Steering the PostgreSQL query optimizer using hinting: State-Of-The-Art and open challenges

Kira Thiessat, Dirk Habich and Wolfgang Lehner

Abstract

In the field of query optimization, a tremendous amount of research has been delivered into fixing and adjusting existing optimizer solutions. Moreover, recent work has also shown that substantial performance gain can be achieved by setting query optimizer instructions appropriately. However, the sets of beneficial instructions may vary greatly for each query. In this contribution, we provide a state-of-the-art review on optimizer hinting and present some experimental evaluations showing hints should not be neglected during evaluation based on preliminary assumptions.

Keywords

Optimizer Configuration, Query Optimization, Search Space Traversal

1. Introduction

In the vast field of database management system (DBMS) design, query optimization has proven to be one of the backbone components of query performance. Modern query optimizers consist of the following three components [1, 2, 3]: (i) plan enumerator, (ii) cost model, and (iii) cardinality estimator. These components are hierarchically dependent. This means that the plan enumerator relies on the calculations of the cost model, which in turn relies on the cardinality estimators' results. Since traditional optimizers have shown to produce error prone estimates [4, 5], current research establishes various approaches of refining plan enumerators [6, 7], cost models [8, 9], and also cardinality estimation [10, 11, 12]. These approaches all aim at adding, correcting, or substituting existing optimizer components to obtain faster query executions.

One of the long established [13], yet only recently popular approaches [14, 15, 5], has been the targeted steering of optimizer behavior through configuration options called hints. These hints can generally be categorized in the class of enumerator approaches. However, instead of substituting or correcting an existing enumerator, hints restrict the search space of enumerators, instructing the planner to ignore certain plans. Nevertheless, hints can not only be used for restricting plan enumeration. Recent research has shown that hinting can even be used to indirectly learn optimizer cost models [14, 15].

Contribution: Our core contributions in this paper are (i) the review of the state-of-the-art research in the field of optimizer hinting and (ii) providing experimental evaluations of current challenges that need to be tackled to further advance in this research field. We do so at the example of the highly popular and open source DBMS PostgreSQL (PSQL).

Outline: In Section 2, we explain query hinting at the example of PSQL. We will go into intricate details about the query optimizer hints present in PSQL and explain which hints can be observed, how they behave, and how the systems' hinting has evolved over the latest versions. Section 3 depicts the state-of-the-art systems for utilizing these hints to harness hidden optimizer potential. We provide insights into leveraging hinting from the presented state-of-the-art in Section 4. We will then be able to obtain valuable information regarding the full potential of hinting in query optimization based on the popular Join-Order-Benchmark (JOB) [3]. Lastly, we open up the discussion in Sections 5 and 6, where we discuss possible highly valuable research directions, hinting in general, and our reflections.

2. Query Hinting in PostgreSQL

Modifying the PSQL optimizer can be done mainly by setting Boolean configuration parameters called query planner method configurations[1]. We restrict our contribution to these Boolean hints as they allow us to steer the optimizer plan enumeration phase rather than interfering with the underlying complex cost model. In their nature, these configurations are mostly instructions to force the optimizer to restrict itself to certain operations while traversing a query's plan space. However, since not all of these configurations can be forcefully switched off[2], the term hint has established [14, 5]. Additionally, hint set refers to multiple hints, and hinting to the act

35th GI-Workshop on Foundations of Databases, May 22-24, 2024, Herdecke, Germany

Kira.Thiessat@tu-dresden.de (K. Thiessat);
Dirk.Habich@tu-dresden.de (D. Habich);
Wolfgang.Lehner@tu-dresden.de (W. Lehner)
0000-0002-7223-6536 (K. Thiessat); 0000-0002-8671-5466 (D. Habich); 0000-0001-8107-2775 (W. Lehner)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1] https://www.postgresql.org/docs/, accessed: March 4th, 2024

[2] as this may hinder the optimizer to construct a query plan at all

CEUR-WS.org/Vol-3710/paper8.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2024-06-24

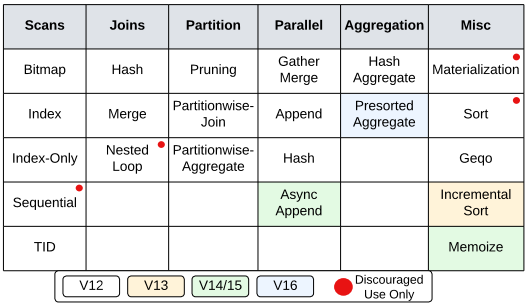


Figure 1: PSQL hint development starting from version 12.

of instructing the PSQL query optimizer with a hint set. Notably, in PSQL, hints are global, which implies that they act upon the optimization of a whole query. This means, that e.g., disabling the hint "Nested Loop Join" will disable the usage of this operator (if possible) for the entire query plan optimization phase.

Even the oldest still supported version (i.e., version 12) of PSQL has a plethora of these hinting options which may be highly valuable to investigate. A rigorous set of Boolean hints beginning from PSQL version 12 up to the latest version of this writing (i.e., version 16) can be found in Figure 1. We observe that the base amount of hints is already plentiful. While the core hints[3] of PSQL consist of six traditional hints, namely index, index-only, and sequential scan, as well as hash, merge, and nested loop join, there are multiple other options to be considered.

Bitmap scans for example scan multiple tuple pointers, which are then sorted by their physical location to allow ordered access over all locations. Especially combining multiple bitmaps is easy as multiple bitmap scans can be combined through simple logical operator combination. **TID Scans** provide fast access with the knowledge of tuple ids (i.e., physical location) of a row. Notably, tuple ids provide no identifying behavior like primary keys as their value may change during the lifespan of their respective database table.

Additional non-core hints include hints for parallel execution, aggregation, and general miscellaneous settings. Within parallel execution, **gather merge** indicates the use of the (gather) merge node within a plan that allows child nodes of a query plan, or in fact, the whole plan to be executed in parallel. While the gather node solely merges results, the gather merge node indicates that subnodes output sorted tuples which are then merged in a sort-preserving manner. The **parallel append** works similarly to the regular append when combining rows from multiple sources. However, instead of processing every child node subsequently in parallel fashion, processes are spread across multiple child nodes to process them simultaneously to allow fully parallel execution of child nodes rather than parallelism within operators of a child node. For **parallel hash**, instead of building a hash table for each process, a common hash table is shared among common processes. The last parallel hint consists of **async append**. This hint allows parallel append plans on foreign data tables, supporting sharded sources. The aggregation hints consist of hash and presorted aggregation.

Hash aggregation allows the use of hash functions for splitting, aggregating, and merging a target column. **Presorted aggregation** allows the query planner to construct plans that produce rows that are already sorted for further aggregation.

Within the miscellaneous section are five hints. **Materialization** allows to enable caching of intermediate results in memory. **Sort** naturally allows the usage of sort operations within a plan. **Geqo** decides whether to use the genetic optimizer of PSQL. This optimizer is by default switched on if the number of joins in a query exceeds twelve. **Incremental sort** enables the usage of an optimized multi-key sorting method that allows to take advantage of already sorted columns. **Memoize** encourages the use of nested loop join, as memoize caches parameterized scans for nested loop joins only.

Additionally, there are three options for usage in a partitioned table environment. However, as these partitioning hints require the building of partitioned tables, they remain mostly unused. While the ability to prune partitioned tables remains **on** as for most hints, the partitionwise hints are **off** by default as partitioning requires additional attention and is not part of the default behavior of PSQL.

Moreover, there are also some hints that cannot be completely switched off to ensure that a plan can be created. Switching these hints off highly discourages their use in the planning phase. These hints are marked in red in Figure 1. Moreover, the use of **parallel hashing** naturally requires **hash joins** to be enabled.

Now, we are able to further understand the possible implications of the presented PSQL hints. In the following, we dive deeper into the state-of-the-art that utilizes the presented hints to gain advantages during the optimization queries.

3. Query Optimization Using Optimizer Hints

While modifying the core components of an optimizer through different algorithms and machine learning models is currently investigated with great interest [8, 9, 11, 10, 12, 6, 7], the steering of optimization using optimizer hints remains to be a topic of deeper research. However, there are still some major contributors in this field.

[3] https://www.postgresql.org/docs/16/planner-optimizer.html, accessed: March 4th, 2024

BAO. The first one is the bandit optimizer (BAO) [14]. BAO is a machine learning model that uses hints to indirectly learn the optimizer's cost model. BAO builds upon a commonly used machine learning technique derived from the successful use in learned query optimization, namely reinforcement learning. In its core, BAO uses a fixed number of hint sets, which are used to obtain different PSQL EXPLAIN plans during optimization. There, each of the resulting plans is used as an input for a Tree Convolutional Neural Network (TCNN). The TCNN of BAO is built similarly to a regular convolutional neural network with an additional initial tree flattening layer to transform the input tree. The output of the TCNN is an estimated query cost value that is used to determine which of the input explain plans to use. This allows BAO to implicitly use the hints through PSQLs explain functionality. Lastly, Thompson sampling is used to enable the reinforcement learning character of BAO, which balances exploration and exploitation by guiding the data, upon which BAO is trained.

However, there are some considerations to account for when using BAO. First, BAO uses only the six main hints of PSQL, which is merely a subset of the already presented hints. Even inside these six hints (i.e., $2^6 = 64$ hint set combinations), only a subset of five hint sets are feasible enough such that optimization time does not start to exceed the actual runtime. Moreover, BAO uses an indirect approach that necessitates evaluating multiple query plans before being able to decide on which to choose. Naturally, doing so does not scale well with increasing numbers of hint sets used. Lastly, BAOs input plans rely on PSQL EXPLAIN plans, which have been shown to produce notoriously error prone estimates [4, 5].

Autosteer. As a successor to BAO, the approach of Autosteer [15] was developed. Just like BAO, Autosteer is a learned cost model by using query hinting. In this approach, the existing challenge of selecting hint sets, which were solved by manually applying expert knowledge in BAO, were tackled. Their hint set traversal algorithm consists of iteratively combining efficient query spans [16]. For each combination, they determine the query's runtime and decide whether to keep the combination or not, based on the default query execution time. While this algorithm leverages on hint set traversal from BAO's fixed hint sets, there are still some challenges.

First, a rather large assumption is made by postulating that not all hint sets are beneficial. While this may hold true for certain workloads, in DBMSs and with specific versions, the benefits of hints cannot be deemed static. Additionally, a key assumption is the prior knowledge of query spans (i.e., efficient hints) of a query. Such knowledge is usually obtained only through rigorous analysis of query workloads or even individual queries, depending on their versatility. As it is infeasible to have such information a priori, and knowledge from previous queries do not necessarily reflect future ones, we argue that determining query spans efficiently for unseen queries is a challenging and under-investigated task.

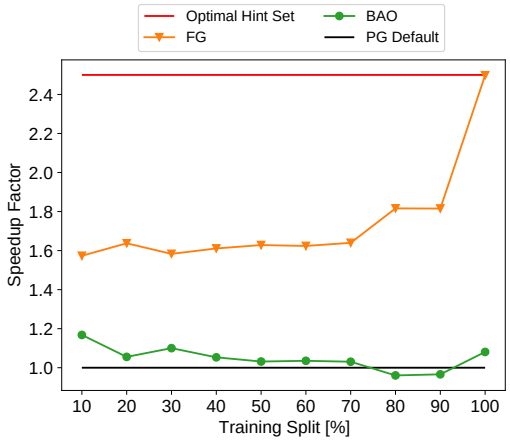


Figure 2: Evaluation of FASTgres and BAO on the real-world workload Stack-Overflow [5].

FASTgres. Lastly, there is another approach, named FASTgres [5] that breaks with the commonly used reinforcement learning approaches for learned optimizers and optimizer steering. FASTgres settles directly at the plan enumerator by restricting its search space through hinting predictions. FASTgres uses a supervised classification approach to predict the most beneficial hint set directly from incoming queries. Here, queries are predetermined into different query classes called contexts. In FASTgres, contexts can be defined in multiple granularities but are commonly distinguished by their table join groups. Within each context, a supervised gradient boosting model is used to predict a query's best hint set. This approach naturally differs from previous work by using a divide-and-conquer approach to provide locally well performing models that are robust against data and workload drifts. This robustness is achieved not only by using local models but also by providing an experience-based query anomaly detection that retrains context models once receiving abnormally performing queries. Additionally, since a supervised approach is applied, labeled data is required. Such labeled query data is obtained by using a grid search strategy with aggressive timeouts determined by the currently best query-hint-set combination.

Generally, FASTgres has shown to provide superior results with the usage of supervised learning for hint set prediction as observable in Figure 2.

While the performance of supervised classification for hint set prediction shows promising results, there are also challenges that need to be overcome. Just like BAO, FASTgres only utilizes six hints. While they traverse the

whole search space rather than only a few combinations of hints, there is still the huge challenge of scalability. Even with their provided aggressive timeout-strategy, search spaces that grow exponentially have to be evaluated fully. If we observe the Boolean hint sets from PSQL in Table 1, the hint set search space would constitute $2^{22} \approx 4.2 \cdot 10^6$ combinations, which is infeasible to be labeled in FASTgres' strategy. Having such scalability restrictions naturally leaves the question open about how much potential is neglected when labeling queries.

To depict open challenges emerging from the state-of-the-art work on hinted optimizer steering, we now investigate the potential of hinting at the example of PSQL.

4. Optimization Potential of Query Hinting

Optimizer hinting is a volatile task in its nature. There are many different influencing aspects that need to be considered such that they become intangible rapidly. Exemplary, when considering the decision of whether to use a hash join in a query or not, the influence factors can depend on the collected statistics, available memory for building hash tables, possible parallel execution, the systems hardware[4], the systems build version[5], and possibly many more. As subsets of these influence factors are subject to constant change, the proper determination of operator choice for a single query becomes a tedious task that is infeasible to fulfill properly during runtime. With these factors in mind, it is only natural to assume that, by default, every hint set can have an influence on a query. With this general assumption, we now investigate the PSQL query optimizer in further detail. For the following experiments, we evaluated - unless otherwise stated - on a PSQL Docker Image that has been properly configured for its hardware (i.e., using PG-Tune[6]) and using an ANALYZE step to build initial database statistics. Additionally, each run has been evaluated once with each query-hint-set combination being run once before measuring, ensuring properly equal pre-warming. The used hardware consists of an Intel Xeon Gold 6216 CPU (Skylake architecture) with 12 cores, 92 GiB of main memory, and 1.8 TiB of HDD storage.

As we observed in Figure 1, there are 22 hints in the latest version of PSQL (i.e., v16) at the time of writing.

Since analyzing these hints can become increasingly time consuming, rather than taking a large workload like Stack-Overflow [14] with 6191 queries, we focus on JOB that contains real world data from IMDB and a query workload of 113 analytical queries. Since JOB does not provide partitioned tables by default, the partition hints for **pruning**, **join**, and **aggregate** will be neglected in the further analysis.

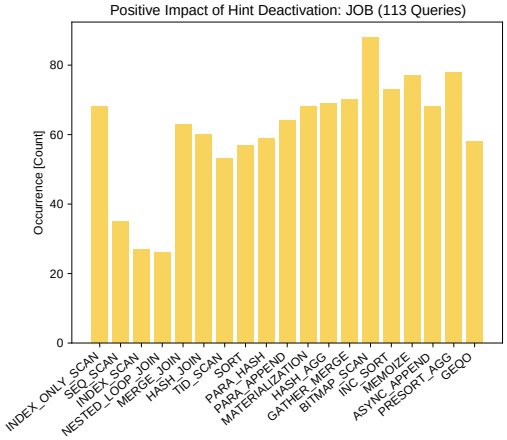


Figure 3: Counts of how many times the option of switching a hint off had a positive impact on a queries performance.

In our first experiment, we conduct an evaluation whether the option of turning off the usage of a hint pertains a positive impact on the overall runtime of a query or not. For each JOB query, the influence of each hint was measured. The results can be observed in Figure 3 for PSQL version 16.2.

Along the x-axis, every hint is displayed. The y-axis shows how many times in the whole workload a positive performance impact was noted by allowing a hint to be switched off from their default setting. Notably, every hint has at least 20 cases out of 113 in which turning them off is beneficial. In detail, allowing for nested loop joins to be switched off amounts for 26 positive impacts (i.e., 23%), while allowing bitmap scans to be switched off even amounts for 88 positive impacts (i.e., 78%). This implies that every hint on their own may have useful cases in which their impact cannot be neglected.

While these insights show that considering every hint is a sensible choice, the exact impact of these hints remains ambiguous. Results of the average speedup gained from these positive impacts can be taken from Figure 4.

Here, the y-axis displays the average speedup factor that is obtained throughout the whole workload within the positively impacting hints. Notably, the option to turn off **sequential scanning** now provides average speedups of factor six, while bitmap scans with the highest previous occurrence only provide speedup factors of just above one. These results imply that the occurrence of positive hints does not need to correlate with proper speedups across the workload.

Moreover, while speedup by factor is a valuable indi-

[4] i.e., Memory, SSD, HDD availability

[5] i.e., their efficiency in implementing the operator and underlying cost factors

[6] https://pgtune.leopard.in.ua/, accessed April 25th, 2024

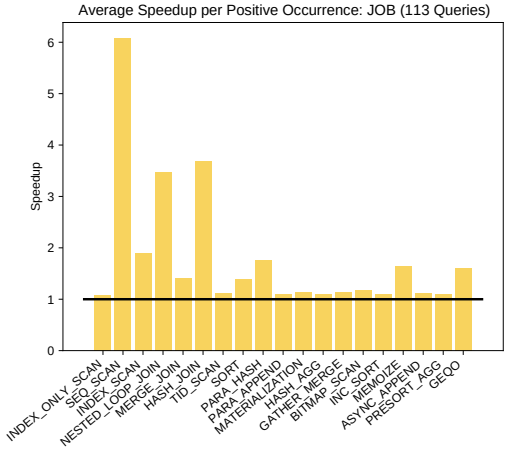


Figure 4: Workload speedup potential by factor from allowing single hints to be turned off. The black bar marks the PSQL default which is used as comparison.

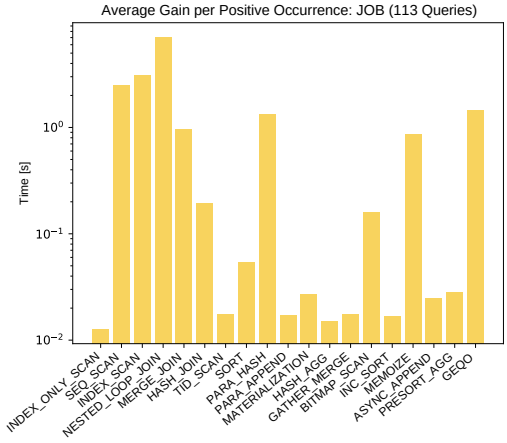


Figure 5: Workload speedup potential by total time from allowing single hints to be turned off.

cator of how much factor-wise gain we can obtain by switching individual hints on or off, another valuable insight can be achieved by looking at the absolute time impact of these hints. Results portraying the absolute gain can be observed in Figure 5.

Along the y-axis, we display the time savings that can be achieved from allowing a hint to be switched off at logarithmic scale. Again, contrary to the indications from Figures 3 and 4, we notice that fundamental time savings lie within the nested loop join, index scan, sequential scan, and merge join, which are part of the six core hints. However, we also find fundamental potential in the parallel hashing, Geqo, and memoize options.

Overall, we observe that within the JOB workload, all hints may vary in occurrence, their speedup, and their absolute time gain without indicating rules that can be followed. Our empirical studies on other PSQL versions show that even between versions, there are no rules to be delineated that could lead to inferences that can be made for future queries.

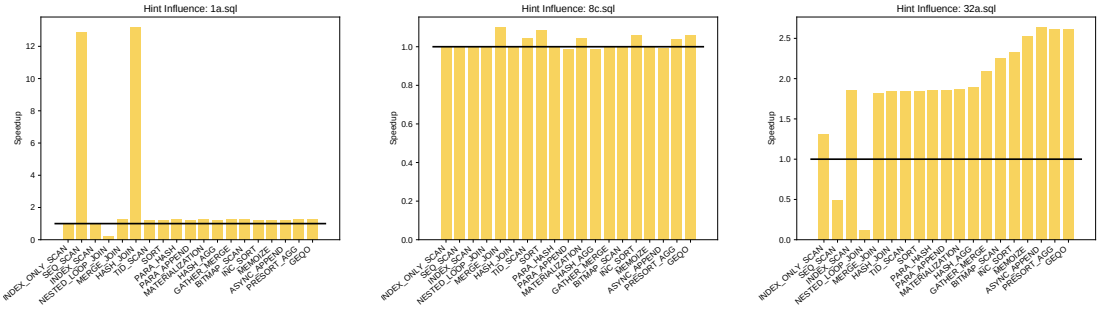
When focusing on single queries in the workload, we obtain a more detailed view about the query variety. We show an excerpt in Figure 6 due to space restrictions, while noting that the varying behavior can be observed throughout the whole workload.

We notice a variety of different behaviors. Query $1a$ from Figure 6a contains five joins and four filters of which two are wildcard filters. The three major hints are sequential scan, nested loop join, and hash join. Having potential speedup factors of over twelve indicates that the default PSQL optimizer had issues correctly determining scan and join operators, favoring sequential scans over index and index-only scans, which indicates too low selectivity estimation for scan operations. Additionally, the potential gain through hash joins indicates that the default optimizer might have estimated too high selectivity join results, leading to the choice of hash joins over nested loop or merge joins. On the other hand, disabling nested loop joins results in catastrophic deterioration of execution time, reinforcing the thought that nested loop joins are the operators that should be predominantly used in this query. Query $8c$ contains five joins and two filter predicates. Figure 6b displays a scenario, where the default optimization is already surprisingly efficient. However, even then, small improvements are still possible on multiple hints with disabling merge joins and sorting as the predominant ones. Query $32a$ contains six joins with one filter predicate. Figure 6c shows a result that is quite common in our analysis. There are a lot of hints which carry speedup potential, while some hints may lead to catastrophic execution times with respect to the default evaluation.

With these experiments, we could show at the example of PSQL and JOB, that there is no panacea for finding the best hints of a query or even workload a priori of evaluation. This leads us to the conclusion that query hinting is a complex topic that necessitates further investigation to properly infer on the usefulness of each query-hint-set combination.

5. Future Research

The potential for future research regarding optimizer hinting is manifold. As we have shown, every hint has to be considered when trying to optimize queries. However, within each query, we have seen that some hints have less influence than others. For example, an algorithmic solution with efficient hint pruning methods and possi-



(a) Single hint influence on query 1a. (b) Single hint influence on query 8c. (c) Single hint influence on query 32a.

Figure 6: Single query hint influences.

bly early stopping mechanisms that can be run without prior knowledge of efficient hints may prove promising. By radically reducing the search space, this might allow easier scalability to traverse query-hint-set combinations. Such an algorithm could evaluate each hint set on their own either from a subtractive (i.e., step-wise disabling) or additive (i.e., step-wise enabling) starting point. Either solution would provide insight into the one-ring neighborhood (i.e., neighboring hint sets that only differ by one hint being switched off or on) of changing hints from a default situation, in which either all hints are switched on or off. Additionally, further steps could be decided based on these initial results. Such further steps can include the use of stopping criteria based on previous results, the current one-ring neighborhood results, previously observed queries, and possibly many more. Lastly, even though current learned models provide reasonable speedup gains, the development of a tailored featurization for query hinting seems lucrative, as currently, only featurization options that are tailored for cardinality estimation are in use. Potentially, such a dedicated featurization method may require smaller input vectors and generalize better, depending on the chosen representation.

6. Conclusion

Lastly, we summarize our findings in this contribution. We showed that there is already valuable research in the field of query optimizer hinting that focuses on a small set of hints that they deem feasible. Additionally, we could show at the example of JOB and PSQL, that hinting is a manifold challenge that cannot be heuristically reduced to a mere subset of hints but must be considered as a whole to be able to extract all possible gains. Moreover, we showed that promising hints cannot be simply predetermined without extensive analysis and that the impact of hint sets varies between different queries and also across different PSQL versions.

Even though [15] provides an algorithmic solution for their hint set traversal problem, there are aspects that still do not suffice for a scalable traversal solution. As shown in our evaluations, the usage of a predetermined set of hints for initial evaluation, namely query spans, is not derivable beforehand and, thus, impractical.

We could show that the research field for hint set traversal offers plenty of optimization potential for future work.

References

[1] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, Access path selection in a relational database management system, in: SIGMOD, ACM, 1979, pp. 23–34.

[2] M. Perron, Z. Shang, T. Kraska, M. Stonebraker, How I learned to stop worrying and love re-optimization, in: ICDE, 2019, pp. 1758–1761.

[3] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, T. Neumann, How good are query optimizers, really?, Proc. VLDB Endow. 9 (2015) 204–215.

[4] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, T. Neumann, Query optimization through the looking glass, and what we found running the join order benchmark, VLDB J. 27 (2018) 643–668.

[5] L. Woltmann, J. Thiessat, C. Hartmann, D. Habich, W. Lehner, Fastgres: Making learned query optimizer hinting effective, Proc. VLDB Endow. 16 (2023) 3310–3322.

[6] R. Marcus, O. Papaemmanouil, Deep reinforcement learning for join order enumeration, in: aiDM@SIGMOD 2018, 2018, pp. 3:1–3:4.

[7] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, I. Stoica, Learning to optimize join queries with

deep reinforcement learning, CoRR abs/1808.03196 (2018).
[8] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, N. Tatbul, Neo: A learned query optimizer, Proc. VLDB Endow. 12 (2019) 1705–1718.
[9] Z. Yang, W. Chiang, S. Luan, G. Mittal, M. Luo, I. Stoica, Balsa: Learning a query optimizer without expert demonstrations, in: SIGMOD, 2022, pp. 931–944.
[10] M. Stillger, G. M. Lohman, V. Markl, M. Kandil, LEO - db2's learning optimizer, in: VLDB, 2001, pp. 19–28.
[11] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, A. Kemper, Learned cardinalities: Estimating correlated joins with deep learning, in: CIDR, 2019.
[12] L. Woltmann, C. Hartmann, M. Thiele, D. Habich, W. Lehner, Cardinality estimation with local deep learning models, in: aiDM@SIGMOD, 2019, pp. 5:1–5:8.
[13] D. K. Burleson, Oracle high-performance SQL tuning, McGraw-Hill, Inc., 2001.
[14] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, T. Kraska, Bao: Making learned query optimization practical, SIGMOD Rec. 51 (2022) 6–13.
[15] C. Anneser, N. Tatbul, D. E. Cohen, Z. Xu, P. Pandian, N. Laptev, R. Marcus, Autosteer: Learned query optimization for any SQL database, Proc. VLDB Endow. 16 (2023) 3515–3527.
[16] P. Negi, M. Interlandi, R. Marcus, M. Alizadeh, T. Kraska, M. T. Friedman, A. Jindal, Steering query optimizers: A practical take on big data workloads, in: SIGMOD, 2021, pp. 2557–2569.