Robust Predicate Transfer with Dynamic Execution

Yiming Qiao
Tsinghua University
qiaoym21@mails.tsinghua.edu.cn

Peter Boncz
CWI
boncz@cwi.nl

Huanchen Zhang
Tsinghua University
huanchen@tsinghua.edu.cn

ABSTRACT

Efficient join query execution remains a key challenge in modern database systems. Although a recent method, Robust Predicate Transfer (RPT), improves robustness against suboptimal join orders, it introduces significant overhead from redundant filter creation and inefficient data scanning. We present RPT+ that addresses these issues through three key improvements. First, we propose asymmetric transfer plans to reduce redundant Bloom filter constructions. Second, we design cascade filters to improve data scanning efficiency by enabling both block-level skipping and tuple-level filtering. Third, we introduce dynamic pipelines to allow runtime filter creation and transfer plan adjustment. We implemented RPT+ in DuckDB (v1.3.0) and evaluated it across multiple benchmarks, including the Join Order Benchmark (JOB), SQLStorm, TPC-H, and Appian. Compared to the baseline DuckDB, RPT+ achieves speedups of 1.47× on JOB, 1.28× on SQLStorm, 1.10× on TPC-H, and 1.01× on Appian. Importantly, it avoids the significant performance regressions observed with the original RPT. These results demonstrate that RPT+ not only improves query performance but also maintains the robustness of RPT across diverse workloads.

PVLDB Reference Format:
Yiming Qiao, Peter Boncz, and Huanchen Zhang. Robust Predicate Transfer with Dynamic Execution. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/embryo-labs/dynamic-predicate-transfer.

1 INTRODUCTION

Joins dominate the execution time of analytical queries, making their optimization critical for database performance [19, 35]. A central challenge is the join ordering problem, where an optimizer must search a vast plan space, often relying on inaccurate cardinality estimates [8, 17, 40]. Despite decades of research, query optimizers frequently select poor join orders, resulting in execution plans that are significantly slower than optimal [4, 16, 20, 34, 43]. To address this issue for acyclic queries, Robust Predicate Transfer (RPT) builds on the Yannakakis algorithm [39] to ensure an efficient, join-order-robust execution [45].

The RPT algorithm operates in two phases: transfer and join. In the transfer phase, Bloom filters (BFs) on the join attributes are propagated across the join tree in forward and backward passes.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

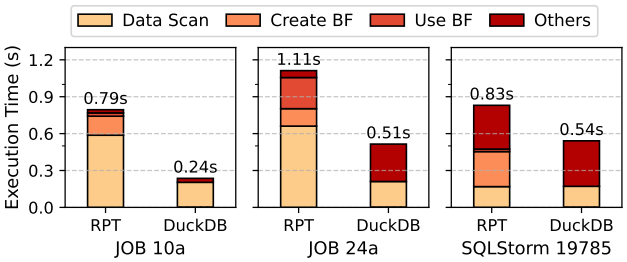


Figure 1: Performance Overhead of RPT in DuckDB v1.3.0 – RPT incurs significant overhead in data scanning and Bloom filter construction for many queries.

This process guarantees that all the irrelevant tuples from the base tables are eliminated. The subsequent join phase then executes the actual join operations on these reduced tables.

Although RPT is asymptotically optimal, it has constant-factor overheads that can slow down many queries. These shortcomings stem from two main issues. First, RPT overlooked the high cost of Bloom filter (BF) construction. Without auxiliary data structures such as Hyperloglog sketches [9], determining the optimal BF size requires a full data scan to count distinct keys, which materializes intermediate results and disrupts execution pipelines [14]. Second, RPT ignores the data layout on storage, which is critical for efficient data scanning [23, 41]. Because the BFs operate at the tuple level, they cannot be used to skip entire data blocks or row groups [13]. Figure 1 compares the performance breakdown of the original RPT and DuckDB on a few queries from the Join Order Benchmark (JOB) [17] and SQLStorm [26]. In these queries, RPT introduced large overheads due to unnecessary BF creation and data scanning.

In this paper, we propose RPT+ that improves the original RPT in three aspects. Our first contribution is the Asymmetric Transfer Plan (ATP), designed to minimize Bloom filter (BF) overhead. This approach modifies the shape of the join tree during RPT's transfer phase and employs distinct transfer plans for the forward and backward passes. This ensures that BFs are created and transferred only when necessary, cutting down on filter construction and probing costs drastically. We provide theoretical proofs of ATP's correctness and optimality.

Our second contribution is the Cascade Filter. It combines a lightweight min-max filter with a more precise Bloom filter (BF). The min-max filter can quickly eliminate row groups (i.e., storage data blocks) that do not contain relevant data, while the BF then operates on the remaining data at the tuple level. This hierarchical approach, i.e., transitioning from a fast, coarse-grained filter to a fine-grained one, significantly improves the overall filtering efficiency.

Finally, we introduce Dynamic Pipeline to make filtering more robust. This assesses predicate selectivity and materialization size at runtime to decide whether building a Cascade Filter is worthwhile.

If the filter is observed to be non-selective and/or very large, its construction is abandoned, removing the sink for its materialization. This runtime decision-making avoids the overhead of propagating useless filters and reduces materialization overhead.

We implemented RPT+ in DuckDB v1.3.0, a state-of-the-art in-process analytical database, and evaluated its performance across several real-world benchmarks: TPC-H [6], the Join Order Benchmark (JOB) [17], the Appian Benchmark [2], and SQLStorm [26]. Compared to baseline DuckDB, RPT+ achieves geometric mean speedups of 1.17× on TPC-H, 1.47× on JOB, 1.01× on Appian, and 1.28× on SQLStorm. Most importantly, RPT+ significantly reduces performance regressions observed with RPT. It eliminates all regressions in Appian and substantially reduces regressions across the 13,308 queries in SQLStorm.

2 BACKGROUND

In this section, we provide essential background on the Yannakakis Algorithm [39] and Robust Predicate Transfer (RPT) [33, 38, 45].

2.1 Yannakakis Algorithm

The Yannakakis algorithm [39] is a provably optimal join algorithm for acyclic queries. It is instance-optimal, guaranteeing a time complexity of $O(N + OUT)$, where N is the total input size and OUT is the final output size. The algorithm's key idea is to aggressively pre-filter tuples that will not participate in the final result [10]. This is done via a series of semi-join reductions. A semi-join ($R \ltimes S$) keeps only the tuples in the left relation (R) that have a matching join key in the right relation (S), i.e., $R \ltimes S = \pi_{\text{attr}(R)}(R \bowtie S)$. Despite its theoretical elegance, the Yannakakis algorithm is rarely used in practice due to its inefficient implementations.

2.2 Robust Predicate Transfer

The recent Robust Predicate Transfer (RPT) [38, 45] makes the Yannakakis algorithm practical in real database systems. For each semi-join $R \ltimes S$, RPT filters R using a Bloom filter (BF) built on S. To preserve the theoretical robustness of the Yannakakis algorithm, RPT applies the LargestRoot algorithm [45] for planning the BFs' propagation. A query is modeled as a join graph $G(V, E)$, where an edge $e \in E$ connecting tables $R, S \in V$ has a weight equal to the number of shared join attributes. RPT then constructs a maximum spanning tree $\mathcal{T}$ (from G) rooted at the table with the highest cardinality by repeatedly selecting the highest-weight edge that connects a node in the tree to a node outside (break ties by choosing the larger table). $\mathcal{T}$ dictates a two-pass BF propagation, i.e., a leaves-to-root forward pass and a root-to-leaves backward pass, to fully reduce the tables before the subsequent join phase of RPT.

However, RPT lacks a specific tie-breaking rule when a candidate node shares maximum-weight edges with multiple nodes in $\mathcal{T}$. This choice significantly impacts tree topology and subsequent performance. Consider query JOB 3a (the query is α-acyclic) in Figure 2. We define an *Equivalence Class* as a set of attributes from different tables that are connected by equality join conditions (e.g., the blue and green attributes in Figure 2). $\mathcal{T}$ initializes with the largest table, `movie_info`, and attaches `movie_keyword`. Ambiguity arises when adding `title`, which shares join attributes with both existing nodes. Connecting `title` to the leaf node (`movie_keyword`) yields a deeper tree (Figure 2b), whereas connecting it to the root (`movie_info`) results in a broader structure (Figure 2c). In practice, RPT's implementation defaults to the former, favoring deeper trees to maximize early filtering during the forward pass.

3 ASYMMETRIC TRANSFER PLAN

This section introduces the *Asymmetric Transfer Plan* (ATP), a novel approach that uses different tree structures for the forward and backward passes in RPT. We first analyze the distinct purpose of each pass and then prove the correctness and optimality of ATP.

3.1 Motivation

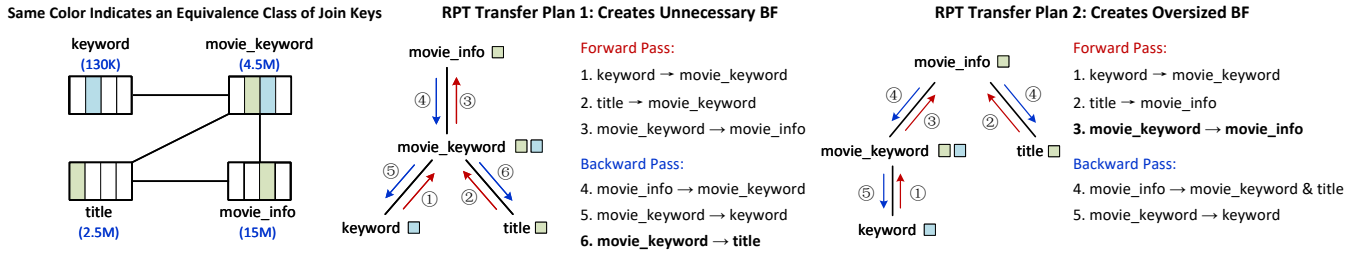
A key weakness of RPT is that it does not account for equivalence relationships among join keys [18]. This leads to two major inefficiencies, redundant filters and oversized filters, as illustrated by the two transfer plans in Figure 2. In Figure 2(b), the BF created from `movie_keyword` in Step 6 is redundant. Because the tables in Steps 4 and 6 share the same equivalence class (green), the filter from Step 4 could be applied directly to `title`. In Figure 2(c), Step 3 is suboptimal because it builds a BF from `movie_keyword` before that table is filtered. A better approach would be to first apply the BF from `title` (created in Step 2) to `movie_keyword`. This would shrink `movie_keyword`, allowing for a much smaller BF to be created subsequently. Therefore, an ideal transfer plan for the query in Figure 2 would combine the forward pass of Plan 1 with the backward pass of Plan 2. However, RPT's current design is limited to a symmetric transfer plan, where the forward and backward passes follow the same join tree.

3.2 Different Roles of Forward/Backward Passes

Our key observation is that RPT's forward and backward passes have fundamentally different goals. The forward pass is responsible for collecting filtering information from all tables, while the backward pass distributes this collected information back to them. This distinction means they benefit from different tree shapes: a deep, chaining tree is ideal for the forward pass, whereas a wide, broadcast tree is best for the backward pass.

A chaining forward pass is superior because it avoids creating oversized BFs by filtering one table before using it to build the next BF. Consider the example at the top of Figure 3, which shows a join over five tables $T_1, \ldots, T_5$ on a common join key. Let K_i be the set of distinct join key values in T_i. The forward pass starts at T_1 and transfers K_1 (as a Bloom filter) to table T_2, where T_2 refines the valid key set to $K_1 \cap K_2$. This transfer process continues (red lines) and eventually narrows down the set of valid keys (i.e., keys that must appear in the final output) to $\bigcap_{j=1}^{5} K_j$.

Similarly, a broadcast backward pass is more efficient because it allows the final, collected filter to be built only once and reused across all tables in the same equivalence class. In contrast, RPT's iterative backward pass (blue lines at the top of Figure 3) is wasteful. It redundantly constructs the same membership filter containing values in $\bigcap_{j=1}^{5} K_j$ for multiple tables. The ideal approach, shown at the bottom of Figure 3, is to construct this final filter just once and simply broadcast it to all relevant tables.



(a) Join Graph **(b) Transfer Plan 1** **(c) Transfer Plan 2**

Figure 2: Join Tree/Transfer Plan Candidates in RPT for the Query JOB 3a - There are two transfer plan candidates for the query: in Plan 1, RPT connects the table `title` to the node `movie_keyword`; while in Plan 2, it is connected to the root node `movie_info`.

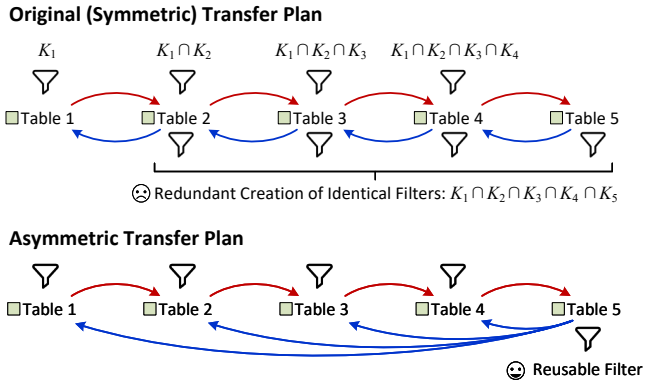


Figure 3: Transfer Plans for a Simple Join Query – RPT's backward pass (blue) constructs redundant membership filters. Instead, a single filter can be reused across all tables.

Algorithm 1: Asymmetric Transfer Plan

Input: Join graph $G(V, E)$
Output: Forward tree $\mathcal{T}_f$, backward tree $\mathcal{T}_b$

$\mathcal{T}_f \leftarrow \emptyset, \mathcal{T}_b \leftarrow \emptyset, \mathcal{R} \leftarrow V, \mathcal{R}' \leftarrow \{R_{\text{largest}}\}$;
while $\mathcal{R}' \neq \mathcal{R}$ **do**
  Find $R \in \mathcal{R} \setminus \mathcal{R}'$ with $e = \{R, S\} \in E(G), S \in \mathcal{R}'$ with the largest weight. Break ties by selecting the largest R;
  Find $S_f \in \mathcal{R}'$ and insert $e_f = \{R, S_f\} \in E(G)$ into $\mathcal{T}_f$. Break ties by selecting S_f with the **greatest** tree depth;
  Find $S_b \in \mathcal{R}'$ and insert $e_b = \{R, S_b\} \in E(G)$ into $\mathcal{T}_b$. Break ties by selecting S_b with the **smallest** tree depth;
  $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{R\}$;
return $\mathcal{T}_f, \mathcal{T}_b$;

3.3 Correctness of Asymmetric Transfer Plan

Our proposed algorithm for generating an asymmetric transfer plan, detailed in Algorithm 1, is a modification of the original Largest-Root algorithm. While it follows the same node insertion order, the key innovation is to apply distinct tie-breaking policies for the forward and backward passes. The algorithm constructs two maximum spanning trees on the weighted join graph, both rooted at the largest relation. It iteratively adds new tables by selecting the highest-weight connecting edge and uses the table size as the tie-breaker. The novelty lies in how the insertion point is chosen. For the forward tree, a new node is attached to the valid parent with the greatest depth, a policy that minimizes fan-out and promotes a deep, chaining structure. Conversely, for the backward tree, the new node is attached to the parent with the smallest depth, which maximizes fan-out and results in a wide, broadcast structure.

We now demonstrate that our asymmetric transfer plan still guarantees the full-reduction property for acyclic queries. First, consider the simple base case of a single attribute equivalence class where any two tables are connected in the join graph. Here, the forward pass collects "survived" keys at the root of its tree $\mathcal{T}_f$, and the backward pass distributes them from the root of its tree $\mathcal{T}_b$. Full reduction is guaranteed as long as both trees share the same root. Since Algorithm 1 explicitly selects the single largest table as the root for both $\mathcal{T}_f$ and $\mathcal{T}_b$, this property holds.

Next, consider an acyclic query with $m > 1$ equivalence classes: $Q = \{\alpha_1, \ldots, \alpha_m\}$. Here, an α can represent a *Compound Equivalence Class* (denoted as $\cap_{i=1}^{k} \alpha_i$) where a single join condition spans multiple (basic) equivalence classes (e.g., $S.a = R.a$ AND $S.b = R.b$). We prove full reduction for our algorithm by showing that each class $\alpha \in Q$ achieves full reduction independently. This involves analyzing the induced subgraph $\mathcal{T}^\alpha$ for each class α, which consists of all tables in the main tree $\mathcal{T}$ that contain attributes from α. Since $\mathcal{T}$ is a join tree, its induced $\mathcal{T}^\alpha$ is connected and is a tree. Because Algorithm 1 preserves the same node insertion order when building the forward tree $\mathcal{T}_f$ and the backward tree $\mathcal{T}_b$, they must share the same root (i.e., the first inserted node in α). Therefore, the proof of full reduction for each class α reduces to the base case above.

3.4 Optimality of Chaining Forward Tree

We then demonstrate ATP's optimality: for a fixed table insertion order, ATP generates tree topologies that minimize total filter creation and probing costs. We analyze the decoupled forward and backward passes independently to prove, among all tree shapes consistent with the insertion order, the resulting trees $\mathcal{T}_f$ and $\mathcal{T}_b$ minimize their respective costs (Section 3.4 and Section 3.5). This section establishes that the chaining forward tree minimizes costs by avoiding oversized filters. We derive this proof first for single equivalence classes before generalizing to multiple classes.

3.4.1 Single Attribute Equivalence Class. Consider an acyclic join over n tables involving a single equivalence class. Let N_i and K_i denote the cardinality and the distinct join keys in Table T_i (assuming $N_i \gg |K_i|$). We define per-tuple cost C_{probe} and C_{build} for BF probing and insertion. Given a join tree $\mathcal{T}$, let $\mathcal{S}_i$ denote the tables

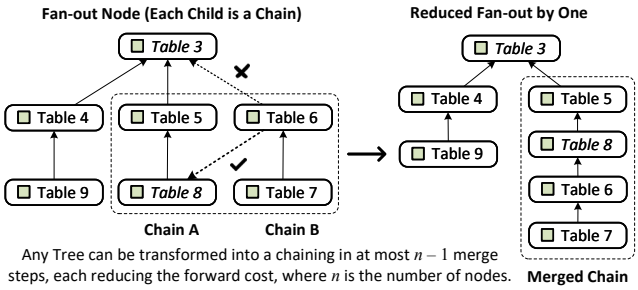


Figure 4: Leaf Chain Merge Operation - Given a node with multiple children, each forming a chain, we merge two chains by attaching one to the end of the other.

in the subtree rooted at T_i, and let C_i denote T_i's direct children. We define the join key intersection in subtree S_i as $K(\mathcal{S}_i) = \bigcap_{T_j \in \mathcal{S}_i} K_j$. During the forward pass, filters from C_i reduce T_i to keys in $K(\mathcal{S}_i)$.

Assuming a uniform distribution of join keys, the cardinality of T_i after filtering is $|T_i'| = |K(\mathcal{S}_i)|/|K_i| \cdot N_i$. Notably, $|T_i'|$ depends solely on $\mathcal{S}_i$, independent of the subtree's internal topology. Therefore, the total forward cost (building + probing) for T_i in $\mathcal{T}$ is:

$$L(T_i) = |K(\mathcal{S}_i)|/|K_i| \cdot N_i \cdot C_{\text{build}} + |C_i| \cdot N_i \cdot C_{\text{probe}}$$

We then prove that a chaining structure minimizes this cost by showing that transforming an arbitrary tree into a chain via iterative "leaf merging" strictly reduces the total cost. Consider a node T_p with child chains A and B (Figure 4). Merging B onto the leaf T_a of chain A affects only the subtree rooted at T_p. For any internal node $T_j \in A$, the subtree expansion implies a smaller key intersection ($|K(\mathcal{S}_j)| \to |K(\mathcal{S}_j')|$), thus reducing filter-building costs:

$$\Delta L(T_j) = (|K(\mathcal{S}_j')| - |K(\mathcal{S}_j)|)/|K_j| \cdot N_j \cdot C_{\text{build}} \leq 0$$

For T_a, its build cost decreases due to the subtree expansion, but it gains a child ($\Delta L(T_a) \leq N_a \cdot C_{\text{probe}}$). On the other hand, T_p reduces its probing cost by losing a child ($\Delta L(T_p) = -N_p \cdot C_{\text{probe}}$). Since Algorithm 1 prioritizes larger tables ($N_p \geq N_a$), the net probing cost decreases, yielding:

$$\Delta L(T_a) + \Delta L(T_p) \leq (N_a - N_p) \cdot C_{\text{probe}} \leq 0.$$

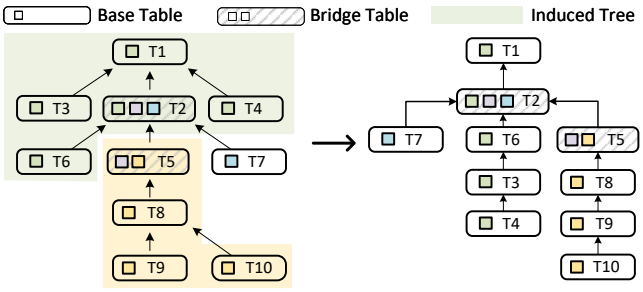
Consequently, the aggregate cost change is negative:

$$\Delta L = \Delta L(T_p) + \Delta L(T_a) + \sum_{T_j \in B,\, T_j \neq T_a} \Delta L(T_j) \leq 0.$$

Therefore, repeatedly applying the merge operations yields a cost-minimal chaining tree.

3.4.2 Multiple Attribute Equivalence Classes. We generalize the analysis to queries with $m > 1$ attribute equivalence classes, $Q = \{\alpha_1, \ldots, \alpha_m\}$. We show that the optimal forward tree organizes tables participating in each equivalence class α into a chain, and these chains are linked via "bridge tables", i.e., the root of the subtree induced by each class.

While the probing cost for each table T_i remains consistently $|C_i| \cdot N_i \cdot C_{\text{probe}}$, the build cost must account for filters from multiple join key classes (take Table 2 in Figure 5 as an example). Let Q_i denote the equivalence classes in which table T_i participates. Let K_j^α be the set of distinct values of α in T_j. Let $K^\alpha(\mathcal{S}_i)$ be the union of distinct values from all tables participated in class $\alpha \in Q_i$ in



(a) Multi-class Forward Tree **(b) After Transformation**

Figure 5: Chaining Forward Tree - In a join query with multiple attribute equivalence classes, the participating tables in each equivalence class form a chain.

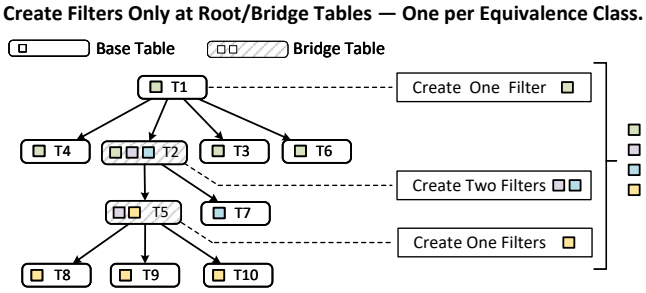


Figure 6: Broadcast Backward Tree - For each attribute equivalence class, a filter is created only once.

the subtree $\mathcal{S}_i$, i.e., $K^\alpha(\mathcal{S}_i) = \cup_{T_j \in \mathcal{S}_i,\, \alpha \in Q_j} K_j^\alpha$. Assuming statistical independence between attributes, the total forward cost compounds multiplicatively across all participating classes:

$$L_{\text{multi}}(T_i) = \Big(\prod_{\alpha \in Q_i} \frac{|K^\alpha(\mathcal{S}_i)|}{|K_i^\alpha|}\Big) \cdot N_i \cdot C_{\text{build}} + |C_i| \cdot N_i \cdot C_{\text{probe}}$$

This cost function also implies that T_i's cost depends solely on the subtree set $\mathcal{S}_i$ and its child count $|C_i|$.

To prove structural optimality, we treat each class $\alpha \in Q$ independently. Let $\mathcal{T}^\alpha$ be the subgraph induced by tables participating in α. Because the query is acyclic, $\mathcal{T}^\alpha$ must be connected [18, 39, 45], which forms a subtree rooted at the *bridge table* (the first table in $\mathcal{T}$ containing α). Since each $\mathcal{T}^\alpha$ involves a single key class, its optimal structure is a chain (as proven in Section 3.4.1). Therefore, transforming every induced subgraph $\mathcal{T}^\alpha$ into a chain while preserving connectivity at the bridge tables yields the global optimal tree (an example is provided in Figure 5(b)).

3.5 Optimality of Broadcast Backward Tree

We next show that the broadcast backward tree minimizes backward-pass costs by eliminating redundant filter construction. After the forward pass, the root holds $K^\alpha(\mathcal{S}_{\text{root}})$ for each equivalence class $\alpha \in Q$. The backward pass then distributes this set of globally valid join values from the root and bridge tables to all other nodes.

3.5.1 Single Attribute Equivalence Class. For a single equivalence class, the optimal transfer plan employs a star topology (i.e., depth $d = 2$) where the root connects directly to all the other $n - 1$ tables.

This requires constructing only a single BF at the root that is reused for all $n-1$ table probes. In contrast, any topology with depth $d > 2$ requires $d-2$ additional intermediate filter constructions to propagate the join value intersection without reducing any probing overhead. Thus, the broadcast topology minimizes the total backward-pass cost.

3.5.2 Multiple Attribute Equivalence Classes. When queries involve multiple equivalence classes, global connectivity is not guaranteed. Let's first assume that connected tables share exactly one attribute (Figure 6). We treat each class $\alpha \in Q$ as an induced subtree $\mathcal{T}_b^{\alpha}$. A single BF is constructed at the subtree's root and broadcast to all its descendants. This strategy minimizes overhead by ensuring exactly one filter creation per class and that each table probes a BF at most once. This optimality extends to multiple equivalence classes (where join conditions span multiple attributes). Filters are generated exclusively at root or bridge tables. Consider a query over tables S, R, T with conditions inducing classes $\alpha_1 = \{a, b, e\}$ and $\alpha_2 = \{c, d\}$. This yields three potential compound equivalence classes, for constructing BFs: α_1, α_2, and $\alpha_1 \cap \alpha_2$. Our broadcast backward tree avoids redundant filter constructions. For example, if no table contains α_2 in isolation, no separate BF is built for the table. Regarding probing, because every non-root table must receive filter data at least once, the theoretical lower bound is $n-1$ probes. We achieve this bound by ensuring every table is probed exactly once and confirm the optimality of the broadcast backward pass.

3.6 Attribute Correlation and Predicates

The forward pass assumes attribute independence and therefore collects predicate information from all tables. However, data correlation often renders specific transfers redundant. Consider a PK–FK join without local selection. a PK-derived BF cannot reduce the FK side due to referential integrity. FK-to-PK filtering is often ineffective as well because fact tables typically span the full join attribute domain [29, 37]. Consequently, PK–FK joins without selective predicates gain negligible benefit from predicate transfer [37, 45] (refer to Section 6.2 Appian Benchmark). Because analytical workloads typically involve selective predicates in practice, an optimization is to restrict filter construction to tables that are *effectively filtered*, i.e., those containing selective predicates. We implement this optimization via the Dynamic Pipeline in Section 5.

4 CASCADE FILTER

In this section, we introduce the concept of *Cascade Filters* and a simple yet effective implementation with min-max and Bloom filters to accelerate semi-join reductions.

4.1 Concept & Implementation

A cascade filter applies increasingly selective filters, starting with lightweight, coarse-grained pruning before computationally expensive, fine-grained checks. Unlike standalone BFs, this multi-stage approach discards the bulk of irrelevant data early, minimizing the load on subsequent more expensive filters.

Our implementation pairs a min–max filter for block-level pruning (i.e., discarding row groups with disjoint ranges [46]) with a BF for tuple-level probabilistic pruning. This design generalizes the hash join's "filter-then-expand" model [4]. It extends the original single exact hash check into a coordinated three-layer pruning: global range pruning (min–max), probabilistic pruning (BF), and exact hash filtering. Cascade filters support multi-hop, bi-directional propagation across the query plan. As shown in Figure 7, filters generated in one pipeline are probed in subsequent ones (denoted using matching colors). To maximize selectivity, min-max filters are pushed to the deepest possible operators. Creating a cascade filter acts as a pipeline breaker (materialized operator), and downstream pipelines cannot begin loading data to probe until the corresponding filters are constructed. Notably, the inter-pipeline materialization cost significantly outweighs the overhead of filter construction.

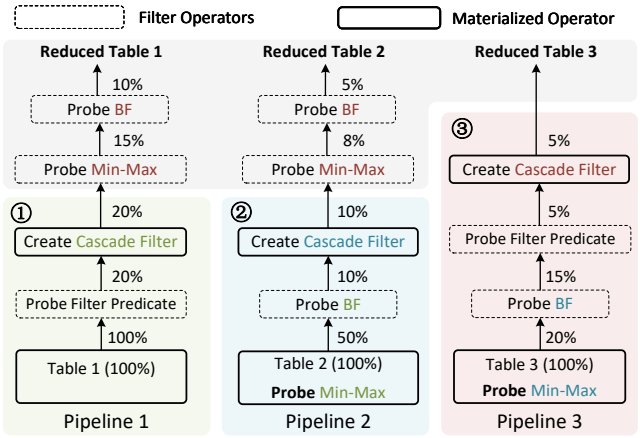


Figure 7: Transfer with Cascade Filters - We show the transfer paths Table 1 → Table 2, Table 2 → Table 3, and Table 3 → Table 1 & 2 across tables. Matching colors indicate the same cascade filter.

4.2 Min-Max Sensitivity to Bloom Filters

Unlike classical Side-Information Passing (SIP) [11], which applies filters locally, our multi-hop transfer can accumulate the filters' one-sided errors. False positives (FPs) in early-stage BFs artificially expand the min–max ranges propagated to subsequent hops. While minor in single-hop SIP, these errors degrade the pruning efficiency of downstream cascade filters.

For example, a few BF false positives can expand the min–max range of an empty-result join from $\varnothing$ to a wide interval and consequently compromise the pruning efficiency. Empirically, improving BF accuracy yields a 4× speedup on JOB query 07c by preventing such range inflation. Formally, consider a domain $T = \{1, \ldots, n\}$. Values inside $[a, b]$ are always kept, while values outside survive as FPs with probability p. For $n \gg 1/p$, the expected min-max range of the survivor set S is approximately: $\mathbb{E}[\max(S) - \min(S)] \approx n - 2/p$. This shows that unless p is very small, the expected range spans nearly the entire domain, rendering min–max pruning ineffective.

Consequently, the BF in our cascade filters must maintain a very low FPR across multiple hops, not merely a low probe cost. Lang et al. [14] show that optimal performance depends on both probe cost and the amount of work a filter avoids. In our setting, the latter is crucial so that accuracy therefore takes precedence over raw throughput. Guided by these considerations, we adopt the Cache-Sectorized Bloom Filter (CSBF) [14]. CSBFs align filter blocks with cachelines and distributing hash bits across sectors, achieving

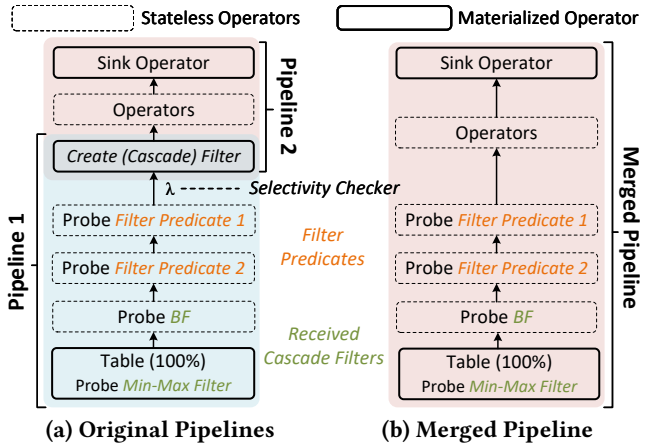


Figure 8: Dynamic Cascade-Filter Creation - The default plan has two pipelines, connected by a cascade-filter creator. If the estimated selectivity λ is high (e.g., 95%), Pipeline 1 is halted, filter creation is skipped, and its operators are moved into Pipeline 2.

Algorithm 2: Dynamic Bloom-Filter Creation

Input: Pipeline P, table row count N_{total}
Initialize: $N_{scan} \leftarrow 0$, $N_{recv} \leftarrow 0$, decided $\leftarrow$ false;
while P *produces a chunk* C **do**
  $N_{scan} \leftarrow N_{scan} + |C|$;
  $N_{recv} \leftarrow N_{recv} + |PassFilter(C)|$;
  if $\neg decided$ *and* $N_{recv} \geq \gamma$ **then**
    decided $\leftarrow$ true
    Est. selectivity: $\lambda \leftarrow N_{recv}/N_{scan}$;
    Est. materialized size: $S \leftarrow \lambda \cdot N_{total} \cdot avg\_tuple\_size$;
    if $\lambda > \tau_{sel}$ ***and*** $N_{scan}/N_{total} < \tau_{prog}$ **then**
      return Abandon BF and Merge Pipelines;
    if $S > M_{avail}$ **then**
      return Abandon BF and Merge Pipelines;
  Materialize $PassFilter(C)$ locally;
return Build BF from all materialized tuples;

throughput comparable to register-blocked BFs [22] while maintaining an FPR close to classic BFs.

5 RUNTIME FILTER SKIPPING

To preclude inefficient cascade filter constructions, RPT+ introduces the *Dynamic Pipeline*. A dynamic pipeline performs runtime checks to adaptively abort the construction of filters for tables with predicates that are found to be ineffective.

5.1 Identifying Efficiently Filtered Tables

Because cascade filters constructed on tables without selective predicates rarely justify their materialization cost (cf. Section 3.6), ATP restricts the forward transfer plan to tables with predicates, plus bridge tables (cf. Section 3.4). This forward-pass heuristic trades the theoretical guarantee of full semi-join reduction for significantly reduced overhead. However, the presence of a predicate does not guarantee utility; a high pass rate (e.g., 99%) makes the transfer overhead outweigh its benefit. Because classic cardinality estimation is often unreliable, we introduce *Dynamic Pipeline* to monitor selectivity at runtime and adaptively abort filter construction if the observed filter selectivity exceeds a viable threshold.

5.2 Dynamic Pipelines

The Dynamic Pipeline leverages runtime progress estimation [5, 15] to modulate execution. By comparing the cardinality at the pipeline source against that at the sink (i.e., materialization point), the system calculates real-time filter selectivity. Filter construction is aborted if (1) selectivity is too high or (2) the estimated result size exceeds the memory budget. This also eliminates the downstream probing of the abandoned filters. This ensures that RPT+ only builds useful filters and avoids the overhead of materializing large intermediate results.

This runtime approach is both practical and robust. The necessary tuple counts are already tracked by many modern analytical databases [25, 27, 28] (displayed in EXPLAIN ANALYZE) for query progress monitoring [5, 15]. The runtime estimation offers superior accuracy to static, histogram-based optimizer estimates [24, 35, 44]. Dynamically adjusting the query plan is also lightweight. It has been proposed in streaming for optimizing tuple routing [3]. Because each cascade filter creation introduces a "pipeline breaker", abandoning a filter simply removes the sink that materializes tuples and thereby undoes the pipeline-cut. This is a minor modification that does not reorder any operators within the now-unified pipeline, making the adjustment process fast and simple.

Our approach differs from Adaptive Query Execution (AQE) in Databricks Photon [36]. Photon's plan adaptiveness occurs only at shuffle boundaries (i.e., pipeline breakers with disk-based synchronization), which are *after* the full materialization, making statistics gathering and plan reoptimization straightforward. In contrast, Dynamic Pipeline targets analytical engines built around streaming pipelines where explicit pipeline breakers are uncommon. Dynamic Pipeline operates *while* the materializing pipeline runs to adjust the transfer logic on-the-fly, avoiding materialization cost that Photon cannot. This enables the fine-grained adaptivity that streaming analytical engines without shuffle-style materialization require.

5.3 Dynamic Cascade-Filter Creation

We now describe how RPT+ dynamically decides whether a cascade filter should be built, following Algorithm 2. We illustrate the process using the example in Figure 8.

In Figure 8(a), the default plan consists of two pipelines: Pipeline 1 scans the base table and applies a series of filter predicates, min–max filters, and received cascade filters; Pipeline 2 contains the operators that consume the intermediate table produced by Pipeline 1. A selectivity checker is injected immediately before the cascade-filter creator to observe how many tuples survive all upstream filters. It maintains two counters during execution: N_{scan} (tuples scanned from the table) and N_{recv} (tuples that pass all upstream filters). Once at least γ tuples have reached the cascade-filter creator, we estimate the overall retained selectivity as $\lambda = N_{recv}/N_{scan}$. This measurement captures the combined filtering effect of all upstream predicates without separately estimating each one.

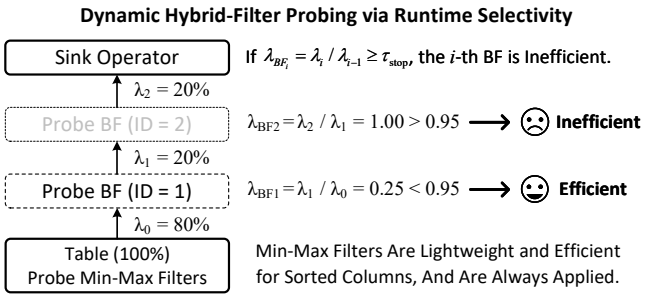


Figure 9: Selective Cascade-Filter Probing – start with a min-max filter, then apply the i-th BF only if $\lambda_i/\lambda_{i-1} < \tau_{stop}$. Here, BF (ID = 1) is applied, while BF (ID = 2) is skipped.

A cascade filter is created only when the retained selectivity is sufficiently low. If the estimated selectivity is high ($\lambda > \tau_{sel}$) and the scan is still in its early stage ($N_{scan}/N_{total} < \tau_{prog}$), Algorithm 2 abandons filter creation. The second condition prevents canceling late in the scan, when most of the materialization cost has already been incurred. In this example, Pipeline 1 halts early, the cascade-filter creator is skipped, and its operators are merged into Pipeline 2. Execution then proceeds in Pipeline 2, as shown in Figure 8(b).

The algorithm also has a memory-safety check, matching the materialized operator shown in Figure 8. Since creating a cascade filter requires materializing all tuples reaching the creator, the algorithm estimates the materialized size S. If this exceeds the memory budget M_{avail}, the filter is similarly abandoned. If none of the abandonment conditions are triggered, Pipeline 1 proceeds to materialize the received tuples and builds the cascade filter as shown in Figure 8(a). And Pipeline 2 continues with the local materialized data.

5.4 Selective Cascade-Filter Probing

We now introduce selective cascade filter probing in dynamic pipelines, using an example in Figure 9. We can bypass probing optional filters with weak filtering ability, as they provide negligible pruning benefits. By skipping these probes, we reduce the probing overhead without compromising correctness.

Figure 9 gives an example of a pipeline with two BF probes. Min-max filters, being lightweight and highly effective on sorted columns, are always applied first. They eliminate entire row groups outside the query range, ensuring that the data seen by subsequent BFs is more representative, thereby stabilizing selectivity estimates.

Each BF-probing operator records two counters to estimate its selectivity: one for the number of tuples entering the operator and one for the number that pass through. After sampling, we compute the ratio of out to in. If this ratio exceeds the stop threshold (e.g., 0.95), we disable further probing of that filter. Disabling is achieved by bypassing the BF operator entirely: all input tuples are forwarded directly to the subsequent operators without any BF lookup. Dynamic probing can be biased on sorted or nearly sorted columns because clustered values make early selectivity samples unrepresentative. Cascade filtering mitigates this by applying the min–max filter before any BF probing. Sorted columns generate tight row-group zonemaps, and the min–max filter prunes groups that fall outside the desired range. BFs are then evaluated only on the remaining groups, so the observed selectivity reflects the BF's true filtering power rather than ordering-induced skew.

In some cases, a BF-probing operator may lack a valid BF because the creating the filter was abandoned by the dynamic pipeline mechanism described earlier. When this occurs, the probe is always deemed ineffective and is disabled. To allow dependent pipelines to proceed without deadlock, we have marked the canceled BF-creation pipeline as "finished," as explained in Section 5.3.

5.5 Implementation Details

For the dynamic pipeline, we receive γ = 100K tuples to estimate the selectivity. For the selective filter creation, we set τ_{sel} = 0.35, τ_{prog} = 0.6, and M_{avail} = 64 GB. For the selective filter probing, we set τ_{stop} = 0.9. We use sequential sampling instead of random sampling because random sampling requires random access patterns that disrupt streaming execution and incur significant overhead. Sequential sampling, by contrast, integrates naturally into the pipeline with minimal cost.

For the cascade filter, because BF probing is compute-bound, we implement CSBFs using a vectorized approach following [14]: elements are processed in batches, enabling the compiler to auto-generate SIMD instructions. Each CSBF block is aligned to a 64-byte cache line and subdivided into 32-bit sectors. For each element, we compute a 64-bit hash and split it into two 32-bit values, allowing the use of 32-bit gather instructions (e.g., `VPGATHERDD`) instead of 64-bit gathers (e.g., `VPGATHERQD`). To achieve a low false-positive rate, we allocate 20 bits per key and use 7 hash functions, so each key sets 7 bits in the filter. Our CSBF implementation achieves a probe latency of 2.48 cycles/tuple with an FPR of 6.1×10^{-5}.

6 END-TO-END EVALUATION

In this section, we integrate RPT+ into DuckDB v1.3.0 [25] and evaluate its end-to-end performance. RPT+ is to reduce the impact of suboptimal join orders, so workloads dominated by large joins are the most suitable evaluation targets. These workloads are analytical in nature and best served by columnar, vectorized systems. DuckDB is a state-of-the-art in-process OLAP engine with columnar storage, vectorized execution, and basic SIP [11], making it a representative system for the evaluation.

Benchmarks. We use four benchmarks in the evaluation: TPC-H (SF=100) [6], Join Order Benchmark (JOB) [17], Appian Benchmark [2], and SQLStorm [26]. TPC-H is a widely used decision support benchmark consisting of 22 analytical queries over a synthetic business schema. We omit four queries that do not involve join operations. The JOB consists of 113 multi-join queries based on the IMDB dataset, offering a challenging and diverse workload. The Appian Benchmark is an industry-driven workload designed to reflect real-world query patterns and data distributions. SQLStorm, built using Large Language Models (LLMs), utilizes the Stack Overflow Math database with over 18,000 queries. It greatly expands the scope of SQL functionality and query constructions.

Baselines. We compare RPT+ with the following four baselines: (1) *NoFT*: A modified version of DuckDB v1.3.0 [25] with all filter transfer mechanisms disabled. It executes pure hash joins without any pruning. (2) *DDB*: An unmodified DuckDB release v1.3.0 that supports min-max filter transfer on join keys from the hash-table build side to the probe side, but does not support Bloom filters (BFs). (3) *SIP*: A modified DuckDB implementation from [30], which

Table 1: Query Distribution by Speedup Range for RPT

Speedup	#Queries (%)	Avg. Speedup	Avg. Latency (ms)
0–0.7	555 (4.2%)	0.62×	82.32→133.01
0.7–0.9	3185 (23.9%)	0.81×	208.61→248.75
0.9–1.1	6323 (47.5%)	0.99×	782.53→788.91
1.1–5.0	3061 (23.0%)	1.85×	377.70→235.49
5.0–100.0	179 (1.3%)	9.09×	841.04→78.45
>100.0	5 (0.0%)	127.48×	4515.65→35.30

Table 2: Query Distribution by Speedup Range for RPT+

Speedup	#Queries (%)	Avg. Speedup	Avg. Latency (ms)
0–0.7	24 (0.2%)	0.61×	69.54→114.58
0.7–0.9	247 (1.9%)	0.85×	639.31→734.16
0.9–1.1	9124 (68.6%)	1.01×	570.86→567.76
1.1–5.0	3389 (25.5%)	1.89×	405.74→260.58
5.0–100.0	512 (3.8%)	8.93×	401.93→40.44
>100.0	12 (0.1%)	226.29×	3200.65→14.40

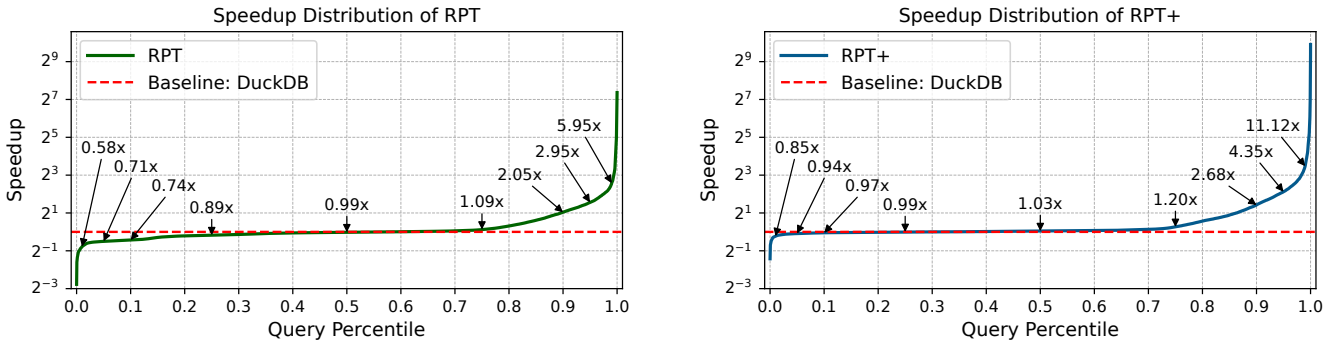


Figure 10: Speedup Distribution Comparison — We compare the percentile-wise query speedups of RPT and RPT+ over the baseline.

realizes the Sideways Information Passing [11], enabling both min-max and Bloom filter (BF) transfer during hash joins. (4) RPT: Robust Predicate Transfer [45], which constructs a join tree using the LargestRoot algorithm and transfers only BFs along the tree.

Experimental Setup. We run experiments using our in-house server with 512 GB of DDR5 main memory at 4800 MHz and a 1 TB Intel® SSD D5-P5530. The server is equipped with two sockets of Intel® Xeon® 8474C 2.1 GHz processors (48 cores), each capable of supporting 96 threads. We use Debian GNU/Linux 12 and GCC 12.2 with -O3 enabled. Experiments are conducted with eight threads unless stated otherwise.

6.1 RPT+ Significantly Reduces Regressions

We first study how RPT+ improves robustness by mitigating RPT-induced regressions. While RPT provides pruning benefits, it also causes noticeable slowdowns on a non-trivial subset of queries. RPT+ is designed to address this. By analyzing the distribution of speedups over RPT, we show that RPT+ not only improves overall performance but, more importantly, suppresses the tail of negative outliers. For this analysis, we use the SQLStorm benchmark, which provides a large and diverse workload. Due to its query complexity, few systems can execute all queries successfully [26]. Following the guidance in the original paper, we terminate any query that exceeds 10 seconds. Out of 18,251 queries, 13,308 finish successfully, 2,837 fail, and 2,106 time out. We compare per-query speedups over DuckDB across percentiles.

The results in Figure 10 show the distribution of speedups of RPT+ and RPT relative to DuckDB. The x-axis represents query percentiles and the y-axis shows speedup. The green and blue curves correspond to RPT and RPT+, and the red dashed line indicates the baseline. RPT+ greatly reduces query regressions. Both figures show two characteristic tails: the left tail reflects regressions and the right tail reflects improvements. RPT produces noticeable slowdowns (speedup $< 0.9\times$) for at least 28% of queries, while RPT+ lowers this to 2.1%. On the improvement side, RPT+ attains higher speedups for more queries than RPT, reaching up to 500×.

For a more detailed comparison, Table 1 and Table 2 report the per-query speedup distributions. Two observations stand out. First, RPT+ includes more queries with high speedups and substantially reduces the number of queries in the $[0.7, 0.9)$ interval. Second, queries that are slow under DuckDB often achieve larger gains. These queries are typically slow because of suboptimal join orders, and RPT+ is designed to correct such cases. In contrast, fast queries, whose join orders are already good, show smaller changes and account for most of the remaining regressions.

6.2 RPT+ Improves Average Performance

We next show that RPT+ not only reduces regressions but also improves average performance across several benchmarks. We compare RPT+ against NoFT, DuckDB, SIP, and RPT. All experiments use 8 threads. Each query is executed once for cache warm-up followed by five measured runs, and we record the average execution time. Unless noted otherwise, per-query speedups are computed by taking the geometric mean of five speedups, each defined relative to the corresponding baseline run.

To characterize selectivity patterns across workloads, we execute all queries in each benchmark and record the selectivity of every predicate. Aggregating these measurements yields a selectivity distribution for each benchmark, as shown in Figure 12. TPC-H and JOB contain mostly highly selective predicates. Appian is dominated by non-selective predicates with limited filtering power. SQLStorm exhibits a mixture of both highly and weakly selective predicates.

Figure 11 reports results on four benchmarks with distinct data and workload characteristics. Across all of them, RPT+ delivers consistently strong performance. (1) TPC-H. Most columns follow uniform distributions, making min–max filters ineffective. NoFT and DuckDB therefore perform similarly. RPT+ outperforms RPT mainly due to lower filter-construction overhead. (2) JOB. Many

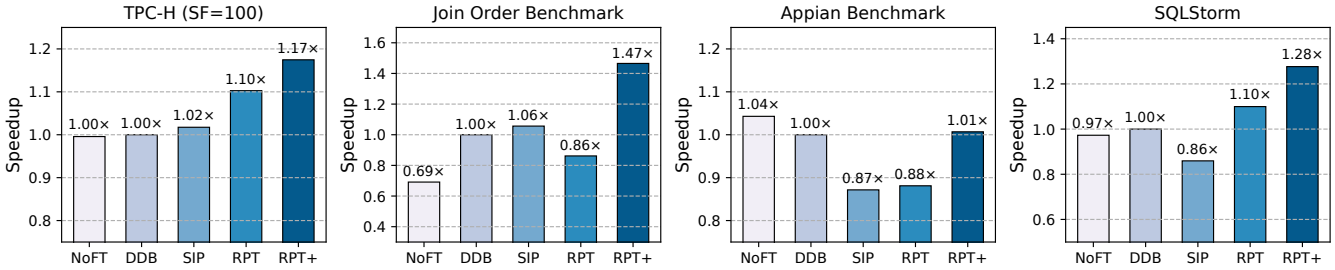


Figure 11: Performance Overview - We report the geometric mean speedup of all methods over DuckDB v1.3.0 across benchmarks with diverse data distributions and workloads. NoFT: no filter transfer; DDB: transfers only min-max filters; SIP: transfers both min-max and BFs.

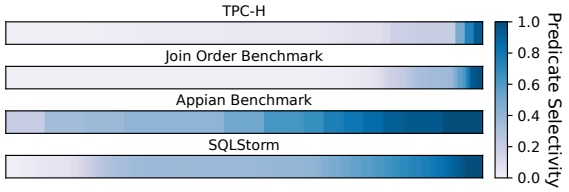


Figure 12: Selectivity Distribution - We measure predicate selectivity across benchmarks. Shallow color means strong filtering.

queries produce tiny intermediate results, sometimes only a few tuples or even empty. Min–max filters are thus highly effective at skipping entire row groups, and methods that use them outperform those that do not (e.g., NoFT, RPT). (3) Appian. Very few base-table rows can be filtered, leaving little pruning opportunity. Skipping filter transfer is ideal. RPT+ keeps overhead minimal in this non-prunable setting due to dynamic pipelines. (4) SQLStorm. This workload reflects real-world data distributions. Both min–max filters and BFs help. RPT+ balances pruning benefit against transfer cost and achieves the best overall trade-off.

6.3 Transfer Plan Comparison

We conduct an apples-to-apples comparison to evaluate the advantage of ATP over LargestRoot, as discussed in Section 3. We compare their speedups and memory usage across the base versions, their Cascade-Filter variants, and the ATP+/LargestRoot+ extensions that incorporate dynamic pipelines. All benchmarks run with 8 threads. For each query, we record execution time and peak memory usage, and report the geometric-mean speedup and average peak memory consumption per benchmark.

Figure 13 summarizes the results. First, the ATP family outperforms the LargestRoot family, benefiting from its more effective transfer plans. Second, Appian contains almost no selective predicates, so predicate transfer adds time and memory overhead; dynamic pipelines address this effectively. Third, on JOB, methods without cascade filters show higher memory usage because disrupted min–max transfer causes additional data to be loaded; Cascade filters enable row-group skipping and restore normal usage. We emphasize that the goal of RPT+ is to avoid the regressions seen in prior approaches; performance gains are a secondary benefit.

6.4 Per-Query Analysis on JOB

To further compare ATP and LargestRoot, we analyze per-query performance on the JOB. We study four methods: ATP, LargestRoot, ATP+, and LargestRoot+, where the latter two extend their base versions with cascade filters and dynamic pipelines. JOB contains 33 query templates, and we group queries by template. For each template, we report the average execution time over all instantiated queries (i.e., different parameterizations of the same template).

Figure 14 shows that both ATP and LargestRoot introduce regressions (Queries 8, 10, and 24) because they prevent the row-group skipping. Inserting a cascade-filter–creation operator on the probe side introduces a materialization point that breaks the pipeline. Instead of waiting for min–max filters from the build side to enable early skipping, the split pipeline begins scanning before the filter is ready, resulting in unnecessary work. Adding cascade filters restores row-group skipping for both ATP+ and LargestRoot+. Moreover, ATP+ inserts fewer BFs, which further reduces regressions (Queries 18, 25, 30, and 31).

6.5 Sensitivity of Dynamic Pipeline Threshold

We evaluate the effect of the give-up threshold τ_{sel} in the dynamic pipeline. The threshold decides whether a cascade filter is created: if the fraction of surviving tuples exceeds the threshold, we build the cascade filter; otherwise, we skip it. Lower thresholds drop filters more aggressively, reducing overhead but also limiting pruning. We present results on the JOB and Appian benchmarks, as the other workloads show similar trends.

Figure 15 shows the speedup of RPT+ over DuckDB under different thresholds. On JOB, moderate thresholds (e.g., 0.1–0.3) preserve useful filters and yield higher speedups, whereas very low thresholds (e.g., 0.0) discard almost all filters and hurt performance. Note that some cascade filters are created before selectivity sampling; this is intentional because filters on small tables are typically beneficial, so a threshold of 0.0 still produces a few BFs. In the Appian benchmark, most predicates are not selective, therefore cascade filters offer little pruning benefit. Cascade filters built on such tables not only fail to reduce data but also add construction and probing overhead, making them unprofitable to keep.

6.6 Case Study

We then conduct a detailed analysis of three queries, JOB-10a, JOB-07c and query 15651 & 13615 in SQLStorm.

6.6.1 **[Asymmetric Transfer Plan]** *JOB-10a.* Query 10a joins seven tables, where RPT incurs substantial BF–building overhead due to a suboptimal transfer plan. We compare LargestRoot and ATP, recording total execution time, cascade-filter build time, data

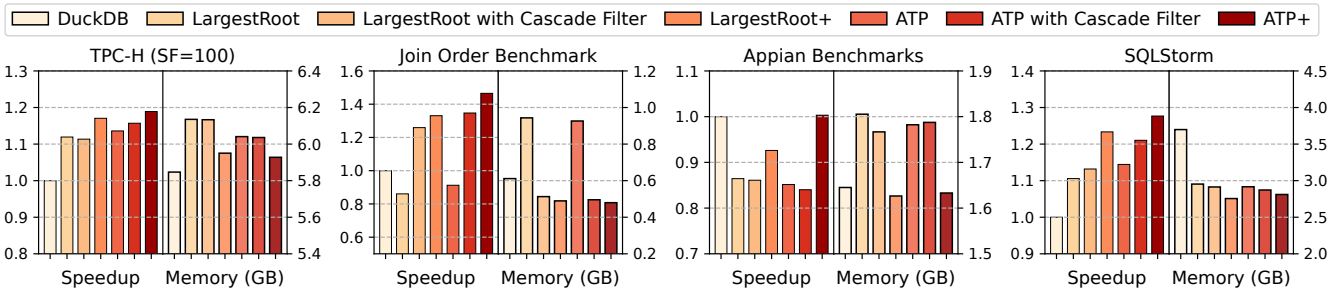


Figure 13: ATP vs. LargestRoot - We compare the speedup and memory usage of ATP and LargestRoot in an apples-to-apples setting: the base versions, their variants with Cascade Filters, and the ATP+/LargestRoot+ extensions that additionally incorporate dynamic pipelines.

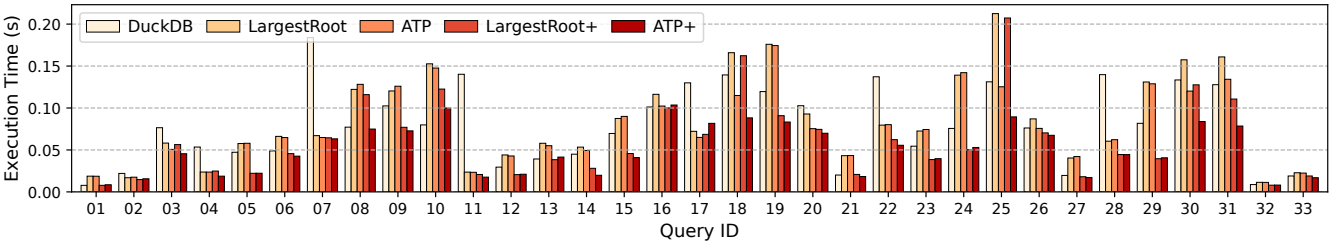


Figure 14: Per-Query Analysis on Join Order Benchmark - We show the execution times of five methods for each query template in JOB, where the execution time of each query template is the mean of the execution times of its instantiations.

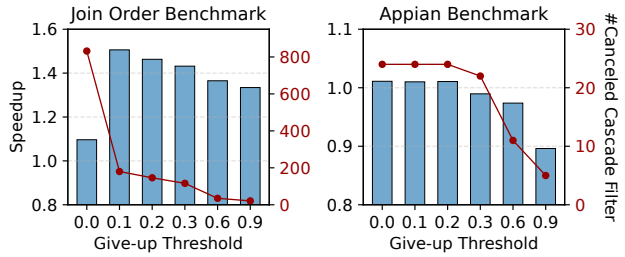


Figure 15: Sensitivity of the Give-up Threshold in Dynamic Pipelines - We vary the give-up threshold τ_{sel} used in the dynamic pipeline to show when and how often this mechanism is triggered.

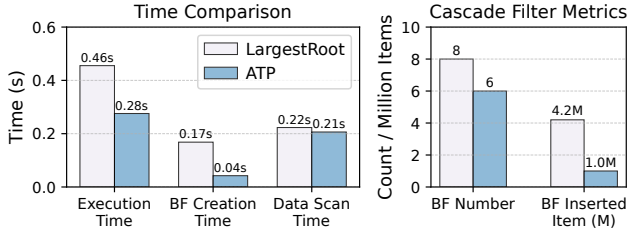


Figure 16: Cascade Filter Creation Overhead in Query JOB-10a - We compare the total execution time, BF creation time, and data scan time of the LargestRoot and ATP, along with the number of BFs created and their total contained item count.

scan time, the number of created BFs, and the total number of rows inserted into them. As shown in Figure 16, ATP outperforms LargestRoot, because LargestRoot creates an unnecessary BF on the second-largest table, which has no filter predicate; this cascade filter provides no pruning benefit and adds significant probing overhead.

6.6.2 **[Cascade Filters]** *JOB-07c.* The BF accuracy is crucial for Query 07c in JOB. Ten cascade filters are created for this query, indexed by their creation order. We vary the bits-per-key and measure how many tuples are pruned by row-group skipping and by min–max/BF filtering. We also record the number of items inserted into each BF in the cascade. As shown in Figure 17, high BF accuracy (24 bits per key) enables better row-group skipping on `cast_info`, yielding about a 4× speedup compared to the low-accuracy setting (16 bits per key). The key reason is that `cast_info` receives cascade filters 4 and 6 during scanning. Under the high-accuracy setting, BF 6 is very small and effective (14 items), whereas under low accuracy it contains 378 items and loses pruning power. We also observe that early BFs insert the same number of items across configurations, while later BFs diverge due to accumulated false-positive errors.

6.6.3 **[Dynamic Pipeline]** *SQLStorm 15651 & 13615.* Query 15651 in SQLStorm joins `Post` and `Users`, filters on `Post.PostTypeId = 1`, orders by `ViewCount`, and returns the top 10 rows. The query has two pipelines: building a hash table on `Users`, and probing `Post` with a top-k optimization that avoids scanning tuples with larger `ViewCount`. A cascade-filter creator inserted into the probing pipeline splits this pipeline and disables the top-k optimization, causing RPT+ (without the dynamic pipeline) to run 3× slower than DuckDB. With the dynamic pipeline enabled, RPT+ detects that the predicate on `Post` offers little pruning and skips filter construction, eliminating the regression. Query 13615 is the worst-performing case for RPT+, with a 0.37× speedup. It has the same issue as Query 15651, except that the dynamic pipeline deems the predicate selective and does not cancel cascade-filter creation. This leads to unnecessary scanning and a performance drop.

7 MICROBENCHMARK EVALUATION

In this section, we use synthetic workloads to evaluate each component of RPT+. We first compare ATP with LargestRoot [45], then

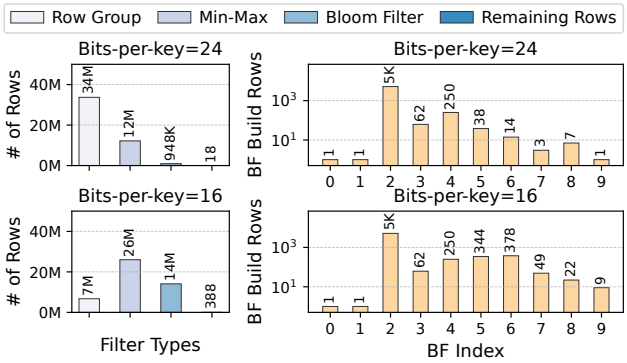


Figure 17: Impact of BF Accuracy on Min-Max Filtering in JOB-07c – We vary the bits-per-key parameter to study its effect on filtering accuracy and interaction with min-max filters. We report filtered row counts at each level and BF insertion counts.

assess the effectiveness of cascade filters, and finally analyze the sensitivity of dynamic pipelines.

7.1 Transfer Plan on Synthetic Data

We consider a natural join over tables $T_1, \ldots, T_n$. Each table contains 1M rows and two columns: `id` (ranging from 1 to 1M) and `info` (a fixed 100-byte string). The join order follows a left-deep plan: hash tables are built for $T_1, \ldots, T_{n-1}$, and T_n probes each of them. We apply filter predicates to the first m tables $T_1, \ldots, T_m$, with each predicate independent and having a selectivity of 0.5. Concretely, the predicate on T_i checks the i-th bit of `id`: $(\texttt{id} \gg (i-1)) \,\&\, 1 = 1$. We vary n from 6 to 18 and m from 0 to 3, and we measure the execution time of each query. We compare four strategies: (1) LargestRoot; (2) ATP; (3) ATP (No Focus), which includes all tables in the forward pass; and (4) the vanilla DuckDB v1.3.0.

Figure 18 presents the results. First, ATP (No Focus) outperforms LargestRoot because it creates fewer filters in the backward pass. As more tables carry predicates, the filtered tables shrink, reducing backward filter-creation cost and narrowing the gap. Second, ATP sharply reduces predicate transfer overhead, even when no tables have filter predicates. As shown in Figure 18, ATP bounds transfer overhead while still guaranteeing full reduction for each table, allowing it to outperform the baseline.

To further study the effect of filter predicate selectivity on transfer plans, we fix the number of tables to eight and apply filter predicates to three of them, varying selectivity from 0.1 to 1. Figure 19 shows the results. With low selectivity (e.g., 0.1), transferring BFs adds little overhead and ATP provides modest gains. As the selectivity increases, execution times rise for all plans because filter transfer becomes less beneficial. Overall, ATP incurs the lowest overhead. The results also highlight the importance of disabling ineffective cascade filters.

7.2 Min-Max Sensitivity to Bloom Filters

We next evaluate how the block-level skipping performance of cascade filters responds to BF accuracy. Consider a join query involving $n = 10$ tables $T_1, \ldots, T_{10}$, with only two filter predicates: $T_1.\texttt{id} \,\%\, 2 = 0$ and $T_2.\texttt{id} \,\%\, 2 = 1$, while the rest of the settings follow those in Section 7.1. The join result of T_1 and T_2 is $\varnothing$, allowing us to build a tight min-max filter for the remaining tables. However, false positives from the BF can expand the generated min–max range.

We consider three BF variants: (1) 32-bit Register-blocked BF, (2) 64-bit Register-blocked BF [31], and (3) CSBF [13]. All BF variants are implemented in a vectorized style. We use the ATP with each of these BF variants to execute the query, recording the execution time and the percentage of rows filtered by min-max filters. We vary the bits per key to adjust the accuracy.

Figure 21 shows the results. First, the min-max range is significantly affected by the number of bits per key. With fewer bits, BFs generate more false positives, which degrade the performance of the min-max filters. Second, the CSBF achieves higher accuracy with the same memory usage compared to other BF variants. It produces only 4 false positives when the number of bits per key exceeds 35, while maintaining comparable speed. For these reasons, we choose the cache-sectorized BF as our BF implementation.

7.3 Sensitivity of Dynamic Filter Probing

We quantify the performance impact of selective cascade-filter probing. We reuse the join queries from Section 7.1, which perform a natural join over $T_1, \ldots, T_n$. A single predicate ($\texttt{id} \,\%\, N = 1$) is applied to T_n, such that ATP creates a cascade filter that propagates to all preceding tables. We vary n from 3 to 4 and adjust N to control filter selectivity. We compare two static strategies: Always Probe and Never Probe. As shown in Figure 20, always probing helps when the filter is selective but introduces overhead when pruning is weak, while never probing avoids overhead but forfeits potential gains. These results underscore the need to probe adaptively. For more complex queries, the threshold shifts rightward. We find that a stop-probing threshold of $\tau_{\text{stop}} = 0.9$ provides robust performance.

7.4 Sensitivity of Dynamic Filter Creation

We then evaluate the sensitivity of dynamic pipelines to correlated queries. A correlated query contains filter predicates that are related through join conditions. We construct a JOB query template that retrieves films whose production country and actors' birth country are both X, where X is one of {USA, France, India, Italy, Japan}. We also generate fully correlated queries. Two tables, T_1 and T_2, contain identical data and receive the same filter predicate. We set predicate selectivity to 0.05, 0.1, and 0.2, and use the selectivity to name the queries in Figure 22. We further vary the give-up threshold τ_{sel} of dynamic pipelines to assess their sensitivity.

Figure 22 reports the results. For the correlated queries constructed from JOB, predicate transfer remains effective. Although the two correlated predicates do not prune each other's tables, the query joins several additional tables, and the transferred predicates help prune along the join path. For the fully correlated queries, predicate transfer provides no benefit: the two tables contain identical data and share identical predicates, so no additional pruning is possible. In these cases, transferring predicates only adds overhead, and a small give-up threshold helps avoid this unnecessary work.

8 RELATED WORK

Sideways Information Passing (SIP) [11, 12] transfers predicates locally between join participants. Some cost-based optimizers [7, 42] attempt to integrate SIP, but this enlarges the search space and

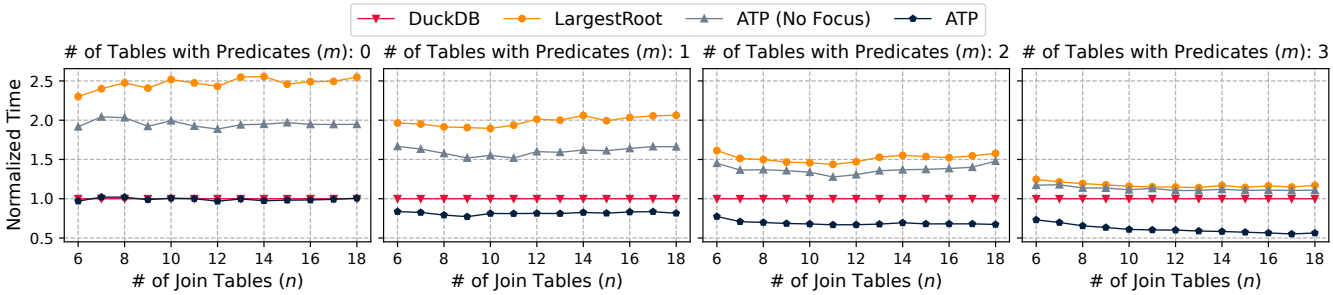


Figure 18: Execution Time of Transfer Plans - We compare various transfer plans using a join query of n tables, of which m tables have filter predicates. All join keys belong to the same equivalence class.

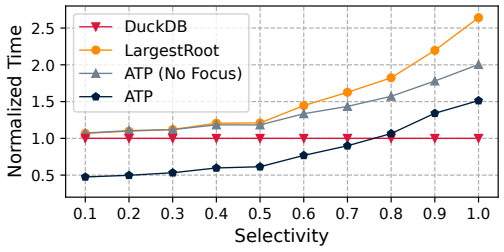


Figure 19: Execution Time vs. Predicate Selectivity - We vary predicate selectivity to compare various transfer plans.

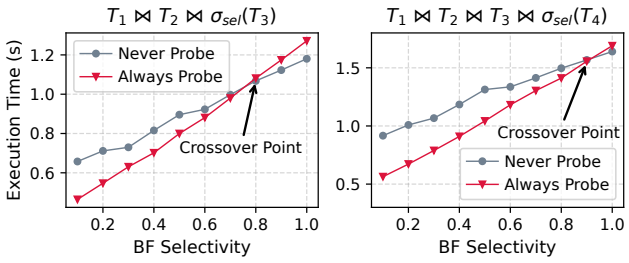


Figure 20: Selective Cascade-Filter Probing – When the BF is selective, probing speeds up queries; when not, it causes overhead. This confirms the need for dynamic probing.

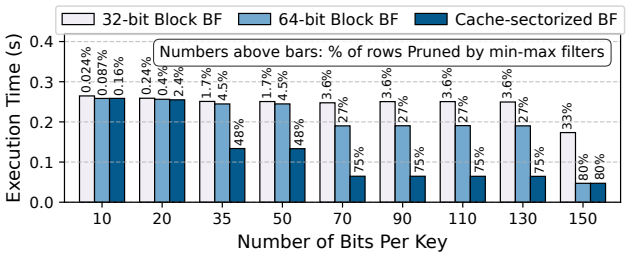


Figure 21: Execution Time vs. Bloom Filter Accuracy - We vary the number of bits per key to show that Min-Max filters are sensitive to the FPR of BFs.

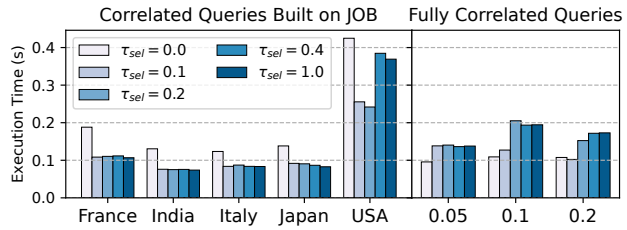


Figure 22: Sensitivity of Dynamic Pipeline to Correlated Queries — We construct two workloads: (1) a query that retrieves films whose production country and actors' birth country are both X, and (2) a fully correlated two-table query, where both tables contain identical data and are constrained by the same predicate.

cannot guarantee full reduction across all tables. Predicate Transfer [38] was the first to transfer predicates globally, but it lacks robustness to join ordering. Robust Predicate Transfer shows that robust join performance is possible even without a cost-based optimizer. However, it incurs many regressions in real-world workloads. Parachute [30] reduces overhead for bidirectional information passing, but requires manual query rewriting and relies on offline statistics. Yannakakis$^+$ [33] identifies semi-joins theoretically redundant in Yannakakis algorithm, but overlooks modern implementations where semi-joins are replaced by BFs.

Besides BFs, OLAP systems use other runtime filters such as min–max filters, hash filters, and bitmaps [1, 7, 25]. These filters store coarse column summaries to quickly discard unlikely probe tuples or irrelevant data blocks. Recent work such as Sieve [32] extends this approach by using learned models to identify row groups relevant to a range filter predicate.

The proposed RPT+ can also be applied to row-store systems. Systems like PostgreSQL [21] support pipelined execution with materialization points, enabling integration of the dynamic pipeline. However, row-stores must scan full tuples even when only a few columns are needed, which limits the pruning effectiveness of predicate transfer. Column-stores are therefore better suited for large analytical joins, where predicate transfer is more effective.

9 CONCLUSION

In this paper, we proposed RPT+, which improves query execution by addressing two limitations of Robust Predicate Transfer (RPT): unnecessary filter transfer and an inefficient filter design. RPT+ uses the asymmetric transfer plan to avoid oversized or redundant filters while preserving full reduction guarantees. Cascade filters strengthen both block-level and tuple-level pruning, and dynamic pipelines detect inefficient predicates at runtime so that filters are created only when beneficial. Compared to baseline DuckDB, RPT+ achieves geometric-mean speedups of 1.47× on JOB, 1.28× on SQL-Storm, 1.17× on TPC-H, and 1.01× on Appian benchmark. Importantly, it avoids the substantial performance regressions observed with the original RPT. These results show that RPT+ improves performance while preserving robustness across diverse workloads.

REFERENCES

[1] Apache Doris Community. 2025. Apache Doris. https://doris.apache.org.

[2] Appian. 2024. Appian Benchmark. https://github.com/duckdb/duckdb/tree/main/benchmark/appian_benchmarks. Accessed: 2025-10-21.

[3] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. *Proceedings of SIGMOD'00* (2000), 261–272.

[4] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proceedings of VLDB'24* 17, 11 (2024), 3215–3228.

[5] Surajit Chaudhuri, Vivek R. Narasayya, and Ravishankar Ramamurthy. 2004. Estimating Progress of Long Running SQL Queries. *Proceedings of SIGMOD'04* (2004), 803–814.

[6] The Transaction Processing Council. 2022. TPC-H Benchmark (Version 3.0.1).

[7] Bailu Ding, Surajit Chaudhuri, and Vivek R. Narasayya. 2020. Bitvector-aware Query Optimization for Decision Support Queries. *Proceedings of SIGMOD'2020* (2020), 2011–2026.

[8] Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altinbüken, Eugene Brevdo, and Campbell Fraser. 2023. Kepler: Robust Learning for Parametric Query Optimization. *Proceedings of SIGMOD'23* 1, 1 (2023), 109:1–109:25.

[9] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science* DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07), Article 10 (Jan 2007).

[10] Xiao Hu. 2025. Output-Optimal Algorithms for Join-Aggregate Queries. *Proceedings of SIGMOD'25* 3, 2 (2025), 104:1–104:27.

[11] Zachary G. Ives and Nicholas E. Taylor. 2008. Sideways Information Passing for Push-Style Query Processing. *Proceedings of ICDE'08* (2008), 774–783.

[12] Srikanth Kandula, Laurel J. Orr, and Surajit Chaudhuri. 2022. Data-induced predicates for sideways information passing in query optimizers. *VLDB J.* 31, 6 (2022), 1263–1290.

[13] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. *Proceedings of SIGMOD'16* (2016), 311–326.

[14] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter A. Boncz. 2019. Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput. *Proceedings of VLDB'19* 12, 5 (2019), 502–515.

[15] Kukjin Lee, Arnd Christian König, Vivek R. Narasayya, Bolin Ding, Surajit Chaudhuri, Brent Ellwein, Alexey Eksarevskiy, Manbeen Kohli, Jacob Wyant, Praneeta Prakash, Rimma V. Nehme, Jiexing Li, and Jeffrey F. Naughton. 2016. Operator and Query Progress Estimation in Microsoft SQL Server Live Query Statistics. *Proceedings of SIGMOD'16* (2016), 1753–1764.

[16] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. *Proceedings of VLDB'24* 17, 7 (2024), 1565–1577.

[17] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of VLDB'15* 9, 3 (2015), 204–215.

[18] David Maier. 1983. *The Theory of Relational Databases.* Computer Science Press.

[19] Riccardo Mancini, Srinivas Karthik, Bikash Chandra, Vasilis Mageirakos, and Anastasia Ailamaki. 2022. Efficient Massively Parallel Join Optimization for Large Queries. *Proceedings of SIGMOD'22* (2022), 122–135.

[20] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proceedings of VLDB'19* 12, 11 (2019), 1705–1718.

[21] PostgreSQL. 2024. https://www.postgresql.org

[22] Felix Putze, Peter Sanders, and Johannes Singler. 2009. Cache-, hash-, and space-efficient bloom filters. *ACM J. Exp. Algorithmics* 14 (2009).

[23] Yiming Qiao, Yihan Gao, and Huanchen Zhang. 2024. Blitzcrank: Fast Semantic Compression for In-memory Online Transaction Processing. *Proceedings of VLDB'2024,* 17, 10 (2024), 2528–2540.

[24] Yiming Qiao and Huanchen Zhang. 2025. Data Chunk Compaction in Vectorized Execution. *Proceedings of SIGMOD'25* 3, 1 (2025), 26:1–26:25.

[25] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. *Proceedings of SIGMOD'19* (2019), 1981–1984.

[26] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proceedings of VLDB'25* 18, 11 (2025), 4144–4157.

[27] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proceedings of VLDB'24* 17, 12 (2024), 3731–3744.

[28] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. *Proceedings of ICDE'19* (2019), 1802–1813.

[29] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 2020. *Database System Concepts, Seventh Edition.* McGraw-Hill Book Company.

[30] Mihail Stoian, Andreas Zimmerer, Skander Krid, Amadou Ngom, Jialin Ding, Tim Kraska, and Andreas Kipf. 2025. Parachute: Single-Pass Bi-Directional Information Passing. *Proceedings of VLDB'25* 18, 10 (2025), 3299–3311.

[31] Apache Arrow Development Team. 2023. Apache Arrow: A cross-language development platform for in-memory data. https://arrow.apache.org/ Accessed: 2025-07-04.

[32] Yulai Tong, Jiazhen Liu, Hua Wang, Ke Zhou, Rongfeng He, Qin Zhang, and Cheng Wang. 2023. Sieve: A Learned Data-Skipping Index for Data Analytics. *Proceedings of VLDB'23* 16, 11 (2023), 3214–3226.

[33] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees. *Proceedings of SIGMOD'25* 3, 3 (2025), 235:1–235:28.

[34] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *Proceedings of VLDB'21* 14, 9 (2021), 1640–1654.

[35] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: A New Cardinality Estimation Framework for Join Queries. *Proceedings of SIGMOD'23* 1, 1 (2023), 41:1–41:27.

[36] Maryann Xue, Yingyi Bu, Abhishek Somani, Wenchen Fan, Ziqi Liu, Steven Chen, Herman Van Hovell, Bart Samwel, Mostafa Mokhtar, Rk Korlapati, Andy Lam, Yunxiao Ma, Vuk Ercegovac, Jiexing Li, Alexander Behm, Yuanjian Li, Xiao Li, Sriram Krishnamurthy, Amit Shukla, Michalis Petropoulos, Sameer Paranjpye, Reynold Xin, and Matei Zaharia. 2024. Adaptive and Robust Query Execution for Lakehouses At Scale. *Proceedings of VLDB'2024* 17, 12 (2024), 3947–3959.

[37] Yifei Yang and Xiangyao Yu. 2025. Accelerate Distributed Joins with Predicate Transfer. *Proceedings of SIGMOD'25* 3, 3 (2025), 122:1–122:27.

[38] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. *Proceedings of CIDR'24* (2024).

[39] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. *Proceedings of VLDB'81* (1981), 82–94.

[40] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection. *Proceedings of VLDB'22* 15, 13 (2022), 3924–3936.

[41] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proceedings of VLDB'23* 17, 2 (2023), 148–161.

[42] Tim Zeyl, Qi Cheng, Reza Pournaghi, Jason Lam, Weicheng Wang, Calvin Wong, Chong Chen, and Per-Åke Larson. 2025. Including Bloom Filters in Bottom-up Optimization. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS, 2025.* 703–715.

[43] Haozhe Zhang, Christoph Mayer, Mahmoud Abo Khamis, Dan Olteanu, and Dan Suciu. 2025. LpBound: Pessimistic Cardinality Estimation Using ℓ_p-Norms of Degree Sequences. *Proceedings of SIGMOD'25* 3, 3 (2025), 184:1–184:27.

[44] Yunjia Zhang, Yannis Chronis, Jignesh M. Patel, and Theodoros Rekatsinas. 2023. Simple Adaptive Query Processing vs. Learned Query Optimizers: Observations and Analysis. *Proceedings of VLDB'23* 16, 11 (2023), 2962–2975.

[45] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. *Proceedings of SIGMOD'25* 3, 3 (2025), 146:1–146:28.

[46] Andreas Zimmerer, Damien Dam, Jan Kossmann, Juliane Waack, Ismail Oukid, and Andreas Kipf. 2025. Pruning in Snowflake: Working Smarter, Not Harder. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS, 2025.* 757–770.