CS 764: Topics in Database Management Systems

Lecture 3: Predicate Transfer

Xiangyao Yu

9/11/2025

Today's Papers: Predicate Transfer

Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries

Yifei Yang, Hangdong Zhao, Xiangyao Yu, Paraschos Koutris
University of Wisconsin-Madison
yyang673@wisc.edu,{hangdong,yxy,paris}@cs.wisc.edu

ABSTRACT

This paper presents *predicate transfer*, a novel method that optimizes join performance by pre-filtering tables to reduce the join input sizes. Predicate transfer generalizes Bloom join, which conducts pre-filtering within a single join operation, to multi-table joins such that the filtering benefits can be significantly increased. Predicate transfer is inspired by the seminal theoretical results by Yannakakis, which uses semi-joins to pre-filter acyclic queries. Predicate transfer generalizes the theoretical results to any join graphs and use Bloom filters to replace semi-joins leading to significant speedup. Evaluation shows predicate transfer can outperform Bloom join by 3.3× on average on TPC-H benchmark.

1 INTRODUCTION

Joins constitute a substantial portion of query execution time, and have been studied and optimized for decades, in topics including binary joins (with a main focus on hash joins) [10, 11, 14, 21], join ordering in multi-way joins [23, 29–31, 34], and recent emerging worst-case optimal join algorithms [16, 26, 35, 36]. One effective principle for enhancing join performance is to minimize the join input sizes by pre-filtering rows that will not appear in the join result. Predicate pushdown [15, 17, 18, 20, 24, 33] exemplifies this principle by applying local predicates on a table before executing any join operation.

The Bloom join [13, 22, 28] extends this principle beyond a single table. In the Bloom join, a Bloom filter is constructed using the join key in one table, and sent to the other table to filter out rows that do not pass the filter—these rows do not match any keys in the first table and will not participate in the join. The Bloom join can effectively reduce the join input sizes thereby reducing the query runtime. However, existing Bloom join solutions can perform such pre-filtering only within a single join operation.

In this paper, we further generalize the pre-filtering principle across multiple joins. Namely, we use predicates on individual tables to pre-filter multiple other tables in the query, further reducing the join input sizes. We call this new technique ***predicate transfer***. A predicate on one table T_1 can be transferred (e.g., in the form of a Bloom filter) to a table T_2 that joins with T_1. T_2 can apply the predicate and further transfer it to table T_3 that joins with T_2 (but T_1 does not necessarily join with T_3). The transfer process can propagate further such that the original predicate can filter multiple other tables (e.g, T_2, T_3, etc.). The conventional Bloom join is a special case of the more generalized predicate transfer—a Bloom join is a one-hop predicate transfer.

 14th Annual Conference on Innovative Data Systems Research (CIDR '24). January 14-17, 2024, Chaminade, USA

The idea of predicate transfer is inspired by the seminal paper [38] by Yannakakis. For an acyclic query that equi-joins multiple tables, the Yannakakis algorithm achieves the theoretically maximum pre-filtering selectivity by adding an additional semi-join phase prior to the actual joins, which filters a table by semi-joining it with other tables. The process filters one table at a time following the tree structure of the query until every predicate is spread across all joining tables.

For all its theoretical elegance, the Yannakakis algorithm has not yet made its way into modern database engines. The main obstacles are the costly hash table accesses and high memory consumption in the semi-join phase. Predicate transfer aims to address these practical limitations. It significantly reduces the overhead of semi-joins by passing succinct data structures like Bloom filters. Although predicate transfer no longer achieves the theoretically maximum filtering selectivity, it achieves much higher performance overall.

In the rest of the paper, we first describe the background and related work of predicate transfer in Section 2, with a focus on the Bloom join and Yannakakis algorithm. We then describe the design space of predicate transfer in detail, and our current heuristics in different design dimensions in Section 3. We report preliminary performance evaluations on TPC-H [1] in Section 4, which shows that on average predicate transfer can outperform Bloom join by 3.3× (up to 61×) and the Yannakakis algorithm by 4.8× (up to 47×) respectively. Finally, Section 5 concludes the paper and discusses future work.

2 BACKGROUND AND RELATED WORK

This section presents the background and related work in Bloom join (Section 2.1) and the Yannakakis algorithm (Section 2.2).

2.1 Bloom join

A Bloom filter [9, 12, 25, 27, 32] is a compact probabilistic data structure that determines whether an element exists in a set. A Bloom filter has no false negative but may have false positives. In a Bloom join of two tables, a Bloom filter is constructed on one table (typically the smaller one) using the join key. The filter is then sent and applied to each row in the other table; if a row does not pass the filter, it matches no row in the first table and should not participate in the join. Since testing a Bloom filter is generally faster than performing a join, Bloom join can speedup query processing, especially when the join is selective. Modern OLAP DBMSs (e.g., Oracle [5], Redshift [6], Snowflake [7], Databricks [8]) widely adopt Bloom filters to accelerate join execution.

Most existing Bloom join algorithms can be applied to only a single join operation. This means the predicate on one table can only be used to pre-filter rows in the other table it joins with; namely, the predicate is transferred in one-hop and one-direction. Some prior work [39] has extended the idea to datasets with star schemas, allowing all dimension tables to transfer local predicates to the fact

CIDR 2024

Debunking the Myth of Join Ordering: Toward Robust SQL Analytics

JUNYI ZHAO, Tsinghua University, China
KAI SU, Tsinghua University, China
YIFEI YANG, University of Wisconsin-Madison, USA
XIANGYAO YU, University of Wisconsin-Madison, USA
PARASCHOS KOUTRIS, University of Wisconsin-Madison, USA
HUANCHEN ZHANG*, Tsinghua University, China

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, TPC-DS, and DSB 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.

CCS Concepts: • **Information systems → Database query processing**.

Additional Key Words and Phrases: Robust query processing, Yannakakis algorithm

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. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 146 (June 2025), 28 pages. https://doi.org/10.1145/3725283

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

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

Authors' Contact Information: Junyi Zhao, Tsinghua University, Beijing, China, zhaojy20@mails.tsinghua.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.

ACM 2836-6573/2025/6-ART146
https://doi.org/10.1145/3725283

SIGMOD 2025

Accelerate Distributed Joins with Predicate Transfer

YIFEI YANG, University of Wisconsin-Madison, USA
XIANGYAO YU, University of Wisconsin-Madison, USA

Join is one of the most critical operators in query processing. One effective way to optimize multi-join performance is to pre-filter rows that do not contribute to the query output. Techniques that reflect this principle include predicate pushdown, semi-join, Yannakakis algorithm, Bloom join, predicate transfer, etc. Among these, predicate transfer is the state-of-the-art pre-filtering technique that removes most non-contributing rows through a series of Bloom filters thereby delivering significant performance improvement.

However, the existing predicate transfer technique has several limitations. First, the current algorithm works only on a single-threaded system while real analytics databases for large workloads are typically distributed across multiple nodes. Second, some predicate transfer steps may not filter out any rows in the destination table thus introduces performance overhead with no speedup. This issue is exacerbated in a distributed environment, where unnecessary predicate transfers lead to extra network latency and traffic. In this paper, we aim to address both limitations. First, we explore the design space of distributed predicate transfer and propose cost-based adaptive execution to maximize the performance for each individual transfer step. Second, we develop a pruning algorithm to effectively remove unnecessary transfers that do not have positive contribution to performance. We implement both techniques and evaluate on a distributed analytics query engine. Results on standard OLAP benchmarks including TPC-H and DSB with a scale factor up to 400 show that distributed predicate transfer can improve the query performance by over 3×, and reduce the amount of data exchange by over 2.7×.

CCS Concepts: • **Information systems → Query optimization**.

Additional Key Words and Phrases: Query optimization, Distributed join processing, Predicate transfer

ACM Reference Format:
Yifei Yang and Xiangyao Yu. 2025. Accelerate Distributed Joins with Predicate Transfer. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 122 (June 2025), 27 pages. https://doi.org/10.1145/3725259

1 Introduction

Join [7, 13, 20, 34] is one of the most critical operators in query processing. One effective principle to optimize multi-join queries is to *pre-filter* rows that do not contribute to the join result prior to actual joins. Optimizations that reflect this principle include predicate pushdown [27, 30, 31, 33, 36, 58], Bloom join [15, 35, 49], Lookahead Information Passing (LIP) [66], semi-join reduction [10], and the Yannakakis algorithm [64]. These algorithms differ in the number of rows that can be pre-filtered and the efficiency of the pre-filtering process. Prominently, the Yannakakis algorithm can pre-filter *all* non-contributing rows for acyclic queries, through a series of semi-join operators across the joining tables.

Recently, predicate transfer [63] was developed as the state-of-the-art protocol following the pre-filtering principle. Predicate transfer replaces semi-joins in the Yannakakis algorithm with Bloom filters, thereby combining the strong theoretical guarantees with high pre-filtering efficiency. Each *transfer* uses a Bloom filter constructed on one table to reduce its neighbor tables. With a

Authors' Contact Information: Yifei Yang, yyang673@wisc.edu, University of Wisconsin-Madison, Madison, USA; Xiangyao Yu, yxy@cs.wisc.edu, University of Wisconsin-Madison, Madison, USA.

ACM 2836-6573/2025/6-ART122
https://doi.org/10.1145/3725259

SIGMOD 2025

Outline

- **Power of pre-filtering**
- Yannakakis algorithm
- Predicate transfer
- Robust predicate transfer
- Distributed predicate transfer
- Discussion and future work

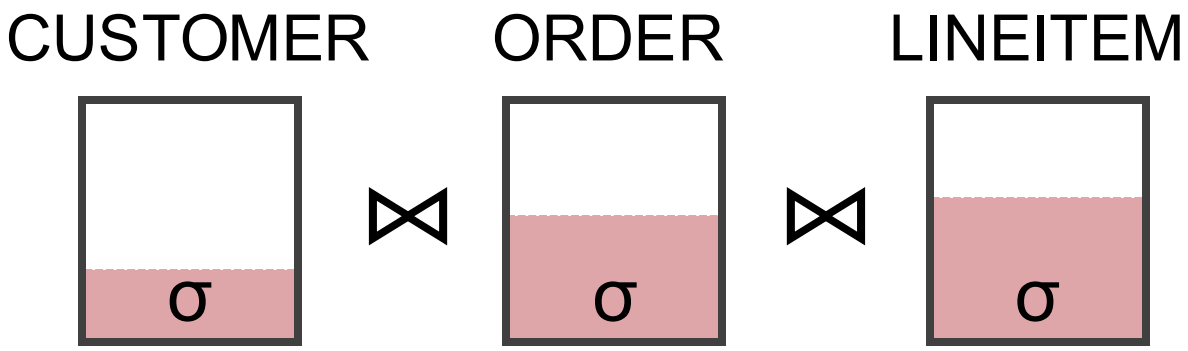
Potential of Pre-Filtering

In multi-join queries, **many rows do not contribute to query result**

Potential of Pre-Filtering

In multi-join queries, **many rows do not contribute to query result**

Example: TPC-H Q3



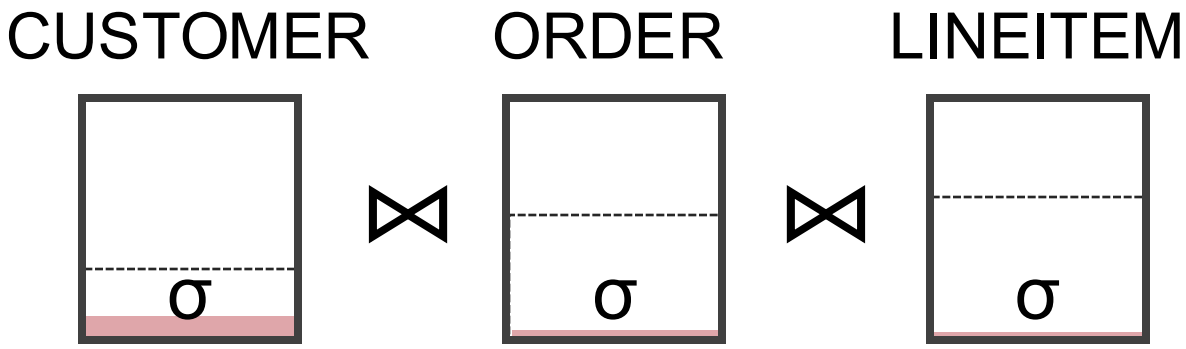
Rows participating in joins after local filtering

20% **49%** **53%**

Potential of Pre-Filtering

In multi-join queries, **many rows do not contribute to query result**

Example: TPC-H Q3



Rows participating in joins after local filtering

20%	**49%**	**53%**

Rows contributing to query results

5.6%	**<1%**	**<0.5%**

4–100× size reduction!

Most rows are filtered during the joins

Potential of Pre-Filtering

In multi-join queries, **many rows do not contribute to query result**

Goal: Identify non-contributing rows and pre-filter them efficiently before executing joins

Rows participating in joins after local filtering	**20%**	**49%**	**53%**
Rows contributing to query results	**5.6%**	**<1%**	**<0.5%**

4–100× size reduction!

Most rows are filtered during the joins

Size Reduction Factor

Orders-of-magnitude reduction of join table size (total # of rows in all joining tables)

TPC-H Query	Optimal pre-filter
Q2	559
Q3	80
Q4	20
Q5	56
Q7	122
Q8	669
Q9	12
Q10	8
Q11	25
Q12	25
Q13	1
Q14	2
Q15	1
Q16	6
Q17	931
Q18	16827
Q19	232
Q20	1489
Q21	23
Q22	2
Geomean	**44**

Size Reduction Factor

Orders-of-magnitude reduction of join table size (total # of rows in all joining tables)

TPC-H Query	Optimal pre-filter
Q2	559
Q3	80
Q4	20
Q5	56
Q7	122
Q8	669
Q9	12
Q10	8
Q11	25
Q12	25
Q13	1
Q14	2
Q15	1
Q16	6
Q17	931
Q18	16827
Q19	232
Q20	1489
Q21	23
Q22	2
Geomean	**44**

Size Reduction Factor

Orders-of-magnitude reduction of join table size (total # of rows in all joining tables)

TPC-H Query	Optimal pre-filter
Q2	559
Q3	80
Q4	20
Q5	56
Q7	122
Q8	669
Q9	12
Q10	8
Q11	25
Q12	25
Q13	1
Q14	2
Q15	1
Q16	6
Q17	931
Q18	16827
Q19	232
Q20	1489
Q21	23
Q22	2
Geomean	**44**

Size Reduction Factor

Orders-of-magnitude reduction of join table size (total # of rows in all joining tables)

TPC-H Query	Optimal pre-filter
Q2	559
Q3	80
Q4	20
Q5	56
Q7	122
Q8	669
Q9	12
Q10	8
Q11	25
Q12	25
Q13	1
Q14	2
Q15	1
Q16	6
Q17	931
Q18	16827
Q19	232
Q20	1489
Q21	23
Q22	2
Geomean	**44**

Size Reduction Factor

Orders-of-magnitude reduction of join table size (total # of rows in all joining tables)

As prior work, Bloom join can pre-filter only a small fraction of non-contributing rows

TPC-H Query	Optimal pre-filter	Bloom join
Q2	559	2
Q3	80	1
Q4	20	17
Q5	56	1
Q7	122	9
Q8	669	22
Q9	12	1
Q10	8	1
Q11	25	15
Q12	25	22
Q13	1	1
Q14	2	1
Q15	1	1
Q16	6	5
Q17	931	1
Q18	16827	1
Q19	232	59
Q20	1489	2
Q21	23	1
Q22	2	3
Geomean	**44**	**3**

Size Reduction Factor

Orders-of-magnitude reduction of join table size (total # of rows in all joining tables)

As prior work, Bloom join can pre-filter only a small fraction of non-contributing rows

Research Question: How to efficiently identify non-contributing rows?

TPC-H Query	Optimal pre-filter	Bloom join
Q2	559	2
Q3	80	1
Q4	20	17
Q5	56	1
Q7	122	9
Q8	669	22
Q9	12	1
Q10	8	1
Q11	25	15
Q12	25	22
Q13	1	1
Q14	2	1
Q15	1	1
Q16	6	5
Q17	931	1
Q18	16827	1
Q19	232	59
Q20	1489	2
Q21	23	1
Q22	2	3
Geomean	**44**	**3**

Size Reduction Factor

Orders-of-magnitude reduction of join table size (total # of rows in all joining tables)

As prior work, Bloom join can pre-filter only a small fraction of non-contributing rows

Research Question: How to efficiently identify non-contributing rows?

Answer (Partial): Yannakakis Algorithm

TPC-H Query	Optimal pre-filter	Bloom join
Q2	559	2
Q3	80	1
Q4	20	17
Q5	56	1
Q7	122	9
Q8	669	22
Q9	12	1
Q10	8	1
Q11	25	15
Q12	25	22
Q13	1	1
Q14	2	1
Q15	1	1
Q16	6	5
Q17	931	1
Q18	16827	1
Q19	232	59
Q20	1489	2
Q21	23	1
Q22	2	3
Geomean	**44**	**3**

Outline

- Power of pre-filtering
- **Yannakakis algorithm**
- Predicate transfer
- Robust predicate transfer
- Distributed predicate transfer
- Discussion and future work

Yannakakis Algorithm

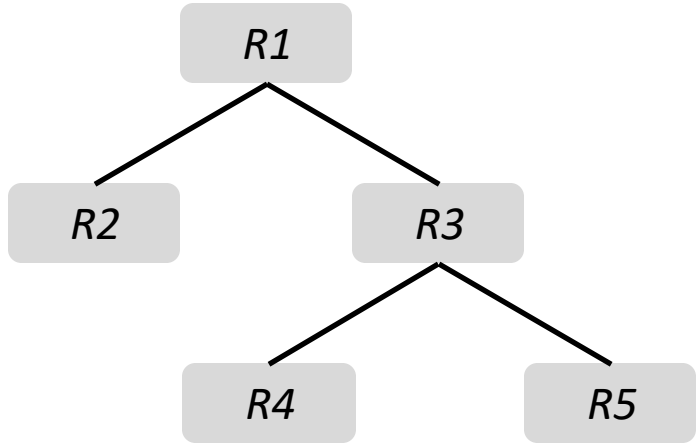
Pre-filter all non-contributing rows for acyclic queries

[1] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. VLDB 1989

Yannakakis Algorithm

Pre-filter all non-contributing rows for acyclic queries

- The join graph forms a tree



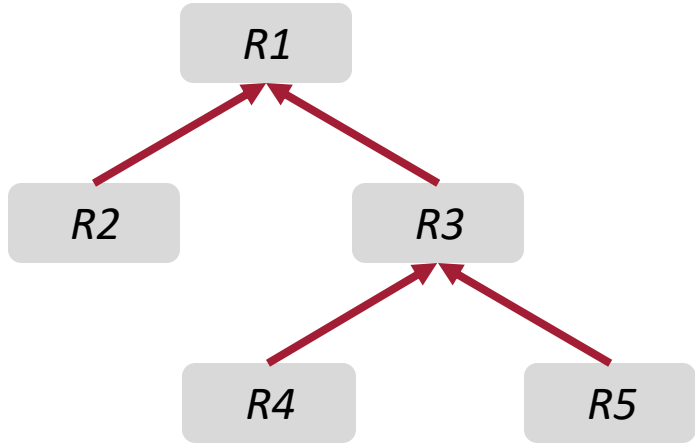
[1] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. VLDB 1989

Yannakakis Algorithm

Pre-filter all non-contributing rows for acyclic queries

- The join graph forms a tree
- **Forward pass**: Semi-join each table with its children

$$R3 \leftarrow R3 \ltimes R4$$
$$R3 \leftarrow R3 \ltimes R5$$
$$R1 \leftarrow R1 \ltimes R2$$
$$R1 \leftarrow R1 \ltimes R3$$



[1] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. VLDB 1989

Yannakakis Algorithm

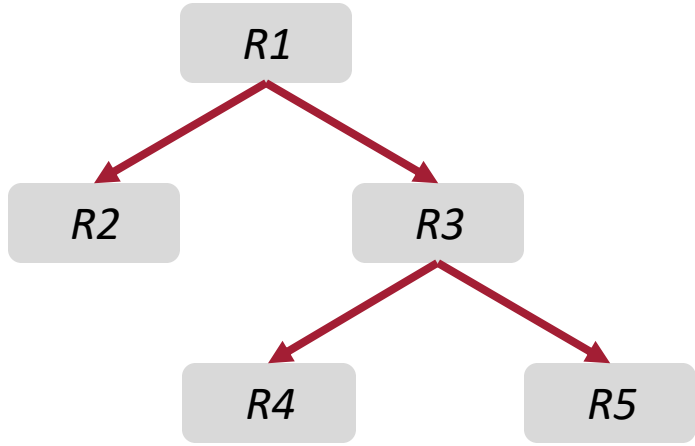
Pre-filter all non-contributing rows for acyclic queries

- The join graph forms a tree
- **Forward pass**: Semi-join each table with its children

$$R3 \leftarrow R3 \ltimes R4$$
$$R3 \leftarrow R3 \ltimes R5$$
$$R1 \leftarrow R1 \ltimes R2$$
$$R1 \leftarrow R1 \ltimes R3$$

- **Backward pass**: Semi-join each table with its parent

$$R2 \leftarrow R2 \ltimes R1$$
$$R3 \leftarrow R3 \ltimes R1$$
$$R4 \leftarrow R4 \ltimes R3$$
$$R5 \leftarrow R5 \ltimes R3$$



[1] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. VLDB 1989

Yannakakis Algorithm

Pre-filter all non-contributing rows for acyclic queries

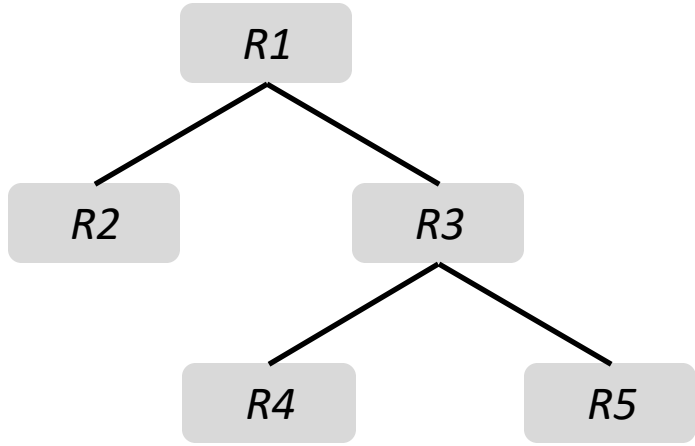
- The join graph forms a tree
- **Forward pass**: Semi-join each table with its children

$$R3 \leftarrow R3 \ltimes R4$$
$$R3 \leftarrow R3 \ltimes R5$$
$$R1 \leftarrow R1 \ltimes R2$$
$$R1 \leftarrow R1 \ltimes R3$$

- **Backward pass**: Semi-join each table with its parent

$$R2 \leftarrow R2 \ltimes R1$$
$$R3 \leftarrow R3 \ltimes R1$$
$$R4 \leftarrow R4 \ltimes R3$$
$$R5 \leftarrow R5 \ltimes R3$$

- Perform regular joins on the reduced tables

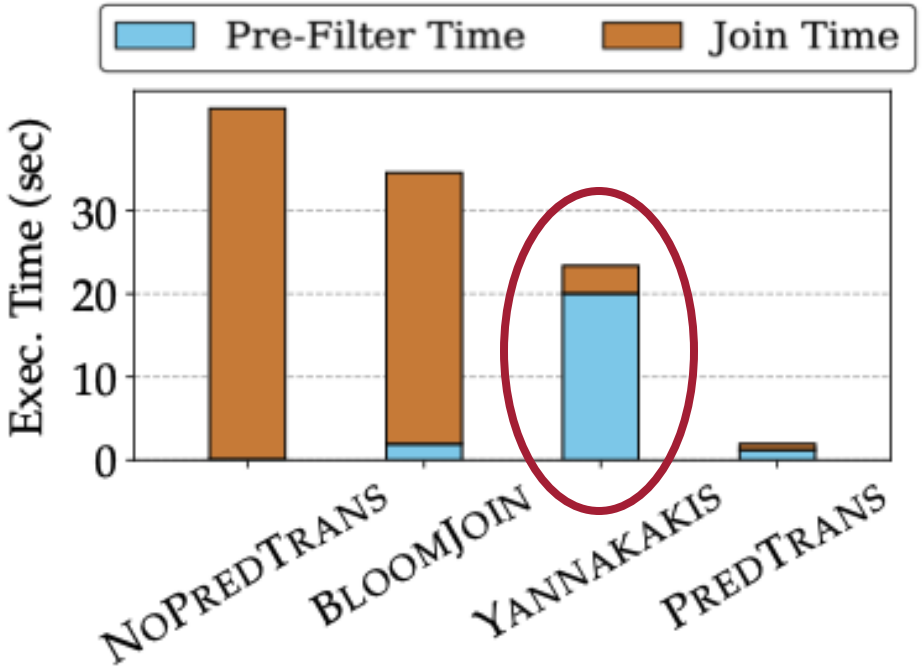


[1] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. VLDB 1989

Yannakakis Algorithm – Limitation

Yannakakis algorithm can pre-filter **all non-contributing rows** for acyclic queries

But semi-joins are expensive!



[1] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. VLDB 1989

Outline

- Power of pre-filtering
- Yannakakis algorithm
- **Predicate transfer**
- Robust predicate transfer
- Distributed predicate transfer
- Discussion and future work

Predicate Transfer

Key Idea: Replace semi-joins with Bloom filters

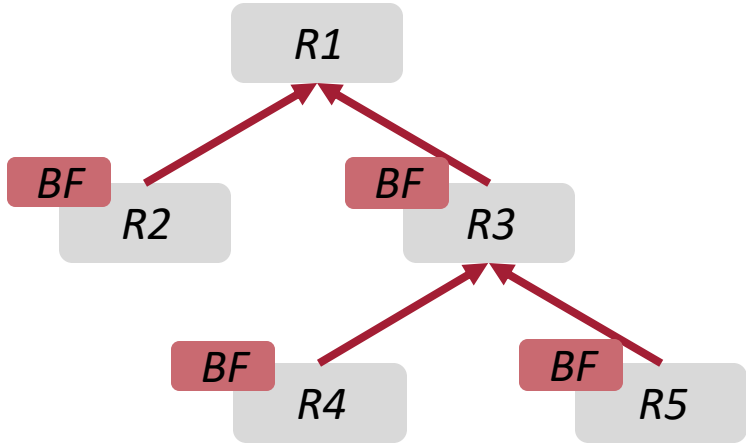
- Bloom filters are much faster than semi-joins
- Bloom filters have false positives, which is ok since false positives will be filtered by the join phase

Predicate Transfer

Key Idea: Replace semi-joins with Bloom filters

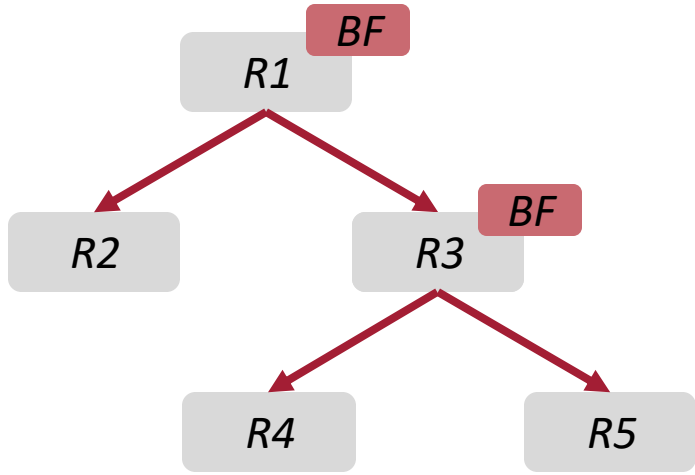
Forward pass

- Create bloom filter on join key
- Use bloom filter to reduce parent table



Predicate Transfer

Key Idea: Replace semi-joins with Bloom filters



Forward pass

- Create bloom filter on join key
- Use bloom filter to reduce parent table

Backward pass

- Create bloom filter on join key
- Use bloom filter to reduce child table

Predicate Transfer

Key Idea: Replace semi-joins with Bloom filters

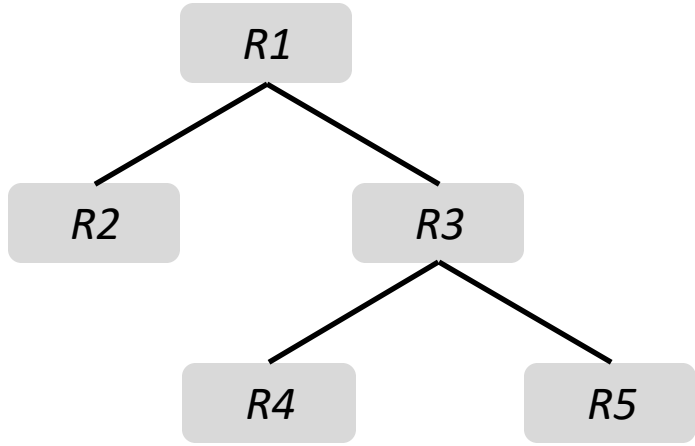
Forward pass

- Create bloom filter on join key
- Use bloom filter to reduce parent table

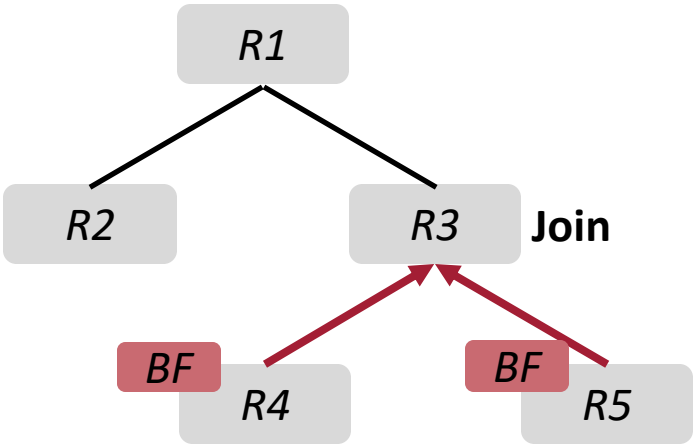
Backward pass

- Create bloom filter on join key
- Use bloom filter to reduce child table

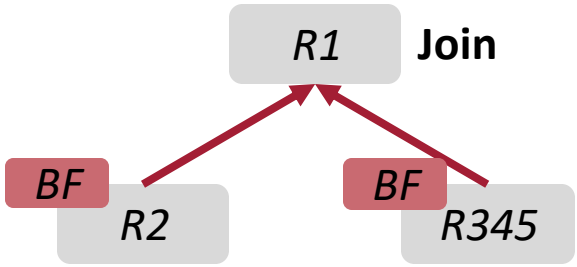
Perform regular joins on the reduced tables



Sideway Information Passing (SIP)



Step 1



Step 2

Sideway information passing (SIP)

– transfer, join, transfer, join, transfer, join, …

Predicate Transfer (PT)

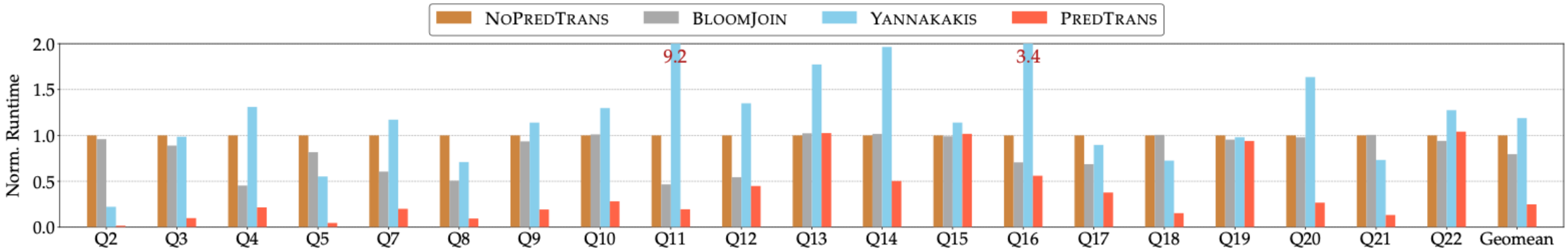
– transfer, transfer, transfer, …, join, join, join, …

Generalization of Predicate Transfer

- Any join graph topology (even cyclic queries)
- Any transfer schedule (our heuristic: from small to large tables)
- Transfer path pruning
- Any filter type
- Support outer-join, aggregation, and UDF
- Multiple transfer graphs

Evaluation

10 GB TPC-H, Single-threaded execution on FlexPushdownDB



Predicate transfer outperforms Bloom Join by **3.3×**

Predicate transfer reduces join table size by **19×** (optimal pre-filter is 44× and Bloom join is 3×)

Yannakakis algorithm underperforms due to semi-join overhead

Predicate Transfer is Robust

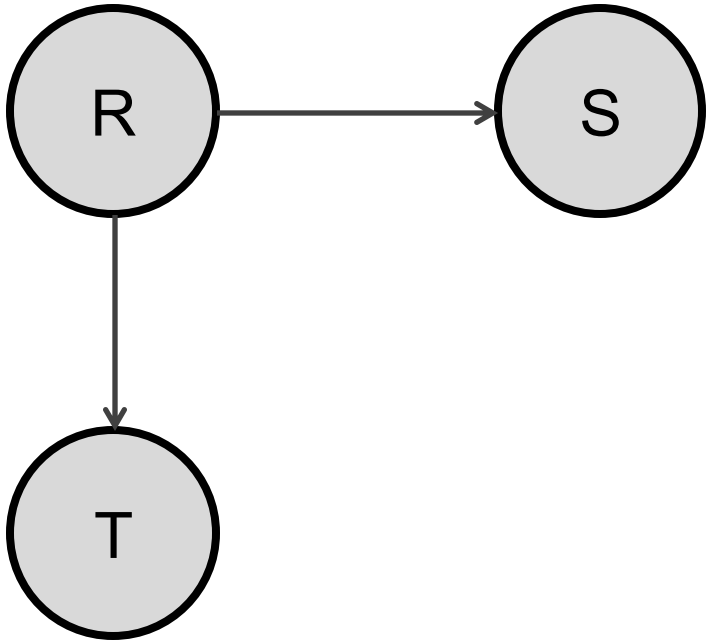
Performance of predicate transfer is **insensitive to join order**

Outline

- Power of pre-filtering
- Yannakakis algorithm
- Predicate transfer
- **Robust predicate transfer**
- Distributed predicate transfer
- Discussion and future work

Transfer Schedule

Default transfer heuristic in PT: **from small to large tables (small2large)**



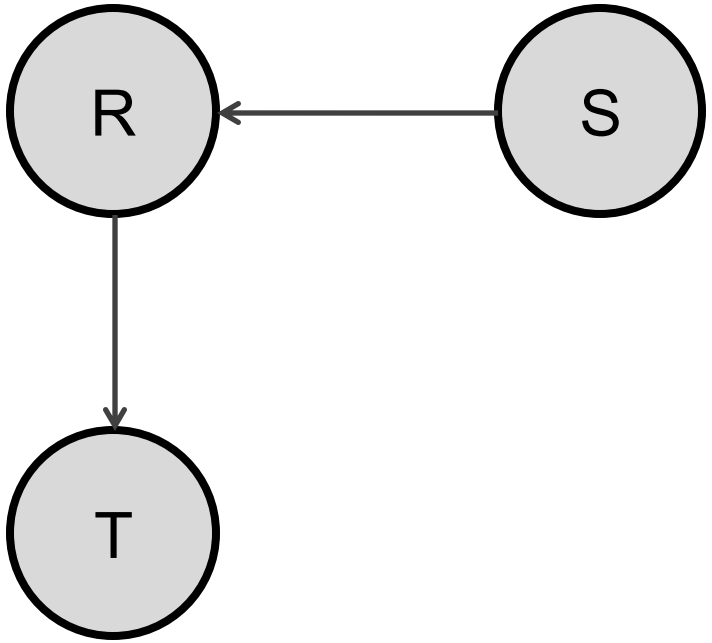
Assume |R| < |S| < |T|

Transfer does not follow a tree

No Yannakakis guarantee!

Transfer Schedule

New transfer heuristic in Robust PT: **pick largest table as root (LargestRoot)**

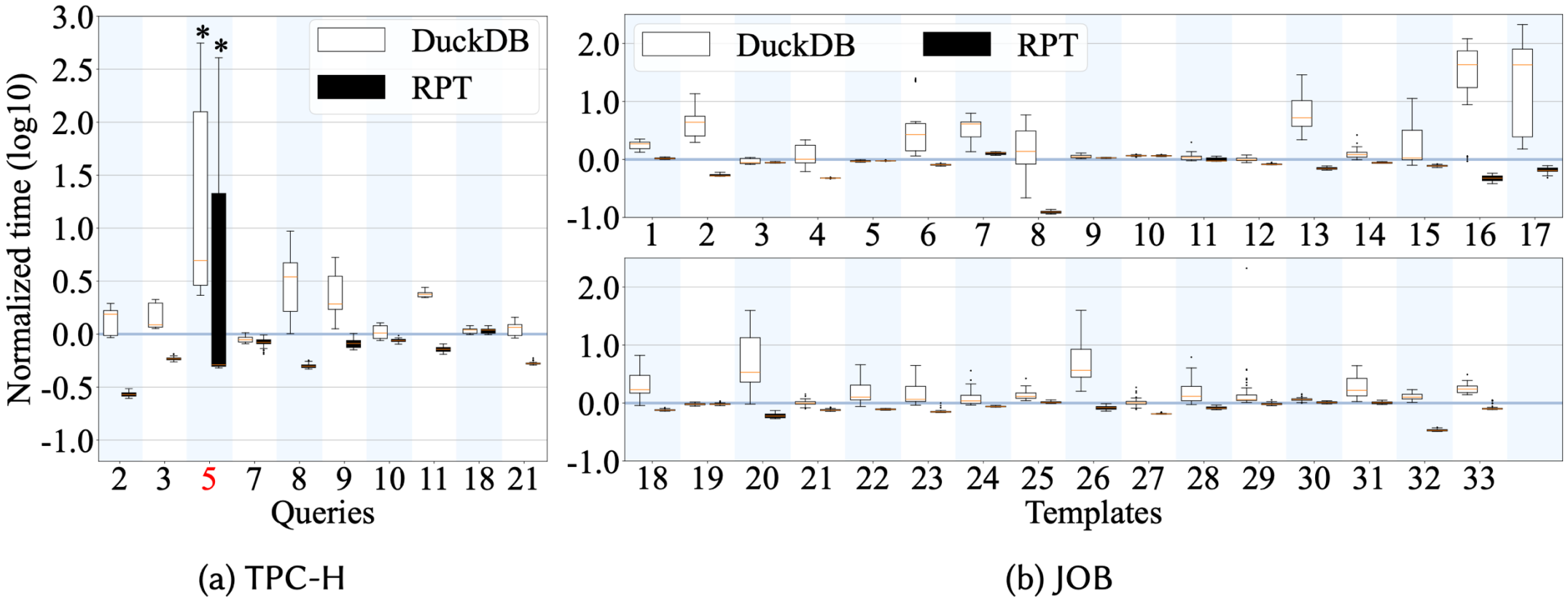


Assume |R| < |S| < |T|

Ensure the transfer graph is always a tree

RPT is more robust than PT

Robustness to Join Ordering



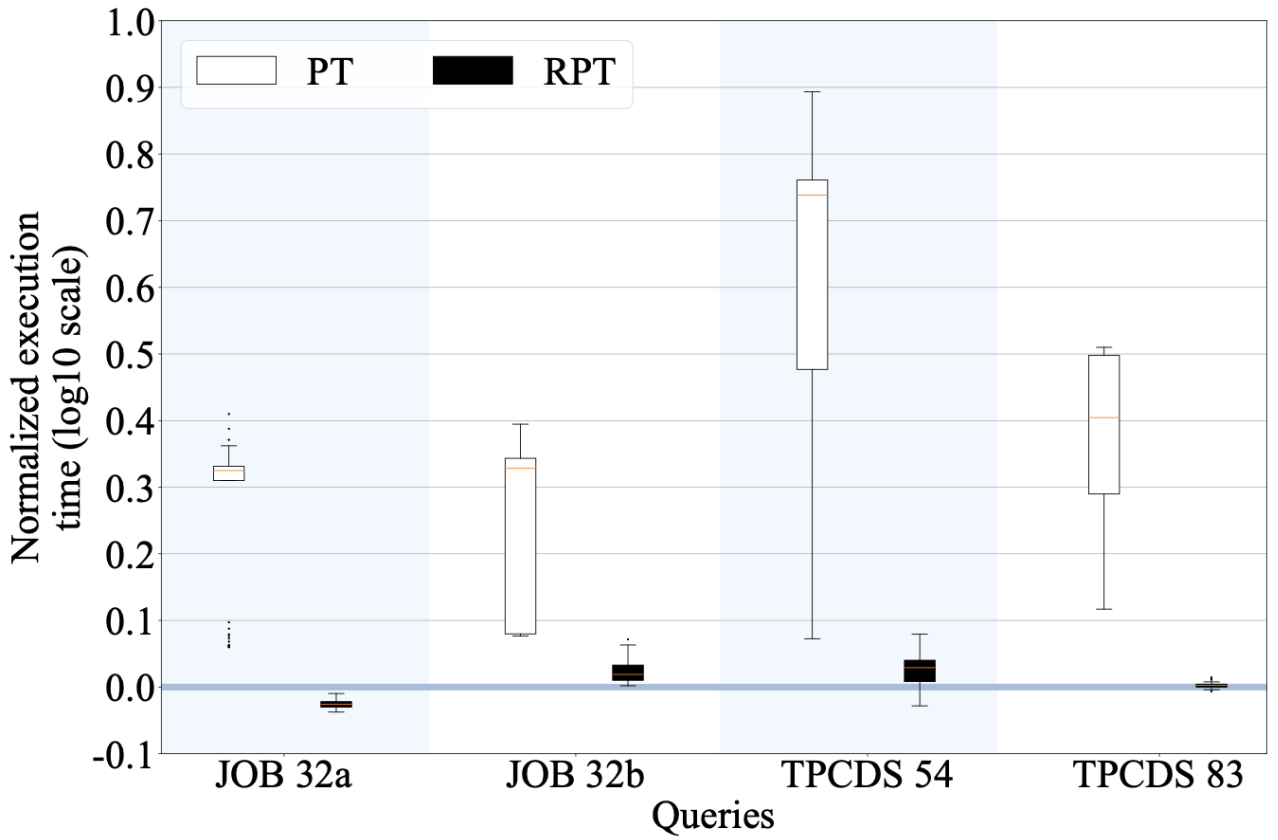
(a) TPC-H (b) JOB

DuckDB + RPT is **44–56%** faster than DuckDB

RPT queries are orders-of-magnitude more **robust to join ordering**

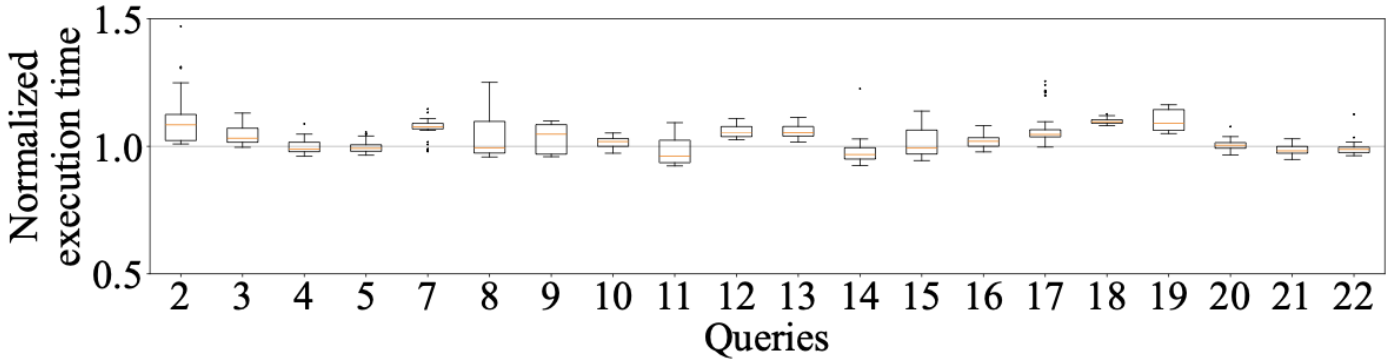
– Randomly generated join orders

PT vs. RPT

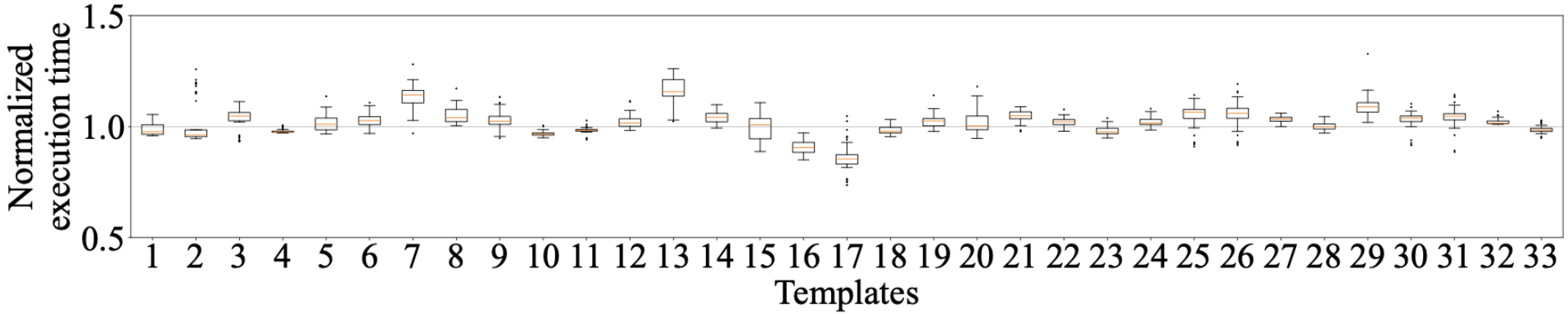


For some queries, *LargestRoot* is more robust than *Small2large*

Robustness to Transfer Schedule



(a) TPC-H



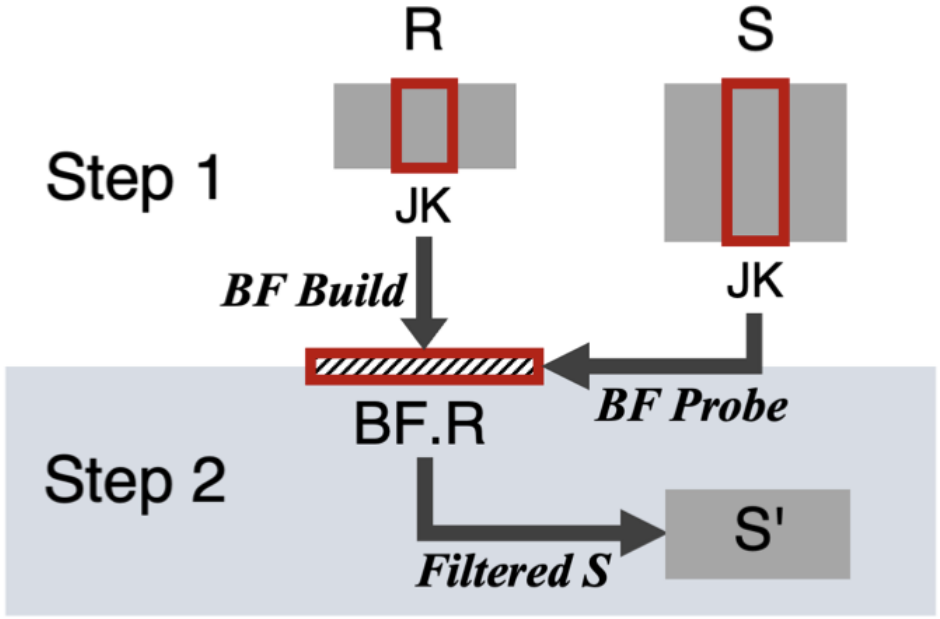
(b) JOB

Performance robust to transfer schedule under *LargestRoot* heuristics

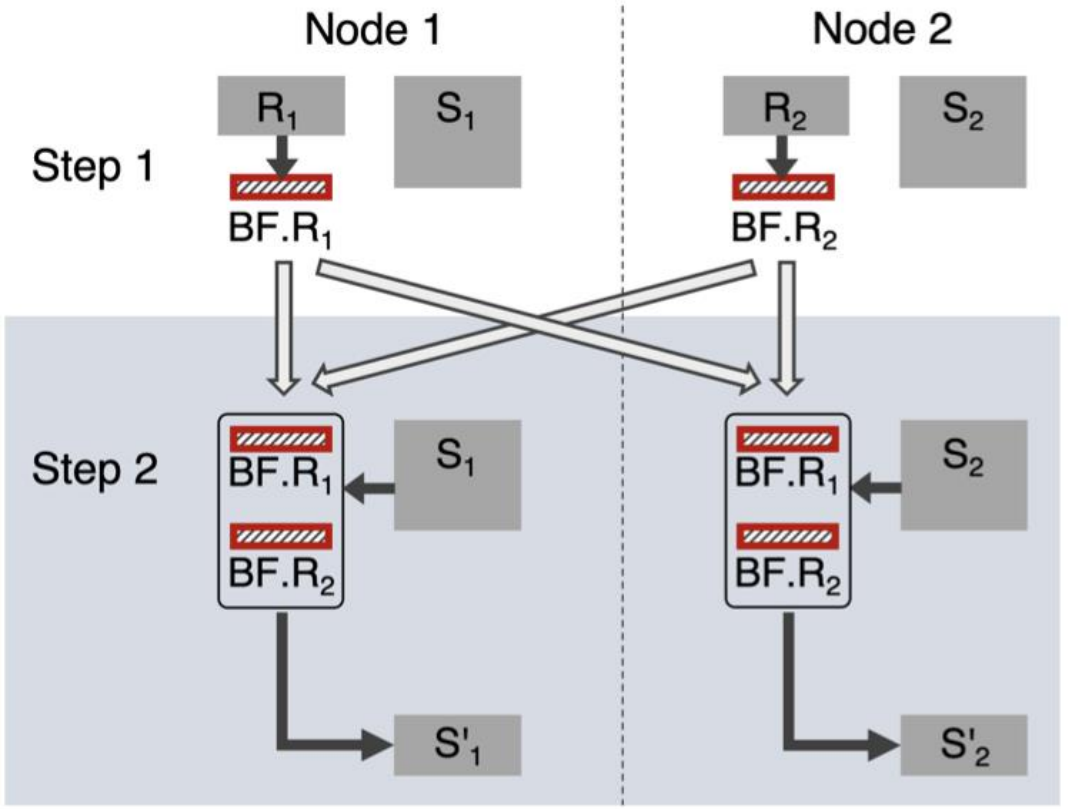
Outline

- Power of pre-filtering
- Yannakakis algorithm
- Predicate transfer
- Robust predicate transfer
- **Distributed predicate transfer**
- Discussion and future work

Distribute Each Transfer Step



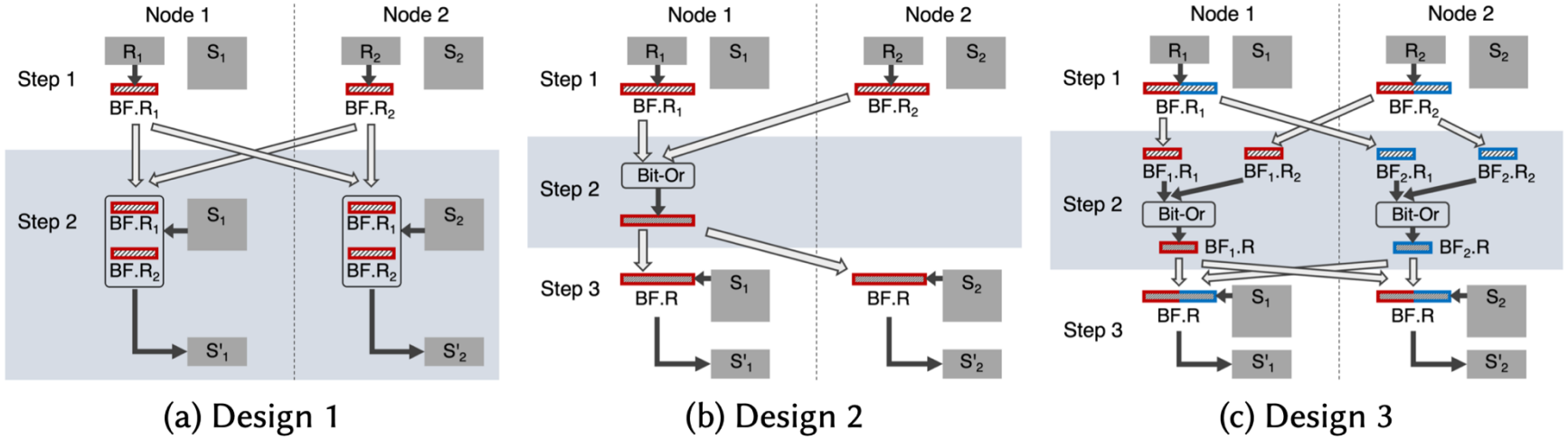
(a) Predicate Transfer Step

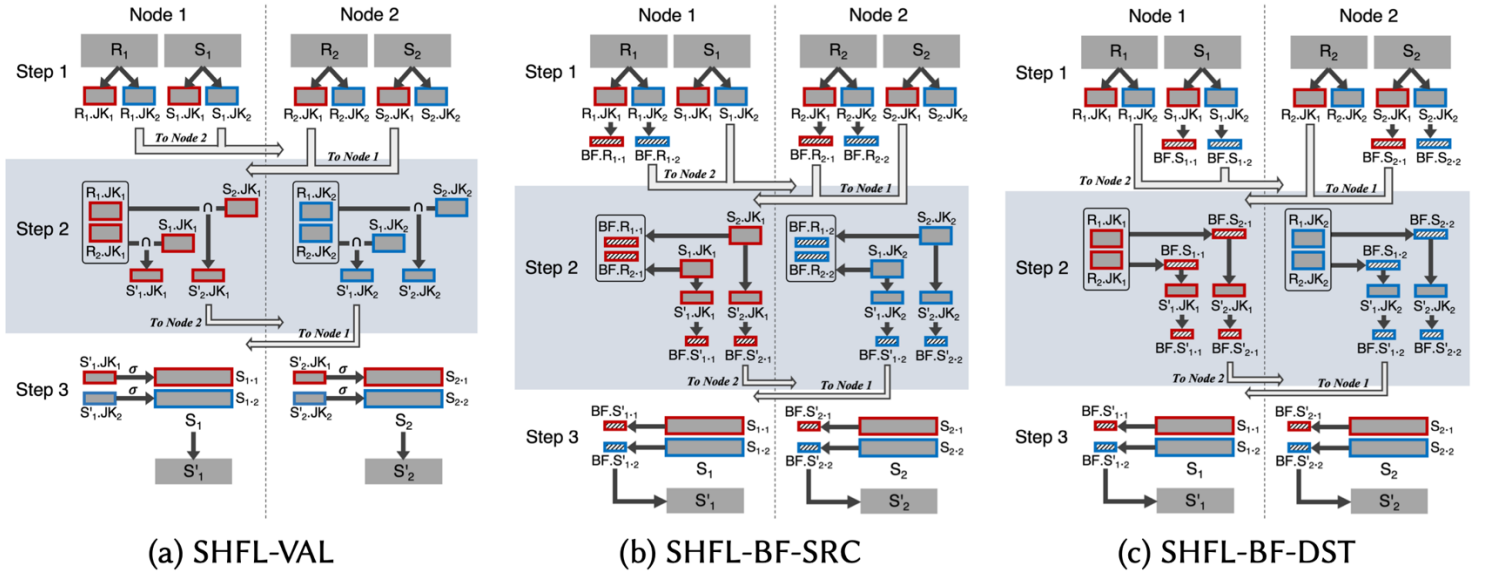


(a) Design 1

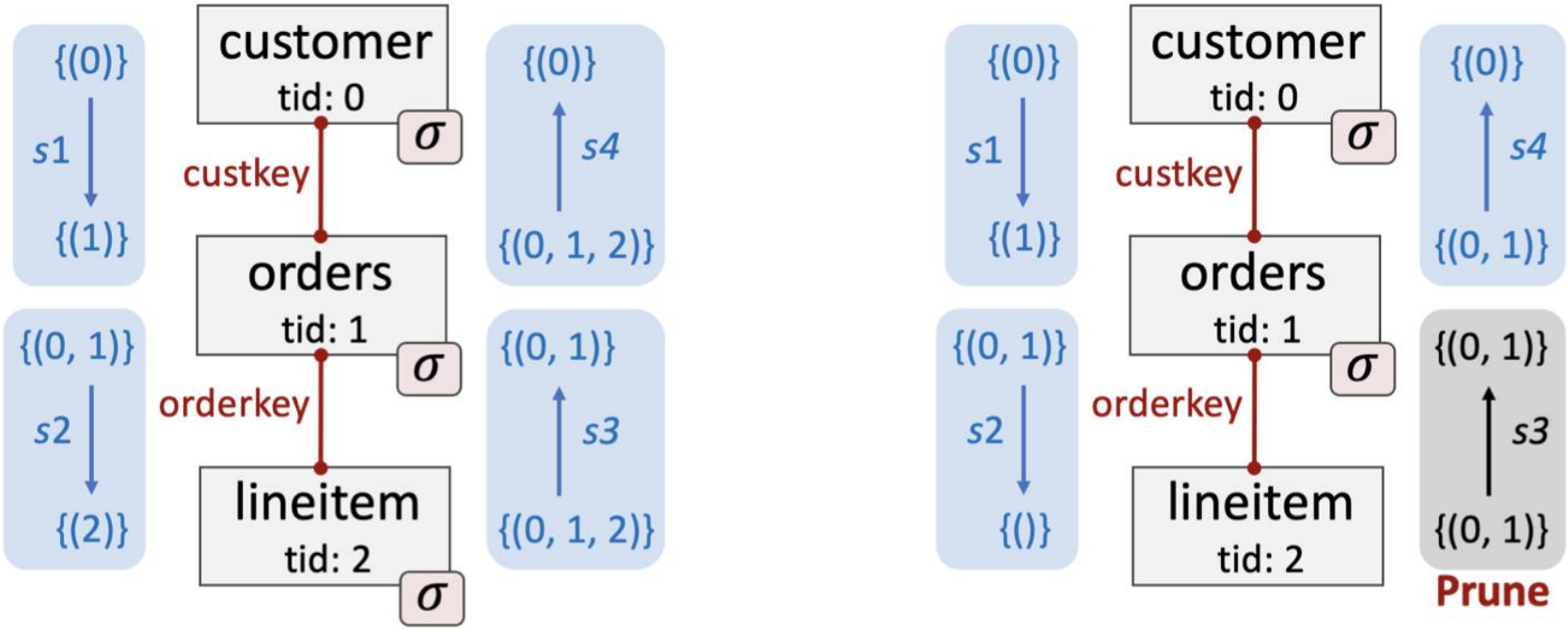
Each node builds local BF, and broadcasts it to other nodes

Distribute Each Transfer Step





Prune Transfer Steps

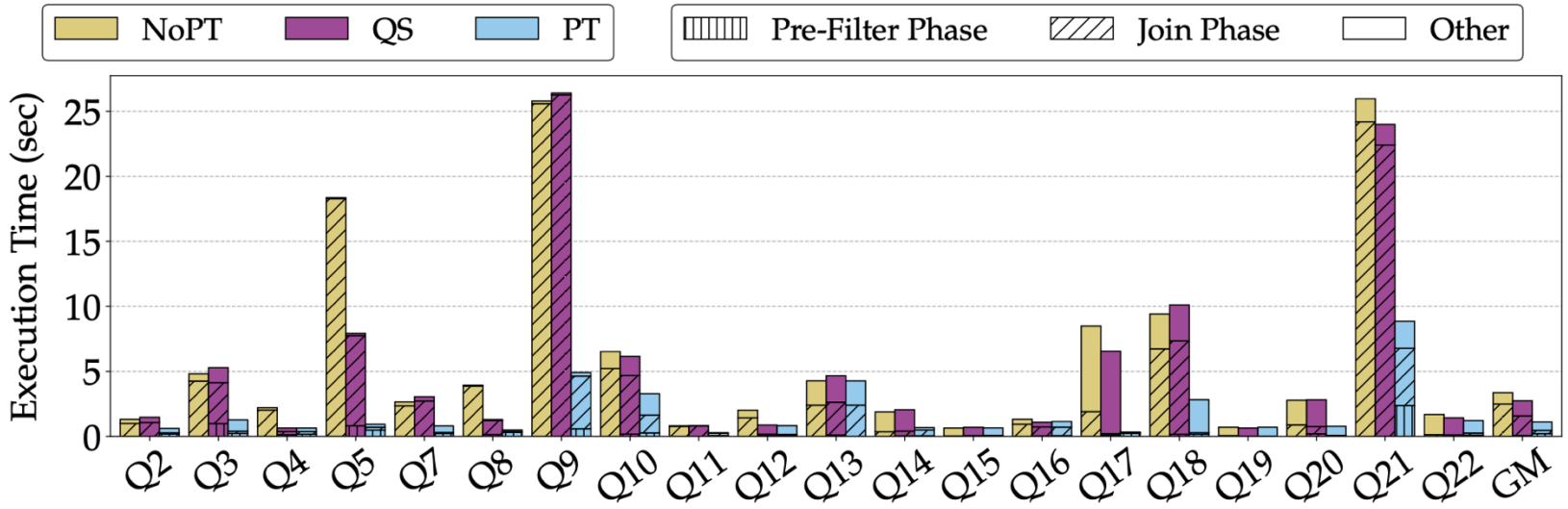


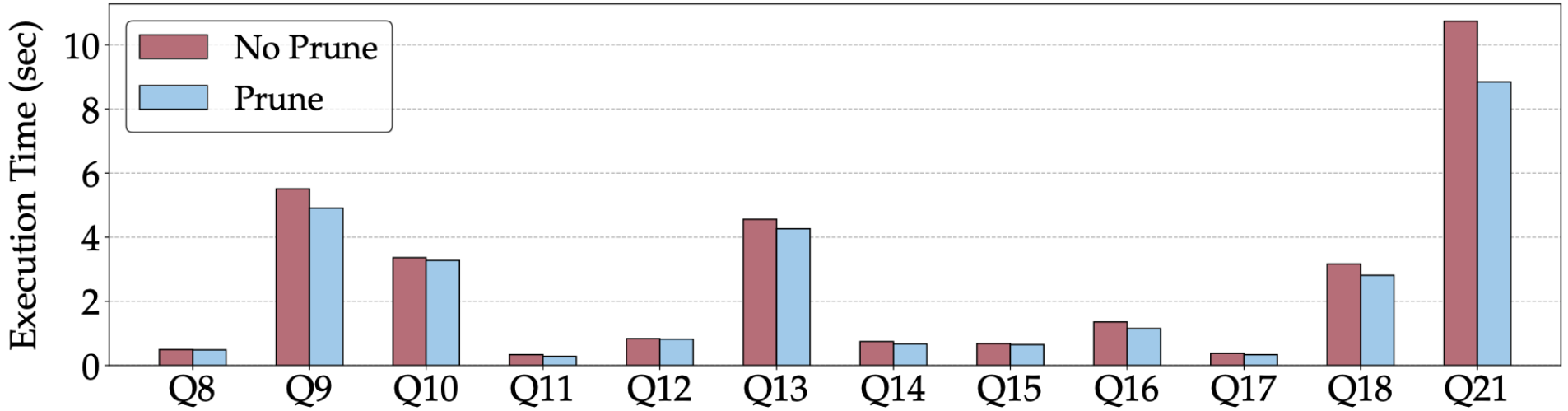
(a) TPC-H Q3 (b) Variant of TPC-H Q3

Fig. 5. Pruning for Predicate Transfer in TPC-H Q3 and its variant.

Ineffective transfer steps can be pruned for better performance

Evaluation


NoPT
QS
PT
Pre-Filter Phase
Join Phase
Other
Execution Time (sec)
0
5
10
15
20
25
Q2
Q3
Q4
Q5
Q7
Q8
Q9
Q10
Q11
Q12
Q13
Q14
Q15
Q16
Q17
Q18
Q19
Q20
Q21
Q22
GM


No Prune
Prune
Execution Time (sec)
0
2
4
6
8
10
Q8
Q9
Q10
Q11
Q12
Q13
Q14
Q15
Q16
Q17
Q18
Q21

Outline

- Power of pre-filtering
- Yannakakis algorithm
- Predicate transfer
- Robust predicate transfer
- Distributed predicate transfer
- **Discussion and future work**

Questions

- Handle recursive or cyclic join graphs?
- Alternatives to Bloom filters?
- What if BF creates too many false positives?
- What if table size estimations are inaccurate?
- How to integrate PT into an existing DB?

Project Ideas—Predicate Transfer

- Enhance PT performance for **cyclic queries**
- Evaluate PT in latest DuckDB version
- Evaluate PT for more workloads
- Dynamically tune Bloom filter size
- More advanced pruning techniques
- Study PT's effect for reducing intermediate data size
- Study PT with workloads that do not fit in memory

Before Next Lecture

Submit review for

Viktor Leis, et al., How Good Are Query Optimizers, Really?. VLDB, 2015