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

Abstract—**Predicate Transfer is a novel method that optimizes join performance by pre-filtering tables to reduce join input sizes. Inspired by the Yannakakis algorithm, which employs semi-joins to pre-filter acyclic queries, Predicate Transfer generalizes Bloom join to multi-table joins, significantly improving filtering benefits and supporting various join graph topologies. However, the current heuristics used in the predicate transfer phase may not always yield optimal results. This paper proposes an optimized predicate transfer algorithm that further enhances performance by incorporating transfer schedule cost estimation and pruning techniques. The algorithm introduces a metric called predicate effectiveness, which measures the ratio of tuples filtered out to tuples probed by a filter. By systematically estimating the cost of different transfer schedules and pruning ineffective predicates based on the effectiveness threshold, the algorithm selects an optimal transfer schedule that achieves faster performance and better adaptability to various query patterns and data characteristics.**

Keywords—database query, join optimization

I. Introduction

Join optimization is a critical aspect of database management systems, as join operations are among the most computationally expensive operations in a database query. Recent advancements in join optimization techniques have focused on reducing the input sizes of join operations, as this can significantly impact overall query performance. Predicate Transfer is a novel method that optimizes join performance by pre-filtering tables to reduce join input sizes. Inspired by the Yannakakis algorithm [28], which employs semi-joins to pre-filter acyclic queries, Predicate Transfer [32] generalizes Bloom join to multi-table joins, significantly improving filtering benefits and supporting various join graph topologies.

However, the current heuristics used in the predicate transfer phase of the Predicate Transfer algorithm may not always yield optimal results. This paper proposes an optimized predicate transfer algorithm that further enhances performance by incorporating transfer schedule cost estimation and pruning techniques. The optimized algorithm introduces a metric called predicate effectiveness, which measures the ratio of tuples filtered out to tuples probed by a filter. By systematically estimating the cost of different transfer schedules and pruning ineffective predicates based on the effectiveness threshold, the algorithm selects an optimal transfer schedule that achieves faster performance and better adaptability to various query patterns and data characteristics.

This paper is organized as follows: Section II provides background information on join optimization and related work in Bloom join and the Predicate Transfer algorithm. Section III presents the optimized predicate transfer algorithm, including cardinality and selectivity estimation, transfer schedule cost estimation, transfer schedule pruning, and an adaptive predicate transfer mechanism. Section IV presents the evaluation results, demonstrating the significant speedup achieved by the predicate transfer algorithm compared to baseline approaches. Finally, Section V concludes the paper and discusses future work.

II. Background and Related Work

This section presents the background of join optimization (Section A) and related work in Bloom Join (Section B), and Predicate Transfer Algorithm (Section C).

A. *Join Optimization*

Query optimization has been a critical aspect of database management systems (DBMS) for decades, focusing on improving the efficiency and performance of database queries [1]. The goal of query optimization is to find the most efficient execution plan for a given query, minimizing the time and resources required to retrieve the desired data. Various techniques have been proposed, ranging from rule-based optimization [2] that rely on a set of predefined rules to transform and optimize query execution plans, to cost-based optimization [3] that estimate the cost of different execution plans and select the plan with the lowest estimated cost and beyond. Rule-based optimization has been widely adopted in commercial DBMS due to its simplicity and effectiveness, and cost-based optimization has become the de facto standard in modern DBMS, as it provides a more accurate and adaptive approach compared to rule-based optimization.

Join operations are among the most computationally expensive operations in a database query, and optimizing joins is crucial for improving query performance [2]. Recent developments in query optimization have focused on leveraging machine learning techniques [5][6], adaptive optimization [7][8], hardware-aware strategies [9][10], and workload-driven approaches [11][12].

In the context of join optimization, recent research has focused on techniques for reducing the input sizes of join operations, as this can significantly impact the overall query performance. Chaudhuri and Shim [13] proposed "join predicate pushdown," which involves applying join predicates before executing the join operation. Kemper and Neumann [14] introduced "join-filtered scans" in the "HyPer" system, generating a compact join index containing only matching tuples. Schuh et al. [15] presented "join filtering" in the "Umbra" system, using Bloom filters to efficiently test tuple participation in the join result.

These advancements in join optimization techniques, particularly those focused on pre-filtering tables to reduce join input sizes, have shown promising results in improving the performance of database queries. As the volume and complexity of data continue to grow, the development and refinement of efficient join optimization strategies will remain an important area of research.

B. *Bloom Filter*

Bloom filter, a space-efficient probabilistic data structure used to test whether an element is a member of a set [16], was first introduced by Burton H. Bloom in 1970 [17]. Since then, it has found widespread application in various domains, including databases, network protocols, and cache management [18].

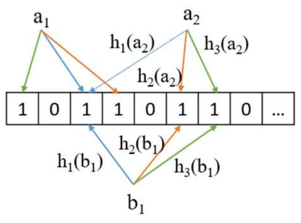

Fig. 1. Example of a bloom filter

A Bloom filter consists of a bit array of size *m* and *k* independent hash functions. To add an element to the set, the element is hashed using each of the k hash functions, and the corresponding bits in the array are set to 1. To test whether an element is a member of the set, the element is hashed using the same k hash functions, and the corresponding bits in the array are checked. If any of the bits are 0, the element is definitely not in the set. If all the bits are 1, the element is probably in the set, with a certain probability of a false positive [19]. The false positive probability can be controlled by adjusting the size of the bit array (*m*) and the number of hash functions (*k*) based on the expected number of elements (*n*) to be inserted [20]. The optimal number of hash functions for minimizing the false positive probability is given by [21]:

$$k = (m / n) * \ln 2 \tag{1}$$

Bloom join [22], a join algorithm that leverages Bloom filters to optimize the performance of join operations, consists of the following steps:

Build. Create a Bloom filter for one of the join relations (typically the smaller relation). Insert all the join keys from this relation into the Bloom filter.

Broadcast. Distribute the Bloom filter to all the nodes that hold partitions of the other join relation.

Probe. For each tuple in the other join relation, probe the Bloom filter to check if its join key might be present in the first relation. If the Bloom filter returns a positive result (i.e., the key might be present), the tuple is kept for further processing. If the Bloom filter returns a negative result (i.e., the key is definitely not present), the tuple is discarded.

Join. Perform the actual join operation between the filtered tuples from the second relation and the tuples from the first relation. This step can be done using any join algorithm, such as hash join or sort-merge join.

Modern Online Analytical Processing (OLAP) DBMSs, including Oracle [23], Amazon Redshift [24], Snowflake [25], and Databricks [26], have widely adopted Bloom filters to accelerate join execution, particularly for large-scale analytical workloads involving complex join operations.

However, most existing Bloom join algorithms are limited to a single join operation, meaning that the predicate on one table can only be used to pre-filter rows in the other table it joins with. In other words, the predicate is transferred in one-hop and one-direction. Some prior work [27] has extended the idea to datasets with star schemas, allowing all dimension tables to transfer local predicates to the fact table, outperforming the baseline Bloom join. Nevertheless, these solutions do not generalize to more complex query plans, such as multi-hop joins, cyclic joins, snowflake schemas, etc., leaving room for further optimization and generalization of Bloom join algorithms. To address these limitations, we propose the Predicate Transfer algorithm, which generalizes the pre-filtering technique across multiple joins and supports various join graph topologies.

C. Predicate Transfer

Our prior work of Predicate Transfer algorithm [32] is a generalization of the pre-filtering technique across multiple joins, inspired by the Yannakakis algorithm [28]. The Yannakakis algorithm efficiently evaluates acyclic join queries by decomposing the query graph into a join tree, performing a bottom-up phase to compute intermediate results, and then propagating the results top-down to obtain the final join result. This approach minimizes the computation of unnecessary joins and provides an efficient way to process acyclic queries with high join selectivity.

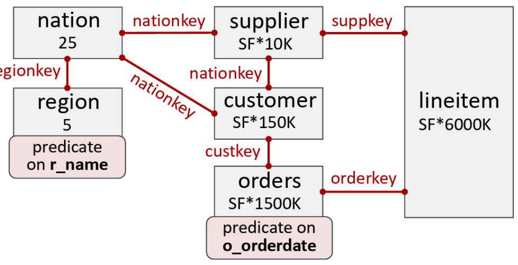

Fig. 2. Example of a Join Graph

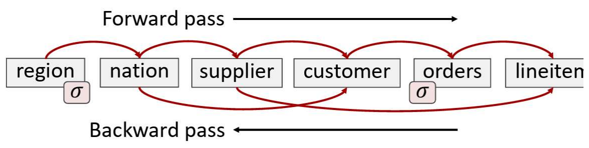

Fig. 3. Example of a Predicate Transfer Graph

The Predicate Transfer algorithm extends this concept by replacing the semi-join operations in the Yannakakis algorithm with faster Bloom joins. The algorithm proceeds in two phases:

- **Predicate Transfer phase.** In this phase, local predicates are constructed as filters (e.g., Bloom filters), transferred across the join graph, and removes all redundant tuples in the join graph through a *forward pass* and a *backward pass*. The forward pass starts from the leaf nodes of the decomposed join tree and computes the intermediate results for each subtree. For each subtree, the algorithm computes the join of the relations in the subtree and projects the result onto the attributes that are needed for joining with the parent node. This process is repeated recursively until the root node is reached. The backward pass start from the root node and propagates the intermediate results down the join tree in similar way, until the leaf nodes are reached. After this phase, the actual input of each join will be substantially smaller if the transferred filters are selective.

- **Join phase.** In this phase, the filtered tables can be joined (e.g., by hash join) in any order without any intermediate table size blow-up over the output size. It can be proven that regardless of the chosen join order, the join phase can be executed in the same time complexity.

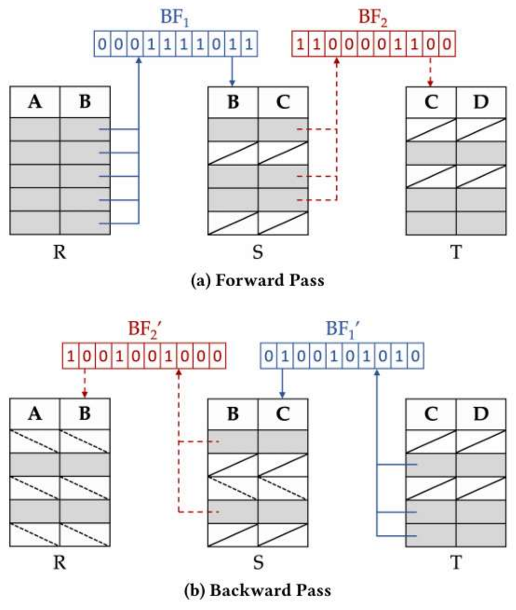

Fig. 4. Example of PredicateTransfer on a Join Query

Compared to the Yannakakis algorithm, the Predicate Transfer algorithm does not provide theoretical optimality, but it offers more versatility:

- It supports both precise filters (like semi-join) and Bloom filters.
- It can handle any join-graph topology, including outer joins and cyclic queries.
- It supports more operators and complex predicate transfer schedules.

This approach maintains near-maximum filtering capabilities at the predicate transfer phase and near-perfect robustness in the join phase by allowing the joins to be executed in any order without significant impact on performance. However, our current heuristics used to implement the predicate transfer phase are largely based on intuition and may not always yield optimal results. While these heuristics provide a solid foundation, there exists a vast design space for further optimizing the predicate transfer phase.

This paper proposed the optimized predicate transfer that further improves the performance by adopting transfer schedule cost estimation and pruning algorithm.

III. Optimized Predicate Transfer

This section presents the optimized predicate transfer algorithm, which enhances the original Predicate Transfer algorithm by incorporating cost estimation, schedule pruning, and adaptive techniques. The key components of the optimized predicate transfer include cardinality and selectivity estimation (Section B), transfer schedule cost estimation (Section C), transfer schedule pruning (Section D), and an adaptive predicate transfer mechanism.

A. Overview

The optimized predicate transfer further implement the predicate transfer phase in the following steps:

- **Cardinality and selectivity estimation.** Cardinality estimation techniques (e.g. histograms and sampling) and selectivity estimation methods (e.g. join sampling and learned models) can be used to get the metadata for the tables in the join graph.
- **Transfer Schedule cost estimation.**
- **Transfer Schedule pruning.** Based on the cardinality and selectivity estimates, the adaptive algorithm prunes the space of possible transfer schedules. Then compare estimated cost of pruned schedules to select the optimal one.
- **Optimized Predicate Transfer.** The selected optimal transfer schedule is executed to transfer the filter across the join graph.

In the next subsections, we delve into the details of each component, explaining their roles in the optimized predicate transfer algorithm and how they contribute to improved performance and adaptability.

B. Cardinality and Selectivity Estimation

Various techniques can be employed, such as histograms [28], sampling [29], machine learning models [30], considering the specific requirements, available resources, and desired accuracy of the estimates. The estimated metadata are then utilized in the subsequent steps of the algorithm.

C. Transfer Schedule cost estimation

With the runtime pre-filtering and post-filtering cardinality and selectivity for each table maintained, we can estimate the overhead of predicate transfer schedule by

$$Overheads = \sum Cost_{create} + \sum Cost_{probe} \qquad (2)$$

With each predicate transfer that create a bloom filter and use it probe another table, the total overhead cost is the sum of the creating cost and probing cost.

Since the predicate is transferring across the join graph, the cost is calculated based on runtime cardinality and selectivity estimation of tables.

When the table creating the bloom filter holds a foreign key in relation to the probing table, the original cardinality of the creating table might be larger than the original cardinality of the probing table. In this case, a record with primary key in probing table is filtered out only when all corresponding foreign key records in creating table is filtered out. Assuming that, we can estimate the selectivity by

$$\rho = 1 - (1 - \sigma)^{\frac{|Table_{create}|}{|Table_{probe}|}} \qquad (2)$$

D. Transfer Schedule Pruning

In the following section, we describe the pruning algorithm for optimizing the transfer schedule in the predicate transfer phase.

During the predicate transfer process, some transfers may not significantly increase filter selectivity but still consume computational resources. For example, consider a scenario where the selectivity of the predicate generated by table "Part" is 1, indicating that it has no effect on filtering tuples from another table. In such cases, transferring this predicate is unnecessary and can be eliminated from the transfer schedule.

To identify redundant predicates, we introduce a metric called predicate effectiveness. The effectiveness of a predicate is defined as the ratio of the number of tuples filtered out to the number of tuples probed by the filter. If the effectiveness metric falls below a pre-defined threshold, the predicate is considered ineffective and can be removed from the transfer schedule to improve the performance of the predicate transfer phase.

The pruning algorithm estimates the cost and effectiveness of each predicate in a Bloom filter-based transfer schedule. It takes as input the initial transfer schedule, along with the cardinality and selectivity estimates for each table. The algorithm then calculates the effectiveness of each predicate transfer by considering the number of tuples filtered out and the number of tuples probed. Based on the effectiveness metric, the algorithm identifies and removes predicates that fall below the specified threshold. By pruning these less effective filters from the full schedule, the algorithm generates an optimized transfer schedule that achieves faster performance while maintaining the desired level of filtering.

TABLE I. SUMMARY OF NOTATION USED

Notation	Meaning
S	transfer schedule
$d_1, \dots, d_N$	runtime cardinality for each table
$\sigma_1, \dots, \sigma_N$	runtime selectivity for each table
$\rho_1, \dots, \rho_N$	runtime selectivity for each table (after predicate transfer)

```
Algorithm: Transfer Schedule Pruning
Input:
        schedule – an array of table pair
    ($Table_{create}, Table_{probe}$), which are binary relations
    between tables
        S – an array of selectivity, $\sigma_1, \dots, \sigma_N$ for each table
        D – an array of cardinality, $|Table_1|, \dots, |Table_N|$
    for each table
Output:
        result – schedule after pruning
        filter num – number of tuples filtering out
        create cost – cost of creating bloom filters
        probe cost – cost of probing tables with bloom filters

result ← $\emptyset$
filter num ← 0
create cost ← 0
probe cost ← 0
$(\rho_1, \dots, \rho_N) \leftarrow (\sigma_1, \dots, \sigma_N)$
// an array of selectivity after filtering for each table
$(d_1, \dots, d_N) \leftarrow (\sigma_1, \dots, \sigma_N) \cdot (|Table_1|, \dots, |Table_N|)$
// an array of cardinality after filtering for each table
```

```
Algorithm: Transfer Schedule Pruning
foreach pair in schedule do
    create cost ← create cost + $d_{create}$
    probe cost ← probe cost + $d_{probe}$
    selectivity ← $\rho_{create}$
    if $Table_{create}$ had been filtered by $Table_{probe}$ then
        selectivity ← $\frac{selectivity}{the\ selectivity\ filtered\ on\ Table_{create}\ by\ Table_{probe}}$
    if $Table_{create}$ serves as a foreign key in relation to $Table_{probe}$ then
        selectivity ← $1 - (1 - selectivity)^{\frac{|Table_{create}|}{|Table_{probe}|}}$

    filter num ← filter num + (1 – selectivity)
    $\rho_{probe} \leftarrow \rho_{probe} * selectivity$
    $d_{probe} \leftarrow d_{probe} * selectivity$

if this filter is effectful then
    result ← result + pair

return result, filter num, create cost, probe cost
```

By considering the effectiveness of each predicate and pruning accordingly, the algorithm ensures that the predicate transfer phase operates efficiently, minimizing unnecessary computations and maximizing the benefits of filtering.

E. Optimized Predicate Transfer

In this step, the optimal transfer schedule is selected to perform the predicate transfer. Compared to Predicate Transfer algorithm, several improvements are as follows:

1) Bloom filter parameter tuning: The efficiency of the predicate transfer phase heavily relies on the proper configuration of Bloom filter parameters, which are the size of the filter and the number of hash functions used. We dynamically adapting these parameters based on the cardinality estimation for tables, determining the parameters with eqation (1), which is proved to be the optimal configuration for a single bloom filter.

2) Transfer schedule optimization: The optimized transfer schedule pruning algorithm consider factors such as join selectivity, data skew, and the structure of the join graph, thus further enhance the efficiency of the predicate transfer phase.

3) Adaptive predicate transfer: The algorithm dynamically adjust the filtering strategy based on runtime statistics, which could help optimize performance in various scenarios. By continuously monitoring the effectiveness of the predicate transfer phase and making informed decisions based on collected metrics, the algorithm could self-tune and adapt to changing data characteristics and query patterns.

IV. EVALUATION

This section presents our preliminary evaluation results for Predicate Transfer algorithm, and significant speedup for Optimized Predicate Transfer algorithm has been observed on TPC-H case study.

The experiment [32] for figure. 5. is conducted on a single AWS EC2 r5.4xlarge instance, with 16vCPU and 128GB memory. The server runs the Ubuntu 20.04 operating system. The widely adopted data analytics benchmark TPC-H with 22 queries in total is used. Both an 1GB data set (a scale factor of 1) and a 10GB data set (a scale factor of 10) are used. Queries are executed on a single CPU core. The test bed is FlexPushdownDB [31], an open-source cloud-native OLAP DBMS.

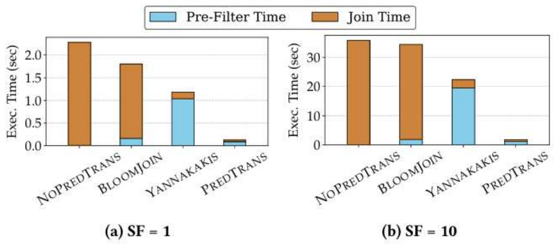

Fig. 5. Performance Breakdown on TPC-H Q5

The experiment compared the performance of our proposed join strategy, PredTrans, against three baseline approaches: NoPredTrans, BloomJoin, and Yannakakis. NoPredTrans represents the traditional approach where no predicate transfer occurs among joining tables, and pairs of tables are joined regularly as in most DBMSs. BloomJoin employs one-hop predicate transfer between joining table pairs, with the build side constructing a Bloom filter to filter the probe side. Yannakakis executes the semi-join phase of the Yannakakis algorithm before the join phase.

The results demonstrate that the PredTrans algorithm achieves significant speedup compared to the baseline approaches. In summary, the evaluation highlights the significant performance benefits of the optimized predicate transfer algorithm.

V. Conclusion

This paper presented an optimized predicate transfer algorithm that enhances the performance of the original Predicate Transfer algorithm by incorporating cost estimation, schedule pruning, and adaptive techniques. The key components of the optimized predicate transfer include cardinality and selectivity estimation, transfer schedule cost estimation, transfer schedule pruning, and an adaptive predicate transfer mechanism.

The optimized algorithm introduces a metric called predicate effectiveness, which measures the ratio of tuples filtered out to tuples probed by a filter. By systematically estimating the cost of different transfer schedules and pruning ineffective predicates based on the effectiveness threshold, the algorithm selects an optimal transfer schedule that achieves faster performance and better adaptability to various query patterns and data characteristics.

Future work includes conducting more extensive evaluations on larger datasets and more complex join queries, as well as exploring the integration of the optimized predicate transfer algorithm with other join optimization techniques, such as adaptive optimization and hardware-aware strategies. Additionally, investigating the applicability of the optimized algorithm to distributed and parallel database systems could further extend its impact and benefits in real-world scenarios.

References

[1] S. Chaudhuri, "An overview of query optimization in relational systems," in Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, 1998, pp. 34–43.

[2] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in Proceedings of the 1979 ACM SIGMOD international conference on Management of data, 1979, pp. 23–34.

[3] Y. E. Ioannidis and S. Christodoulakis, "On the propagation of errors in the size of join results," in Proceedings of the 1991 ACM SIGMOD international conference on Management of data, 1991, pp. 268–277.

[4] G. Graefe, "Query evaluation techniques for large databases," ACM Computing Surveys (CSUR), vol. 25, no. 2, pp. 73–169, 1993.

[5] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in Proceedings of the 2018 International Conference on Management of Data, 2018, pp. 489–504.

[6] R. Marcus, O. Papaemmanouil, and S. Semenova, "Towards a hands-free query optimizer through deep learning," arXiv preprint arXiv:1904.11194, 2019.

[7] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," in Proceedings of the 2000 ACM SIGMOD international conference on Management of data, 2000, pp. 261–272.

[8] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis, "SkinnerDB: Regret-bounded query evaluation via reinforcement learning," in Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 1153–1170.

[9] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake, "GPU-accelerated database systems: Survey and open challenges," in Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXVII, Springer, Berlin, Heidelberg, 2018, pp. 1–35.

[10] T. Karnagel, D. Habich, and W. Lehner, "Adaptive work placement for query processing on heterogeneous computing resources," Proceedings of the VLDB Endowment, vol. 10, no. 7, pp. 733–744, 2017.

[11] A. Pavlo et al., "Self-driving database management systems," in CIDR, 2017.

[12] J. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya, "AI meets AI: Leveraging query executions to improve index recommendations," in Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data, 2019, pp. 1241–1258.

[13] S. Chaudhuri and K. Shim, "Optimization of queries with user-defined predicates," ACM Transactions on Database Systems (TODS), vol. 24, no. 2, pp. 177–228, 1999.

[14] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in 2011 IEEE 27th International Conference on Data Engineering, 2011, pp. 195–206.

[15] S. Schuh, X. Chen, and J. Dittrich, "An experimental comparison of thirteen relational equi-joins in main memory," in Proceedings of the 2016 International Conference on Management of Data, 2016, pp. 1961–1976.

[16] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," Internet Mathematics, vol. 1, no. 4, pp. 485–509, 2004.

[17] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," Communications of the ACM, vol. 13, no. 7, pp. 422–426, 1970.

[18] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," IEEE Communications Surveys & Tutorials, vol. 14, no. 1, pp. 131–155, 2012.

[19] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," Random Structures & Algorithms, vol. 33, no. 2, pp. 187–218, 2008.

[20] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable bloom filters," Information Processing Letters, vol. 101, no. 6, pp. 255–261, 2007.

[21] S. Geravand and M. Ahmadi, "Bloom filter applications in network security: A state-of-the-art survey," Computer Networks, vol. 57, no. 18, pp. 4047–4064, 2013.

[22] L. L. Gremillion, "Designing a bloom filter for differential file access," Communications of the ACM, vol. 25, no. 9, pp. 600–604, 1982.

[23] "Bloom filters in Oracle," Oracle Database Documentation, [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/19/tgdba/bloom-filters.html

[24] "Tuning tables and queries - Amazon Redshift," AWS Documentation, [Online]. Available: https://docs.aws.amazon.com/redshift/latest/dg/c_tuning_tables_queries.html

[25] "Join optimization - Snowflake Documentation," Snowflake Documentation, [Online]. Available: https://docs.snowflake.com/en/user-guide/joins-optimizations.html

[26] "Bloom filter - Databricks," Databricks Documentation, [Online]. Available: https://docs.databricks.com/spark/latest/spark-sql/language-manual/functions/bloom_filter.html

[27] 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.

[28] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, "Improved histograms for selectivity estimation of range predicates," in Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, 1996, pp. 294-305.

[29] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes, "Sampling-based estimation of the number of distinct values of an attribute," in Proceedings of the 21st International Conference on Very Large Data Bases, 1995, pp. 311-322.

[30] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik, "Learning-based query performance modeling and prediction," in Proceedings of the 28th International Conference on Data Engineering, 2012, pp. 390-401.

[31] 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.

[32] Y. Yang, H. Zhao, X. Yu, and P. Koutris, "Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries," arXiv preprint arXiv:2307.15255, 2023.