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

table, which outperformed the baseline Bloom join. However, these solutions do not generalize to more complex query plans.

2.2 Yannakakis algorithm

The Yannakakis algorithm [38] is a classic algorithm that can pre-filter out all rows from tables that do not appear in the final join result, thereby achieving the theoretically maximum filtering selectivity. The algorithm applies to *acyclic* join queries. The acyclicity is more formally termed as *α-acyclicity* [38]. The algorithm is proven to run in $O(N + \text{OUT})$ time, where N is the size of input relations and OUT is the query output size. Thus, the Yannakakis algorithm is known to be *instance optimal* since $N+\text{OUT}$ is the unavoidable time cost of reading the input and enumerating the output for a query. The algorithm starts by choosing a rooted join tree arbitrarily, and then proceeds with a *semi-join phase* and a *join phase*.

Semi-join phase. The semi-join phase contains two passes: the *forward pass* and the *backward pass*. The forward pass traverses the join tree in a bottom-up fashion. At each vertex, we filter the table by a sequence of semi-joins with its children. A *semi-join* of two tables R and S is defined as $R \ltimes S = \Pi_{\text{attr}(R)}(R \bowtie S)$, which effectively removes all tuples in R that do not join with any tuple in S. The forward pass stops when the root node is reached. Similarly, the backward pass traverses the join tree in a top-down fashion. At each vertex, the table is filtered by a semi-join with its parent. The backward pass stops when all leaf nodes are reached. It is proven that both passes can be executed in $O(N)$ time and all tuples that will not contribute to the output are removed.

Join phase. The join phase can join the filtered tables in any order. It is proven that regardless of the chosen join order, the join phase can be executed in $O(\text{OUT})$ time.

As a reflection, the semi-join phase filters all redundant tuples and the join phase executes the join with automatic robustness: it can join the tables in any order without any intermediate table size blow-up over the output size. The algorithm was later extended by Joglekar et al. [19] to handle aggregations on top of join queries.

3 PREDICATE TRANSFER

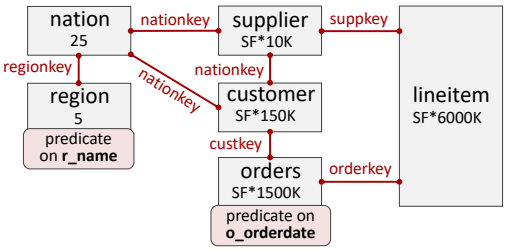
This section describes the proposed predicate transfer algorithm. We use Query 5 in TPC-H benchmark [1] (Figure 1) as a running example. This query contains six tables, six inner joins, and two predicates on tables `region` and `orders` respectively. The discussion assumes equi-join between tables.

3.1 Overview

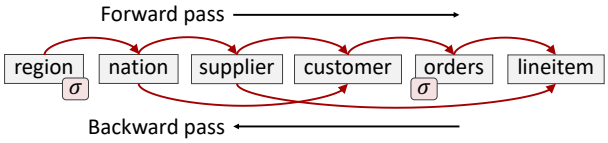
Similar to the Yannakakis algorithm, predicate transfer executes a query in two phases.

Phase 1: Predicate Transfer Phase. A join graph is constructed for a query, where each vertex is a table and each edge is a join operation. A local predicate is constructed as a filter (e.g., a Bloom filter) and be transferred across the join graph. The schedule of the predicate transfer phase introduces a large design space, which we discuss in Section 3.2.

Phase 2: Join Phase. After the transfer phase finishes, each table has multiple filters, including both local filters and transferred filters. The database can now apply the filters and perform regular joins. The actual inputs of each join will be substantially smaller if the transferred filters are selective. We discuss the join phase in Section 3.3.



(a) Join Graph



(b) Predicate Transfer Graph

Figure 1: Predicate Transfer for TPC-H Q5.

In the next two subsections, we will describe the design space of these two phases and the heuristics we currently use to implement predicate transfer in our prototype. These heuristics are largely intuition-based and a more thorough theoretical analysis is left for future work.

3.2 Predicate Transfer Phase

In the rest of this section, we layout the design space of the transfer phase and describe the design choices we adopt in our prototype.

Filter Transformation. When transferring a filter across edges that have different join keys, the filter must be transformed. For example, a filter constructed on `region` can be transferred to `nation`, but the same filter cannot be directly sent to `supplier` from `nation` in the next transfer hop since the join keys do not match. We use the following algorithm to handle the join key mismatch between incoming and outgoing edges on `nation`. When the incoming filter is received, an empty outgoing filter is created. Then, the columns for both incoming and outgoing join keys in `nation` are scanned (assuming columnar store; otherwise scan the entire table). Inherent filters of `nation` are applied during the scan. Then for each row, the incoming join key is used to probe the incoming filter. If a match occurs, the corresponding outgoing join key is added to the outgoing filter. At the end of the scan, the outgoing filter is sent to downstream tables (i.e., `supplier`). The algorithm is efficient as it requires scanning the join keys only once.

Figure 2 shows an example of filter transformation on table R. Table R has three columns, and participates into three different joins on columns A, B, and C respectively. In the predicate transfer phase, R receives two incoming filters (assume we are using Bloom filters) on join attributes A and B respectively. Column A is used to probe the incoming filter on join attribute A, where all rows except the

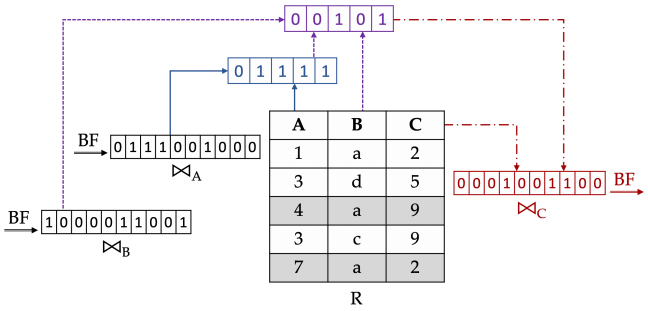


Figure 2: Filter Transformation — Table R receives two incoming filters on join attributes A and B, and generates a transformed outgoing filter on join attribute C.

first satisfy the join condition (with false positives). Then column B on the remaining rows is used to probe the incoming filter on join attribute B, with another two rows (row 2 and 4) filtered out. Finally, column C on the remaining two rows (row 3 and 5) is used to build the outgoing filter.

Predicate Transfer Graph. The join graph determines the topology of predicate transfer. Figure 1a shows the join graph for Query 5 in TPC-H. Each equi-join is represented as an edge. A *predicate transfer graph* is a directed subgraph of the join graph. Transfers happen along the selected edges in the subgraph—local predicates of the source vertex are transferred to the target vertex as a filter. Figure 1b shows one predicate transfer graph of TPC-H Q5.

The topology of the predicate transfer graph affects the performance of the predicate transfer phase and also the selectivity of the transferred filters. In this paper, we use a simple heuristic that points an edge from a smaller table to a bigger table. The intuition is the same as why Bloom join builds Bloom filter at the smaller table—to reduce Bloom filter size and increase filter selectivity. Our current heuristic does not remove any edge in the join graph when generating the predicate transfer graph. It also guarantees that the resulting graph is a Directed Acyclic Graph (DAG). The predicate transfer graph in Figure 1b follows this heuristic.

Transfer Schedule. The transfer schedule determines when and how the predicates are transferred across the predicate transfer graph. Numerous design decisions can be made in the schedule. In particular, the schedule specifies which tables in the query should construct initial local filters to start the transfer process, and the order of issuing the remaining transfers. For each table that sends the local filter out, the schedule determines when the transfer happens—multiple transfers may happen in serial or parallel. Moreover, the transfer can happen back and forth, following both directions of certain edges. Pruning may be adopted to avoid non-beneficial transfers, and the transfer direction may be dynamically adjusted at runtime. Identifying a good transfer schedule is critical to the system performance.

In this paper, we adopt a heuristic that builds the transfer schedule using one *forward pass* and one *backward pass* similar to the Yannakakis algorithm. The predicate transfer graph is determined at planning time and remains fixed during runtime. In the forward pass, we build initial local filters on the leaf nodes in the predicate transfer graph (i.e., nodes with only outgoing edges but no incoming edge). These filters are transferred following the topological order of the predicate transfer graph, which exists because the graph is a DAG. If one node has one or more incoming edges, the node will collect all the incoming filters before performing the transformation to produce outgoing filters (LIP-style [39] incoming filter ordering can be utilized for further optimization); the transformation will scan the table only once, regardless of the number of incoming or outgoing edges. The forward pass finishes once all filters are fully transferred.

The system then starts the backward pass, where we simply reverse the direction of all edges and repeat the same process in the forward pass. After both passes are done, each table has been reduced based on the transformed filters it receives. The later join phase will start from these pre-filtered tables.

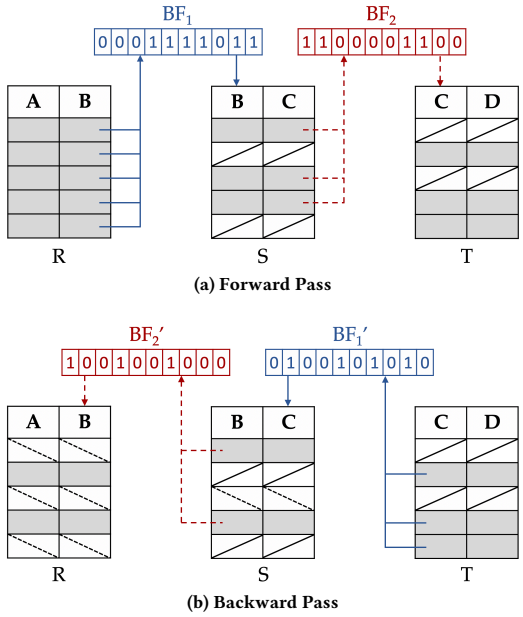


Figure 3: Example of Predicate Transfer on a Join Query — $R \bowtie S \bowtie T$.

Figure 3 presents an example of predicate transfer on a join query, which joins three tables R, S, and T. Assume the transfer starts at table R and all inherit local filters are already consumed. In the forward pass (Figure 3a), R first constructs a Bloom filter (BF_1) on join attribute B and sends it to S. S applies BF_1 to its join attribute B, which filters out two rows (row 2 and 5), and the remaining three rows are used to construct a new Bloom filter (BF_2) on another join attribute C. When T receives the transformed filter from S, the join column C is used to probe the filter where row 1 and 3 are removed. The backward pass (Figure 3b) starts with input as the remaining rows of the forward pass. Table T constructs a Bloom filter (BF_1') on join attribute C, which is applied on $S.C$, with row 3 filtered out. Then S constructs another Bloom filter (BF_2') on the join column B, which is able to filter three rows (row 1, 3, and 5) from R.

A more complicated example is TPC-H Q5, which is shown in Figure 1b. The first Bloom filter is constructed on `region`, and sent to `nation`. The filter is then transformed into two outgoing filters which are sent to `customer` and `supplier` respectively. Similarly, `supplier` transfers two outgoing filters following the edges to `customer` and `lineitem`. At `customer`, two separate incoming filters are applied with one outgoing filter produced and sent to `order`, which is then transformed and sent to `lineitem`. The forward pass finishes when both incoming filters arrive at `lineitem`, and after that the backward pass begins in a symmetric way.

Filter Type. Our discussion so far uses Bloom filters to represent the predicates. In fact, other representation of filters can also be used. If a precise representation is used, i.e., the filter precisely encodes all the join keys, then a transfer becomes a semijoin and the algorithm becomes similar to Yannakakis. An ideal filter should be efficient to construct and check, and achieve low false positive rates. We use Bloom filters in our prototype since it is the best candidate available today. But predicate transfer can automatically benefit from any potential improvement in filtering techniques.

Transfer Path Pruning. As discussed above, our current scheduling heuristic makes two full passes of the predicate transfer graph. In practice, some transfers may not increase filter selectivity but consume computational resources. An intelligent transfer scheduler should identify such scenarios and stop transferring these filters further to avoid wasting CPU cycles. Such transfer path pruning can be done at either planning time or runtime. Our current prototype does not incorporate any pruning and always performs the forward and backward passes in full. We observe this already demonstrates significant performance improvement, and believe incorporating path pruning will lead to even larger speedups.

3.3 Join Phase

After the predicate transfer phase completes, each table may have already been processed by several filters, including the inherent filters from the query and the transferred filters. The join phase basically executes the original query with the reduced input tables.

Unified Query Plan. As a straightforward design, the database can directly execute the query plan as a regular query in the join phase, with the leaf nodes (i.e. scan) replaced by the filtered tables produced by the predicate transfer phase. The predicate transfer schedule is essentially also a query plan. The two query plans can be concatenated such that the leaf nodes in the join plan are just the output nodes of the predicate transfer schedule. This avoids rescanning in the join phase and requires no changes to the executor—the executor is oblivious to the predicate transfer phase and executes the modified query plan regularly.

More Accurate Cardinality Estimation. The predicate transfer phase updates the cardinality of the input tables in the join phase. Therefore, the original query plan generated beforehand may become suboptimal based on the stale cardinalities. A replanning step between the two phases may produce a better plan that leads to further performance improvement. Although join performance will be more robust to join orders (as will be shown in Section 4.3), performance can still be affected by the quality of the query plan, with the factors including the size of materialized intermediate tables, which table to build the hash table and which table to probe, etc. Moreover, similar to the Yannakakis algorithm, predicate transfer bounds the size of the intermediate join tables in the join phase (Section 3.5), which can be utilized to improve cardinality estimation.

3.4 Extension to General Queries

In the discussion above, we assume table joins are inner equi-joins, and cover queries with only joins and local filters (filters over base tables). In this section, we extend the predicate transfer mechanism to further support general queries.

Supporting More Operators in Predicate Transfer Graph. We first extend predicate transfer to support outer joins. In particular, a left outer join operation can be incorporated into the predicate transfer graph by allowing predicate transfer in only one direction, i.e., from the left table to the right table; but the other transfer direction is blocked. Therefore, such a transfer can happen in either forward pass or backward pass, but not in both passes. A right outer join can be supported in a similar way. A full outer join, however, cannot be incorporated into the predicate transfer graph.

Considering more general operators, we note that an operator will block predicate transfer if it does not preserve the join key during the computation (e.g., perform scalar aggregations on the join key). In particular, we identify the following operators that can also be incorporated into the predicate transfer graph.

- Operators including filters between intermediate join tables, column projection, sorting, and top-K do not block predicate transfer.
- Grouped aggregation does not block predicate transfer when the join key is a subset of the group key.
- Scalar user-defined functions do not block the transfer to the downstream join, but may block the transfer to the upstream join if the function is not invertible.

Beyond a Single Predicate Transfer Graph. Some queries may contain operators that cannot be incorporated into a predicate transfer graph. Example operators include but are not limited to full outer joins, scalar aggregations, and group-by aggregations where the join key is being aggregated. When such a scenario is encountered, we can apply predicate transfer only on a subset or several subsets of the query execution plan, and use conventional methods to execute the rest of the query. For example, this means a query can be partially executed first, leading to a subquery plan that can be represented as a predicate transfer graph in order to apply predicate transfer. After the predicate transfer phase and the join phase, the rest of the query can continue execution. It is also possible that predicate transfer can be applied multiple times to different parts of the query plan—the predicate transfer phase and regular query execution can alternate.

In our current prototype, we apply the heuristic that first identifies and executes single-table subquery plans (e.g., group by aggregation on a single table) before the predicate transfer phase.

3.5 Cost Analysis

Compared to the Yannakakis algorithm, predicate transfer does not provide theoretical optimality, but it is more versatile. Predicate transfer supports both precise filters (like semi-join) and Bloom

filters, any join-graph topology, outer joins and cyclic queries, more operators, and complex predicate transfer schedules.

In this section, we present a simple cost analysis of predicate transfer compared to the Yannakakis algorithm and show that predicate transfer is more efficient and robust than Yannakakis, and can achieve close to optimal pre-filtering efficiency. Our key idea is to show that using the cheap Bloom filters drastically reduces the cost of excessive hash probes in the semi-join phase of Yannakakis, filters out most tuples not participating in the joins, and only incurs bounded amount of false positives to be removed in the join phase.

Cost Model. Let t be the number of tables in a given join query and N be the input size (i.e. the total number of tuples in all joining tables). We assign a unit cost to each per-tuple scan, hash table insertion or probe, and a β cost per-tuple for Bloom filter insertion or probe. As a Bloom filter is of a small size and thus likely to be cache resident, Bloom filter operations are typically much cheaper than hash table operations, i.e. $\beta \ll 1$. We assume that the Bloom filter has a false positive rate of $\epsilon \ll 1$ that can be appropriately configured (e.g., we can tune ϵ to be smaller by increasing the Bloom filter size or number of hash functions, but this makes β larger). The reader can refer to [39] for an in-depth study on Bloom filter configurations.

Yannakakis algorithm. At the semi-join phase, scanning tables to build or probe hash tables cost N units, independent of the direction of the forward/backward semi-join passes. The cost of building or probing intermediate hash tables can be bounded by $c_y \cdot N$, where c_y is a constant highly sensitive to the choice of the rooted join tree of the query. An ideal join tree and orientation can drastically reduce the size of intermediate hash tables, leading to a cheaper semi-join phase (smaller c_y). The join phase of Yannakakis is perfectly robust, as every join order costs $t \cdot \text{OUT}$ units of hash table accesses.

Predicate Transfer. At the predicate transfer phase, scanning tables to build or probe Bloom filters costs N units. As we only build or probe Bloom filters, the cost can be bounded by $\beta \cdot c_p \cdot N$ units, where c_p is a constant that depends on the predicate transfer graph topology and the transfer schedule. As $\beta \ll 1$, the sensitivity of the runtime to the constant c_p shrinks by a factor of β.

Let T_k, T_k^* be the size of the kth join table before and after a semi-join phase of Yannakakis. The predicate transfer phase, however, passes a larger table of size $T_k^* + (T_k - T_k^*) \cdot \epsilon$ to the join phase, by an approximated factor of

$$p = \prod_{k=1}^{t} \left(1 + \frac{T_k - T_k^*}{T_k^*} \cdot \epsilon\right).$$

Assume

$$\hat{k} = \arg\max_k \frac{T_k - T_k^*}{T_k^*},$$

then

$$p \le \left(1 + \frac{T_{\hat{k}} - T_{\hat{k}}^*}{T_{\hat{k}}^*} \cdot \epsilon\right)^t = (1 + \epsilon')^t \approx 1 + \epsilon' t,$$

ignoring higher order terms of ϵ', and assuming

$$\epsilon' = \frac{T_{\hat{k}} - T_{\hat{k}}^*}{T_{\hat{k}}^*} \epsilon = \left(\frac{1}{Sel_{\hat{k}}} - 1\right) \epsilon \ll 1,$$

where $Sel_{\hat{k}}$ denotes the smallest join selectivity in the joining tables.

Then in the join phase, the cost of predicate transfer can be approximated as $t \cdot \text{OUT} \cdot (1 + \epsilon' t)$ units. The choice of the join order only affects the extra $\epsilon' t^2 \cdot \text{OUT}$ term. Under our assumption that $\epsilon' \ll 1$ (and thus $\epsilon' t^2 \cdot \text{OUT}$ is small), the join phase still attains promising robustness.

As a summary, the Yannakakis algorithm guarantees maximum filtering at the semi-join phase and perfect robustness at the join phase, but at the cost of a much more expensive and unstable semi-join phase (our evaluation in Section 4.3 verifies this). In contrast, predicate transfer addresses the shortcomings via a more stable and efficient Bloom filter transfer scheme,while maintaining near-maximum filtering capabilities at the predicate transfer phase and near-perfect robustness in the join phase.

4 EVALUATION

This section presents our preliminary evaluation results. We describe the experimental setup in Section 4.1 and compare predicate transfer with baseline join strategies in Section 4.2. Then we perform a deep drive to understand the performance on TPC-H Q5 in Section 4.3.

4.1 Experimental Setup

We conduct experiments on a single AWS EC2 r5.4xlarge instance, with 16vCPU and 128GB memory. The server runs the Ubuntu 20.04 operating system. We use the widely adopted data analytics benchmark, TPC-H, with 22 queries in total. We use both an 1GB data set (a scale factor of 1) and a 10GB data set (a scale factor of 10). Queries are executed on a single CPU core. For all the experiments, we measure the in-memory query performance by running the query twice, where the first run loads all the tables into the memory, and we measure the performance of the second run.

The testbed we use on evaluation is FlexPushdownDB [37] (*FPDB* in short), an open-source cloud-native OLAP DBMS. Table data is placed on local disks in Parquet [4] format unsharded. FPDB leverages join and Bloom filter implementation of Apache Arrow [3]. The evaluation results may vary on different DBMSs, depending on the performance ratio between the join and Bloom filter implemented.

We compare the proposed join strategy PredTrans with three other baselines: NoPredTrans, BloomJoin, and Yannakakis. NoPredTrans does not transfer predicates among joining tables—pairs of tables are joined regularly as in most DBMSs. BloomJoin performs one-hop predicate transfer between joining table pairs, where the build side constructs a Bloom filter which is used to filter the probe side. Yannakakis executes the semi-join phase of the Yanakakis algorithm ahead of the join phase.

Since the vanilla Yannakakis algorithm is only applicable on acyclic conjunctive queries, we make two extensions to make it applicable on all TPC-H queries. First, we adopt the same mechanisms that PredTrans deploys to handle the case of outer joins and non-join operators in the query plan. Second, for cyclic queries like Q5 and Q9, we break the cycle in the join graph by randomly picking a root node and, performing a BFS search from the root. The result join tree represents the transfer order of the semi-join phase in Yannakakis.

4.2 TPC-H Performance

Figure 4 shows the execution time of different predicate transfer strategies on TPC-H queries. Since Q1 and Q6 involve no joins, we exclude them from the benchmark. On average, PredTrans outperforms NoPredTrans by 4.1×, BloomJoin by 3.3×, and Yannakakis by 4.2× for SF1, and 4.0×, 3.2×, and 4.8× for SF10, respectively. 15 queries out of 20 see performance improvement on both data sets.

PredTrans achieves significant performance improvement on queries with a large amount of joins. Half of the queries include joins across at least four tables. Among this, Q2 (joins across nine tables) benefits most from predicate transfer, which outperforms NoPredTrans and BloomJoin by 47× and 44× on SF1, as well as 63× and 61× on SF10, respectively. Through predicate transfer, filter predicates on tables `Part` and `Region` are sent to every other table in the join graph through lightweight Bloom filters. As a result, the predicate transfer phase of PredTrans reduces the size of input tables that participate in the join phase by over 99%, such that the expensive join operations are only performed on a tiny fraction of data. The Yannakakis algorithm outperforms both NoPredTrans and BloomJoin baselines by over 4× on both data sets. In fact, compared to PredTrans, Yannakakis can pre-filter even more unnecessary data records ahead of the join phase since the Bloom filters leveraged by PredTrans incur false positives. However, the small performance benefit within the join phase is overwhelmed by the large overhead brought by semi-joins, making Yannakakis perform worse that PredTrans.

On SF1, we observe the highest speed up on queries Q2, Q5, Q17, Q18, and Q21, between 7× to over 44×. In these queries, there is a subquery executed with the results joined with the tables in the main query, and the large fact table are accessed by both the main query and the subquery (e.g., `Lineitem` in Q17 and Q18). Since NoPredTrans and BloomJoin perform no predicate transfer and one-hop transfer only, a single filter predicate cannot be sent to both the main query and the subquery to pre-filter the corresponding fact table. Conversely, PredTrans broadcasts every filter predicate globally inside the join graph, such that both fact tables in the main query and subquery can be filtered. Moreover, Q17 joins base tables with aggregation results. By executing the aggregation beforehand, predicate transfer is able to achieve a higher selectivity by starting transfer from a smaller intermediate result table. On SF10, Q2 and Q5 achieve more improvement on PredTrans than SF1. The speedup on Q18 is similar on both data sets. However, for query Q17 and Q21, compared to SF1, the speedup on SF10 of PredTrans shrinks — PredTrans outperforms BloomJoin by 1.8× on Q17 and 7.6× on Q21. On Q17, the grouped aggregation on the large fact table dominates the execution time, where the pre-filtering cannot make significant impact. Q21 involves a large amount of joins, which amplifies the amount of false positives brought by Bloom filters. As a result, the amount of remaining rows after the transfer phase is still about 2× as the amount of rows that actually contribute to the final join output.

Queries with fewer join operations benefit less from predicate transfer (e.g., Q13, Q14), since one-hop predicate transfer may already be enough to forward local filter predicates to the global. However, we still observe a large speedup on Q3 (over 9× on both SF1 and SF10). Q3 joins three relatively large tables `customer`, `orders`, and `lineitem`. Since all three tables have local filters, BloomJoin can only transfer a portion of them within a single hop. Instead, PredTrans can make sure each table receives the transformed filter predicates of every other table, which maximizes the effectiveness of the pre-filter phase.

Another interesting observation is that Yannakakis may not always outperform NoPredTrans and BloomJoin baselines. For example, Yannakakis underperforms BloomJoin by 16× and 20× on Q11 on both data sets respectively, and by 4.8× on Q16 with SF10. One cause is that the Yannakakis algorithm does not specify the root of the join tree in the semi-join phase, and a bad semi-join order may construct several large hash tables at the beginning. However, this is not an issue in PredTrans since we use a heuristic to transfer from smaller tables to larger tables (see Section 3.2), minimizing the memory stalls incurred by bitmap operations.

We further identify queries that have operators that block predicate transfer (e.g. outer joins), which are Q13, Q15, Q16, and Q22. Q13 and Q16 have left or right outer joins which block predicate transfer on one direction, Q15 has a join where one input is the result of scalar aggregation, and Q22 contains blocking operators of both kinds. The speedup on these queries of PredTrans is less than other queries, since the transfer is restricted within a small portion of the join graph (e.g. between only two tables), which weakens the pre-filtering power in the predicate transfer phase.

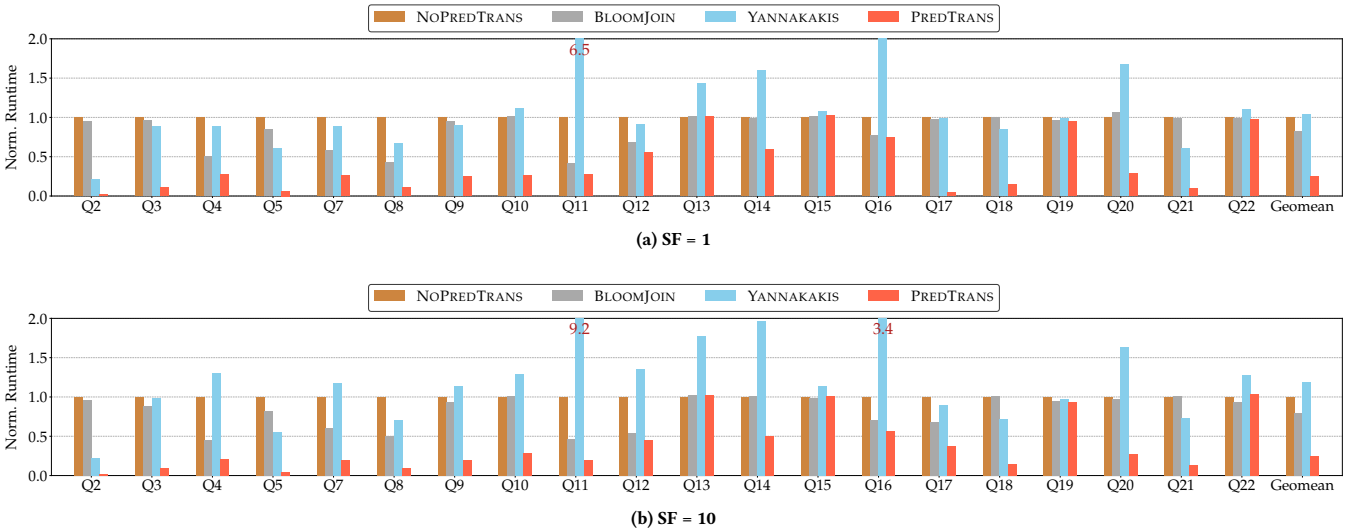
4.3 Case Study — TPC-H Q5

To get a deeper understanding of the performance benefits, we conduct a detailed analysis on Q5, one of the complex queries in TPC-H. The query performs inner joins across six tables, and the join graph is shown in Figure 1a.

Table 1: Join Table Size in Q5 (SF = 1) —*HT* denotes the number of rows in the hash table, and *PR* represents the number of rows that probe the hash table.

	NoPredTrans		BloomJoin		Yannakakis		PredTrans	
	HT	**PR**	**HT**	**PR**	**HT**	**PR**	**HT**	**PR**
Join 1	10K	6M	10K	6M	2K	181K	2K	74K
Join 2	228K	6M	228K	1M	133K	181K	44K	67K
Join 3	150K	910K	150K	44K	69K	193K	15K	60K
Join 4	25	36K	25	36K	5	8K	5	7K
Join 5	1	36K	1	7K	1	8K	1	7K

Join Table Size. We measure the sizes of both input tables of each join, following the join order specified in the query plan (FPDB utilizes Apache Calcite [2] for query optimization like join ordering), and the results are shown in Table 1 and Table 2. PredTrans reduces the join table size by 98% over NoPredTrans, 96% over BloomJoin, and 64% over Yannakakis on SF1 (Table 1), and 98% over NoPredTrans, 92% over BloomJoin, and 67% over Yannakakis on SF10 (Table 2). In BloomJoin, the largest fact table `lineitem` can only be pre-filtered after the first join, where the inner table `Orders` owns local filter predicates that can be trasferred to `lineitem`. PredTrans shows the superiority to be able to pre-filter all join tables ahead of the entire join phase.



(a) SF = 1

(b) SF = 10

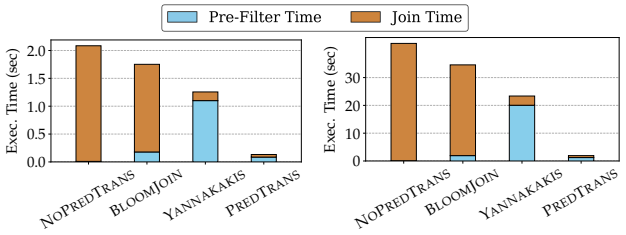
Figure 4: Performance Evaluation of Predicate Transfer on TPC-H (normalized to NoPredTrans).

Table 2: Join Table Size in Q5 (SF = 10) —*HT* denotes the number of rows in the hash table, and *PR* represents the number of rows that probe the hash table.

	NoPredTrans		BloomJoin		Yannakakis		PredTrans	
	HT	PR	HT	PR	HT	PR	HT	PR
Join 1	100K	60M	100K	60M	20K	1.8M	20K	835K
Join 2	2.3M	60M	2.3M	9.3M	1.2M	1.8M	305K	659K
Join 3	1.5M	9.1M	1.5M	499K	689K	1.9M	150K	376K
Join 4	25	364K	25	364K	5	75K	5	73K
Join 5	1	364K	1	73K	1	75K	1	73K

We observe a higher selectivity in the predicate transfer phase achieved by PredTrans, compared to Yannakakis. This is because Yannakakis can only guarantee the optimal pre-filtering on acyclic queries. For a cyclic query (like Q5), some edges in cycles are removed to form a tree, which sacrifices the overall filtering power. Instead, the heuristic adopted by PredTrans allows us to perform transfer for every join (in Q5 there is no blocking operator) regardless the cyclicity of the join graph, resulting in more base table records pre-filtered ahead of the join phase.

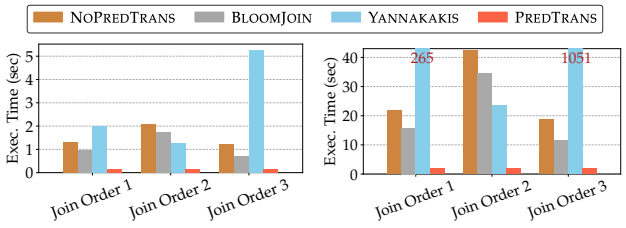
Performance Breakdown. Figure 5 demonstrates the performance breakdown of Q5 in different predicate transfer strategies. The execution time is divided into pre-filter time and join time. Compared to NoPredTrans and BloomJoin, joins are accelerated by 44× and 34× in PredTrans on SF1, as well as 60× and 46× in SF10, due to the significant size reduction of the input join tables. Yannakakis is also able to achieve a shrinkage on the input join tables, but the pre-filtering power is 3.3× and 4.7× less than PredTrans on SF1 and SF10 respectively since not all edges in the join graph are traversed. Moreover, the semi-joins it relies on are computationally expensive and dominate the entire execution time. As a result, the predicate transfer phase in PredTrans outperforms the semi-join phase in Yannakakis by 13× and 16× on both data sets respectively, since bit operations used in Bloom filters are much cheaper than the construction and probe of the hash tables.



(a) SF = 1

(b) SF = 10

Figure 5: Performance Breakdown on TPC-H Q5.



(a) SF = 1

(b) SF = 10

Figure 6: Performance of Different Join Orders on TPC-H Q5.

Robustness. We next evaluate the sensitiveness on join orders for different predicate transfer strategies. We pick three different join orders and the result is shown in Figure 6. In both data sets, PredTrans achieves the best performance and outperforms other

predicate transfer strategies on all the join orders. Notably, the the join order makes a much smaller performance variance in PredTrans (up to 12%) compared to other strategies (up to 45×). We further notice that for just the join phase, Yannakakis achieves the similar robustness as PredTrans does. Both Yannakakis and PredTrans is able to bound the size of the intermediate join results, making their join phase robust to different join orders. The performance of the expensive semi-joins used in Yannakakis is unstable since the join tree construction is not deterministic, making it overall highly sensitive to the join order. Conversely, the heuristic adopted in the predicate transfer phase of PredTrans always points to the same transfer topology and schedule regardless of the join order.

5 CONCLUSION AND FUTURE WORK

A new join optimization, predicate transfer, is proposed in this paper. Inspired by Yannakakis algorithm, predicate transfer generalizes Bloom join to transfer table-local filters to pre-filter multiple other tables. We laid out the design space of predicate transfer and described the heuristics used in our prototype. Evaluation showed an average 3.3× speedup over Bloom join on TPC-H benchmark.

Predicate transfer opens up substantial research opportunities, including better heuristics in the predicate transfer schedule, deeper theoretical analysis on the performance guarantees, and extending the technique to parallel and distributed environment. Discussions of these topics are left as future work.

REFERENCES

[1] 1999. TPC-H Benchmark. http://www.tpc.org/tpch/.
[2] 2014. Apache Calcite. https://calcite.apache.org/.
[3] 2016. Apache Arrow. https://arrow.apache.org/.
[4] 2016. Apache Parquet. https://parquet.apache.org/.
[5] 2019. Getting started with Oracle Database In-Memory Part IV - Joins In The IM Column Store. https://blogs.oracle.com/in-memory/post/getting-started-with-oracle-database-in-memory-part-iv-joins-in-the-im-column-store.
[6] 2020. Improved speed and scalability in Amazon Redshift. https://aws.amazon.com/blogs/big-data/improved-speed-and-scalability-in-amazon-redshift/.
[7] 2021. Best Practices for Optimizing Your DBT and Snowflake Deployment. https://www.snowflake.com/wp-content/uploads/2021/10/Best-Practices-for-Optimizing-Your-dbt-and-Snowflake-Deployment.pdf.
[8] 2022. Introducing Apache Spark™ 3.3 for Databricks Runtime 11.0. https://www.databricks.com/blog/2022/06/15/introducing-apache-spark-3-3-for-databricks-runtime-11-0.html.
[9] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable bloom filters. *Inform. Process. Lett.* 101, 6 (2007), 255–261.
[10] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. 2014. Memory-Efficient Hash Joins. *Proc. VLDB Endow.* 8, 4 (2014), 353–364.
[11] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. 37–48.
[12] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
[13] Kjell Bratbergsengen. 1984. Hashing Methods and Relational Algebra Operations. In *Proceedings of the 10th International Conference on Very Large Data Bases*. 323–333.
[14] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving Hash Join Performance through Prefetching. *ACM Trans. Database Syst.* 32, 3 (2007), 17–es.
[15] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 418–431.
[16] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1891–1904.
[17] Joseph M. Hellerstein. 1998. Optimization Techniques for Queries with Expensive Methods. *ACM Trans. Database Syst.* 23, 2 (1998), 113–157.
[18] Joseph M. Hellerstein and Michael Stonebraker. 1993. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. 267–276.
[19] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. AJAR: Aggregations and Joins over Annotated Relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16)*. Association for Computing Machinery, New York, NY, USA, 91–106. https://doi.org/10.1145/2902251.2902293
[20] Srikanth Kandula, Laurel Orr, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates through Joins in Big-Data Clusters. *Proc. VLDB Endow.* 13, 3 (2019), 252–265.
[21] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proc. VLDB Endow.* 2, 2 (2009), 1378–1389.
[22] Paraschos Koutris. 2011. Bloom Filters in Distributed Query Execution. *Principles of DBMS - University of Washington* (2011).
[23] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.
[24] Alon Y Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. 1994. Query optimization by predicate move-around. In *VLDB*. 96–107.
[25] Lailong Luo, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. 2018. Optimizing bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1912–1949.
[26] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *Journal of the ACM (JACM)* 65, 3 (2018), 1–40.
[27] Anna Pagh, Rasmus Pagh, and S Srinivasa Rao. 2005. An optimal Bloom filter replacement.. In *Soda*. 823–829.
[28] Sukriti Ramesh, Odysseas Papapetrou, and Wolf Siberski. 2009. Optimizing distributed joins with bloom filters. In *Distributed Computing and Internet Technology: 5th International Conference, ICDCIT 2008 New Delhi, India, December 10-12, 2008. Proceedings 5*. Springer, 145–156.
[29] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. 23–34.
[30] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. 1997. Heuristic and randomized optimization for the join ordering problem. *The VLDB journal* 6 (1997), 191–208.
[31] Arun Swami and Anoop Gupta. 1988. Optimization of Large Join Queries. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*. 8–17.
[32] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2011. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials* 14, 1 (2011), 131–155.
[33] Jeffrey D. Ullman. 1988. *Principles of Database and Knowledge-Base Systems, Vol. I.* Computer Science Press, Inc.
[34] Bennet Vance and David Maier. 1996. Rapid Bushy Join-Order Optimization with Cartesian Products. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. 35–46.
[35] Todd L Veldhuizen. 2014. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*.
[36] Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–23.
[37] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *VLDB* 14, 11 (2021), 2101–2113.
[38] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7* (Cannes, France) *(VLDB '81)*. VLDB Endowment, 82–94.
[39] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust: Making the Initial Case with in-Memory Star Schema Data Warehouse Workloads. *Proc. VLDB Endow.* 10, 8 (apr 2017), 889–900. https://doi.org/10.14778/3090163.3090167