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

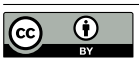
ACM Reference Format:
Yifei Yang and Xiangyao Yu. 2025. Accelerate Distributed Joins with Predicate Transfer. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 122 (June 2025), 27 pages. https://doi.org/10.1145/3725259

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

series of DAG-structured transfers, almost all non-contributing rows can be pre-filtered from the joining tables with a small and bounded amount of CPU time. Prior work [63] has demonstrated significant speedup of predicate transfer in a single-threaded analytics query engine.

Despite its great performance, we identify the following two limitations of the existing predicate transfer technique.

Limitation #1. Lack of Distribution Support. The existing predicate transfer algorithm works only on a single-threaded system while real analytics databases are typically distributed or at least parallelized across multiple threads. Applying predicate transfer to a distributed system is challenging since it requires to construct a Bloom filter over entire column(s) of a table, such that a Bloom filter must be distributed as well. The design is further complicated when the joining tables are partitioned on different keys, leading to potential re-partitioning of Bloom filters.

Limitation #2. Ineffective Pre-Filtering Steps. Some predicate transfer steps may not filter out any rows in the destination table, introducing performance overhead but no speedup. This issue is exacerbated in distributed environments, where unnecessary transfers can lead to higher network latency in terms of extra round-trips and network traffic, which can degrade query performance.

In this paper, we aim to address both limitations above to make predicate transfer more widely applicable. Our first contribution is to extend predicate transfer to a distributed environment where each individual transfer step is performed across multiple nodes. Following distributed join techniques, our designs are either *broadcast-based* or *shuffle-based*. Different from distributed joins, however, the usage of Bloom filters in predicate transfer introduces extra design complexity. We develop a set of distributed transfer designs that excel at different workload scenarios. We also develop an adaptive model that dynamically chooses the optimal design for each predicate transfer step to minimize network traffic.

Our second contribution is to propose an algorithm that can prune unnecessary predicate transfer steps to reduce the overhead of the transfer phase without sacrificing pre-filtering effectiveness. We identify two properties that collaboratively enable transfer pruning: *join key containment* and *predicate containment*. With join key containment, the join key of the destination table is a subset of the join key of the source table prior to considering existing predicates. Predicate containment ensures that the destination table's predicates are at least as selective as those of the source table.

In summary, the paper makes the following key contributions:

- We extend the existing predicate transfer algorithm to a distributed environment, and propose multiple variants of distributed predicate transfer.
- We develop cost-based adaptive execution to select the most effective distributed predicate transfer algorithm for each individual transfer, such that the network cost is minimized.
- We propose a comprehensive algorithm to prune unnecessary transfers which make no contribution to pre-filtering.
- We implement and evaluate the proposed techniques on FlexPushdownDB [61, 62], an open-source[1] distributed analytics query engine. Results on TPC-H and DSB benchmarks show that distributed predicate transfer can improve the query performance by over 3×, and reduce data exchange by over 2.7×.

The rest of the paper is organized as follows. Section 2 provides the background of predicate transfer and distributed joins. In Section 3, we propose multiple designs of distributed predicate transfer algorithm. Section 4 develops a cost model for all proposed distributed predicate transfer approaches, and adaptive execution that selects the most effective distributed predicate transfer design for each individual transfer step. In Section 5, we discuss the pruning algorithm for ineffective transfers. Section 6 presents additional optimization and implementation details. Section 7 evaluates

[1] https://github.com/cloud-olap/FlexPushdownDB/tree/sigmod_25

the performance of the proposed techniques. Finally, Section 8 discusses the related work and Section 9 concludes the paper.

2 Background

This section presents the background of predicate transfer (Section 2.1) and major distributed join techniques (Section 2.2), which lay the foundation for the design of distributed predicate transfer.

2.1 Predicate Transfer

Predicate transfer [63] is the state-of-the-art pre-filtering technique built on top of the classic Yannakakis algorithm [64]. For acyclic queries, the Yannakais algorithm can pre-filter all rows of the joining tables that do not appear in the final join result. 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. The Yannakakis algorithm performs pre-filtering through a semi-join phase. It starts with constructing a join tree for the query, where each node is a table and each edge is an equi-join operation. The algorithm then performs a forward pass and a backward pass. The forward pass starts from the leaf nodes in the join tree, reducing the size of each table by performing semi-joins with its children tables. Then the backward pass traverses the join tree in a top-down fashion where each table is filtered by a semi-join with its parent table. After both passes, all the tables are pre-filtered and the database can perform regular joins on the reduced tables (i.e., the join phase).

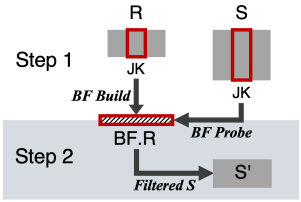
Although the Yannakakis algorithm has nice theoretical guarantees, it has not been widely deployed in modern database engines. The main obstacles are the costly hash table accesses and high memory consumption in the semi-join phase. Predicate transfer enhances the Yannakakis algorithm by adopting Bloom filters [1, 14, 38, 42, 55] for pre-filtering, which achieves high performance due to its efficient bit-level representations. The false positives in Bloom filters do not affect the correctness of query since they will be removed in the join phase; false positives also have low impact on query performance [63].

Figure 1 illustrates how predicate transfer works. Predicate transfer consists of a set of transfer steps. As demonstrated in Figure 1(a), a single transfer step builds a Bloom filter from the join key (i.e., $\mathcal{JK}$) of the source table R to reduce the destination table S to S'. Subsequent predicate transfer steps that have S as the source table, if any, will use the reduced S' instead. The complete predicate transfer phase consists of a forward pass and a backward pass. The existing predicate transfer design [63] adopted the heuristics that transfers are performed from the smaller table to the larger table in the forward pass. Figure 1(b) shows an example for TPC-H Q20. Local filters are applied first. In the forward pass, four transfer steps are constructed based on the join graph, marked in the four red arrows. After that, the backward pass begins in a symmetric way but in the opposite direction, as shown in the four blue dashed arrows. In Q20, local predicates alone can reduce the join input size by 5×. With predicate transfer, the remaining join input can be further reduced by 50×, leading to a total reduction of 250× of the join table sizes.

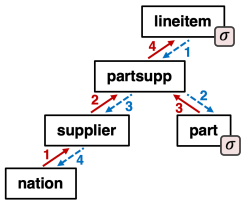
2.2 Distributed Join Strategies

In a distributed environment where the table data is partitioned, joins may not be directly performed, since the joining tables may be partitioned on different columns, and the partition column(s) can be different from the join key. Two distributed join strategies have been prevalently adopted—*broadcast-based distributed joins* and *shuffle-based distributed joins* [3, 5, 22, 28, 48].

With broadcast-based distributed joins, one of the two joining tables is selected to be broadcasted to all the nodes, and is joined with each partition of the other table to form the complete join result. Typically, the smaller table is chosen for broadcasting to minimize network traffic. For outer joins, there are restrictions regarding which table can be broadcasted. For example, in a left outer join,



(a) Predicate Transfer Step



(b) Predicate Transfer in TPC-H Q20

Fig. 1. Demonstration of a Single Predicate Transfer Step (a) and Predicate Transfer in TPC-H Q20 (b).

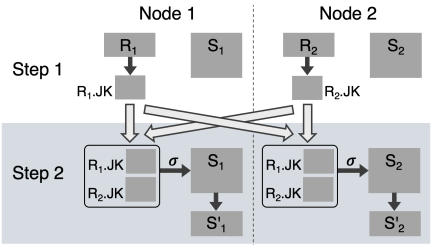


Fig. 2. Broadcast-Based Value-Exchange (BCAST-VAL) — Transfer is from the source table R to the destination table S.

only the right table can be broadcasted. Broadcast-based distributed joins are more efficient than shuffle-based joins when one of the two tables is sufficiently smaller than the other table.

With shuffle-based distributed joins, both tables are repartitioned on the join key across all the nodes, such that the rows with matching join keys are co-located on the same node. Each node can then perform the join locally. Shuffle-based distributed joins are efficient for large clusters and when both tables are large and have similar sizes. Shuffle-based distributed joins are also preferred when one or both of the tables are already partitioned on the join key.

3 Distributed Predicate Transfer

The growing data volumes in modern applications demand distributed analytics databases. The efficient and effective pre-filtering power of predicate transfer can potentially improve distributed join performance substantially, through reducing join computation on each node and network transfer of intermediate join results. However, supporting predicate transfer in a distributed environment is non-trivial with multiple challenges as discussed in Section 1.

We notice that, in a high level, the predicate transfer phase is inherently similar to the join phase—building or probing a Bloom filter is similar to building or probing a hash table. Therefore, we can leverage existing strategies in distributed joins to implement distributed predicate transfer. As discussed in Section 2.2, distributed joins follow either a broadcast-based or shuffled-based approach. Following this, we design broadcast-based (Section 3.1) and shuffle-based (Section 3.2) distributed predicate transfer. Our approaches do not impose any assumptions on the partitioning scheme of the source and destination tables. In some special cases where a table is already partitioned by the join key column(s), our algorithm can be simplified to achieve better performance.

3.1 Broadcast-Based Approaches

3.1.1 Broadcast Join Key Values (BCAST-VAL). Figure 2 demonstrates a straightforward broadcast-based approach that transfers predicates from the source table R to the destination table S, where both tables are initially partitioned across multiple nodes. Specifically, the join key column(s) of R are extracted from each local partition and broadcasted to all the other nodes, such that each node possesses a complete view of table R's join key columns(s) (Step 1). After that, the complete join key of R is used to filter out rows that do not have a match for each partition of S (Step 2). The filtering can potentially be performed in various ways, such as semi-joins, Bloom filters, etc. Our implementation leverages Bloom filters for their high efficiency and low memory consumption. Previous works on distributed semi-join reduction for partitioned tables [59, 65] also adopted this approach, but used semi-joins instead. We observed a 3× slowdown of semi-join reduction compared to Bloom filter reduction. The primary bottleneck lies in the computational overhead introduced by the costly local semi-join operations.

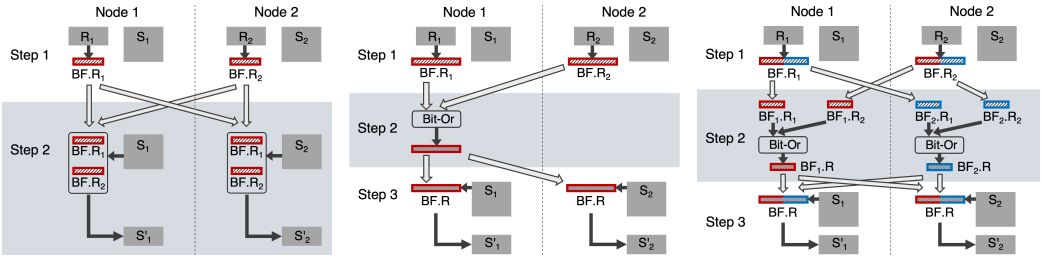


Fig. 3. Broadcast-Based Bloom Filter Exchange (BCAST-BF) — Transfer is from the source table R to the destination table S.

3.1.2 Broadcast Bloom Filters (BCAST-BF). One key insight of predicate transfer is to utilize more compact Bloom filters to reduce the memory footprint and thus achieve higher filtering efficiency. Following the same intuition, broadcast-based distributed predicate transfer may directly broadcast Bloom filters instead of the values in the join key column(s) to reduce the network traffic. Below we discuss several design variants.

Design 1: Broadcast Multiple Bloom Filters. Intuitively, instead of broadcasting the join key column(s) of table R, as shown in BCAST-VAL, we can construct a Bloom filter for the join key at each node, and only send the Bloom filters over the network, as Figure 3(a) shows.

After the broadcast, each node will receive multiple Bloom filters of table R. The filtering of table S is processed as probing all these Bloom filters—a row in S is removed if it fails all the filter checks. Probing multiple Bloom filters for a row can complicate the calculation of false positive rate, which affects the sizing of the Bloom filters of each partition of R. We assume a single Bloom filter is configured with a false positive rate of p. Then the overall false positive rate p' of probing n Bloom filters is expressed as follows.

$$p' = 1 - (1 - p)^n \approx np \tag{1}$$

As Equation 1 shows, naively probing multiple Bloom filters causes the false positive rate to increase. To compensate, we need to use larger Bloom filters. Using m to represent the original number of bits per key in the Bloom filter bit array, to preserve the same desired false positive rate in distributed settings, the adjusted number of bits allocated for each inserted key, m', can be expressed as follows. Here we leverage the theoretical formula of Bloom filter size from [1].

$$m = \frac{|\ln p|}{(\ln 2)^2}, \qquad m' = m + \frac{\ln n}{(\ln 2)^2} \tag{2}$$

Note the above formula is the theoretical bound assuming the optimal number of hash functions. An optimized Bloom filter implementation can stay relatively close to this bound.

Design 2: Merge and Broadcast a Global Bloom Filter. One shortcoming of Design 1 is that each probe checks multiple Bloom filters, which incurs extra computational overhead, particularly when the cluster is large. To address this issue, an alternative design is to construct a global Bloom filter over the entire table R.

As Figure 3(b) shows, each node constructs a partial Bloom filter on the join key column(s) of the local partition (Step 1). Specifically, the bit array of each partial Bloom filter should be sized based on the total number of rows of the entire table R, and the same hash functions should be used for each partial Bloom filter. This allows us to merge all the partial Bloom filters later and achieve the

Table 1. Summary on Variants of BCAST-BF — n represents the number of nodes, N represents the number of rows in the source table R, m denotes the number of bits allocated for each key when probing a single Bloom filter, and m' denotes the adjust number of bits allocated for each key when probing multiple Bloom filters.

	Design 1	**Design 2**	**Design 3**
Compute Cost (per probe)	$O(n)$	$O(1)$	$O(1)$
Network Cost (bits)	$(n-1)m'N$	$2(n-1)mN$	$2(n-1)mN$
Merge Required	×	✓	✓
Load Balancing	✓	×	✓

desired overall false positive rate. After that, all the partial Bloom filters are gathered at one node, and merged through cheap bitwise-or operations, producing a global Bloom filter representing the entire table R (Step 2). Finally, the global Bloom filter is broadcasted to all the other nodes to filter each local partition of table S (Step 3).

Design 3: Merge Bloom filters in a Distributed Manner. In Design 2, we observe a major bottleneck at the node that performs the merge. This single node gathers and broadcasts all partial Bloom filters and performs the merge operation. This causes imbalance of computation and network traffic across the nodes.

Design 3 aims to balance both the network traffic and the merging computation, as demonstrated in Figure 3(c). Assume the cluster consists of n nodes. After the partial Bloom filters are constructed on each local partition of table R, the bit array of each partial Bloom filter is split evenly into n chunks (Step 1). The chunks are then shuffled across the nodes—the i-th chunks of all the partial Bloom filters are gathered at the i-th node. Next, chunks are merged locally, such that each node produces a chunk of the merged global Bloom filter (Step 2). Finally, merged chunks are broadcasted, allowing each node forming a complete global Bloom filter (Step 3).

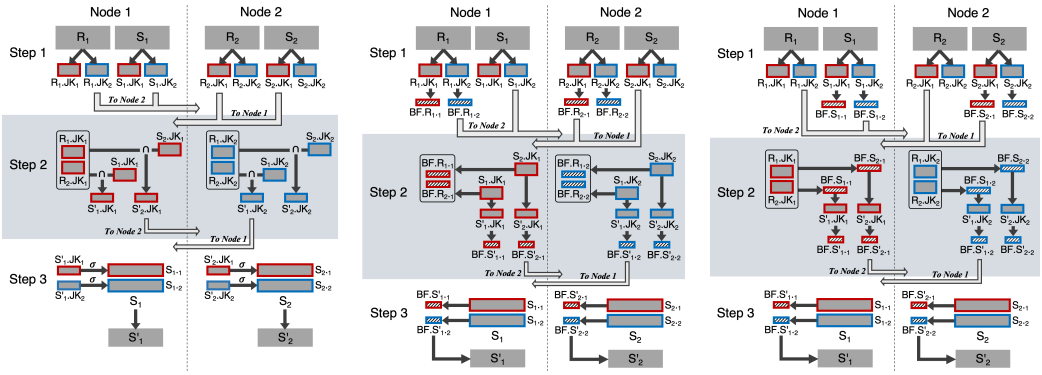
Summary. Table 1 compares the three designs for BCAST-BF distributed predicate transfer. Both Design 2 and Design 3 require only a single Bloom filter check per probe, whereas Design 1 involves n Bloom filter checks. In terms of network traffic, both Design 2 and Design 3 incur the same amount of data exchange, which is lower than Design 1 when the cluster is large enough (i.e., $n > e^{(\ln 2)^2 m}$). Due to load balancing, Design 3 can perform the merge much more efficiently compared to Design 2. Based on our evaluation results (Section 7.3.4), Design 3 achieves the best overall performance. Therefore, for the rest of the paper, the term BCAST-BF will by default refer to Design 3.

3.2 Shuffle-Based Approaches

Broadcast-based distributed predicate transfer incurs network traffic that is proportional to the source table size and the number of nodes in the cluster. Therefore, it is efficient when the source table R is significantly smaller than the destination table S, and when the cluster is relatively small. In other cases, shuffle-based algorithms can be more efficient, since they incur data exchange proportional to the input table sizes regardless of the node count.

3.2.1 Shuffle Join Key Values (SHFL-VAL). In a shuffle-based distributed join, the shuffle operation is performed on the entire table. For distributed predicate transfer, shuffling only the join key column(s) is sufficient. In a high level, to filter the destination table S, essentially what we need is to compute the intersection of the join keys of both the source table R and the destination table S; the non-join-key columns do not need to be involved.

As Figure 4(a) shows, initially both tables are repartitioned based on the join key columns(s), and only the join keys are projected out and exchanged over the network (Step 1). After that, we compute the intersection of the join keys of both tables at each node locally, forming the filtered join



(a) SHFL-VAL (b) SHFL-BF-SRC (c) SHFL-BF-DST

Fig. 4. Shuffled-Based Distributed Predicate Transfer — Transfer is from the source table R to the destination table S.

key for the destination table S (Step 2). Specifically, we compute the intersection of the aggregated input of table R (e.g., $R_1.\mathcal{J}K_1 + R_2.\mathcal{J}K_1$) and each input of table S (e.g., $S_1.\mathcal{J}K_1$, $S_2.\mathcal{J}K_1$). This allows us to isolate the filtered join keys (e.g., $S'_1.\mathcal{J}K_1$, $S'_2.\mathcal{J}K_1$) among different partitions of table S.

Afterwards, the filtered join keys are sent back to their original locations respectively. Specifically, $S'_2.\mathcal{J}K_1$ is sent back to node 2 and $S'_1.\mathcal{J}K_2$ is sent back to node 1. Finally, rows of table S that do not have a match against its filtered join key are removed (Step 3).

3.2.2 Shuffle Bloom Filters for the Source Table (SHFL-BF-SRC). The key insight of shuffle-based distributed predicate transfer is to compute the intersection of the join keys. We note that it is not necessary to obtain the join keys of both tables in order to perform the intersection, as done in the design above. We can replace the join key values by Bloom filters to reduce network traffic.

Figure 4(b) illustrate the design where we shuffle the Bloom filters for the source table R. For the destination table S, we still shuffle the original join key column(s). After the local repartitioning, each join key partition of table R is used to construct a Bloom filter, and the Bloom filters are then exchanged over the network (Step 1). After the shuffle, the intersection of the join keys are computed (Step 2) by probing a set of Bloom filters of table R (e.g., $BF.R_{1\cdot1} + BF.R_{2\cdot1}$) using each shuffled join key of table S (e.g., $S_1.\mathcal{J}K_1, S_2.\mathcal{J}K_1$).

In a sense, the Bloom filter probe operations follow the Design 1 of BCAST-BF distributed predicate transfer, where each probe involves checking multiple Bloom filters. One can also think of an alternative approach as constructing partial Bloom filters and merging into a global one afterwards, similar to Design 2 and Design 3 of BCAST-BF. However, these partial Bloom filters have to size their bit arrays based on the total number of input rows across all partial Bloom filters, which incurs network traffic proportional to the node count, violating the design principle of shuffle-based approaches.

Finally, after the intersection has been computed, we can also send Bloom filters instead of the reduced join keys of table S to their original locations (Step 3), which can further reduce network traffic. Since the cheaper Bloom filters are utilized for the source table R, this design should be more efficient when the destination table S is relatively small.

3.2.3 Shuffle Bloom Filters for the Destination Table (SHFL-BF-DST). Symmetrically, we can shuffle the Bloom filters for the destination table S instead, which is shown in Figure 4(c). For the source table R, we still shuffle the original join key column(s). In this design, to compute the intersection

Table 2. Parameters Used in the Cost Model.

Parameter	Description
N_{src}, N_{dst}	Number of rows in the source and destination table
s	Filtering selectivity for the destination table
l	Number of bits occupied per element in the join key
m	Number of bits allocated per element for probing a single Bloom filter
m'	Adjusted number of bits allocated per element for probing multiple Bloom filters
p	Desired false positive rate of Bloom filters
n	Number of nodes in the cluster

Table 3. Summary of Cost for Different Distributed Predicate Transfer Algorithms.

Algorithm	Cost
BCAST-VAL	$(n-1)N_{src}l$
BCAST-BF	$2(n-1)N_{src}m$
SHFL-VAL	$\frac{n-1}{n}(N_{src}+N_{dst}(1+s))l$
SHFL-BF-SRC	$\frac{n-1}{n}(N_{src}m'+N_{dst}(l+(p+s-ps)m))$
SHFL-BF-DST	$\frac{n-1}{n}(N_{src}(l+npm)+N_{dst}(1+s-ps)m)$

of the join keys, we probe each shuffled Bloom filter of table S (e.g., $BF.S_1.1, BF.S_2.1$) using the aggregated input of table R (e.g., $R_1.\mathcal{JK}_1 + R_2.\mathcal{JK}_1$).

This design can be more efficient than SHFL-BF-SRC when the source table R is relatively small, since the cheaper Bloom filters are shuffled for the destination table S. Besides, when computing the intersection, the amount of false positives incurred by Bloom filter operations is determined by the size of shuffled join keys of table R. A smaller table R leads to fewer false positives.

4 Cost-Based Adaptive Execution

In this section, we develop cost-based execution that selects the cheapest distribute predicate transfer algorithm adaptively at runtime. In a high level, distributed predicate transfer incurs both a network cost and a computation cost, due to data exchange and Bloom filter operations respectively. Our cost model focuses primarily on analyzing the network cost—the computation cost is typically proportional to the network cost in distributed predicate transfer.

Table 2 lists the required parameters used in the cost model, and Table 3 summaries the derived cost for different distributed predicate transfer algorithms. During query execution, the algorithm with the lowest cost is selected for each predicate transfer step. Calculating these costs at runtime is straightforward since the required parameters are easy to collect prior to the processing of each transfer. For example, N_{src} and N_{dst} can be determined once the source and destination tables are fully produced, and s can be obtained from the cardinality estimator within the query optimizer. Other parameters are statically configured. We next explain the cost of each distributed predicate transfer algorithm in detail.

The cost of broadcast-based distributed predicate transfer is straightforward. For BCAST-VAL, each of the source table's join key is sent to the other $n-1$ nodes. In BCAST-BF, Bloom filters are required to be merged prior to filtering, which incurs two rounds of data exchange consequently. Recall BCAST-BF by default refers to Design 3 in Section 3.1.2.

All shuffle-based distributed predicate transfer algorithms exchange data with two rounds, specifically, shuffling the join keys (or Bloom filters), denoted by $Cost_{shuffle}$ below, and sending back the filtered join keys (or Bloom filters) to their original locations, denoted by $Cost_{send\text{-}back}$ below.

For SHFL-VAL, the network traffic incurred by shuffling the join keys of both tables can be expressed as follows.

$$Cost_{shuffle}(SHFL\text{-}VAL) = \frac{n-1}{n}(N_{src} + N_{dst})l \tag{3}$$

After the intersection of the join keys is computed, the amount of filtered join keys that are sent back is shown in Equation 4.

$$Cost_{send\text{-}back}(SHFL\text{-}VAL) = \frac{n-1}{n}sN_{dst}l \tag{4}$$

SHFL-BF-SRC shuffles Bloom filters instead of the join key for the source table. The shuffle cost can be expressed as follows.

$$Cost_{shuffle}(SHFL\text{-}BF\text{-}SRC) = \frac{n-1}{n}(N_{src}m' + N_{dst}l) \tag{5}$$

SHFL-BF-SRC sends back Bloom filters constructed from the filtered join keys for the destination table. Such Bloom filters produce false positives. Representing the amount of filtered join keys as $N_{filtered}$, we can express the desired false positive rate in Equation 6.

$$p = \frac{N_{filtered} - sN_{dst}}{(1-s)N_{dst}} \tag{6}$$

Therefore, the cost of sending back filtered join keys (i.e., $N_{filtered}$) can be formulated as follows.

$$Cost_{send\text{-}back}(SHFL\text{-}BF\text{-}SRC) = \frac{n-1}{n}N_{filtered}m = \frac{n-1}{n}(p + s - ps)N_{dst}m \tag{7}$$

Finally, for SHFL-BF-DST, Bloom filters are shuffled for the destination table while the source table still shuffles the join key. Its shuffle cost is expressed as follows.

$$Cost_{shuffle}(SHFL\text{-}BF\text{-}DST) = \frac{n-1}{n}(N_{src}l + N_{dst}m) \tag{8}$$

Again, the computation of intersection using Bloom filters produce false positives. Assuming the amount of actual filtered join keys of the destination table is $N_{filtered}$, the desired false positive rate p can be formulated as follows.

$$p = \frac{N_{filtered} - sN_{dst}}{nN_{src} - sN_{dst}} \tag{9}$$

As a result, the cost of sending back filtered join keys (i.e., $N_{filtered}$) can be formulated as follows.

$$Cost_{send\text{-}back}(SHFL\text{-}BF\text{-}DST) = \frac{n-1}{n}N_{filtered}m = \frac{n-1}{n}(npN_{src} + (1-p)sN_{dst})m \tag{10}$$

The adaptively chosen distributed predicate transfer algorithm is independent of the join algorithm selected by the query optimizer, since the predicate transfer phase is completely separate from the join phase. However, the use of predicate transfer may influence the query optimizer's choice of join algorithm, since pre-filtering can alter the input table sizes in the join phase. Despite this, when predicate transfer is used, different variants of our distributed predicate transfer algorithms always lead to the same join algorithm chosen by the optimizer. This is because these algorithm variants ensure consistent filtering selectivity when reducing the join tables.

5 Pruning

The existing predicate transfer algorithm [63] processes all transfer steps entirely, as shown in Figure 1(b). However, we observe that some predicate transfer steps may not filter out any rows in the destination table. These ineffective transfers cannot speedup the join phase but introduces performance overhead in the predicate transfer phase. Such overhead is particularly prominent in a distributed environment since it involves not only wasted computation, but also extra network latency and traffic.

In this section, we present an algorithm to discover unnecessary transfer steps prior to the predicate transfer phase, without sacrificing any filtering capability. Intuitively, a predicate transfer step cannot filter out any rows if the join key of the destination table is entirely covered by the join key of the source table, since every row of the destination table can find a match in the source table. Following this, we discover two key characteristics that contribute to the prunability of a predicate transfer step—*join key containment* (Section 5.1) and *predicate containment* (Section 5.2), and then develop a pruning algorithm based on these insights (Section 5.3).

5.1 Join Key Containment

We begin by investigating the prunability of a transfer step prior to applying any potential predicates (both local and transferred predicates). Technically, we can possibly prune a predicate transfer step if the join key of the destination table is a subset of the join key of the source table—there is a join key containment from the source table to the destination table. Below we discuss several cases that induce join key containment.

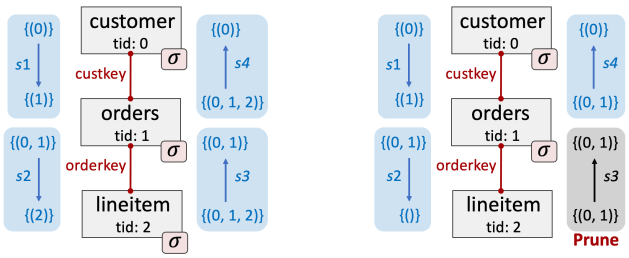
Referential Constraints. One representative case that preserves join key containment is a referential constraint where the join key of the source table is the primary key, and the join key of the destination table represents the foreign key. Join key containment is satisfied by definition.

Reversed Containment of Referential Constraints. Besides the regular referential constraints, we can perform table column analysis that discovers more potential join key containments. Such information can be collected offline and stored within the schema metadata. However, it appears impractical to examine every possible column pair due to the potential high arity and cardinality of the joining tables. We make the observation that join key containment is prevalently held from the foreign key column(s) to the primary key column(s) as well. For example, over the eight referential constraints defined in the TPC-H [56] schema, seven of them satisfy the reversed containment. In this paper we only inspect reversed containment based on defined referential constraints, which is both effective and efficient. The exploration of other join key containment is left for future work. For referential constraints with composite keys, we inspect potential reversed containment for all the subsets of the key columns, since the full reversed containment may not be satisfied while some subsets may do.

Self-Joins. Another possibility to fulfill the join key containment is self-joins. Specifically, within a predicate transfer step, the source and destination tables refer to the same base table. In such cases, the domains of the join keys from the both tables are the same.

5.2 Predicate Containment

In practice, a predicate transfer step may involve applying predicates to both tables prior to the transfer, such as predicates received from previous transfers. Join key containment alone cannot ensure prunability after these predicates are applied. Therefore, we also examine *predicate containment*—whether the predicates of the source table are fully contained by the predicates of the destination table. Essentially, predicate containment ensures that the destination table's predicates



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

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

are at least as selective as those of the source table. A transfer can be safely pruned when both join key containment and predicate containment are satisfied.

Predicate Origins. For each transferred predicate, we trace the table origin which produces the predicate. Intuitively, each predicate is initially a local predicate from a base table, and then transferred to other tables. For the case when join key containment is not satisfied, it is considered as a local predicate produced and transferred from the source table.

For each table, we record the predicate origin for each received predicate, and update the predicate origins during the inspection of a transfer. Specifically, the predicate origins of the source table are absorbed by the predicate origins of the destination table, simulating that the transfer had already been performed. Essentially, predicate containment is satisfied if the predicate origins of the source table are a subset of the predicate origins of the destination table.

Figure 5(a) shows a running example using TPC-H Q3. For each table we issue a unique table ID. Initially, the predicate origins of all the tables are initialized as themselves, representing that each table has a local predicate. The first transfer $s1$ happens between `customer` ($tid = 0$) and `orders` ($tid = 1$). After that, `orders` expands its predicate origins by absorbing the predicates of `customer`. At transfer $s3$, the source table `lineitem` has three predicates, its own local predicate, and predicates from `customer` and `orders` respectively, which are then absorbed by `orders` for the last transfer. None of the four transfers satisfies predicate containment.

In Figure 5(b), we illustrate with a variant of TPC-H Q3, where we remove the local predicates of `lineitem`. In this example, predicate containment is satisfied by transfer $s3$—the backward transfer from `lineitem` to `orders` does not have stronger filtering power than the filters that `orders` has already observed in the forward pass. Since $s3$ also preserves join key containment, it can be pruned.

5.3 The Pruning Algorithm

Algorithm 1 presents the pruning algorithm for individual transfer steps. Specifically, we inspect the pruning opportunities for the forward pass (line 1) and then the backward pass (line 17). Initially, we visit each table (i.e., vertex in the transfer graph) to check whether the table has a local predicate (line 4), and add one to its predicate origins if so (line 5). Then we traverse each transfer (i.e., edge in the transfer graph) following the topological order (line 6), simulating how predicate transfer is actually performed. We first inspect join key containment. If it is not satisfied, then the transfer cannot be pruned (lines 9–10). Such a case is regarded as a local predicate produced by the source table is transferred to the destination table (line 11). When join key containment is satisfied (line 12), we examine predicate containment which determines the final prunablility of the transfer step (line 13). For each transfer that cannot be pruned (line 14), we add the predicate origins of the source table to the predicate origins of the destination table (line 15).

Algorithm 1: Pruning for Predicate Transfer Steps

```
   Input: Predicate Transfer Graph G = (V, E)
1  forward ← true
2  foreach v in V do
3      v.predOrigins ← ∅
4      if HasLocalPred(v) then
5          v.predOrigins.AddLocal()
6  E_sort ← TopologicalSort(E)
7  foreach e = (src, dst) in E_sort do
8      e.prune ← true
9      if not JoinKeyContainment(src, dst) then
10         e.prune ← false
11         dst.predOrigins.AddTransferred(src)
12     if e.prune then
13         e.prune ← PredicateContainment(src.predOrigins, dst.predOrigins)
14     if not e.prune then
15         dst.predOrigins.AddTransferred(src.predOrigins)
16 if forward then
17     E_rev ← Reverse(E)
18     E_sort ← TopologicalSort(E_rev)
19     forward ← false
20     goto line 7
```

After the forward pass has been visited entirely (line 16), we start pruning for the backward pass. Specifically, we reverse the direction of all the edges in the predicate transfer graph (line 17), recompute the topological order (line 18), and then perform the pruning procedure again on the reversed transfer graph (line 20).

6 Additional Optimizations and Implementation

In this section, we present additional optimizations and implementation details regarding how we deploy distributed predicate transfer.

Efficient Multi-Threaded Bloom Filter Operations. We extend the existing predicate transfer implementation to support multiple threads in a single machine. For constructing the Bloom filter, we utilize Apache Arrow's [4] implementation, which applies radix-partitioning to the input data batch to efficiently resolve write conflicts. Parallelizing the Bloom filter probing process is straightforward, as the probe operations are read-only.

Right-Sizing the Hash Values. Arrow's Bloom filter employs a cache-efficient, *block-based* design. Specifically, the bit array of the Bloom filter is chunked into small, fixed-length blocks, and each Bloom filter operation is confined to a specific block. By default, we access Bloom filters with 32-bit instead of 64-bit hash values for small memory footprint. Half of the hash bits are used to identify the correct block, and the remaining bits determine how the data within the block should be manipulated. When the input data to construct the Bloom filter is large such that a bit array larger than 64K blocks is required, 32 bits are not sufficient to locate the block. In such cases, 64-bit

hash values are used instead. As a result, different predicate transfer steps may require different amount of hash bits, which is dynamically chosen at runtime.

Judiciously Pushing Predicate Transfer Below Aggregations. In a distributed environment, we observed that the choice of whether to perform predicate transfer before aggregations can significantly impact query performance, assuming the aggregations do not block the transfer. Specifically, when the aggregation reduces the number of unique keys, such as with a "having" clause, it is more advantageous to execute the aggregation first, since this can generate more selective predicates for the transfer. However, when the aggregation does not reduce the number of unique keys, pushing predicate transfer below the aggregation can be more efficient, which mitigates the performance bottleneck of aggregating a large input. In our distributed query engine, this adaptive approach is applied during the query optimization phase.

Separating Predicate Transfer and the Join Phase. Our current implementation separates predicate transfer and the join phases. There are potential future optimizations. For example, we can potentially interleave the join phase with the backward transfer pass. This would allow the initial joins to start earlier, rather than waiting for the entire predicate transfer process to finish.

7 Evaluation

In this section, we evaluate distributed predicate transfer and the transfer step pruning algorithm using standard OLAP benchmarks.

7.1 Experimental Setup

Server Configuration. We conduct all experiments using AWS EC2 [2] r5.4xlarge virtual machines with 16 vCPU, 128 GB memory, and up to 10 Gbps network bandwidth. The servers run the Ubuntu 20.04 operating system. We use various sizes of clusters including 4, 8, and 16 nodes. The testbed we use is FlexPushdownDB [61, 62], an open-source distributed analytics query engine.

Benchmarks. We use the widely adopted data analytics benchmark, TPC-H [56]. TPC-H contains 22 queries in total. We exclude Q1 and Q6 since they do not contain joins and thus predicate transfer will not make any performance impact. Besides TPC-H, we further evaluate on the DSB [25] benchmark, which contains 15 single-block queries and 22 multi-block queries. DSB is an adaptation of the widely-used industry standard TPC-DS [57] benchmark, enriched with features such as advanced data distribution (e.g., incorporating skews and correlations) and semantically richer query templates. These enhancements aim to simulate more realistic and practical scenarios. We use scale factors of 100 (i.e., 100 GB data set when uncompressed), 200, and 400. By default we conduct experiments using 4 servers with scale factor 100. Such setups are used by many commercial systems on benchmarking their performance [23, 24]. Table data is sharded into partitions of roughly 150 MB, and preloaded into the memory of the executor nodes. Since the proposed distributed predicate transfer techniques do not impose any assumption on the underlying data partitioning schemes, we use the round-robin partitioning scheme, which ensures a roughly equal distribution of data without relying on any specific data attributes to determine placement.

Bloom Filters. We configure all Bloom filters with $m = 8$ and $k = 5$, where m represents the number of bits allocated per element during insertions, and k denotes the number of hash functions used. These settings result in a false positive rate of less than 2%.

Baselines. We compare distributed predicate transfer (i.e., `PT`) against the following baselines:

- `NoPT`: Distributed query execution without predicate transfer.
- `QS`: Distributed query execution that incorporates techniques accommodated from Quick-Step [43] and LIP [66]. These techniques were originally designed for single-node execution.

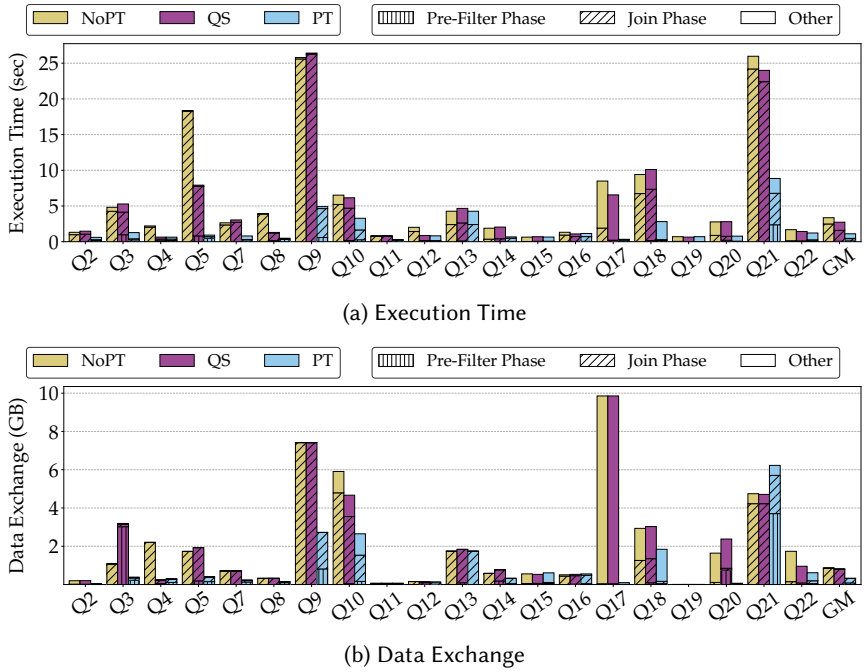


Fig. 6. TPC-H Performance of Distributed Predicate Transfer (SF100, 4 nodes).

To enable their application in distributed environments, we adapted `BCAST-BF` into `QS` when constructing Bloom filters from dimension tables, since `QS` can only reduce the fact table using Bloom filters over the entire dimension tables.

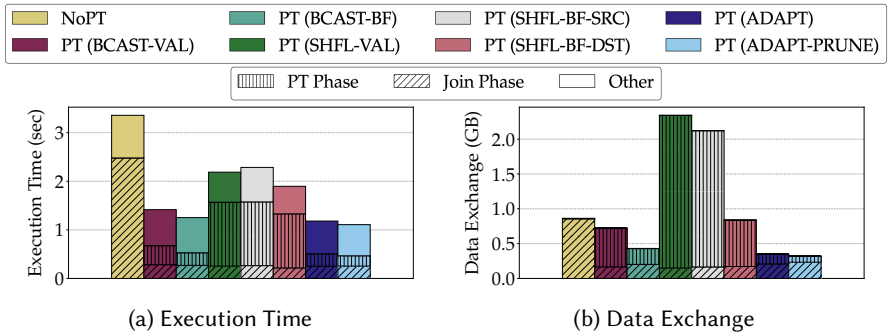
Measurement. For all the experiments, we measure the end-to-end query execution time and network traffic. In the join phase, we have implemented both broadcast-based and shuffle-based distributed joins, and an adaptive, cost-based solution. By default, we conduct experiments using cost-based adaptive distributed joins.

7.2 TPC-H Performance

This subsection evaluates the performance using TPC-H under the default configuration (i.e., a scale factor of 100 and a 4-node cluster). For query execution with `PT`, we use the cost-based adaptive execution which selects the cheapest distribute predicate transfer variant at runtime for each individual transfer step. Pruning of unnecessary transfers is also enabled.

Figure 6(a) reports the query execution time. On average, `PT` is 3× faster than `NoPT` and 2.5× faster than `QS`. Of the 20 queries evaluated, 14 queries show a performance improvement with `PT` over both `NoPT` and `QS`, including 3 queries with more than a 5× speedup over the both baselines. Specifically, `PT` outperforms `NoPT` and `QS` on Q5 by 20× and 9×, respectively, and achieves speedups of 25× and 20× on Q17, respectively. Figure 6(b) presents the corresponding data exchange volume. `PT` reduces the amount of data exchanged by 2.7× over `NoPT`, and by 2.5× over `QS`. 13 queries show a reduction in network traffic over the both baselines. Specifically, `PT` can reduce network traffic by 99× on Q17 over both `NoPT` and `QS`, and achieves a 24× and 36× reduction on Q20, respectively.

`QS` outperforms `NoPT` by 23%, primarily due to pre-filtering using LIP, which applies Bloom filter operations for star schemas with left-deep join trees, progressing from smaller dimension tables to



(a) Execution Time (b) Data Exchange

Fig. 7. TPC-H Performance of Different Distributed Predicate Transfer Algorithms — GeoMean across all queries (SF100, 4 nodes).

the larger fact table. The amount of data exchange is reduced by 6%. The most significant speedup is observed in Q4 and Q8, where QS outperforms NoPT by 3×. However, for queries like Q3, Q7, Q9, and Q18, QS slightly underperforms NoPT because the pre-filtering within left-deep join subtrees eliminates only a small subset of unnecessary rows. On average, QS reduces the join input table size by 2.6×, while PT achieves a reduction of 17×.

To evaluate the overhead introduced by the predicate transfer algorithm, we present a performance breakdown in Figure 6. As Figure 6(a) shows, the join phase dominates the execution time of both NoPT and QS. This is because NoPT does not perform pre-filtering, and the pre-filtering in QS is limited to star-schema subtrees. In contrast, PT significantly reduces the join phase's execution time. Notably, the pre-filtering phase accounts for 19% of the total execution time in PT. This overhead corresponds to 6% of the execution time in NoPT. We observed from Figure 6(b) that the join phase incurs the most network traffic in all solutions. In PT, pre-filtering incurs 27% of the data exchange volume, which is relatively small due to transmitting only the join key columns or more compact Bloom filters. Our experiments on larger clusters and scales (i.e., SF200, 8-node and SF400, 16-node) reveal the same amount of overhead in pre-filtering.

7.3 Distributed Predicate Transfer Algorithms

In this subsection, we compare performance among different variants of distributed predicate transfer algorithms (Section 7.3.1), and evaluate the effectiveness of both the cost-based adaptive execution (Section 7.3.2) and the pruning technique (Section 7.3.3). We also compare different design variants of BCAST-BF (Section 7.3.4).

7.3.1 Distributed Predicate Transfer Algorithms. Figure 7 shows the average performance of different distributed PT algorithms on TPC-H. As Figure 7(a) shows, all proposed distributed PT algorithms outperform NoPT, between the largest speedup of 2.7× observed on BCAST-BF and the smallest speedup of 47% on SHFL-BF-SRC. We show a comparison on the amount of data exchange in Figure 7(b). BCAST-VAL and BCAST-BF reduce amount of data exchange by 16% and 2× respectively. Network traffic of SHFL-BF-DST is close to NoPT. SHFL-VAL and SHFL-BF-SRC increase the amount of data exchange by 2.7× and 2.5× respectively, compared to NoPT.

In general, we observed a relatively higher performance and lower amount of data exchange on broadcast-based designs, compared to shuffle-based designs. In the forward pass, the transfer is mostly performed from the smaller table to the larger table. Broadcast-based designs only require to exchange data for the smaller source table. Shuffle-based designs, on the contrary, need to perform the shuffle operation for the join keys of both tables, and network traffic from the large destination table can be substantial.

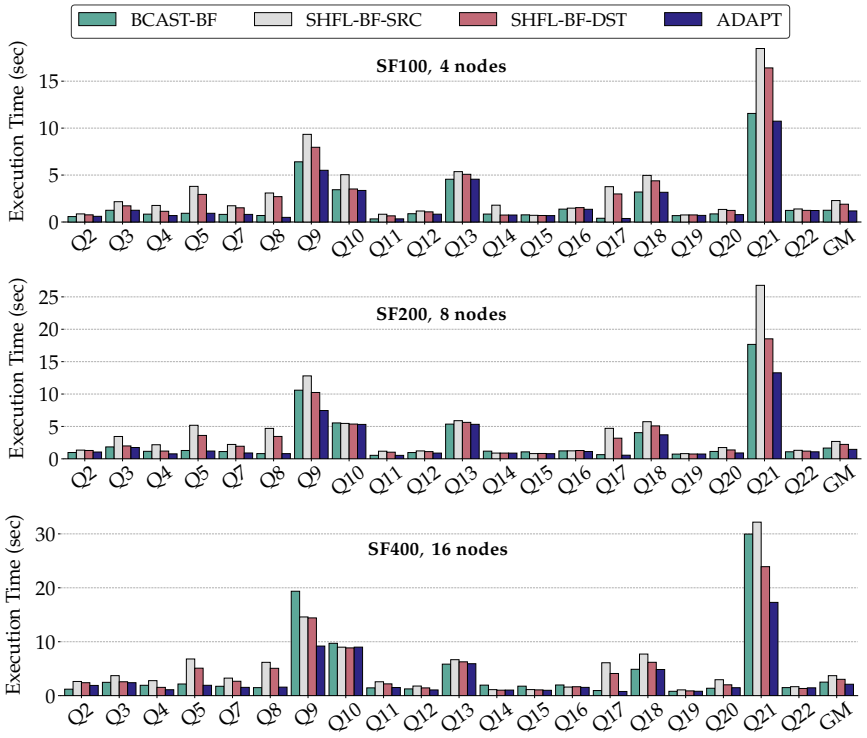


Fig. 8. TPC-H Performance of Distributed Predicate Transfer Algorithm and Cost-Based Adaptive Execution on Individual Queries.

BCAST-BF outperforms BCAST-VAL, since transmitting Bloom filters instead of join key column(s) can reduce network traffic and eliminate duplicated constructions of the global Bloom filter at separate executor nodes. Similarly, in shuffle-based approaches, both SHFL-BF-SRC and SHFL-BF-DST can reduce network traffic compared to SHFL-VAL. However, while SHFL-BF-DST performs better than SHFL-VAL, the performance of SHFL-BF-SRC is actually worse. This is because each probe in SHFL-BF-SRC has to check multiple Bloom filters, incurring additional computational overhead.

We also demonstrate a performance breakdown in Figure 7. The major difference among all the distributed predicate transfer algorithms is reflected in the predicate transfer phase, which implies that they achieve roughly the same overall pre-filtering selectivity.

Finally, we observed that the cost-based adaptive execution consistently outperforms all static algorithm variants, with a speedup of 6% and a network traffic reduction of 18% over the best static algorithm variant (i.e., BCAST-BF). Pruning unnecessary transfer steps further enhances performance by 7% and reduces data exchange by 8%. Detailed results on adaptive execution and transfer pruning are presented in Section 7.3.2 and Section 7.3.3, respectively.

7.3.2 Cost-Based Adaptive Execution. Figure 8 illustrates the effectiveness of cost-based adaptive execution compared to static distributed predicate transfer algorithms. Transfer step pruning is not applied. In our experiments, BCAST-VAL and SHFL-VAL are never selected during any transfer step in the adaptive execution, because algorithms that utilize Bloom filters (i.e., BCAST-BF, SHFL-BF-SRC, SHFL-BF-DST) generally offer superior performance. Therefore we exclude BCAST-VAL and SHFL-VAL in Figure 8. In addition to the default configuration (i.e., SF100, 4 nodes), we also evaluated on larger clusters (i.e., SF200, 8 nodes and SF400, 16 nodes).

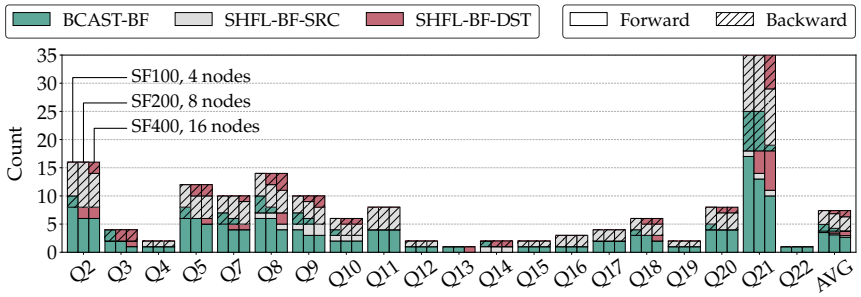


Fig. 9. Breakdown of Selected Distributed Predicate Transfer Algorithms in Cost-Based Adaptive Execution on TPC-H.

Cost-based adaptive execution consistently outperforms all static algorithm variants across all three cluster sizes. Specifically, the adaptive solution surpasses the best-performing static variant by 6%, 14%, and 19% on 4 nodes, 8 nodes, and 16 nodes, respectively, and the second-best static variant by 61%, 52%, and 43% respectively.

Among the static algorithms, `BCAST-BF` consistently performs the best across all scales, outperforming the second-best variant by 51%, 34%, and 21% on 4 nodes, 8 nodes, and 16 nodes, respectively. However, its performance advantage degrades on larger clusters due to increased network traffic, which scales proportionally with the number of nodes. In contrast, shuffle-based approaches become more efficient on larger clusters, as their data exchange volumes are almost not impacted by the cluster size.

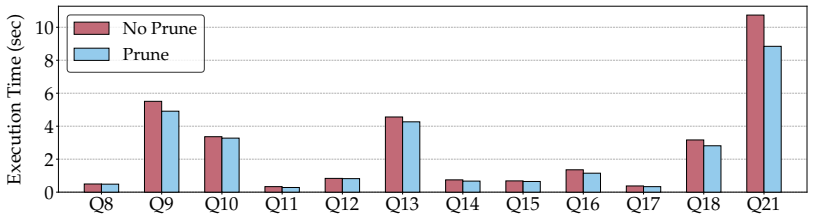
The adaptive solution's advantage lies in its ability to intelligently select the most beneficial algorithm for **each individual transfer step**, enabling it to match or even surpass the best static variant in most queries. For instance, on Q9 and Q21, the adaptive solution dynamically applies a mix of `BCAST-BF`, `SHFL-BF-SRC`, and `SHFL-BF-DST` during transfers. Consequently, the adaptive execution outperforms the best static algorithm variant by 16%, 37%, and 57% on 4 nodes, 8 nodes, and 16 nodes, respectively for Q9, and by 8%, 33%, and 38% for Q21.

To further understand the benefit of the adaptive solution, we recorded the count of each distributed predicate transfer algorithm variant chosen by the adaptive solution, which is shown in Figure 9. In the forward pass, `BCAST-BF` is the most prominent algorithm (e.g., Q5, Q8, Q9, etc.). This is because the source table are usually significantly smaller than the destination tables, for instance, when transferring predicates from a dimension table like `supplier` to a fact table like `lineitem`. In such cases, broadcast-based approaches are likely to be more efficient than shuffle-based approaches.

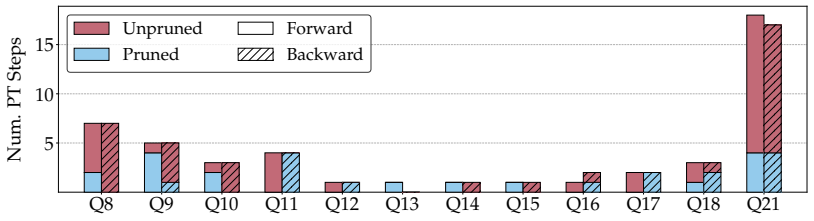
In the backward pass, `SHFL-BF-SRC` dominates the transfer steps. Transfers in the backward pass are typically from the fact tables, which are potentially large even though already reduced in the forward pass, to the dimension tables, which are small. `SHFL-BF-SRC` incurs the least potential network traffic in such scenarios.

As discussed, with the cluster size increasing, the efficiency of broadcast-based designs degrades. Therefore, fewer transfer steps select `BCAST-BF` as the running algorithm. In contrast, `SHFL-BF-DST` occurs more frequently, which avoids incurring expensive broadcast operations. For example, on a 4-node cluster, 6 out of 10 transfer steps in Q9 select `BCAST-BF`. However, this number decreases to 4 and 3 transfer steps on 8-node and 16-node clusters, respectively.

7.3.3 Pruning. We next study the effectiveness of pruning unnecessary transfer steps. We conduct experiments using cost-based adaptive distributed predicate transfer. We report performance and



(a) Execution Time (SF100, 4 nodes)



(b) Pruned PT Steps

Fig. 10. Amount of Transfer Step Pruning Occurred on TPC-H.

Table 4. Discovery of Join Key Containment on TPC-H.

Property	Referential	Reversed Referential	Self-Join
Forward	12	0	4
Backward	2	9	4

the number of pruned transfer steps on each query in Figure 10. Only queries in which pruning occurs are presented. Besides, we performed a sanity check to make sure that the pruned transfers do not affect the pre-filtering capability of PT—only unnecessary transfer steps are pruned.

Pruning happens on 12 of 20 TPC-H queries, which improves performance for most of the affected queries. The highest speedup occurs on Q21, which is improved by 21%. As Figure 10(b) shows, 8 transfer steps of 35 are pruned out in Q21, and most of them involve the fact table `lineitem`, which are usually costly.

As Figure 10(b) shows, overall 33% of the total predicate transfer steps are pruned, and both the forward and backward passes have roughly the same pruning ratio. More than half of the transfers are pruned in 8 out of all the queries. These queries typically only have one join table with local predicates. Therefore, only the transfers that sends these local predicates to all the other join tables are necessary—the corresponding reversed transfers are uncesessary.

One key property to satisfy for pruning is join key containment. Table 4 presents a breakdown on the amount of predicate transfer steps that satisfy join key containment. In the forward pass, join key containment is majorly satisfied by the defined referential constraints in the schema. In the backward pass, join key containment is mostly fulfilled by the discovered reversed containment of referential constraints. Besides, join key containment on Q21 is induced by self-joins.

The other key property to satisfy for pruning is predicate containment, and we observed three major scenarios in the experiments. First, in the forward pass, predicate containment is satisfied if both the source and destination tables have no predicates. Examples include Q8, Q9, Q10, and Q13. Second, in the backward pass, predicate containment is satisfied if both tables receive no

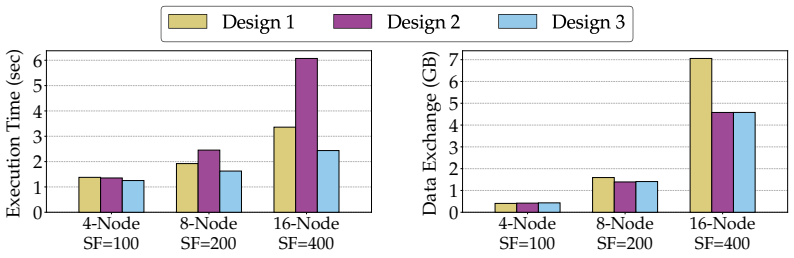


Fig. 11. TPC-H Performance of BCAST-BF Designs — GeoMean of performance across all measured queries.

additional predicates after the previously performed reversed transfer step in the forward pass—the destination table has already absorbed the predicates of the source table. Examples include Q9, Q11, Q12, and Q17. Finally, in some transfers, the predicates received by both tables are syntactically the same, which is reflected by Q21.

7.3.4 Design Variants of BCAST-BF. In this part, we evaluate variants of BCAST-BF proposed in Section 3.1.2. We conduct experiments under multiple scales of the cluster and data set. The results are shown in Figure 11.

As Figure 11(a) shows, Design 3 performs the best in all cases. When the cluster is large (i.e., 16 nodes), it outperforms Design 1 and Design 2 by 38% and 2.5× respectively; when the cluster is small (i.e., 4 nodes), it slightly outperforms the other two designs.

Figure 11(b) presents the amount of data exchange correspondingly. We observed that Design 2 and 3 incur the same amount of data exchange. However, in Design 2, the network traffic of partial Bloom filters and the merge process is all handled by a single node, which becomes the major performance bottleneck. Instead, Design 3 is able to evenly distribute both the network traffic and the merge computation to all the nodes.

Compared to Design 3, Design 1 achieves a comparable amount of data exchange on a small cluster, but incurs more network traffic on a large cluster. This is because Design 1 needs to increase the number of bits allocated for each element in the Bloom filter, to preserve the desired false positive rate. Besides, on a large cluster, Design 1 has to check quite a few Bloom filters to decide whether a row can be filtered out, causing additional computational overhead.

7.4 Case Study

In this subsection, we present a case study on representative queries. We first demonstrate the detailed filtering metrics (e.g., amount of input rows in the destination table, filtering selectivity, etc.) for each transfer step. Specifically, we analyze for TPC-H Q5 and Q10 (other queries have similar behaviors), and the results are presented in Table 5 and 6, respectively.

In Q5, most of the filtering occurs during the forward pass, where the selectivity is typically around or below 20%. The transfer from `supplier` to `lineitem` eliminates 474M rows and incurs an overhead of 359ms, which accounts for 39% of the total query execution time. The transfer from `orders` to `lineitem` is the most selective, pruning approximately 123M rows. Filtering in the backward pass is relatively less selective, but the associated operation overhead is also significantly lower. The size of the constructed Bloom filters remains no larger than 9MB for all steps.

In Q10, the forward pass processes only one transfer step, as the other two preceding steps are pruned by our transfer pruning algorithm. This step filters `lineitem` based on predicates from `orders`, eliminating 135M rows. The total overhead is 128ms, which accounts for 4% of the query

Table 5. Detailed Filtering Metrics of Individual Predicate Transfer Steps on TPC-H Q5 (SF100, 4 nodes).

Step		Algorithm	Filtering (DST table)		Bloom Filter		
			Input Rows	Sel.	Size (bytes)	Build (ms)	Probe (ms)
Forward	region → nation	BCAST-BF	25	20%	64	0.4	0.1
	nation → supplier	BCAST-BF	1M	20%	64	0.3	0.4
	supplier → customer	BCAST-BF	15M	20%	262K	0.4	5
	supplier → lineitem	BCAST-BF	600M	21%	262K	0.4	359
	customer → orders	BCAST-BF	23M	21%	4M	2	10
	orders → lineitem	BCAST-BF	128M	4%	8M	3	54
Backward	lineitem → orders	SHFL-BF-SRC	5M	58%	8M	3	4
	orders → customer	BCAST-BF	3M	48%	4M	1	2
	lineitem → supplier	BCAST-BF	200K	100%	9M	3	1
	customer → supplier	SHFL-BF-SRC	200K	100%	3M	1	1
	supplier → nation	SHFL-BF-SRC	4	100%	329K	0.6	0.3
	nation → region	SHFL-BF-SRC	1	100%	2K	2	0.3

Table 6. Detailed Filtering Metrics of Individual Predicate Transfer Steps on TPC-H Q10 (SF100, 4 nodes).

Step		Algorithm	Filtering (DST table)		Bloom Filter		
			Input Rows	Sel.	Size (bytes)	Build (ms)	Probe (ms)
Forward	nation → customer	PRUNED	⁄	⁄	⁄	⁄	⁄
	customer → orders	PRUNED	⁄	⁄	⁄	⁄	⁄
	orders → lineitem	BCAST-BF	148M	9%	8M	5	123
Backward	lineitem → orders	SHFL-BF-SRC	6M	86%	42M	19	38
	orders → customer	BCAST-BF	15M	26%	8M	6	46
	customer → nation	SHFL-BF-SRC	25	100%	9M	4	1

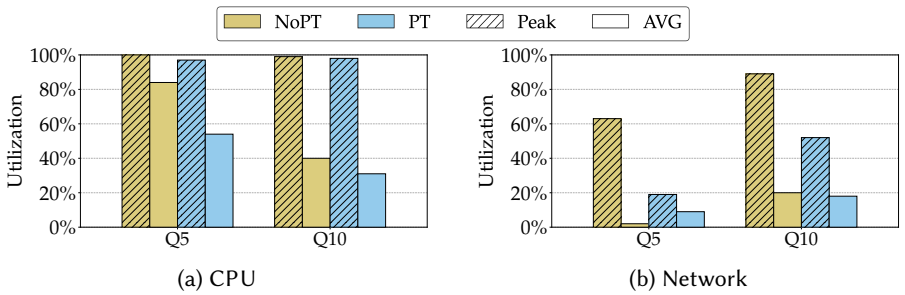
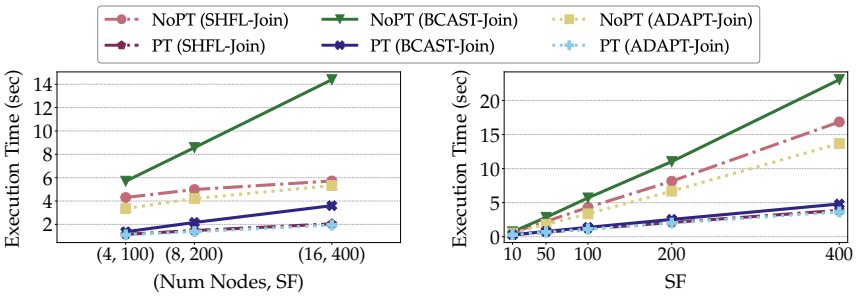


Fig. 12. Resource Utilization on Representative Queries of TPC-H (SF100, 4 nodes).

execution time. The backward pass removes 12M rows in total, but constructs larger Bloom filters due to the relatively large size of the source table (i.e., `lineitem`).

We observed that some steps in the backward pass do not filter out any rows, even after applying our transfer pruning algorithm, for example, the transfer from `lineitem` to `supplier` in Q5. These transfers are typically lightweight with negligible overhead. As a future work, we could potentially design an adaptive mechanism to complete the backward pass and begin the join phase earlier.

We further analyze CPU and network resource utilization for the two representative queries, as shown in Figure 12. In both cases, CPU utilization is higher than network utilization. In Q5, we



(a) Cluster Scaling (b) Data Scaling (4 nodes)

Fig. 13. Scalability of Distributed Predicate Transfer on TPC-H — GeoMean of performance across all measured queries.

observed that with PT, CPU utilization decreases while network utilization increases. This occurs because PT significantly reduces the computational cost of joins, shifting the bottleneck to the network. In Q10, PT reduces both computation and network traffic to a similar extent, resulting in comparable CPU and network utilization.

7.5 Scalability

In this subsection, we investigate the scalability of distributed predicate transfer. Specifically, we conduct experiments with cluster scaling—increase the node count and the data size proportionally, and data scaling—increase the data size with the fixed node count. In the join phase, we adopt multiple distributed join strategies.

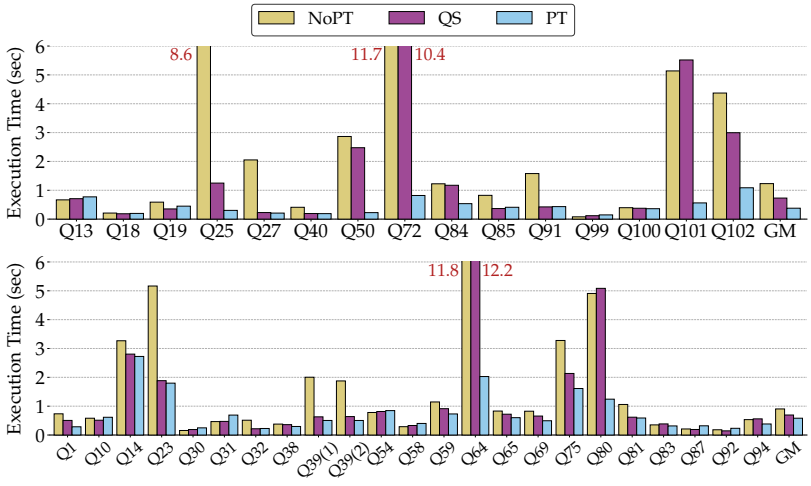
As Figure 13(a) shows, when using shuffled distributed joins and adaptive distributed joins, both PT and NoPT exhibit good scalability with cluster scaling, while the executions using broadcast distributed joins are not as scalable. This is because broadcast joins incur linear amount of data exchange with respect to the cluster size. The same phenomenon happens on broadcast-based distributed PT. However, since the predicate transfer phase is much more lightweight, the overall scalability of PT can still be preserved.

As Figure 13(b) shows, both PT and NoPT exhibit superior scalability with data scaling on all types of distributed joins. With the data size growing, the speedup of PT over NoPT becomes more significant. For instance, with adaptive distributed joins, PT outperforms NoPT by 3×, 3.3×, and 3.8× on the SF100, SF200, and SF400 data sets, respectively. PT can effectively avoid producing large intermediates, of which the performance benefit may scale beyond the data size. Furthermore, even on smaller-scale data sets, PT demonstrates considerable performance gains. On the SF10 and SF50 data sets, PT outperforms NoPT by 73% and 2.8×, respectively, when adopting adaptive distributed joins. However, these improvements are relatively smaller compared to larger data sets. On smaller data sets, the bottleneck from distributed joins is less severe, while the fixed overhead of network round-trip latencies is more pronounced.

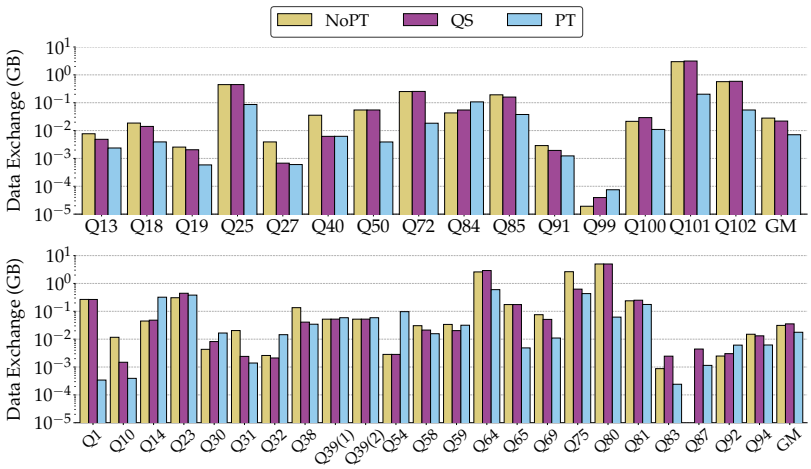
We have also evaluated the scalability of predicate transfer on a single multi-core machine, where we observed that PT and NoPT are both scalable on multiple cores. PT is slightly less scalable compared to NoPT since building a Bloom filter breaks the execution pipeline. We omit the details due to space constraints.

7.6 More Complex Workloads

To thoroughly evaluate the proposed techniques, we further assessed them using the DSB benchmark, which includes more complex workloads and advanced data distribution mechanisms, such



(a) Execution Time (single-block queries on the top; multi-block queries below)



(b) Data Exchange (single-block queries on the top; multi-block queries below)

Fig. 14. DSB Performance of Distributed Predicate Transfer (SF100, 4 nodes).

as skewed data and column correlations. We use the cost-based adaptive execution and apply transfer step pruning for PT.

Figure 14(a) compares query execution time of PT against NoPT and QS. On average, PT outperforms NoPT by 3.3× for single-block queries and 55% for multi-block queries, and surpasses QS by 93% and 19% for the two types of queries respectively. Due to the effective pre-filtering, PT achieves an average reduction of 10× of the joining table sizes, whereas QS also reduces the input size by 2×.

Out of the total 37 queries, 27 benefit from PT's optimization. Among these, 5 queries experience a speedup of more than 5× over both NoPT and QS. The largest speedups of PT are observed on Q25 and Q72, which are 28× and 14× over NoPT respectively. Such queries have large complex join graphs and as a result, the input tables are reduced by a factor of 937× and 177× respectively.

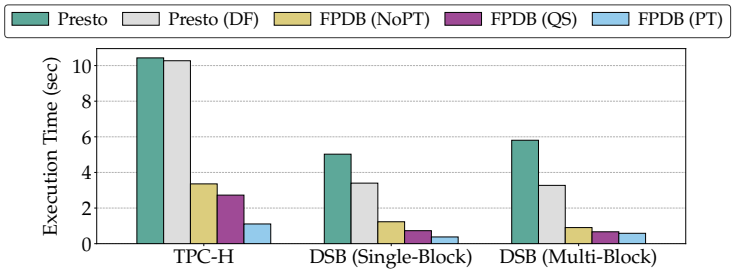


Fig. 15. Performance Comparison of FPDB against Presto — GeoMean across all measured queries (SF100, 4 nodes); Presto (DF) represents Presto execution with "dynamic filtering" enabled.

Figure 14(b) presents data exchange volumes across different distributed pre-filtering solutions, in log-scale. Overall, PT reduces network traffic by 4× and 3.1× compared to NoPT and QS on single-block queries, and by 43% and 2× on multi-block queries, respectively. However, further profiling reveals that the network resources are not saturated, indicating that network traffic is not the major bottleneck in DSB queries. Due to the skews introduced in the datasets, most joins are highly selective and produce either extremely small or no intermediate results (e.g., Q30, Q83, Q87, Q99, etc.). Such small queries benefit little from PT, and the overhead from scheduling and coordination during Bloom filter transfers and merges may outweigh the benefits.

7.7 Comparison to Other Existing Systems

We further compare the performance against Presto [52, 54], a widely used open-source distributed data warehouse. We use the latest version of Presto (v0.290), preload data into the memory of the executor nodes through the built-in Alluxio SDK cache [47], and perform "analyze" for each table to collect column statistics before the evaluation. Experiments are conducted on a 4-node cluster using SF100 data sets of both TPC-H and DSB. Presto provides a LIP-style join optimization called "dynamic filtering" [46] which we also incorporate in the evaluation.

As shown in Figure 15, NoPT of FPDB outperforms vanilla Presto by 3.1×, 4.1×, and 6.4× on the TPC-H benchmark, DSB single-block queries, and DSB multi-block queries, respectively. Even when dynamic filtering is enabled for Presto, NoPT of FPDB still achieves a speedup of over 2.8× across all benchmarks.

With PT, FPDB delivers even greater performance gains over Presto. Specifically, PT of FPDB surpasses vanilla Presto by 9.4×, 13.1×, and 10× on TPC-H, DSB single-block queries, and DSB multi-block queries, respectively. Besides, it outperforms Presto with dynamic filtering by 9.3×, 9×, and 5.6× on these benchmarks. These results underscore the efficiency of our testbed system.

Dynamic filtering enhances Presto's performance on DSB benchmarks, achieving speedups of 48% and 77% for single-block and multi-block queries, respectively. However, it has limited impact on TPC-H due to constraints in Presto's dynamic filtering mechanism, such as the 10KB Bloom filter size limit, which is often exceeded by TPC-H's larger dimension tables. Notably, we observed similar speedups from LIP-style techniques on both Presto and FPDB. Given that PT already significantly outperforms LIP-style techniques on FPDB, we anticipate substantial performance improvement if PT were to be integrated into Presto in the future.

8 Related Work

Sideways Information Passing (SIP). SIP techniques accelerate join queries by propagating filters from one joining table to another before the actual join is executed. Bloom Joins [15, 35, 49] are one common practice of SIP, and can be regarded as a special case of predicate transfer with

one hop and one direction. Magic Sets [6, 40, 41] leverage SIP [9] to perform query rewriting, such that irrelevant rows are filtered out early. QuickStep [43] and LIP [66] generalized Bloom Joins to star schemas, and adaptively reorders the Bloom filters when applying to the fact table. Bitvector Filters [26] explored the plan space of right-deep trees to optimize star and snowflake queries. [12] splits join operations into Lookup and Expand sub-operators, with the former performing the pre-filtering. However, these techniques are designed for single-node environments and are confined to pre-filtering within a portion of the input join graph, while distributed predicate transfer can address these limitations.

Data-induced predicates (diPs) [33] offered a pre-filtering technique applied at query planning time, using lightweight coarse-grained min-max or range-set filters. diPs are typically effective when tables are sorted by predicate columns or join keys, and when predicate columns and join keys are correlated. Compared to diPs, distributed predicate transfer pre-filters tables at runtime before the actual joins, offering greater filtering power and selectivity.

Distributed Semi-Join Reduction. Semi-join reduction [10] is a well-known technique for efficiently pre-filtering the joining tables. The Yannakakis algorithm [64] performs semi-join reduction prior to the actual join, ensuring optimal theoretical pre-filtering for acyclic queries. Predicate transfer [63] adapted the Yannakakis algorithm for practical use by leveraging Bloom filters, which forms the foundation of the techniques presented in this paper.

Semi-join reduction has also been explored in distributed settings [10, 11, 17–19, 21, 29, 32, 37, 45, 53, 60]. However, most of these studies focused on the scenario where the input relations are not partitioned—each relation is stored on a single node. In this case, the semi-join reduction involves a sequence of semi-join operations, where each operation transmits the join keys from the source relation to the node hosting the destination relation, enabling the filtering of non-matching rows in the destination relation.

[59, 65] discussed distributed semi-join reduction in scenarios where tables are horizontally partitioned. These solutions are similar to BCAST-VAL distributed predicate transfer proposed in this paper, but reduce the destination table using semi-joins instead of Bloom filters, and thus are less efficient compared to BCAST-VAL. TrackJoin [44] optimized distributed semi-join reduction by adaptively scheduling the broadcast process based on data placement. However, unlike distributed predicate transfer, it transmits entire tuples rather than join keys or more compact Bloom filters.

Distributed Joins. Joins on distributed settings have been studied for decades. Broadcast-based distributed joins and shuffle-based distributed joins [28] are the most prominent algorithms in modern distributed query engines [3, 5, 22, 48]. [8] compared radix hash join and sort-merge join algorithms and discussed their implementation at a large scale. [16, 51] optimized distributed joins over skewed data partitions. SquirrelJoin [50] studied distributed joins on heterogeneous network topologies. [39] proposed optimized data placement to accelerate distribute joins. Predicate transfer in distributed settings draws on key principles from broadcast-based and shuffle-based joins, accelerating distributed joins by reducing the size of joining tables.

9 Conclusion

Predicate transfer is the state-of-the-art pre-filtering approach for join queries, which was inspired by the Yannakakis algorithm and generalized Bloom joins to transfer table-local filters to multiple other tables. In this paper, we addressed limitations of the existing predicate transfer framework, which includes extending predicate transfer into general distributed settings, and pruning ineffective predicate transfer steps. Our evaluation on standard OLAP benchmarks showed that distributed predicate transfer can accelerate distributed query execution by more than 3×, and reduce the amount of data exchange by over 2.7×. This work was supported in part by NSF award IIS-2144588.

References

[1] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable bloom filters. *Inform. Process. Lett.* 101, 6 (2007), 255–261.

[2] Amazon Web Services. 2006. Amazon EC2. https://aws.amazon.com/ec2/

[3] Amazon Web Services. 2024. Data Distribution and Redistribution. https://docs.aws.amazon.com/redshift/latest/dg/c_data_redistribution.html.

[4] Apache Software Foundation. 2016. Apache Arrow. https://arrow.apache.org/

[5] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1383–1394.

[6] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. 1–15.

[7] 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.

[8] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed join algorithms on thousands of cores. *Proceedings of the VLDB Endowment* 10, 5 (2017), 517–528.

[9] C. Beeri and R. Ramakrishnan. 1987. On the power of magic. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 269–284.

[10] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (1981), 25–40.

[11] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie. 1981. Query processing in a system for distributed databases (SDD-1). *ACM Trans. Database Syst.* 6, 4 (1981), 602–625.

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

[13] 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.

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

[15] Kjell Bratbergsengen. 1984. Hashing Methods and Relational Algebra Operations. In *Proceedings of the 10th International Conference on Very Large Data Bases*. 323–333.

[16] Nicolas Bruno, YongChul Kwon, and Ming-Chuan Wu. 2014. Advanced join strategies for large-scale distributed computation. *Proc. VLDB Endow.* 7, 13 (2014), 1484–1495.

[17] Chen and Li. 1984. Improvement Algorithms for Semijoin Query Processing Programs in Distributed Database Systems. *IEEE Trans. Comput.* C-33, 11 (1984), 959–967.

[18] M.-S. Chen and P.S. Yu. 1993. Combining joint and semi-join operations for distributed query processing. *IEEE Transactions on Knowledge and Data Engineering* 5, 3 (1993), 534–542.

[19] Ming-Syan Chen and P.S. Yu. 1994. A graph theoretical approach to determine a join reducer sequence in distributed query processing. *IEEE Transactions on Knowledge and Data Engineering* 6, 1 (1994), 152–165.

[20] 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.

[21] Dah Ming Chiu and Yu-Chi Ho. 1980. A methodology for interpreting tree queries into optimal semi-join expressions. In *Proceedings of the 1980 ACM SIGMOD international conference on Management of data*. 169–178.

[22] Google Cloud. 2023. BigQuery Admin Reference Guide: Query Optimization. https://cloud.google.com/blog/topics/developers-practitioners/bigquery-admin-reference-guide-query-optimization.

[23] Apache Doris Community. 2022. TPC-H Benchmark on Doris. https://doris.apache.org/zh-CN/blog/tpch/.

[24] StarRocks Community. 2024. TPC-H Benchmarking on StarRocks. https://docs.starrocks.io/docs/benchmarking/TPC-H_Benchmarking/.

[25] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: a decision support benchmark for workload-driven and traditional database systems. *Proc. VLDB Endow.* 14, 13 (2021), 3376–3388.

[26] Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. 2020. Bitvector-aware Query Optimization for Decision Support Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2011–2026.

[27] 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.

[28] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2nd ed.). Prentice Hall, Upper Saddle River, NJ.

[29] Mohamed G Gouda and Umeshwar Dayal. 1981. Optimal semijoin schedules for query processing in local distributed database systems. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 164–175.
[30] Joseph M. Hellerstein. 1998. Optimization Techniques for Queries with Expensive Methods. *ACM Trans. Database Syst.* 23, 2 (1998), 113–157.
[31] 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.
[32] Yahiko Kambayashi, Masatoshi Yoshikawa, and Shuzo Yajima. 1982. Query processing for distributed databases using generalized semi-joins. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*. 151–160.
[33] 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.
[34] 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.
[35] Paraschos Koutris. 2011. Bloom Filters in Distributed Query Execution. *Principles of DBMS - University of Washington* (2011).
[36] Alon Y Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. 1994. Query optimization by predicate move-around. In *VLDB*. 96–107.
[37] Xuemin Lin, M.E. Orlowskat, and Xiaofang Zhou. 1995. Using parallel semi-join reduction to minimize distributed query response time. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, Vol. 2. 517–526 vol.2.
[38] 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.
[39] Manish Mehta and David J DeWitt. 1997. Data placement in shared-nothing parallel database systems. *The VLDB journal* 6, 1 (1997), 53–72.
[40] Inderpal Singh Mumick, Sheldon J Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. Magic conditions. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 314–330.
[41] I. S. Mumick, S. J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. Magic is relevant. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. 247–258.
[42] Anna Pagh, Rasmus Pagh, and S Srinivasa Rao. 2005. An optimal Bloom filter replacement.. In *Soda*. 823–829.
[43] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: a data platform based on the scaling-up approach. *Proc. VLDB Endow.* 11, 6 (2018), 663–676.
[44] Orestis Polychroniou, Rajkumar Sen, and Kenneth A. Ross. 2014. Track join: distributed joins with minimal network traffic. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 1483–1494.
[45] Sakti Pramanik and David Vineyard. 1987. Optimizing join queries in distributed databases. In *Foundations of Software Technology and Theoretical Computer Science: Seventh Conference, Pune, India December 17–19, 1987 Proceedings 7*. 282–304.
[46] PrestoDB. 2020. Presto Release 0.241. https://prestodb.io/docs/current/release/release-0.241.html
[47] PrestoDB. 2023. Presto Documentation: Alluxio SDK Cache. https://prestodb.io/docs/current/cache/local.html
[48] PrestoDB. 2024. Presto Documentation: Properties. https://prestodb.io/docs/current/admin/properties.html.
[49] 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*. 145–156.
[50] Lukas Rupprecht, William Culhane, and Peter Pietzuch. 2017. SquirrelJoin: network-aware distributed join processing with lazy partitioning. *Proc. VLDB Endow.* 10, 11 (2017), 1250–1261.
[51] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. 2016. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 1194–1205.
[52] 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. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813.
[53] K. Stocker, D. Kossmann, R. Braumandi, and A. Kemper. 2001. Integrating semi-join-reducers into state-of-the-art query processors. In *Proceedings 17th International Conference on Data Engineering*. 575–584.
[54] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, Nikhil Collooru, Ke Wang, Sameer Agarwal, Arjun Gupta, Dionysios Logothetis, Kostas Xirogiannopoulos, Amit Dutta, Varun Gajjala, Rohit Jain, Ajay Palakuzhy, Prithvi Pandian, Sergey Pershin,

Abhisek Saikia, Pranjal Shankhdhar, Neerad Somanchi, Swapnil Tailor, Jialiang Tan, Sreeni Viswanadha, Zac Wen, Biswapesh Chattopadhyay, Bin Fan, Deepak Majeti, and Aditi Pandit. 2023. Presto: A Decade of SQL Analytics at Meta. *Proc. ACM Manag. Data* 1, 2 (2023).

[55] 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.

[56] Transaction Processing Performance Council. 1999. TPC-H Benchmark. https://www.tpc.org/tpch/

[57] Transaction Processing Performance Council. 2005. TPC-DS Benchmark. https://www.tpc.org/tpcds/

[58] Jeffrey D. Ullman. 1988. *Principles of Database and Knowledge-Base Systems, Vol. I.* Computer Science Press, Inc.

[59] Patrick Valduriez. 1982. Semi-join algorithms for multiprocessor systems. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*. 225–233.

[60] C. Wang, V.O.K. Li, and A.L.P. Chen. 1991. Distributed query optimization by one-shot fixed-precision semi-join execution . In *Proceedings. Seventh International Conference on Data Engineering*. 756–763.

[61] 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.

[62] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2024. FlexpushdownDB: rethinking computation pushdown for cloud OLAP DBMSs. *The VLDB Journal* 33, 5 (2024), 1643–1670.

[63] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[64] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*. 82–94.

[65] C. T. Yu and C. C. Chang. 1983. On the design of a query processing strategy in a distributed database environment. In *Proceedings of the 1983 ACM SIGMOD International Conference on Management of Data*. 30–39.

[66] 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 (2017), 889–900.

Received October 2024; revised January 2025; accepted February 2025