Optimizing Query Processing in PostgreSQL Through Learned Optimizer Hints

Jerome Thiessat[1] Lucas Woltmann[1] Claudio Hartmann[1] Dirk Habich[1]

Abstract:

Query optimization in database systems is a crucial issue and despite decades of research, it is still far from being solved. Nowadays, query optimizers usually provide hints to be able to steer the optimization on a query-by-query basis. However, setting the best-fitting optimizer hints is challenging. To tackle that, we present a learning-based approach to predict the best-fitting hints for each incoming query. In particular, our learning approach is based on simple gradient boosting, where we learn one model per query context for fine-grained predictions rather than a single global context-agnostic model as proposed in related work. We demonstrate the efficiency as well as effectiveness of our learning-based approach using the open-source database system PostgreSQL and show that our approach outperforms related work in that context.

Keywords: Query Optimization; Hint Set Prediction; Gradient Boosting

1 Introduction

Every database system features a query compiler that converts each incoming declarative SQL query into a query execution plan (QEP). The most important component of such a query compiler is the query optimizer. The task of this optimizer is to determine the most efficient QEP. Despite decades of research activities, query optimization is still far from being solved [Le15]. According to [Ch98], the most challenging issues for the optimization of complex SQL queries are: (i) finding a good join order and (ii) selecting the best-fitting physical join implementation for each join within the chosen join order. To solve these challenges, a traditional query optimizer uses three components: the enumerator which spans – according to the relational algebra – the search space of all possible QEPs, the cost model to assess the cost of any given QEP prior to its execution, and the cardinality estimator which delivers the size of intermediate results and base tables as most crucial input to the cost model.

Such a traditional query optimizer can be found in open-source database systems, e.g., PostgreSQL [Po]. However, a disadvantage of this traditional optimizer approach is that the determined QEP for a query can vary widely in quality [Le15]. The quality variance possibly originates from miss-predicted intermediate results from PostgreSQL's cost estimator and

[1] Technische Universität Dresden, Dresden Database Research Group, 01062 Dresden, Germany,
{jerome.thiessat,lucas.woltmann,claudio.hartmann,dirk.habich}@tu-dresden.de

doi:10.18420/BTW2023-74

	PostgreSQL w/ default hints	PostgreSQL w/ **our *learned hint approach***
Stack-Benchmark	5,445.78 sec	2,802.54 sec

Tab. 1: Workload execution times of the real-world Stack benchmark [Ma22] with default and our learned optimal physical operator hints (PostgreSQL v14.2). More details in Section 4.

PostgreSQL falling back to a genetic optimizer upon surpassing a certain amount of joins in a query. To overcome this issue, PostgreSQL provides a set of well-defined optimizer hints to steer the optimizer on a query-by-query basis. For instance, the usage of the physical join operator `hash join` can be enabled or disabled using a specific hint. In general, PostgreSQL features six Boolean hints for physical operators; three for joins and three for scans. In the default setting, all six physical operator hints are activated to allow the optimizer's enumerator to span the largest possible search space.

To show the significance of hinting, Table 1 compares the workload execution times of the real-world Stack benchmark [Ma22] for PostgreSQL using (i) the default hint setting and (ii) our *learned hint approach* for the six physical optimizer hints. As shown, the utilization of our *learned hints* dramatically reduces the workload execution time. We achieve this benefit by learning a simple gradient boosting model per query context for fine-grained hint predictions. Moreover, our *learning component* is intentionally designed as a separate loosely-coupled component for PostgreSQL to guarantee broad applicability for different PostgreSQL versions. To sum up, our contributions in this paper are:

- In Section 2, we introduce preliminaries and describe the related work in this context.
- Based on these considerations, we describe the key features of our *context-aware learning approach* for hint sets in Section 3.
- Then, we present selected evaluation results to show the efficiency and effectiveness of our approach compared to state-of-the-art and related work in Section 4.

Finally, we conclude our findings with a short summary in Section 5.

2 Preliminaries and Related Work

Fundamental for our contribution is the procedure of hinting the PostgreSQL query compiler. PostgreSQL offers various hints as planner method options[2]. In the following, we use the terminology *hint* as PostgreSQL's planner methods options, *hint set* as a combination of hints, and *hinting* as the procedure of setting the planner method options in PostgreSQL accordingly. Hinting in PostgreSQL is syntactically a trivial task and can be realized by e.g., `set enable_hashjoin = false;` to disable hash joins as a prefix annotation to an SQL query. As already stated in Section 1, we focus on the six Boolean hints that are considered by PostgreSQL's planner[3] for physical operators. These hints consider three joins, i.e., *hash*,

[2] https://www.postgresql.org/docs/current/runtime-config-query
[3] https://www.postgresql.org/docs/14/planner-optimizer.html

nested-loop, and *merge join* as well as three scan operations, i.e., *index*, *sequential*, and *index-only scan*. These hints only allow to enable or disable the corresponding physical operators which however influence the whole optimization procedure.

As clearly demonstrated in [Ma22], these six physical operator hints can be efficiently used to steer the query optimization to produce more efficient QEPs, but hinting is a challenging task in general. To tackle that challenge, [Ma22] proposed a learning-based approach called BAO – the bandit optimizer, which is the most relevant related work for our approach. From a high-level perspective, BAO learns a mapping between an incoming query and the optimizer hints the query optimizer should use for that query using reinforcement learning. For that, BAO treats each hint set as an arm in a contextual multi-armed bandit problem and learns a single model that predicts which hints will provide the best run-time for an incoming query. In general, BAO works as follows: For every SQL query, the underlying PostgreSQL query optimizer produces n QEPs; one for each hint set. Afterwards, each QEP is transformed into a vector tree and the resulting vector trees are fed into a tree convolutional neural network (BAO's single model) predicting the execution time of each QEP. The QEP with the least predicted execution time is finally selected for execution. Once the QEP is executed, the selected QEP and the real execution time is added to BAO's experience. These experiences are used to periodically retrain the single model.

To the best of our knowledge, BAO is the only work that relates closely to our challenge of predicting hint sets for incoming queries. However, BAO has the following shortcomings. Firstly, BAO uses a single *global* model across all incoming queries to predict hint sets. This does not allow for fine-grained nuanced predictions for queries that differ only marginally, for example in predicates. Secondly, BAO predicts hint sets *indirectly* by predicting execution times and then inferring on the best hint set afterward. This indirection step is not necessarily beneficial, as multiple hint combinations need to be evaluated during query optimization time to determine the best-performing QEP. Lastly, BAO only investigates a reduced amount of hint sets to keep the necessary additional effort during query optimization time as low as possible. That means, for the six physical operator hints, there are $2^6 = 64$ possible hint sets, but only 25 are considered in BAO. In these 25 hint sets, the globally optimal solution might not even be included.

3 Context-Aware Hinting

To overcome the above presented shortcomings of BAO, we introduce a novel *learning-based approach* called *POSGB* to predict the best-fitting hints for each incoming query in this section. The key features of *POSGB* are: Firstly, we deploy context-sensitive models, where we build a learned model for each set of joined tables of a workload. The idea behind this is that each set of joined tables represents a self-contained context, since the queries per context are thus reasonably homogeneous with respect to the joins and differ only in the filter predicates. Using this divide-and-conquer approach allows us to predict on a fine-grained basis, where BAO uses a context-agnostic approach. Secondly, within each

context, our models follow supervised classification. This means that we directly predict a hint set from a query, rather than inferring indirectly on multiple QEPs with estimated costs. By relying on a classification task, we are also able to consider the whole search space of the 2^6 possible hint sets rather than a reduced subset. Lastly, to reduce the additional effort of using these models during query optimization time, we utilize a classical gradient boosting model within each context, rather than one single global neural network. Gradient boosting models are a learning method using a sequential ensemble of smaller models (i.e., weak base learners) typtically trained using momentum based optimization (e.g., gradient descent). Based on that, for each query during query optimization time, a context classification and a prediction with one small simple model has to be conducted. Naturally, these learned context models traverse a training phase before being able to predict an incoming query.

Training Phase: *POSGB*'s training phase follows the same principle for every context. Within each context, a set of input queries is first featurized query compiler independently (i.e., not relying on QEPs) in the fashion of [Ki19; Wo19] by encoding filter predicates. Moreover, since we deploy supervised learning, we also label each training query with the optimal hint set by exhaustively evaluating each query-hint-set combination. Notably, we also investigated the use of PostgreSQL's *EXPLAIN* functionality to reduce our labeling effort to a minimum. However, evaluating the Pearson correlation coefficient between guessed and real cost of an appropriate sample did not result in any notable correlation. Since we could not even observe any relation in the order of best-to-worst hint set, we deemed *EXPLAIN* not suitable for our task. This requires the execution of all training queries with all hint sets. Then, the hint set that produces the least execution time is determined and used as a label for each query.

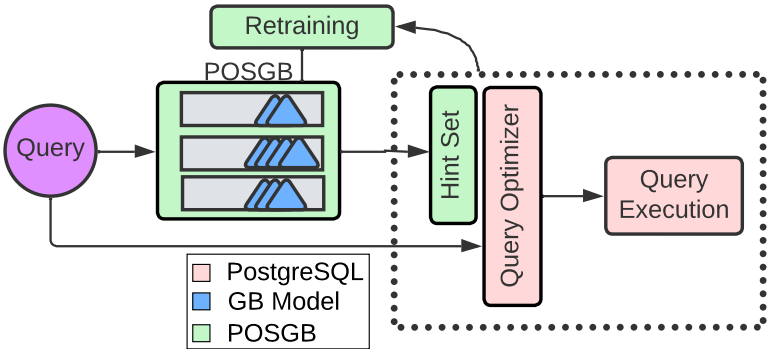

Fig. 1: Workflow of *POSGB*

Model Inference: *POSGB*'s query inference follows Figure 1. Firstly, the optimal hint set of an incoming query is predicted by *POSGB*. There, the query is assigned to a context and featurized analogously to the training phase. *POSGB* then predicts the label – a hint set – from the featurized input query. Secondly, the query with the predicted hint set is propagated to PostgreSQL for query optimization and execution.

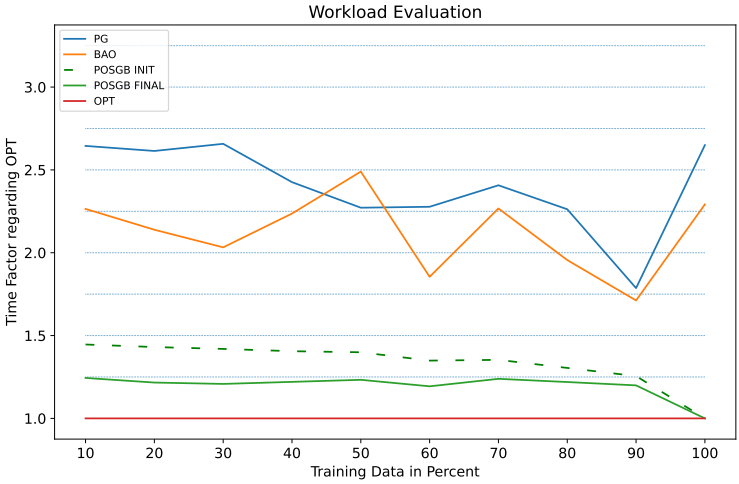

Fig. 2: Workload Evaluation of Multiple Data Splits

An important challenge in hint set prediction and generally learned query optimization, is handling unseen, badly performing queries. Generally, our model does not support detection of unusually long executing queries. For this reason, we provide a model adaptation by detecting such queries. Each query is executed with a timeout (i.e., statement timeout in PostgreSQL), where the timeout is context-sensitive and based on already executed queries within the specific context. The longest running seen query per context defines the timeout. By doing so, any query that exceeds the specified context-sensitive timeout threshold is considered as critical. On the one hand, such critical queries are canceled and re-executed with the PostgreSQL default hints. On the other hand, the critical queries are exhaustively evaluated in an asynchronous manner to determine the optimal hint set. Based on this new experience, an updated model for the specific context is trained. Upon having the newly trained model ready, the old model is exchanged by the new one.

4 Evaluation

To show the efficiency as well as effectiveness of *POSGB*, we conducted a comprehensive evaluation on a machine with an Intel Xeon Gold 6126 CPU, an ASPEED Graphics Family GPU, and 95 GiB memory. Our whole evaluation is based on the Stack benchmark, consisting of *100* GB data as well as *6191* queries from real-world examples [Ma22]. We evaluated four different scenarios: (i) PostgreSQL native with default hint setting (PG), (ii) BAO [Ma22], (iii) the initial evaluating of *POSGB* (INIT), and (iv) after retraining has been deployed (FINAL). Figure 2 shows the most important result. Depicted are training splits on the x-axis and the relative workload time factor regarding the global optimal solution on the y-axis. We determined the global optimal solution by an exhaustive search over all

queries and all hint sets. Important to note is that the Stack queries are classified into eleven contexts and that the training splits are fully random. Additionally, the *100%* split marks representative learning, which uses all data for training and testing. Notably, every but the *100%* split are vaulted (i.e., test queries are not seen by the model).

The most important results can be summarized as follows. Firstly, we observe that *POSGB* dramatically reduces the workload execution times for all training splits compared to PostgreSQL native as well as to the most related approach BAO [Ma22]. In particular, the workload times using *POSGB* are much closer to the global optimal solution. Secondly, with more training data, the workload times are continuously reduced, which is not the case with BAO as already shown in [He22] due to catastrophic forgetting. Moreover, *POSGB* performs well even for the small splits like *10%*. Notably, this performance comes with a caveat as each query has to be labeled. This sums up to roughly *12h* for the *10%* split. However, we deem this time still feasible as it is not impractical and does not interfere with the model's on-line behavior. Furthermore, representative learning shows our model is capable of learning all data that it has been confronted with, which is not the case for BAO. Lastly, our refinement of detecting critical queries shows that the model improves. Deploying this model refinement naturally implies labeling and retraining phases. However, these phases can be handled asynchronously.

5 Summary and Outlook

In this paper, we showed that proper hinting of SQL queries in PostgreSQL can have a positive impact on the overall query execution time. For that, we started by describing PostgreSQL and BAO [Ma22], the state-of-the-art approaches of predicting hint sets, and elaborated on its shortcomings, which we tackled in our novel approach called *POSGB*. *POSGB* is a learning-based approach based on simple gradient boosting, where we learn one model per query context for fine-grained predictions rather than a single global context-agnostic model as done in BAO. In our evaluation, we demonstrated that hinting with *POSGB* produces better QEPs than BAO using the Stack benchmark [Ma22]. In particular, *POSGB* is consistently better than BAO resulting in much lower workload execution times over all training splits – closer to the optimal solution found throughout labeling. Nevertheless, we have not yet reached the optimal solution and rely on hintable, sub-optimally performing query compilers, which offers enough potential for further work in this area.

References

[Ch98] Chaudhuri, S.: An Overview of Query Optimization in Relational Systems. In (Mendelzon, A. O.; Paredaens, J., eds.): Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA. ACM Press, pp. 34–43, 1998, URL: https://doi.org/10.1145/275487.275492.

[He22] Hertzschuch, A.; Hartmann, C.; Habich, D.; Lehner, W.: Turbo-Charging SPJ Query Plans with Learned Physical Join Operator Selections. Proc. VLDB Endow. 15/11, pp. 2706–2718, 2022, URL: https://www.vldb.org/pvldb/vol15/p2706-hertzschuch.pdf.

[Ki19] Kipf, A.; Kipf, T.; Radke, B.; Leis, V.; Boncz, P. A.; Kemper, A.: Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In: 9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings. www.cidrdb.org, 2019, URL: http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf.

[Le15] Leis, V.; Gubichev, A.; Mirchev, A.; Boncz, P. A.; Kemper, A.; Neumann, T.: How Good Are Query Optimizers, Really? Proc. VLDB Endow. 9/3, pp. 204–215, 2015, URL: http://www.vldb.org/pvldb/vol9/p204-leis.pdf.

[Ma22] Marcus, R.; Negi, P.; Mao, H.; Tatbul, N.; Alizadeh, M.; Kraska, T.: Bao: Making Learned Query Optimization Practical. SIGMOD Rec. 51/1, pp. 6–13, 2022, URL: https://doi.org/10.1145/3542700.3542703.

[Po] PostgreSQL: The World's Most Advanced Open Source Relational Database, URL: https://www.postgresql.org.

[Wo19] Woltmann, L.; Hartmann, C.; Thiele, M.; Habich, D.; Lehner, W.: Cardinality estimation with local deep learning models. In (Bordawekar, R.; Shmueli, O., eds.): Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019. ACM, 5:1–5:8, 2019, URL: https://doi.org/10.1145/3329859.3329875.