Turbo-Charging SPJ Query Plans with Learned Physical Join Operator Selections

Axel Hertzschuch, Claudio Hartmann, Dirk Habich, Wolfgang Lehner
TU Dresden, Dresden Database Research Group
Dresden, Germany
firstname.surname@tu-dresden.de

ABSTRACT

The optimization of select-project-join (SPJ) queries entails two major challenges: (i) finding a good join order and (ii) selecting the best-fitting physical join operator for each single join within the chosen join order. Previous work mainly focuses on the computation of a good join order, but leaves open to which extent the physical join operator selection accounts for plan quality. Our analysis using different query optimizers indicates that physical join operator selection is crucial and that none of the investigated query optimizers reaches the full potential of optimal operator selections. To unlock this potential, we propose *TONIC*, a novel cardinality estimation-free extension for generic SPJ query optimizers in this paper. *TONIC* follows a *learning-based* approach and revises operator decisions for arbitrary join paths based on learned query feedback. To continuously capture and reuse optimal operator selections, we introduce a lightweight yet powerful *Query Execution Plan Synopsis* (*QEP-S*). In comparison to related work, *TONIC* enables transparent planning decisions with consistent performance improvements. Using two real-life benchmarks, we demonstrate that extending existing optimizers with *TONIC* substantially reduces query response times with a *cumulative* speedup of up to 2.8x.

PVLDB Reference Format:
Axel Hertzschuch, Claudio Hartmann, Dirk Habich, Wolfgang Lehner. Turbo-Charging SPJ Query Plans with Learned Physical Join Operator Selections. PVLDB, 15(11): 2706 - 2718, 2022.
doi:10.14778/3551793.3551825

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/axhertz/TONIC.

1 INTRODUCTION

Query optimization of arbitrary select-project-join (SPJ) queries has been a core research topic for decades, but it is still far from being solved [7, 26]. According to [6], the most challenging issues for the optimization of complex SPJ 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. A textbook query optimizer solves these challenges using three modules: First, the plan enumerator which spans – according to the relational algebra – the search space of all possible query execution plans (QEPs). Second, the cost model to assess the cost of any given query execution plan prior to its execution. And third, the cardinality estimator which delivers the size of intermediate results as most crucial input to the cost model. However, achieving reliable and accurate cardinality estimates for arbitrary joins with acceptable overhead is and remains an open challenge [27].

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551825

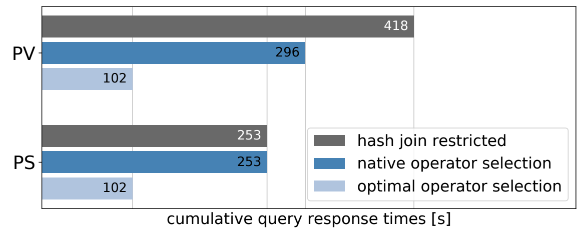


Figure 1: Join-Order-Benchmark: Cumulative query response times (also denoted as benchmark response time) for two different optimizers and three physical operator selection methods. For more details see Section 2.

Despite many years of research and to the best of our knowledge, no clear understanding has evolved yet to which extent the selection of physical join operators accounts for the plan quality of complex SPJ queries. In line with recent work [4, 9, 14, 29], we conducted a comprehensive experimental analysis using a recent open-source PostgreSQL. In our evaluation, we ran the Join-Order-Benchmark (JOB) [25] with different join orders and fine-grained physical operator selections. To determine different *good* join orders, we utilized the textbook-oriented built-in PostgreSQL optimizer denoted as *Postgres Vanilla* (*PV*) and, for instance, an alternative optimizer based on an upper bound called *Postgres Simplicity* (*PS*) [14].

We first executed the determined QEPs using only hash joins for each JOB query. As demonstrated in Figure 1, the resulting benchmark response times (execution and planning) are quite different due to different join orders, this leads to the conclusion that the join order is essential for SPJ query optimization. This effect is reinforced when physical optimization is also considered. In addition to the join order, the traditional Postgres optimizer *PV* natively selects physical join operators according to the estimated join cardinalities. Executing *PV's* native join orders and operator selections significantly reduces the query response times as shown in Figure 1. However, the overall benchmark response time is still greater than for *PS* with the restriction to hash joins. Due to the upper bound (overestimation) for intermediate result sizes, *PS* lacks a fine-grained operator selection and relies on hash joins.

To analyze the potential of optimal operator selections, we executed the following steps for *PV* and *PS*: (i) determine the join

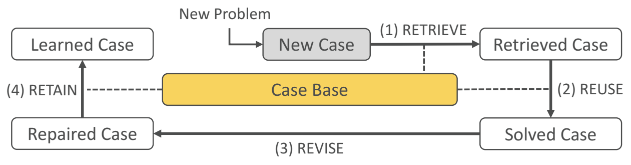


Figure 2: Case-Based-Reasoning Life-Cycle.

order for each JOB query, (ii) systematically execute each join order with all possible physical operator combinations, and (iii) extract the *optimal operator choices* for the *executions with the lowest run times*. Then, we ran the join orders of *PV* and *PS* with the optimal operator selection determined previously. As indicated in Figure 1, the optimal selection has a decisive impact as benchmark response times are reduced by a factor of 2.5 to 2.8 compared to an execution with the native operator selections. Most importantly, we observe that the different join orders of *PV* and *PS* with their respective optimal physical operator assignments lead to a similar JOB run time. Thus, we may conclude that the physical operator selection really matters and leaves room for improvement.

Contribution: This paper presents a lightweight *learning-based physical execuTiOn plaN refInement Component* called *TONIC* to turbo-charge QEPs with learned physical join operator selections. *TONIC* bootstraps an existing cost model and learns from previous query executions. *TONIC* is explicitly designed to augment any relational query optimizer. The input of *TONIC* is a QEP determined by a generic optimizer, while the output is a QEP with the same join order but with learned physical operator selections. To solve the selection problem within *TONIC*, we apply Case-Based-Reasoning (CBR), a well-established paradigm in the area of artificial intelligence. As illustrated in Figure 2, the core of CBR is the *case base*, which is a collection of previously made and stored solutions called cases. Transferred to *TONIC*, the case base is a collection of already executed join orders with a summary of the exact costs for the available physical join operators. In particular, a case based reasoner solves new problems by reusing solutions from cases in the case base. For this purpose, one or several relevant cases are collected (RETRIEVE-stage). Once a similar case is selected, the solution is adapted to solve the current problem (REUSE-stage). Transferred to *TONIC*, we match the join order of a new QEP with join orders of previous QEPs and assign physical join operators according to the stored solutions. Finally, when a new solution to the new problem is found (REVISE-stage), the new solution is stored in the case base (RETAIN-stage), thus reflecting the learning behavior. Transferred to *TONIC*, costs of physical join operators are computed and stored after query execution when exact cardinalities of intermediate results are known. Advantages of the CBR concept are: (i) no need for an elaborate training phase for initialization and (ii) continuous learning to adapt to new circumstances. However, challenges are: (i) defining the problem statement for CBR, (ii) organizing the case base to quickly find similar prior problem solutions, and (iii) adapting the prior solution to fit new needs.

Outline. To present *TONIC*, the remainder of the paper is structured as follows: We start with a problem statement using an experimental analysis to investigate the effect of physical operator selection on fixed join orders in Section 2. In Section 3, we describe our *execuTiOn plaN refInement Component* (*TONIC*) to turbo-charge QEPs with near-optimal operator choices. In particular, we introduce a novel *Query Execution Plan Synopsis* (*QEP-S*) as main ingredient of *TONIC*. While Section 3 gives a general overview of *TONIC*, Section 4 details different design considerations for the query execution plan synopsis. In Section 5, we evaluate *TONIC* in combination with different query optimizers and two real-life datasets. To show the effectiveness and applicability of *TONIC*, we also provide a detailed comparison to a recent machine learning approach. We further highlight interesting results and implications of our work for future research. After discussing related work in Section 6, we conclude the paper with a short summary in Section 7.

2 PHYSICAL JOIN OPERATORS MATTER

To demonstrate the effect of physical operator selections on plan quality for complex SPJ queries, we present results of an exhaustive experimental evaluation based on the popular *Join-Order-Benchmark* (JOB) [25]. JOB comprises 113 analytical queries with multiple joins and different complex filter predicates over real-world data that covers various challenges such as varying distinct values, correlation, and skew with a total size of 10GiB. To test a diversity of query plans – in particular join orders – and to simultaneously evaluate the impact of cardinality estimation, we investigate different query optimizer designs using: (1) *fast but error-prone cardinality estimates*, e.g., based on standard histograms, (2) *costly but precise cardinality estimates*, e.g., based on machine learning or sketch building, and (3) *cardinality estimation-free join enumeration*. In particular, we use the following instances as representatives:

(1) Postgres Vanilla (v9.6 / v12.4): A vanilla installation of the open-source disk-centric row store PostgreSQL [36].
(2) Postgres Sketch (v9.6): An instance, modified by Cai et al. [4] that provides precise cardinality estimates. It is publicly available at [12]. Based on the precise estimates, Postgres determines the join order and physical operator selection.
(3) Postgres Simplicity (v9.6 / v12.4): Postgres Vanilla enhanced with the simplistic upper bound approach of [14] without fine-grained physical operator selections.

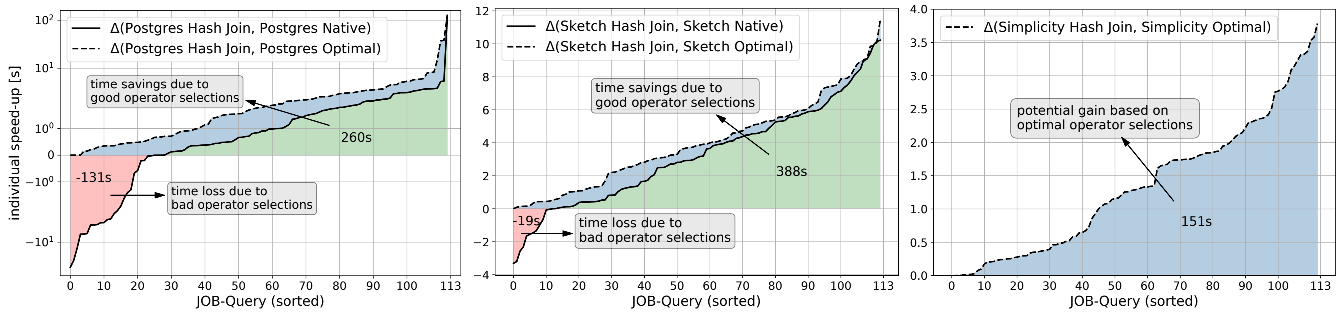
To assess the effect of physical join operator decisions, we keep the logical join order fixed and only substitute physical join operators. In particular, given the query execution plan of a generic optimizer, we compare the optimizer's native plan (*native*) to (i) an equivalent plan only using hash joins (*hash*) and (ii) to an equivalent plan with optimal physical operator selections (*optimal*). The optimal selections are determined by exhaustive execution of all possible operator combinations. Moreover, the restriction to (index) nested loop joins is not considered as multiple plans did not finish in this case – despite providing foreign-key and primary-key indices. We run all queries after a warm-up phase on a 64-bit Linux machine with a single-socket Intel Core i7-6700 CPU, 16GiB of main memory, and SSD storage. To force the execution of a particular physical join operator selection, we use the *pg_hint_plan extension* [10] and SQL hints described in [14] to bypass logical join reordering.

2.1 Results

Table 1 comprises the cumulative query response time for all 113 JOB queries using different query optimizers and different physical operator selections alternatives. Since *Sketch* is only available on

Table 1: Cumulative query response times using different query optimizers and different physical plan alternatives (hash restricted, native, and optimal). The relative speedup compares the native and optimal operator selection.

	hash join restricted	native operator selection	optimal operator selection	relative speedup
Vanilla (v12.4)	418s	296s	102s	2.8x
Simplicity (v12.4)	253s	253s	102s	2.5x
Vanilla (v9.6)	641s	did not finish	191s	∞
Simplicity (v9.6)	658s	658s	244s	2.7x
Sketch (v9.6)	622s (+1325s planning)	252s (+1325s planning)	167s (+1325s planning)	1.5x (1.1x with planning)



(a) Vanilla (v12.4) **(b) Sketch (v9.6)** **(c) Simplicity (v12.4)**

Figure 3: Distribution of sub-optimal planning decisions showing the effect of good physical operator selections.

Postgres v9.6, we report the results of *Vanilla* and *Simplicity* using both, a recent Postgres instance (v12.4), and Postgres v9.6 as well. Moreover, as *Sketch* imposes a substantial planning overhead, we break out the planning time for this approach. With regard to Table 1, Figure 3 indicates the distribution of sub-optimal planning decisions for physical join operators. The individual queries are sorted according to their time difference in comparison to their hash join restricted plan equivalent for (i) using the native optimizer's operator selection and (ii) using the optimal operator selection. While query plans of the red area execute faster with the restriction to hash joins, the green area represents native plans that outperform their hash join restricted equivalent. The blue gap indicates the missed potential with respect to the optimal operator selection.

Postgres Vanilla. While a recent Postgres (v12.4) achieves decent query response times for JOB, using an older version (v9.6) fails to execute JOB within a 1h time limit (cf. Table 1). However, by limiting the physical execution to hash joins, the benchmark completes, showcasing the implications of sub-optimal operator selections. Comparing the hash join restricted and native plans, we observe that as many as 20% of the queries execute significantly faster when avoiding loop joins completely (cf. red area in Figure 3a). The distribution of potential time savings due to the optimal operator selection (cf. dashed line in Figure 3a) shows that all 113 benchmark queries benefit from the revised operator selection. Substituting the native operator selections with the optimal join operators reveals a significant speed-up of factor 2.8x.

Sketch. The sketch-based approach of Cai et al. [4] captures data correlations and provides high-quality estimates for intermediate result sizes (cardinalities). As a result, *Sketch* performs close to the optimal operator selection. We observe that less than 10% of the native plans execute marginally faster when avoiding loop joins completely (cf. Figure 3b). However, as reported in Table 1, online sketch building imposes a substantial planning overhead that exceeds plan execution time by a factor of more than 6x. Unfortunately, such serious planning overhead renders this approach impractical for queries requiring plenty of joins. Despite the high-quality estimates, a marginal speed-up of 1.1x *with* planning time and 1.5x *without* planning time can be achieved when using the optimal selection of join operators across all queries.

Simplicity. The *Simplicity* strategy reflects a lightweight upper bound approach to minimize the risk of disastrous planning decisions. The bound calculation only requires the selectivity of base table filter expressions and basic frequency counts (top-k statistics). Besides using a generic greedy enumeration for n:m joins, *Simplicity* considers 1:n joins as special filters as they may shrink but never expand the size of the foreign key table. Similar to the push down of regular filters, 1:n joins are prioritized over the potentially expanding n:m joins. In line with the upper bound n:m join enumeration, *Simplicity* heavily relies on hash joins as they generally outperform loop joins given large intermediate results [42]. Thus, there is no difference between the native and the hash join restricted plan (cf. Table 1). Accordingly, Figure 3c only highlights the difference to the optimal operator selection. This distribution of potential time savings demonstrates that a significant speed-up requires good operator selection across all queries instead of optimizing single outliers. Overall, comparing the native (hash restricted) plans to the optimal operator assignment reveals a substantial speed-up of factor 2.5x that is missed due to the simplistic operator selection.

2.2 Lesson Learned

Our experimental analysis exhaustively tested the execution plans of three different query optimizers while systematically substituting the native physical operator selections for JOB. By comparing

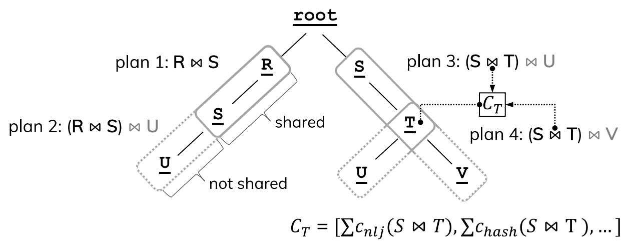


Figure 4: *TONIC*'s case base: QEP-S trie.

the response times of the hash join restricted plans of each optimizer, we see the implications of different join orders produced by the optimizers. Moreover, while the fast but error-prone cardinality estimates of *Vanilla* v12.4 result in better operator selections than purely relying on hash joins, *Vanilla* v9.6 proposes plans that cannot be executed within the given time limit. In contrast, *Sketch* showcases that high-quality cardinality estimates achieve operator combinations that perform close to the optimal operator selection. However, these accurate estimates often entail substantial overheads. Moreover, our results show that good operator selection matters and usually requires accurate cardinality estimates. Still, none of the considered optimizers fully reaches the potential of optimal operator selections. By applying the optimal operator selection to the native plans, the cumulative JOB response times land in the same ballpark, regardless of the join ordering concept. Providing the lightweight optimizers (*Vanilla*, *Simplicity*) with optimal operator selections achieves a speed-up between 2.5x and 2.8x compared to the native operator selections. On the one hand, we observe that good join operator selections can substantially improve single long running queries. For instance, the *Vanilla* (v12.4) QEP of query 19d executes in 138s in the hash restricted scenario. In case of *Vanilla*, this is as much as 30% of the cumulative response time of all 113 benchmark queries. Turbo-charging query 19d with the optimal operator selection reduces the response time from 138s to 8s. On the other hand, all *Simplicity* query plans execute in less than 10s. Still, running the turbo-charged QEPs of *Simplicity*, the benchmark executes more than twice as fast as with the native plans.

3 *TONIC* OVERVIEW

To turbo-charge QEPs with optimal operator selections, we propose *TONIC*, a novel *learning-based physical execuTiOn plaN refInement Component* that replaces cardinality estimation with collected empirical data. The core idea of *TONIC* is to continuously learn and reuse the exact costs of physical join operator alternatives. The exact costs are determined after query execution, allowing optimal physical join operator selections. *TONIC's* assumption is that different SPJ queries share join orders such that optimal physical join operator selections of previous queries can be reused. To realize that, *Case-Based Reasoning* (CBR) as introduced in Section 1, is the perfect foundation of *TONIC*.

3.1 Case Base: Query Execution Plan Synopsis

As illustrated in Figure 2, CBR requires a *case base* storing target problems and their experienced solutions. As target problem for our *TONIC* reasoner, we define the join order given by a generic query optimizer (e.g., extracted from the determined QEP for an

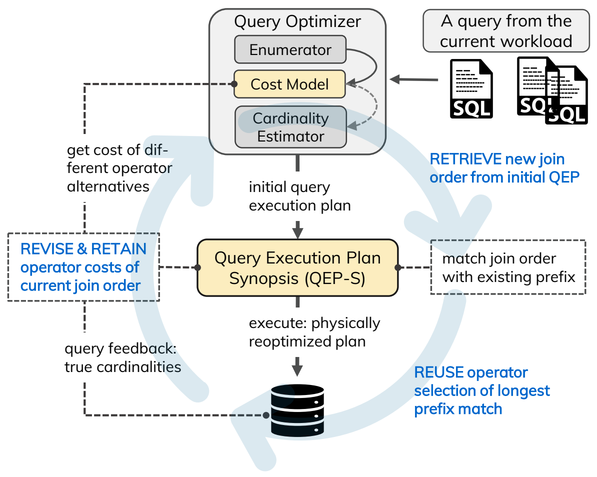

Query Optimizer
Enumerator
Cost Model
Cardinality Estimator
A query from the current workload
SQL
SQL
SQL
RETRIEVE new join order from initial QEP
get cost of different operator alternatives
initial query execution plan
REVISE & RETAIN operator costs of current join order
Query Execution Plan Synopsis (QEP-S)
match join order with existing prefix
execute: physically reoptimized plan
query feedback: true cardinalities
REUSE operator selection of longest prefix match

Figure 5: *TONIC's* operation mode.

SPJ query). Accordingly, the solution of the target problem requires the selection of *optimal* physical join operators for the given join order to minimize the cost or query response time, respectively.

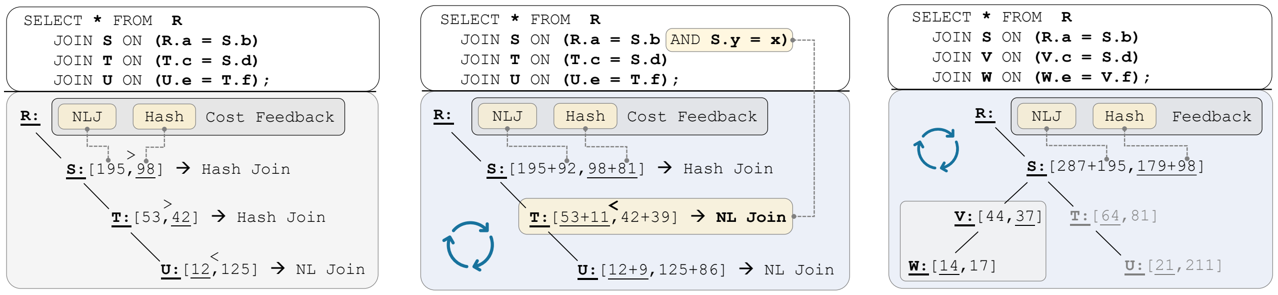
To enable an efficient *case base* representation for *TONIC*, we introduce the concept of a *Query Execution Plan Synopsis* (*QEP-S*) that captures core characteristics of query execution plans in a concise data structure. In particular, *QEP-S* is a *prefix tree* (*trie*) that reflects arbitrary join orders from multiple query execution plans. Since join orders already comply with a tree-like structure, we build the *QEP-S* as a trie where nodes represent tables or intermediate results and edges represent the respective joins. Additionally, each *QEP-S* node $\mathcal{T}$ stores a cost summary $C_{\mathcal{T}}$ for different physical operator implementations – e.g., nested loop (nlj) and hash join (hash) – for joining table $\mathcal{T}$ with the join intermediate result that combines all tables in the node's prefix. Thus, the cost summary at every node sketches multiple query execution plans of previous queries that share join orders to some extent. Based on previous executions, we are able to reuse already existing *QEP-S* prefixes to select physical join operators of minimal cost for new queries.

Figure 4 showcases the prefix recycling for the join orders of four query plans. As the result set of $R \bowtie S$ is the first intermediate result of $(R \bowtie S) \bowtie U$, both join orders share the same prefix. In the shown example, C_T is the cost summary for the join $S \bowtie T$, which stores the sum of costs for different join operators from previous query executions. Given C_T, we can identify the physical join operator of minimal cost that we reuse for the first intermediate result of $(S \bowtie T) \bowtie U$ and $(S \bowtie T) \bowtie V$.

3.2 Life-Cycle Overview

Next, we apply *QEP-S* as *TONIC's* case base following the general CBR life cycle. Starting with an empty *QEP-S*, *TONIC* continuously copies the join order of each executed query and annotates it with the cost of each physical join operator alternative. Figure 5 gives an overview of *TONIC's* operation mode, consisting of the four stages of *Case-Based-Reasoning*:

① **RETRIEVE:** In the first stage, *TONIC* receives the execution plan of a new SPJ query determined by a generic query optimizer,


```
SELECT * FROM  R
   JOIN S ON (R.a = S.b)
   JOIN T ON (T.c = S.d)
   JOIN U ON (U.e = T.f);
```

```
SELECT * FROM  R
  JOIN S ON (R.a = S.b AND S.y = x)
  JOIN T ON (T.c = S.d)
  JOIN U ON (U.e = T.f);
```
R:
NLJ
Hash
Cost Feedback
S:[195+92,98+81] → Hash Join
T:[53+11,42+39] → NL Join
U:[12+9,125+86] → NL Join

```
SELECT * FROM  R
  JOIN S ON (R.a = S.b)
  JOIN V ON (V.c = S.d)
  JOIN W ON (W.e = V.f);
```

(a) 1st query: build initial QEP-S

(b) 2nd query: conflict at node T

(c) 3rd query: add new branch

Figure 6: Detailed Example of the QEP-S Life-Cycle. Join orders are depicted by the explicit join syntax.

extracts the join order and searches for the longest prefix match within the *QEP-S* to find the most similar case.

② **REUSE:** In the second stage it will be tried to use the retrieved prefix for solving the operator selection for the new QEP. The physical join operators that resulted in the lowest cost in previous executions are selected for each node (join) in the retrieved prefix. Then, we overwrite the given execution plan with the selected operators for matching joins and keep the default optimizer's operator selections for the remaining joins. Afterward, the re-optimized query execution plan is executed.

③ **REVISE:** After query execution, the *actual* sizes of join intermediate results as feedback from the execution engine are available. In the third stage, we use the actual join cardinalities as input for the default optimizer's cost model to determine the cost of every physical join operator alternative. Based on the cost feedback, we revise the operator decisions of the previous stage.

④ **RETAIN:** After retrieving the exact costs for each join and each physical operator alternative of the refined QEP, we integrate the adjusted solution into *QEP-S* in the fourth stage. For existing prefixes, we only add the determined cost feedback to the corresponding *QEP-S* nodes. If the considered join order includes joins that are not yet represented by a *QEP-S* prefix, we extend an existing *QEP-S* branch or add a new branch to the trie where we store the determined operator costs to derive new solutions. Thus, this stage represents the learning phase.

To sum up, *TONIC* builds and continuously maintains a lightweight query execution plan synopsis (*QEP-S*) which is used to turbo-charge execution plans with learned physical join operator selections of previous, similar plans.

3.3 Example Walk Through

As query optimizers are commonly challenged with the choice between hash- and index nested loop joins, we use these physical join operator variants as a running example. However, our *QEP-S* case base can capture an arbitrary number of operator alternatives, e.g., different hash join or sort-merge join implementations [37].

Figure 6 illustrates the evolution of the *QEP-S* case base for three query plans. For ease of reading, we assume that the join orders of the query execution plans correspond to the join orders given in the explicit SQL syntax. Let $((R \bowtie S) \bowtie T) \bowtie U$ be the join order of tables R, S, T, U. Starting from an empty trie, we execute every join according to the default optimizer's decisions. After query execution, *TONIC* uses the *actual* join cardinalities to initialize the *QEP-S* with the cost of all available join operator alternatives.

As Figure 6a depicts, the corresponding *QEP-S* node of T stores the operator costs $c_{nlj}(I_1 \bowtie T), c_{hash}(I_1 \bowtie T)$, where I_1 is the intermediate result $R \bowtie S$, c_{nlj} the cost function of a nested loop join, and c_{hash} the cost function of a hash join. Analogous to traditional query optimization, *TONIC* searches for the operator sequence that minimizes the combined cost, e.g., minCost $= 98 + 42 + 12$ for using a hash join for I_1, $I_2 = I_1 \bowtie T$, and a loop join for $I_3 = I_2 \bowtie U$.

As illustrated by Figure 6b, adding a filter condition to the first query may still result in the same join order. However, the optimal operator selection now requires a loop join for $I_1 \bowtie T$ instead of a hash join due to the filter selectivity. To decide which operator to recommend, *TONIC* keeps adding the cost of operator alternatives for each execution of a particular join path. Thus, rather than capturing optimal individual join operator selections, the *plain QEP-S design* captures operator selections that are – based on the current workload – considered most efficient for an abstract join order.

Figure 6c depicts another query that continues the existing prefix R-S by joining V, W. Since the prefix R-S is already contained, *TONIC* explicitly uses the *QEP-S* to recommend a hash join. Thus, although the join order of the third query is not yet fully contained in the *QEP-S*, we can still exploit the operator recommendation of the already existing prefix. As the prefixes R-S-V and R-S-V-W have not yet been considered, *TONIC* implicitly falls back to the default optimzer's operator selections for $I'_2 = I_1 \bowtie V, I'_3 = I'_2 \bowtie W$. After query execution, the respective cost feedback for I'_2, I'_3 is stored in a new branch V-W under the existing prefix R-S.

Algorithm 1 summarizes the *QEP-S'* life-cycle as foundation of our *Case-Based-Reasoning selection strategy*. Given the logical join order from the underlying optimizer, we traverse the join order while searching for corresponding *QEP-S* nodes (Line 3-15). For the operator recommendation, we distinguish the following cases: If the prefix does not contain at least two tables, there is no join and therefore no operator selection (Line 14-15). If the join path considered so far does not match any existing *QEP-S* prefix, we initialize a new *QEP-S* node and fall back to the operator selection of the underlying query optimizer (Line 7-12). Otherwise, we reuse an existing *QEP-S* prefix and apply the operator with the minimal associated cost for the join (Line 15). After executing the query

Algorithm 1: QEP-S life-cycle

```
   Input: logical join order logicalPlan, plan synopsis QEP-S
   /* retrieve and reuse                                        */
1  initialize empty prefix;
2  QNode = root node of QEP-S;
3  while logicalPlan.hasNode() do
4      nextNode = logicalPlan.nextNode();
5      nextId = nextNode.identifier;
6      add nextId to prefix;
7      if not QEP-S.contains(prefix) then
8          initialize new QEP-S node newQNode;
9          newQNode.identifier = nextId;
10         newQNode.costSummary = empty list;
11         newQNode.recommended =
             operator selection of DBM's native optimizer;
12         QNode.childNodes[nextId] = newQNode;
       /* get matching QEP-S node                               */
13     QNode = QNode.childNodes[nextId];
14     if prefix contains IDs of at least two tables then
15         use QNode.recommended to join next table;
16 execute query with recommended join operators;
   /* revise and retain                                         */
17 collect actual cardinalities of intermediate results;
18 foreach QEP-S node corresponding to logicalPlan do
19     get cost of operator alternatives from cost model;
20     add operator cost to the nodes's cost summary;
21     set node.recommended to operator with minimal cost;
```

with the recommended join operators (Line 16), we reiterate and update all previously considered *QEP-S* nodes according to the cost feedback based on the actual join cardinalities (Line 17-21).

4 QEP-S DESIGN CONSIDERATIONS

The *Query Execution Plan Synopsis* (*QEP-S*) as case base is the most important ingredient of *TONIC*. To give a general overview of *TONIC*, we previously introduced only a rudimentary version of the *QEP-S*. This section is dedicated to elaborate on specific challenges and their solutions. These challenges are as follows: (C1) While a *QEP-S* prefix corresponds to a linear join order so far, how can we incorporate bushy join plans into the *QEP-S* trie? (C2) While *QEP-S* nodes continuously accumulate the cost of operator alternatives for each query execution, how can we ensure the freshness of the cost feedback? (C3) While two query plans may have the same join order, how can we account for different optimal operator selections due to different base table filters?

4.1 C1 - Integrating Bushy Query Plans

This section discusses the *QEP-S* representation of *bushy join plans*. In the following, we use the term *sub-query* as a synonym for independent branches of a bushy join tree. Having a *QEP-S* containing the prefix R-S-T-U for joining tables R, S, T, U, how can we distinguish the bushy plan $(R \bowtie S) \bowtie (T \bowtie U)$ from the linear join path $((R \bowtie S) \bowtie T) \bowtie U$? While the *QEP*-S prefix R-S-T-U tracks

```
SELECT * FROM  R
  JOIN S ON (R.a = S.b)
  JOIN (SELECT c FROM T
    JOIN U ON (T.e = U.f)) AS subquery
      ON (subquery.c = S.d)
  JOIN ...
```

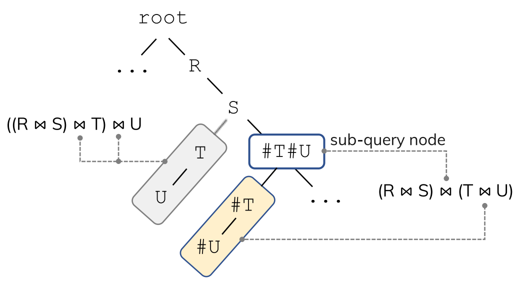


Figure 7: Dealing with sub-queries.

the cost of subsequently joining *single* tables to the intermediate result of the preceding joins, the bushy plan requires the cost of joining *two* intermediate results. Further, the linear prefix stores the cost for joining U with $R \bowtie S \bowtie T$ but the bushy plan requires a sub-query where U joins with T before joining with $R \bowtie S$.

Node identifier. The integration of bushy trees into the *QEP-S* trie requires the identification of join intermediate results. In case of a linear join order, we implicitly used the base table IDs as identifier for *QEP-S* nodes along the prefix. To distinguish *single table nodes* from *sub-query nodes*, we use "#" as unique tag and combine the IDs of all tables considered in the sub-query, e.g., #T#U for sub-query $(T \bowtie U)$. Accordingly, a *QEP-S* node can either represent a single table or a join intermediate result. Analogously to single table nodes, sub-query nodes summarize the cost feedback for joining the embodied intermediate result with the result of the preceding joins. To account for single table joins within a sub-query $(T \bowtie U)$, we add a dedicated *QEP-S* branch #T-#U which tracks the cost of joining T with U before joining $T \bowtie U$ with $R \bowtie S$.

Figure 7 illustrates the previous example for storing bushy join plans. The sub-query join $T \bowtie U$ is highlighted yellow. We store the sub-query's join path #T–#U to indicate that this *QEP-S* branch participates in a sub-query. The result of the sub-query is represented by the sub-query node #T#U on the *QEP-S* prefix R–S–#T#U while the prefix R-S-T-U depicts a linear join plan.

By integrating sub-queries, we get a rudimentary *QEP-S* version that supports arbitrary join orders. We call this version *plain QEP-S*.

4.2 C2 - Adapting to Changes

So far, *TONIC* recommends physical join operators according to the *QEP-S* prefix tree, where each node contains an *unbiased* cost summary from previous query feedback. That is, *TONIC* decides for the operator associated with the smallest sum of costs:

$$\sum_{i=0}^{n} c_i = \underbrace{c_0 + c_1}_{\text{stale over time}} + \cdots + \underbrace{c_n}_{\text{most recent}}, \qquad (1)$$

where c_i is an operator's cost for the i-th query (cf. Figure 6). As more queries are processed, we can be sure that the operator with minimal accumulated cost is –holistically seen– the single best choice for the encountered workload. However, the stored costs of previously executed queries might become stale over time due to changing workload or data characteristics. Therefore, it is appealing to give more weight to recent queries. To strike the right balance between an exact cost accumulation and a fast workload adaptation, we use the following *gamma-weighted* sum:

$$C_n = \sum_{i=0}^{n} \gamma^{n-i} c_i = \gamma^n c_0 + \gamma^{n-1} c_1 + \ \dots\ + \gamma^0 c_n, \qquad \gamma \in (0, 1]$$

As we only add the current cost c_n to the previous sum C_{n-1}, we can use the gamma-weighting recursively:

$$C_n = c_n + \gamma * C_{n-1}, \qquad \text{with } C_0 = c_0.$$

Thereby, the choice of γ mediates the bias towards the most recent queries. The smaller the gamma-weight, the more we focus on the most recent history and vice versa. Using $\gamma = 1$ will degenerate the weighted sum to Equation (1). Further, using $\gamma < 1$ circumvents an ever-increasing cumulative sum of costs for frequent queries. That is, considering a specific join with operator-cost *cost*, which is executed an arbitrary number of times, the weighted sum will never exceed the following fixpoint:

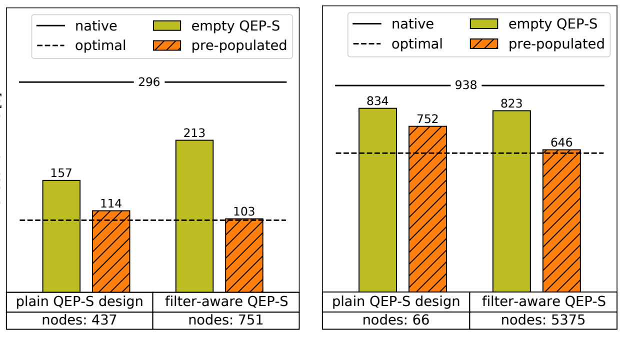
$$\gamma * \text{fixpoint} + cost = \text{fixpoint} \iff \frac{cost}{1-\gamma} = \text{fixpoint}$$

Thus, the gamma-weighted sum bounds the cost representation within a *QEP-S* node, mediates the bias to recent queries, and can be seamlessly integrated due to its recursive equivalent.

4.3 C3 - Filter Sensitive Branching

So far, we have described the *plain QEP-S* as synopsis which captures abstract join orders of query execution plans. Recall that *TONIC* only adds new branches to the *plain QEP-S* when no existing prefix matches the given join order. Otherwise, existing branches are updated with the join operator costs of the current query. While two queries can result in the same logical join order, their physical join costs and optimal operator decisions can differ due to different base table filters. For instance, given filter expressions with small selectivities, a chain of *index* nested loop joins might be more cost efficient than scanning and building hash maps from base tables. These situations are not covered by the *plain QEP-S* design.

Filter-aware QEP-S. To reduce conflicting operator decisions for identical join orders, we propose the *filter-aware QEP-S*, a *QEP-S* that is highly sensitive to filter expressions. Instead of defining a prefix as plain concatenation of table IDs, we additionally add the respective filter expressions to the prefix, thus achieving a *filter-sensitive* distinction of *QEP-S nodes*. Accordingly, the *filter-aware QEP-S* might introduce new branches for already considered join orders to account for different filter expressions. Let $(R \bowtie S)$ and $(\sigma_{R.x=y}(R) \bowtie S)$ be two logical query plans for joining R, S that only differ in the base table select statement $\sigma_{R.x=y}(R)$. While a *plain QEP-S* branch R—S would match both plans, the *filter-aware* version of *TONIC* adds a dedicated *QEP-S* branch R[x=y]—S to account for the filter expression. Essentially, we only extend the identifier of a *QEP-S* node and use a more detailed prefix to distinguish join orders subject to different base table filter expressions. Other than that, all previously considered concepts stay the same.



(a) JOB (b) Stack

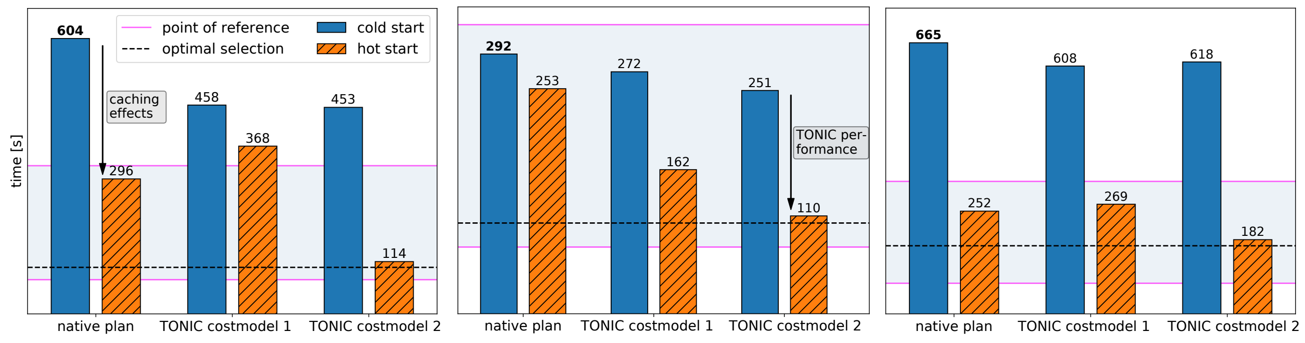
Figure 8: Tonic performance overview (Vanilla v12.4).

5 EVALUATION

To evaluate *TONIC*, we use two different real-world SPJ benchmarks; namely the *Join-Order-Benchmark* (JOB) [25] and *Stack* [29]. While **JOB** is based on 10GiB of data extracted from the *Internet Movie Database*, **Stack** contains over 18 million questions and answers from StackExchange websites with a total size of 100GiB. From a query perspective, JOB consists of 113 SPJ queries that can be separated into 33 SPJ-patterns. In contrast, *Stack* provides fewer join patterns than JOB but vastly more queries with more diverse filter predicates per SPJ-pattern. For the *Stack* benchmark, we restrict ourselves to 1,000 randomly selected queries from 10 patterns.

All experiments are carried out under the same hardware environment as reported in Section 2. We implemented our core datastructure *QEP-S* in C++ and use Python scripts as communication link between Postgres and *TONIC*. We decided for this loosely-coupled integration in order to perform a flexible evaluation with different Postgres versions – namely v12.4 and v9.6. Moreover, to enable non-trivial decisions between hash and *index* loop joins, we run the benchmarks with all foreign key indices available and analyze particular index requirements in Section 5.5. Further and in line with related work, we execute all benchmark queries in a random order. The following results are consistent with a downstream re-evaluation based on different execution orders.

General performance. As shown in Figure 8, *TONIC* significantly reduces response times for both benchmarks. The response times were determined using Postgres *Vanilla* v12.4 with an explicit warm-up run. In this warm-up run, the respective benchmark was executed to fill the database caches. Moreover, both branching policies (*plain* and *filter-aware*) for our *QEP-S* have been evaluated and the *TONIC* learning approach uses a perfect cost model based on actual response times. The term *empty QEP-S* indicates that *TONIC* starts with an *empty trie* for the benchmark evaluation and updates the *QEP-S* after each query execution. As a result, already learned operator selections for abstract join paths become available for the remaining benchmark queries. The term *pre-populated* indicates that *TONIC* has already seen every query, e.g. during the preceding *empty QEP-S* run. That is, the *pre-populated QEP-S* stores operator selections for all relevant join paths. As each join has a guaranteed prefix match, the operator selection is fully dictated by *TONIC's* knowledge base. While running the benchmark from an *empty QEP-S* evaluates adaptation capabilities, we can judge the information loss (distance to optimal planning decision) when running the benchmark with a *pre-populated QEP-S.*



(a) Vanilla (v12.4) **(b) Simplicity (v12.4)** **(c) Sketch (v9.6)**

Figure 9: Response times with and without *TONIC*. To raise awareness of the different time scales we add a point of reference.

From the JOB experiment shown in Figure 8a, we see that *TONIC* with the *plain QEP-S* already heavily reduces query response times during the first benchmark iteration (*empty QEP-S* run). Re-running the benchmark and therefore applying the *pre-populated QEP-S* reveals that the *filter-aware QEP-S* is able to match the best-performing operator selection (fastest benchmark response time).

Additionally, Figure 8b reveals interesting characteristics of the Stack benchmark. Again, *TONIC* considerably reduces benchmark response times. However, the vast difference between nodes counts for the *plain* and *filter-aware* design indicates less complex join patterns and a stronger focus on filter expressions. As a result, the *filter-aware QEP-S* clearly outperforms the *plain* design in terms of query response times but requires substantially more nodes.

5.1 Performance Factors

The results so far only promote a rough impression of *TONIC's* promising behavior. In this section, we provide a representative in-depth analysis of external performance factors such as different join orders and cost models. Therefore, we examine *TONIC* with multiple optimizers (cf. Sec. 2), where (i) *Vanilla* is the standard Postgres v12.4 optimizer, (ii) *Sketch* is the standard Postgres v9.6 optimizer with access to *costly but precise cardinality estimates* [4], and (iii) *Simplicity* is a *pessimistic join ordering concept restricted to hash joins* [14]. Moreover, we evaluate the following scenarios:

cold start. We run the benchmark *without* a dedicated warm-up phase of the database. This scenario corresponds to loading data and directly executing an analytical query workload.

hot start. We run the benchmark *after* a dedicated warm-up phase. This scenario corresponds to periodically issuing a query workload, e.g., to update a dashboard application.

cold start + TONIC. We start the benchmark according to the cold setting with an *empty QEP-S*.

hot start + TONIC. We run the benchmark according to the hot start setting with a *pre-populated QEP-S*.

Since *TONIC* bootstraps the cost model of the Postgres optimizer, we further evaluate *TONIC's* performance with two cost functions:

costmodel 1. We use the *untuned* cost model of a default Postgres instance. Based on the actual join cardinalities, we search for the operator selection that minimizes the cost approximation of the default optimizer. Using the optimizer's cost model requires no additional query execution. As a cost function usually is a relatively simple algebraic expression, the imposed overhead is $\ll 1\%$, and thus, can be neglected.

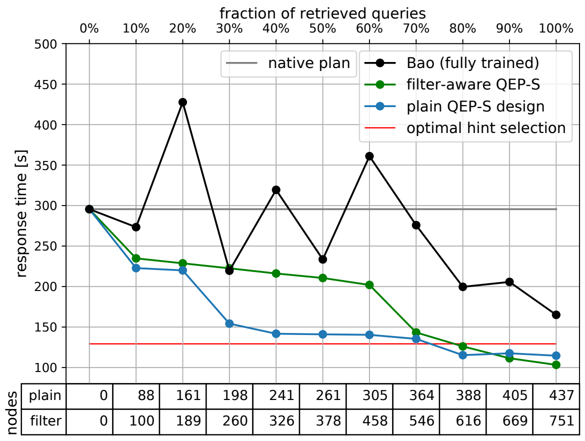
costmodel 2. To avoid approximation errors, the second model simulates a *perfect* cost model based on real response times. That is, the operator selection that minimizes the cost of *costmodel 2* directly minimizes the response time of the benchmark. As this model requires additional plan executions, its imposed overhead is tied to the respective query.

Since *TONIC* might integrate several cost models for asynchronous *QEP-S* updates, we do not report dedicated cost model overheads. For instance, we can use *costmodel 1* for ad-hoc *QEP-S* updates and *costmodel 2* to fine-tune the *QEP-S* during low traffic times.

Figure 9 shows representative evaluation results for JOB and *TONIC* using the *plain QEP-S*. The following statements apply in a similar way to Stack and *TONIC* using the *filter-aware QEP-S*. In accordance with Figure 8, Figure 9 demonstrates that building *TONIC's* case-base in the *cold start*, *empty QEP-S* setting positively affects query response times in all considered scenarios – no matter the underlying query optimizer and cost model. Applying a *QEP-S* pre-populated with feedback from *costmodel 2* substantially outperforms the native operator selections with query response times close to the optimal selection. In line with Section 2, *Vanilla* and *Simplicity* receive substantial improvements while *Sketch* performs already well due to high-quality cardinality estimates. In contrast to *Simplicity*, we observe a vast difference between the hot start and cold start setting when using the unmodified *Vanilla* plans. This gap indicates a strong caching behavior for the native plans. Unfortunately, Postgres' native cost function (*costmodel 1*), does not account for these caching effects. Surprisingly, *TONIC's* refined plans result in smaller cost estimates according to *costmodel 1* but execute slower than the *Vanilla* native plans in the hot start setting. However, using *costmodel 2* fixes this issue as it accounts for the caching behavior. As individual outliers and the caching behavior of Postgres *Vanilla* combined with *costmodel 1* make it hard to separate *TONIC's* performance from caching effects, we use *costmodel 2* and a *hot database state* in the following experiments.

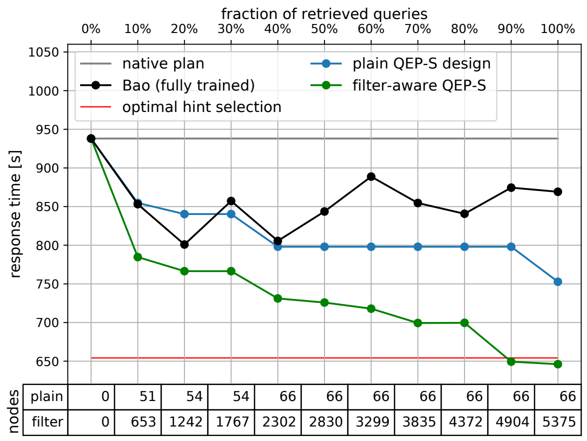
5.2 Rate of Improvement

As has been shown, *TONIC* with a fully *pre-populated QEP-S* delivers the fastest response times for the considered optimizers and



nodes	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
plain	0	88	161	198	241	261	305	364	388	405	437
filter	0	100	189	260	326	378	458	546	616	669	751

(a) Join-Order-Benchmark



nodes	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
plain	0	51	54	54	66	66	66	66	66	66	66
filter	0	653	1242	1767	2302	2830	3299	3835	4372	4904	5375

(b) Stack-Benchmark

Figure 10: Rate of improvement: Comparison between TONIC and a recent ML approach.

benchmarks. This *pre-population* can also be interpreted as a training phase. In this section, we analyze this aspect in more detail and show that *TONIC* achieves satisfying results with significantly less training samples than *Bao* [29]. In the following, we refer to the achieved response time improvement based on a certain amount of training data as *rate of improvement.*

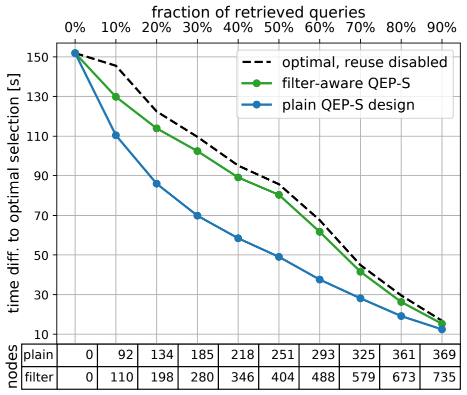
Vanilla: In the context of this work, *Bao* [29] is a very relevant and recent competitor. Like *TONIC*, *Bao* builds on top of an existing query optimization infrastructure – e.g., Postgres *Vanilla* – and issues planning decisions on a per-query base. *Bao* employs state-of-the-art tree convolutional neural networks to recognize meaningful query patterns and to learn when to activate (or deactivate) optimization features. Therefore, the *Bao* model selects certain query hints, e.g. `"set enable_nestloop to false"` to guide planning decisions of the underlying optimizer for every query. However, according to [29], these hints are incapable of making fine-grained planning decision, e.g. avoiding a loop join between table *A* and *B*, while still allowing a loop join between table *C* and *D*.

The *Bao* Postgres extension is available at [28] and builds on top of Postgres' default optimizer. In order to ensure a fair comparison, we run *TONIC* and *Bao* with Postgres *Vanilla* v12.4. Moreover, we evaluate *TONIC's* rate of improvement in comparison to *Bao*. That means, we train *TONIC* and *Bao* based on a certain fraction of retrieved queries and freeze the respective models (*QEP-S* and *Neural Network*) afterwards. We use the frozen snapshot and run the *full* benchmark queries to assess the rate of improvement for both approaches. To enable *Bao's* full potential, we exhaustively use *Bao's exploration mode* [11] where *Bao* executes and collects actual response times for every query and hint combination. Thus, *Bao* is trained with the maximum amount of feedback possible.

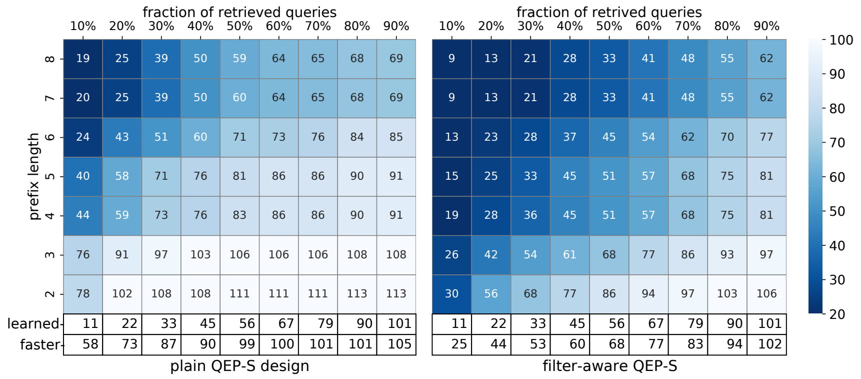
Figure 10 reports benchmark response times of *Bao* and *TONIC* depending on the number of learned queries. The red line marks the optimal benchmark response time as theoretical optimum within *Bao's* search space. Figure 10a reveals that *Bao* significantly reduces query response times after receiving 80% of JOB-queries. Given only few training examples, *Bao's* performance seems unreliable and strongly fluctuates between training fractions. Interestingly, *TONIC* already performs well with few training queries and is able to surpass the optimal hint selection of *Bao* due to more fine-grained planning decisions. Further, Figure 10b shows that *Bao* reliably outperforms the native Postgres plans on the Stack-Benchmark with a performance close to *TONIC* with the *plain QEP-S* design. However, using *TONIC* with the *filter-aware* branching policy substantially decreases query response times even for small training fractions, resulting in a much better rate of improvement. In contrast to *Bao*, *TONIC* falls back to the underlying optimizer for completely unknown query patterns and therefore shows stable improvements with an increasing number of retrieved queries.

Simplicity: To demonstrate a broad applicability, we reiterate the previous experiment with the *Simplicity* join orders and additionally detail the reuse of already retrieved query patterns (prefixes) to understand why *TONIC* is able to achieve a good performance given only few training examples. Figure 11a shows the benchmark response time *difference* to the best operator selection depending on the fraction of already retrieved queries. As a baseline we use the optimal operator selection for already retrieved queries and the native operator selection for the *remaining* benchmark queries. Comparing node counts for the *filter-aware* and *plain QEP-S*, we see that the *filter-aware* branching again manifests in a significantly higher number of *QEP-S* nodes. Using the *plain QEP-S*, *TONIC* is able to transfer meaningful operator decisions to unknown queries. After retrieving 30% of the queries, the *plain QEP-S* already reduces the cumulative benchmark time by 82s (>50% of maximum reduction). In contrast, the *filter-aware QEP-S* performs close to the native operator selection of *Simplicity*.

To analyze the reuse of learned query patterns, we attach a counter to *QEP-S* nodes that is increased whenever a node is accessed. In line with the previous experiment, we freeze the *QEP-S* after receiving a certain fraction of queries and re-run the *full* benchmark with the frozen snapshot while only incrementing node counters whenever a query pattern matches the respective prefix. Figure 11b comprises the frequency with which prefixes of a particular length are accessed across *all* benchmark queries depending



(a) TONIC: rate of improvement



(b) QEP-S prefix reutilization

Figure 11: TONIC + Simplicity: rate of improvement and prefix reutilization.

on the number of retrieved training queries and the *QEP-S* branching policy. Access counts are indicated by a blue color gradient where brighter tones mean more accesses. To highlight differences between the *filter-aware* and *plain QEP-S* design, the presentation is limited to prefixes with up to eight nodes (maximum length is 18). The table at the bottom reports the absolute number of retrieved (learned) queries used to populate the *QEP-S*. Further, out of the 113 JOB queries, we see the number of queries that execute faster than the native plans based on the number of learned queries.

With regard to Figure 11a, we observe that the smaller node counts of the *plain QEP-S* coincide with a stronger prefix reutilization. That is, after retrieving 10% of the workload, *TONIC* with the *plain QEP-S* already improves response times for more than 50% of the JOB queries. After retrieving 40% of the workload, 90% of the queries have a guaranteed prefix match for the first two joins (prefix length three). Note that not all queries with a prefix match receive a response time improvement as the underlying optimizer might already select the optimal operators in some cases.

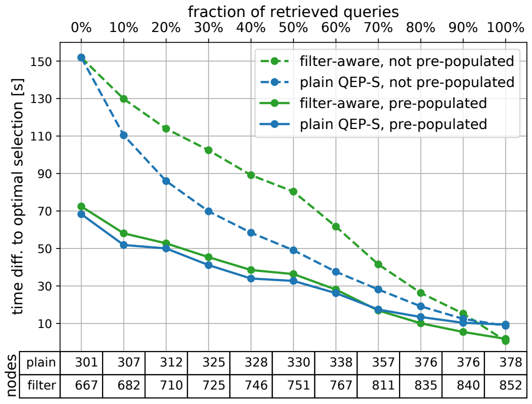
5.3 Data Shift

Due to *TONIC's* learning capabilities, a quick adaptation to query changes within workloads is possible. In this section, we analyze *TONIC's* adaptation capabilities under a heavy data shift. Therefore, we create another JOB instance where we randomly remove half the tuples from tables with at least 100k tuples. Due to the multiplicity of join cardinalities the combined tuple count of all query result sets decreases from 28.8Mio to 196K. To simulate a data shift, we first run *TONIC* with the reduced dataset and use the resulting *pre-populated QEP-S* to run the default benchmark afterward. That is, instead of starting with an *empty QEP-S*, we consecutively reuse and update the *pre-populated QEP-S*.

Figure 12a comprises the benchmark response time *difference* to the best operator selection and node counts depending on the fraction of retrieved queries for the *pre-populated QEP-S*. As a comparison, we mark the initial performance for starting with an *empty* (*not pre-populated*) *QEP-S* by dashed lines. Interestingly, this comparison reveals that both *QEP-S* designs heavily benefit from query feedback collected during a single run over the reduced dataset. We already observe a response time reduction of 80s for applying the *pre-populated QEP-S* without any feedback from the default dataset (query fraction 0%). Thus, the *QEP-S* structure enables *TONIC* to maintain meaningful operator decisions under a heavy data shift. Since the reduced dataset leads to different join orders than the default JOB dataset, we see a slight increase of *QEP-S* nodes compared to the initial *empty QEP-S* run.

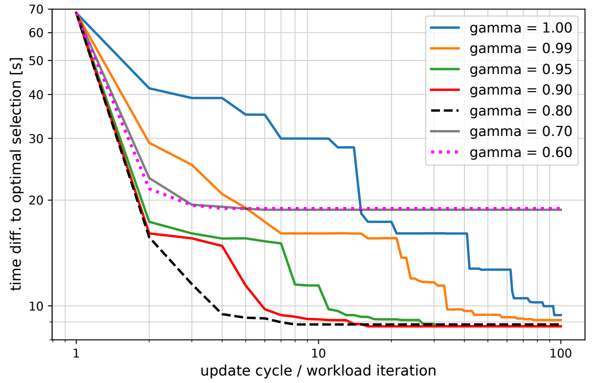
Gamma. To understand to which extent the gamma-weighted cost feedback (cf. Sec. 4.2) affects adaptation speed, we evaluate different values of γ next. By running queries several times, we continuously add cost feedback to the *QEP-S* nodes and increase the cost difference between operator alternatives. Thus, previously accumulated costs may prevent fast adaptation under data shifts as we need to override the outdated decisions with a similar amount of feedback. To simulate stagnating operator decisions, we use *TONIC* with the *plain QEP-S* design and run the benchmark queries 100 times on the *reduced dataset* while continuously updating the *QEP-S*. Afterward, we run the queries 100 times over the *default dataset* while updating the *pre-populated QEP-S* with fresh feedback.

As shown in Figure 12b, we consider $\gamma \in [0.6, 1]$ and report the time *difference* to the optimal operator selection depending on the number of completed update cycles (workload iterations). Note that the x-axis is log-scaled as we are most interested in the first iterations. Iteration 1 corresponds to the first data point of Figure 12a, that is, applying the *pre-populated QEP-S'* operator decision without feedback from the default dataset. While setting $\gamma = 1$ corresponds to an unweighted cost accumulation, it entails the slowest adaptation speed. Due to the heavy bias on the deprecated operator decisions, *TONIC* needs to revise and retain the *QEP-S* more than 60 times for each query to reach near peak performance. Setting $\gamma = 0.99$ considerably accelerates the adaptation speed. Using $0.8 \leq \gamma \leq 0.95$ achieves the fastest adaption while preserving the decision quality of the unweighted feedback history ($\gamma = 1$). Using $0.6 \leq \gamma \leq 0.7$ still achieves a similar performance as the larger γ values when directly applying the deprecated operator decisions (iteration 1). However, since these smaller values of γ add a massive bias towards the very recent feedback, they provide quick adaptation but degrade the holistic operator selection quality.



nodes	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
plain	301	307	312	325	328	330	338	357	376	376	378
filter	667	682	710	725	746	751	767	811	835	840	852

(a) ***TONIC's*** **performance after heavy data shift.**



(b) Adaptation speed of ***plain QEP-S*** **with regard to gamma.**

Figure 12: Data shift: Adaptivity after single workload iteration (left) and 100 iterations (right) depending on gamma.

In particular, using $\gamma = 0.7$ indicates a breakpoint where *TONIC* erratically switches to the optimal operators selection of the most recent query. Therefore, frequently used prefixes optimize single queries instead of capturing operator combinations that minimize the workload-wide cost of all matching join paths.

5.4 TONIC - Runtime Traits

To test the runtime overhead of *TONIC*, we anlayze our prototypical `C++` implementation of the *QEP-S*. Recall that the *plain QEP-S* constitutes a standard prefix tree where nodes are composed of a table identifier, the accumulated cost feedback, and references to child nodes. Child nodes are stored in a standard STL map where keys correspond to the identifiers of the next join partners considered along the logical join path (prefix). For the *filter-aware QEP-S*, we extend the string type identifier with base table filter expressions to achieve a filter sensitive distinction of the abstract join patterns.

Table 2 comprises the memory consumption of each fully populated *QEP-S* design after benchmark completion. We observe that the *plain QEP-S* consumes very little memory due to its simple structure and small number of nodes. Although the *filter-aware QEP-S* uses the same data structure as the *plain QEP-S*, it requires more memory due to the higher node count and the extended identifier. Moreover, the table comprises the cumulative time spent in any interaction with the *QEP-S* structure split into *maintenance* and *reuse*. *Maintenance* is the time required to generate the *pre-populated QEP-S* during the first workload iteration. This includes all *QEP-S* life-cycle stages (cf. Sec. 3.2). In contrast, *reuse* only reports lookup times for existing prefixes and thus excludes the time to integrate new feedback. This is equivalent to a second workload iteration with a *pre-populated, frozen QEP-S*. On average, both *QEP-S* designs spent less than 20µs per query to recommend operator selections and integrate new query feedback. Due to the STL map's constant lookup time complexity ($O(1)$), maintenance and reuse times are mostly unaffected by the branching degree of the *QEP-S*.

Table 2: QEP-S runtime traits.

$\sum$ JOB	memory	maintenance	reuse (lookup)
plain QEP-S	33.8 KB	0.42 ms	0.01 ms
filter-aware	102.3 KB	1.03 ms	0.15 ms

5.5 Discussion

Comparing the *plain* and *filter-aware* design, we observe a trade-off between fast generalization and fine-grained case-by-case decisions. Therefore, we may conclude that the *plain* design is more practical for applications that are expected to run an analytical workload once, while the *filter-aware* design better fits dashboard-like applications that periodically issue a set of analytical queries.

Node Eviction. Despite the low memory consumption, we note an occasionally strong increase of *QEP-S* nodes —especially for the *filter-aware* design— in some scenarios (cf. Fig. 10). To limit the size of the *QEP-S*, we consider adding timestamps or access counters to evict nodes according to a standard policy like LFU or LRU.

Index availability. We show that *TONIC's* learned operator selections for hash joins and *index* loop joins considerably reduce query response times. However, as index loop joins require the respective indices, *TONIC's* performance also depends on index availability. Since Postgres uses clustered primary-key indices as default, we only consider changing foreign-key indices. Starting with all indices available, we systematically analyze response time regressions after dropping foreign-key indices. Interestingly, since *TONIC* accumulates the cost of operator alternatives, we can approximate an index significance rating from the *QEP-S* cost history. Therefore, standard cost functions can be used to simulate index-based joins without having access to the actual index [38]. Using Algorithm 2, we can asses the relevance of (potential) indices by traversing all *QEP-S* nodes and summarizing the (potential) performance gain of index loop joins for each join participant.

To test the relevance of a particular index, we run JOB with and without having access to the respective index using *TONIC* with the *plain QEP-S* design. Figure 13 reports benchmark response time regressions depending on the index availability sorted by the *actual* response time difference. Moreover, the figure indicates *TONIC's* approximated time difference for different values of γ. Remarkably, for high values of γ (≥ 0.9), the ranking lines up with the actual

Algorithm 2: QEP-S foreign-key analysis

```
1 idxGain = dictionary that maps join participants to relative
    cost savings due to (potential) fk-indices;
2 for node in QEP-S do
3     if not node.identifier in idxGain.keys then
4         idxGain[node.identifier] = 0;
5     if node.hashJoinCost > node.indexLoopJoinCost then
6         idxGain[node.identifier] +=
            node.hashJoinCost - node.indexLoopJoinCost;
```

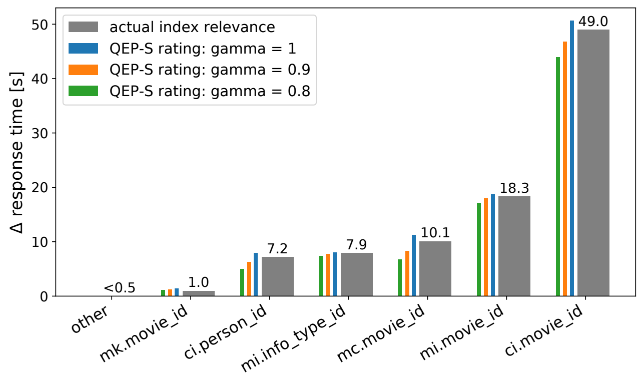


Figure 13: Indices sorted by response time savings.

response time differences. Further, Algorithm 2 commonly identifies indices (e.g. `pi.person_id`) that do not contribute to any response time savings, and thus, can be dropped to free resources.

Two-stage optimizer design. By extending different query optimizers, we demonstrate a broad applicability of *TONIC*. Interestingly, the combination *Postgres Simplicity* and *TONIC* offers a new and comprehensive way for SPJ query optimization. As described in [14], *Simplicity* uses a lightweight upper bound and prioritizes 1:n over n:m joins instead of relying on error-prone cardinality estimates. In the same vein, *TONIC* incorporates direct query feedback without using a dedicated cardinality estimator. Since *Simplicity* lacks a fine-grained operator selection strategy, *TONIC* perfectly complements the –initially– hash join restricted plans. By combining *Simplicity* and *TONIC*, we envision a novel *cardinality estimation-free two-stage optimizer design* that separates logical join enumeration from fine-grained operator selection. While *stage one* relies on the pessimistic join enumeration of *Simplicity*, *stage two* turbo-charges the pessimistic join orders by appropriate operator selections according to *TONIC*. In our future research, we seek to further investigate this approach.

6 RELATED WORK

According to Chaudhuri [6], traditional optimization of arbitrary SPJ queries requires precise estimates of intermediate result sizes (cardinalities). Unfortunately, as shown in [35], ad-hoc estimation techniques are unlikely to achieve such precise estimates. In the same vein, Leis et al. [25] provide empirical evidence that cost-based optimizers are prone to disastrous planning decisions if precise cardinality estimates cannot be provided. To tackle this issue, recent work investigates more computationally intensive sketches [4, 18, 22] or machine learning (ML) approaches [16, 21, 34, 40, 41] to achieve precise cardinality estimates. Beyond cardinality estimation, some ML approaches [23, 29–31] apply reinforcement learning (RL) for holistic query plan optimization. In particular, *Bao* [29] learns and injects `SQL` hints to guide general planning decisions of the underlying optimizer. In Section 5.2, we provide an in-depth comparison with *Bao* as the most recent RL competitor.

Another related research branch studies *adaptive query optimization* [2] to re-optimize execution plans based on a profiler routine that tries to identify more suitable plan alternatives. Approaches like *Eddies* [1] and *Cuttlefish* [20] change join orders (or physical operators, respectively) on a tuple granularity at query runtime.

DB2's optimizer *LEO* [39] took a different route and pioneered the integration of a query feedback loop to account for estimation errors. While *LEO* builds a fine-grained secondary statistic of relative adjustments to repair cardinality misestimates, our approach is completely decoupled from cardinality estimation. The key differences are as follows: (i) *LEO* inherently relies on cardinality estimates on a per-query basis, while *TONIC* uses a workload-dependent cost history/summary of abstract join paths, and (ii) *LEO's* healing statistic is unable to capture arbitrary predicate correlations – especially join-crossing correlations. Contrary, *TONIC* uses a novel synopsis to store cost feedback based on the *exact sizes* of join intermediate results that can be subject to *arbitrary predicate correlations.*

Moreover, the comprehensive survey by Cormode et al. [8] highlights designs and applications of several data-specific synopses. According to Cormode et al., synopses are compact representations of massive data sets to capture core characteristics such as distinct counts, frequency moments, and variance. Data synopses are the foundation of *Approximate Query Processing* [5] and cardinality estimation for ad-hoc query optimization [24]. The literature covers a wide spectrum of data synopses, ranging over histograms [3, 17], wavelets [13, 32], sketches [4, 19], and sampling [15, 33]. In contrast to data synopses, our *QEP-Synopsis* is an abstract of query execution plans as novel means for physical operator selection.

7 CONCLUSION

Despite many years of research, there has not yet evolved a clear picture to which extent physical join operator selection accounts for the plan quality of complex SPJ queries. Therefore, we systematically substituted join operators of various query execution plans. From our evaluation it is evident that good physical join operator selection is crucial to achieve high-quality plans. To unlock the potential of optimal operator selections, we present *TONIC*, a lightweight, learning-based *execuTiOn plaN refInement Component.* To continuously capture and reuse optimal operator selections, *TONIC* uses a novel *Query Execution Plan Synopsis* (*QEP-S*). We demonstrate that extending state-of-the-art optimizers with *TONIC* substantially improves query response times with little overhead. Further, *TONIC* paves the way for new interesting research. In particular, combining *TONIC* with *Simplicity* enables the concept of a novel *two-stage cardinality estimation-free optimizer.*

ACKNOWLEDGMENTS

This work was funded by the German Research Foundation (DFG) within the RTG 1907 (RoSI). We would like to thank the anonymous reviewers for their valuable and constructive feedback.

REFERENCES

[1] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*. ACM, 261–272. https://doi.org/10.1145/335191.335420

[2] Shivnath Babu and Pedro Bizarro. 2005. Adaptive Query Processing in the Looking Glass. In *CIDR 2005*.

[3] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2001. STHoles: A Multidimensional Workload-aware Histogram. In *SIGMOD*. 211–222. https://doi.org/10.1145/375663.375686

[4] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *SIGMOD*. 18–35. https://doi.org/10.1145/3299869.3319894

[5] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 2001. Approximate query processing using wavelets. In *PVLDB*. VLDB Endowment, 199–223. https://doi.org/10.1007/s007780100049

[6] S. Chaudhuri. 1998. An overview of query optimization in relational systems. In *PODS*. ACM, 34–43. https://doi.org/10.1145/275487.275492

[7] Surajit Chaudhuri. 2009. Query Optimizers: Time to Rethink the Contract?. In *SIGMOD*. ACM, 961–968. https://doi.org/10.1145/1559845.1559955

[8] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. 2011. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases* 4, 1–3 (2011), 1–294. https://doi.org/10.1561/1900000004

[9] Asoke Datta, Yesdaulet Izenov, Brian Tsan, and Florin Rusu. 2021. Simpli-Squared: A Very Simple Yet Unexpectedly Powerful Join Ordering Algorithm Without Cardinality Estimates. *arXiv preprint arXiv:2111.00163* (2021).

[10] Kyotaro Horiguchi et al. 2021. Postgres pg_hint_plan extension. https://pghintplan.osdn.jp/pg_hint_plan.html. Accessed: 2021-4-20.

[11] R. Marcus et al. 2022. Bao documentation. https://rmarcus.info/bao_docs/. Accessed: 2022-05-11.

[12] W. Cai et al. 2020. Modified Postgres v. 9.6. https://github.com/waltercai/pqo-opensource. Accessed: 2020-08-07.

[13] Minos Garofalakis and Amit Kumar. 2005. Wavelet Synopses for General Error Metrics. 30, 4 (Dec. 2005), 888–928. https://doi.org/10.1145/1114244.1114246

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

[15] Axel Hertzschuch, Guido Moerkotte, Wolfgang Lehner, Norman May, Florian Wolf, and Lars Fricke. 2021. Small Selectivities Matter: Lifting the Burden of Empty Samples. In *SIGMOD*. ACM, 697–709. https://doi.org/10.1145/3448016.3452805

[16] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries!. In *PVLDB*. VLDB Endowment, 992–1005. https://doi.org/10.14778/3384345.3384349

[17] Yannis Ioannidis. 2003. The History of Histograms (Abridged). In *PVLDB*. VLDB Endowment, 19–30. https://doi.org/10.1016/B978-012722442-8/50011-2

[18] Yesdaulet Izenov, Asoke Datta, Florin Rusu, and Jun Hyung Shin. 2021. COMPASS: Online Sketch-based Query Optimization for In-Memory Databases. In *SIGMOD*. ACM, 804–816. https://doi.org/10.1145/3448016.3452840

[19] Yesdaulet Izenov, Asoke Datta, Florin Rusu, and Jun Hyung Shin. 2021. Online Sketch-based Query Optimization. *CoRR* abs/2102.02440 (2021).

[20] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. 2018. Cuttlefish: A lightweight primitive for adaptive query processing. *arXiv preprint arXiv:1802.09180* (2018).

[21] 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*.

[22] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating Cardinalities with Deep Sketches. *arXiv preprint arXiv:1904.08223* (2019).

[23] 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).

[24] Per-Ake Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. 2007. Cardinality estimation using sample views with quality assurance. In *SIGMOD*. ACM, 175–186. https://doi.org/10.1145/1247480.1247502

[25] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really?. In *PVLDB*. VLDB Endowment, 204–215. https://doi.org/10.14778/2850583.2850594

[26] 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. In *PVLDB*. VLDB Endowment, 643–668. https://doi.org/10.1007/s00778-017-0480-7

[27] Guy Lohmann. 2014. Is Query Optimization a "Solved" Problem? https://wp.sigmod.org/?p=1075. Accessed: 2019-09-23.

[28] R. Marcus. 2022. Bao Postgres extension. https://github.com/learnedsystems/BaoForPostgreSQL. Accessed: 2022-05-11.

[29] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *SIGMOD*. ACM, 1275–1288. https://doi.org/10.1145/3448016.3452838

[30] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).

[31] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *aiDM*. ACM. https://doi.org/10.1145/3211954.3211957

[32] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. 1998. Wavelet-Based Histograms for Selectivity Estimation. In *SIGMOD*. ACM, 448–459. https://doi.org/10.1145/276304.276344

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

[34] Parimarjan Negi, Ryan C. Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. In *PVLDB*. VLDB Endowment, 2019–2032. https://doi.org/10.14778/3476249.3476259

[35] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. 2019. How I Learned to Stop Worrying and Love Re-optimization. In *ICDE*. 1758–1761.

[36] Postgres Team. 2020. PostgresSQL. https://www.postgresql.org/. Accessed: 2020-07-22.

[37] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing. In *PVLDB*. VLDB Endowment, 96–107. https://doi.org/10.14778/2850583.2850585

[38] K-U Sattler, Eike Schallehn, and Ingolf Geist. 2004. Autonomous query-driven index mining. In *IDEAS*. IEEE, 439–448. https://doi.org/10.1109/IDEAS.2004.1319819

[39] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO-DB2's learning optimizer. In *PVLDB*. VLDB Endowment, 19–28.

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

[41] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2021. NeuroCard: One Cardinality Estimator for All Tables. In *PVLDB*. VLDB Endowment, 61–73. https://doi.org/10.14778/3421424.3421432

[42] Hansjörg Zeller and Jim Gray. 1990. An Adaptive Hash Join Algorithm for Multiuser Environments. In *PVLDB*. VLDB Endowment, 186–197.