Debunking the Myth of Join Ordering: Toward Robust SQL Analytics

Junyi Zhao
Tsinghua University
Beijing, China
zhaojy20@mails.tsiinghua.edu.cn

Kai Su
Tsinghua University
Beijing, China
suk23@mails.tsinghua.edu.cn

Yifei Yang
University of Wisconsin-Madison
Madison, USA
yyang673@wisc.edu

Xiangyao Yu
University of Wisconsin-Madison
Madison, USA
yxy@cs.wisc.edu

Paraschos Koutris
University of Wisconsin-Madison
Madison, USA
paris@cs.wisc.edu

Huanchen Zhang*
Tsinghua University
Beijing, China
huanchen@tsinghua.edu.cn

arXiv:2502.15181v1 [cs.DB] 21 Feb 2025

Abstract

Join order optimization is critical in achieving good query performance. Despite decades of research and practice, modern query optimizers could still generate inferior join plans that are orders of magnitude slower than optimal. Existing research on robust query processing often lacks theoretical guarantees on join-order robustness while sacrificing query performance. In this paper, we rediscover the recent Predicate Transfer technique from a robustness point of view. We introduce two new algorithms, `LargestRoot` and `SafeSubjoin`, and then propose Robust Predicate Transfer (RPT) that is provably robust against arbitrary join orders of an acyclic query. We integrated Robust Predicate Transfer with DuckDB, a state-of-the-art analytical database, and evaluated against all the queries in TPC-H, JOB, and TPC-DS benchmarks. Our experimental results show that RPT improves join-order robustness by orders of magnitude compared to the baseline. With RPT, the largest ratio between the maximum and minimum execution time out of random join orders for a single acyclic query is only 1.6× (the ratio is close to 1 for most evaluated queries). Meanwhile, applying RPT also improves the end-to-end query performance by ≈1.5× (per-query geometric mean). We hope that this work sheds light on solving the practical join ordering problem.

ACM Reference Format:
Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. In *Proceedings of International Conference on Management of Data (SIGMOD 25').* ACM, New York, NY, USA, 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

A query optimizer is a critical and perhaps most difficult component to develop in a relational database management system (RDBMS). Despite decades of research and practice, modern query optimizers are still far from reliable [52]. Among the many challenges, join ordering is the crown jewel of query optimization. Determining an optimal join order requires not only an efficient algorithm to search the enormous plan space but also an accurate cardinality estimation of the intermediate results. The latter is extremely difficult despite recent efforts to bring machine learning techniques to the problem [51, 76]. The reality is that the optimizers today constantly generate plans that are orders of magnitude slower than optimal [35, 50, 55].

Prior research on robust query processing typically approaches the problem in two ways. The first is to prefer plans with more stable performance against cardinality estimation uncertainties during query optimization [18, 25, 42, 43]. Such a "conservative" plan, however, often sacrifices query performance, and there is no theoretical guarantee of the plan's robustness. Another approach (i.e., re-optimization) is to collect the true cardinalities of intermediate results and reinvoke the optimizer at query execution time to generate better (remaining) plans [17, 44, 45, 57, 66, 87]. Nonetheless, the overhead of materializing the intermediate results at pre-defined re-optimization points often offsets the benefit of switching to a more efficient plan.

Fortunately, the seminal Yannakakis algorithm offers encouraging theoretical results [83]. The algorithm guarantees a complexity linear to the input + output size for any acyclic query regardless of its join order. The key idea is to perform a full semi-join reduction on the input relations (i.e., the semi-join phase) before joining them (i.e., the join phase) so that the remaining tuples must appear in the query's final output. Despite the appealing theoretical guarantee, Yannakakis algorithm received few adoptions because of the costly semi-join operation.

The recent Predicate Transfer (PT) algorithm proposes to speed up the above semi-joins by building Bloom filters instead of full hash tables [80]. The original paper focused on the impressive performance advantages of the technique with an order-of-magnitude improvement over the default query plans on a prototype system. Although Predicate Transfer was inspired by the Yannakakis algorithm, it fails to inherit the strong theoretical guarantee for acyclic

*Huanchen Zhang is also affiliated with the Shanghai Qi Zhi Institute. Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD 25', June 22-27, Berlin, Germany
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

queries because the algorithm does not ensure a full reduction of the input relations.

In this paper, we rediscover Predicate Transfer from a robustness point of view. We propose **Robust Predicate Transfer** (RPT) with two new algorithms on top of the original PT to guarantee join-order robustness. We first introduce the `LargestRoot` algorithm, which finds a *join tree* of an acyclic query by constructing a maximum spanning tree on its weighted join graph, to warrant a full reduction in the semi-join phase (aka transfer phase in RPT). To guarantee the robustness in the join phase of RPT, we propose the `SafeSubjoin` algorithm to verify the "safety" of a join order (i.e., its runtime cost is at most a constant factor away from the optimal) if the query is not γ-acyclic.

We implemented the Robust Predicate Transfer algorithm in DuckDB, a state-of-the-art in-process analytical database management system. The modifications to DuckDB were non-invasive: we introduced two new operators for building and probing Bloom filters and inserted an RPT optimization step/submodule into the optimizer's workflow. Our evaluation includes the three most widely used benchmarks for analytical workloads: TPC-H [2], JOB [3], and TPC-DS [1]. We measure the *join-order robustness* of a query using the performance gap between executing different random join orders. The smaller the gap, the more robust the query.

The experimental results are promising. Compared to the baseline (i.e., DuckDB without RPT integrated), RPT improves the robustness factor (i.e., ratio between the maximum and minimum execution time out of 200 random join orders) by orders of magnitude for acyclic queries (which accounts for 94% of the queries in the benchmarks). RPT allows most queries to have a robustness factor close to 1, and the largest performance gap between the best and worst join orders is only 2.8× among all the acyclic queries in TPC-H, JOB, and TPC-DS. We then zoomed in and verified the robustness of the `LargestRoot` algorithm. Furthermore, applying RPT improves the end-to-end execution time per query by ≈1.5× (geometric mean) over the baseline. We also concluded that it is not worthwhile to consider bushy plans for RPT because they brought little performance gain compared to left-deep in our evaluation.

The implications of our results could impact the design of future query engines and optimizers. With Robust Predicate Transfer, join order optimization is no longer a critical challenge for acyclic queries because of RPT's strong theoretical guarantee and practical efficiency. Future optimizers could limit their search space to left-deep plans (or simply pick a random join order) and become much more tolerant against cardinality estimation errors. Despite our promising results in achieving practical join order robustness, whether an instance-optimal join algorithm exists for cyclic queries remains an open problem.

We make three primary contributions in this paper. First, we propose two new algorithms (with rigorous proofs) to make Predicate Transfer robust against arbitrary join orders. Second, we show that our Robust Predicate Transfer algorithm is easy to integrate by implementing it in DuckDB, a state-of-the-art analytical system. Finally, we discover through experiments that RPT exhibits outstanding robustness while improving the overall query performance at the same time, a big step toward solving the practical join ordering problem.

2 Preliminaries

In this section, we first discuss the challenges and prior efforts in solving the join ordering problem. We then describe the Yannakakis algorithm and Predicate Transfer in detail.

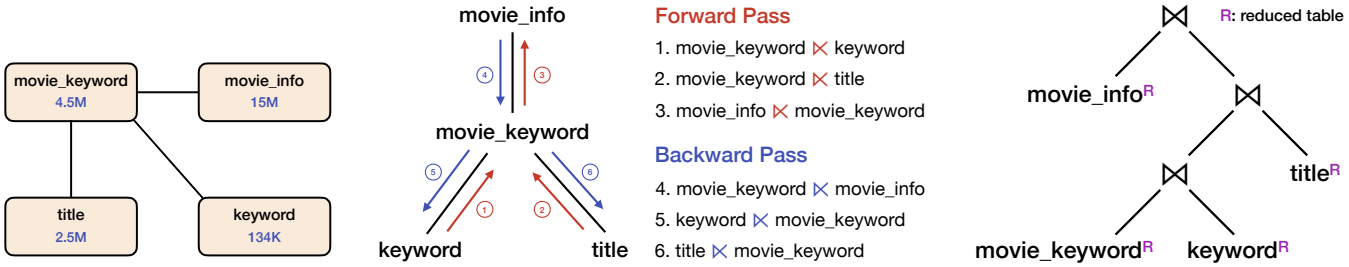
2.1 Join Order Optimization

Optimizing join orders is one of the most important tasks in query optimization. A bad join order leads to large intermediate results and can be orders of magnitude slower than the optimal plan [35, 52, 72]. Obtaining an optimal join order in a modern query optimizer requires accurate cardinality estimation and efficient plan enumeration. Both remain difficult after over 40 years of research.

Cardinality estimation (CE) predicts the number of tuples produced by each operator in a query plan. Obtaining accurate estimations of the join cardinalities is extremely difficult. Without detailed statistics, the query optimizer typically makes the following assumptions: (1) Uniformity: values in a column are uniformly distributed within the global min/max; (2) Independence: values in different columns are uncorrelated; (3) Inclusion: every value from the probe side of the join must appear in the build side. These assumptions are rarely valid in real-world applications. Although having further statistics (e.g., histograms on joint distributions) can improve the accuracy of join cardinality estimation, it is prohibitively expensive to maintain comprehensive cross-column statistics. Even worse, studies show that a small estimation error will propagate exponentially with respect to the number of joins [41, 84]. Leis et al. [52] reported that none of the optimizers in real-world DBMSs (including the commercial ones) can estimate join cardinalities accurately: most of them under-estimate by 2-4 orders of magnitude when the number of joins ≥ 5. Recent proposals tackle the CE problem using machine learning and deep learning techniques [29, 33, 36, 37, 47, 54, 65, 73, 79, 81, 82, 90], but none of them so far has shown evidence of robust estimation.

Plan enumeration refers to the process of searching equivalent join orders and finding the query plan that has the minimal cost. Plan enumeration has been proven to be NP-hard [40]. Prior work developed efficient algorithms based on dynamic programming [60, 61], and they are sufficient when the number of joins is small (e.g., < 10). However, because the search space grows exponentially with respect to the number of joins, it becomes impractical to perform an exhaustive search for a query with a large number of joins. Optimizers must fall back to heuristics-based approaches (e.g., the genetic algorithm in PostgreSQL [5] and the greedy algorithm in DuckDB [6]), sacrificing plan optimality for a reasonable optimization complexity.

A query execution is **robust** if its performance is never too far from the optimal even if the cardinality estimations are way off (which is inevitable) [35, 84]. Under the context of join ordering, it means that the risk of choosing a catastrophic join order due to wild CE errors is low. There are typically two ways to improve the robustness of SQL execution. The first is to favor a "**robust plan**" rather than the cost-optimal one to take into account the uncertainty during query optimization [18, 25, 42, 43]. For example, the optimizer would estimate cardinalities using intervals (rather than single values) [19] or probability distributions [18] and choose plans that have stable costs within certain confidence intervals. However,



(a) Join Graph (b) Semi-Join Phase (c) Join Phase

Figure 1: Yannakakis algorithm on JOB 3a.

such a robust plan may not exist, and the plan chosen often exhibits a noticeable performance hit compared to the optimal [84].

The second approach to improving plan robustness is through **re-optimization** [17, 22, 28, 34, 44, 45, 57, 66, 87]. The main idea is to correct CE mistakes while executing the query. Re-optimization must define specific materialization points in the query plan (usually at pipeline breakers) and collect statistics to obtain the true cardinalities at those points. If there is a large gap between the true cardinality and the estimated one, the system will re-invoke the optimizer, hoping to generate a better plan for the remaining operations. Although re-optimization enables self-correction at run time, materializing intermediate results is often costly and might compromise the end-to-end query performance.

2.2 Yannakakis algorithm & Predicate Transfer

An alternative thought of approaching join-order robustness is to design a join algorithm with bounded intermediate result sizes. Given a join query Q, let N be the total number of tuples in all the input relations, and let OUT be the number of tuples in the query output. The classic **Yannakakis algorithm** [83] guarantees a query complexity of $O(N + OUT)$[1], which is the same as simply scanning the input and writing the output. Therefore, Yannakakis algorithm is instance optimal. The key idea is to pre-filter the tuples in the input relations that will not appear in the final output. The pre-filtering is realized via a series of semi-join reductions. A semi-join $R \ltimes S$ outputs tuples from the left relation that have a match in the right relation. More formally, $R \ltimes S = \pi_{\text{attr}(R)}(R \bowtie S)$. In other words, a semi-join uses the right table as a filter to eliminate unmatched tuples in the left table.

Given a *join graph* of a query where each vertex is a table scan, and each edge represents an equi-join (e.g., Figure 1a), the Yannakakis algorithm first picks an arbitrary vertex as root and obtains a *join tree* (e.g., Figure 1b) via the GYO ear removal algorithm [85]. The algorithm requires that the join graph is *acyclic* (α-acyclic to be precise [83]) so that a join tree always exists. Yannakakis algorithm then proceeds to the **semi-join phase** [20], consisting of a *forward pass* and a *backward pass*. In the forward pass, the algorithm traverses the join tree from leaf to root (e.g., post-order traversal). For each node R, suppose its children are $S_1, S_2, \cdots, S_n$. The algorithm performs semi-join reduction on R using all of its children (i.e., `for` $i = 1, 2, \cdots, n : R \ltimes S_i$). An example is shown in Figure 1b. Once the forward pass reaches the root, the algorithm starts the backward pass from root to leaf (e.g., level-order traversal). For each node R with its parent P, $R \ltimes P$ is performed. The backward pass ends when all the leaf nodes are visited.

After this, the Yannakakis algorithm enters the **join phase**, where normal binary joins (e.g., hash joins) are carried out on the reduced tables, as shown in Figure 1c. Each binary join must map to an edge in the join tree from the semi-join phase to guarantee a non-decreasing intermediate result. Because the semi-join phase ensures that *all* tuples that will not contribute to the query output are removed (i.e., a full reduction), the join phase is proven to complete in $O(OUT)$ time.

Although Yannakakis algorithm exhibits appealing theoretical guarantees, few modern database management systems adopt it because the traditional hash-table-based implementation of the semi-joins makes the algorithm slow. The recent **Predicate Transfer** (PT) technique proposed by Yang et al. [80] solves this performance problem by using Bloom filters to conduct approximate semi-joins in the Yannakakis algorithm. Specifically, for each $R \ltimes S$ in the semi-join phase (it is called the Predicate Transfer phase in PT), PT builds a Bloom filter $\mathcal{B}_S$ with the join keys in S and then uses the tuples in R to probe $\mathcal{B}_S$. If the probe returns false for a tuple t in R, t is eliminated. Otherwise, t is inserted into a different Bloom filter $\mathcal{B}_R$ (could use a different join key) to prepare for the next semi-join in either the forward or backward pass.

Compared to the original Yannakakis algorithm, Predicate Transfer trades a *small* accuracy loss (caused by false positives of the Bloom filters) for a faster semi-join reduction. An inaccurate pre-filtering result does not affect the algorithm's correctness because the false positives will be removed during the subsequent join phase. Besides performance improvement, Predicate Transfer generalizes Yannakakis algorithm to arbitrary join graphs, including cyclic ones. Instead of converting an acyclic join graph into a join tree, Predicate Transfer transforms any join graph into a DAG (i.e., a *transfer graph*) using a simple heuristic that assigns the direction of each edge from the smaller table to the larger one. Unfortunately, Predicate Transfer does not inherit the strong theoretical guarantee from Yannakakis algorithm for acyclic queries because it could generate transfer schedules that lead to incomplete semi-join reductions. We will propose a new algorithm to fix this in the next section. For

[1] Considering the query size to be constant.

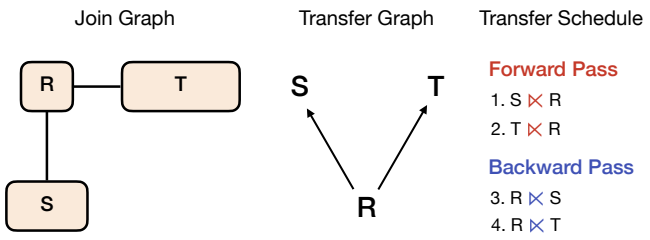


Figure 2: An example of the `Small2Large` algorithm in the original Predicate Transfer

cyclic joins, although Predicate Transfer improved the query performance empirically in many cases, there is no theoretical guarantee on the intermediate result sizes.

3 Toward Join-Order Robustness

This section introduces new algorithms with analyses to make Predicate Transfer robust for acyclic queries. In Section 3.1, we propose the `LargestRoot` algorithm in the transfer phase (i.e., the counterpart of the semi-join phase in Yannakakis) that not only guarantees a full reduction but also minimizes the Bloom filter construction time. Section 3.2 discusses approaches to guarantee that the join order selected in the join phase is "safe" (i.e., there is no intermediate result blowup).

3.1 Generating a Robust Transfer Schedule

The transfer phase in the original Predicate Transfer algorithm [80] adopts `Small2Large`, a simple heuristic-based algorithm to build the transfer graph. As described in Section 2.2, `Small2Large` assigns the direction for each edge in the (undirected) join graph from the smaller table to the larger table to form a DAG. Predicate Transfer then generates a *transfer schedule* (i.e., the forward and backward passes of Bloom filters) by following the edges in this DAG. The `Small2Large` algorithm, however, does not guarantee a full reduction for acyclic queries. As shown in Figure 2, for example, consider the natural join $R(A,B) \bowtie S(A,C) \bowtie T(B,D)$ where $|R| < |S| < |T|$. In this case, `Small2Large` will generate a transfer graph that leads to a forward pass of $S \ltimes_b R$ and $T \ltimes_b R$ followed by a backward pass of $R \ltimes_b S$ and $R \ltimes_b T$. This transfer schedule fails to "connect" S and T: if S has a predicate, this filter information can never reach T via the transfer of Bloom filters (and vice versa), leading to an incomplete reduction.

Although `Small2Large` cannot pre-filter all the non-result tuples, pushing larger tables toward the end of the transfer schedule is insightful because a smaller table is likely a more selective filter. The new `LargestRoot` will preserve this strategy while guaranteeing a full reduction. Before diving into our new algorithm, let us define the concepts of a *join tree* and *acyclicity* precisely.

Without loss of generality, we only consider natural joins with a connected join graph in this section[2]. For a natural join query q, its *join graph* G_q is an undirected graph where the vertices are the relations in q. If two relations have attributes in common, they are connected by an edge in G_q. A *join tree* T_q is a spanning tree of G_q such that for every attribute A, the relations containing A induce a *connected* subgraph T_q^A of T_q. The join tree is then used to define the *acyclicity* of a query:

[2]For equality predicates such as $R.A = S.B$, we treat A and B as the same attribute in this context. If the join graph has multiple components, we can generalize the concept of join tree to join forest

Algorithm 1: `LargestRoot`

Input: join graph G_q
Output: tree T

```
T ← ∅; R ← all relations; R' ← {R_max};
while R' ≠ R do
    Find an edge e = {R, S} ∈ E(G_q) with the largest weight
      such that R ∈ R \ R', S ∈ R'. Choose the edge with the
      largest R to break ties;
    Add e to T with direction from R to S;
    R' ← R' ∪ {R};
end
return T;
```

Definition 3.1 (α-acyclicity [83]). A natural join query q is *acyclic* if and only if there exists a *join tree* of q.

Acyclicity is crucial for Yannakakis algorithm to achieve the $O(N + OUT)$ complexity because it guarantees a non-decreasing intermediate result in the join phase. If the subgraph for an attribute A is *not* connected, a tuple may survive in the first join involving A but later get eliminated by the second join using A. This breaks the above non-decreasing property. An acyclic natural join satisfies the following lemma:

Lemma 3.2 ([56]). *Let q be an acyclic natural join query. For each edge (R,S) in the join graph G_q, where R and S are the vertices (i.e., relations), define the weight of the edge $w(R,S)$ as the number of shared attributes between R and S: $w(R,S) = |\text{attr}(R) \cap \text{attr}(S)|$. Then, a subgraph of G_q is a join tree of q if and only if it is a maximum spanning tree for G_q.*

The intuition behind the lemma is that for a spanning tree T of G_q, T's total weight equals the summation of the edge count of each attribute-induced subgraph. T is a join tree means that every attribute-induced subgraph T^A is connected. This is equivalent to saying that every T^A is a subtree, and it is impossible for any T^A to have more edges (otherwise T will not be a tree). T must be a maximum spanning tree (MST). Note that the weights defined on the edges are not considered heuristics for join costs. They are used to transform the problem of finding a join tree into the problem of finding an MST in the join graph.

We now know that for an acyclic query, a join tree guarantees a full (semi-join) reduction of the query, and we can find a join tree by constructing a maximum spanning tree on its weighted join graph. We next introduce our `LargestRoot` algorithm. As shown in Algorithm 1, we use Prim's algorithm to construct a maximum spanning tree T on the join graph G_q. The edges in T point from leaf to root, indicating a forward pass schedule. Because the algorithm starts with the largest relation R_{max} in $\mathcal{R}'$, R_{max} is the root of T (hence the name `LargestRoot`). And because of Lemma 3.2, T is a join tree if query q is acyclic, guaranteeing a full reduction in the transfer phase.

Placing the largest relation at the root of the join tree is important, especially for queries following a star schema. It is more efficient to filter the much larger fact table using the dimension tables first before building a Bloom filter on the fact table. In addition, `LargestRoot` pushes larger relations toward the root by including them early in T in the tie-breaking strategy in Line 3. This allows larger relations to get filtered first by probing other Bloom filters before building their own, thus minimizing the total Bloom filter construction time in the transfer phase. Notice that Line 3 in `LargestRoot` does not specify a tie-breaking policy for choosing $S \in \mathcal{R}'$. In reality, most edges have weight 1 because relations typically join on only one attribute. Although the choice of S does not compromise the theoretical guarantee of the algorithm producing an MST, it could affect the shape of the join tree. In general, a flatter tree allows more parallelism in building the Bloom filters, while a deeper tree might allow filtering irrelevant tuples out earlier. Both the tie-breaking policies for R and S do not affect the strong theoretical guarantee (i.e., a full reduction) of `LargestRoot`.

Unlike Yannakakis algorithm, `LargestRoot` also applies to cyclic queries. The algorithm's output is still a spanning tree with the largest relation at the root, but it is not a join tree. In this case, the transfer schedule generated by `LargestRoot` does not guarantee a fully reduced instance for the subsequent join phase. Still, it transfers any predicate to all relations at least once and is effective empirically, as we will show in the experiments in Section 5.

3.2 Choosing a Safe Join Order

Once the transfer phase generates a fully reduced instance of the database, the algorithm enters the join phase to produce the final output. According to Yannakakis algorithm, the join order is derived from the join tree used in the semi-join phase by performing the joins bottom up. Although such a (almost fixed) join order guarantees the asymptotic complexity of Yannakakis algorithm (i.e.,$O(N + OUT)$), it prevents the optimizer from exploring more join orders that potentially have smaller costs. Ideally, we want to leverage the cost models in the optimizer to search for cheaper plans, but we want to constrain the optimizer to only consider join orders with intermediate results always upper bounded by the output size. Such a "safe" join order provides a (theoretical) robustness guarantee: its runtime cost is at most a constant factor away from the optimal. In other words, even ill-behaved data distributions will not cause the runtime to deviate more than some bounded quantity.

Definition 3.3 ([12]). Let q be an acyclic natural join query. A subjoin q' of q is *safe* if for every fully reduced instance I, we have $q'(I) = \pi_{\text{attr}(q')}(q(I))$.

The above definition ensures that if a subjoin q' is safe, then the output of q' is a projection of the final output, and thus $|q'(I)| \leq |q(I)|$. If every subjoin of a join order is safe, then the cumulative intermediate result size is within a constant factor of $|q(I)|$ (i.e., the optimal). It is straightforward to see that subjoins that involve Cartesian products can be unsafe. But unsafe subjoins are not restricted to Cartesian products. Consider the natural join $q = R(A,B,C) \bowtie S(A,B) \bowtie T(B,C)$. Let I be the fully reduced instance: $R = \{(1,1,1),(2,1,2),\ldots,(n,1,n)\}$, $S = \{(1,1),(2,1),\ldots,(n,1)\}$, and $T = \{(1,1),(1,2),\ldots,(1,n)\}$. Then subjoin $q' = S(A,B) \bowtie T(B,C)$ is unsafe because $|q'(I)| = n^2$, while $|q(I)| = n$. Therefore, any query plan that joins S with T first – even on a fully reduced instance – will create a quadratic blowup on the intermediate result.

One approach to avoid unsafe join orders is to *identify the class of acyclic queries* for which any join order that does not involve Cartesian products is safe.

Definition 3.4 (γ-acyclicity [30]). A natural join query q is γ-acyclic if and only if there is no γ-cycle in q. This is equivalent to (1) q is α-acyclic, and (2) we cannot find three relations R, S, T with attributes x, y, z that form a γ-cycle of size 3: $R(x,y), S(y,z), T(x,y,z)$.

Lemma 3.5 ([30]). *Every connected join expression[3] θ of q is monotone (i.e., no tuple gets removed while executing any binary join in θ) if and only if q is γ-acyclic.*

Theorem 3.6. *Every subjoin (without Cartesian products) for natural join query q is safe if and only if q is γ-acyclic.*

Proof. It is sufficient to show that every subjoin is safe if and only if every connected join expression is monotone.

Consider any connected join expression θ' for subjoin q', θ'_1 for subjoin q'_1, and θ'_2 for subjoin q'_2, where $\theta' = \theta'_1 \bowtie \theta'_2$. Because every subjoin without Cartesian products is safe, we have $q'(I) = \pi_{\text{attr}(q')}(q(I))$, $q'_1(I) = \pi_{\text{attr}(q'_1)}(q(I))$, and $q'_2(I) = \pi_{\text{attr}(q'_2)}(q(I))$. Because $\text{attr}(q'_i) \subseteq \text{attr}(q')$ for $i = 1, 2$, we have $|\pi_{\text{attr}(q')}(q(I))| \geq |\pi_{\text{attr}(q'_i)}(q(I))|$. Therefore, θ' is monotone.

For the other direction, consider any connected join expression θ' of a subjoin q'. Extend θ' to a complete join expression θ of q. Because θ' is part of θ and every connected join expression of q is monotone, we have $q'(I) = \pi_{\text{attr}(q')}(q(I))$ for any fully reduced instance I. Therefore, q' is safe. □

Theorem 3.6 gives a strong **robustness guarantee**: if a query is γ-acyclic, we can fully trust the optimizer for join ordering on a fully-reduced instance (i.e., the join phase) because it can never pick an unsafe join order. γ-acyclic queries are a subset of α-acyclic (i.e., acyclic) queries according to Definition 3.4. To quickly check for γ-acyclicity in practice, it is sufficient (not necessary) to show that no two relations in the join graph are directly connected by more than one edge (i.e., no composite-key joins).

For queries that are acyclic but not γ-acyclic, we must *supervise the optimizer* to check whether a given subjoin is safe. A safe subjoin can be characterized by the following lemma:

Lemma 3.7 ([12]). *Let q be an acyclic natural join query. A subjoin q' of q is safe if and only if there exists some join tree of q such that the relations in q' are connected.*

For the example natural join $q = R(A,B,C) \bowtie S(A,B) \bowtie T(B,C)$, there is only one join tree for q: $S-R-T$. Hence, both $R \bowtie S$ and $R \bowtie T$ are safe subjoins, but $S \bowtie T$ is not. Using Lemma 3.7, we developed the `SafeSubjoin` algorithm to detect whether a subjoin q' is safe. As shown in Algorithm 2, `SafeSubjoin` first computes a maximum spanning tree T' for q' using the `LargestRoot` algorithm. It then continues to run another instance of `LargestRoot` by modifying the initialization step as $T \leftarrow T'$; $\mathcal{R} \leftarrow$ all relations in q; $\mathcal{R}' \leftarrow$ all relations in q'. `SafeSubjoin` returns true if the resulting spanning tree T is a maximum spanning tree of q (i.e., a join tree of q).

[3] No Cartesian products, binary joins only.

Algorithm 2: `SafeSubjoin`

Input: natural join q, subjoin q'
Output: True or False

```
T' ← LargestRoot(G_q');
T ← LargestRoot(G_q) with the initialization step as:
    T ← T'; R ← all relations in q; R' ← all relations in q';
if T is a maximum spanning tree of q then
    return True;
else
    return False;
end
```

4 Integration with DuckDB

We describe how to integrate the *Robust Predicate Transfer* (RPT) algorithm into DuckDB (v0.9.2) [69], a fast data analytics system, in this section. We first describe DuckDB's execution model and its optimizer briefly in Section 4.1. We next introduce the new Bloom filter operators in Section 4.2. Finally, we introduce the new Robust Predicate Transfer module that inserts the Bloom filter operators into the query plan based on the transfer schedule obtained by running the `LargestRoot` algorithm in Section 3.1.

4.1 DuckDB Preliminaries

DuckDB is a state-of-the-art in-process analytical database management system. It adopts a push-based vectorized execution engine, where each pipeline (i.e., a sequence of physical operators) processes tuples in batches (i.e., a data chunk, default batch size = 2048) to amortize the interpretation overhead and improve CPU parallelism. As shown in Figure 3, each physical operator can be in one of the following three roles: *source*, *operator*, and *sink*, depending on its position within the pipeline. The source implements the `GetData` function to retrieve a new data chunk at the beginning of a pipeline. Intermediate operators implement the `Execute` interface that computes on an input data chunk and then outputs the result chunk. The sink operator is located at the end of a pipeline and is typically a pipeline breaker. Its interface consists of three functions: `Sink`, `Combine`, and `Finalize`. `Sink` is called to receive and buffer the data chunks until incoming data is exhausted. Next, `Combine` and `Finalize` are called to perform some final computations to get ready to distribute data to the next pipeline (or final output). `Combine` is called once per thread, while `Finalize` is called when all threads are finished.

DuckDB's optimizer includes separate logical and physical optimization phases, as shown in Figure 4. Logical optimization performs a sequence of steps such as expression rewrite and filter pushdown, each of which is a separate submodule in the logical optimizer. DuckDB's join order submodule uses dynamic programming for join order optimization [61] and falls back to a greedy algorithm for large/complex join graphs.

4.2 Bloom Filter Operators

To implement Robust Predicate Transfer in DuckDB, we introduce two new physical operators based on Bloom filters: `CreateBF` and `ProbeBF`. We use the Bloom filter implementation from Apache Arrow 16.0 [4]. It is a blocked Bloom filter [67] with operations accelerated using AVX2 instructions. Because a vectorized probe to the Bloom filter returns a bit vector while DuckDB uses a selection vector to mark valid entries in a data chunk, we implemented an efficient bit-to-selection vector conversion according to [53].

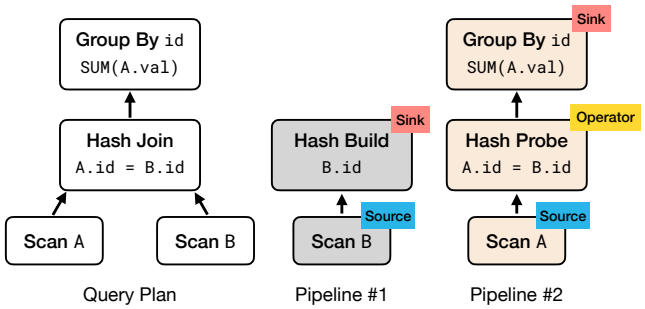


Figure 3: Example pipelines & operator roles in DuckDB [68]

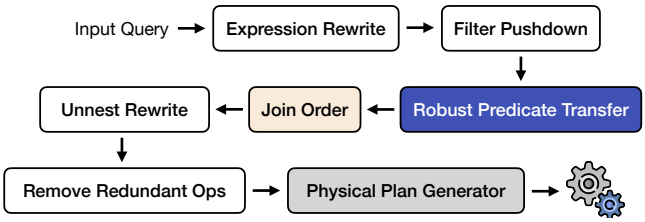


Figure 4: Workflow of DuckDB's optimizer

`CreateBF` is a physical operator that gathers/buffers the input data chunks and creates one or more Bloom filters on given columns. Its logical counterpart `LogicalCreateBF` will be used in the logical optimization. `CreateBF` can act as both a *sink* and a *source* (more about this in Section 4.3). In the `Sink` function, we receive input data chunks and keep them in thread-local buffers. No computation is needed for `Combine`. At `Finalize`, we traverse each thread-local data buffer to create a Bloom filter for each given column. The false positive rate (FPR) of the Bloom filter is set to 2% (Arrow's default). When using `CreateBF` as a source, we implement `GetData` by assigning each thread a disjoint range of chunk IDs in the data buffers for parallel scanning.

`ProbeBF` is another physical operator that outputs the Bloom filter result for each tuple in the input data chunk. Similarly, it has a logical counterpart `LogicalProbeBF`. `ProbeBF` is used as an intermediate operator. The `Execute` function takes in a data chunk, uses the tuples to probe the Bloom filter(s) in a vectorized fashion, and outputs the data chunk with an updated selection vector.

4.3 Robust Predicate Transfer Module

We introduce the Robust Predicate Transfer module in DuckDB's logical optimizer to insert `LogicalCreateBF` and `LogicalProbeBF` operators into the query plan, as shown in Figure 4. The RPT module constructs a join graph from the input plan and runs the `LargestRoot` algorithm to obtain a transfer schedule (including forward and backward passes). For each semi-join $R \ltimes S$ in the transfer schedule, we insert a `LogicalCreateBF` for S and a `LogicalProbeBF`

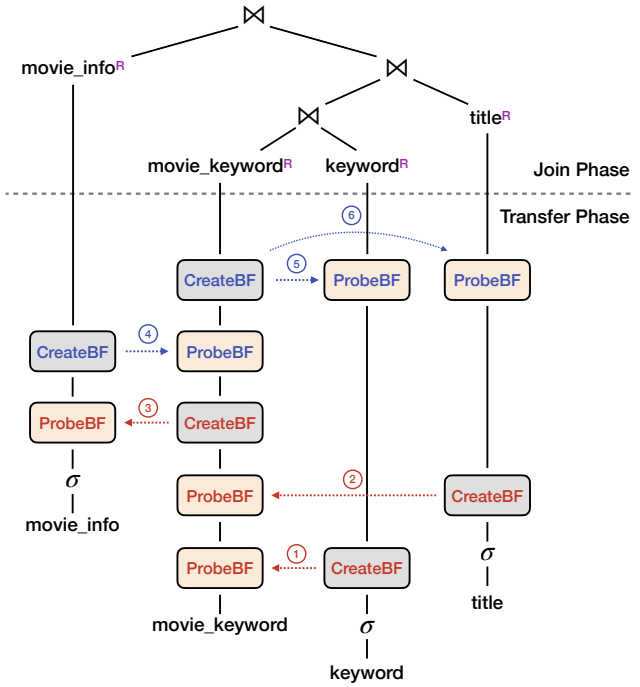


Figure 5: Query plan of JOB 3a with Robust Predicate Transfer integrated – red denotes forward pass while blue denotes backward pass.

for R using S's Bloom filter. These logical operators are later replaced by `CreateBF` and `ProbeBF` in the physical plan generator.

Take the query plan for JOB 3a as an example. Suppose the transfer schedule generated by `LargestRoot` is the same as that in Figure 1b. Then Figure 5 shows the physical plan after inserting `CreateBF` and `ProbeBF`. The solid black lines represent the flow of data chunks (from the bottom up) while the dashed red/blue arrows indicate the transfer of Bloom filters (via shared memory). Each `CreateBF` first acts as a *sink* operator that buffers the data chunks at the end of the pipeline and creates a Bloom filter. Then `CreateBF` functions as the *source* operator of the next pipeline, where it feeds the buffered data chunks to subsequent operators such as `ProbeBF` and hash join.

We also implemented optimizations to prune unnecessary Bloom filter operations. In particular, if the build-side relation in a primary-foreign-key join has not been filtered before, we can omit the pair of `CreateBF` and `ProbeBF` because the semi-join is trivial (i.e., it does not eliminate any tuple). We can also skip the entire backward pass if the transfer order aligns with the join order in the join phase.

5 Evaluation

We evaluate the robustness of RPT-integrated DuckDB in this section[4]. We conduct the experiments on a physical machine with two Intel® Xeon® Platinum 8474C @ 2.1GHz, 512GB DDR5 RAM, and 8TB Samsung 870 QVO SATA III 2.5" SSD. The operating system is Debian 12.5. We compare vanilla DuckDB (labeled as DuckDB) against DuckDB equipped with RPT (labeled as RPT) using four standard benchmarks: TPC-H (SF = 100) [2], Join Order Benchmark (JOB) [3], TPC-DS (SF = 100) [1], and DSB (SF = 100) [27]. We run the experiments under DuckDB's main-memory setting where the tables are pre-loaded and decompressed in the buffer pool. We examine the case where the base tables and intermediate results do not fit in memory in Section 5.4. We execute the queries using a single thread except for the multi-threaded experiments in Section 5.3.

Table 1: Robustness Factors for left-deep joins.

RF	TPC-H			JOB			TPC-DS		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
DuckDB	2.7	1.2	9.3	30.4	1.1	371	7.2	1.0	224
RPT	1.3	1.2	1.5	1.2	1.0	1.6	1.1	1.0	1.5

Table 2: Robustness Factors for bushy joins.

RF	TPC-H			JOB			TPC-DS		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
DuckDB	5.1	1.2	13.7	120	1.1	1747	35.0	1.0	1226
RPT	1.8	1.2	3.0	1.6	1.1	7.7	1.8	1.0	4.2

Table 3: Average speedups over DuckDB (optimizer's plan)

Speedup	TPC-H	JOB	TPC-DS	DSB
Bloom Join	1.15×	1.13×	1.05×	1.06×
PT	1.53×	1.46×	1.27×	1.18×
RPT	1.53×	1.46×	1.56×	1.54×

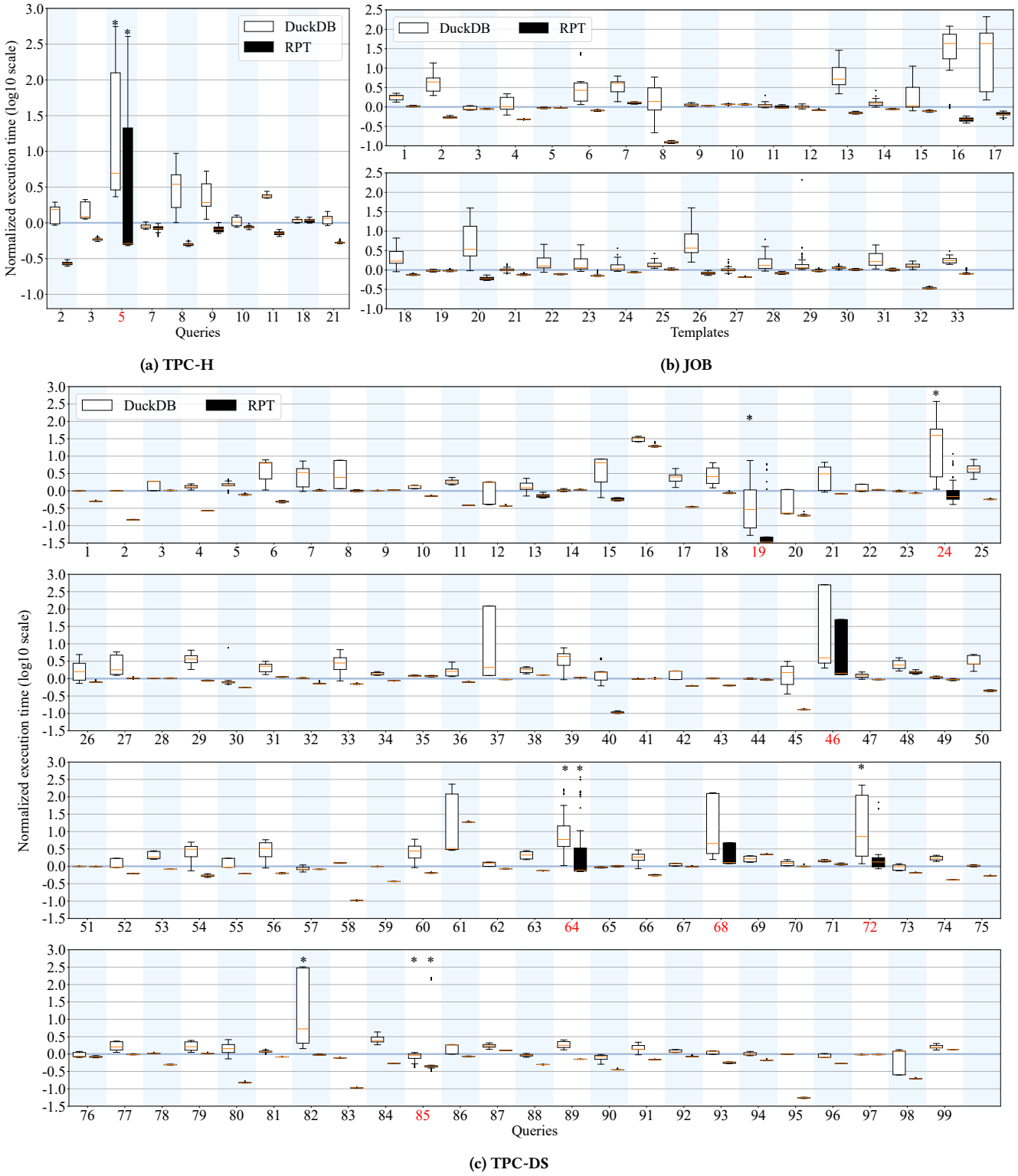
5.1 End-to-end Robustness

In the following experiments, we modified DuckDB's optimizer to generate random join orders. For each evaluated query, we generate N random *left-deep* plans and N random *bushy* plans, where N is proportional to the number of joins m in that query. Specifically, we set $N = 20$ for the simplest 3-join queries and $N = 1000$ for the most complex query (i.e., Query 29 from JOB) with 17 joins, and therefore $N = 70m - 190$ for $3 \leq m \leq 17$. To produce a left-deep plan, we randomly pick a base table that is joinable[5] with the current (intermediate) table as the right-most leaf at each iteration. For bushy plans, we randomly remove two joinable tables from the candidate set (which initially contains all base tables) and insert their intermediate table back at each iteration until the set contains only one element (i.e., the final plan).

5.1.1 Acyclic Queries (left-deep). Figure 6 shows the distribution of the end-to-end execution time of the random left-deep plans for each query. We omit queries in TPC-H with less than two joins because they are trivial in terms of join ordering. For JOB queries,

[4]Our source code can be found in https://github.com/zzjjyyy/PredTransDuckDB

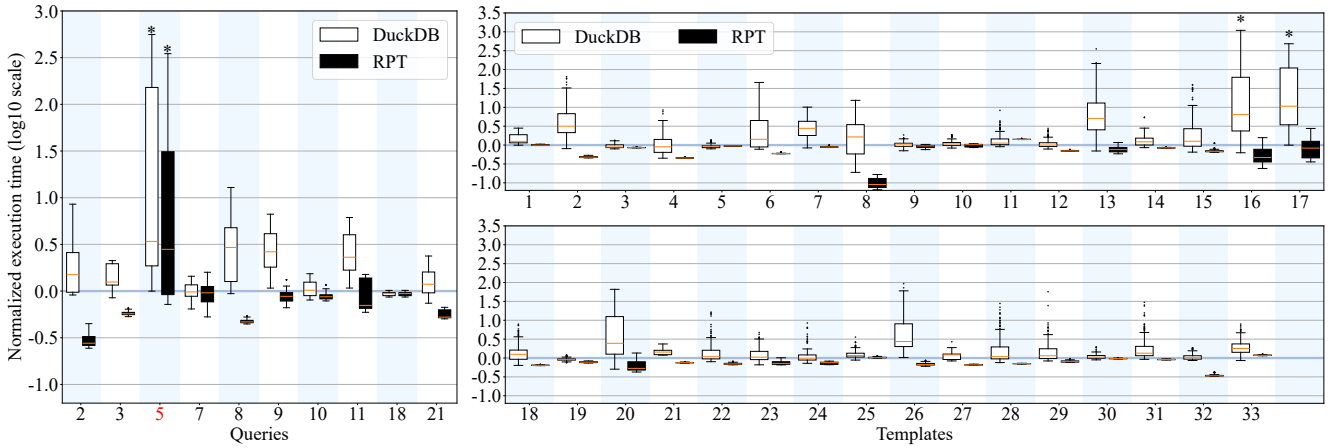
[5]Has an edge in the join graph, i.e., no Cartesian product.



(a) TPC-H (b) JOB

(c) TPC-DS

Figure 6: Distribution of the execution time of random left-deep plans for each query in TPC-H, JOB, and TPC-DS – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers). '*' indicates timeouts. Cyclic queries are in red.



(a) TPC-H (b) JOB

Figure 7: Distribution of the execution time of random bushy plans for each query in TPC-H and JOB – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers). '*' indicates timeouts. Cyclic queries are in red.

we present one result for each of the 33 query templates. The execution times (for the baseline and RPT) for each query are normalized by the time t_{opt} of DuckDB running its default optimizer's plan. The figure is in *log scale* with the normalization line (i.e., horizontal zero) highlighted. We set the timeout to $1000 \times t_{opt}$. The '*' above a bar indicates that at least one of the random plans incurs a timeout for this query. Cyclic queries are marked by red query numbers.

We observe impressive join order robustness when using RPT in DuckDB for *all* acyclic queries. To quantify this, we define the *Robustness Factor* (RF) as the ratio between the maximum and the minimum execution time. Table 1 presents the average, min, and max RFs for DuckDB and RPT in each benchmark. The average RF for DuckDB with RPT is consistently near 1 with the max (i.e., worst-case) RF = 1.473 for Query 13 in TPC-DS. This is orders-of-magnitude more robust compared to the baseline.

Additionally, applying RPT improves the end-to-end query performance for most queries in the benchmarks (most RPT boxes are below 0 in Figure 6). Table 3 presents the average speedups of RPT over the default DuckDB running its optimizer's plan (i.e., t_{opt}). We also included Bloom Join [23] and the original Predicate Transfer [80] (PT) as references[6]. Besides robustness guarantees, applying RPT reduces the execution time per query by $\approx 1.5\times$ on average (geometric mean). Bloom join only achieves a marginal speedup over the baseline, and it does not improve join-order robustness[7]. RPT outperforms the original PT in TPC-DS and DSB thanks to the `LargestRoot` algorithm. More importantly, RPT guarantees query robustness. Figure 8 shows selected queries from JOB and TPC-DS where the performance of the original PT is sensitive to different join orders. The root cause is that the transfer schedules produced by PT can lead to an incomplete reduction in the semi-join phase, as discussed in Section 3.1.

[6]The execution time of each query can be found in Appendix A
[7]The full robustness results for Bloom join and PT can be found in Appendix B

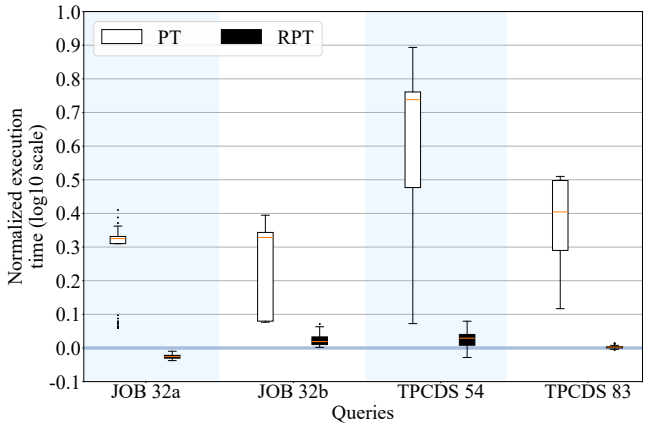


Figure 8: The distribution of the execution time of PT and RPT with random left deep plans for the selected query in JOB and TPC-DS – Normalized by the execution time of RPT with the optimizer's join order. The figure is log-scaled.

The above results are encouraging. They show that join order optimization might no longer be a critical challenge at least for acyclic queries (which are the majority) if we implement joins using RPT. In fact, the execution time of RPT using the optimizer's join order is within the horizontal lines (i.e., min to max excluding outliers) for *every* acyclic query in Figure 6. Future optimizers could, therefore, become much more efficient: they can better tolerate cardinality estimation errors, and they require simpler join enumeration algorithms because a left-deep plan is already good enough.

A few acyclic queries (13, 29, and 48) in TPC-DS have slightly larger variances than the others in Figure 6c. Query 13 and 48 include predicates that cannot be pushed down before the tables

are joined in DuckDB, e.g., ($R.a < 100$ AND $S.b < 200$) OR ($R.a > 500$ AND $S.b > 400$). It is preferable to join R and S earlier if the predicate is selective. Query 29 is acyclic but not γ-acyclic. According to the analysis in Section 3.2, certain join orders are unsafe. Although these special cases exhibit adequate robustness with random plans, they can still benefit from the optimizer.

5.1.2 Acyclic Queries (bushy). We show the distribution of the end-to-end execution time of random bushy plans in Figure 7 with the robustness factors summarized in Table 2. We omit results for individual queries of TPC-DS in Figure 7[8]. When including bushy plans, RPT exhibits similar robustness measures against random join orders as in the left-deep case, with the average RF < 1.8 and the max RF = 7.7 for Query 17e in JOB.

We notice a slight robustness degradation for a few queries (e.g., TPC-H Q7 and JOB 16b, 17e) when switching from left-deep to bushy plans. They share the common reason that the optimizer mistakenly placed the larger table on the build side of hash joins in the worst plans (out of random bushy plans). As shown in Figure 10, for example, picking the wrong build side for the top hash join alone in JOB 17e slows down the query by 37%. Such a mistake is unlikely in a left-deep plan because each base table (i.e., build side) is typically filtered heavily in the transfer phase of RPT while the size of the intermediate result (i.e., probe side) increases monotonically in the join phase.

To demonstrate the performance gain of considering bushy plans, we select the best (i.e., with minimum execution time) random left-deep plan and the best random bushy plan for each query and compare their performance in Figure 9. We also include the left-deep plan and the bushy plan produced by DuckDB's optimizer for each query (labeled as Optimizer's Left Deep and Optimizer's Bushy, respectively) as references in the figure. We observe that considering bushy plans in the join phase of RPT only speeds up the end-to-end execution by 6% and 11% for TPC-H and JOB, respectively compared to left-deep. Most optimizer's plans are slightly slower than the best ones from our randomly generated join orders, but the relative speedups of considering bushy plans remain small (10% for TPC-H and 5% for JOB). The semi-join reduction carried out in the transfer phase of RPT significantly reduces the benefit of exploring a larger plan enumeration space. Therefore, we conclude that it is unnecessary to explore bushy plans when applying Robust Predicate Transfer because bushy plans could sacrifice robustness for modest performance improvement.

5.1.3 Cyclic Queries. RPT does not provide robustness guarantees for cyclic queries, as shown by the red-labeled queries in Figure 6 and Figure 7 (i.e., TPC-H Q5, TPC-DS 19, 24, 46, 64, 68, 72, and 85). Although RPT improves the execution time in most cases, the performance gap between the best and worst plans for a cyclic query is still huge. We propose that a robust execution engine in the future should adopt a hybrid approach to handle joins: executing the cyclic part of the query using worst-case optimal joins while processing the rest with Robust Predicate Transfer.

5.1.4 Case Study. We present a case study on JOB 2a to better illustrate the robustness guarantees brought by RPT. Figure 11 shows the best and worst left-deep plans for the baseline and RPT (join phase only) with the size of each base or intermediate table marked. Without RPT, the worst join order produces 179× more intermediate tuples than the best. The worst join order suffers from the "diamond problem" described in [21]: small input → large intermediate result → small output, thus wasting computation. In comparison, the ratio of total intermediate results between the worst and best plans reduces to 1.2× with RPT. No matter what the join order is, the size of each intermediate table is bounded by the output size (i.e., 7.8k) and is monotonically increasing as the query executes.

We also notice that even the best plan from the baseline must process a much larger (≈5×) intermediate result than any RPT plan. This is because RPT has a strict complexity supremacy over the baseline. Figure 12 shows an example where query $R \bowtie S \bowtie T$ outputs nothing but any baseline plan (w/o RPT) must process $N^2/2$ tuples, a quadratic explosion compared to RPT plans. This example can be extended to create an exponential explosion as the number of tables increases. In comparison, Yannakakis algorithm guarantees that the size of $\sum$intermediate results for RPT can be at most $n\times$ the output size, where n is the number of joins.

5.2 Robustness of `LargestRoot`

We next zoom in to evaluate the robustness of the transfer phase in RPT (i.e., the `LargestRoot` algorithm). We modified `LargestRoot` to generate 50 random join trees, but each of them still has the largest relation as the root. Specifically, we replaced the original Line 3 in `LargestRoot` with "Find an edge $e = \{R, S\} \in E(G_q)$ such that $R \in \mathcal{R} \setminus \mathcal{R}', S \in \mathcal{R}'$". We fix the join order in each run to be the one produced by DuckDB's default optimizer. Other experiment settings follow those in Section 5.1.

Figure 13 shows the distribution of the end-to-end execution time with random `LargestRoot` transfer graphs for each query in TPC-H and JOB. The 50 execution times for each query are normalized by the query time achieved using the unmodified `LargestRoot`. We observe that the performance of the queries is robust against different transfer graphs (i.e., join trees for acyclic queries) as long as the algorithm keeps the largest relation at the root. Additionally, we notice that most boxes in Figure 13 are above 1.0 (i.e., slower than the original `LargestRoot`), indicating that the edge-picking heuristic used in Line 3 of `LargestRoot` is effective in speeding up the transfer phase of RPT.

5.3 Robustness with Multi-Threaded Execution

We repeat the left-deep experiments (i.e., Figure 6) in Section 5.1 with 32 threads to investigate how multi-threaded execution affects the robustness of RPT. As shown in Figure 14, RPT still exhibits outstanding query robustness with orders-of-magnitude improvement over the baseline on the Robustness Factor (RF). Compared to Figure 6, we notice that the variance of the execution times across different left-deep plans increases for some of the queries when switching from single-threaded to multi-threaded execution. This is because some random left-deep plans placed a relatively small (reduced) table on the probe side of the long (probing) pipeline, which does not have enough data chunks to distribute across 32 parallel threads to fully utilize the computation. The problem is orthogonal to the robustness guarantees offered by RPT.

[8]They can be found in Appendix C

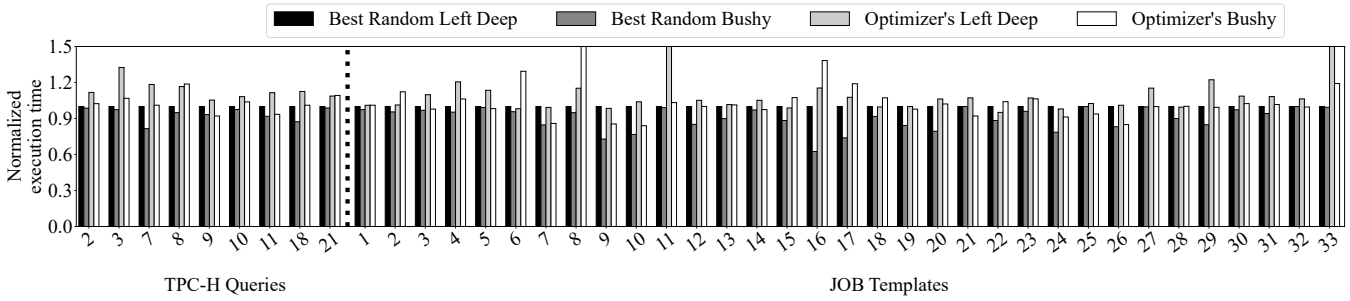


Figure 9: Speed up of bushy over left-deep plans. – We draw the minimum execution time of RPT out of random left-deep/bushy plans as well as the execution time of RPT with the optimizer's left-deep/bushy plan for each query in TPC-H and JOB.

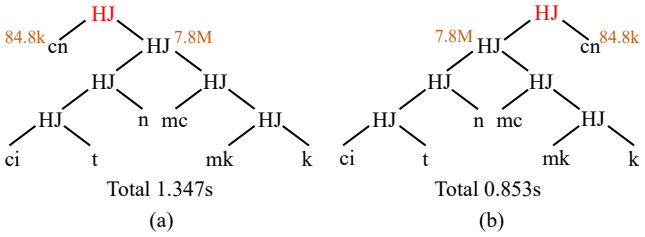


Figure 10: Slowdown caused by picking the wrong build side of a hash join (JOB 17e) – (a) is a random plan with the incorrect build side for the top HJ; (b) is the fixed plan with the build side and probe side flipped.

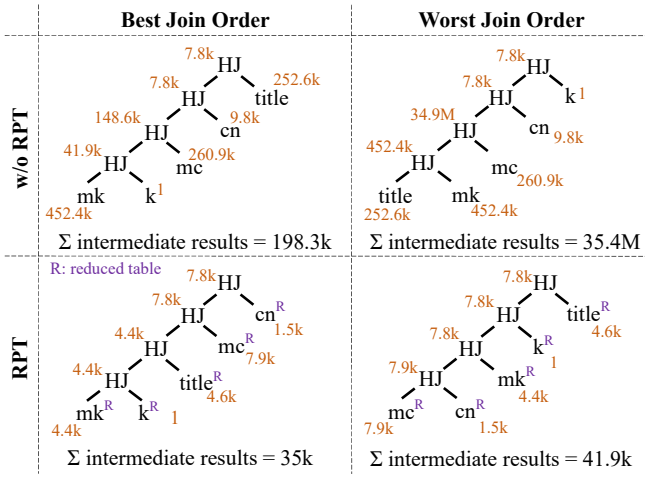


Figure 11: Case study on the robustness of JOB 2a – We consider the reduced tables in RPT (i.e., filtered base tables after the transfer phase) as intermediate results here.

5.4 Performance with Data On Disk

We extend our evaluation to the case where (1) the base tables reside on disk (labeled as "on-disk"), and (2) some intermediate results of RPT do not fit in memory (labeled as "+spill"). The intermediate results of RPT refer to the materialized data chunks that contain the remaining tuples after the forward pass in the semi-join phase. We evaluate the optimizer's plan for DuckDB and RPT for each query in TPC-H and JOB. For "+spill", we configure the available memory to be ≈ 50% of RPT's peak memory usage for each query and make sure that the spilled data reside on disk. As shown in Figure 15, RPT still archives an average (geometric mean) speedup of 1.3× and 1.5× over the default DuckDB for the "on-disk" and "on-disk+spill" cases, respectively. Although the backward pass in the semi-join phase of RPT incurs repeated data accesses, the overhead is small. This is because (1) the volume of the materialized data after the forward pass is small due to the selective semi-join filters, and (2) the backward-pass scans on the materialized data are sequential.

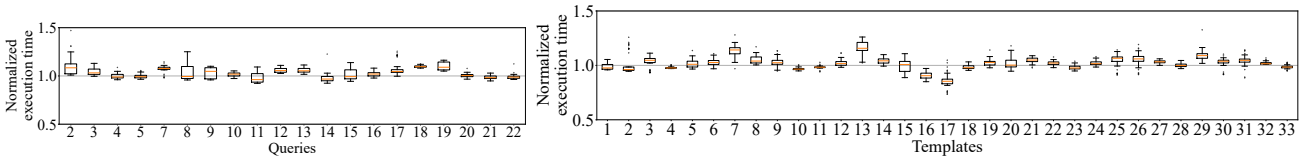
R			S			T	
A	**B**	⋯	**B**	**C**	⋯	**C**	⋯
1	1		1	1		2	
2	1		⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	1	1		2	
N/2	1		2	2		2	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
N	1		2	2		2	

Figure 12: An example query (with an empty output) where any w/o RPT plan must process $N^2/2$ tuples.

5.5 Performance of Bloom Filters

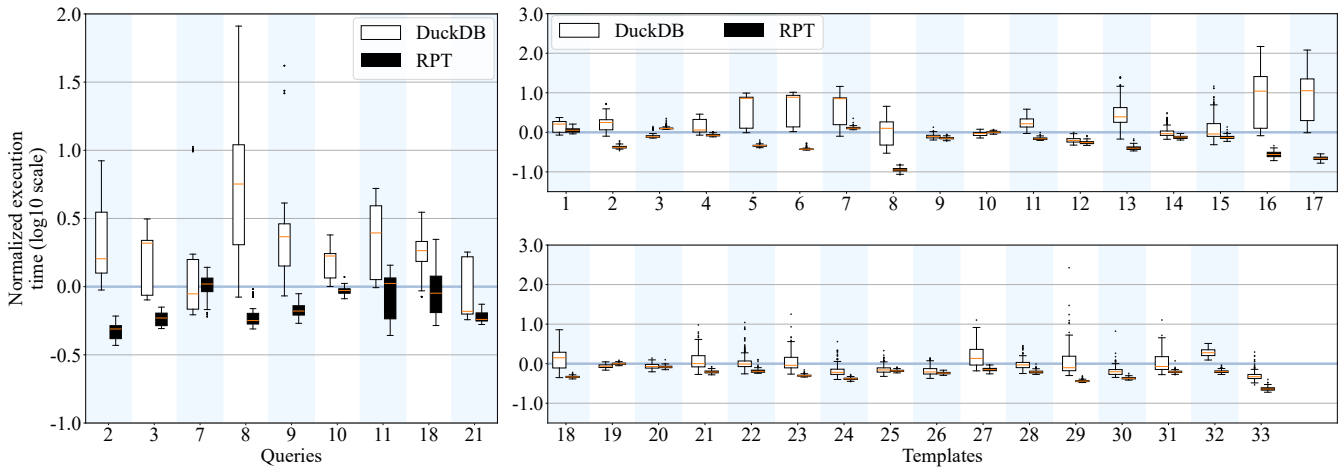
We presented in Figure 6 that RPT improves the overall query performance by ≈1.5× besides robustness, and our performance breakdown shows that the Bloom filter operations in the transfer phase of RPT account for on average 28%, 12%, and 46% of the total execution time in TPC-H, JOB, and TPC-DS, respectively. In this section, we evaluate the performance gap between Bloom filter probes and hash table probes through a microbenchmark. We create a synthetic dataset with two single-column tables. We fix the size of the probe-side table to 1 billion rows while varying the size of the build-side table. The integer values of each column are uniformly distributed between 0 and 2^{30}.

Figure 16 reports the execution time of performing 1 billion probes on hash tables or Bloom filters with different sizes. We use DuckDB's vectorized hash table implementation for hash probes and our modified version of Arrow's blocked Bloom filter for Bloom



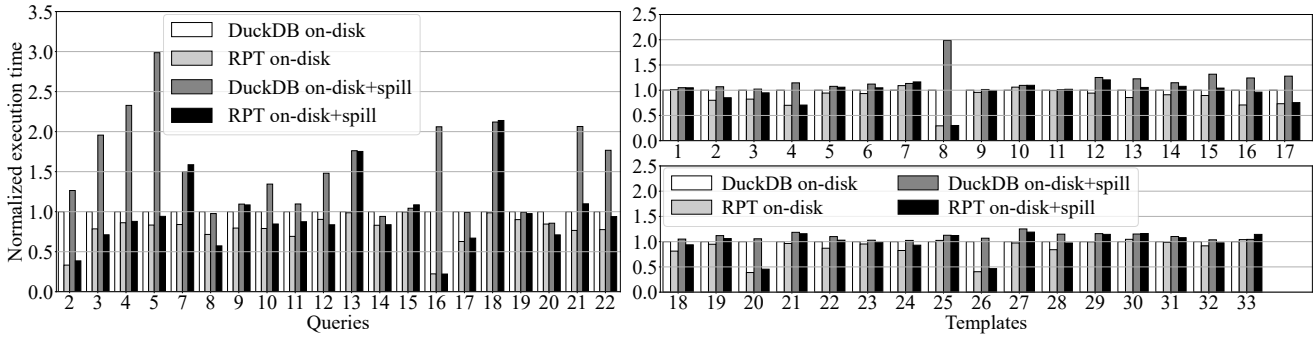
(a) TPC-H (b) JOB

Figure 13: Distribution of the execution time of 50 random `LargestRoot` transfer graphs for each query in TPC-H and JOB – The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers).



(a) TPC-H (32 threads) (b) JOB (32 threads)

Figure 14: Distribution of multi-threaded execution time of random left-deep plans for each acyclic query in TPC-H and JOB – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers).



(a) TPC-H (b) JOB

Figure 15: Comparison of the execution time of DuckDB and RPT (with optimizer's plan) for each query in TPC-H and JOB when the base tables reside on disk (on-disk) + the intermediate results do not fit in memory (+spill) – Normalized by the execution time of default DuckDB with base tables on disk.

probes. The blue (red) vertical lines denote the points where the size of the hash table (Bloom filter) exceeds L1, L2, and L3 caches. We observe that the SIMD version of Bloom probes outperforms vectorized hash probes by 2 – 7×. The performance gap grows as the size of the hash table / Bloom filters increases, indicating a potentially greater performance advantage of RPT on larger datasets.

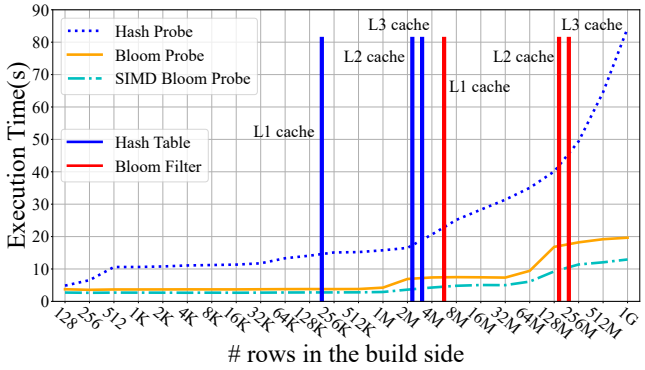


Figure 16: Microbenchmark on Bloom Probe vs. Hash Probe – We fix the probe side to 1 billion entries while varying the size of the build side on the x-axis.

6 Related Work

6.1 Sideways Information Passing (SIP)

Sideways Information Passing (SIP) refers to techniques that optimize join operations by transmitting predicate information to the target table to facilitate tuple pre-filtering in a database. Existing SIP techniques can be categorized as Bloom join [23, 48, 70, 89] and semi-join reduction [20]. In Bloom join, a Bloom filter is generated on the build side of a hash join and passed to the probe side to filter tuples before accessing the hash table. Semi-join reduction, on the other hand, applies a semi-join operation to pre-filter tuples before conducting the actual hash join.

Lookahead Information Passing (LIP) [89] can be considered a special case of Robust Predicate Transfer with star schema. LIP constructs Bloom filters for each dimension table and uses them to pre-filter the large fact table before performing the joins. LIP focuses on techniques to reorder the Bloom filters dynamically and adaptively to reduce the computational overhead of the SIP process. These techniques are orthogonal to our work and can also be applied to RPT.

In contrast to the existing SIP approaches, Robust Predicate Transfer provides strong theoretical guarantees on query robustness by applying pre-filtering (with Bloom filters) systematically based on the Yannakakis algorithm, rather than focusing on particular joins locally.

6.2 Robust Query Processing

Previous studies [35, 84] offer a comprehensive survey of robust query optimization methods. These methods target mitigating the impact of inaccurate cardinality estimations, and they can classified into two categories: robust plans [9, 13, 18, 25, 42, 43, 78] and re-optimization [17, 22, 28, 34, 44, 45, 57, 66, 87].

Robust plans, such as Least Expected Cost [18, 25], estimate the distributions of the filter/join selectivities. In contrast, the Cost-Greedy approach reduces the search space by low-cardinality approximations to favor the choices of performance-stable plans [42]. Similarly, SEER applies low-cardinality approximations to accommodate arbitrary estimation errors [43], while [9, 13, 78] propose metrics to quantify the robustness of execution plans during query optimization.

ReOpt [22, 45] introduces mid-query re-optimization, where the query engine detects cardinality estimation errors at execution time and re-invokes the optimizer to refine the remaining query plan. Eddies routes data tuples adaptively through a network of query operators during execution [17]. The POP algorithm introduces the concept of a "validity range" for selected plans, triggering re-optimization when the actual parameter values fall outside this range [34, 57]. Plan Bouquet eliminates the need for estimating operator selectivities by identifying a set of "switchable plans" that can accommodate runtime selectivity variations [28]. Experiments in [66] demonstrate that query re-optimization achieves excellent performance on PostgreSQL with the Join Order Benchmark. QuerySplit [87] introduces a novel re-optimization technique to minimize the probability of explosive intermediate results during re-optimization. POLAR [44] avoids intertwining query optimization and execution by inserting a multiplexer operator into the physical plan.

A few recent works [21, 39] developed algorithms fundamentally equivalent to the Yannakakis algorithm. They focused on avoiding performance regression when applying semi-join reductions even on worst-case input (i.e., input where pre-filtering is ineffective).

Compared to RPT, most existing robust query processing approaches lack theoretical guarantees on join-order robustness. Nevertheless, some of the techniques related to physical operator selections and operators beyond join are orthogonal to RPT and can complement our approach to boost query performance further.

6.3 Worst-Case Optimal Join

While the Yannakakis algorithm performs acyclic joins in optimal time (linear in the input and output size), answering general cyclic queries in polynomial time in terms of input, output, and query size is impossible unless P = NP.

A tractable extension for the cyclic case is near-acyclic queries, whose intricacy can be measured by different notions of width, such as treewidth [71], query width [24], hypertree width [32], and submodular width [58]. Generally speaking, a query with a width of k has an upper bound $O(N^k + OUT)$ on the time complexity. The hierarchy of bounds is summarized in a survey [74] and a recent result [11].

Worst-case optimal join (WCOJ) algorithms are developed to guarantee the above bounds on the running time. Binary joins are ubiquitous in relational DBMS but fail short on certain database instances compared to WCOJ algorithms. NPRR [62] is the first algorithm that achieves the AGM bound [16], and then an existing algorithm LFTJ is also proved to be running in the AGM bound [75]. These algorithms are unified as the Generic Join [63, 64], which determines one variable at a time using tries. The PANDA algorithm [10, 46] eliminates one inequality at a time using horizontal partitioning and achieves the polymatroid bound. Variants of WCOJ algorithms have been adopted in distributed query processing [14, 26, 49], graph processing [8, 14, 38, 59, 86, 88], and general-purpose query processing [7, 15, 31]. WCOJ algorithms are becoming practical as their performance surpasses traditional binary joins for certain queries [77].

Unlike WCOJ algorithms, Robust Predicate Transfer only provides theoretical guarantees on the runtime for acyclic queries. However, it is strictly better than WCOJ algorithms because it bounds the runtime to the instance-specific output size rather than a more generalized upper bound.

7 Conclusion

We proposed the Robust Predicate Transfer algorithm that is provably robust against arbitrary join orders of an acyclic query. Our evaluation in DuckDB shows that RPT ensures a small variation in the execution time between random join orders for acyclic queries while improving their end-to-end performance at the same time. We hope that our results advance the state-of-the-art of robust SQL analytics and will simplify the join optimization logic in future query optimizers.

References

[1] TPC-DS Benchmark. http://www.tpc.org/tpcds/, 1999.
[2] TPC-H Benchmark. http://www.tpc.org/tpch/, 1999.
[3] Join Order Benchmark. http://github.com/gregrahn/join-order-benchmark/, 2015.
[4] Apache Arrow. http://arrow.apache.org/, 2016.
[5] Postgresql. http://www.postgresql.org/, 2024.
[6] DuckDB. https://duckdb.org, 2024.
[7] Christopher Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. Levelheaded: A unified engine for business intelligence and linear algebra querying. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 449–460, 2018.
[8] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):1–44, 2017.
[9] M. Abhirama, Sourjya Bhaumik, Atreyee Dey, Harsh Shrimal, and Jayant R. Haritsa. On the stability of plan costs and the costs of plan stability. *Proc. VLDB Endow.*, 3(1):1137–1148, 2010. doi: 10.14778/1920841.1920983. URL http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/R101.pdf.
[10] Mahmoud Abo Khamis, Hung Q Ngo, and Dan Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 429–444, 2017.
[11] Mahmoud Abo Khamis, Vasileios Nakos, Dan Olteanu, and Dan Suciu. Join size bounds using lp-norms on degree sequences. *Proceedings of the ACM on Management of Data*, 2(2):1–24, 2024.
[12] Foto N. Afrati. Safe subjoins in acyclic joins, 2022. URL https://arxiv.org/abs/2208.09671.
[13] Khaled Hamed Alyoubi, Sven Helmer, and Peter T. Wood. Ordering selection operators under partial ignorance. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management, CIKM 2015*, pages 1521–1530. doi: 10.1145/2806416.2806446. URL https://doi.org/10.1145/2806416.2806446.
[14] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed evaluation of subgraph queries using worst-case optimal lowmemory dataflows. *Proceedings of the VLDB Endowment*, 11(6):691–704, 2018.
[15] Molham Aref, Balder Ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1371–1382, 2015.
[16] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.
[17] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, 2000.
[18] Brian Babcock and Surajit Chaudhuri. Towards a robust query optimizer: a principled and practical approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 119–130, 2005.
[19] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 107–118, 2005.
[20] Philip A Bernstein and Dah-Ming W Chiu. Using semi-joins to solve relational queries. *Journal of the ACM (JACM)*, 28(1):25–40, 1981.
[21] Altan Birler, Alfons Kemper, and Thomas Neumann. Robust join processing with diamond hardened joins. *Proceedings of the VLDB Endowment*, 17(11):3215–3228, 2024.
[22] Sophie Bonneau and Abdelkader Hameurlain. Hybrid simultaneous scheduling and mapping in sql multi-query parallelization. In *International Conference on Database and Expert Systems Applications*, pages 88–98, 1999.
[23] Kjell Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 323–333, 1984.
[24] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. In *Database Theory—ICDT'97: 6th International Conference Delphi, Greece, January 8–10, 1997 Proceedings 6*, pages 56–70. Springer, 1997.
[25] Francis Chu, Joseph Halpern, and Johannes Gehrke. Least expected cost query optimization: what can we expect? In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 293–302, 2002.
[26] Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 63–78, 2015.
[27] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. DSB: A decision support benchmark for workload-driven and traditional database systems. *Proc. VLDB Endow.*, 14(13):3376–3388, 2021. doi: 10.14778/3484224.3484234. URL http://www.vldb.org/pvldb/vol14/p3376-ding.pdf.
[28] Anshuman Dutt and Jayant R Haritsa. Plan bouquets: A fragrant approach to robust query processing. *ACM Transactions on Database Systems (TODS)*, 41(2):1–37, 2016.
[29] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment*, 12(9):1044–1057, 2019.
[30] Ronald Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM (JACM)*, 30(3):514–550, 1983.
[31] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment*, 13(12):1891–1904, 2020.
[32] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 21–32, 1999.
[33] Max Halford, Philippe Saint-Pierre, and Franck Morvan. An approach based on bayesian networks for query selectivity estimation. In *Database Systems for Advanced Applications: 24th International Conference, DASFAA 2019, Chiang Mai, Thailand, Proceedings, Part II 24*, pages 3–19, 2019.
[34] Wook-Shin Han, Jack Ng, Volker Markl, Holger Kache, and Mokhtar Kandil. Progressive optimization in a shared-nothing parallel database. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 809–820, 2007.
[35] Jayant R Haritsa. Robust query processing: Mission possible. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 2072–2075, 2019.
[36] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. Deep learning models for selectivity estimation of multi-attribute queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1035–1050, 2020.
[37] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. Deepdb: learn from data, not from queries! *Proceedings of the VLDB Endowment*, 13(7):992–1005, 2020.
[38] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. A worst-case optimal join algorithm for sparql. In *The Semantic Web–ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part I 18*, pages 258–275, 2019.
[39] Zeyuan Hu, Yisu Remy Wang, and Daniel P. Miranker. Treetracker join: Turning the tide when a tuple fails to join, 2024. URL https://arxiv.org/abs/2403.01631.
[40] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems (TODS)*, 9(3):482–502, 1984.
[41] Yannis E Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of data*, pages 268–277, 1991.
[42] Harish D Pooja N Darera Jayant and R Haritsa. On the production of anorexic plan diagrams. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 1081–1092, 2007.
[43] Harish D Pooja N Darera Jayant and R Haritsa. Identifying robust plans through plan diagram reduction. In *Proceedings of the 34th International Conference on Very Large Data Bases*, pages 1124–1140, 2008.
[44] David Justen, Daniel Ritter, Campbell Fraser, Andrew Lamb, Allison Lee, Thomas Bodner, Mhd Yamen Haddad, Steffen Zeuch, Volker Markl, and Matthias Boehm. Polar: Adaptive and non-invasive join order selection via plans of least resistance. *Proceedings of the VLDB Endowment*, 17(6):1350–1363, 2024.
[45] Navin Kabra and David J DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 106–117, 1998.

[46] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. Panda: Query evaluation in submodular width, 2024. URL https://arxiv.org/abs/2402.02001.

[47] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. URL http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf.

[48] Paraschos Koutris. Bloom filters in distributed query execution. *University of Washington, CSE*, 544, 2011.

[49] Paraschos Koutris, Paul Beame, and Dan Suciu. Worst-case optimal algorithms for parallel query processing. In *19th International Conference on Database Theory (ICDT 2016)*, 2016.

[50] Hai Lan, Zhifeng Bao, and Yuwei Peng. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering*, 6:86–101, 2021.

[51] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. Is your learned query optimizer behaving as you expect? a machine learning perspective. *Proceedings of the VLDB Endowment*, 17(7):1565–1577, 2024.

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

[53] Daniel Lemire. Really fast bitset decoding for "average" densities. https://lemire.me/blog/2019/05/03/really-fast-bitset-decoding-for-average-densities/, 2019.

[54] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. Fauce: fast and accurate deep ensembles with uncertainty for cardinality estimation. *Proceedings of the VLDB Endowment*, 14(11):1950–1963, 2021.

[55] Guy Lohman. Is query optimization a "solved" problem. In *Proc. Workshop on Database Query Optimization*, volume 13, page 10, 2014.

[56] David Maier. *The Theory of Relational Databases*. 1983.

[57] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 659–670, 2004.

[58] Dániel Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. In *STOC*, pages 735–744. ACM, 2010.

[59] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment*, 12(11): 1692–1704, 2019.

[60] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 930–941, 2006.

[61] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 539–552, 2008.

[62] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, page 37–48, New York, NY, USA, 2012.

[63] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014.

[64] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

[65] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. Quicksel: Quick selectivity learning with mixture models. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1017–1033, 2020.

[66] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. How i learned to stop worrying and love re-optimization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1758–1761, 2019.

[67] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-and space-efficient bloom filters. In *Experimental Algorithms: 6th International Workshop*, pages 108–121, 2007.

[68] Mark Raasveldt. Duckdb: In-process analytical database system. https://15721.courses.cs.cmu.edu/spring2023/slides/22-duckdb.pdf, 2023.

[69] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, 2019.

[70] Sukriti Ramesh, Odysseas Papapetrou, and Wolf Siberski. Optimizing distributed joins with bloom filters. In *Distributed Computing and Internet Technology: 5th International Conference, ICDCIT 2008 New Delhi, India, December 10-12, 2008. Proceedings 5*, pages 145–156, 2009.

[71] Neil Robertson and P.D Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.

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

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

[74] Dan Suciu. Applications of information inequalities to database theory problems. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–30, 2023.

[75] Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *17th International Conference on Database Theory (ICDT 2014)*, 2014.

[76] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. Are we ready for learned cardinality estimation? *Proceedings of the VLDB Endowment*, 14(9), 2021.

[77] Yisu Remy Wang, Max Willsey, and Dan Suciu. Free join: Unifying worst-case optimal and traditional joins. *Proceedings of the ACM on Management of Data*, 1 (2), 2023.

[78] Florian Wolf, Michael Brendle, Norman May, Paul R. Willems, Kai-Uwe Sattler, and Michael Grossniklaus. Robustness metrics for relational query execution plans. *Proc. VLDB Endow.*, 11(11):1360–1372, 2018. doi: 10.14778/3236187.3236191. URL http://www.vldb.org/pvldb/vol11/p1360-wolf.pdf.

[79] Peizhi Wu and Gao Cong. A unified deep model of learning from both data and queries for cardinality estimation. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2009–2022, 2021.

[80] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. Predicate transfer: Efficient pre-filtering on multi-join queries. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org, 2024. URL https://www.cidrdb.org/cidr2024/papers/p22-yang.pdf.

[81] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep unsupervised cardinality estimation. *Proceedings of the VLDB Endowment*, 13(3), 2019.

[82] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: one cardinality estimator for all tables. *Proceedings of the VLDB Endowment*, 14(1), 2021.

[83] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the 7th International Conference on Very Large Data Bases*, volume 81, pages 82–94, 1981.

[84] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. Robust query optimization methods with respect to estimation errors: A survey. *ACM SIGMOD Record*, 44(3):25–36, 2015.

[85] C.T. Yu and M.Z. Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *Computer Software and The IEEE Computer Society's Third International Applications Conference*, pages 306–312, 1979.

[86] Wangda Zhang, Reynold Cheng, and Ben Kao. Evaluating multi-way joins over discounted hitting time. In *2014 IEEE 30th International Conference on Data Engineering (ICDE)*, pages 724–735. IEEE, 2014.

[87] Junyi Zhao, Huanchen Zhang, and Yihan Gao. Efficient query re-optimization with judicious subquery selections. *Proceedings of the ACM on Management of Data*, 1(2):1–26, 2023.

[88] Guanghui Zhu, Xiaoqi Wu, Liangliang Yin, Haogang Wang, Rong Gu, Chunfeng Yuan, and Yihua Huang. Hymj: A hybrid structure-aware approach to distributed multi-way join query. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1726–1729. IEEE, 2019.

[89] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M Patel. Looking ahead makes query plans robust: Making the initial case with in-memory star schema data warehouse workloads. *Proceedings of the VLDB Endowment*, 10(8):889–900, 2017.

[90] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. Flat: fast, lightweight and accurate method for cardinality estimation. *Proceedings of the VLDB Endowment*, 14(9), 2021.

A RPT Performance with Optimizer's Plan

In Appendix A, we present the full performance results of Robust Predicate Transfer using the optimizer's plan, compared to our baseline methods: DuckDB, Bloom Join, and Predicate Transfer.

Figure 17 shows the execution time with the optimizer's plan for each query in TPC-H. Note that we exclude Q1 and Q6, as they only involve scanning and filtering a single table. On average (geometric mean), Robust Predicate Transfer outperforms vanilla DuckDB by 1.53× and Bloom Join by 1.33×. Additionally, Robust Predicate Transfer achieves the same performance as Predicate Transfer. This is because TPC-H queries are relatively simple, and the transfer scheduling of Predicate Transfer and Robust Predicate Transfer does not differ significantly.

Figure 18 displays the execution time with the optimizer's plan for one result from each of the 33 query templates in the JOB. On average (geometric mean), Robust Predicate Transfer outperforms vanilla DuckDB by 1.46× and Bloom Join by 1.29×, while matching the performance of Predicate Transfer.

Figure 19 presents the execution time with the optimizer's plan for each query in TPC-DS. On average (geometric mean), Robust Predicate Transfer outperforms DuckDB by 1.56×, Bloom Join by 1.48×, and Predicate Transfer by 1.23×. However, for certain queries (e.g., Q16, Q61, and Q69), Robust Predicate Transfer performs poorly compared to vanilla DuckDB and Bloom Join. This is due to the result being empty for these queries, causing Robust Predicate Transfer to scan more tables than vanilla DuckDB and Bloom Join, as query execution stops upon encountering empty intermediate results.

In Figure 20, we show the execution time with the optimizer's plan for each query in DSB. On average (geometric mean), Robust Predicate Transfer outperforms vanilla DuckDB by 1.54×, Bloom Join by 1.45×, and Predicate Transfer by 1.23×. Similar to TPC-DS, for some specific queries, Robust Predicate Transfer exhibits poor performance for certain queries due to empty intermediate results, resulting in additional table scans compared to vanilla DuckDB and Bloom Join.

B Additional Robustness Results (Left Deep)

In Appendix B, we present the distribution of execution times with random left-deep plans for each query of Robust Predicate Transfer, compared to our baseline methods: DuckDB, Bloom Join, and Predicate Transfer. These results are shown in Figure 21a (TPC-H), Figure 22 (JOB), Figure 23 (TPC-DS query 1-52), Figure 24 (TPC-DS query 53-99), Figure 25 (DSB query 1-52) and Figure 26 (DSB query 53-99).

For most acyclic queries, both Robust Predicate Transfer and Predicate Transfer outperform vanilla DuckDB and Bloom Join in terms of robustness. However, for specific acyclic queries (e.g., JOB 32a and 32b, TPC-DS 54 and 83, DSB 54 and 83), Predicate Transfer is also not robust. This is because the `Small2Large` transfer algorithm used by Predicate Transfer lacks a theoretical guarantee.

Even for cyclic queries, Robust Predicate Transfer can improve robustness to some extent. However, due to the absence of the theoretical guarantee for cyclic queries, Robust Predicate Transfer fails to constrain their maximum execution time.

C Additional Robustness Results (Bushy)

In Appendix C, we present the distribution of execution time with random bushy plans for each query of Robust Predicate Transfer, compared to our baseline methods: vanilla DuckDB, Bloom Join, and Predicate Transfer. These results are shown in Figure 21b (TPC-H) and Figure 27 (JOB), Figure 28 (TPC-DS query 1-52), Figure 29 (TPC-DS query 53-99), Figure 30 (DSB query 1-52) and Figure 31 (DSB query 53-99).

The conclusions are consistent with the left-deep results, but we observe a deterioration in robustness. As discussed in the paper, this can be attributed to incorrect probe-build side selection during the hash join.

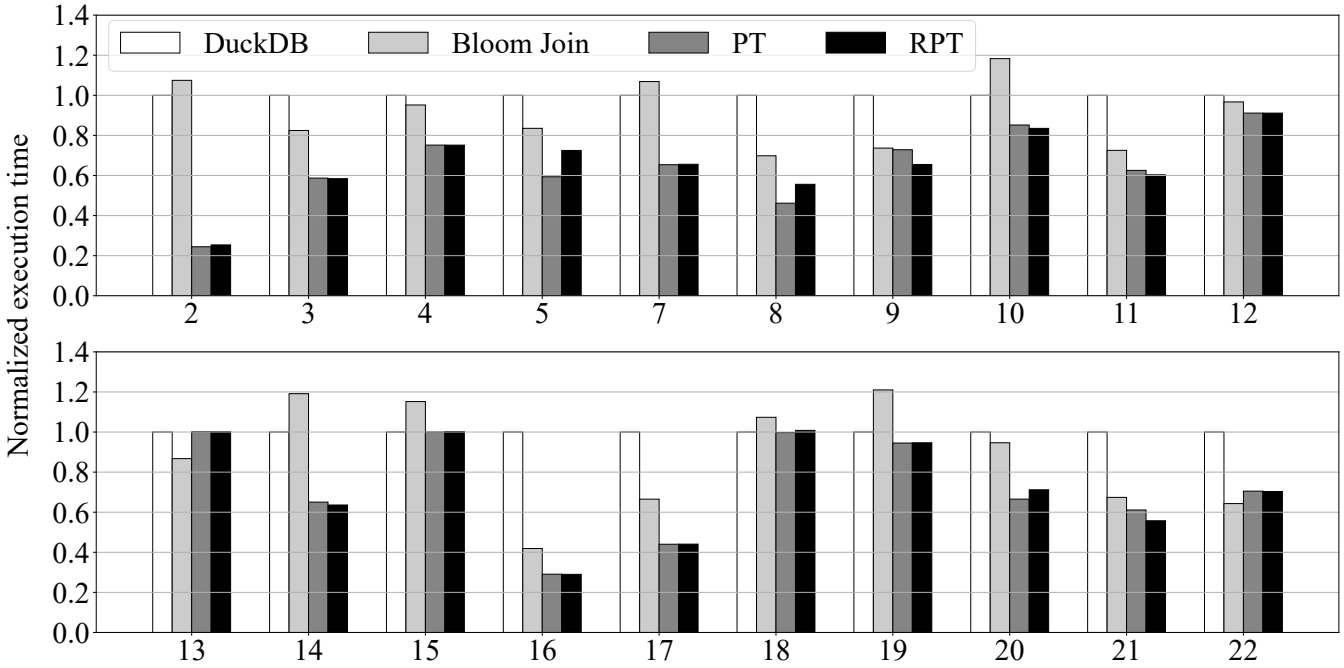


Figure 17: The execution time with optimizer's plans for each query in TPC-H – Normalized by the execution time of default DuckDB. We omit Q1 and Q6 as they are only the table scan and filtering.

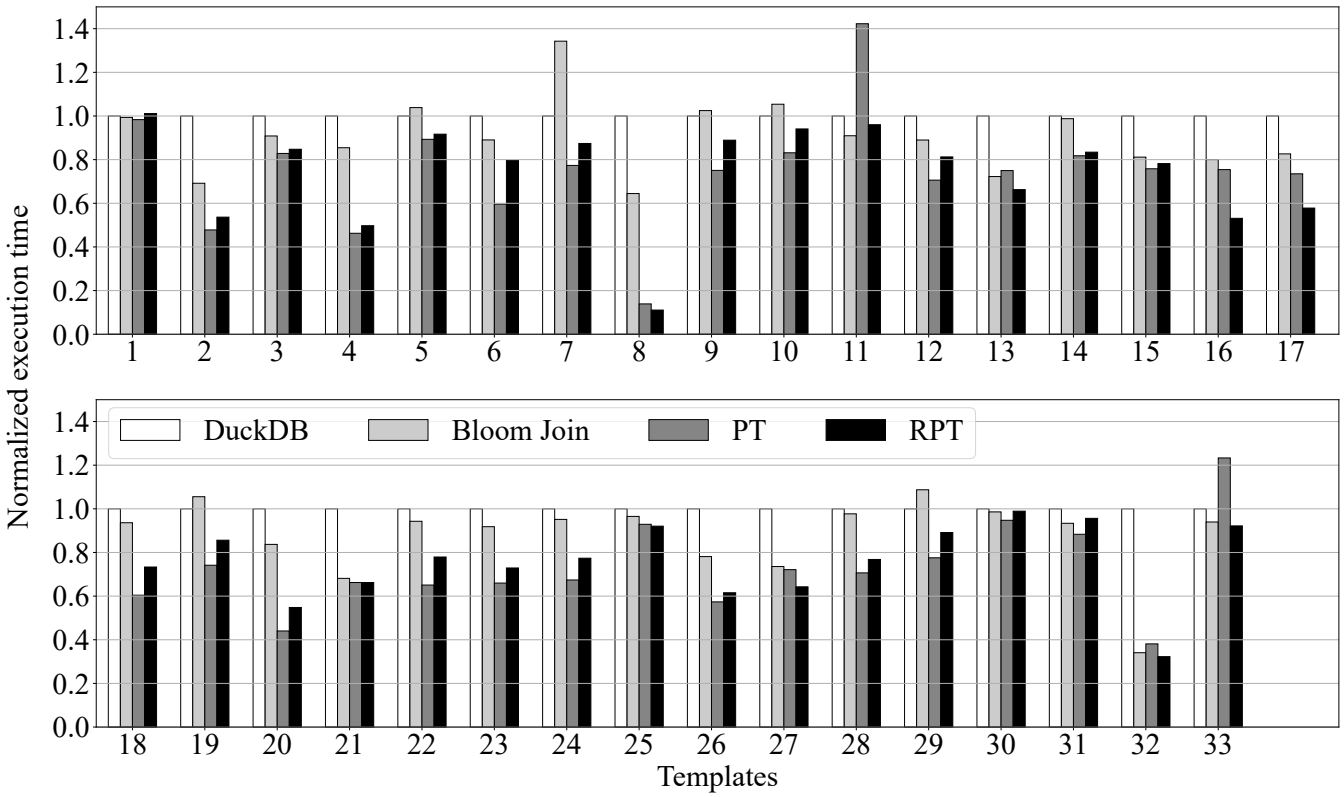


Figure 18: The execution time with optimizer's plans for each query in JOB – Normalized by the execution time of default DuckDB.

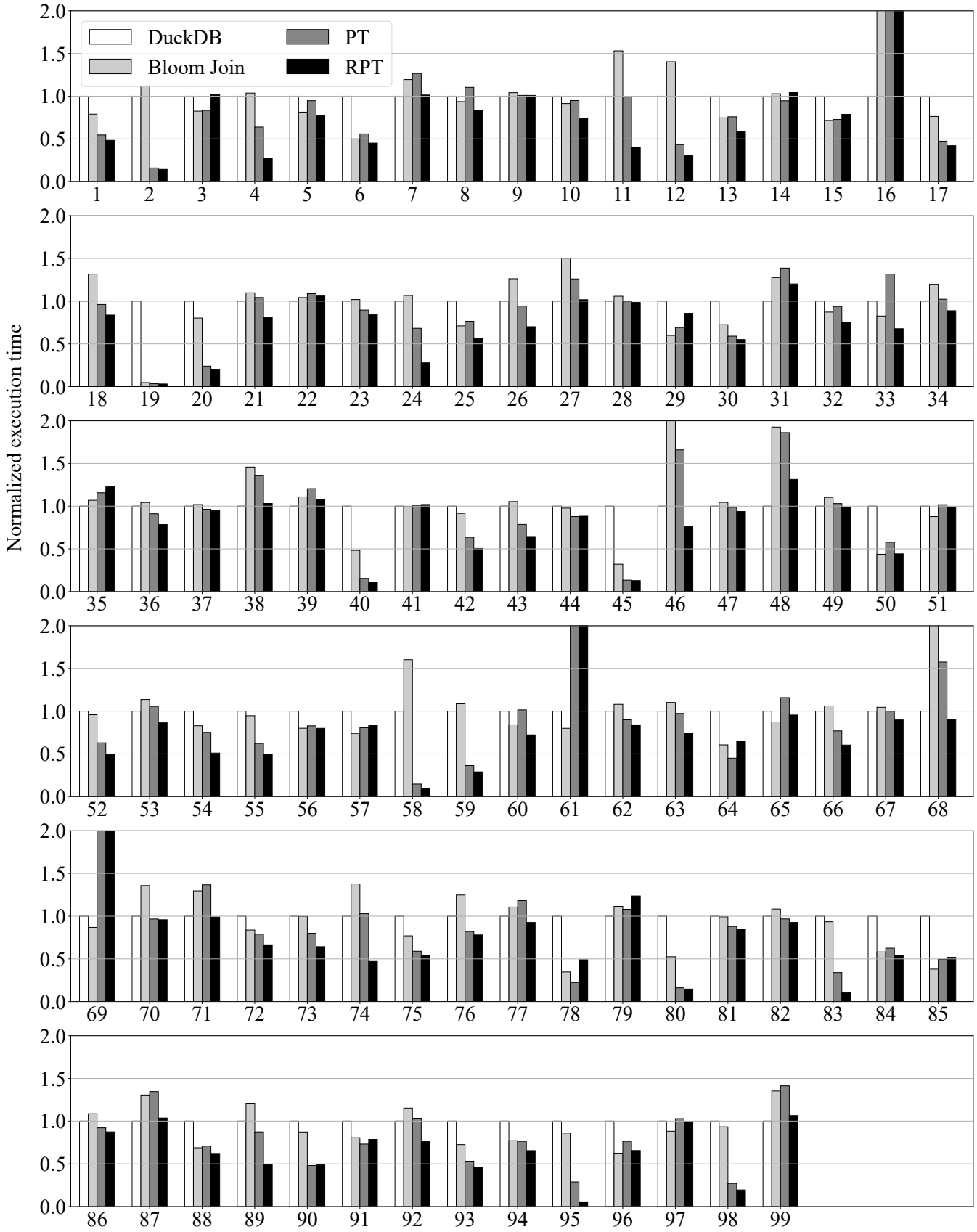


Figure 19: The execution time with optimizer's plans for each query in TPC-DS – Normalized by the execution time of default DuckDB.

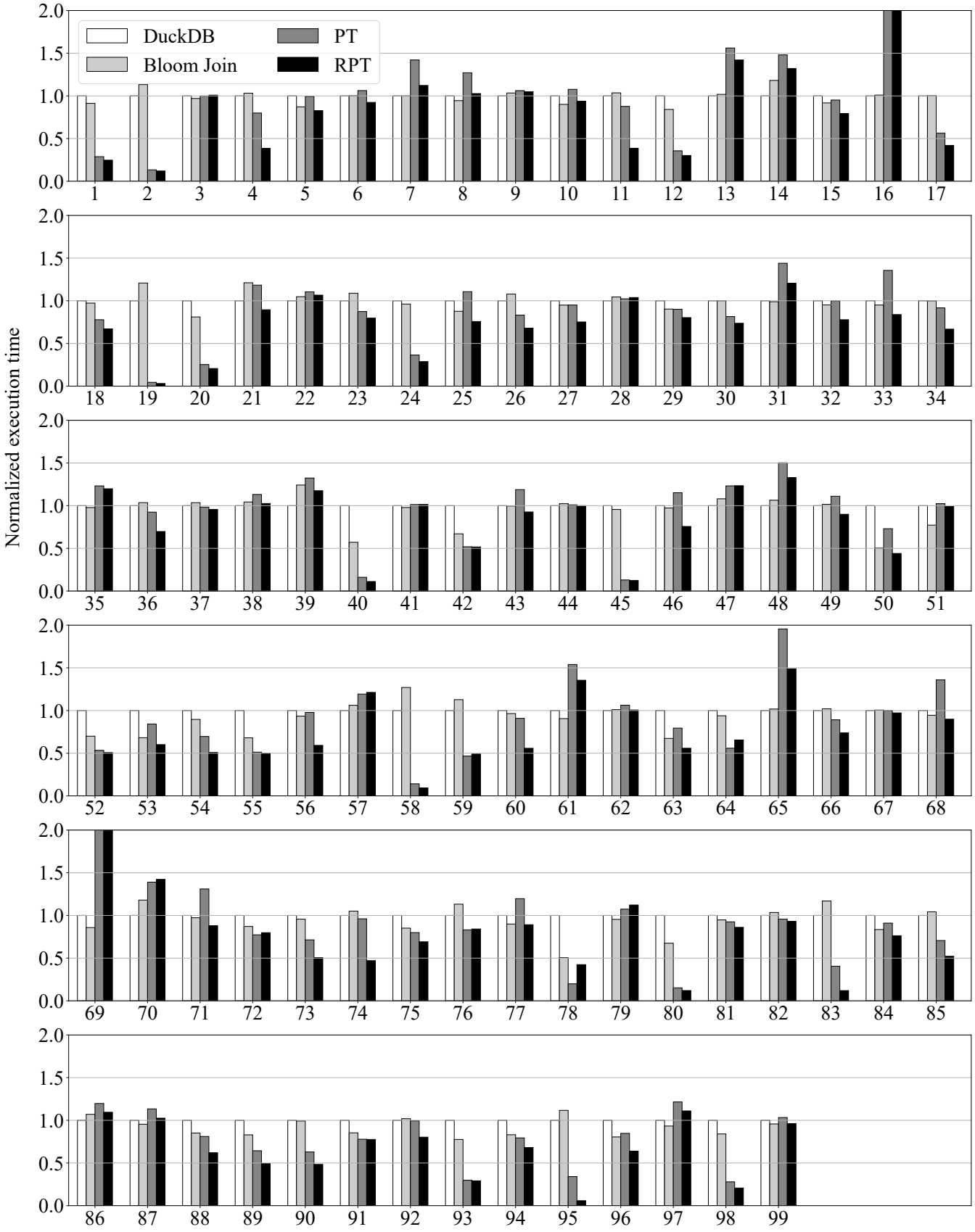
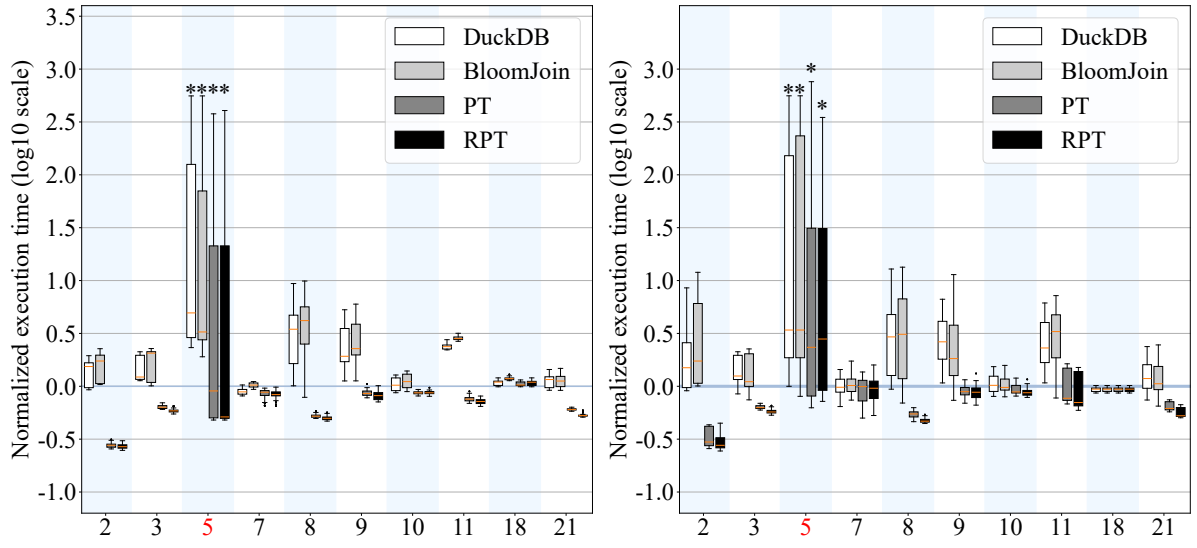


Figure 20: The execution time with optimizer's plans for each query in DSB – Normalized by the execution time of default DuckDB.



(a) Only left deep (b) Bushy

Figure 21: The distribution of execution time with random left deep plans for each query in TPC-H – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers). '*' indicates timeouts. Cyclic queries are in red.

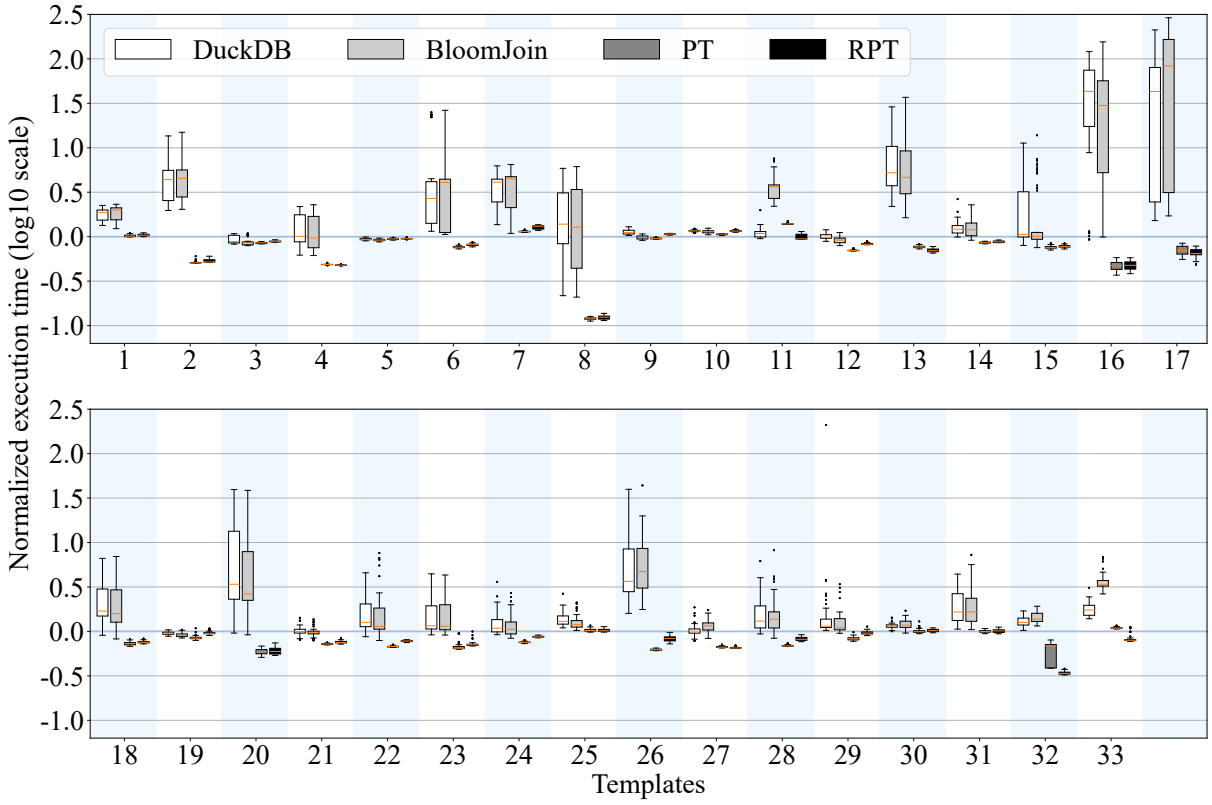


Figure 22: The distribution of execution time with random left deep plans for each template in JOB – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers).

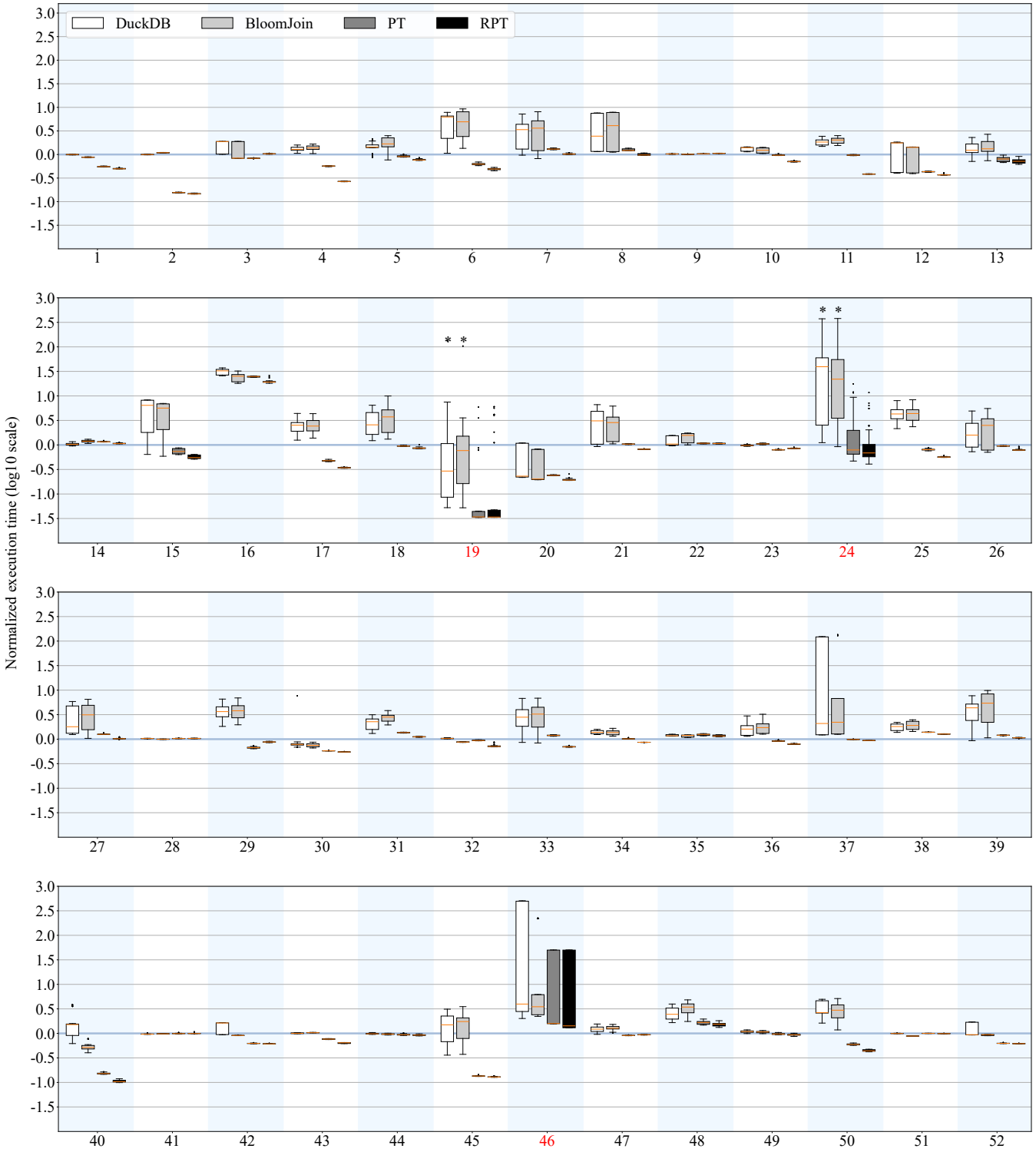


Figure 23: The distribution of execution time with random left deep plans for each query (1 - 52) in TPC-DS – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers). '*' indicates timeouts. Cyclic queries are in red.

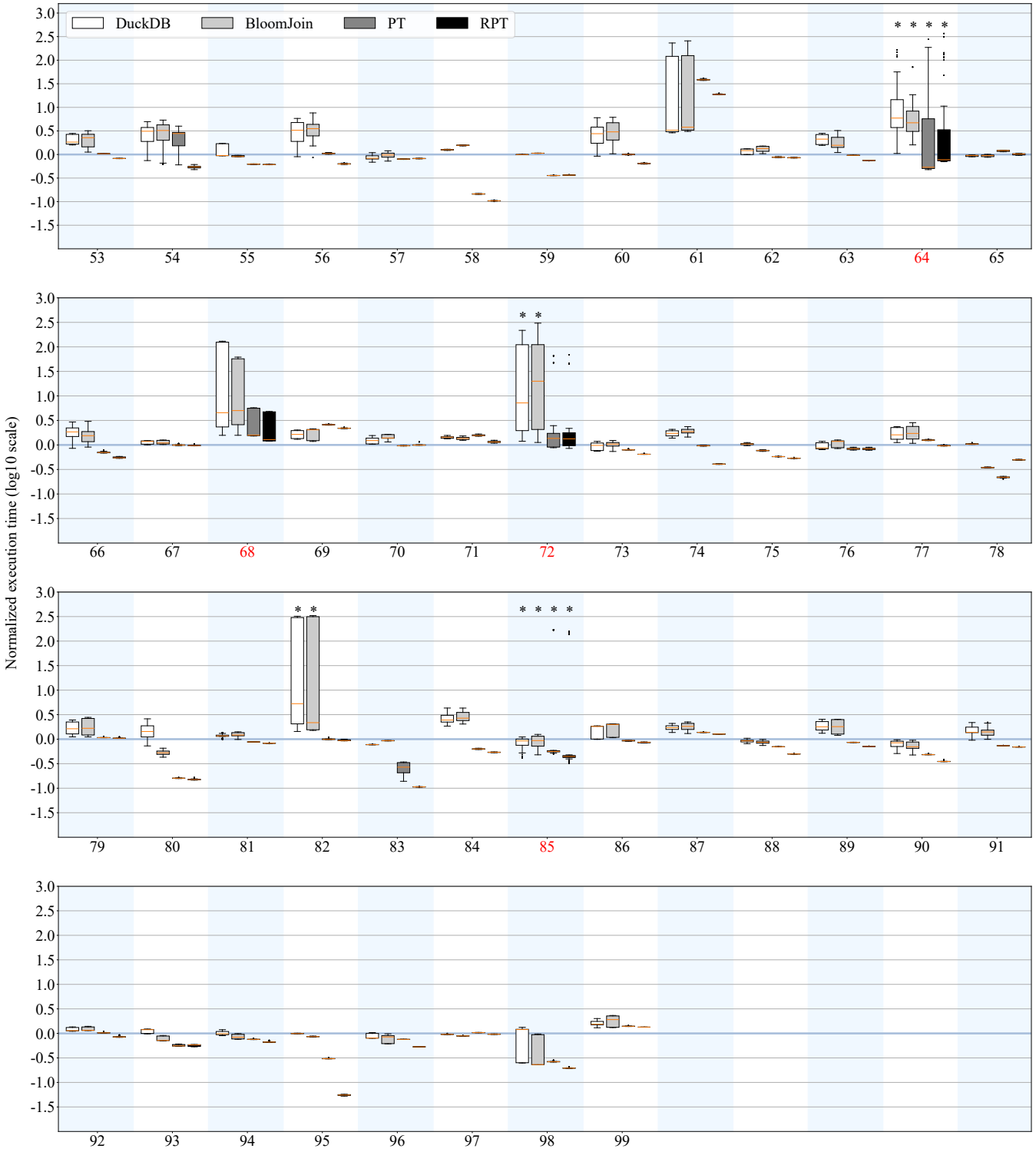


Figure 24: The distribution of execution time with random left deep plans for each query (53 - 99) in TPC-DS – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers). '*' indicates timeouts. Cyclic queries are in red.

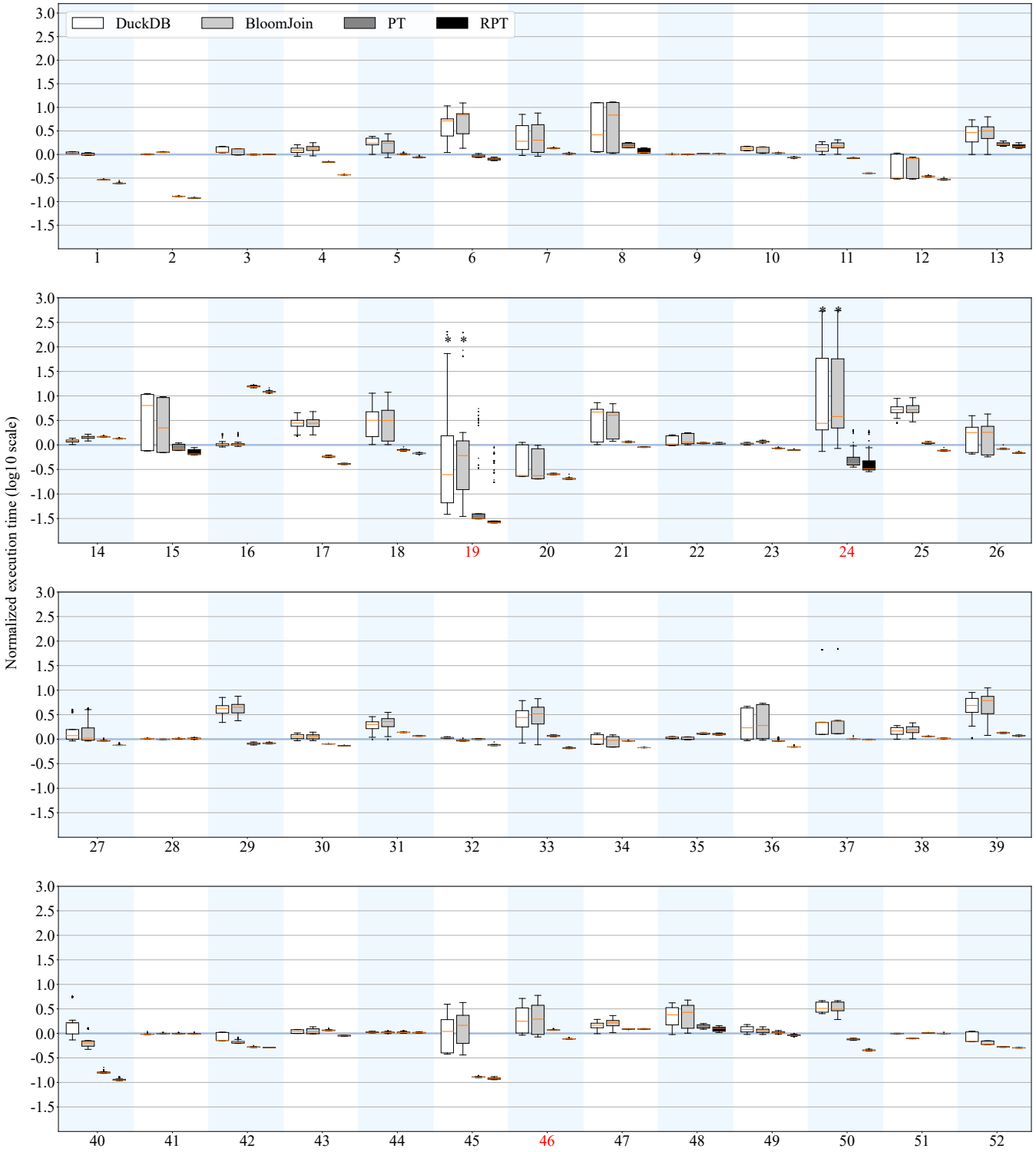


Figure 25: The distribution of execution time with random left deep plans for each query (1 - 52) in DSB – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers). '*' indicates timeouts. Cyclic queries are in red.

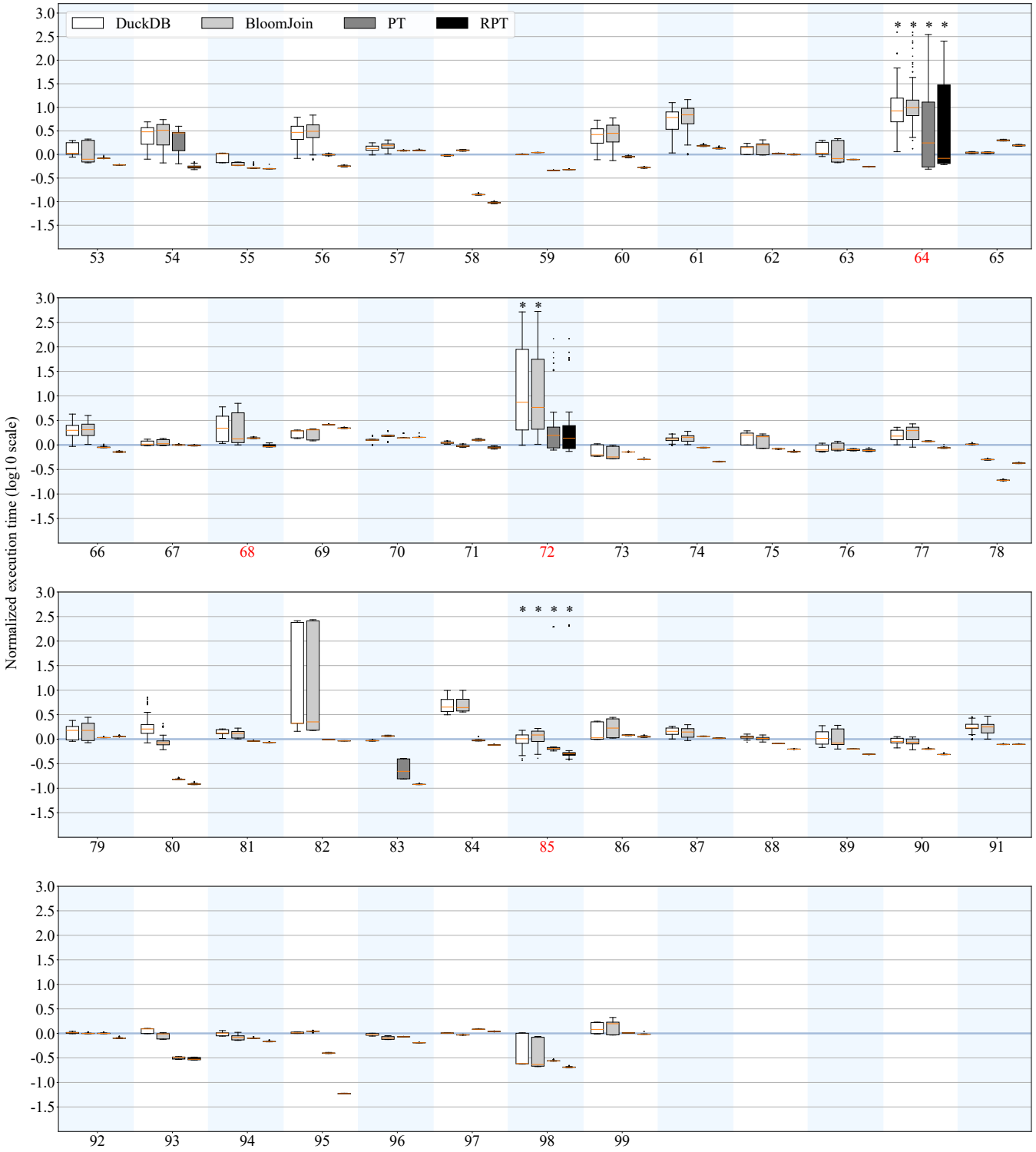


Figure 26: The distribution of execution time with random left deep plans for each query (53 - 99) in DSB – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers). '*' indicates timeouts. Cyclic queries are in red.

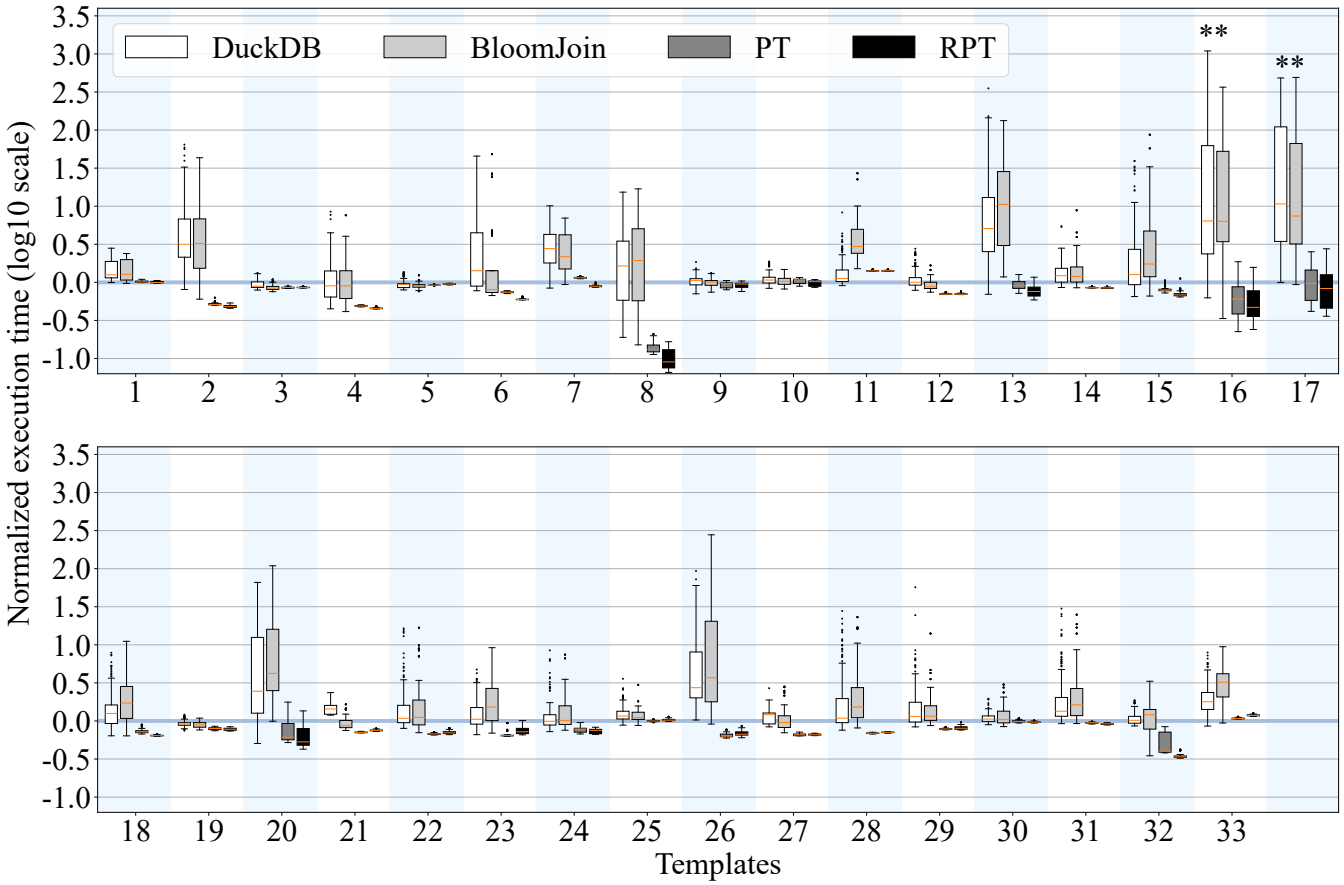


Figure 27: The distribution of execution time with random bushy plans for each query in JOB – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers).

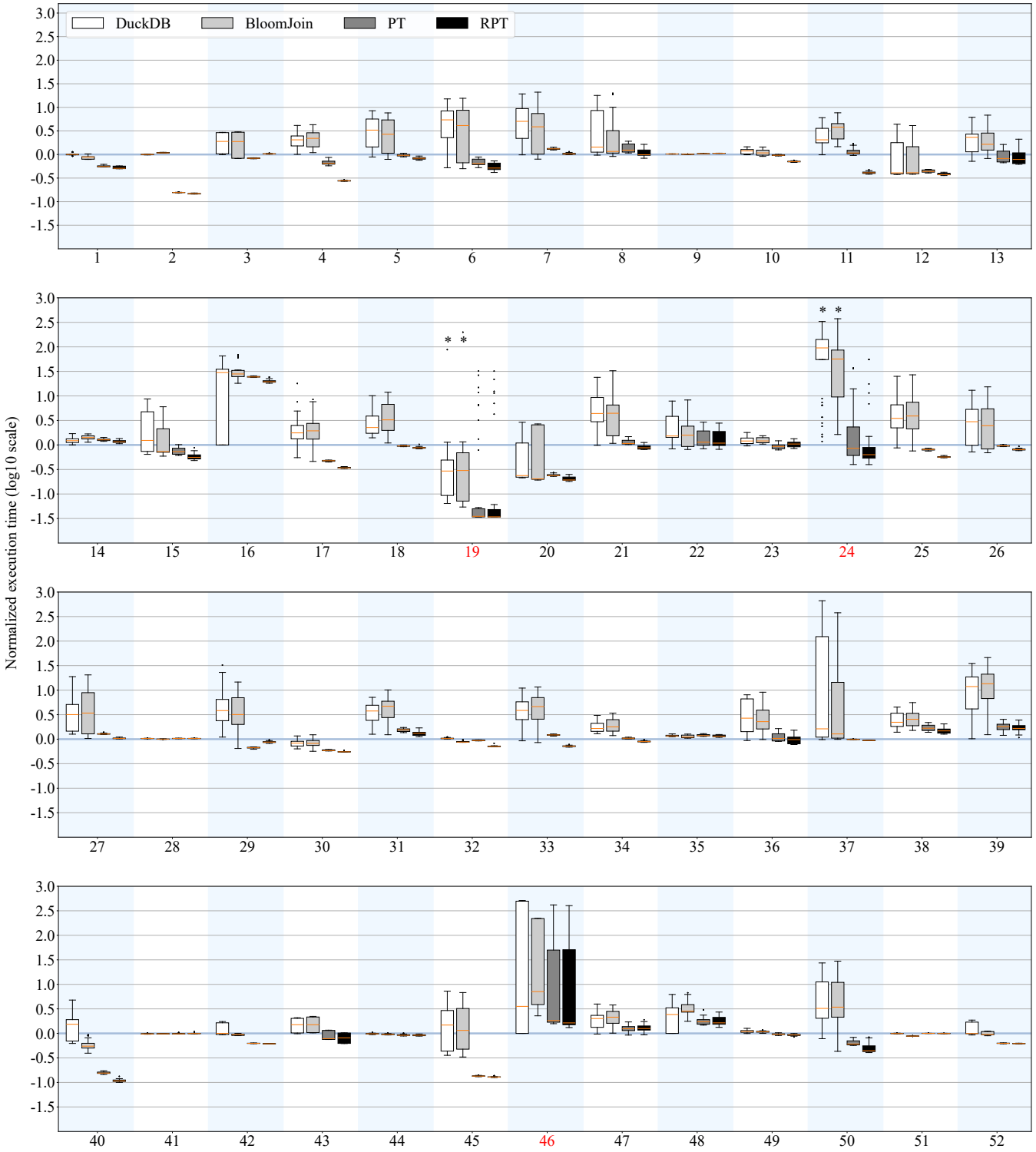


Figure 28: The distribution of execution time with random bushy plans for each query (1-52) in TPC-DS – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers). '*' indicates timeouts. Cyclic queries are in red.

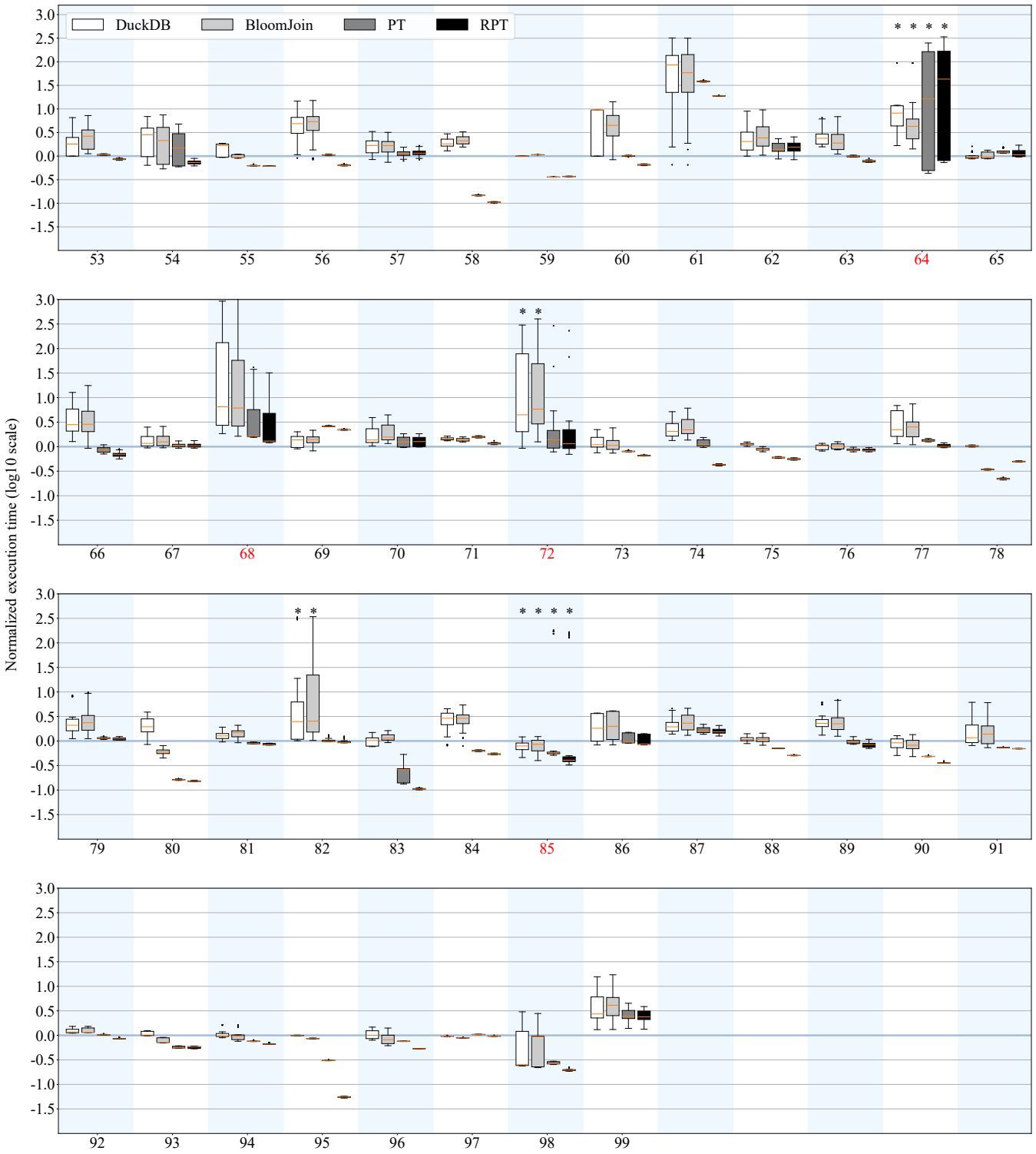


Figure 29: The distribution of execution time with random bushy plans for each query (53-99) in TPC-DS – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers). '*' indicates timeouts. Cyclic queries are in red.

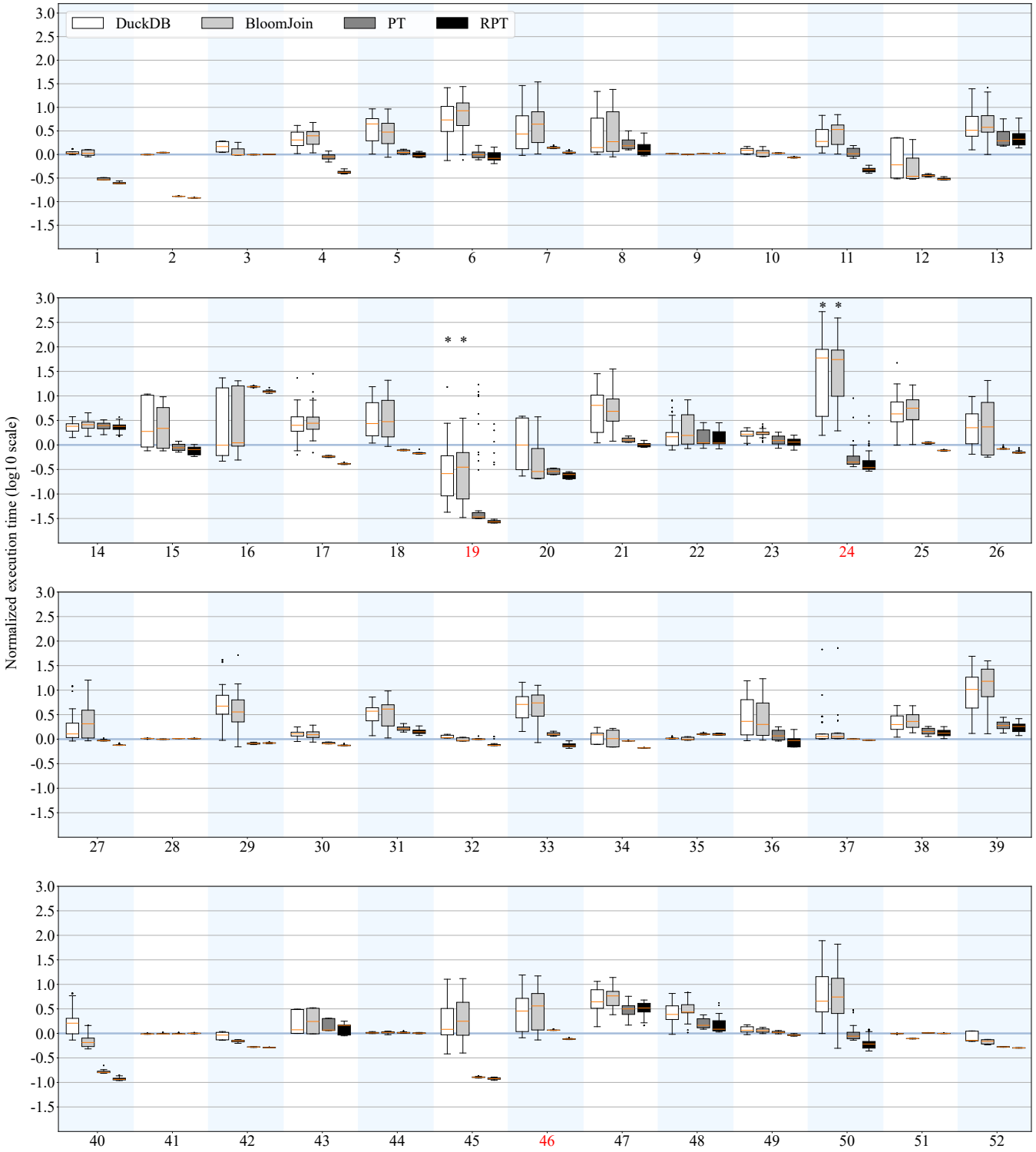


Figure 30: The distribution of execution time with random bushy plans for each query (1 - 52) in DSB – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers). '*' indicates timeouts. Cyclic queries are in red.

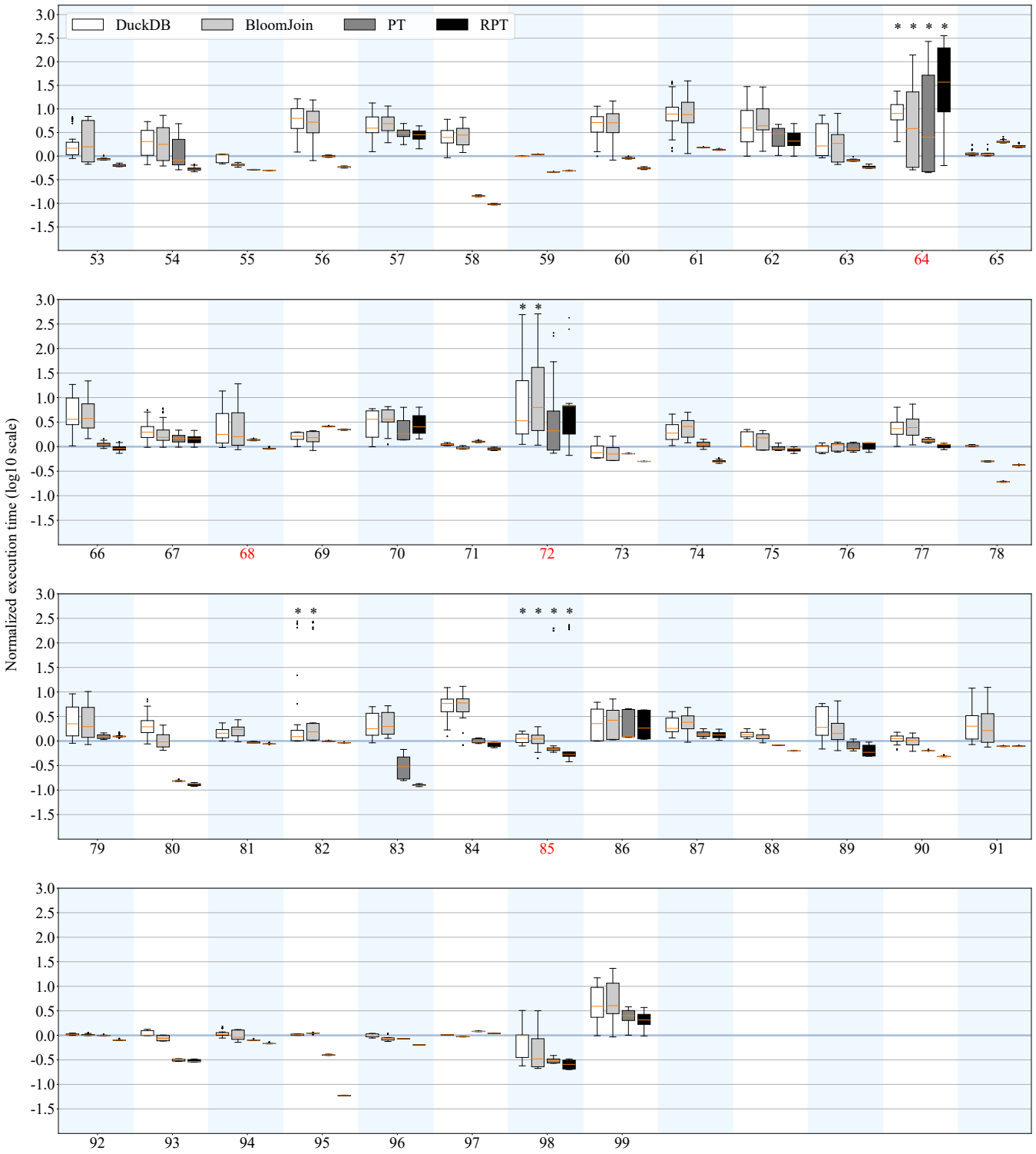


Figure 31: The distribution of execution time with random bushy plans for each query (53 - 99) in DSB – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers). '*' indicates timeouts. Cyclic queries are in red.