often a certain hint set is used to reach the minimal query response times in the benchmarks. As we can see, nearly all hint sets are used and no pattern is visible for a general recommendation. Moreover, we observe fine-grained differences, which are decisive. For example, the most frequent hint set for the Stack benchmark in PG v12.4 is 43 (binary representation 101011; count 1251), while in PG v14.6, it is the hint set 47 (binary representation 101111; count 1848). Both hint sets are similar, except that the hint for sequential scans is enabled in v14.6, while it is disabled in v12.4. Additionally, there are differences between the benchmarks. Compared to the most frequent hint set 42 for Stack in PG v12.4, hint set 14 (binary representation: 001110) is crucial for JOB as it is the most frequently used one in this case. Thus, we can conclude that the hint set distributions are different depending on the workload as well as the optimizer version and that these differences are important. This is backed by an additional analysis result shown in Figure 3, where we depict the relative cross appearance of all hints in the optimal hint sets. The principle diagonal contains the relative occurrences of a single hint in optimal hint sets. Here, we can see that the direct effect of a combination of hints in a hint

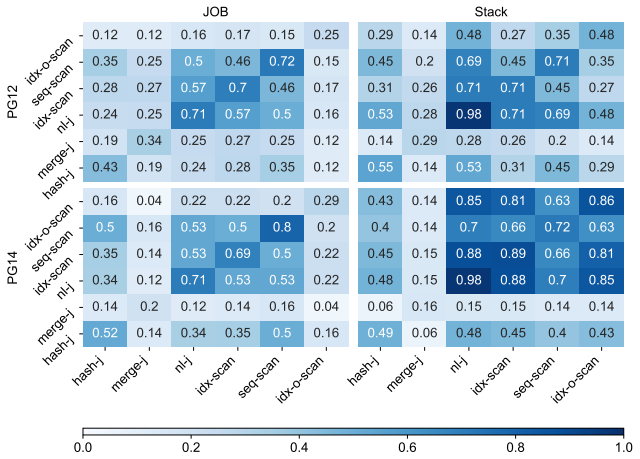


Figure 3: Relative co-occurrences of hints in optimal hint sets for different benchmarks and PG versions.

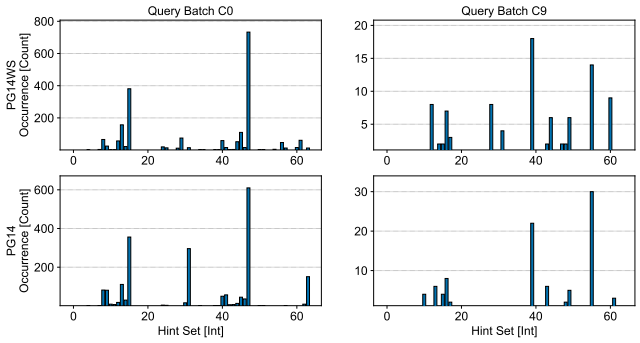


Figure 4: Distribution of optimal hint set occurrences for two different Stack query batches and PG versions.

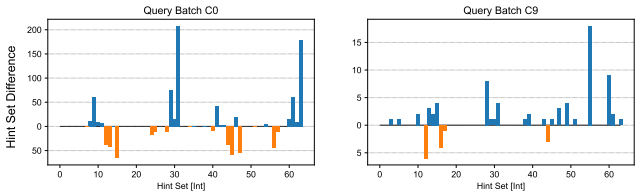


Figure 5: Distribution differences of optimal hint set occurrences for two different Stack query batches for PG v14.6 on two different hardware machines.

set is not trivial, as the hint set distribution varies greatly across the heat map. Again, no general pattern is visible. Each hint is used with a wide range of other hints in different combinations for QEPs with the best query response times regarding the chosen hint sets.

So far, our analysis has focused on all queries within the benchmarks and we have not been able to detect any hint set pattern. However, the benchmark queries are usually diverse with regard to different joins as well as filter patterns. Thus, the question of how this diversity influences the optimal hint set selection arises. To answer this question, we re-grouped the Stack queries into 11 batches named C0 to C10. All queries in a specific batch join the same set of tables with the same join predicates and vary only in the filter predicates. Figure 4 shows the distribution of the optimal hint set occurrences for two representative query batches, namely C0 and C9. While C0 contains 2,009 queries, C9 has only 100 queries. Despite the limitation to similar queries per batch, nearly all hint sets are present per batch. Additionally, we see that even within these batches, the hint set distributions differ between the PG versions. Additionally, *query hinting* depends not only on the query and PG version but also on the underlying hardware. To show that, we repeated our entire evaluation on a different hardware environment. In this case, we used a second machine – further referred to as CoffeeLake machine – with an Intel Xeon E-2186M (Coffee Lake architecture) with six physical cores, 64 GiB of main memory, and 1.8 TiB NVMe SSD storage. The results on this CoffeeLake machine reveal the same findings, although the hint set distributions are different. Figure 5 illustrates this for the Stack query batches C0 and C9 by showing the differences in the hint set distributions for PG 14.6 on both hardware environments with striking and decisive differences. Thus, we can conclude that *hinting* offers great optimization potential, but it is a non-trivial challenge requiring fine-grained decisions. In particular, *hinting* depends on the query, the PG version, and the underlying hardware.

2.2 Related Work using Hints

To the best of our knowledge, the *Bandit Optimizer (BAO)* [18] is the only work that uses query optimizer hints to optimize queries. In particular, BAO utilizes the same six Boolean hints for joins and scans. Unfortunately, the authors did not present a comprehensive analysis of the potential of query hinting and its associated challenges. In general, BAO is a learning-based approach extending the traditional cost-based optimizer by continuously learning an additional runtime-oriented cost model in the form of a tree convolutional neural network (TCN) based on experiences from executed QEPs. To determine a QEP for a query, BAO conducts the three following steps: (*Step 1*) BAO creates different QEPs by calling the traditional query optimizer multiple times. In each call, BAO steers the traditional optimizer with a different hint set from a limited predefined set of hint sets. (*Step 2*) The resulting QEPs are re-evaluated with the runtime-oriented cost model to select the best-fitting QEP with regard to its model. In detail, the resulting QEP trees are used as input for the TCN that predicts the execution time (i.e., quality) of each query plan tree based on the learned model. (*Step 3*) Finally, the QEP with the lowest predicted execution runtime is used for the actual query execution. Once the execution is completed, the combination of the selected QEP tree and the observed performance is used to adapt the learned TCN model continuously.

To include BAO in our comprehensive evaluation, we used the available BAO PG extension [17]. In particular, we trained BAO with the maximum amount of feedback possible based on a certain fraction of randomly selected Stack queries and froze the corresponding learned TCN model afterward. Then, we used the frozen snapshot and ran the *remaining* benchmark queries to evaluate the degree of acceleration. For each fraction, we performed five experiments, each with different randomly selected queries and we only show the average speedup for Stack depending on the number of learned queries over these five experiments in Figure 6. BAO slightly accelerates analytical queries for Stack but does not exploit the full potential of query hinting marked by the red line. There are many reasons why the potential is not even close to being reached. Most importantly, BAO uses only the following five hard-coded hint sets: 35, 43, 47, 55, and 63 (denoted as BAO Default in Figure 6). However, our analysis has shown that *hinting* is hardware-dependent, among

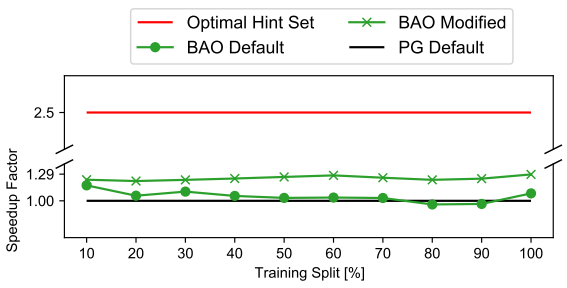


Figure 6: Evaluation of BAO using Stack.

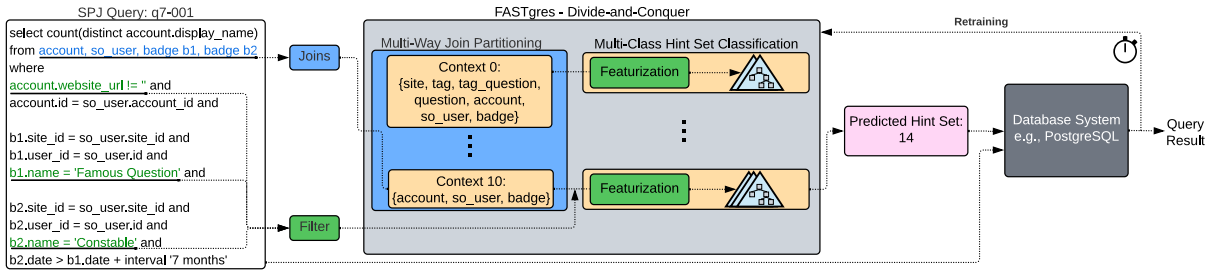


Figure 7: System model of *FASTgres*.

other things, and that the hint sets 12, 14, 43, 46, and 63 are the five most common for the Stack workload on our Skylake machine. Thus, we modified BAO accordingly (denoted as BAO Modified in Figure 6), resulting in a much better acceleration, closer to the original results [18] but still far from the maximum possible potential. This once again reflects the non-trivial challenge of *hinting*.

2.3 Lessons Learned

Our comprehensive experimental analysis shows that *query hinting* offers a great potential for accelerating analytical queries. The advantage of hinting is that we can consider existing query optimizers as black boxes and steer them using available hints. However, hinting is non-trivial because of different hint set distributions per workload, hardware, and optimizer, even when considering table set batches. With BAO [18], there is a recent approach that indirectly uses hints for optimization but cannot harness its full potential. All this motivates an approach that directly predicts an appropriate learned hint set per query and passes this hint set with the query to the unmodified optimizer. Moreover, the hint sets are learned separately by workload and hardware to provide adaptability.

3 FASTGRES: OVERALL SYSTEM DESIGN

To accelerate analytical queries with optimal optimizer hint sets, we propose *FASTgres*, a novel lightweight, learning-based context-aware classification model for hint set prediction. As illustrated in Figure 7, *FASTgres* is a learned standalone component predicting a optimal hint set for each incoming analytical query. After the going through *FASTgres*, the query and its predicted hint set are forwarded to an unmodified query optimizer and execution engine.

Design: From a high-level perspective, *FASTgres* is composed of the following simple and lightweight concepts: (1) using a divide-and-conquer approach, the overall problem space is partitioned into smaller and specific sub-problems called *contexts*, (2) within each context, a separate supervised multi-class classification model for hint set prediction is learned, (3) a context-aware and challenge-specific training phase is supplied to minimize the training overhead, and (4) a retraining of individual context-sensitive models at runtime is deployed if necessary. While (1)-(3) are explained in detail in Section 4, Section 5 exclusively deals with (4).

Assumptions and Limitations: Generally, *FASTgres* is not limited to a specific DBS. However, it assumes that the corresponding optimizer can be hinted with a finite set of Boolean hints to enable or disable physical operators. This assumption is realistic since several DBSs offer appropriate optimizers such as PG [25]. Moreover, *FASTgres* assumes that hinting can be conducted at the granularity of a single query resulting in semantically equivalent QEPs. To use the full potential of these hint sets for query optimization, the hint set search space cannot be restricted, as shown in our comprehensive evaluation. Thus, for k hints, all possible 2^k hint sets have to be examined and we propose a supervised learning approach as solution with the assumption that a set of representative workload queries is available that can be used during an explicit model training phase. While training our model may be costly, applying the model at runtime is orders of magnitude faster and adds only little overhead to the query optimization process. To avoid model staleness, *FASTgres* features a component for runtime maintenance. This maintenance component determines if retraining is necessary on a per-query basis and triggers it accordingly.

4 SUPERVISED LEARNING IN FASTGRES

A learning-based approach is an *arbitrary function approximation* and thus, the high-level challenge for *FASTgres* is to learn a function being able to map a SQL query to an optimal hint set directly:

$$\text{SQL query} \rightarrow \text{hint set}$$

Since ML models are not naturally compatible with SQL string inputs, SQL queries have to be transformed into numerical representations initially. Transforming queries into a numerical representation, i.e., *feature vector*, is referred to as *query featurization*. These feature vectors serve as input to a ML model. Since we focus on analytical queries, we are mainly interested in tables, join predicates, and filter predicates in workload queries. Thus, the supervised learning challenge for *FASTgres* can be specialized as follows:

$$\text{tables} \times \text{join predicates} \times \text{filter predicates} \rightarrow \text{hint set}$$

4.1 Divide-and-Conquer Approach

To solve this challenge, *FASTgres* adopts a divide-and-conquer solution. The diversity of query structures in a workload is usually limited since these queries work on the same relational database with a finite set of tables. Thus, workload queries can be partitioned into query groups. One of these query groups is characterized by using the same *multi-way join group*. A multi-way join group is defined as a set of workload queries that contain the same set of joined tables with the same join predicates. Accordingly, different multi-way join groups are defined by different joined tables and join predicates. There is the possibility that the multi-way join groups overlap or include each other, but two groups never contain the

same exact set of joined tables and join predicates. Within a multi-way join group, the queries only differ in their filter predicates. In the following, a single partition of a workload representing one multi-way join group will be referred to as a *context*. Our analysis in Section 2 shows that (i) queries within a multi-way join group utilize various different optimal hint sets, and (ii) the optimal hint set distributions differs between different multi-way join groups. To reflect this knowledge in *FASTgres*, we follow a two-phase ensemble approach as illustrated in Figure 7. In the first phase, we create a set of well-defined contexts by partitioning the query workload into sub-groups with respect to the different multi-way join groups. Thus, the number of distinct multi-way joins in the workload determines the number of contexts. In the second phase, we learn a separate and context-sensitive ML model for each context. This divide-and-conquer approach offers several advantages. On the one hand, our two-phase approach reduces the overall complexity because joins are modeled outside the learned part. This allows simpler ML models to be used. On the other hand, each ML model for each multi-way join group in a context can focus on learning fine-grained characteristics based on the filter predicates. Thus, the two-phase approach of *FASTgres* can be formulated as follows:

$$\forall \text{ multi-way join groups} : \text{filter predicates} \rightarrow \text{hint set}$$

According to this, our approach is *context-aware* as it adapts to different properties of a query, i.e., multi-way joins and filter predicates, by dividing the problem space into sub-problems. This divide-and-conquer character maps every query to exactly one context.

As stated above, the contexts are extracted from the workload queries via the joined query tables and join predicates. To obtain different layers of context-awareness, we identify three different levels of granularity for contexts. For the coarsest level, namely $FASTgres_{coarse}$, we use no divide-and-conquer approach at all and create only one context representing all possible multi-way join groups. In contrast, the finest context granularity level, $FASTgres_{fine}$, corresponds to splitting the workload into every multi-way join groups, even if they overlap. For example, a query joining tables `A` and `B` with the join predicate `A.x = B.y` would be in a different context than a query joining `A` and `B` with the join predicate `A.x = B.z`. However, this may lead to infeasible amounts of contexts to consider. To keep the number of contexts as small as possible while still retaining table join characteristics, we propose a third granularity that settles between $FASTgres_{coarse}$ and $FASTgres_{fine}$. $FASTgres_{table}$ divides the workload according to each set of joined tables ignoring the join predicates. For example, all queries joining tables `A` and `B`, and `C` belong to the same context independently of the used join predicates, while queries joining tables `A` and `B` belong to a different context.

4.2 Hint Set Classification

The second phase of *FASTgres* uses the defined contexts from the first phase as preliminary context-aware states and builds upon them. The core idea of this second phase is that each context gets its own local ML model to learn the mapping between filter predicates and optimal hint sets. Doing so allows us to learn fine-grained differences between filter predicates and optimal hint sets without considering join predicates. Moreover, we propose how context-sensitive models can be used to directly predict hint sets in the form of their integer representation (i.e., class).

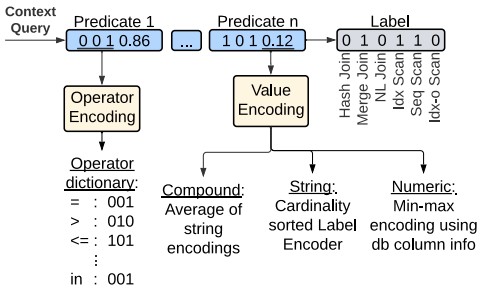


Figure 8: Featurization of queries in *FASTgres*.

To achieve this property, we define our classification problem for hint set prediction as a learned function mapping from a numerical representation of query X to a multi-class label y. Each label represents a hint set as an integer containing all binary optimizer hints. We decide to use multi-class instead of multi-label prediction since a multi-label approach would predict all hints separately, which neglects all complex correlations between single hints. However, as shown in our comprehensive evaluation (cf. Figure 3), hints have complex correlations with each other. Contrarily, a multi-class approach predicts the whole hint set as a class for a query at once. Thus, such co-dependencies are treated appropriately.

For the numerical representation of a query, we include filter information within each context, reducing the sparsity of the featurization. Thus, we use a predicate-based encoding as presented in Figure 8. In our query featurization, we first collect all filter predicates and their operators. Operators in a query are then binary encoded from a lookup table. Next, we identify the type of each predicate value. For numeric values, we min-max normalize the value using database column statistics. For strings, it uses the normalized rank of the predicate ordered by the cardinalities of all predicates on the base tables. For compound values, e.g., predicates with the `IN` operator, the average of all string encodings is used. This featurization is naturally adapted within each context. Extending this featurization to include join information is possible, but this would lead to sparser query representations, which may lower the model's quality [23].

For modeling the classification itself, we use Gradient Boosting (GB). GB models use the sum of predictions from weak learners, i.e., decision trees, where every single learner is trained on the residuals of the previous learner [20]. To predict the hint set hs of a query q, a GB model uses P predictions F_p of weak learners scaled with λ_P and a constant c.

$$hs(q) = \sum_{p=1}^{P} \lambda_p F_p(q) + c$$

Through its simple structure, it is fast in training and small in memory footprint [23]. This simplicity of our model enables our approach to scale well with increasing amounts of contexts. Moreover, related work has already shown that using several GB models for solving problems in DBSs is faster and produces better quality predictions than using one large model, like a single specialized

neural network [23, 29]. We also tested models like random forests or neural networks. However, GB shows to be the best combination of simplicity, expressiveness, and fast training time. GB models have two primary hyperparameters: the number of estimators, i.e., decision trees, and the maximum depth of estimators. The maximum depth is set to infinite as [8] have shown that endlessly deep tries (i.e., prefix trees) can capture the complexity of predicates and their influence on physical operators adequately. Based on experiments, we fix the number of decision trees to 100.

4.3 Training

Through the supervision character, *FASTgres* requires an explicit training phase to produce expressive models, which takes some time. However, the training phase can profit from extensive optimization since it does not interfere with the prediction of hint sets during runtime. Compared to reinforcement learning approaches, like BAO [18], the separated training phase also eliminates the effect of catastrophic forgetting during runtime. This section aims to present techniques to reduce the required time to set up a model by improving the training phase of *FASTgres*. In general, the setup of *FASTgres* requires two parts: *labeling* and *training*. In the following, we show how both parts can be improved.

Supervised learning requires a labeled set of training queries where a label contains the best, i.e., optimal hint set for the query. Retrieving these labels, or *labeling*, is a time-consuming step, but it only needs to be run once for a fixed set of training queries. Our labeling needs to retrieve the optimal runtimes and the corresponding hint sets for all queries in the training data. The naïve approach is to run every query using every possible hint set, store the corresponding runtimes and hint set, and choose the best ones for every query. This is very ineffective as it is not only very time consuming but also stresses the database enormously. To overcome this issue, we introduce a *smart labeling strategy*. The core idea is a runtime timeout per query, after which the query is aborted and the corresponding hint set can be safely discarded because it is not the best one. Our smart labeling strategy iterates over all queries in a training data set, which is a subset of a given workload. Initially, the current query is executed with the PG default hint set and the resulting response time is set as a timeout as we are only interested in hint sets with a reduced response time. To further speed up the labeling process, we aggressively lower the timeout whilst evaluating each query for every hint set. Whenever a hint set leads to a response time lower than the current timeout, this new response time replaces the timeout. Additionally, the hint set is collected as the (current) best hint set for the query. Thus, the labeling process speeds up over time when better hint sets with lower query response times are retrieved for a query. This leads to reasonable runtimes even for larger training data sets while generating the complete set of optimal hint set labels at the same time.

$t_{best} = q$ with $h_{PostgreSQL}\ Y[q] = h_{PostgreSQL}\ \ timeout = t_{best}$

For *training*, each context-aware model in *FASTgres* requires data in the form of labeled analytical queries. To achieve this, most other ML-based approaches take random samples from the available workload of queries [10, 16, 28, 30]. This approach is conditionally applicable in *FASTgres* because we need representative queries from every context to train expressive context-aware models. To tackle this query demand, we propose a *stratified sampling approach* that samples queries from every context according to the split criteria. For example, an 80-20 split would sample 80% of queries from every context separately and collect these queries in a training dataset. The remaining 20% from every context form the test dataset for evaluation. Especially for small training splits, our stratified sampling provides equal opportunity. After sampling and labeling a certain amount of queries, the GB models are trained with the labeled training data for every context. Here, the training times profit from the simple but powerful structure of GB models based on decision trees.

5 RUNTIME MAINTENANCE

The supervised approach of *FASTgres* is possibly too rigid since every model within each context is fixed after training. Additionally, the quality of each model depends on how well the training queries enable a model to learn the mapping between filter predicates and optimal hint sets. Thus, it is likely that *FASTgres* might not predict an ideal hint, e.g., due to data or workload drifts. To overcome this issue, a naïve approach would be to initiate a retraining with every incoming query. This would be possible from a model point of view because each query is assigned to exactly one small GB model in *FASTgres* and only this model would be subjected to retraining. In our case, the retraining of a single GB model is negligible. However, the frequency of retraining during runtime already makes this approach impractical. This is further complicated by the fact that for each retraining, the corresponding query would have to be executed with all 2^k possible hint sets to determine the optimal hint set for the retraining. Although this labeling can be optimized as described in the training phase, this still creates an additional high load on the DBS.

Since we cannot completely avoid retraining to achieve good model qualities all time, we propose a novel approach based on *active learning*. Instead of triggering a retraining with every incoming query, we want to actively initiate a retraining of a context model in *FASTgres* when an update is necessary. To recognize the proper situations, our divide-and-conquer approach is again beneficial: Once incoming queries are executed with a predicted hint set, we also supply a per-context timeout to the query compiler such that abnormally long-running queries are detected. This timeout is calculated for each context model based on the best response times for every query collected during labeling. Timeout f is then a mapping for every context c, corresponding observed best times $times_c$ during labeling, and a combination of absolute and percentage-based timeout $t_a \in \mathbf{R}, t_p \in \{0, \ldots, 100\}$ as follows:

$$f(times_c, t_p, t_a) := max(t_a, percentile(times_c, t_p))$$

Doing so has a variety of implications for our model. Firstly, we use percentile-based timeouts, as choosing a value in the range of observed times allows us to enforce different retraining behaviors. A small value t_p enforces greater retraining rates as the probability of exceeding a smaller timeout increases. Contrarily, high values of t_p may decrease the retraining rate. This results in retraining only by observing hinted queries that exceed all previously seen response times. Moreover, using an absolute threshold t_a implies a minimal timeout value. This is especially useful for contexts consisting solely of fast-running queries with lower query response times than t_a.

Table 2: Overview and details of the multi-way join contexts for the Stack benchmark.

Context	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
#Tables within Multi-way Join	7	6	4	8	3	6	7	7	6	7	3
#Queries	2,009	1,008	1,072	1,202	100	100	200	200	100	100	100
Average Query Response Time (PG Default)	1.09s	0.35s	0.34s	0.59s	1.40s	0.52s	0.70s	0.04s	0.40s	49.12s	1.53s
Average Smart Labeling Time	54s	18s	18s	31s	14s	30s	33s	1s	10s	11m 38s	1m 30s
Average Smart Labeling Speedup	1.3	1.2	1.2	1.2	6.4	1.2	1.4	2.6	2.6	4.5	1.1

Hinted queries will then not trigger a timeout until exceeding t_a. This value can be chosen in a workload-dependent manner. For the three investigated benchmarks, a value of $0.1s$ is sufficient. Notably, the chosen context-sensitive timeouts are not static after being calculated. Upon retraining within a context, abnormal queries are asynchronously labeled and then added to our models. After each retraining, we add the optimal hint set time of the labeled queries to our experienced times $times_c$. The corresponding timeout is then updated accordingly. This active learning-based retraining introduces runtime adaptivity to our supervised classification approach, making it more flexible and precise during application runtime.

In addition, changes in workload queries or database schema may occur at runtime. In the worst case, changes in the workload queries or schema manifest themselves in context changes (i.e., multi-way join groups). Thus, either new contexts have to be created or existing contexts have to be dropped. Both situations are uncritical for *FASTgres* since a separate model exists for each context. If a new context evolves at runtime, a new model can be created and learned. To build this model, incoming queries can be used for labeling and training, which can be done asynchronously. As long as no corresponding model exists, *FASTgres* predicts the default hint set. With this, we do not achieve a speedup and neither do we make queries artificially slower. If a context is no longer needed, the corresponding model can be deleted.

6 EVALUATION

To evaluate *FASTgres*, we use three different benchmarks: Stack [18], JOB [13], and TPC-H [6]. Unless otherwise described, we show the experimental results of our Skylake machine with an Intel Xeon Gold 6216 CPU (Skylake architecture) with 12 cores, 92 GiB of main memory, and 1.8 TiB of HDD storage. We fully implemented *FASTgres* in Python and used two different PG versions – namely v12.4 and v14.6 – as underlying black box DBS. The optimizers of both systems can be steered with the six Boolean hints described in Section 2, which is also in the focus of our evaluation. In line with related work [8, 18], we do not adopt any particular benchmark query order. Instead, each experiment was run several times, each run had a different random query ordering. Therefore, the following results are always average values over runs.

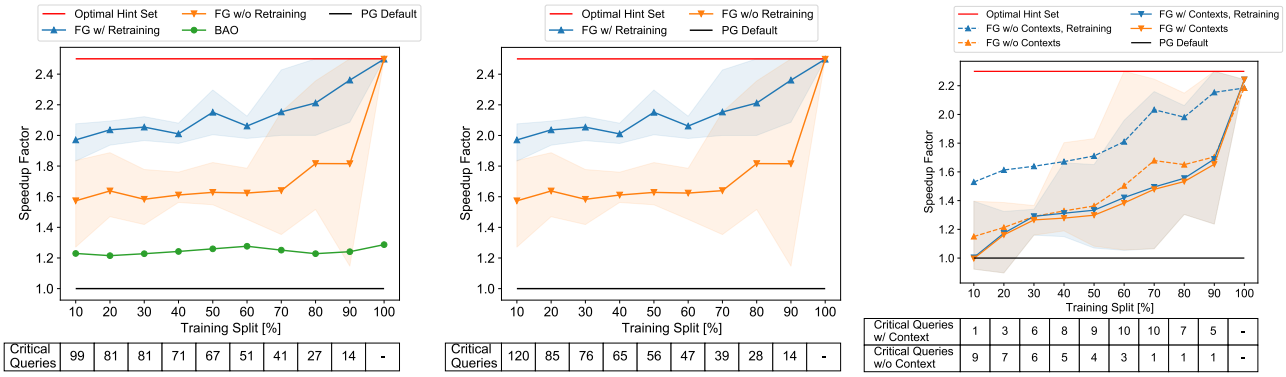
Since *FASTgres* adopts a divide-and-conquer approach using *multi-way join groups* (i.e., contexts), we first analyze the benchmarks regarding contextualization. The Stack benchmark consists of ten relational tables and 6,191 workload queries, which can be divided into eleven different contexts as shown in Table 2. Each context joins a different number of tables, ranging from three to eight. While seven contexts contain either 100 or 200 queries, the remaining four contexts consist of more than 1,000 queries each, the largest having 2,009 queries. Moreover, for Stack, multi-way joins are built such that the granularity levels $FASTgres_{fine}$ and $FASTgres_{table}$, as introduced in Section 4.1, coincide. This means, there are no contexts joining the same set of tables with different join predicates. In contrast, the JOB workload has a total of 113 queries, distributed over 33 contexts, while TPC-H has 22 workload queries over 18 contexts. This results in scarcely any queries per context for JOB and TPC-H as well. Again, the granularity levels of $FASTgres_{fine}$ and $FASTgres_{table}$ coincide. Thus, a distinction between $FASTgres_{fine}$ and $FASTgres_{table}$ is redundant.

6.1 Exploiting the Potential

In the first set of experiments, we investigated whether the optimization potential of *hinting* is achievable with *FASTgres*. For this purpose, we enabled *FASTgres* to learn with all benchmark queries in the training phase (i.e., *train-to-represent*). Then, we executed all benchmark queries again, where *FASTgres* predicts that hint set per query that it deems optimal. The resulting benchmark response times for the different PG versions are shown in Table 3. For comparison, the benchmark times for PG using the optimal hint sets (denoted as optimal hint set) are shown as well. The benchmark times for PG with default settings are depicted in Table 1. Additionally, the BAO benchmark runtimes for PG v12.4 are depicted since the BAO extension does not run on PG v14.6. BAO was also allowed to use all benchmark queries for training. For Stack, we modified BAO as described in Section 2. Since BAO itself does not address the adaptation of the hint sets used, we evaluated JOB and TPC-H with the BAO default hint sets. As shown in Table 3, BAO does not reach the potential offered by hints, even if the relevant hint sets are taken as demonstrated for Stack. In contrast, *FASTgres* approaches close proximity to the optimal hint set performance. In some cases, *FASTgres* does not fully exploit its potential, as the number of queries may not be sufficient for adequate model training or due to featurization collision (i.e., two queries mapping to the same featurization but having different optimal hint sets), though not actively observed. We also ran these experiments on our second machine with a Coffee Lake CPU and NVMe SSD storage. In this case, the resulting benchmark times are different, but the achieved

Table 3: Evaluation results for exploiting the potential.

Benchmark & Configuration		PG 12.4	PG 14.6
Stack	opt. hint set (speedup)	7,234s (2.6x)	3,561s (2.5x)
	BAO (speedup)	14,666s (1.29x)	n/a
	FASTgres (**speedup**)	7,760s (**2.5x**)	3,649s (**2.49x**)
JOB	opt. hint set (speedup)	87s (2.3x)	79s (2.3x)
	BAO (speedup)	243s (0.84x)	n/a
	FASTgres (**speedup**)	88s (**2.3x**)	95s (**1.94x**)
TPC-H	opt. hint set (speedup)	119s (3.7x)	110 (1.3x)
	BAO (speedup)	388s (1.15x)	n/a
	FASTgres (**speedup**)	137s (**3.25x**)	127s (**1.13x**)



(a) Stack on PG v12.4. **(b) Stack on PG v14.6.** **(c) JOB on PG v14.6.**

Figure 9: Evaluating the influence of the training set size. BAO only runs on PG v12.4.

speedups are comparable to the shown speedups on the Skylake machine. Thus, we can conclude that *FASTgres* is able to harness the offered potential of query hinting for the considered benchmarks on two different PG versions on two hardware environments.

6.2 Influence of the Training Set

Train-to-represent, as discussed above, is not a realistic scenario as future incoming queries can not be known beforehand. To overcome this shortcoming, only a subset of the workload queries may be used for training. In the second set of experiments, we evaluated the influences of training set size by dividing the workload queries into a *training set* and a *testing set*, as defined by a percentage. For example, a split of 80-20 means that 80% of the queries are used for training and the remaining 20% are used for testing with the learned models. To generate such a split, training queries are randomly sampled from the workload, leaving the unsampled queries for the test set. For a test set, we used the predicted *FASTgres* hint set for optimization and execution. Afterward, we determined the speedup regarding an execution with the default configuration of the optimizer. To evaluate the robustness of *FASTgres*, we run every experiment ten times with different queries in training as well as testing and report the average and standard deviation of all runs.

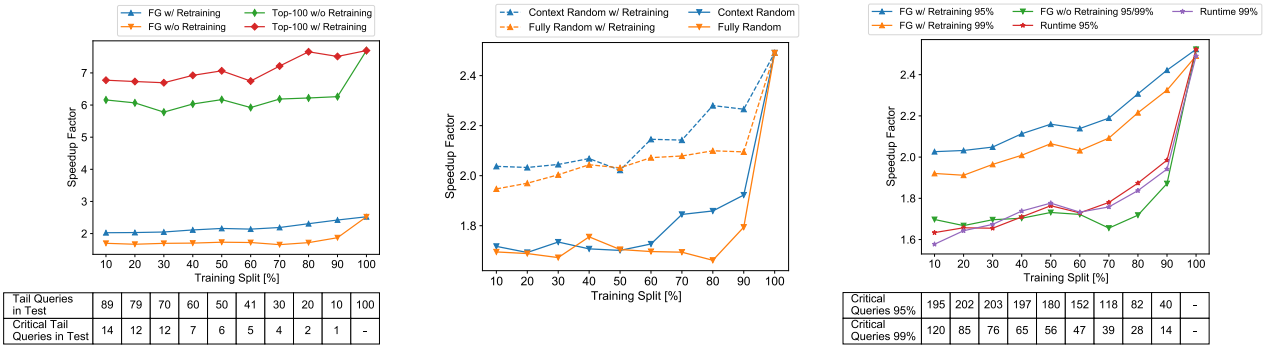
The results for Stack and JOB with increasing training fractions are shown in Figure 9. The average speedups are represented by lines, i.e., solid orange lines denoted as FG w/o Retraining, while deviation is shown as shaded regions around the average. In each diagram, the red line marks the optimal speedup as the theoretical optimum with regard to the optimal hints. The black line corresponds to the benchmark runtime with the default optimizer configuration. For Stack and PG 12.4 (cf. Figure 9a), we added BAO as our main competitor. BAO slightly accelerates Stack for all splits, but *FASTgres* already performs better with few training queries on Stack with a higher speedup than BAO. For larger training fractions, the *FASTgres* speedup increases slightly with the exception of Stack on PG 14.6 (cf. Figure 9b). For JOB on PG 14.6 (Figure 9c), we observe that the *FASTgres* speedup is lower for less training fractions while increasing with growing training fractions. Notably, we do not reach the fully possible speedup for JOB since the number of queries is not representative enough for the high number of occurring contexts. The same applies to TPC-H. Nevertheless, the standard deviations show a robust prediction behavior. We can conclude that *FASTgres* leads to significant and robust speedups even with a small subset of the workload queries in training.

So far, the results summarize over all workload queries and therefore do not allow any conclusions to be drawn about tail latency queries. The speedups of tail latency queries – green line (denoted as Top-100) – are explicitly illustrated in Figure 10a. The starting point are the 100 slowest Stack queries according to the default optimizer configuration. As shown in Figure 10a, the average speedup for the tail latency queries is much higher than for all other queries. The first row in the table below the diagram in Figure 10a depicts the average number of tail queries in the testing set. As expected, the number decreases with increasing training fraction since more tail queries are used for training. Thus, we can conclude that *FASTgres* accelerates tail latency queries especially well, even if only a small subset of the workload queries is used in training.

However, we notice that *FASTgres'* speedup gain does not continually increase with growing training fractions. From our point of view, the reason for this behavior is that not all contexts are equally represented by the random choice of training queries. To tackle this issue, we use stratified sampling, as presented in Section 4.3. Figure 10b shows the impact of this approach for Stack on PG 14.6. In contrast to an entirely random choice (i.e., solid orange lines denoted as Fully Random) our stratified sampling (i.e., the solid blue line denoted as Context Random) leads to a steady gain in average speedup gain with increasing training fractions. Thus, we can conclude that our stratified sampling improves our model quality leading to higher speedup gains.

6.3 Influence of Granularity Level

So far, we built *FASTgres* with the finest granularity $FASTgres_{fine}$, as introduced in Section 4.1, and learned a separate model for each context. In the third experiment, we also investigated *FASTgres* using the coarsest granularity level $FASTgres_{coarse}$. There, we used only one context for all workload queries. For Stack, we observe that our divide-and-conquer approach is always beneficial, as exemplified in Table 4. Here, the achieved speedup with contexts is 10% higher than the speedup without contexts for a 10-90 split. This observation applies to Stack in general since there are enough



Tail Queries in Test	89	79	70	60	50	41	30	20	10	100
Critical Tail Queries in Test	14	12	12	7	6	5	4	2	1	-

Critical Queries 95%	195	202	203	197	180	152	118	82	40	-
Critical Queries 99%	120	85	76	65	56	47	39	28	14	-

(a) Speedup of the tail latency queries. **(b) Fully random vs. stratified sampling.** **(c) Impact of retraining.**

Figure 10: Further evaluation details for Stack on PG v14.6.

Table 4: Comparing *FASTgres* with and without contexts for a 10-90 split using the Stack benchmark on PG 14.6.

Speedup *FASTgres* w/o Contexts	Speedup *FASTgres* w/ Contexts	Rate of Improvement
1.65	1.81	10%

queries per context. In contrast, *FASTgres* without contexts performs better for JOB than *FASTgres* with contexts as illustrated in Figure 9c (i.e., dotted orange line denoted as FG w/o Contexts). The same applies to TPC-H. Due to the high number of contexts and the very small number of queries per context, it is beneficial to learn a single model suiting all queries for JOB and TPC-H because the GB model can generalize better with divergent training data than with too little training data.

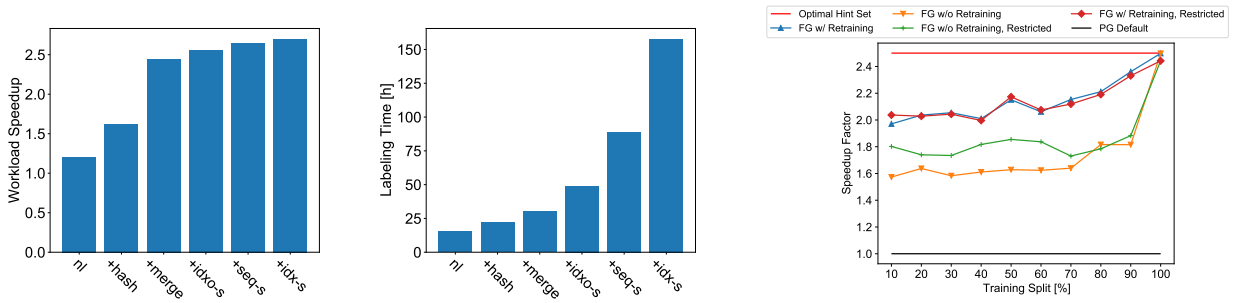
6.4 Influence of Retraining

A key feature of *FASTgres* is runtime maintenance. As described in Section 5, abnormally long-running queries with predicted hint sets are detected using a per-context timeout. These queries, also called critical queries, are used for retraining as they do not fit into the previously observed runtime behavior within a context. The reason for doing so lies within the possibly enormous misprediction penalty a hint set can incur. In the fourth experiment, we repeated the second experiment for training set influences, but this time with runtime maintenance enabled. The tables below the diagrams in Figure 9 show the number of critical queries per training fraction using a 99% timeout percentile. Again, the numbers are averaged over ten runs. Since the effect of retraining on the speedup depends on the query order, we first show the maximum possible speedup in the diagrams in Figure 9. For this purpose, we include the critical queries in the training set. Our retraining (i.e., solid blue lines denoted as FG w/ Retraining in Figure 9) leads to a significant speedup gain, with the number of critical queries decreasing as the training fraction increases. Overall, *FASTgres* discovers the critical queries at runtime to increase the quality of the model in terms of robustness. This is valid for *FASTgres* with contexts and without context, as visible in Figure 9c (i.e., blue lines denoted as FG w/ Contexts, Retraining and FG w/o Contexts, Retraining). Furthermore, considering JOB, the speedup can be increased with a few important retrainings at runtime.

However, this is only theoretically the maximum possible gain since no runtime overhead has been considered so far. The overhead includes (i) the critical queries that are aborted by the per-context timeout and therefore need to be re-executed, (ii) the discovery of the optimal hint set, and (iii) the training of the corresponding context model that needs to be adjusted. We simulated a possible strategy where the overhead should be as low as possible. Each detected critical query is aborted and triggers an asynchronous process where the optimal hint set is determined and the corresponding model is adjusted. Thus, the optimal hint set is determined in a highly parallel manner. In the meantime, incoming queries continue to be processed by the unmodified original model. After the model is retrained, it replaces the existing model and the execution of the critical query is triggered again. In this strategy, the per-context critical query timeout produces an additional overhead and decreases the speedup, as shown in Figure 10c. However, this decrease is not dramatic because retraining leads to an overall speedup gain. Lastly, Figure 10c also displays results for a 95% percentile timeout enforcing more frequent retraining phases due to additional detected critical queries. Moreover, having more retraining phases leads to higher speedup gains.

6.5 Influence of Set of Hints

In the training phase, the optimal hint sets for each query within the training set need to be determined. A naïve approach is to exhaustively search every hint set (i.e., 64 hint sets for our six chosen hints) and select the best for every query. However, this approach is not feasible since some hint sets may lead to extreme query response times. To overcome this challenge, we presented our smart labeling approach in Section 4.3. In general, our approach leads to reasonable average labeling times per query, as shown in Table 2, where our chosen hint sets are evaluated sequentially. To show the benefit of smart labeling, we simulate an optimized naïve approach with enabled smart labeling using the response time under the optimizer's default configuration as our initial timeout. Table 2 shows that the dynamic timeout adjustment leads to an improvement with an average speedup gain ranging from 1.1 to 6.4 for Stack. Especially for the four largest contexts, our smart labeling time is still in the range of seconds. The exceptions are the two small contexts C9 and C10, where the query response times are generally high. Overall, our smart labeling shows an efficient average labeling time of 44s per query for Stack on PG 14.6, after which the optimal hint set is known. A further optimization would



(a) Speedup potential. **(b) Labeling time.** **(c) Impact on resulting speedup.**

Figure 11: Evaluating the influence of hint combinations for Stack on PG v12.4.

be to run the hint set evaluation in parallel, which is beyond the scope of this contribution. We observe the same characteristics for JOB and TPC-H. However, due to the large number of contexts (33 for JOB and 22 for TPC-H) and the sparse query distribution, we refrain from a detailed display.

In general, labeling queries is a computationally expensive task with an exponential search space (2^k hint sets for a set of k Boolean hints), even with our smart labeling strategy. To tackle this challenge, a solution is to reduce the search space to the most impactful hints. To investigate this solution, we repeated our experimental analysis from Section 2 with an increasing number of hints starting with one hint – `nested loop (nl)`. Then, we consecutively added hints in the order of `hash join (hash)`, `merge join (merge)`, `index-only scan (idxo-s)`, `sequential scan (seq-s)`, `index scan (idx-s)`. Figure 11 shows (i) the speedup potential for the different sets of hints on the full Stack workload and (ii) the labeling runtimes on the lowest possible amount of data required (10%) for stable prediction from *FASTgres*. Every bar in both plots corresponds to a set of hints where one hint was added to the set of hints compared to the previous bar. From Figure 11a, we observe that the three hints for the join operators have the most impact. This is consistent with [8] claiming that join operators are most beneficial. However, every additional hint, including the ones for scans, positively impacts the speedup, each with a different margin. This is important considering Figure 11b. There, the marginal improvement in speedup does not justify the extreme amount of time spent in labeling when we include the three hints for scans. Notably, we observe that for multiple arbitrary but fixed hint orders, the speedup gain per hint naturally fluctuates due to the different orders while keeping its tendency.

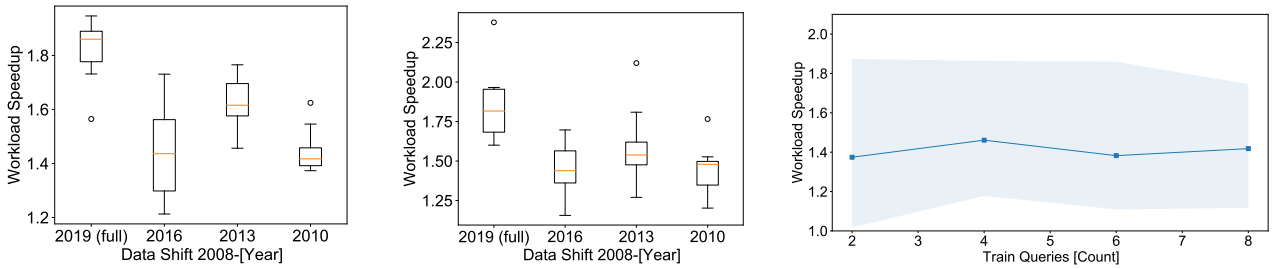
Based on this insight, we repeated our experiments for increasing training fractions with and without retraining using only the three hints for the join operators. The achieved results for Stack on PG v12.4 are illustrated in Figure 11c. The resulting speedups for the restricted set of hints are unsurprisingly higher than for all six hints for *FASTgres* without retraining. When retraining is enabled, they do not vastly differ as the hints leading to the most speedup increase are included in either case. We also observed a similar behavior for JOB and TPC-H. Thus, we deem only using a restricted amount of hints feasible. In our experimental setup, focusing on three join hints reduces the labeling time by 75% compared to the labeling for six hints. This is expected as hint restriction exponentially decreases the search space, opening up an interesting research direction.

6.6 Influence of Data and Workload Drift

So far, we have shown that *FASTgres* is capable of predicting hint sets appropriately on a given workload. However, model staleness is an issue that can possibly deteriorate prediction performance enormously, especially in light of changes in data and queries.

Data Shift: To investigate data shift, we use Stack as a representative example. Stack considers data between the years 2008 and 2019. We constructed our data shift experiment as follows. From 2019 downward, we consider a data shift as a three-year gap. Thus, we obtain multiple data versions of Stack, each considering the years 2008 – 2019 (i.e., the full data set), 2016, 2013, and 2010. Therefore, the smallest data set only consists of two years of data instead of eleven. The evaluation results are shown in Figure 12a and 12b for training-testing splits 10-90 and 50-50. These figures depict the data shift, considering workload speedups along the y-scale. Since we evaluated every experiment ten times, the results are displayed as boxplots. Notably, every data shift experiment speedup is obtained by training *FASTgres* on the reduced data and evaluating the final speedup on the whole data. By doing so, we measure the robustness of *FASTgres* trained on old data towards the up-to-date one. *FASTgres* performs well on the full data set while deteriorating in performance once trained on fewer data. However, such deterioration is unavoidable to a certain extent, as future data distribution is not an easily predictable task. Nevertheless, *FASTgres* always outperforms the PG default optimizer configurations, even with only two years of data information. Thus, we conclude that *FASTgres* shows robust behavior even when trained on stale data.

Workload Drift: This scenario emerges when obtaining a previously unseen query that behaves differently than the previously observed one. As *FASTgres* deploys a context-wise solution, two scenarios have to be considered. Firstly, an unforeseen query can be classified into an existing context. In this case, *FASTgres* is able to predict a hint set that is then used for query evaluation. If the execution behaves abnormally, it is labeled and used for retraining, as explained in Section 5. Secondly, if a new query does not fit into an existing context, a new context has to be created. For such a scenario, we conducted an experiment to investigate the minimal amount of context queries needed to handle a new context adequately. Figure 12c shows the result over all Stacks contexts averaged across ten runs. Here, we train *FASTgres* on merely a few queries per context, as shown on the x-axis. Afterward, we evaluate every query of the remaining workload. The resulting speedup is displayed along the y-axis. Moreover, we shaded the minimum and



(a) Data drift with 10% training queries. **(b) Data shift with 50% training queries.** **(c) Workload drift**

Figure 12: Evaluating the influence of data and workload drift for Stack on PG v12.4.

maximum speedup of each x-axis entry along the graph. We obtain two major insights: (1) we only need two queries for each new context to obtain speedups that outperform the default setting of PG, and (2) even though speedups naturally fluctuate while using little training data, we obtain feasible performance. Thus, we can conclude that *FASTgres* is robust towards data and workload shift, performs well on highly stale data, and also performs competitively while only using minimal amounts of context queries.

7 RELATED WORK

Query optimization of SQL queries has been a research topic for decades, but it is still not solved [3, 13]. According to [2], query optimization depends on the accuracy of cardinality estimates, particularly for intermediate results sizes. However, traditional cardinality estimation techniques frequently rely on basic heuristics that may assume predicate independence and uniform distribution of attribute values [13]. To overcome this issue, various sophisticated techniques have been proposed. For example, DB2's optimizer LEO integrates query feedback to account for estimation errors [27]. However, it is unable to capture arbitrary join predicate correlations. More recent work investigates sampling [5, 14, 21, 33] or computationally intensive sketches [1, 9, 11] to achieve precise cardinality estimates. A common disadvantage of these approaches is that they must be tightly integrated into the optimizer since they affect one of the internal core components. In contrast, *FASTgres* does not need to be integrated because it assumes the optimizer as a black box, and *FASTgres* is decoupled from cardinality estimation.

With the influence of ML, query optimization solutions are increasingly built by replacing core optimizer functionalities with learned models. The most apparent application area for ML in query optimization is cardinality estimation. [10, 16, 23, 28, 30] are recent supervised learning approaches for cardinality estimation or query costs. A disadvantage of these approaches is that extensively labeled training data is required and, unlike *FASTgres*, all queries must be executed extensively to retrieve all cardinalities.

Beyond cardinality estimation, other approaches apply reinforcement learning for holistic query plan optimization [12, 18, 19]. For example, [31] try to avoid using expert optimizers and cost models in their approach called BALSA. Here, partial query plans and their resulting latency are learned by a neural network in a two-stage manner. In contrast, [19] provide input for a model using query and plan encoding in their model NEO. The plan encoding is provided by an expert optimizer that generates a query plan tree. The result is then used as an input for a TCN that outputs a query execution time prediction. Both approaches demonstrate improvements to mean query performance, but only after a long training period. The most recent and most relevant ML approach using reinforcement learning with respect to *FASTgres* is BAO [18]. We already discussed and evaluated BAO in Sections 2 and 6.

Moreover, [8] presents a Case-Based-Reasoning approach called TONIC to improve QEPs with learned physical join operators. The input of TONIC is a QEP determined by a cost-based optimizer, while the output is a QEP with the same join order but with learned physical operator selections. For this purpose, TONIC collects, for each executed query, the QEP with a summary of the exact costs for the used operators in a case base. For each incoming query, its QEP is retrofitted by TONIC using a stored QEP from the case base. However, the performance benefit with a cost-based optimizer is limited since the QEP, especially the join order, is not changed by TONIC. Thus, TONIC is reactive, while *FASTgres* is proactive and does not have this limitation.

There is also related work by Yu et al. [32] labeled as hint-based query plan optimization with a hybrid learned and cost-based optimizer. However, this work has a different understanding of hints. Here, hints are prefix trees with partial join orders used as indicators for the final join order to be inferred by a ML model. *FASTgres* only uses global hint parameters and does not need to interact with the individual query structure. Additionally, the model-based join ordering and plan selection are deeply integrated into the database system. Our approach is not dependent on a system implementation but only on the exposed hints. Nonetheless, the authors also recognize the importance of contexts (called *templates*) and they also use the PG standard optimizer as a fallback solution.

8 CONCLUSION

Despite many years of research, a clear picture has not yet emerged of the extent to which Boolean query optimizer hints for physical operators can be used to optimize analytical queries. Therefore, we systematically evaluated this aspect for different PG versions and our evaluation shows that optimizer hints offer great acceleration potential. However, *query hinting* is a non-trivial challenge with different optimal hint set distributions per workload, optimizer, and hardware environment. To unlock this potential, we present *FASTgres*, a lightweight, learning-based, context-aware classification strategy for predicting hint sets directly for incoming queries. Generally, *FASTgres* substantially improves benchmark response times, where the traditional cost-based query optimizer is just steered with predicted learned hints for each query.

REFERENCES

[1] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *SIGMOD*. 18–35.

[2] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *SIGMOD*, Alberto O. Mendelzon and Jan Paredaens (Eds.). 34–43.

[3] Surajit Chaudhuri. 2009. Query optimizers: time to rethink the contract?. In *SIGMOD*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). 961–968.

[4] Surajit Chaudhuri, Umeshwar Dayal, and Vivek R. Narasayya. 2011. An overview of business intelligence technology. *Commun. ACM* 54, 8 (2011), 88–98.

[5] Yu Chen and Ke Yi. 2017. Two-Level Sampling for Join Size Estimation. In *SIGMOD*. 759–774.

[6] Transaction Processing Council. 2023. TPC-H. https://www.tpc.org/tpch/. Accessed: 2023-03-25.

[7] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2021. Simplicity Done Right for Join Ordering. In *CIDR*.

[8] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2022. Turbo-Charging SPJ Query Plans with Learned Physical Join Operator Selections. *Proc. VLDB Endow.* 15, 11 (2022), 2706–2718.

[9] Yesdaulet Izenov, Asoke Datta, Florin Rusu, and Jun Hyung Shin. 2021. COMPASS: Online Sketch-based Query Optimization for In-Memory Databases. In *SIGMOD*. 804–816.

[10] 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*.

[11] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating Cardinalities with Deep Sketches. In *SIGMOD*. 1937–1940.

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

[13] 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.

[14] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR*.

[15] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668.

[16] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality estimation using neural networks. In *CASCON*. 53–59.

[17] R. Marcus. 2022. Bao Postgres extension. https://github.com/learnedsystems/BaoForPostgreSQL. Accessed: 2023-03-01.

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

[19] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718.

[20] Llew Mason, Jonathan Baxter, Peter L. Bartlett, and Marcus R. Frean. 1999. Boosting Algorithms as Gradient Descent. In *Advances in Neural Information Processing Systems 12, NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999*, Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller (Eds.). The MIT Press, 512–518.

[21] Guido Moerkotte and Axel Hertzschuch. 2020. alpha to omega: the G(r)eek Alphabet of Sampling. In *CIDR*.

[22] MySQL. 2023. Optimizer Hints. https://dev.mysql.com/doc/refman/8.0/en/optimizer-hints.html. Accessed: 2023-03-01.

[23] Magnus Müller, Lucas Woltmann, and Wolfgang Lehner. 2023. Enhanced Featurization of Queries with Mixed Combinations of Predicates for ML-based Cardinality Estimation. In *EDBT*. 273–284.

[24] Oracle. 2023. Optimizer Hints. https://docs.oracle.com/cd/B12037_01/server.101/b10752/hintsref.htm. Accessed: 2023-03-01.

[25] PostgreSQL. 2023. Query Planning. https://www.postgresql.org/docs/current/runtime-config-query.html. Accessed: 2023-03-01.

[26] SQL Server. 2023. Hints (Transact-SQL) - Query. https://learn.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query?view=sql-server-ver16. Accessed: 2023-03-01.

[27] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *VLDB*. 19–28.

[28] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319.

[29] Lucas Woltmann, Patrick Damme, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2023. Learned Selection Strategy for Lightweight Integer Compression Algorithms. In *EDBT* (Ioannina, Greece) *(EDBT 2023)*. Ioannina, Greece, 13.

[30] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. 2019. Cardinality estimation with local deep learning models. In *aiDM@SIGMOD 2019*. ACM, 5:1–5:8. https://doi.org/10.1145/3329859.3329875

[31] 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*. 931–944.

[32] 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. https://www.vldb.org/pvldb/vol15/p3924-li.pdf

[33] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *SIGMOD*. 1525–1539.