Simple Adaptive Query Processing vs. Learned Query Optimizers: Observations and Analysis

Yunjia Zhang
University of Wisconsin-Madison
yunjia@cs.wisc.edu

Yannis Chronis
University of Wisconsin-Madison
chronis@cs.wisc.edu

Jignesh M. Patel*
Carnegie Mellon University
jignesh@cmu.edu

Theodoros Rekatsinas†
ETH Zurich
theo.rekatsinas@inf.ethz.ch

ABSTRACT

There have been many decades of work on optimizing query processing in database management systems. Recently, modern machine learning (ML), and specifically reinforcement learning (RL), have gained increased attention as a means to develop a query optimizer (QO). In this work, we take a closer look at two recent state-of-the-art (SOTA) RL-based QO methods to better understand their behavior. We find that these RL-based methods do not generalize as well as it seems at first glance. Thus, we ask a simple question: *How do SOTA RL-based QOs compare to a simple, modern, adaptive query processing approach?* To answer this question, we choose two simple adaptive query processing techniques and implemented them in PostgreSQL. The first adapts an individual join operation on-the-fly and switches between a Nested Loop Join algorithm and a Hash Join algorithm to avoid sub-optimal join algorithm decisions. The second is a technique called *Lookahead Information Passing* (LIP), in which adaptive semijoin techniques are used to make a pipeline of join operations execute efficiently. To our surprise, we find that this simple adaptive query processing approach is not only competitive to the SOTA RL-based approaches but, in some cases, outperforms the RL-based approaches. The adaptive approach is also appealing because it does not require an expensive training step, and it is fully interpretable compared to the RL-based QO approaches. Further, the adaptive method works across complex query constructs that RL-based QO methods currently cannot optimize.

PVLDB Reference Format:
Yunjia Zhang, Yannis Chronis, Jignesh M. Patel, and Theodoros Rekatsinas. Simple Adaptive Query Processing vs. Learned Query Optimizers: Observations and Analysis . PVLDB, 16(11): 2962 - 2975, 2023.
doi:10.14778/3611479.3611501

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

*Work done while at U. Wisconsin.
†Currently at Apple.

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611501

1 INTRODUCTION

Query optimizers (QOs) are performance-critical components of database management systems (DBMS) as they pick efficient physical query plans for input declarative queries. At the same time, QOs are notoriously hard to develop, fine-tune, and maintain. As a result, machine learning (ML) has been explored as a means to either improve the performance of an existing QO or to shorten the effort required to develop a new QO [44, 45, 57].

However, *learned query optimizers* have several potential drawbacks. From the perspective of a query optimizer developer, before applying a new method to the query optimizer, they need to understand two critical questions: 1) where does the improvement come from, and 2) when could the learned optimizer fail. As with all ML-based software, these two questions are difficult to answer for learned query optimizers. First, ML models have a black-box nature; their decisions and performance for different inputs are hard to explain. Second, learned QOs may overfit to the data and scenarios used during training, thus, exhibiting weak robustness to workload variations. This is a classic problem with ML-based software, and while recent works attempt to resolve it using methods such as transfer learning, it remains challenging. Finally, current learned QOs have focused mainly on non-complex query structures and it is unclear if they can generalize to complex SQL structures such as *common table expressions* (CTEs). Due to the above challenges, to the best of our knowledge, to date, no mainstream database engine (commercial or open source) has adopted a learned QO approach, which may indicate that there may be some way to go before learned query optimizers go mainstream. However, a learned QO is not the only approach that aims for intelligent query processing.

Adaptive query processing is a different popular approach to efficient query processing. There is a rich body of work here, as these techniques have been studied by the database community for decades (e.g., [11, 19, 24, 25, 28, 33, 35, 37, 59]). An adaptive approach to query processing aims to use *runtime methods* to adjust the query processing steps to achieve high performance. In contrast to a learned QO approach, an adaptive query processing approach does not intrinsically overfit to any workload since there is no static training step. Moreover, adaptive query processing techniques have already been adopted by several DBMS products, including Oracle [15], SQL Server [56], Spark SQL [52], and SQLite [29].

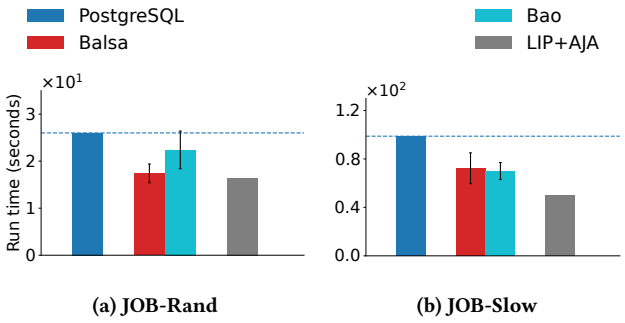
An interesting observation is that philosophically there is a key similarity between a reinforcement learning (RL) based optimizer and adaptive query processing. Compared with a traditional QO, a learned QO utilizes neural network-based models that are trained

on latency observations over static training workloads to 1) refine cost estimation and 2) predict a join order and configuration of physical operators that will minimize the processing time of a query. In an adaptive query processing engine, such cost estimations are replaced with more accurate runtime statistics. These statistics are then used to optimize both joins [59] and physical operators [19]. Effectively, both RL and adaptive query processing methods refine the traditional cost estimations to pick a more efficient plan. The adaptive approach performs the refinement using runtime statistics, whereas an RL-based QO uses a pre-trained model. Given this observation of their similarity, we ask the central question of the paper: *How do learned query optimizers to efficient query processing compare to an adaptive query processing approach?*

First, we analyze a state-of-the-art (SOTA) learned QO to obtain a better understanding of the key facts that lead to better query performance and its proneness to overfitting. To this end, we conduct an *ablation study* on the two optimization aspects that RL-based query optimization considers: 1) join order selection, and 2) physical operator selection for each join. We use Balsa [57], a leading RL-based QO that considers the entire optimization space of join orders and physical operators, to perform this study. We find that the join orders selected by Balsa contribute more towards improved performance, making join order selection a critical contribution of Balsa. Moreover, we find that these performance gains are greater for queries that have the same structural pattern as the queries in the training set. For queries with patterns that are not present in the training set, we find that using the plans generated by the learned QO may even lead to lower performance. Our evaluation highlights the importance of join order selection for improved performance, but it also highlights that the RL-based QO does not generalize effectively. Interestingly, adaptive query processing methods support a contrary view. They have shown that even in the presence of bad join orders, one can achieve efficient query execution times for equijoin queries [59].

To answer the central question of how RL-based query optimizers compare with adaptive query processing, we implement two representative adaptive processing techniques in PostgreSQL and evaluate them against two state-of-the-art learned query optimizers, Balsa [57] and Bao [44]. PostgreSQL was chosen as the platform for this comparison because it has been widely used in previous research on learned query optimization. These two adaptive processing techniques are well-established adaptive query processing techniques that are easy to implement and maintain, and have been widely used in industrial database systems [15, 29, 56]. Like RL-based query optimizers, these two techniques tackle two key aspects of adaptation: 1) adapting query processing to mitigate the impact of bad join orders, and 2) picking a performant join algorithm at runtime.

For the first aspect, we use a simple adaptive mechanism called *Lookahead Information Passing* (LIP) [59]. LIP generalizes semijoin techniques to optimize the pipeline of equijoin operations. It uses bloom filters to implement semijoins and uses an adaptive technique to order the filters at runtime. The adaptive ordering of the bloom filters results in the bloom filter sequence matching the optimal join order, and in practice, a bad query plan with LIP often has a performance that is similar to the performance of a query plan with an optimal join order [48, 59].



(a) JOB-Rand **(b) JOB-Slow**

Figure 1: A simple adaptive query processing (LIP+AJA) approach vs. two learned query optimizers (Balsa and Bao).

The second adaptive method is another simple approach, which is to simply pick the join algorithm at runtime to avoid common sub-optimal choices. An *adaptive join algorithm* (AJA) checks at runtime if a hash join algorithm should be switched to a nested loop join algorithm instead. Commercial systems often implement such adaptations. For example, this method has been used in SQL Server for many years [2, 56].

Our key finding is that with these two adaptive query processing techniques (*LIP+AJA*), query execution time is not only comparable to learned QOs (*Balsa* and *Bao*), but in many cases, *LIP+AJA* even outperforms the learned QOs in reducing the query run time. Figure 1 highlights a key result: Following the approach of Balsa [57], we derive two workloads from the *join order benchmark* (JOB) [41]: 1) JOB-Rand and 2) JOB-Slow. JOB-Rand workload tests the performance of learned QOs and *LIP+AJA* on randomly selected queries. It consists of 19 randomly selected queries in the test set, while the rest 94 JOB queries are used to train the learned QOs. As for JOB-Slow, it is designed to test learned QOs and *LIP+AJA* on the *slowest* queries in terms of PostgreSQL run time. JOB-Slow reserves *the 19 slowest queries* as the test set, and the remaining ones are considered as training queries.

Figure 1 shows that on the JOB-Rand workload, using the adaptive query processing (*LIP+AJA*) approach results in performance that matches *Balsa* and outperforms *Bao*. On the JOB-Slow workload, the adaptive *LIP+AJA* approach outperforms *Balsa* and *Bao*: *LIP+AJA* improves PostgreSQL by 2.0×, while *Balsa* and *Bao* improve PostgreSQL by 1.4× (additional details are presented in Section 6.2). Further, the adaptive approach with *LIP+AJA* also works with CTEs, which the learned QO methods have not been able to tackle so far. An additional advantage of the adaptive approach is that it does not require any expensive training/re-training step.

The key contributions of this paper are:

- We experimentally analyze a recent SOTA learned query optimizer, Balsa, and show that the learned QO-based approach may have key limitations related to generalization.
- We propose using a simple adaptive query processing approach (LIP+AJA) with two key techniques: Lookahead Information Passing (LIP) and Adaptive Join Algorithm (AJA). Like learned QOs, LIP+AJA tackles two orthogonal aspects that can lead to higher query performance: equijoin order selection, and the choices of physical equijoin algorithms.

Although the two techniques LIP and AJA have been separately proposed before, we combine these two techniques in a unified adaptive query processing framework.
- We evaluate the LIP+AJA approach on the commonly used JOB [41] and Stack [44] benchmark queries. We show that this approach speeds up the JOB and Stack queries significantly and, in many cases, is better than what Balsa and Bao can achieve.
- We demonstrate the versatility of the LIP+AJA approach by applying it to six TPC-H queries that contain CTEs. A speedup of 1.4x was observed in this workload, which learned QOs have not been able to tackle to-date.

2 BACKGROUND

This section introduces the necessary background and the terminologies that are used throughout the paper.

2.1 Query Optimization

There are two critical optimization dimensions considered by most query optimizers: 1) join order, and 2) physical operators. Both dimensions largely affect the execution latency of the query plans.

Join Order. In a query plan with multiple equijoins (the focus of this paper and also the focus of previous work on ML-based QO), the join order defines the order of relations in which the equijoins are performed. A good join order starts with the join operation that prunes most of the data and minimizes the data that is passed to subsequent joins. While optimizing the join order is critical to the efficiency of a query plan, obtaining a good join order is challenging given the large exponential search space and inaccurate cost estimation models [41].

Physical Operators. In a physical plan, the physical operators define the algorithm that is used to perform the logical operations. In this paper, we only consider the two important operators, namely scan and equijoin. There are two commonly supported physical operators for the scan operation: 1) index scan, and 2) sequential scan. An index scan can be more efficient when a large number of records are eliminated by the selection criteria on the indexed attribute, otherwise, a sequential scan is more efficient. Three algorithms are commonly used for the equijoin operation: 1) hash join, 2) nested loop join, and 3) merge join. Hash join can be more efficient for large relations; nested loop join can be a better choice for small relations; merge join can outperform the other two when the two input relations are sorted on the join key. The choice of physical operators is also critical when assembling an efficient query plan.

Cost Models. The goal of a query optimizer is to select a plan that minimizes the execution cost. Since the real execution cost cannot be retrieved upfront, a query optimizer uses a *cost model* to make an estimate, and use a search algorithm to identify which choice of join order and physical join algorithms will minimize the query cost given the cost model [50]. However, the cost model often provides weak estimates of the true cost of a query plan due to inaccurate cardinality estimates, especially for intermediate results [41]. This phenomenon is especially true when there are data correlations and/or dependencies in the underlying database. Thus, even an exhaustive exploration of the plan search space (using dynamic programming [8, 50]) can produce a suboptimal query plan with a high execution cost.

2.2 Reinforcement Learning in QO

Reinforcement Learning (RL) has recently become a popular foundation for proposals that aim to rethink query optimizers [44, 45, 57]. In this paper, we focus on two SOTA RL-based query optimizers: Balsa [57] and Bao [44].

In Balsa, *states* correspond to partial plans, and *actions* correspond to steps in building a query plan. An RL-based optimizer generates query plans in a bottom-up fashion: it starts with a single scan node and builds plans by adding new joins to the current node, constructing *partial plans*, until there are no tables that can be added to the join. For example, consider a query that needs to join the tables $\{A, B, C\}$. The RL-based optimizer starts by considering performing a single scan over one of the tables A, B, or C. Given a chosen table, the next possible actions correspond to joining this table with one of the other tables. For example, if Table A is chosen first, the next actions can be $A \bowtie B$ or $A \bowtie C$. The search proceeds in an iterative manner until the desired join conditions are met.

At the core of Balsa lies a cost model powered by a neural network (NN). The input of the NN is a (partial) physical query plan with information on both the join order and the physical operators. Given this input, the NN outputs a real number that estimates the overall minimum query latency corresponding to the plan [57]. At each step during the search, partial plans are given to the learned cost model, and the cheapest partial plan is kept for further searching. The NN-based cost model is trained on collected true query latencies by executing physical query plans in the underlying DBMS. During training, at each iteration, the RL-based optimizer first generates plans for training queries with the current cost model. The generated plans are then fed into the DBMS execution engine to retrieve the true execution cost. Finally, the NN-based cost model is re-trained/updated using the plans and their corresponding execution costs. RL-based query optimizers also balance exploitation (selecting the min-cost plan) and exploration (selecting an unseen plan) to learn a more accurate cost model.

As for Bao, although Bao also adopts an NN-based cost model that predicts the performance of query plans, the action space of Bao is restricted by a limited set of hints that steer the underlying query optimizer. These hints specify the usage of a combination of different join operators (hash, merge, and nested loop joins) and scan operators (sequential, index, and index-only scans) [4]. Bao predicts hints of physical operators for a given query and relies on the underlying query optimizer to select the join order. Given a query, Bao first builds query plans for each of the hints by passing the query with hints to the underlying query optimizer, and then uses the NN-based cost model to select a plan to execute. The training loop for Bao is similar to that of Balsa: it uses the actual execution cost of the query plan as the training ground truth, and it also strikes a balance between exploitation and exploration.

3 ANALYZING LEARNED QUERY OPTIMIZERS

Given a declarative query, the optimization space considered by learned query optimizers is 1) the order in which joins are evaluated and 2) the physical operators. To better understand the individual

Table 1: Test query sets from JOB, Stack, and TPC-H.

	# of queries	avg. # of join	min. # of join	max. # of join
JOB-Join-1	23	4	3	4
JOB-Join-2	18	6	5	6
JOB-Join-3	21	7	7	7
JOB-Join-4	21	8	7	8
JOB-Join-5	21	11	10	11
JOB-Join-6	9	14	13	16
JOB-Rand	19	7	4	11
JOB-Slow	19	7	4	11
JOB-Rand-CV	19×5	-	-	-
Stack-Join-1	1,172	3	3	3
Stack-Join-2	1,108	5	5	5
Stack-Join-3	2,409	6	6	6
Stack-Join-4	1,202	7	7	7
Stack-Join-5	100	8	8	8
Stack-Join-6	200	10	10	11
Stack-Rand	1,238	6	3	11
Stack-Slow	1,238	6	3	11
Stack-Rand-CV	$1,238 \times 5$	-	-	-
TPC-H-Complex	6	4	3	8

impact of these two dimensions, we conduct an ablation study on the predictions of the learned query optimizer. For learned query optimizers, these optimization decisions can be made either directly (Balsa) or indirectly through hints provided to the underlying optimizer (Bao). Since Bao only focuses on steering hints for physical operators and relies on the underlying query optimizer for join orders, we choose to use Balsa for our study in this section as it covers the entire optimization space.

3.1 Where do the Improvements Come from?

Balsa jointly predicts join orders and physical operators, and both predictions contribute to better query plans. To uncover the individual contributions of these dimensions, we perform an ablation study over each dimension by considering the following configurations:

(1) We consider using the exact query plan generated by Balsa, i.e., we use both the predicted join order and the physical operators in the generated plan; we refer to this configuration as *Balsa* in the figures and the discussions that follow.
(2) We consider using only the join order that is predicted by Balsa, and couple that decision with the physical join algorithms that are chosen by the original PostgreSQL optimizer; we refer to this configuration as *Balsa-JO*.

Our hypothesis is that if *Balsa-JO* is as good as *Balsa*, then the join order selection contributes more to the performance gains. Otherwise, picking the join algorithm is the more critical decision.

Benchmarks and Metrics. We evaluate Balsa on two workloads: 1) JOB [41] and 2) Stack [44]. We follow the analysis setup presented in the Balsa work [57] and consider two train-test splits of each workload: 1) randomly selected queries as the test set, and 2) the slowest queries (in terms of PostgreSQL run times) as the test set. Specifically, for JOB, we use JOB-Rand and JOB-Slow as described in Section 1. As for Stack, we create two similar train-test splits: 1) Stack-Rand and 2) Stack-Slow. In Stack-Rand, we randomly select 1,238 queries added to the test set and use the remaining ones as training queries. For Stack-Slow, we use the slowest 1,238 queries in PostgreSQL as the test queries, and the remaining queries are used to train the learned QOs. Table 1 summarizes these four workloads (along with other workloads introduced later in Section 6).

Given that different initialization of the Balsa model can result in diverse workload performance, we conduct five independent training of Balsa using distinct random initialization. In the following experiments, we report the *mean workload run times* of the five different predicted query plan sets, along with the *standard errors* of the mean workload run times.

Training Setups. Similar to [57], we use Balsa with PostgreSQL as our test framework. We explore four configurations:

- ***Balsa-overfit***. In this setup, all queries in the workload (all 113 queries in the JOB benchmark [41], or all 6,191 queries in the Stack benchmark [44]) are added to the training set. We use the trained model to predict the plans (both physical join algorithms and join orders) for the queries in the test set. With this setup, we seek to evaluate Balsa's memorization performance and the best-case improvements that it can provide for fixed query workloads.
- ***Balsa-overfit-JO***. In this setup, we use the same model as *Balsa-overfit*, i.e., a model trained on all queries, but we only consider the predicted join order. Then, we use the PostgreSQL optimizer to select the join algorithms. This configuration allows us to understand the contribution of the join order in isolation.
- ***Balsa***. This setup is the original Balsa configuration. The model is trained only on the training portions of the JOB-Rand and the JOB-Slow workloads.
- ***Balsa-JO***. This configuration uses the learned Balsa model obtained by the previous configuration but, again, we only consider the predicted join order provided by *Balsa* and use the PostgreSQL optimizer to select the join algorithms.

For all settings of Balsa, we limit the wall-clock time to 10 hours (including training, planning, and execution) for all JOB train-test splits, and 100 hours for all Stack train-test splits. We use the model after the last iteration to predict the query plans. We choose this cutoff time as it matches the one used in the Balsa paper [57]; we also use the implementation provided by the authors [5].

Results. The results on the four workloads (JOB-Rand, JOB-Slow, Stack-Rand, and Stack-Slow) are shown in Figure 2. For JOB-Rand (Figure 2a), *Balsa-overfit* improves the workload execution time over PostgreSQL by 2.5× (15.6s saved) when the tested queries are also present in the training set. If we only apply the join orders predicted, *Balsa-overfit-JO* yields a comparable speedup of 2.1× (12.2s saved) over PostgreSQL. This result highlights that when the queries are in the training set, the join order predicted by Balsa's RL model is the critical decision that improves execution time.

When trained and tested on different queries, *Balsa* has a speedup of 1.5× (8.6s saved). As expected, the improvement is lower than that of *Balsa-overfit*. If we only use the join orders generated by the learned model, i.e., we use the *Balsa-JO* configuration, we observe similar average runtime as the original PostgreSQL , and with larger standard errors. In addition, the performance gap between the *Balsa*

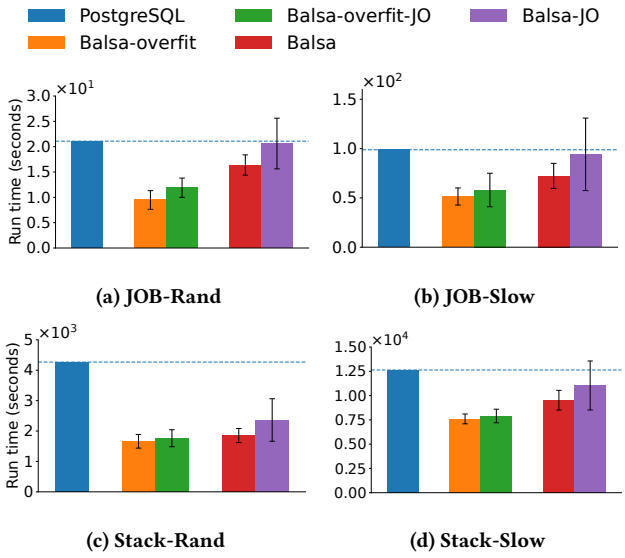


Figure 2: Performance comparisons of Balsa when all queries are used for training (Balsa-overfit) and when only the join order prediction is considered (JO).

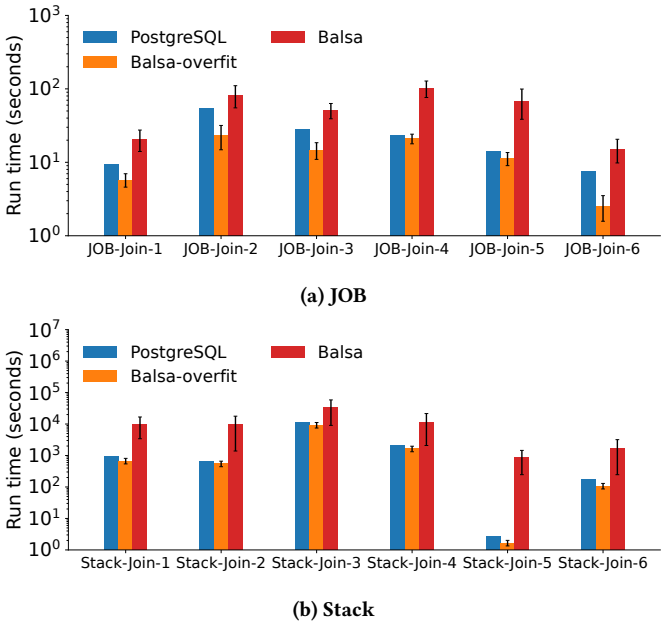


Figure 3: Balsa on queries with different numbers of joins.

and the *Balsa-JO* configurations is not statistically significant as the standard error intervals overlap, which aligns with our observation that join orders contribute to performance improvement. Our findings are similar for JOB-Slow, Stack-Rand, and Stack-Slow as shown in Figure 2b, 2c, and 2d.

Takeaways. Our analysis reveals that the join order decision plays a more significant role in performance improvements. The performance difference between utilizing both join order and join algorithm decisions and relying only on join order selection is not statistically significant. In addition, given that *Balsa-JO* has a larger standard error compared with *Balsa* and they share the same join orders, we conclude that *Balsa* predicts join orders that are closely tied with some specific join algorithms to have better run times, which the PostgreSQL query optimizer is not capable of predicting.

3.2 How Generalizable is a Learned QO?

Next, we turn our attention to the generalization properties of Balsa. From the results in Section 3.1, we know that Balsa generates good plans in many cases. However, since both the JOB and Stack queries are built from underlying query templates [41, 44], the training sets in the JOB-Rand, JOB-Slow, Stack-Rand, and Stack-Slow workloads have underlying structural similarities. Thus, the results described in Section 3.1 do not fully characterize how Balsa performs on completely "new" queries, especially queries with new join patterns. Such generalization is important in practice, especially when the workload changes frequently. To this end, we experimentally study how Balsa generalizes to new join patterns.

Benchmark and Training Setup. To evaluate the generalization of *Balsa* to unseen join patterns, we split the queries in the JOB and Stack benchmarks into six groups with increasing number of joins in the query groups. These groups are denoted as JOB-Join-1 to JOB-Join-6 for the JOB benchmark, and Stack-Join-1 to Stack-Join-6 for the Stack benchmark in Table 1. JOB-Join-1 and Stack-Join-1 have the least number of joins, while JOB-Join-6 and Stack-Join-6 with the most join operations. We perform a *cross-validation* using the six groups of queries. In each fold, we hold out the selected test query group, and train *Balsa* on the remaining JOB or Stack benchmark queries. Below, we also report *Balsa-overfit*, which is trained on all the queries in the benchmarks (Section 3.1).

Results. The results for JOB and Stack benchmarks are shown in Figure 3. On JOB benchmark (Figure 3a), *Balsa-overfit* improves query performance from 1.1× (JOB-Join-4) to 3.0× (JOB-Join-6) compared to PostgreSQL. However, when *Balsa* is used to optimize queries with join patterns that are not in the training set, it is unable to improve query performance. We find that the query plans generated by *Balsa* yield worse run times compared to vanilla PostgreSQL(compare the blue and red bars). For the Stack benchmark shown in Figure 3b, *Balsa-overfit* constantly surpasses PostgreSQL, while *Balsa* cannot produce query plans with better run time since the join patterns are not seen in the training set.

Takeaway. We show that for queries with join patterns not in the training set, the plans generated by *Balsa* lead to worse execution time than the plans created by the original PostgreSQL optimizer.

4 ADAPTIVE QUERY PROCESSING

Adaptive query processing takes an approach in which rather than assuming that a query plan generated by a query optimizer is optimal, it optimizes select aspects of the query processing pipeline by leveraging runtime measurements and pipeline characteristics.

Adaptive query processing and RL-based optimizers share a similar "objective": minimize the processing cost for join and physical operators. In contrast to an RL-based QO, which uses a pre-trained

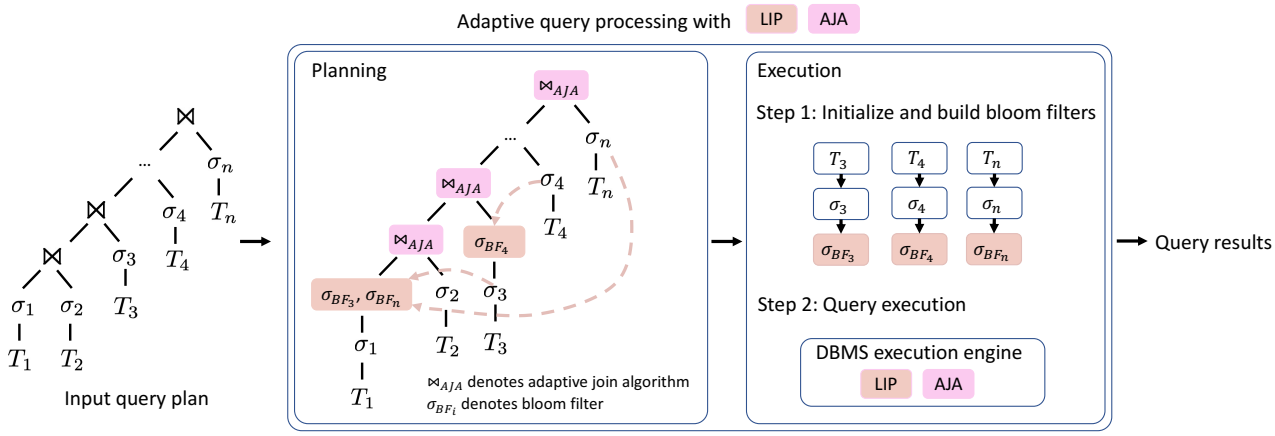


Figure 4: Overview of adaptive query processing with LIP+AJA.

cost model to estimate plan costs, adaptive query processing optimizes joins and physical operators using statistics collected at runtime. Since adaptive query processing is not trained, it is intrinsically not overfitting to any query pattern or data set.

While there are a large number of adaptive query processing techniques in the community (e.g., [2, 21, 26, 33, 38]), in this paper, we take a relatively simple approach and pick two techniques to build a rudimentary adaptive query processing framework. Our goal is not to build and evaluate a comprehensive adaptive query processing framework (that is an orthogonal research topic), but to show how a simple adaptive framework compared to learning-based QO approaches. The simple adaptive query processing framework that we consider in this paper combines two existing adaptive mechanisms: *Lookahead Information Passing* (LIP) and *Adaptive Join Algorithm* (AJA).

4.1 The LIP+AJA Adaptive Framework

Figure 4 shows an overview of our proposed LIP+AJA framework. This framework takes a physical query plan as input and optimizes the execution of that plan. The input is a plan that can be generated by *any* query optimizer (learned or not). For all experiments, we use the PostgreSQL optimized plans as input to our adaptive framework, except for the experiments discussed in Section 6.3 where LIP+AJA uses plans that are generated by Balsa and Bao, and for the experiment discussed in Section 6.4 where random plans are fed into LIP+AJA.

For a given execution plan, LIP+AJA first uses LIP to rewrite the execution plan by injecting bloom filters in the equijoin portion of the pipeline. Next, it replaces all physical join operators with adaptive join operators (AJA). Finally, it executes the rewritten plan using an adaptive execution strategy.

LIP generalizes the well-known semijoin technique to optimize an equijoin pipeline [59]. The main idea behind LIP is that a good join order applies the most selective join operation first, reducing tuples that are passed to subsequent join operations. Likewise, LIP uses lookahead bloom filters [59] to push down the selection predicates in the equijoin pipeline, making the query execution not sensitive to the join orders [59].

Specifically, during planning, for each selection predicate σ_i on T_i in the query plan, LIP examines whether there are tables T_j that equijoin with T_i on attribute a_{ij} in the "lower levels" of the query plan. If such T_j tables exist, then a lookahead bloom filter σ_{BF_i} is planned to push down σ_i to T_j. During execution, the bloom filter σ_{BF_i} is built by scanning $\sigma_i(T_i)$ and adding the set of join keys a_{ij} to σ_{BF_i}. Later when the DBMS execution engine is processing the join on T_j, each join key a_{ij} in T_j is tested for membership in σ_{BF_i}. If there are multiple bloom filters σ_{BF_i}s on the same table, LIP has an adaptive reordering mechanism of the bloom filters based on the collected runtime statistics [59]. This adaptive reordering mechanism makes the bloom filter probing order converge to the optimal join order over time. For further details about LIP, we refer the reader to [29, 48, 59].

In our adaptive query processing framework, all joins are executed using an adaptive join algorithm (AJA) that selects the join algorithm on-the-fly to avoid a bad join algorithm selection. AJA starts the join operation using a hash join algorithm by default, and if the number of keys inserted in the hash table is smaller than a threshold T, AJA switches to a faster nested loop join algorithm [2, 56].

There are two strategies of nested loop join algorithms that AJA considers: 1) simple (naive) nested loop join, and 2) index nested loop join [30]. Compared to simple nested loop join, index nested loop join algorithm utilizes the index built on the inner table join key to avoid exhaustive scanning of the inner table. Given the different complexity of the two nested loop join algorithms, we apply two different values of the threshold T ($T = V_{ind}$ or $T = V_{nonind}$) depending on the index availability on the inner side join key. If an index is available on the inner table join key and the built hash table is smaller than the threshold $T = V_{ind}$, AJA switches to an index nested loop join, using the rows in the built hash table as the outer loop and the other table with the index as the inner loop. Otherwise, if there is no such index, AJA uses the threshold $T = V_{nonind}$ to determine whether to switch the original hash join to a simple nested loop join.

The pseudocode for AJA is shown in Algorithm 1. AJA takes two join components R_1, R_2 (tables or intermediate results), along

Algorithm 1 Adaptive join algorithm (AJA)

Input: Join components R_1, R_2, threshold values V_{ind}, V_{nonind}
Output: join result R_o
1. $hash\_table = HashBuild(R_1)$ // *Build the hash table on* R_1.
2. if index is available on R_2 and $|hash\_table| \leq V_{ind}$:
3. $\quad R_o = IndexNestedLoop(hash\_table, R_2)$
4. else if index is not available on R_2 and $|hash\_table| \leq V_{nonind}$:
5. $\quad R_o = SimpleNestedLoop(hash\_table, R_2)$
6. else:
7. $\quad R_o = HashProbe(R_2, hash\_table)$ // *Continue with hash join.*
8. return R_o

with the two threshold values (V_{ind} and V_{nonind}) as inputs. The two threshold values V_{ind} and V_{nonind} are universal for the underlying database. As in SQL Server 2019 [2, 56], AJA starts by operating as a hash join operator. It first builds a hash table (Line 1). If there exists an index on the join key of R_2 and the number of keys in the hash table ($|hash\_table|$ in Line 2) is smaller than the threshold $T = V_{ind}$, it switches to an index nested loop join (Line 2 and 3). In the index nested loop join, the outer loop is built using the existing hash table to avoid rescanning R_1. If there is no such index on R_2, the threshold value V_{nonidx} is used to determine whether to switch to a simple nested loop join (Line 4 and 5). Finally, if AJA decides not to switch to any type of the nested loop join, it continues the hash join by probing the hash table with rows in R_2 (Line 7).

The two values V_{ind} and V_{nonind} of threshold T are system dependent and differentiate the costs between the hash join and the two flavors of nested loop join algorithms. We use a grid search method with a 1-join query template to determine the two values. We present how we pick the values of the threshold in Section 5.3.

4.2 Discussion

Compared to learned query optimizers, the adaptive LIP+AJA approach does not overfit to any data or query workload. The only input signals to LIP+AJA are the statistics collected during query execution, including the selectivities of the bloom filters (LIP) and the number of keys added to hash tables (AJA). This adaptive framework also provides potential robustness to bad input query plans: First, for a sub-optimal join order, LIP reduces data movement by pushing down a semijoin filter to a lower level in the query plan, and this method has been shown to be robust to a variety of input join orders [48, 59]. Second, AJA is robust to the original join algorithm selection since it picks the join algorithm on the fly. We experimentally demonstrate the robustness of LIP+AJA in Sections 6.3 and 6.4. In addition, since the bloom filter only uses an atomic bit-setting primitive in the construction phase and is read-only in the probe phase, LIP is fully parallelizable. Thus, LIP is compatible with parallel query execution plans.

We note that LIP+AJA also introduces runtime overhead. Specifically, LIP introduces the overhead of building and probing bloom filters, and AJA introduces the overhead of building and rescanning a hash table. In practice, these overheads are small, as described in Sections 5.4.

5 IMPLEMENTATION

Although LIP+AJA can be integrated into any DBMS, we chose PostgreSQL as the target DBMS as it is also the platform used in many previous works on learned QO. We implement LIP as a PostgreSQL extension and perform a proof-of-concept simulation of AJA. This section introduces the implementation and simulation details. We also analyze the overheads introduced by LIP+AJA.

5.1 LIP as a PostgreSQL Extension

We implemented LIP as a PostgreSQL extension. The input to LIP is the optimized plan from PostgreSQL. The LIP extension consists of multiple user-defined C PostgreSQL functions [9] to initialize, build, and probe the bloom filters. Table 2 lists the key functions that are used in the LIP extension. To execute queries with LIP , we rewrite the queries into two SQL blocks with these functions.

Bloom Filter Building Block. During initialization, the number of the bloom filters needed (cf. Section 4.1) for the query at hand is decided, and the memory space is declared in the corresponding shared memory (using the `pg_lip_bloom_init` function). LIP uses shared memory to store the bloom filters since PostgreSQL supports parallel query plans (for example, parallel sequential scan and parallel hash join) and it may start multiple workers that use the bloom filters at the same time. We use the method outlined in [18] to pick parameters for our bloom filters. We feed into the model in [18] a false positive rate threshold of 0.01 and a key threshold of 10^7, and get an optimized bloom filter configuration, which is 11 MiB in size and uses seven hash functions. To build a bloom filter for a table, we scan the table once and add the valid keys to the bloom filter using the function called `pg_lip_bloom_add`, which internally uses MurmurHash2 as the hash function.

Query Execution Block. We rewrite an input query using the probing function `pg_lip_bloom_probe` as a predicate. For a given key, it probes the bloom filter and returns a boolean value.

Example. We use Query 17a from the JOB benchmark to illustrate the use of the LIP extension functions. Figure 5 shows the original query and the LIP-rewritten query. To build the bloom filters, we add a SQL block that consists of queries to build the bloom filers using the functions `pg_lip_bloom_init` and `pg_lip_bloom_add`. Then, the original query is rewritten using subqueries to probe the bloom filters using the `pg_lip_bloom_probe` function.

5.2 Optimization Rules of LIP

Not all bloom filters that LIP builds are equally effective. Turning off ineffective bloom filters improves query execution time as both building and probing the filters introduce overhead. The effectiveness of a bloom filter depends on the number of rows pruned, and this cost has to be balanced with the overhead that is incurred in building and probing the bloom filter. While these decisions can be more tightly integrated within the core data processing engine (e.g., as was done in [48]), given our implementation that is based on making changes to PostgreSQL using only the external query rewrite and extensibility mechanisms (as is also the approach taken in ML-based QO), we use two heuristics to aid our implementation.

Optimizing Bloom Filter Building. First, if the predicate σ_i used to build the bloom filter cannot prune more than bf_build_pr of the join keys (according to the cardinality estimator in PostgreSQL),

Table 2: Key functions supported in the PostgreSQL extension of LIP.

Function	Parameter	Return
`pg_lip_bloom_init(int)`	The number of bloom filters initialized for a given query (integer).	-
`pg_lip_bloom_add(int, int)`	The index of the target bloom filter (integer); the value to be added (integer).	-
`pg_lip_bloom_probe(int, int)`	The index of the target bloom filter (integer); the value being probed (integer).	Boolean

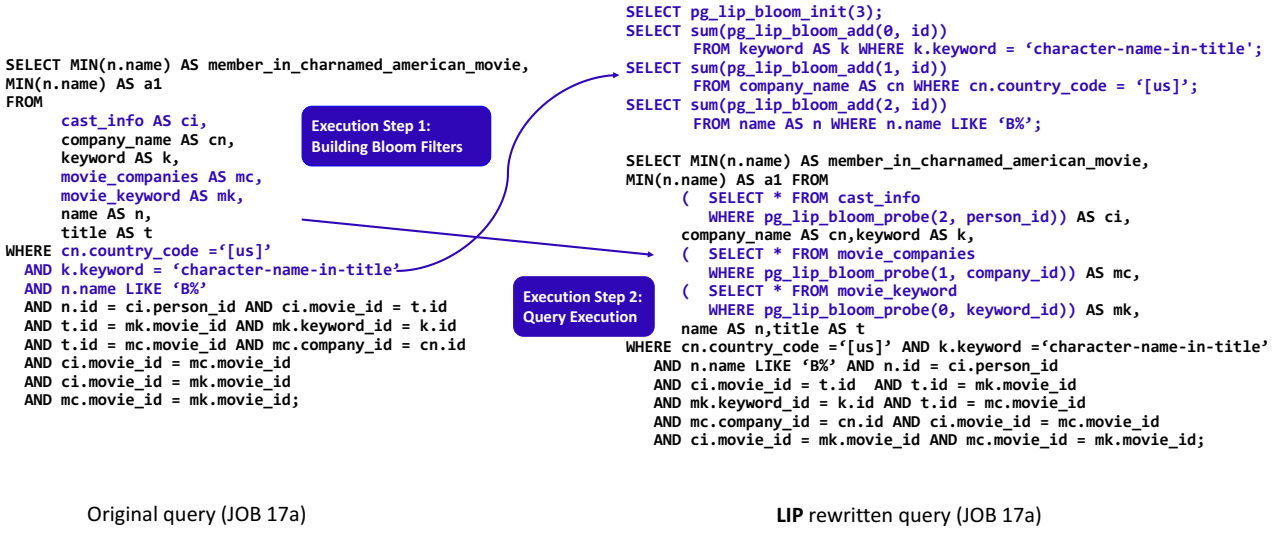


Figure 5: Query rewriting example with LIP.

then we turn off the bloom filter σ_{BF_i}; i.e., such bloom filters are deemed ineffective, and they are simply not constructed. Second, if a bloom filter σ_{BF_i} cannot be pushed to at least one level down in the input query pipeline, we turn off that filter. Collectively, these two heuristics prune bloom filters that are unlikely to be effective.

Optimizing Bloom Filter Probing. While we may choose to build a bloom filter σ_{BF_i} for a table T_i if σ_i is selective, when we apply σ_{BF_i} to the next table T_j in the join order, the filter σ_{BF_i} may not prune sufficient rows. Hence, paying the cost to probe the filter σ_{BF_i} may not result in improved performance. To ameliorate this situation, we introduce an optimization heuristic that adaptively disables the probing of specific bloom filters at runtime. During query processing, we measure the pruning rate of each filter on the first $valid\_rows$ rows that it scans. Any filter that prunes fewer than bf_probe_pr percentage of the input rows, is deactivated. This decision is remembered for the next $10 \times valid\_rows$ input rows that are scanned, after which the pruning rate of the filter is re-evaluated, and it is activated if needed. To further optimize the probing performance, if multiple bloom filters are activated on a single table, we rearrange the order in which they are probed, prioritizing the most selective filters first to minimize the number of bloom filter probes [59].

Discussion. The hyper-parameters bf_build_pr, bf_probe_pr, and $valid\_rows$ determine which lookahead bloom filters are built and probed. Intuitively, increasing bf_build_pr and bf_probe_pr results in fewer bloom filters being built and probed, leaving the most "effective" bloom filters in place. Decreasing the two constants loosens the restriction on the bloom filters, and more bloom filters will be built and probed. The appropriate setting for bf_build_pr and bf_probe_pr is dependent on the efficiency of the LIP implementation. Since our first version of LIP still has space for further optimization (see Section 5.4), we aggressively set bf_build_pr to 0.9, and bf_probe_pr to 0.1. For the parameter $valid\_rows$, we set it to a value of 1000 for all our experiments.

5.3 Simulating AJA

Algorithm 1 shows the adaptive join algorithm following SQL Server [2, 56]. In Algorithm 1, the hash table must be fully populated. For the simulation, we first execute the query with a hash join algorithm to determine the actual cardinality of each hash table.

Beyond the cardinality computation step, our simulation follows the same steps as Algorithm 1: if there is an index and the true cardinality collected is smaller than V_{ind}, we use an index nested loop join algorithm; if there is no such index available and the true cardinality collected is smaller than V_{nonind}, we apply a simple nested loop join algorithm; otherwise, AJA uses a hash join algorithm. We specify the physical join algorithm in PostgreSQL by adding execution hints to the query [1]. Since both the above simulation and a deeper implementation of AJA use the same true cardinality, the only runtime difference corresponds to the overhead of building (Line 1) and re-scanning the hash table (Line 3 or 5). We analyze this overhead in Section 5.4.

Determining Threshold Values V_{ind} and V_{nonind}. In Algorithm 1, the two values V_{ind} and V_{nonind} of threshold T determines when

AJA switches from a hash join algorithm to one of the nested loop join algorithms. As V_{ind} and V_{nonind} are relevant to the computation costs of the join operators, we perform a grid search using a simple select-join query template shown below from the IMDB data set [6]. In this template, we use a large fact table *cast_info* as R_2, which allows the generated threshold T to be "safe" for large tables. In practice, for data sets other than IMDB, this 1-join template can be created accordingly using the two largest fact tables as R_1 and R_2.

```
WITH R1 AS (SELECT * FROM title LIMIT n)
SELECT * FROM R1, cast_info as R2
WHERE R1.id = R2.movie_id;
```

Using the template, we generate queries with varying values of the parameter n. Each query is run using a hash join algorithm, and two nested loop join algorithms – index nested loop join and simple nested loop join (we construct the appropriate index on the underlying inner table to evaluate the index nested loop join option). In practice, by performing a grid search on n, we establish the two cut-off values: $V_{ind} = 500,000$ and $V_{nonind} = 1$.

5.4 Analyzing the Overheads

In addition to speeding up the query execution, applying LIP and AJA may also introduce overheads to the query execution. In this section, we describe such overheads in detail.

Overheads associated with LIP. There are two sources of overheads associated with the LIP mechanism: 1) the cost to build bloom filters, and 2) the cost associated with probing the bloom filters.

During the bloom filter building phase, we scan input tables and populate the bloom filters with join keys. This overhead grows linearly as the number of keys added.

During the probing phase, LIP calculates the hash of the key and checks the bloom filter for a hit. This overhead increases linearly with the number of probes. If the bloom filter is selective, fewer tuples will pass through the pipeline, reducing the effect of this overhead. The heuristic proposed in Section 5.2 aims to disable ineffective filters and reorder the enabled ones. The total building and probing overheads of LIP account for 26% of the *LIP+AJA* query execution time in the JOB-Rand workload and 18% in the JOB-Slow workload (cf. Figure 1).

Overheads of AJA. As shown in Algorithm 1, AJA introduces overheads if it decides to switch join algorithm. The overhead comes from building a hash table that is not used for the join, and the cost to rescan the hash table to start the nested loop join algorithm.

We estimate these overheads as follows: First, the hash table building phase can be pipelined with the scan phase [7], resulting in low latency impact, never exceeding two milliseconds in our experiments. Thus, the estimated hash table building overhead is $2 \times \mathbb{1}_{\{\text{AJA}\rightarrow\text{NL}\}}$, where $\mathbb{1}_{\{\text{AJA}\rightarrow\text{NL}\}}$ indicates whether AJA switches from hash join to a nested loop join. Additionally, we measured the scanning cost of an in-memory hash table, finding that PostgreSQL scans approximately 4000 rows per millisecond (in a single thread). If the total number of keys in the hash table is n, then the estimated hash table scanning overhead is $\frac{n}{4000} \times \mathbb{1}_{\{\text{AJA}\rightarrow\text{NL}\}}$. Thus, the total overhead is $(2 + \frac{n}{4000}) \times \mathbb{1}_{\{\text{AJA}\rightarrow\text{NL}\}}$. This overhead is low in practice since we only switch the join algorithm when n is small. For example, on JOB queries, AJA overhead takes less than 1% of the query execution time.

5.5 Discussion

As noted earlier, we have a simple implementation of LIP+AJA, which follows the philosophy of "simplicity in implementation" as previous papers in RL-based QO [44, 45, 57]. Nevertheless, there are opportunities to make this implementation more efficient. In our implementation, the bloom filter construction overhead is high as the input tables are scanned twice and the bloom filter probe functions are not efficient (cf. Section 5.4). A more sophisticated implementation directly in the core code of PostgreSQL could reduce these overheads. In addition, although our initial simulation of AJA provides a framework to carry out our proof-of-concept evaluations, implementing AJA as another joining method in PostgreSQL could make it more general.

6 EVALUATION

In this section, we experimentally answer the central question of this paper: *How do the SOTA RL-based query optimizers compare to the adaptive query processing with LIP+AJA?* Our key finding is that LIP+AJA is not only comparable, but it also outperforms RL-based query optimizers in many cases. In addition, we also evaluate:

(1) Can LIP+AJA further optimize the plans generated by learned query optimizers? See Section 6.3.
(2) How robust is LIP+AJA to random, possibly bad, query plans? See Section 6.4.
(3) How does LIP+AJA perform on queries with subplans? See Section 6.5.

6.1 Experimental Setup

Baselines. We use two representative but different RL-based query optimizers, *Balsa* and *Bao*, with PostgreSQL as our baseline methods. For *Balsa* and *Bao*, we use the source code provided by the authors [3, 5]. For JOB, we limit the wall-clock training time to 10 hours for both *Balsa* and *Bao*, including the time to train the neural networks, optimize the queries, and execute the workload. We selected such time limitations as similar training times are reported in [57]. As for Stack, since there is a larger number of queries in the workload, we increase the training time limit to 100 hours.

Our Method. For *LIP+AJA*, we use the query plans generated by the PostgreSQL query optimizer as input unless otherwise specified. We report the end-to-end execution time of *LIP+AJA*, including all the overheads introduced by *LIP+AJA* (cf. Section 5.4). As discussed in Section 4.2, LIP and AJA are compatible with parallel query execution plans. Therefore, we have enabled parallel plans in PostgreSQL and let PostgreSQL's optimizer determine the number of workers to be launched. Parallel execution as provided by PostgreSQL is also enabled for the learned QOs, *Balsa* and *Bao*.

Benchmarks. We use queries from three commonly used benchmarks, JOB [41], Stack [44], and TPC-H [49]. A summary of the test query sets is shown in Table 1.

JOB workloads. Same as Section 3.1, we follow the analysis setup presented in Balsa [57] and evaluate query performance on 1) randomly selected queries (**JOB-Rand**) and 2) slow queries (**JOB-Slow**). In addition, to reduce the impact of randomness in JOB-Rand, we employ a *Monte Carlo cross-validation* analysis and create **JOB-Rand-CV** workload. In JOB-Rand-CV, we randomly select 19

queries as the test set five times, and for each of the five test sets, the learned query optimizers are trained on the remaining queries.

Stack workloads. We also adopt the Stack workload [44] in our study. This query workload consists of more than six thousand analytical queries. As in Section 3.1, we adopt the two train-test splits: 1) **Stack-Rand** and 2) **Stack-Slow**. Similar to the case of the JOB workload, we create a **Stack-Rand-CV** workload to perform a *Monte Carlo cross-validation*. The Stack-Rand-CV workload consists of five sets of 1,238 randomly selected queries as the test queries.

TPC-H workloads. To evaluate *LIP+AJA* on queries with complex subplans, we select TPC-H queries [49] with joins and sub-plans to form a new query set, which we call **TPC-H-Complex**. Specifically, this workload contains the TPC-H queries Q2, Q7, Q8, Q9, Q20, and Q21. We use TPC-H data with a scale factor of 10 for this workload.

For each of the train-test configurations (except TPC-H-Complex), *Balsa* and *Bao* are trained on the training part of the workload and tested on the queries in the test set. For *LIP+AJA*, we directly run queries in the test set, as no training step is required in this case. Since current ML-based query optimizers are not able to tackle queries with complex SQL structures (e.g., CTEs) in the TPC-H-Complex workload, we only present results with *LIP+AJA*.

Metrics. We use the wall-clock query execution time as the main evaluation metric. We repeat *Balsa* and *Bao*-related experiments five times with different random initialization. We report the mean workload run times of the five different predicted query plan sets, along with the standard errors associated with the mean workload run times. For *LIP+AJA*, we also run the test workload five times and report the mean run time. Note that the run times reported for *LIP+AJA* include all the overheads described in Section 5.4.

System Configurations. We use a machine with 40 CPU cores and 250GB RAM for all experiments. We use an NVIDIA Tesla V100 GPU with 32GB of GPU memory to train the learned query optimizers (*Balsa* and *Bao*). We use a configuration of PostgreSQL similar to [41]: We use PostgreSQL 12.5 and set the memory limit per operator (`work_mem`) to 4GB, buffer pool size (`shared_buffers`) to 4GB, and the buffer cache size (`effective_cache_size`) to 32GB. We allow at most eight parallel workers to work on a query. In addition, we disable the genetic query optimizer in PostgreSQL, allowing queries with a large number of joins to utilize the same query optimizer as smaller queries. To allow faster query execution, we pre-construct indices on foreign key attributes in all datasets [41, 57].

6.2 Performance of LIP+AJA

We first evaluate the end-to-end performance of all methods on the JOB-Rand, JOB-Slow, Stack-Rand, and Stack-Slow workloads. In addition, we break down the JOB-Rand workload to show the query-by-query performance. Further, we evaluate *Balsa*, *Bao*, and *LIP+AJA* on different random sets of queries by performing a *Monte Carlo cross-validation* using JOB-Rand-CV and Stack-Rand-CV.

Performance. Figure 6 shows the performance of *LIP+AJA*, *Balsa* and *Bao*. On the JOB-Rand workload, *Balsa* and *Bao* improve the PostgreSQL by 1.5× (8.6s saved) and 1.2× (3.6s saved) respectively, while *LIP+AJA* improves performance by 1.5× (8.5s saved). *Balsa* and *LIP+AJA* have similar performance on random queries.

Turning our attention to JOB-Slow, we observe that *Balsa* and *Bao* both improve PostgreSQL by 1.4× on average (*Balsa* saved 26.5s, *Bao* saved 28.8s), while the improvement with *LIP+AJA* is 2.0× (48.9s saved). In addition, comparing with the training performance of learned query optimizers, *LIP+AJA* is comparable to *Balsa-overfit* (1.9x, 47.3s saved) and *Bao-overfit* (1.8x, 44.8s saved) on JOB-Slow.

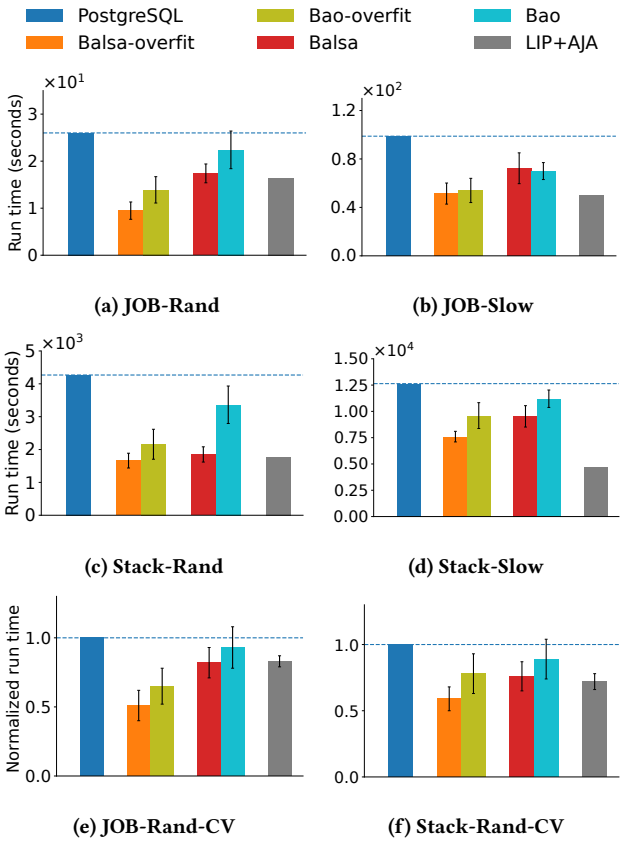


Figure 6: Performance of LIP+AJA vs. learned QOs.

As for Stack-Rand, *LIP+AJA* (2.5x, 2409s saved) has a similar performance to *Balsa-overfit* (2.6x, 2605s saved) and *Bao-overfit* (2.3x, 2417s saved). However, *LIP+AJA* outperforms *Balsa* (2.3x, 1851s saved) and *Bao* (1.3x, 905s saved). In addition, for Stack-Slow, *LIP+AJA* (2.6x, 7877s saved) can outperform the best-performing learned QO configuration *Balsa-overfit* (1.7x, 5038s saved).

Performance Breakdown. To demonstrate performance variations among different queries, we provide a runtime breakdown of the JOB-Rand workload. We compare three approaches, namely *Balsa*, *Balsa-overfit*, and *LIP+AJA*, with PostgreSQL, as depicted in Figure 7. Negative values indicate improved performance. The queries in JOB-Rand are sorted in descending order based on their PostgreSQL runtimes. Therefore, query 17e is the slowest query on PostgreSQL, while query 15b is the fastest query in Figure 7.

Among *LIP+AJA*, *Balsa*, and *Balsa-overfit*, the performance gain for the JOB-Rand workload is primarily in the slower queries (from 17e to 20a), which are also part of the JOB-Slow workload.

However, for the faster queries (from 22c to 15b), *Balsa-overfit* achieves a total runtime of 5.48s, outperforming PostgreSQL (7.15s).

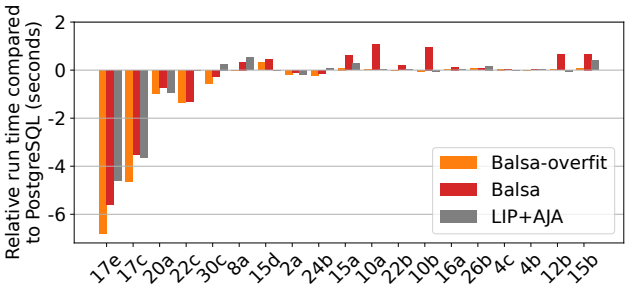


Figure 7: Performance breakdown of JOB-Rand.

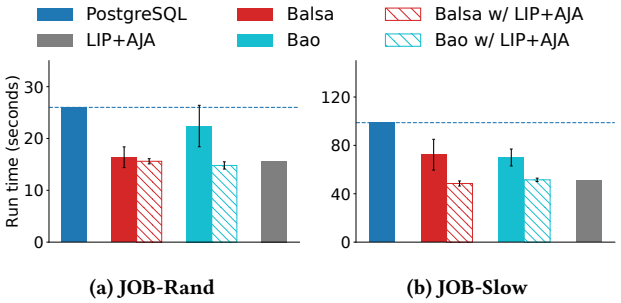


Figure 8: Applying LIP+AJA to plans from Balsa and Bao.

On the other hand, both *Balsa* (10.52s) and *LIP+AJA* (8.65s) introduce additional query runtime. The overhead in *LIP+AJA* is attributed to the construction of ineffective bloom filters. Our optimization rules (Section 5.2) do not prune these bloom filters due to the lack of awareness regarding data correlation. Future work on developing smarter optimization techniques can help mitigate this overhead. Despite the overhead, *LIP+AJA* still demonstrates similar average performance compared to *Balsa*. Additionally, *Balsa-overfit* performs similarly to PostgreSQL on these faster queries.

Cross-Validation. Figures 6e and 6f show the cross-validation results with JOB-Rand-CV and Stack-Rand-CV workloads respectively. Since the five random test query sets have different workload run times, we normalize the run time of each method relative to PostgreSQL for each instance. On average, *Balsa* reduces the normalized run time of JOB-Rand-CV to 0.82, while *Bao* achieves 0.92. *LIP+AJA* achieves a normalized run time of 0.83. For Stack-Rand-CV, *LIP+AJA* optimizes the average run time to 0.72, whereas the best non-overfitting configuration, *Balsa*, reduces it to 0.76.

Takeaway. The above results lead to our key finding in this paper: the performance of adaptive query processing using *LIP+AJA* is comparable to that of the most recent SOTA RL-based query optimizers, namely *Balsa* and *Bao*. In many cases, our "quick" implementation of *LIP+AJA* outperforms both *Balsa* and *Bao*. Additionally, it is worth noting that *LIP+AJA* does not require any training step and does not face the generalization challenge (Section 3.2).

6.3 Improving Plans Generated by Learned QOs

The general nature of the *LIP+AJA* method implies that it can be used to optimize *any* query plan. In this experiment, we explore the impact of using *LIP+AJA* on the query plans that are generated by learned query optimizers. Thus, in this experiment, we simply apply *LIP+AJA* on the query plans that *Balsa* and *Bao* generate. For this experiment, we use the JOB-Rand and the JOB-Slow workloads.

Results. Figure 8 shows the results, denoted as *Balsa w/ LIP+AJA* and *Bao w/ LIP+AJA*. For both the JOB-Rand and the JOB-Slow workloads, we observe that *Balsa w/ LIP+AJA* and *Bao w/ LIP+AJA* have performance that is similar to *LIP+AJA*, whose inputs are the plans generated by the PostgreSQL query optimizer. In addition, the standard errors are significantly reduced with *LIP+AJA*.

Takeaway. The above results show that *LIP+AJA* can also improve the execution of the query plans generated by the learned QOs, providing improvements in both workload execution time and robustness. Furthermore, our results demonstrate the robustness of *LIP+AJA*, as it performs similarly even when given "suboptimal" input plans generated by the PostgreSQL optimizer. This highlights the potential for *LIP+AJA* to effectively process different input plans. It is also important to note that in this experiment, the learned QOs are not trained with *LIP+AJA*. Thus, enabling *LIP+AJA* during the training may have the potential to further enhance performance.

6.4 Robustness of Adaptive Query Processing

In this experiment, we use randomly generated query plans to evaluate the robustness of LIP+AJA across a range of query plans. We generate several random query plans based on the JOB-Rand and the JOB-Slow workloads (as described below), and then feed these random plans as input to the adaptive query processing module.

Random Plan Generation. We generate random left-deep plans according to the query's join graph. We keep a list of tables as the join order, which is initialized by randomly selecting one table. We iteratively append one equijoin-able table until all the tables are included. We simply use the PostgreSQL optimizer to select the physical algorithms for the random plan. We repeat the generation five times, generating five random plan sets for both workloads.

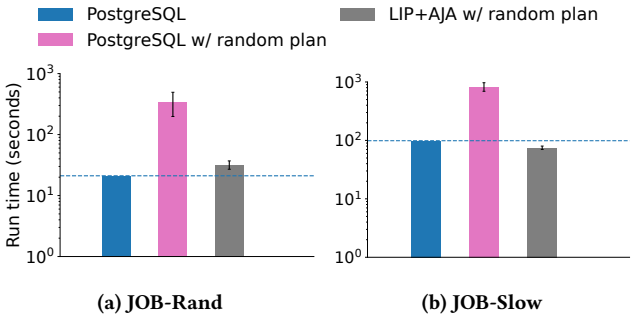


Figure 9: Performance of LIP+AJA on JOB-Rand and JOB-Slow with random input query plans.

Results. Figure 9 shows the results for this experiment. For these randomly generated query plans, *LIP+AJA w/ random plan* reduces both the run times and the standard errors over PostgreSQL with random plans. For the JOB-Rand case, *PostgreSQL w/ random plan*

has an average run time of 346 seconds, while *LIP+AJA w/ random plan* brings down the run time to 31 seconds. For JOB-Slow, *LIP+AJA* reduces the run time from 832 seconds to 75 seconds.

Takeaway. Although we do not exhaustively generate (the exponential number of) all random plans for the queries in the JOB-Rand and JOB-Slow workloads, the above results demonstrate the robustness of *LIP+AJA* to a variety of random input query plans.

6.5 Adaptive Query Processing on Subplans

Unlike learned query optimizers that have a static model architecture, LIP+AJA is more flexible and can be applied to queries that have complex SQL constructs. In this experiment, we test LIP+AJA on queries with complex subqueries using the TPC-H-Complex workload. Many of the queries in this workload contain an ORDER BY clause, making the latency sensitive to the order of the output rows. Since in the preliminary AJA (Algorithm 1), the order of the results may not be the same as originally planned (for example, if the join algorithm is switched to a nested loop join algorithm), applying AJA to order-sensitive queries may introduce extra reordering overhead. Thus, for this experiment, we let PostgreSQL's query optimizer choose the physical join operators, and we turn off the AJA mechanism when the query has an ORDER BY subclause.

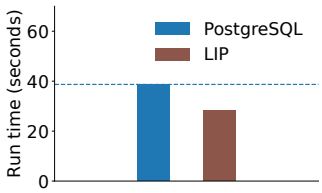


Figure 10: Performance of LIP on TPC-H-Complex.

Results. The results of this experiment are shown in Figure 10. For this workload, *LIP* improves PostgreSQL by 1.4×.

Takeaway. *LIP+AJA* can be used on complex SQL query constructs, making it more broadly applicable than current RL-based QOs.

7 RELATED WORK

The shortcomings of traditional optimization techniques have been well documented and often stem from estimation errors [41]. Query optimizers use estimation models to predict the cardinality of (partial) query plans, estimate their cost, and choose the most efficient plan. The simplifying assumptions used to build estimation models fail to accurately capture real data distributions, leading query optimizers to rely on predictions with large errors.

Many approaches have been proposed to address the mentioned shortcomings. Feedback loops can enhance estimate quality by using observed statistics from previous queries [10, 21, 22, 47]. Another approach allows for query plans to change during execution if the observed statistics differ significantly from the predicted statistics used at optimization time. Another set of techniques is to selectively re-optimize sub-optimal plans/sub-plans while minimizing the cost of re-optimization [17, 20, 23, 32, 35, 37]. Adaptive query processing is a third line of work that moves certain optimization decisions to the execution layer of the database engine, making execution more efficient and robust [11, 12, 14, 19, 24, 28, 31, 33, 54, 59]. Adaptive query processing generally delays making decisions until some execution statistics have been collected, and thus it dramatically reduces the dependence on having accurate estimates upfront. The granularity of the adaptiveness varies and includes tuning the physical implementation of an operator (e.g., [2, 19, 31]) and making the entire equijoin pipeline more robust (e.g., [36, 59]). LIP and AJA are selected as two typical adaptive methods addressing orthogonal aspects of equijoin order selection and join operator algorithms. LIP can be seen as a special case of SIP [36] as it reduces the tuples from the fact table by using the classic semijoin technique [13, 16, 43, 55] to minimize unnecessary data movement. LIP is also similar to the algorithm proposed in [58], but the reduction step and the join step in LIP are interleaved. LIP also optimizes the bloom filters in the query execution pipeline by introducing adaptiveness in both the building and probing phases, which can significantly improve its performance. For further details about LIP and its connection to related work, we refer the readers to [59].

The recent success of ML has led to new ideas to improve query optimization. To mitigate the impact of inaccurate estimates, ML models have been proposed for cardinality estimation [27, 34, 39, 42, 51, 53]. These learned cardinality estimators still need to work with a query optimizer to generate complete physical plans. Reinforcement learning (RL) has recently become popular for end-to-end query optimization [40, 44–46, 54, 57]. DQ [40] and ReJOIN [46] use a neural network to learn the plan enumeration strategy. Bao [44] optimizes queries by learning the optimization flags to set for each input query. Neo [45] and Balsa [57] train neural network-based cost models and generate plans that minimize the estimated costs.

8 CONCLUSIONS

In this paper, we compare a specific, simple, adaptive query processing approach, LIP+AJA, with two SOTA RL-based query optimizers, Balsa and Bao. We show that LIP+AJA is not only comparable to the RL-based optimizers in terms of query execution performance, but it is also better in many cases. In addition, the adaptive query processing approach LIP+AJA is able to optimize complex query constructs, which current RL-based query optimizers cannot tackle. Further, the adaptive approach does not require an expensive retraining step if the workload changes. Given the flexibility and effectiveness of adaptive approaches to query processing, our community should continue to acknowledge their appeal, and future ML-based query optimization proposals must consider comparing their approach with adaptive query processing techniques.

ACKNOWLEDGMENTS

This work was supported in part by DARPA under grant ASKEM HR001122S0005. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor. This work was also partly supported by funding from the Wisconsin Alumni Research Foundation. Additional support was provided by the National Science Foundation (NSF) under grants OAC-1835446 and CCF-2312739.

REFERENCES

[1] 2012. *pg_hint_plan.* Retrieved July 15, 2023 from https://pghintplan.osdn.jp/pg_hint_plan.html

[2] 2019. *Adaptive join in Microsoft SQL Server.* Retrieved July 15, 2023 from https://techcommunity.microsoft.com/t5/sql-server-blog/introducing-batch-mode-adaptive-joins/ba-p/385411

[3] 2020. *Bao source code repository.* Retrieved July 15, 2023 from https://github.com/learnedsystems/BaoForPostgreSQL

[4] 2020. *Hints used by Bao.* Retrieved July 15, 2023 from https://rmarcus.info/appendix.html

[5] 2022. *Balsa source code repository.* Retrieved July 15, 2023 from https://github.com/balsa-project/balsa/

[6] 2022. *IMDB dataset.* Retrieved July 15, 2023 from https://www.imdb.com/interfaces/

[7] 2022. *Overview of PostgreSQL Internals.* Retrieved July 15, 2023 from https://www.postgresql.org/docs/15/executor.html

[8] 2022. *Overview of PostgreSQL Query Optimizer.* Retrieved July 15, 2023 from https://www.postgresql.org/docs/12/planner-optimizer.html

[9] 2022. *PostgreSQL user-defined C functions.* Retrieved July 15, 2023 from https://www.postgresql.org/docs/current/xfunc-c.html

[10] Ashraf Aboulnaga and Surajit Chaudhuri. 1999. Self-tuning histograms: Building histograms without looking at data. *ACM SIGMOD Record* 28, 2 (1999), 181–192.

[11] L. Amsaleg, A. Tomasic, M.J. Franklin, and T. Urhan. 1996. Scrambling query plans to cope with unexpected delays. In *Fourth International Conference on Parallel and Distributed Information Systems.* 208–219. https://doi.org/10.1109/PDIS.1996.568681

[12] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Dallas, Texas, USA) *(SIGMOD '00).* Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/342009.335420

[13] Edward Babb. 1979. Implementing a relational database by means of specialzed hardware. *ACM Transactions on Database Systems (TODS)* 4, 1 (1979), 1–29.

[14] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. 2004. Adaptive Ordering of Pipelined Stream Filters. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04).* Association for Computing Machinery, New York, NY, USA, 407–418. https://doi.org/10.1145/1007568.1007615

[15] Pete Belknap, Ali Cakmak, Sunil Chakkappen, Immanuel Chan, Deba Chatterjee, Dinesh Das, Leonidas Galanis, Bruce Golbus, Shantanu Joshi, Tom Kyte, et al. 2013. Oracle Database SQL Tuning Guide, 12c Release 1 (12.1) E15858-15. (2013).

[16] Philip A Bernstein and Dah-Ming W Chiu. 1981. Using semi-joins to solve relational queries. *Journal of the ACM (JACM)* 28, 1 (1981), 25–40.

[17] Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. 2009. Progressive Parametric Query Optimization. *IEEE Transactions on Knowledge and Data Engineering* 21, 4 (2009), 582–594. https://doi.org/10.1109/TKDE.2008.160

[18] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[19] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. 2015. Smooth scan: Statistics-oblivious access paths. In *2015 IEEE 31st International Conference on Data Engineering.* IEEE, 315–326.

[20] Nicolas Bruno and Rimma V Nehme. 2008. Configuration-parametric query optimization for physical design tuning. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data.* 941–952.

[21] Chungmin Melvin Chen and Nick Roussopoulos. 1994. Adaptive selectivity estimation using query feedback. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data.* 161–172.

[22] Chungmin Melvin Chen and Nick Roussopoulos. 1994. Adaptive Selectivity Estimation Using Query Feedback. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA) *(SIGMOD '94).* Association for Computing Machinery, New York, NY, USA, 161–172. https://doi.org/10.1145/191839.191874

[23] Richard L. Cole and Goetz Graefe. 1994. Optimization of Dynamic Query Evaluation Plans. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA) *(SIGMOD '94).* Association for Computing Machinery, New York, NY, USA, 150–160. https://doi.org/10.1145/191839.191872

[24] Amol Deshpande, Joseph M Hellerstein, et al. 2004. Lifting the burden of history from adaptive query processing. In *VLDB.* Citeseer, 948–959.

[25] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. 2007. Adaptive Query Processing. *Found. Trends Databases* 1, 1 (2007), 1–140. https://doi.org/10.1561/1900000001

[26] Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. 2020. Bitvector-aware query optimization for decision support queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 2011–2026.

[27] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1044–1057.

[28] Kwanchai Eurviriyanukul, Norman W. Paton, Alvaro A. A. Fernandes, and Steven J. Lynden. 2010. Adaptive Join Processing in Pipelined Plans. In *Proceedings of the 13th International Conference on Extending Database Technology* (Lausanne, Switzerland) *(EDBT '10).* Association for Computing Machinery, New York, NY, USA, 183–194. https://doi.org/10.1145/1739041.1739066

[29] Kevin P Gaffney, Martin Prammer, Larry Brasfield, D Richard Hipp, Dan Kennedy, and Jignesh M Patel. 2022. SQLite: Past, Present, and Future. *Proceedings of the VLDB Endowment* 15, 21 (2022), 3535–3547.

[30] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)* 25, 2 (1993), 73–169.

[31] Goetz Graefe. 2011. A generalized join algorithm. *Datenbanksysteme für Business, Technologie und Web (BTW)* (2011).

[32] G. Graefe and K. Ward. 1989. Dynamic Query Evaluation Plans. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, USA) *(SIGMOD '89).* Association for Computing Machinery, New York, NY, USA, 358–366. https://doi.org/10.1145/67544.66960

[33] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Samuel Madden, Vijayshankar Raman, and Mehul A. Shah. 2000. Adaptive query processing: Technology in evolution. *IEEE Data Eng. Bull.* 23, 2 (2000), 7–18.

[34] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proceedings of the VLDB Endowment* 13, 7, 992–1005.

[35] Yannis E Ioannidis, Raymond T Ng, Kyuseok Shim, and Timos K Sellis. 1997. Parametric query optimization. *The VLDB Journal* 6, 2 (1997), 132–151.

[36] Zachary G Ives and Nicholas E Taylor. 2008. Sideways information passing for push-style query processing. In *2008 IEEE 24th International Conference on Data Engineering.* IEEE, 774–783.

[37] Navin Kabra and David J DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data.* 106–117.

[38] Srikanth Kandula, Laurel Orr, and Surajit Chaudhuri. 2019. Pushing data-induced predicates through joins in big-data clusters. *Proceedings of the VLDB Endowment* 13, 3 (2019), 252–265.

[39] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[40] 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).

[41] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[42] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality estimation using neural networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering.* 53–59.

[43] Lothar F Mackert and Guy M Lohman. 1986. R* optimizer validation and performance evaluation for local queries. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data.* 84–95.

[44] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making learned query optimization practical. *ACM SIGMOD Record* 51, 1 (2022), 6–13.

[45] 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).

[46] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management.* 1–4.

[47] Volker Markl, Guy M Lohman, and Vijayshankar Raman. 2003. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal* 42, 1 (2003), 98–106.

[48] Jignesh M Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A data platform based on the scaling-up approach. *Proceedings of the VLDB Endowment* 11, 6 (2018), 663–676.

[49] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *ACM Sigmod Record* 29, 4 (2000), 64–71.

[50] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data.* 23–34.

[51] Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. 2020. Astrid: accurate selectivity estimation for string predicates using deep learning. *Proceedings of the VLDB Endowment* 14, 4 (2020).

[52] Apache Spark. 2018. Spark SQL, DataFrames and datasets guide.

[53] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *arXiv preprint arXiv:1906.02560* (2019).

[54] Immanuel Trummer, Samuel Moseley, Deepak Maram, Saehan Jo, and Joseph Antonakakis. 2018. Skinnerdb: regret-bounded query evaluation via reinforcement learning. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2074–2077.

[55] Patrick Valduriez and Georges Gardarin. 1984. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems (TODS)* 9, 1 (1984), 133–161.

[56] Bob Ward. 2019. *SQL Server 2019 Revealed: Including Big Data Clusters and Machine Learning*. Springer.

[57] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. *arXiv preprint arXiv:2201.01441* (2022).

[58] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *VLDB*, Vol. 81. 82–94.

[59] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust: Making the Initial Case with in-Memory Star Schema Data Warehouse Workloads. *Proc. VLDB Endow.* 10, 8 (apr 2017), 889–900. https://doi.org/10.14778/3090163.3090167