

High-Performance OLAP Across the Stack: From Algorithmic Data Reduction to Cloud- and GPU-Native Execution Engines

by

Yifei Yang

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2026

Date of final oral examination: 01/09/2026

The dissertation is approved by the following members of the Final Oral Committee:

Xiangyao Yu, Assistant Professor, Computer Sciences

Paraschos Koutris, Associate Professor, Computer Sciences

Ming Liu, Assistant Professor, Computer Sciences

Tsung-Wei Huang, Associate Professor, Electrical and Computer Engineering

To all the kind souls who walked alongside me on this journey

Acknowledgments

Completing this PhD would have been impossible without the guidance and support of many people, to whom I am deeply grateful.

First and foremost, I would like to express my sincere gratitude to my advisor, Xiangyao Yu, whose mentorship shaped not only my research but also my approach to problem-solving and learning. His critical insights, combined with patience and encouragement, have been invaluable throughout this journey. I deeply appreciate the effort he devoted to providing detailed feedback on every piece of my work — whether a paper, a presentation, or my approach to thinking through and communicating ideas — which has influenced both my research and personal growth in many ways. I could not have wished for a better advisor.

I also want to thank the members of my dissertation committee, Paris Koutris, Ming Liu, and Tsung-Wei Huang, for serving on my committee and for their helpful comments during my examinations and dissertation defense.

Working alongside my colleagues at UW–Madison was both inspiring and enjoyable. I am grateful to my research collaborators — Bobbi Yogatama, Hangdong Zhao, Kevin Kristensen, Devesh Sarda, Abigale Kim, and Simon Frisk — for many productive discussions and collaborative efforts that greatly enriched my research experience. I would also like to give special thanks to Paris Koutris for his guidance and thoughtful advice throughout our collaborations, which helped me grow both professionally and academically.

Beyond Madison, I have been fortunate to collaborate with researchers from other institutions. I would like to thank Marco Serafini, Ashraf Aboulnaga, Huanchen Zhang, Yu Teng, Joshua Patterson, Gregory Kimball, Yujun He, Matt Youill, Matthew Woicik, Junyi Zhao, and Kai Su for their insights, collaboration, and generous sharing of ideas.

I am especially grateful to Michael Stonebraker for his time, guidance, and gen-

erosity in sharing his perspective. His profound influence on the database field and his thoughtful advice provided invaluable context and inspiration for my own work.

During my internships, I benefited greatly from the guidance and support of many people in industry. I would like to thank Sanket Hase, Chunbin Lin, and Orestis Polychroniou for their mentorship during my internship at Amazon Web Services, as well as Zhaozhe Song and Pavan Edara for their support and guidance during my internship at Google. These experiences provided valuable professional perspectives and contributed meaningfully to my growth as an engineer.

I am also thankful for the supportive and friendly environment created by my labmates and friends at Madison. In particular, I would like to thank Zhihan Guo, Elena Milkai, Wenjie Hu, and Hokeun Cha for being wonderful colleagues and companions throughout this journey. I am also grateful to my friends Yuyuan Kang, Xiangpeng Hao, and Ling Zhang, whose friendship made life in Madison more enjoyable and balanced.

I am also grateful to my girlfriend, Liying Yang, for her patience, understanding, and unwavering emotional support. Her presence brought me immense encouragement and reassurance throughout this journey.

Finally, I would like to express my deepest gratitude to my parents, Dian Wang and Wenjun Yang, for their unconditional love, understanding, and encouragement. Their support has always been my strongest source of motivation.

Contents

Contents iv

List of Tables vii

List of Figures ix

Abstract xiii

1 Introduction 1

1.1 *Innovations in Algorithmic Data Reduction* 3

1.2 *Innovations in the Infrastructure Stack — Cloud- and GPU-Native Execution Engines* 6

1.3 *Organization* 9

I Innovations in Algorithmic Data Reduction 10

2 Efficient Pre-filtering through Predicate Transfer 11

2.1 *Background* 12

2.2 *Predicate Transfer* 14

2.3 *Toward Join-Order Robustness: the LargestRoot Heuristic* 23

2.4 *Evaluation* 26

2.5 *Conclusion* 35

3 Accelerate Distributed Joins with Predicate Transfer 37

3.1 *Distributed Predicate Transfer* 39

3.2 *Cost-Based Adaptive Execution* 46

3.3 *Pruning* 49

- 3.4 *Additional Optimizations and Implementation* 53
- 3.5 *Evaluation* 54
- 3.6 *Related Work* 70
- 3.7 *Conclusion* 71

II Innovations in the Infrastructure Stack — Cloud- and GPU-Native Execution Engines 72

- 4 Hybrid Pushdown and Caching in Cloud-Native Databases 73
 - 4.1 *Background and Motivation* 75
 - 4.2 *FPDB Overview* 78
 - 4.3 *Hybrid Query Executor* 80
 - 4.4 *Cache Manager* 85
 - 4.5 *Implementation* 90
 - 4.6 *Evaluation* 92
 - 4.7 *Related Work* 103
 - 4.8 *Conclusion* 105
- 5 Enhance Pushdown Computation for Cloud-Native Databases 106
 - 5.1 *Background and Motivation* 108
 - 5.2 *Adaptive Pushdown* 110
 - 5.3 *Analysis of Pushdown Operators* 116
 - 5.4 *Implementation* 125
 - 5.5 *Evaluation* 128
 - 5.6 *Related Work* 139
 - 5.7 *Conclusion* 140
- 6 Sirius: A Drop-In GPU-Native SQL Engine 141
 - 6.1 *GPU Era for Data Analytics: Why Now* 143
 - 6.2 *Sirius Design and Architecture* 146
 - 6.3 *Evaluation* 151
 - 6.4 *Conclusion* 156
- 7 Future Directions 158
 - 7.1 *Future Directions in the Algorithmic Stack* 158

7.2	<i>Future Directions in the Infrastructure Stack</i>	165
8	Conclusion	169
	References	172

List of Tables

2.1	Join Table Size in Q5 (SF = 1) — HT denotes the number of rows in the hash table, and PR represents the number of rows that probe the hash table.	30
2.2	Join Table Size in Q5 (SF = 10) — HT denotes the number of rows in the hash table, and PR represents the number of rows that probe the hash table.	30
2.3	Robustness Factors for Left-Deep Joins.	33
3.1	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.	43
3.2	Parameters Used in the Cost Model.	46
3.3	Summary of Cost for Different Distributed Predicate Transfer Algorithms.	47
3.4	Discovery of Join Key Containment on TPC-H.	62
3.5	Detailed Filtering Metrics of Individual Predicate Transfer Steps on TPC-H Q5 (SF100, 4 nodes).	64
3.6	Detailed Filtering Metrics of Individual Predicate Transfer Steps on TPC-H Q10 (SF100, 4 nodes).	65
4.1	Performance Comparsion between Presto and FPDB — The runtime (in seconds) of different architectures in both systems.	97
4.2	Network Usage (GB) of different architectures.	102
4.3	CPU Usage (with dedicated compute servers) — CPU time (in minutes) of different architectures (normalized to the time of 1 vCPU).	102
4.4	CPU Usage (with multi-tenant compute servers) — CPU time (in minutes) of different architectures (normalized to the time of 1 vCPU).	102

5.1	Supported Pushdown Operators in Existing Cloud OLAP DBMSs — FPDB refers to the work [208] discussed in Chapter 4, which further supports hybrid pushdown and caching query execution. Marked with *: Pushdown of grouped aggregation and Bloom filters are not efficiently supported by PushdownDB, join pushdown in PolarDB-X requires both tables co-located on the join key, and top-K pushdown in PolarDB-MySQL requires indexes on the sort key.	117
6.1	Comparison of CPU and GPU Instances	141
6.2	TPC-H End-to-End Query Performance in the Distributed Setting.	154

List of Figures

2.1	Predicate Transfer for TPC-H Q5.	15
2.2	Filter Transformation — Table R receives two incoming filters on join attributes A and B , and generates a transformed outgoing filter on join attribute C	16
2.3	Example of Predicate Transfer on a Join Query — $R \bowtie S \bowtie T$	18
2.4	An Example of Small2Large, the Default Predicate Transfer Heuristic . .	24
2.5	Performance Evaluation of Predicate Transfer on TPC-H (normalized to NoPredTrans).	28
2.6	Performance Breakdown on TPC-H Q5.	32
2.7	Performance of Different Join Orders on TPC-H Q5.	32
2.8	Distribution of the execution time of random left-deep plans for each query in TPC-H, JOB, and TPC-DS – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers). ‘*’ indicates timeouts. Cyclic queries are in red.	36
3.1	Broadcast-Based Value-Exchange (BCAST-VAL) — Transfer is from the source table R to the destination table S	40
3.2	Broadcast-Based Bloom Filter Exchange (BCAST-BF) — Transfer is from the source table R to the destination table S	41
3.3	Shuffled-Based Distributed Predicate Transfer — Transfer is from the source table R to the destination table S	44
3.4	Pruning for Predicate Transfer in TPC-H Q3 and its variant.	51
3.5	TPC-H Performance of Distributed Predicate Transfer (SF100, 4 nodes). .	56

3.6	TPC-H Performance of Different Distributed Predicate Transfer Algorithms — GeoMean across all queries (SF100, 4 nodes).	57
3.7	TPC-H Performance of Distributed Predicate Transfer Algorithm and Cost-Based Adaptive Execution on Individual Queries.	59
3.8	Breakdown of Selected Distributed Predicate Transfer Algorithms in Cost-Based Adaptive Execution on TPC-H.	60
3.9	Amount of Transfer Step Pruning Occurred on TPC-H.	62
3.10	TPC-H Performance of BCAST-BF Designs — GeoMean of performance across all measured queries.	63
3.11	Resource Utilization on Representative Queries of TPC-H (SF100, 4 nodes).	65
3.12	Scalability of Distributed Predicate Transfer on TPC-H — GeoMean of performance across all measured queries.	66
3.13	DSB Performance of Distributed Predicate Transfer (SF100, 4 nodes).	68
3.14	Performance Comparison of FPDB against Presto — GeoMean across all measured queries (SF100, 4 nodes); Presto (DF) represents Presto execution with “dynamic filtering” enabled.	69
4.1	Distributed Database Architectures.	76
4.2	Performance trade-off between caching, computation pushdown, and an ideal hybrid approach.	77
4.3	System Architectures — High level architectural comparison between <i>Caching-only</i> , <i>Pushdown-only</i> , and FPDB.	77
4.4	Example of a Separable Query Plan — The hybrid query plan contains a parallel <i>merge</i> operator that combines the results from cache and computation pushdown.	82
4.5	Separable Query Plan — For the query in Listing 4.1.	83
4.6	Integration of Executor and Cache Manager — The cache manager determines what segments should stay in cache.	86
4.7	Performance Comparison (c5a.8xlarge) — The runtime with different (a) cache sizes and (b) access skewness.	94
4.8	Performance Comparison (c5n.9xlarge) — The runtime with different (a) cache sizes and (b) access skewness.	95
4.9	Per-Query Speedup — The average speedup of each representative case in different architectures.	96

4.10	Performance of different architectures with different cache sizes on the Parquet data (<i>Pushdown-only</i> as <i>PD-only</i> , <i>Caching-only</i> as <i>CA-only</i>). Results on the CSV data (Figure 4.7a, 4.8a) are added for reference.	98
4.11	Weighted-LFU Cache Replacement Policy — Runtime comparison of Weighted-LFU (WLFU) and baseline replacement policies (i.e., LFU and Belady) with varying access skewness.	99
4.12	Network Bandwidth — The runtime of different (a) architectures and (b) cache replacement policies as the network bandwidth varies.	100
5.1	Performance of <i>No pushdown</i> and <i>Eager pushdown</i> on Sample Queries — Q1 and Q19 in TPC-H.	109
5.2	Architecture of Adaptive pushdown — The <i>Adaptive Pushdown Arbitrator</i> determines whether to accept a pushdown request for execution or push it back.	111
5.3	Selection Bitmap Pushdown (from the Storage Layer) — Selection bitmap constructed at storage can be used to filter cached data at the compute layer.	121
5.4	Selection Bitmap Pushdown (from the Compute Layer) — Storage can use the compute-layer selection bitmap to perform filtering without touching the predicate columns.	122
5.5	Distributed Data Shuffle Pushdown — Data is directly redistributed to the target compute node from the storage layer.	123
5.6	Distributed Query Execution in FPDB.	125
5.7	Performance Evaluation of Adaptive Pushdown on TPC-H (normalized to <i>No pushdown</i>).	130
5.8	Percentage of the Number of Admitted Pushdown Requests over the Total Number of Requests.	132
5.9	Total Amount of Data Transfer through the network (in a unit of gigabytes) of the Two Representative Queries (Q12 and Q14) among Different Pushdown Strategies.	133
5.10	Performance Breakdown on Two Representative Queries (Q12 and Q14).	133
5.11	Evaluation of Awareness of Pushdown-Amenability in Concurrent Executions.	134
5.12	Number of Admitted Pushdown Requests at storage in Concurrent Executions.	135

5.13	Total Resource Consumption of the Storage Layer in Concurrent Executions (CPU consumption is measured by the total CPU time that is normalized to the time of 1 vCPU, and network consumption is measured by the total amount of data transfer, in a unit of gigabytes).	135
5.14	Performance Impact of Selection Bitmap Pushdown (Results are normalized to <i>baseline pushdown</i>) — The selection bitmap is constructed at the storage layer.	136
5.15	Performance Impact of Selection Bitmap Pushdown (Results are normalized to <i>baseline pushdown</i>) — The selection bitmap is constructed at the compute layer.	137
5.16	Performance Evaluation of Distributed Data Shuffle Pushdown on TPC-H (normalized to <i>no pushdown</i>).	139
6.1	Recent hardware trends	144
6.2	Sirius Architecture	148
6.3	Distributed Query Lifecycle in Vanilla Doris and Sirius-Accelerated Doris.	151
6.4	TPC-H End-to-end Query Performance on a Single Node.	153
6.5	TPC-H Performance Breakdown.	153
6.6	Normalized Total Cost of Ownership (TCO) Relative to Sirius in ClickBench.	156

Abstract

Large-scale SQL analytics is a foundational component of modern data-driven systems, yet its performance is increasingly challenged by growing data volumes, complex join patterns, and evolving cloud and hardware infrastructures. Addressing these challenges requires innovations that span both algorithmic query processing and system-level execution architectures. This thesis presents a set of performance optimizations for large-scale SQL analytics across the algorithmic and infrastructure stacks, with a particular focus on reducing intermediate data volumes and improving execution efficiency on modern infrastructures.

On the algorithmic side, we develop practical data-reduction techniques for complex join queries. We introduce predicate transfer, a novel pre-filtering method inspired by the theoretical foundations of the Yannakakis algorithm, which generalizes Bloom joins by propagating table-local predicates across multiple joins. Predicate transfer significantly reduces intermediate join sizes and accelerates query execution while maintaining low overhead. We further study the robustness of predicate transfer and propose heuristic scheduling techniques that guarantee full input reduction and mitigate join-order sensitivity. To make these benefits available at scale, we extend predicate transfer to distributed environments, designing both broadcast-based and shuffle-based algorithms and an adaptive model that minimizes network traffic. We additionally propose pruning techniques that eliminate unnecessary transfer steps without sacrificing filtering effectiveness. For complex cyclic join queries, we make an initial exploration into partition-based join processing, bringing theoretically grounded approaches into practical systems and demonstrating promising performance improvements on real-world workloads.

On the infrastructure side, we investigate execution optimizations for cloud-native and GPU-native database systems. For cloud-native databases with disaggregated compute and storage, we present FlexPushdownDB, which incorporates a hybrid

execution framework that unifies caching and computation pushdown, with a benefit-based cache replacement policy tailored to the pushdown context, to alleviate the network bottleneck between the compute and storage. We further enhance pushdown computation with an adaptive mechanism that prevents storage-layer computation from becoming a performance bottleneck in multi-tenant environments. We also derive a general principle for identifying pushdown-amenable operators and demonstrate its effectiveness by enabling new pushdown capabilities. Finally, we present Sirius, a GPU-native SQL analytics engine that provides drop-in acceleration for existing database systems. Leveraging modern GPU hardware and composable software stacks, Sirius achieves substantial performance and cost-efficiency improvements in both single-node and distributed settings.

Collectively, this thesis advances the state of the art in large-scale SQL analytics by bridging theory and practice in data reduction and by designing infrastructure-aware execution strategies. The proposed techniques contribute toward more efficient, adaptive, and scalable query engines capable of fully exploiting modern cloud and hardware platforms.

Chapter 1

Introduction

Modern organizations increasingly rely on large-scale data analysis to support decision-making, automation, and business intelligence. As data collection expands across applications [57], ranging from transactional systems to sensors and user activity logs, analytical workloads continue to grow in both volume and complexity [100]. This growth has intensified the demand for efficient Online Analytical Processing (OLAP) systems capable of executing SQL queries at scale.

Modern data analytics techniques are embracing new innovations across the stack. Advances span from query execution algorithms to cloud and hardware infrastructure, each pushing the limits of performance from a different angle. At the algorithmic stack, the query optimizer plays a central role in determining execution efficiency, particularly for analytical queries involving multiple large tables, nested subqueries, and complex predicates. Traditional optimization techniques include join ordering [185, 177, 139, 193, 181] and cardinality estimation [219, 132, 114], which aim to identify the most efficient sequence of operations for processing data. Data reduction methods, such as predicate pushdown [191, 116, 140, 118, 127, 98], Bloom filters [78, 187, 144, 157, 61], and metadata-based skipping [1, 7, 58, 152, 136, 29, 12, 180], improve performance by filtering unnecessary data early in the execution pipeline. Adaptive query processing [95, 117, 67] allows the system to adjust execution strategies dynamically in response to runtime statistics, mitigating performance degradation caused by unpredictable data distributions.

Alongside algorithmic advances, the execution environment for analytics has shifted dramatically. With massive amounts of data migrating to the cloud, cloud-native designs have become essential for large-scale query processing. Modern cloud-native databases often adopt a storage-disaggregation architecture, separating

compute and storage to allow independent scaling and virtually unlimited elasticity. This separation, however, places the network between compute and storage as a major performance bottleneck. Pushdown computation [203, 101, 21, 70, 217] mitigates this bottleneck by performing computation closer to the storage layer, minimizing data movement across the network. Meanwhile, GPUs have emerged to provide substantial throughput improvements for data-parallel workloads. With larger GPU memory, higher-bandwidth interconnects, and faster network and storage, GPUs have become viable execution targets for general SQL analytics.

In this thesis, we introduce innovations across both the algorithmic and infrastructure stacks. On the algorithmic side (Section 1.1), we focus on developing runtime data-reduction techniques. We begin with predicate transfer [211], an efficient pre-filtering approach for complex multi-join queries. Beyond its performance benefits, we further investigate how predicate transfer enables robust query processing [222]. Building on this foundation, we extend predicate transfer from its original single-node design to support large-scale workloads in distributed environments [209]. While predicate transfer offers strong performance and robustness guarantees for acyclic queries, we also address cyclic queries by proposing SplitJoin [115], a framework that reduces intermediate sizes for cyclic queries by enabling different query plans for heavy and light partitions of input tables.

On the infrastructure side (Section 1.2), we present practical techniques and implementations for both cloud-native and GPU-native query processing. In cloud-native databases with disaggregated storage and compute, the network between the two layers often becomes a major performance bottleneck. To mitigate this, we introduce FlexPushdownDB [208], an OLAP cloud DBMS prototype that combines caching and computation pushdown to reduce network traffic. We then explore enhancements to existing pushdown layers [210], by designing adaptive pushdown as a new mechanism to avoid throttling the storage-layer computation, deriving a general principle for identifying pushdown-amenable computational tasks, and introduce selection bitmap and distributed data shuffle as two new pushdown operators. Finally, we present Sirius [214], a prototype open-source GPU-native SQL engine that offers drop-in acceleration for diverse data systems such as DuckDB [166] and Apache Doris [26].

1.1 Innovations in Algorithmic Data Reduction

Joins are among the most complex relational operators in query processing. For decades, researchers have sought to reduce intermediate data volumes when executing join queries, leading to a variety of techniques in areas such as join ordering and cardinality estimation. In this section, we present our journey toward efficient data reduction algorithms for complex join queries. For acyclic queries, we introduce predicate transfer (Section 1.1.1), an efficient and practical pre-filtering technique that reduces join input sizes while providing strong theoretical guarantees. We further extend predicate transfer to distributed environments (Section 1.1.2). For cyclic queries, we propose SplitJoin (Section 1.1.3), a framework that bounds intermediate join results by partitioning input tables based on skewness and adapting them to potentially different query plans.

1.1.1 Efficient Pre-filtering through Predicate Transfer

Joins constitute a substantial portion of query execution time, and have been studied and optimized for decades. One effective principle for enhancing join performance is to minimize the join input sizes by pre-filtering rows that will not appear in the join result. For example, predicate pushdown [191, 116, 140, 118, 127, 98] exemplifies this principle by applying local predicates on a table before executing any join operation.

The Bloom join [168, 79, 135] extends this principle beyond a single table. In the Bloom join, a Bloom filter is constructed using the join key in one table, and sent to the other table to filter out rows that do not pass the filter — these rows do not match any keys in the first table and will not participate in the join. The Bloom join can effectively reduce the join input sizes thereby reducing the query runtime. However, existing Bloom join solutions can perform such pre-filtering only within a single join operation.

We introduce *predicate transfer* [211], which further generalizes the pre-filtering principle across multiple joins. Namely, we use predicates on individual tables to pre-filter multiple other tables in the query, further reducing the join input sizes. The idea of predicate transfer is inspired by the seminal paper [212] by Yannakakis. For an acyclic query that equi-joins multiple tables, the Yannakakis algorithm achieves the theoretically maximum pre-filtering selectivity by adding an additional semi-join phase prior to the actual joins, which filters a table by semi-joining it with other tables.

The process filters one table at a time following the tree structure of the query until every predicate is spread across all joining tables.

For all its theoretical elegance, the Yannakakis algorithm has not yet made its way into modern database engines. The main obstacles are the costly hash table accesses and high memory consumption in the semi-join phase. Predicate transfer aims to address these practical limitations. It significantly reduces the overhead of semi-joins by passing succinct data structures like Bloom filters. Although predicate transfer no longer achieves the theoretically maximum filtering selectivity, it achieves much higher performance overall. Evaluation shows predicate transfer can outperform Bloom join by $3.3\times$ (up to $61\times$) and the Yannakakis algorithm by $4.8\times$ (up to $47\times$), respectively, on TPC-H [54] benchmark.

Motivated by the strong theoretical guarantees of the Yannakakis algorithm, which ensures complexity linear in the input plus output size for any acyclic query regardless of its join order, we reconsider predicate transfer from a robustness point of view. We propose *robust predicate transfer* [222], with a new heuristic called *LargestRoot* to ensure join-order robustness while preserving its performance benefits. We implement robust predicate transfer in DuckDB [166], a state-of-the-art in-process analytical database management system, and evaluate on major analytics benchmarks including TPC-H [54], JOB [139], TPC-DS [53], and DSB [96]. Our results show that robust predicate transfer can improve join-order robustness by orders of magnitude relative to the baseline. *The work of robust predicate transfer was led by Junyi Zhao, and my contribution was to assist with the design of LargestRoot and efficient Bloom filter implementations in DuckDB.*

1.1.2 Accelerate Distributed Joins with Predicate Transfer

Despite the performance benefit and robustness for join processing, 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.

We aim to address both limitations above to make predicate transfer more widely

applicable. First, we 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*. 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.

Second, we 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.

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 $3\times$, and reduce the amount of data exchange by over $2.7\times$.

1.1.3 Accelerate Cyclic Queries with Per-Split Query Plans

Although the seminal Yannakakis algorithm and predicate transfer offer strong guarantees for acyclic queries, cyclic queries remain an open challenge to minimize intermediate join results in both theory and practical systems. Worst-case optimal join (WCOJ) algorithms [196, 154, 175] guarantee runtimes bounded by the maximum possible output size, thereby avoiding the quadratic intermediate blow-ups that can arise in traditional plans. Despite this theoretical elegance, WCOJ has not been widely adopted in practice due to its complex implementation such as indexing structures [205].

Complementary to WCOJs, *partition-based strategies* [130, 93, 94, 119] mitigate skew directly by separating heavy and light values in the join key domains and evaluating them under different subplans. Building on this insight, we propose *SplitJoin*, which introduces “split” as a first-class operator in query processing. The split operator partitions an input table into light and heavy portions, which are then processed with potentially different query plans. This thesis briefly discusses the initial design of SplitJoin and presents preliminary results in Section 7.1.2. *This work was led by Yujun*

He, and my contribution was to assist with practical SplitJoin heuristics and benchmarking on DuckDB.

1.2 Innovations in the Infrastructure Stack — Cloud- and GPU-Native Execution Engines

Beyond recent algorithmic advances, the underlying infrastructure for large-scale SQL analytics has also undergone significant changes. First, modern DBMSs are increasingly migrating to the cloud to achieve greater elasticity and lower cost. We present several optimization techniques for more efficient handling of cloud-native analytics workloads (Sections 1.2.1 and 1.2.2). Second, compute-intensive workloads are progressively moving to GPUs to achieve potentially substantial performance gains. We introduce Sirius (Section 1.2.3), a GPU-native SQL query engine that provides both higher performance and lower cost as a drop-in solution for existing DBMSs.

1.2.1 Hybrid Pushdown and Caching in Cloud-Native Databases

Modern cloud DBMSs adopt a storage-disaggregation architecture that divides computation and storage into separate layers of servers connected through the network, simplifying provisioning and enabling independent scaling of resources. However, the network in the disaggregation architecture typically has lower bandwidth than local disks, making it a potential performance bottleneck. Two solutions have been explored to mitigate this network bottleneck: caching [92, 199, 13] and computation pushdown [203, 101, 21, 70, 217], where both can reduce the amount of data transferred between the two layers. Existing DBMSs consider caching and computation pushdown as orthogonal, and most systems implement only one of them.

We argue that caching and computation pushdown are not orthogonal techniques, and that the rigid dichotomy of existing systems leaves potential performance benefits unexploited. We propose FlexPushdownDB [208] (FPDB in short), an OLAP cloud DBMS prototype that supports *fine-grained hybrid execution of caching and pushdown*. FPDB introduces the concept of separable operators and a merge operator, such that the results of local computation and pushdown computation are combined at a fine granularity. Our evaluation shows the hybrid execution outperforms

both caching-only and pushdown-only architectures by $2.2\times$ on the Star Schema Benchmark (SSB) [156].

Separable operators open up new possibilities for caching. Traditional cache replacement policies assume that each miss requires loading the missing data block to the cache, which incurs a constant cost if the blocks have the same size. In FPDB, however, this assumption is no longer true because we can push down computation instead of loading data. The cost of a miss now depends on how amenable the block is to pushdown — for misses that can be accelerated with pushdown (e.g., high-selectivity filters), the cost of a miss is lower. Therefore, we develop a new benefit-based caching framework and a new caching policy called *Weighted-LFU*, which incorporates caching and pushdown into a unified cost model to predict the best cache eviction decisions. Our evaluation shows that *Weighted-LFU* outperforms the original LFU by 37%.

1.2.2 Enhance Pushdown Computation for Cloud-Native Databases

As presented, pushdown is a widely adopted technique to alleviate the performance bottleneck in a storage-disaggregated architecture. However, existing pushdown implementations in cloud-native databases suffer from several shortcomings. First, existing cloud databases statically decide whether to push down computation during the query optimization phase and do not consider the storage layer’s computational capacity and load. Second, there is a lack of a general principle that determines which operators are amenable for pushdown. Existing systems design and implement pushdown features empirically, which ends up picking a limited set of pushdown operators respectively.

We address both limitations. First, we design *adaptive pushdown* as a new mechanism to avoid throttling the storage-layer computation during pushdown. Instead of having the database engine making the pushdown decisions, adaptive pushdown lets the storage layer to decide whether to execute an incoming pushdown request, or to push the request back to the computation layer. When a pushback happens, the computation layer reads the raw data from the storage layer and processes the task locally. Our evaluation shows that adaptive pushdown outperforms baselines of no pushdown and eager pushdown by up to $1.9\times$ on the TPC-H [54] benchmark.

Second, we derive a general principle to identify pushdown-amenable computational tasks, through summarizing common patterns of pushdown capabilities

in existing systems. We summarize that pushdown tasks should be local (i.e., the pushdown computation should access data only within a single storage node and not incur data transfer within the storage layer), and bounded (i.e., a pushdown task should require at most linear CPU and memory resources with respect to the accessed data size). Following the principle, we further identify two operators that can benefit from pushdown to the storage layer — *selection bitmap* and *distributed data shuffle*. Our evaluation shows that the two new pushdown operators can further accelerate end-to-end query processing on TPC-H by up to $3.0\times$ and $1.7\times$ respectively.

1.2.3 Sirius: A Drop-In GPU-Native SQL Engine

Modern GPUs are rapidly surpassing CPUs in both compute throughput and memory bandwidth, making them the default choice for compute-intensive workloads. With recent advances in hardware (e.g., larger GPU memory, faster interconnect and IO, and declining cost) and software (e.g., composable data systems and mature libraries), we argue that the key barriers that have limited the wider adoption of GPU data analytics have been removed.

We present Sirius [214], a prototype open-source SQL engine that uses GPU as the primary execution device and delivers drop-in acceleration across diverse data systems. Sirius integrates seamlessly with existing database systems via the standard Substrait query representation. Sirius builds on GPU libraries such as `libcudf` [36], RMM [50], and NCCL [44], reusing optimized implementations of core relational operators like joins, filters, aggregations, and data shuffle. Thanks to its modular design, Sirius also allows developers to easily switch the operator implementation between these GPU libraries and custom CUDA kernels. This flexibility reduces engineering overhead while enabling Sirius to achieve end-to-end GPU acceleration. Our detailed evaluation shows that Sirius delivers $8.3\times$ and $7.4\times$ better cost efficiency on TPC-H and ClickBench, respectively, when used as a drop-in accelerator for DuckDB [166] (single-node), and up to $12.5\times$ speedup as a drop-in accelerator for Apache Doris [26] (distributed). *This project was co-led by Bobbi Yogatama and me. Bobbi contributed to the primary design of Sirius and its integration with DuckDB, while I contributed to the design of the distributed exchange service and the integration of Sirius with Apache Doris.*

1.3 Organization

The rest of this thesis is organized as follows. The thesis is mainly divided into two parts.

Part I: Innovations in Algorithmic Data Reduction. This part focuses on analyzing large-scale SQL analytics using data reduction algorithms, such as predicate transfer.

- **Chapter 2** presents predicate transfer, an efficient pre-filtering approach for complex multi-join queries, achieving both substantial performance improvements and strong performance robustness.
- **Chapter 3** extends predicate transfer from its original single-node design to support large-scale workloads in distributed environments.

Part II: Innovations in the Infrastructure Stack — Cloud- and GPU-Native Execution Engines.

- **Chapter 4** introduces FlexPushdownDB, an OLAP cloud-native DBMS prototype that combines caching and computation pushdown to reduce network traffic in a storage-disaggregated architecture.
- **Chapter 5** enhances pushdown processing for cloud-native DBMSs by introducing an adaptive pushdown mechanism, a general principle for identifying pushdown-amenable tasks, and two new pushdown operators — selection bitmap and distributed data shuffle.
- **Chapter 6** presents Sirius, a prototype open-source GPU-native SQL engine that offers drop-in acceleration for diverse data systems such as DuckDB and Apache Doris.

Finally, we discuss potential future directions in Chapter 7 and summarize the thesis in Chapter 8.

Part I

**Innovations in Algorithmic Data
Reduction**

Chapter 2

Efficient Pre-filtering through Predicate Transfer

Joins constitute a substantial portion of query execution time, and have been studied and optimized for decades, in topics including binary joins (with a main focus on hash joins) [77, 87, 131, 69], join ordering in multi-way joins [185, 177, 139, 193, 181], and recent emerging worst-case optimal join algorithms [154, 103, 196, 202]. One effective principle for enhancing join performance is to minimize the join input sizes by pre-filtering rows that will not appear in the join result. Predicate pushdown [191, 116, 140, 118, 127, 98] exemplifies this principle by applying local predicates on a table before executing any join operation.

The Bloom join [168, 79, 135] extends this principle beyond a single table. In the Bloom join, a Bloom filter is constructed using the join key in one table, and sent to the other table to filter out rows that do not pass the filter — these rows do not match any keys in the first table and will not participate in the join. The Bloom join can effectively reduce the join input sizes thereby reducing the query runtime. However, existing Bloom join solutions can perform such pre-filtering only within a single join operation.

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

filter multiple other tables (e.g, T_2 , T_3 , etc.). The conventional Bloom join is a special case of the more generalized predicate transfer — a Bloom join is a one-hop predicate transfer.

The idea of predicate transfer is inspired by the seminal paper [212] by Yannakakis. For an acyclic query that equi-joins multiple tables, the Yannakakis algorithm achieves the theoretically maximum pre-filtering selectivity by adding an additional semi-join phase prior to the actual joins, which filters a table by semi-joining it with other tables. The process filters one table at a time following the tree structure of the query until every predicate is spread across all joining tables.

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

In the rest of this chapter, we first describe the background and related work of predicate transfer in Section 2.1, with a focus on the Bloom join and Yannakakis algorithm. Next, Section 2.2 presents the design space of predicate transfer in detail, along with our current heuristics (i.e., *Small2Large*) across different design dimensions. We then demonstrate how predicate transfer can achieve join-order robustness in Section 2.3 with a new heuristic called *LargestRoot*. Section 2.4 reports performance evaluations on major analytics benchmarks, including TPC-H [54], JOB [139], TPC-DS [53], and DSB [96]. Finally, we conclude in Section 2.5. This chapter is adapted from previously published work [211, 222].

Acknowledgement. The work of robust predicate transfer heuristic (Section 2.3 and Section 2.4.4) was led by Junyi Zhao, where my contribution was to assist with the design of *LargestRoot* and efficient Bloom filter implementations in DuckDB.

2.1 Background

This section presents the background and related work in Bloom join (Section 2.1.1) and the Yannakakis algorithm (Section 2.1.2).

2.1.1 Bloom join

A Bloom filter [78, 187, 144, 157, 61] is a compact probabilistic data structure that determines whether an element exists in a set. A Bloom filter has no false negative but may have false positives. In a Bloom join of two tables, a Bloom filter is constructed on one table (typically the smaller one) using the join key. The filter is then sent and applied to each row in the other table; if a row does not pass the filter, it matches no row in the first table and should not participate in the join. Since testing a Bloom filter is generally faster than performing a join, Bloom join can speedup query processing, especially when the join is selective. Modern OLAP DBMSs (e.g., Oracle [90], Redshift [82], Snowflake [4], Databricks [107]) widely adopt Bloom filters to accelerate join execution.

Most existing Bloom join algorithms can be applied to only a single join operation. This means the predicate on one table can only be used to pre-filter rows in the other table it joins with; namely, the predicate is transferred in one-hop and one-direction. Some prior work [224] has extended the idea to datasets with star schemas, allowing all dimension tables to transfer local predicates to the fact table, which outperformed the baseline Bloom join. However, these solutions do not generalize to more complex query plans.

2.1.2 Yannakakis algorithm

The Yannakakis algorithm [212] is a classic algorithm that can pre-filter out all rows from tables that do not appear in the final join result, thereby achieving the theoretically maximum filtering selectivity. The algorithm applies to *acyclic* join queries. The acyclicity is more formally termed as α -*acyclicity* [212]. The algorithm is proven to run in $O(N + \text{OUT})$ time, where N is the size of input relations and OUT is the query output size. Thus, the Yannakakis algorithm is known to be *instance optimal* since $N + \text{OUT}$ is the unavoidable time cost of reading the input and enumerating the output for a query. The algorithm starts by choosing a rooted join tree arbitrarily, and then proceeds with a *semi-join phase* and a *join phase*.

Semi-join phase. The semi-join phase contains two passes: the *forward pass* and the *backward pass*. The forward pass traverses the join tree in a bottom-up fashion. At each vertex, we filter the table by a sequence of semi-joins with its children. A *semi-join* of two tables R and S is defined as $R \ltimes S = \Pi_{\text{attr}(R)}(R \bowtie S)$, which effectively

removes all tuples in R that do not join with any tuple in S . The forward pass stops when the root node is reached. Similarly, the backward pass traverses the join tree in a top-down fashion. At each vertex, the table is filtered by a semi-join with its parent. The backward pass stops when all leaf nodes are reached. It is proven that both passes can be executed in $O(N)$ time and all tuples that will not contribute to the output are removed.

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

As a reflection, the semi-join phase filters all redundant tuples and the join phase executes the join with automatic robustness: it can join the tables in any order without any intermediate table size blow-up over the output size. The algorithm was later extended by Joglekar et al. [122] to handle aggregations on top of join queries.

2.2 Predicate Transfer

This section describes the proposed predicate transfer algorithm. We use Query 5 in TPC-H benchmark [54] (Figure 2.1) as a running example. This query contains six tables, six inner joins, and two predicates on tables *region* and *orders* respectively. The discussion assumes equi-join between tables.

2.2.1 Overview

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

Phase 1: Predicate Transfer Phase. A join graph is constructed for a query, where each vertex is a table and each edge is a join operation. A local predicate is constructed as a filter (e.g., a Bloom filter) and be transferred across the join graph. The schedule of the predicate transfer phase introduces a large design space, which we discuss in Section 2.2.2.

Phase 2: Join Phase. After the transfer phase finishes, each table has multiple filters, including both local filters and transferred filters. The database can now apply the filters and perform regular joins. The actual inputs of each join will be substantially smaller if the transferred filters are selective. We discuss the join phase in Section 2.2.3.

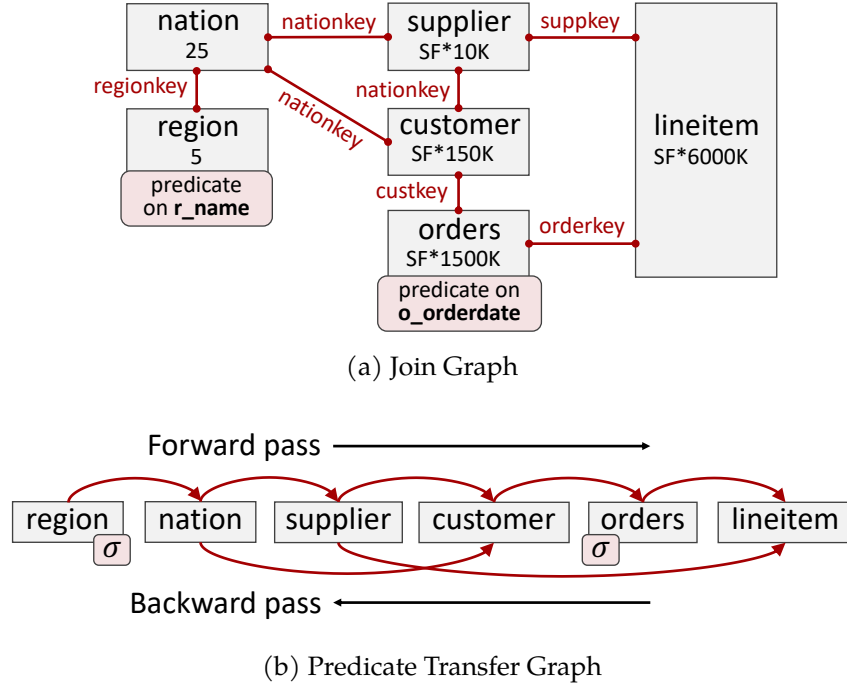


Figure 2.1: Predicate Transfer for TPC-H Q5.

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

2.2.2 Predicate Transfer Phase

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

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

used to probe the incoming filter. If a match occurs, the corresponding outgoing join key is added to the outgoing filter. At the end of the scan, the outgoing filter is sent to downstream tables (i.e., supplier). The algorithm is efficient as it requires scanning the join keys only once.

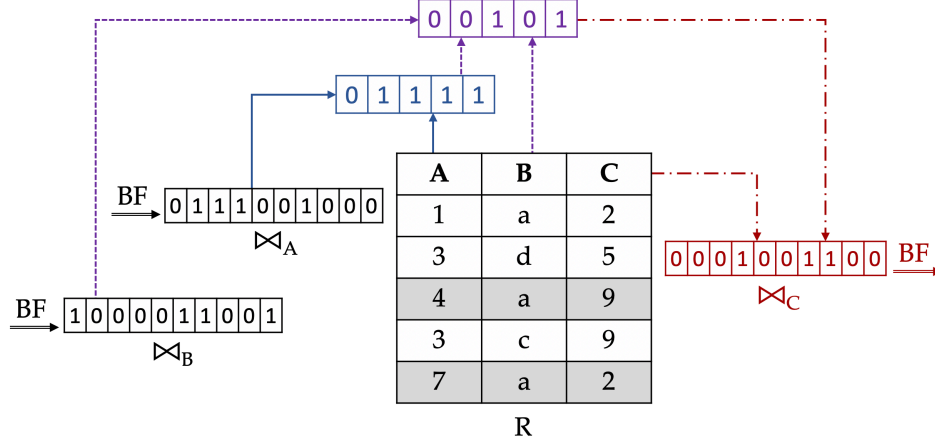


Figure 2.2: Filter Transformation — Table R receives two incoming filters on join attributes A and B, and generates a transformed outgoing filter on join attribute C.

Figure 2.2 shows an example of filter transformation on table R. Table R has three columns, and participates into three different joins on columns A, B, and C respectively. In the predicate transfer phase, R receives two incoming filters (assume we are using Bloom filters) on join attributes A and B respectively. Column A is used to probe the incoming filter on join attribute A, where all rows except the first satisfy the join condition (with false positives). Then column B on the remaining rows is used to probe the incoming filter on join attribute B, with another two rows (row 2 and 4) filtered out. Finally, column C on the remaining two rows (row 3 and 5) is used to build the outgoing filter.

Predicate Transfer Graph. The join graph determines the topology of predicate transfer. Figure 2.1a shows the join graph for Query 5 in TPC-H. Each equi-join is represented as an edge. A *predicate transfer graph* is a directed subgraph of the join graph. Transfers happen along the selected edges in the subgraph — local predicates of the source vertex are transferred to the target vertex as a filter. Figure 2.1b shows one predicate transfer graph of TPC-H Q5.

The topology of the predicate transfer graph affects the performance of the predicate transfer phase and also the selectivity of the transferred filters. We use a simple heuristic called *Small2Large* that points an edge from a smaller table to a bigger table.

The intuition is the same as why Bloom join builds Bloom filter at the smaller table — to reduce Bloom filter size and increase filter selectivity. Our current heuristic does not remove any edge in the join graph when generating the predicate transfer graph. It also guarantees that the resulting graph is a Directed Acyclic Graph (DAG). The predicate transfer graph in Figure 2.1b follows this heuristic.

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

In the Small2Large heuristic, we build the transfer schedule using one *forward pass* and one *backward pass* similar to the Yannakakis algorithm. The predicate transfer graph is determined at planning time and remains fixed during runtime. In the forward pass, we build initial local filters on the leaf nodes in the predicate transfer graph (i.e., nodes with only outgoing edges but no incoming edge). These filters are transferred following the topological order of the predicate transfer graph, which exists because the graph is a DAG. If one node has one or more incoming edges, the node will collect all the incoming filters before performing the transformation to produce outgoing filters (LIP-style [224] incoming filter ordering can be utilized for further optimization); the transformation will scan the table only once, regardless of the number of incoming or outgoing edges. The forward pass finishes once all filters are fully transferred.

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

Figure 2.3 presents an example of predicate transfer on a join query, which joins three tables R, S, and T. Assume the transfer starts at table R and all inherit local filters are already consumed. In the forward pass (Figure 2.3a), R first constructs

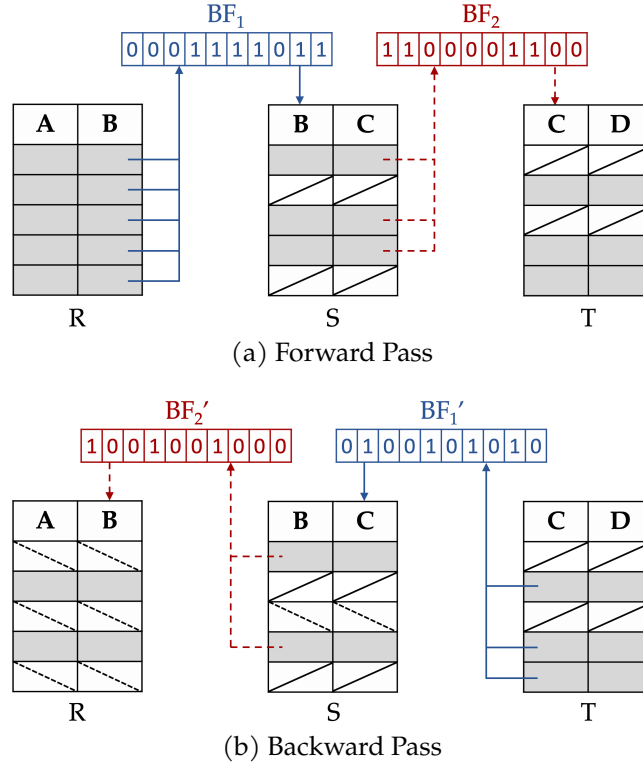


Figure 2.3: Example of Predicate Transfer on a Join Query — $R \bowtie S \bowtie T$.

a Bloom filter (BF_1) on join attribute B and sends it to S. S applies BF_1 to its join attribute B, which filters out two rows (row 2 and 5), and the remaining three rows are used to construct a new Bloom filter (BF_2) on another join attribute C. When T receives the transformed filter from S, the join column C is used to probe the filter where row 1 and 3 are removed. The backward pass (Figure 2.3b) starts with input as the remaining rows of the forward pass. Table T constructs a Bloom filter (BF'_1) on join attribute C, which is applied on S.C, with row 3 filtered out. Then S constructs another Bloom filter (BF'_2) on the join column B, which is able to filter three rows (row 1, 3, and 5) from R.

A more complicated example is TPC-H Q5, which is shown in Figure 2.1b. The first Bloom filter is constructed on region, and sent to nation. The filter is then transformed into two outgoing filters which are sent to customer and supplier respectively. Similarly, supplier transfers two outgoing filters following the edges to customer and lineitem. At customer, two separate incoming filters are applied with one outgoing filter produced and sent to order, which is then transformed and sent to lineitem. The forward pass finishes when both incoming filters arrive at lineitem,

and after that the backward pass begins in a symmetric way.

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

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

2.2.3 Join Phase

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

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

More Accurate Cardinality Estimation. The predicate transfer phase updates the cardinality of the input tables in the join phase. Therefore, the original query plan

generated beforehand may become suboptimal based on the stale cardinalities. A replanning step between the two phases may produce a better plan that leads to further performance improvement. Although join performance will be more robust to join orders (as will be shown in Section 2.4.3), performance can still be affected by the quality of the query plan, with the factors including the size of materialized intermediate tables, which table to build the hash table and which table to probe, etc. Moreover, similar to the Yannakakis algorithm, predicate transfer bounds the size of the intermediate join tables in the join phase (Section 2.2.5), which can be utilized to improve cardinality estimation.

2.2.4 Extension to General Queries

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

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

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

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

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

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

2.2.5 Cost Analysis

Compared to the Yannakakis algorithm, predicate transfer does not provide theoretical optimality, but it is more versatile. Predicate transfer supports both precise filters (like semi-join) and Bloom filters, any join-graph topology, outer joins and cyclic queries, more operators, and complex predicate transfer schedules.

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

Cost Model. Let t be the number of tables in a given join query and N be the input size (i.e. the total number of tuples in all joining tables). We assign a unit cost to each per-tuple scan, hash table insertion or probe, and a β cost per-tuple for Bloom filter insertion or probe. As a Bloom filter is of a small size and thus likely to be cache resident, Bloom filter operations are typically much cheaper than hash table

operations, i.e. $\beta \ll 1$. We assume that the Bloom filter has a false positive rate of $\epsilon \ll 1$ that can be appropriately configured (e.g., we can tune ϵ to be smaller by increasing the Bloom filter size or number of hash functions, but this makes β larger). The reader can refer to [224] for an in-depth study on Bloom filter configurations.

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

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

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

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

Assume

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

then

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

ignoring higher order terms of ϵ' , and assuming

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

where Sel_k denotes the smallest join selectivity in the joining tables.

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

As a summary, the Yannakakis algorithm guarantees maximum filtering at the semi-join phase and perfect robustness at the join phase, but at the cost of a much more expensive and unstable semi-join phase (our evaluation in Section 2.4.3 verifies this). In contrast, predicate transfer addresses the shortcomings via a more stable and efficient Bloom filter transfer scheme, while maintaining near-maximum filtering capabilities at the predicate transfer phase and near-perfect robustness in the join phase.

2.3 Toward Join-Order Robustness: the LargestRoot Heuristic

Among the many challenges in a query optimizer, join ordering is the crown jewel of query optimization. Determining an optimal join order requires not only an efficient algorithm to search the enormous plan space but also an accurate cardinality estimation of the intermediate results. The latter is extremely difficult despite recent efforts to bring machine learning techniques to the problem [201, 138]. The reality is that the optimizers today constantly generate plans that are orders of magnitude slower than optimal [143, 113, 137].

Fortunately, the seminal Yannakakis algorithm [212] provides strong theoretical guarantees (Section 2.1.2). Specifically, it achieves linear complexity in the combined size of the input and output for any acyclic query, independent of the join order. Predicate transfer is designed to inherit this theoretical property for acyclic queries. The default predicate transfer heuristic, *Small2Large*, prioritizes minimizing the memory footprint of the constructed Bloom filters and delivers strong performance benefits in practice. However, while effective, it does not always realize the full theoretical reduction of input relations guaranteed by the Yannakakis algorithm. Therefore, in this section, we propose a new variant of predicate transfer heuristics, *LargestRoot* that can guarantee a full reduction of the input tables.

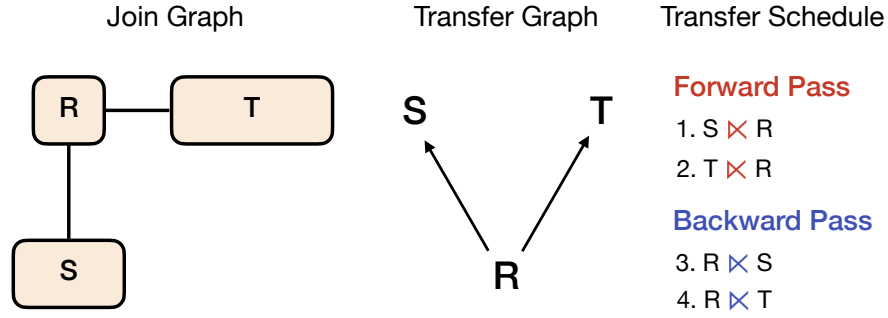


Figure 2.4: An Example of Small2Large, the Default Predicate Transfer Heuristic

As shown in Figure 2.4, for example, consider the natural join $R(A, B) \bowtie S(A, C) \bowtie T(B, D)$ where $|R| < |S| < |T|$. In this case, Small2Large will generate a transfer graph that leads to a forward pass of $R \rightarrow S$ and $R \rightarrow T$ followed by a backward pass of $S \rightarrow R$ and $T \rightarrow R$. This transfer schedule fails to “connect” S and T : if S has a predicate, this filter information can never reach T via the transfer of Bloom filters (and vice versa), leading to an incomplete reduction.

Although Small2Large cannot pre-filter all the non-result tuples, pushing larger tables toward the end of the transfer schedule is insightful because a smaller table will produce Bloom filters with less memory consumption and higher efficiency, and is likely to be more selective. The new LargestRoot heuristic will preserve this strategy while guaranteeing a full reduction. Before diving into our new algorithm, let us define the concepts of a *join tree* and *acyclicity* precisely.

Without loss of generality, we only consider natural joins with a connected join graph in this section¹. For a natural join query q , its *join graph* G_q is an undirected graph where the vertices are the relations in q . If two relations have attributes in common, they are connected by an edge in G_q . A *join tree* T_q is a spanning tree of G_q such that for every attribute A , the relations containing A induce a *connected* subgraph T_q^A of T_q . The join tree is then used to define the *acyclicity* of a query:

Definition 2.1 (α -acyclicity [212]). *A natural join query q is acyclic if and only if there exists a join tree of q .*

Acyclicity is crucial for the Yannakakis algorithm to achieve the $O(N + \text{OUT})$ complexity because it guarantees a non-decreasing intermediate result in the join phase. If the subgraph for an attribute A is *not* connected, a tuple may survive in the

¹For equality predicates such as $R.A = S.B$, we treat A and B as the same attribute in this context. If the join graph has multiple components, we can generalize the concept of join tree to join forest.

first join involving A but later get eliminated by the second join using A . This breaks the above non-decreasing property. An acyclic natural join satisfies the following lemma:

Lemma 2.2 ([145]). *Let q be an acyclic natural join query. For each edge (R, S) in the join graph G_q , where R and S are the vertices (i.e., relations), define the weight of the edge $w(R, S)$ as the number of shared attributes between R and S : $w(R, S) = |\text{attr}(R) \cap \text{attr}(S)|$. Then, a subgraph of G_q is a join tree of q if and only if it is a maximum spanning tree for G_q .*

The intuition behind the lemma is that for a spanning tree T of G_q , T 's total weight equals the summation of the edge count of each attribute-induced subgraph. T is a join tree means that every attribute-induced subgraph T^A is connected. This is equivalent to saying that every T^A is a subtree, and it is impossible for any T^A to have more edges (otherwise T will not be a tree). T must be a maximum spanning tree (MST). Note that the weights defined on the edges are not considered heuristics for join costs. They are used to transform the problem of finding a join tree into the problem of finding an MST in the join graph.

Algorithm 1: LargestRoot

Input: join graph G_q

Output: tree T

- 1 $T \leftarrow \emptyset$; $\mathcal{R} \leftarrow$ all relations; $\mathcal{R}' \leftarrow \{R_{\max}\}$;
 - 2 **while** $\mathcal{R}' \neq \mathcal{R}$ **do**
 - 3 Find an edge $e = \{R, S\} \in E(G_q)$ with the largest weight such that $R \in \mathcal{R} \setminus \mathcal{R}'$, $S \in \mathcal{R}'$. Choose the edge with the largest R to break ties;
 - 4 Add e to T with direction from R to S ;
 - 5 $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{R\}$;
 - 6 **return** T ;
-

We now know that for an acyclic query, a join tree guarantees a full (semi-join) reduction of the query, and we can find a join tree by constructing a maximum spanning tree on its weighted join graph. We next introduce our LargestRoot algorithm. As shown in Algorithm 1, we use Prim's algorithm to construct a maximum spanning tree T on the join graph G_q . The edges in T point from leaf to root, indicating a forward pass schedule. Because the algorithm starts with the largest relation $R_{\max}$ in

$\mathcal{R}'$, $R_{\max}$ is the root of T (hence the name LargestRoot). And because of Lemma 2.2, T is a join tree if query q is acyclic, guaranteeing a full reduction in the transfer phase.

Placing the largest relation at the root of the join tree is important, especially for queries following a star schema. It is more efficient to filter the much larger fact table using the dimension tables first before building a Bloom filter on the fact table. In addition, LargestRoot pushes larger relations toward the root by including them early in T in the tie-breaking strategy in 3. This allows larger relations to get filtered first by probing other Bloom filters before building their own, thus minimizing the total Bloom filter construction time in the transfer phase. Notice that 3 in LargestRoot does not specify a tie-breaking policy for choosing $S \in \mathcal{R}'$. In reality, most edges have weight 1 because relations typically join on only one attribute. Although the choice of S does not compromise the theoretical guarantee of the algorithm producing an MST, it could affect the shape of the join tree. In general, a flatter tree allows more parallelism in building the Bloom filters, while a deeper tree might allow filtering irrelevant tuples out earlier. Both the tie-breaking policies for R and S do not affect the strong theoretical guarantee (i.e., a full reduction) of LargestRoot.

Unlike the Yannakakis algorithm and similar to Small2Large, LargestRoot also applies to cyclic queries. The algorithm’s output is still a spanning tree with the largest relation at the root, but it is not a join tree. In this case, the transfer schedule generated by LargestRoot does not guarantee a fully reduced instance for the subsequent join phase. Still, it transfers any predicate to all relations at least once and is effective empirically, as we will show in the experiments in Section 2.4.4.

2.4 Evaluation

This section presents the evaluation results of predicate transfer. We describe the experimental setup in Section 2.4.1 and compare predicate transfer with baseline join strategies in Section 2.4.2. Then we perform a deep dive to understand the performance on TPC-H Q5 in Section 2.4.3. Additionally, we evaluate the robustness of the LargestRoot predicate transfer heuristic in Section 2.4.4.

2.4.1 Experimental Setup

We conduct experiments on a single AWS EC2 r5.4xlarge instance, with 16vCPU and 128GB memory. The server runs the Ubuntu 20.04 operating system. We use the

widely adopted data analytics benchmark, TPC-H, with 22 queries in total. We use both an 1GB data set (a scale factor of 1) and a 10GB data set (a scale factor of 10). Queries are executed on a single CPU core. For all the experiments, we measure the in-memory query performance by running the query twice, where the first run loads all the tables into the memory, and we measure the performance of the second run.

The testbed we use on evaluation is FlexPushdownDB [208] (*FPDB* in short), an open-source cloud-native OLAP DBMS². Table data is placed on local disks in Parquet [30] format unsharded. FPDB leverages join and Bloom filter implementation of Apache Arrow [24]. The evaluation results may vary on different DBMSs, depending on the performance ratio between the join and Bloom filter implemented.

We compare the proposed join strategy *PREDTRANS* with three other baselines: *NO-PREDTRANS*, *BLOOMJOIN*, and *YANNAKAKIS*. *NO-PREDTRANS* does not transfer predicates among joining tables — pairs of tables are joined regularly as in most DBMSs. *BLOOMJOIN* performs one-hop predicate transfer between joining table pairs, where the build side constructs a Bloom filter which is used to filter the probe side. *YANNAKAKIS* executes the semi-join phase of the Yannakakis algorithm ahead of the join phase. For *PREDTRANS*, we adopt *Small2Large*, which is the default heuristic to produce predicate transfer schedules.

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

2.4.2 TPC-H Performance

Figure 2.5 shows the execution time of different predicate transfer strategies on TPC-H queries. Since Q1 and Q6 involve no joins, we exclude them from the benchmark. On average, *PREDTRANS* outperforms *NO-PREDTRANS* by $4.1\times$, *BLOOMJOIN* by $3.3\times$, and *YANNAKAKIS* by $4.2\times$ for SF1, and $4.0\times$, $3.2\times$, and $4.8\times$ for SF10, respectively. 15 queries out of 20 see performance improvement on both data sets.

²https://github.com/cloud-olap/FlexPushdownDB/tree/cidr_24

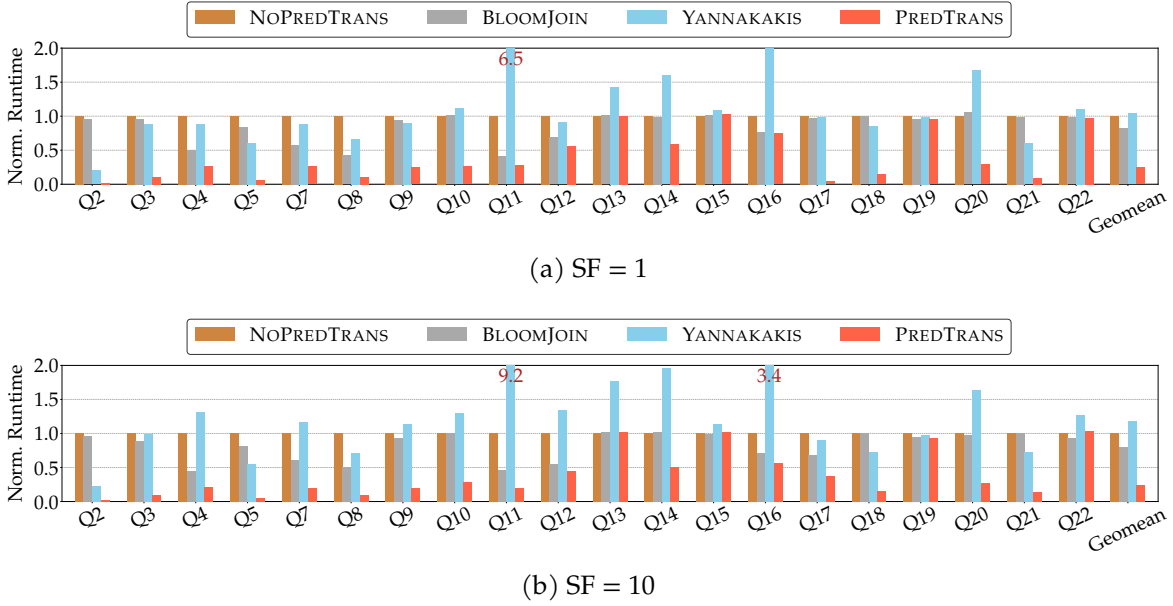


Figure 2.5: Performance Evaluation of Predicate Transfer on TPC-H (normalized to NoPredTrans).

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

On SF1, we observe the highest speed up on queries Q2, Q5, Q17, Q18, and Q21, between $7\times$ to over $44\times$. In these queries, there is a subquery executed with the results joined with the tables in the main query, and the large fact table are accessed by both the main query and the subquery (e.g., Lineitem in Q17 and Q18). Since

NO_{PREDTRANS} and BLOOM_{JOIN} perform no predicate transfer and one-hop transfer only, a single filter predicate cannot be sent to both the main query and the subquery to pre-filter the corresponding fact table. Conversely, _{PREDTRANS} broadcasts every filter predicate globally inside the join graph, such that both fact tables in the main query and subquery can be filtered. Moreover, Q17 joins base tables with aggregation results. By executing the aggregation beforehand, predicate transfer is able to achieve a higher selectivity by starting transfer from a smaller intermediate result table. On SF10, Q2 and Q5 achieve more improvement on _{PREDTRANS} than SF1. The speedup on Q18 is similar on both data sets. However, for query Q17 and Q21, compared to SF1, the speedup on SF10 of _{PREDTRANS} shrinks — _{PREDTRANS} outperforms BLOOM_{JOIN} by $1.8\times$ on Q17 and $7.6\times$ on Q21. On Q17, the grouped aggregation on the large fact table dominates the execution time, where the pre-filtering cannot make significant impact. Q21 involves a large amount of joins, which amplifies the amount of false positives brought by Bloom filters. As a result, the amount of remaining rows after the transfer phase is still about $2\times$ as the amount of rows that actually contribute to the final join output.

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

Another interesting observation is that YANNAKAKIS may not always outperform NO_{PREDTRANS} and BLOOM_{JOIN} baselines. For example, YANNAKAKIS underperforms BLOOM_{JOIN} by $16\times$ and $20\times$ on Q11 on both data sets respectively, and by $4.8\times$ on Q16 with SF10. One cause is that the Yannakakis algorithm does not specify the root of the join tree in the semi-join phase, and a bad semi-join order may construct several large hash tables at the beginning. However, this is not an issue in _{PREDTRANS} since we use a heuristic to transfer from smaller tables to larger tables (see Section 2.2.2), minimizing the memory stalls incurred by bitmap operations.

We further identify queries that have operators that block predicate transfer (e.g. outer joins), which are Q13, Q15, Q16, and Q22. Q13 and Q16 have left or right

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

2.4.3 Case Study: TPC-H Q5

To get a deeper understanding of the performance benefits, we conduct a detailed analysis on Q5, one of the complex queries in TPC-H. The query performs inner joins across six tables, and the join graph is shown in Figure 2.1a.

Table 2.1: Join Table Size in Q5 (SF = 1) — *HT* denotes the number of rows in the hash table, and *PR* represents the number of rows that probe the hash table.

	NoPredTRANS		BLOOMJOIN		YANNAKAKIS		PredTRANS	
	HT	PR	HT	PR	HT	PR	HT	PR
Join 1	10K	6M	10K	6M	2K	181K	2K	74K
Join 2	228K	6M	228K	1M	133K	181K	44K	67K
Join 3	150K	910K	150K	44K	69K	193K	15K	60K
Join 4	25	36K	25	36K	5	8K	5	7K
Join 5	1	36K	1	7K	1	8K	1	7K

Table 2.2: Join Table Size in Q5 (SF = 10) — *HT* denotes the number of rows in the hash table, and *PR* represents the number of rows that probe the hash table.

	NoPredTRANS		BLOOMJOIN		YANNAKAKIS		PredTRANS	
	HT	PR	HT	PR	HT	PR	HT	PR
Join 1	100K	60M	100K	60M	20K	1.8M	20K	835K
Join 2	2.3M	60M	2.3M	9.3M	1.2M	1.8M	305K	659K
Join 3	1.5M	9.1M	1.5M	499K	689K	1.9M	150K	376K
Join 4	25	364K	25	364K	5	75K	5	73K
Join 5	1	364K	1	73K	1	75K	1	73K

Join Table Size. We measure the sizes of both input tables of each join, following the join order specified in the query plan (FPDB utilizes Apache Calcite [25] for query

optimization like join ordering), and the results are shown in Table 2.1 and Table 2.2. PREDTRANS reduces the join table size by 98% over NOPREDTRANS, 96% over BLOOMJOIN, and 64% over YANNAKAKIS on SF1 (Table 2.1), and 98% over NOPREDTRANS, 92% over BLOOMJOIN, and 67% over YANNAKAKIS on SF10 (Table 2.2). In BLOOMJOIN, the largest fact table `lineitem` can only be pre-filtered after the first join, where the inner table `Orders` owns local filter predicates that can be transferred to `lineitem`. PREDTRANS shows the superiority to be able to pre-filter all join tables ahead of the entire join phase.

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

Performance Breakdown. Figure 2.6 demonstrates the performance breakdown of Q5 in different predicate transfer strategies. The execution time is divided into pre-filter time and join time. Compared to NOPREDTRANS and BLOOMJOIN, joins are accelerated by $44\times$ and $34\times$ in PREDTRANS on SF1, as well as $60\times$ and $46\times$ in SF10, due to the significant size reduction of the input join tables. YANNAKAKIS is also able to achieve a shrinkage on the input join tables, but the pre-filtering power is $3.3\times$ and $4.7\times$ less than PREDTRANS on SF1 and SF10 respectively since not all edges in the join graph are traversed. Moreover, the semi-joins it relies on are computationally expensive and dominate the entire execution time. As a result, the predicate transfer phase in PREDTRANS outperforms the semi-join phase in YANNAKAKIS by $13\times$ and $16\times$ on both data sets respectively, since bit operations used in Bloom filters are much cheaper than the construction and probe of the hash tables.

Robustness. We next evaluate the sensitiveness on join orders for different pre-filtering strategies. We pick three different join orders and the result is shown in Figure 2.7. In both data sets, PREDTRANS achieves the best performance and outperforms other predicate transfer strategies on all the join orders. Notably, the join order makes a much smaller performance variance in PREDTRANS (up to 12%) compared to other strategies (up to $45\times$). We further notice that for just the join phase, YANNAKAKIS achieves the similar robustness as PREDTRANS does. Both YANNAKAKIS

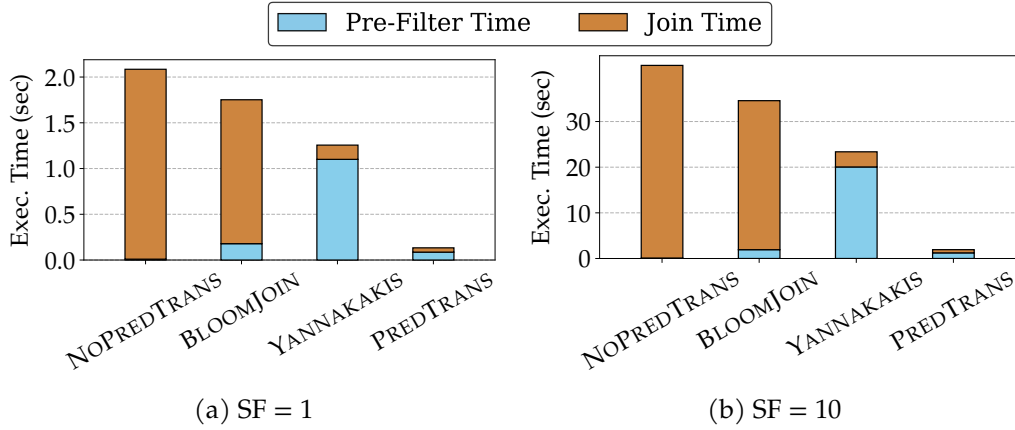


Figure 2.6: Performance Breakdown on TPC-H Q5.

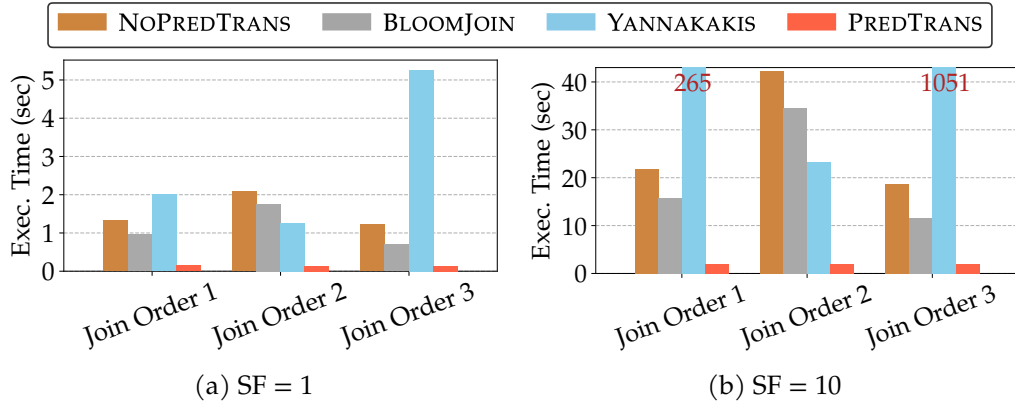


Figure 2.7: Performance of Different Join Orders on TPC-H Q5.

and PREDTRANS is able to bound the size of the intermediate join results, making their join phase robust to different join orders. The performance of the expensive semi-joins used in YANNAKAKIS is unstable since the join tree construction is not deterministic, making it overall highly sensitive to the join order. Conversely, the heuristic adopted in the predicate transfer phase of PREDTRANS always points to the same transfer topology and schedule regardless of the join order.

2.4.4 Join-Order Robustness with the LargestRoot Heuristic

Section 2.4.3 presents preliminary results of performance robustness across multiple join orders using the default Small2Large predicate transfer heuristic. In this subsection, we evaluate the robustness of the alternative LargestRoot heuristic, which guarantees a full reduction of the input joining tables. We implement predicate trans-

Table 2.3: Robustness Factors for Left-Deep Joins.

RF	TPC-H			JOB			TPC-DS		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
DuckDB	2.7	1.2	9.3	30.4	1.1	371	7.2	1.0	224
RPT	1.3	1.2	1.5	1.2	1.0	1.6	1.1	1.0	1.5

fer with LargestRoot into DuckDB (v0.9.2) [166], by adding an extra optimization submodule and inserting Bloom filter operations into the query plan³. We conduct the experiments on a physical machine with two Intel[®]Xeon[®] Platinum 8474C @ 2.1GHz, 512GB DDR5 RAM, and 8TB Samsung 870 QVO SATA III 2.5" SSD. The operating system is Debian 12.5.

We compare vanilla DuckDB (labeled as DuckDB) against DuckDB equipped with robust predicate transfer (labeled as RPT) using four standard benchmarks: TPC-H (SF = 100) [54], Join Order Benchmark (JOB) [139], TPC-DS (SF = 100) [53], and DSB (SF = 100) [96]. In this thesis, we focus on evaluating single-threaded, in-memory execution with *left-deep* join plans. Additional evaluation results for multi-threaded execution, *bushy* join plans, and on-disk data are reported in [222, 223].

We modified DuckDB’s optimizer to generate random join orders. For each evaluated query, we generate N random *left-deep* plans and N random *bushy* plans, where N is proportional to the number of joins m in that query. Specifically, we set N = 20 for the simplest 3-join queries and N = 1000 for the most complex query (i.e., Query 29 from JOB) with 17 joins, and therefore $N = 70m - 190$ for $3 \leq m \leq 17$. To produce a left-deep plan, we randomly pick a base table that is joinable⁴ with the current intermediate result as the rightmost leaf, repeating this process until all join tables are exhausted.

2.4.4.1 Acyclic Queries

Figure 2.8 shows the distribution of the end-to-end execution time of the random left-deep plans for each query. We omit queries in TPC-H with less than two joins because they are trivial in terms of join ordering. For JOB queries, we present one result for each of the 33 query templates. The execution times (for the baseline and RPT) for each query are normalized by the time t_{opt} of DuckDB running its default

³<https://github.com/embryo-labs/Robust-Predicate-Transfer>

⁴Has an edge in the join graph, i.e., no Cartesian product.

optimizer’s plan. The figure is in log scale with the normalization line (i.e., horizontal zero) highlighted. We set the timeout to $1000 \times t_{\text{opt}}$. The ‘*’ above a bar indicates that at least one of the random plans incurs a timeout for this query. Cyclic queries are marked by red query numbers.

We observe impressive join order robustness when using RPT in DuckDB for all acyclic queries. To quantify this, we define the *robustness factor* (RF in short) as the ratio between the maximum and the minimum execution time. Table 2.3 presents the average, min, and max RFs for DuckDB and RPT in each benchmark⁵. The average RF for DuckDB with RPT is consistently near 1 with the max (i.e., worst-case) RF = 1.6 for Query 17e in JOB. This is orders-of-magnitude more robust compared to the baseline.

The above results are encouraging. They show that join order optimization might no longer be a critical challenge for acyclic queries (which are the majority) if we implement joins using RPT. In fact, the execution time of RPT using the optimizer’s join order is within the horizontal lines (i.e., min to max excluding outliers) for every acyclic query in Figure 2.8. Future optimizers could, therefore, become much more efficient: they can better tolerate cardinality estimation errors, and they require simpler join enumeration algorithms because a left-deep plan is already good enough.

A few acyclic queries (13 and 48) in TPC-DS have slightly larger variances than the others in Figure 2.8c. Query 13 and 48 include predicates that cannot be pushed down before the tables are joined in DuckDB, e.g., $(R.a < 100 \text{ AND } S.b < 200) \text{ OR } (R.a > 500 \text{ AND } S.b > 400)$. It is preferable to join R and S earlier if the predicate is selective. Although these special cases exhibit adequate robustness with random plans, they can still benefit from the optimizer.

2.4.4.2 Cyclic Queries

RPT does not provide robustness guarantees for cyclic queries, as shown by the red-labeled queries in Figure 2.8 (i.e., TPC-H Q5, TPC-DS 19, 24, 46, 64, 68, 72, and 85). Although RPT improves the execution time in most cases, the performance gap between the best and worst plans for a cyclic query is still huge. We propose that a robust execution engine in the future should adopt a hybrid approach to handle joins: executing the cyclic part of the query using worst-case optimal joins while processing the rest with RPT.

⁵The result for DSB is similar to that for TPC-DS and can be found in the technical report [223].

2.5 Conclusion

A new join optimization, predicate transfer, is proposed in this chapter. Inspired by the Yannakakis algorithm, predicate transfer generalizes Bloom join to transfer table-local filters to pre-filter multiple other tables. We lay out the design space of predicate transfer and described the heuristics employed in our prototype. Evaluation shows an average $3.3\times$ speedup over Bloom join on TPC-H benchmark. Moreover, we develop an additional transfer heuristic to enable robust query processing with strong theoretical guarantees. Our evaluation shows that robust predicate transfer ensures a small variation in the execution time between random join orders for acyclic queries. We hope that our results advance the state-of-the-art of robust SQL analytics and will simplify the join optimization logic in future query optimizers.

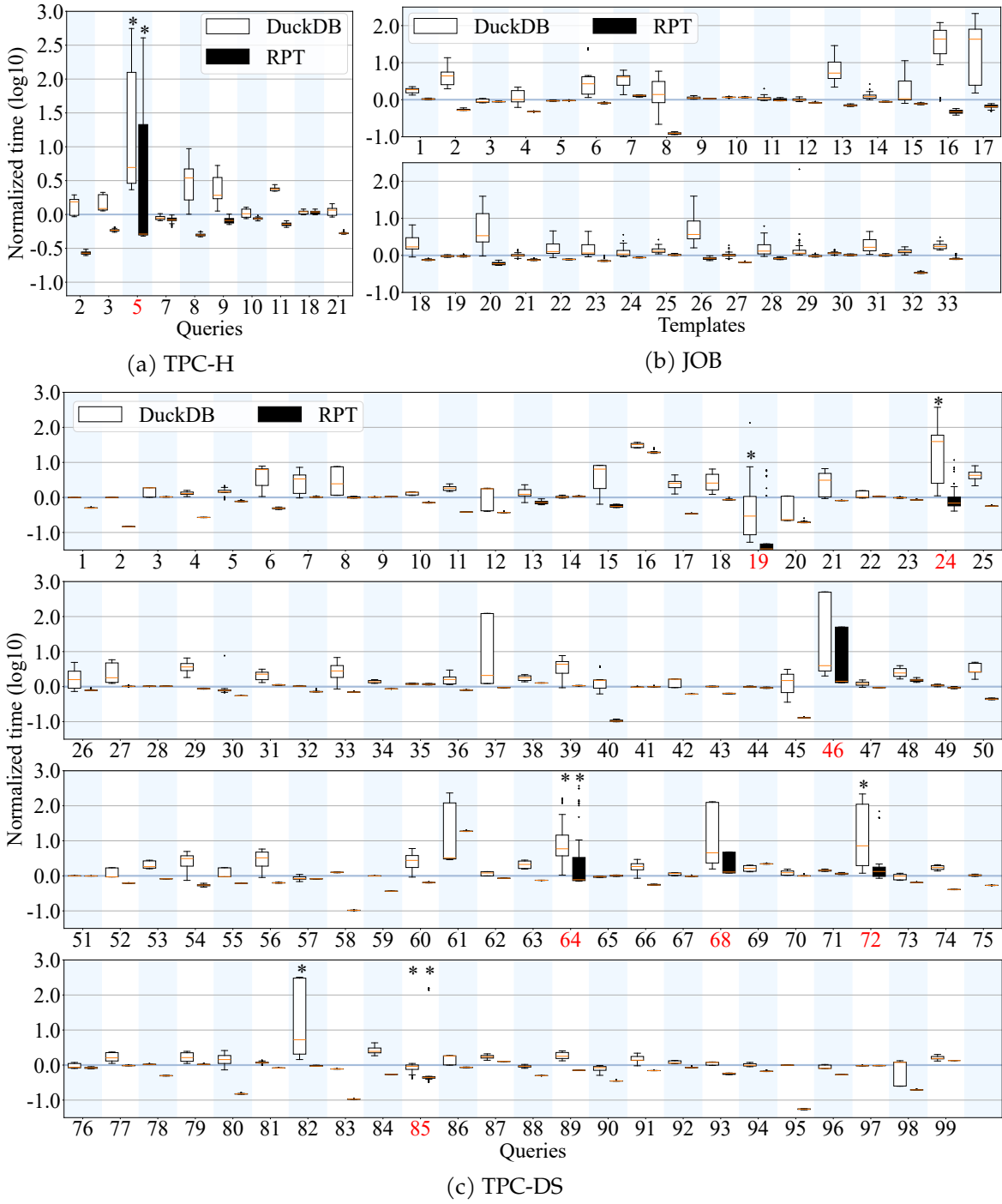


Figure 2.8: Distribution of the execution time of random left-deep plans for each query in TPC-H, JOB, and TPC-DS – Normalized by the execution time of default DuckDB. The figure is log-scaled. The box denotes 25- to 75-percentile (with the orange line as the median), while the horizontal lines denote min and max (excluding outliers). '*' indicates timeouts. Cyclic queries are in red.

Chapter 3

Accelerate Distributed Joins with Predicate Transfer

As presented in Chapter 2, predicate transfer [211] 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. Despite the performance benefit and robustness for join processing, 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 chapter, 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, this work 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 [208, 210], an open-source¹ distributed analytics query engine. Results on TPC-H and DSB benchmarks show that distributed predicate transfer can improve the query performance by over $3\times$, and reduce data exchange by over $2.7\times$.

The rest of the chapter is organized as follows. In Section 3.1, we propose multiple designs of distributed predicate transfer algorithm. Section 3.2 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

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

transfer step. In Section 3.3, we discuss the pruning algorithm for ineffective transfers. Section 3.4 presents additional optimization and implementation details. Section 3.5 evaluates the performance of the proposed techniques. Finally, Section 3.6 discusses the related work and Section 3.7 summarizes the chapter. This chapter is adapted from previously published work [209].

3.1 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, as discussed, supporting predicate transfer in a distributed environment is non-trivial with multiple challenges.

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. Broadcast-based distributed joins and shuffle-based distributed joins [106] are the most prominent algorithms in modern distributed query engines [64, 14, 37, 121]. Following this, we design broadcast-based (Section 3.1.1) and shuffle-based (Section 3.1.2) 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.

3.1.1 Broadcast-Based Approaches

3.1.1.1 Broadcast Join Key Values (BCAST-VAL)

Figure 3.1 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

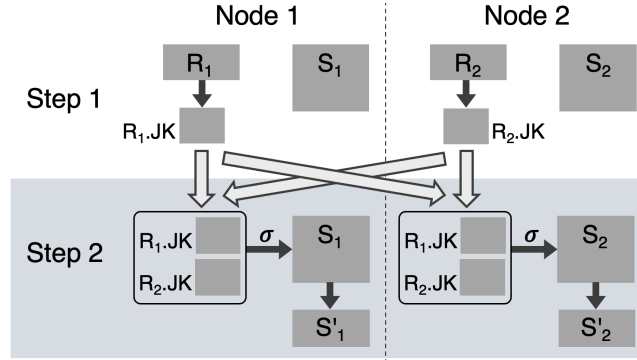


Figure 3.1: Broadcast-Based Value-Exchange (BCAST-VAL) — Transfer is from the source table R to the destination table S .

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 [216, 192] also adopted this approach, but used semi-joins instead. We observed a $3\times$ 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.

3.1.1.2 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 3.2a 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

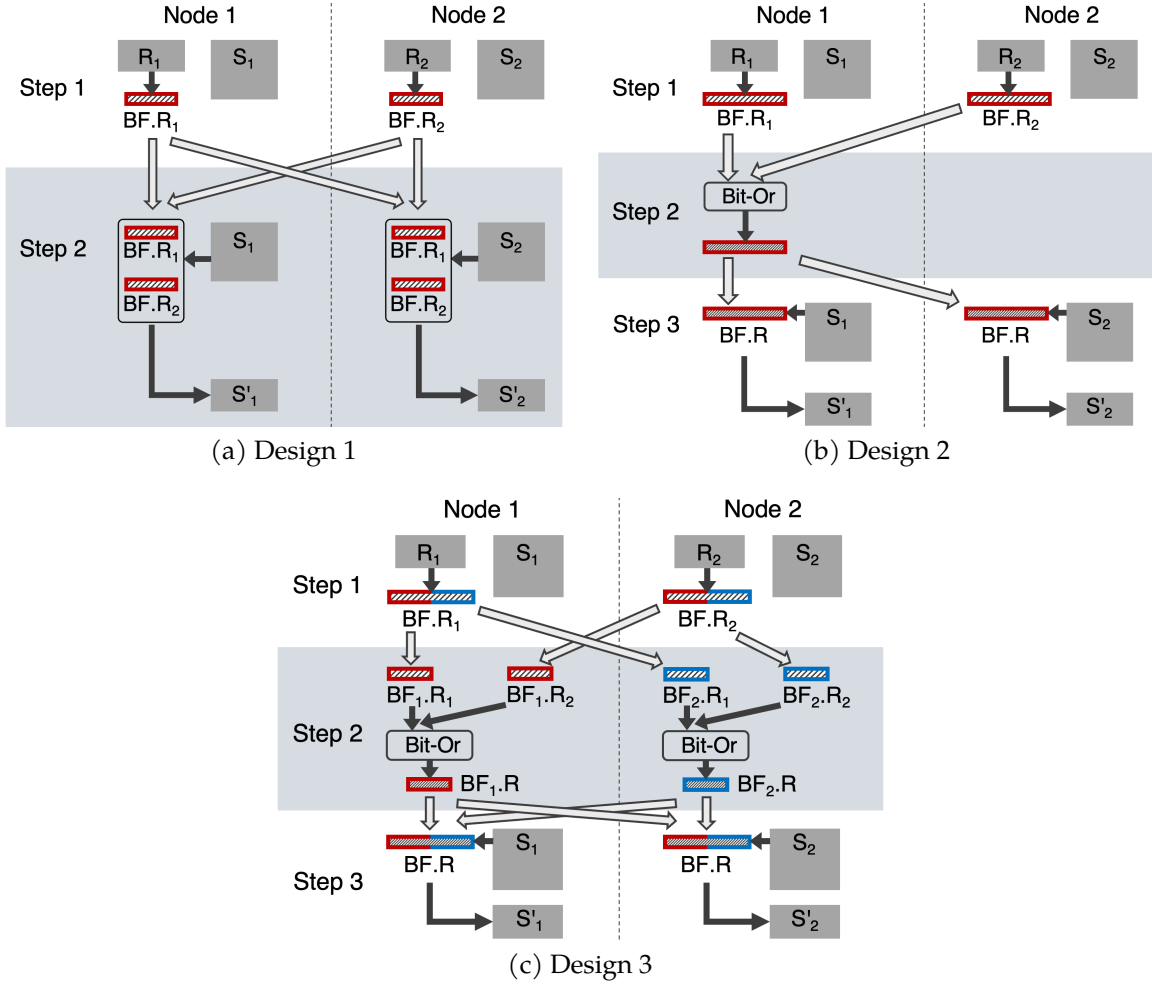


Figure 3.2: Broadcast-Based Bloom Filter Exchange (BCAST-BF) — Transfer is from the source table R to the destination table S .

false positive rate of p . Then the overall false positive rate p' of probing n Bloom filters is expressed as follows.

$$p' = 1 - (1 - p)^n \approx np \quad (3.1)$$

As Equation 3.1 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 [61].

$$m = \frac{|\ln p|}{(\ln 2)^2}, \quad m' = m + \frac{\ln n}{(\ln 2)^2} \quad (3.2)$$

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 3.2b 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 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 3.2c. 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 3.1 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,

Table 3.1: 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	$\times$	$\checkmark$	$\checkmark$
Load Balancing	$\checkmark$	$\times$	$\checkmark$

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 3.5.3.4), Design 3 achieves the best overall performance. Therefore, for the rest of the chapter, the term BCAST-BF will by default refer to Design 3.

3.1.2 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.

3.1.2.1 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 3.3a 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

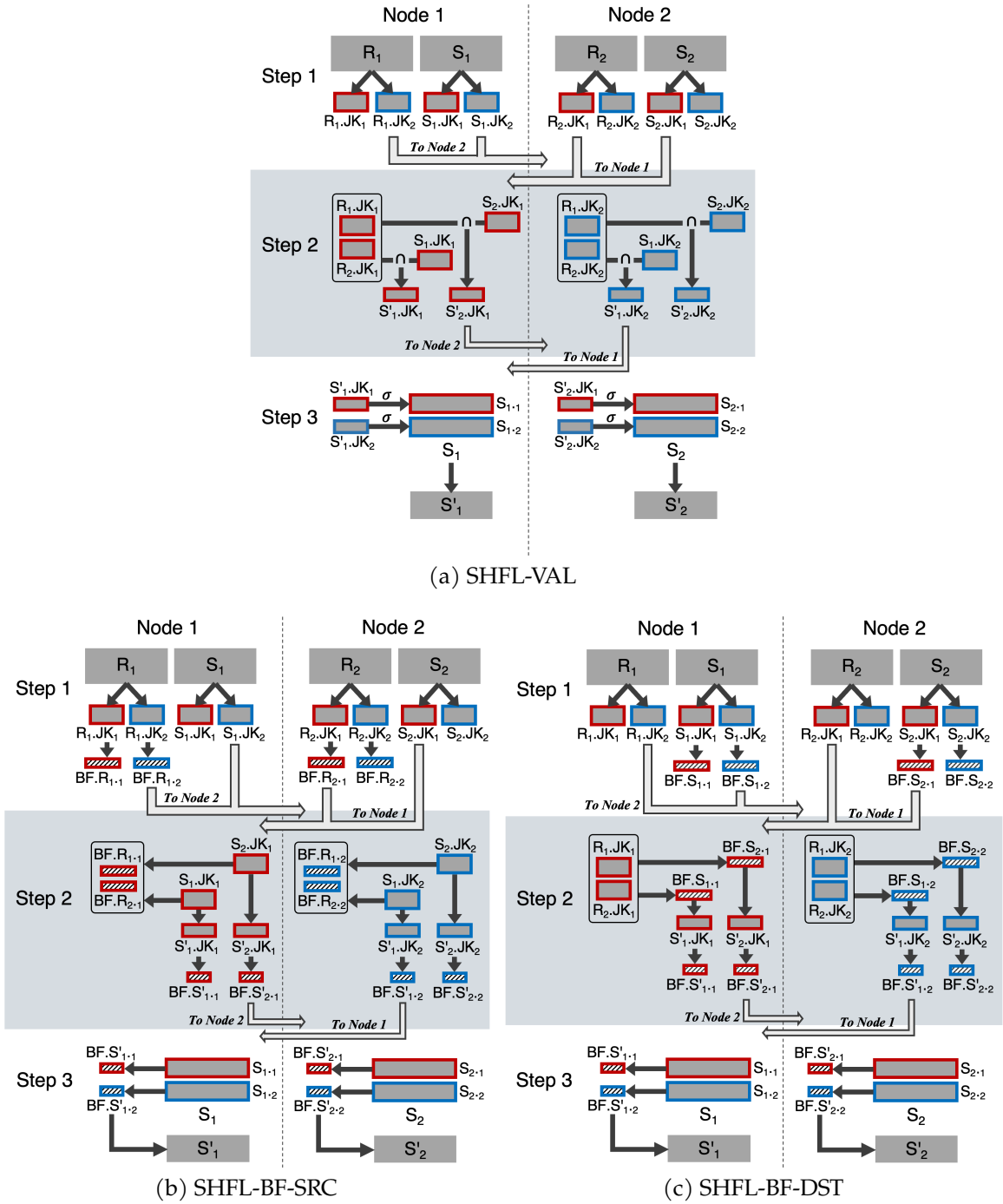


Figure 3.3: Shuffled-Based Distributed Predicate Transfer — Transfer is from the source table R to the destination table S .

tables at each node locally, forming the filtered join key for the destination table S (Step 2). Specifically, we compute the intersection of the aggregated input of table R (e.g., $R_1.JK_1 + R_2.JK_1$) and each input of table S (e.g., $S_1.JK_1, S_2.JK_1$). This allows us to isolate the filtered join keys (e.g., $S'_1.JK_1, S'_2.JK_1$) among different partitions of table S .

Afterwards, the filtered join keys are sent back to their original locations respectively. Specifically, $S'_2.JK_1$ is sent back to node 2 and $S'_1.JK_2$ 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).

3.1.2.2 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 3.3b 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., $BF.R_{1.1} + BF.R_{2.1}$) using each shuffled join key of table S (e.g., $S_1.JK_1, S_2.JK_1$).

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.

Table 3.2: 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

3.1.2.3 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 3.3c. For the source table R , we still shuffle the original join key column(s). In this design, to compute the intersection 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., $R_1.JK_1 + R_2.JK_1$).

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.

3.2 Cost-Based Adaptive Execution

In this section, we develop cost-based execution that selects the cheapest distributed 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 3.2 lists the required parameters used in the cost model, and Table 3.3 summarizes the derived cost for different distributed predicate transfer algorithms.

Table 3.3: 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	$\frac{n-1}{n}(N_{src}(l + npm) + N_{dst}(1 + s - ps)m)$

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 3.1.1.2.

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 $Cost_{send-back}$ below.

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

$$Cost_{shuffle}(SHFL-VAL) = \frac{n-1}{n}(N_{src} + N_{dst})l \quad (3.3)$$

After the intersection of the join keys is computed, the amount of filtered join keys that are sent back is shown in Equation 3.4.

$$Cost_{send-back}(SHFL-VAL) = \frac{n-1}{n}sN_{dst}l \quad (3.4)$$

SHFL-BF-SRC shuffles Bloom filters instead of the join key for the source table.

The shuffle cost can be expressed as follows.

$$\text{Cost}_{\text{shuffle}}(\text{SHFL-BF-SRC}) = \frac{n-1}{n} (N_{\text{src}} m' + N_{\text{dst}} l) \quad (3.5)$$

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 3.6.

$$p = \frac{N_{\text{filtered}} - sN_{\text{dst}}}{(1-s)N_{\text{dst}}} \quad (3.6)$$

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

$$\begin{aligned} \text{Cost}_{\text{send-back}}(\text{SHFL-BF-SRC}) &= \frac{n-1}{n} N_{\text{filtered}} m \\ &= \frac{n-1}{n} (p + s - ps) N_{\text{dst}} m \end{aligned} \quad (3.7)$$

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.

$$\text{Cost}_{\text{shuffle}}(\text{SHFL-BF-DST}) = \frac{n-1}{n} (N_{\text{src}} l + N_{\text{dst}} m) \quad (3.8)$$

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.

$$p = \frac{N_{\text{filtered}} - sN_{\text{dst}}}{nN_{\text{src}} - sN_{\text{dst}}} \quad (3.9)$$

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

$$\begin{aligned} \text{Cost}_{\text{send-back}}(\text{SHFL-BF-DST}) &= \frac{n-1}{n} N_{\text{filtered}} m \\ &= \frac{n-1}{n} (npN_{\text{src}} + (1-p)sN_{\text{dst}}) m \end{aligned} \quad (3.10)$$

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.

3.3 Pruning

The existing predicate transfer algorithm [211] processes all transfer steps entirely, as shown in Figure 2.1b. 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 3.3.1) and *predicate containment* (Section 3.3.2), and then develop a pruning algorithm based on these insights (Section 3.3.3).

3.3.1 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 [54] schema, seven of them satisfy the reversed containment. In this work 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.

3.3.2 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 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 contain-

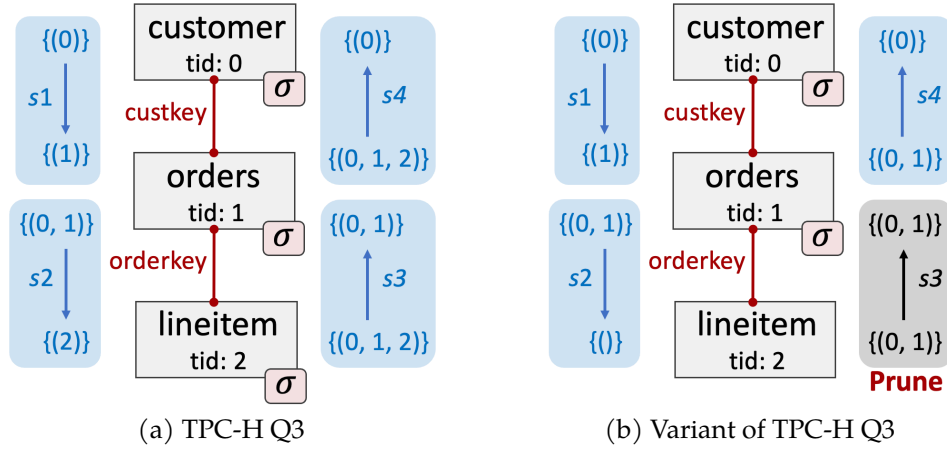


Figure 3.4: Pruning for Predicate Transfer in TPC-H Q3 and its variant.

ment 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 3.4a 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 s_1 happens between customer ($tid = 0$) and orders ($tid = 1$). After that, orders expands its predicate origins by absorbing the predicates of customer. At transfer s_3 , 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 3.4b, 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 s_3 — 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 s_3 also preserves join key containment, it can be pruned.

3.3.3 The Pruning Algorithm

Algorithm 2: Pruning for Predicate Transfer Steps

Input: Predicate Transfer Graph $G = (V, E)$

```

1 forward  $\leftarrow$  true
2 foreach  $v$  in  $V$  do
3    $v.\text{predOrigins} \leftarrow \emptyset$ 
4   if HASLOCALPRED( $v$ ) then
5      $v.\text{predOrigins}.\text{AddLOCAL}()$ 
6  $E_{\text{sort}} \leftarrow \text{TOPOLOGICALSORT}(E)$ 
7 foreach  $e = (\text{src}, \text{dst})$  in  $E_{\text{sort}}$  do
8    $e.\text{prune} \leftarrow \text{true}$ 
9   if not JOINKEYCONTAINMENT( $\text{src}, \text{dst}$ ) then
10     $e.\text{prune} \leftarrow \text{false}$ 
11     $\text{dst}.\text{predOrigins}.\text{AddTRANSFERRED}(\text{src})$ 
12   if  $e.\text{prune}$  then
13     $e.\text{prune} \leftarrow \text{PREDICATECONTAINMENT}(\text{src}.\text{predOrigins}, \text{dst}.\text{predOrigins})$ 
14   if not  $e.\text{prune}$  then
15     $\text{dst}.\text{predOrigins}.\text{AddTRANSFERRED}(\text{src}.\text{predOrigins})$ 
16 if forward then
17    $E_{\text{rev}} \leftarrow \text{REVERSE}(E)$ 
18    $E_{\text{sort}} \leftarrow \text{TOPOLOGICALSORT}(E_{\text{rev}})$ 
19   forward  $\leftarrow \text{false}$ 
20   goto line 7
  
```

Algorithm 2 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 prunability 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).

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).

3.4 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 [24] 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.

3.5 Evaluation

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

3.5.1 Experimental Setup

Server Configuration. We conduct all experiments using AWS EC2 [20] 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 [208, 210], an open-source distributed analytics query engine.

Benchmarks. We use the widely adopted data analytics benchmark, TPC-H [54]. 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 [96] benchmark, which contains 15 single-block queries and 22 multi-block queries. DSB is an adaptation of the widely-used industry standard TPC-DS [53] 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 [6, 17]. 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 QuickStep [160] and LIP [224]. These techniques were originally designed for single-node execution. 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.

3.5.2 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 3.5a reports the query execution time. On average, PT is $3\times$ faster than NoPT and $2.5\times$ 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 $5\times$

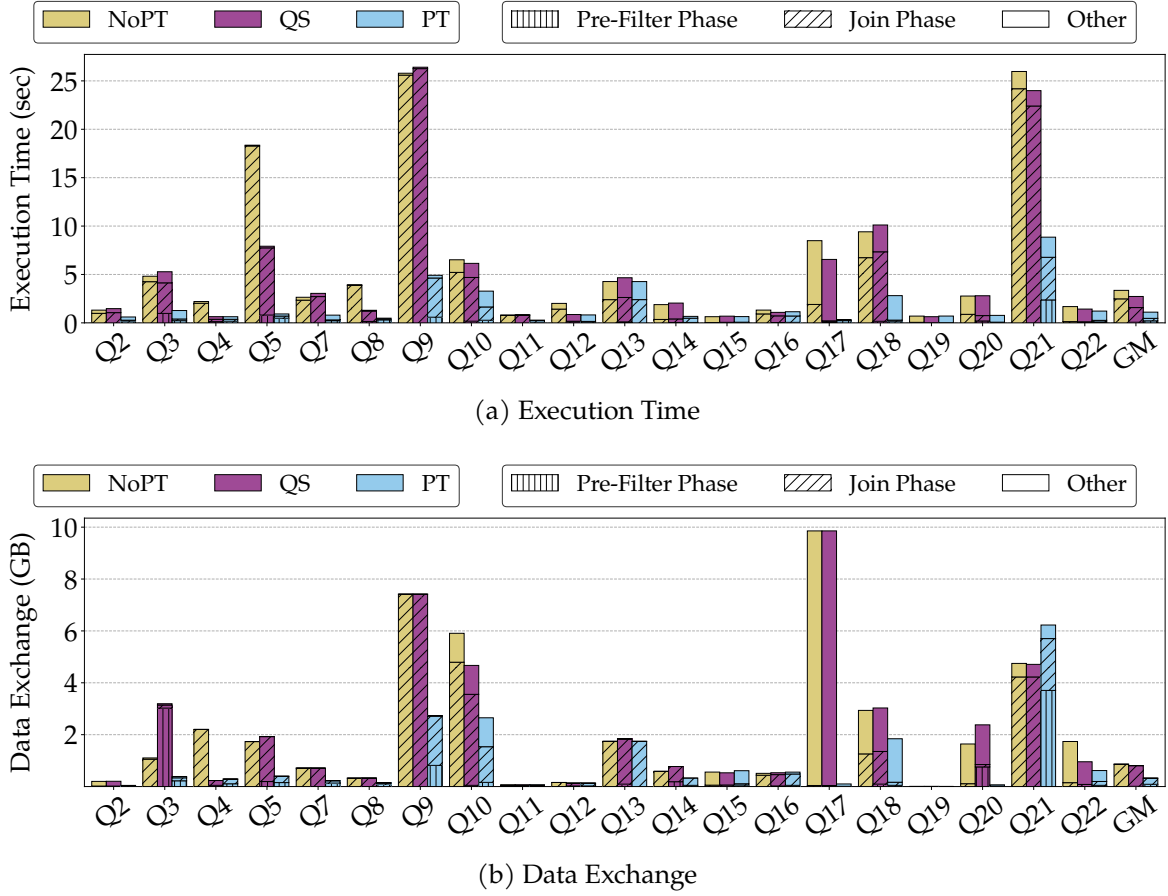


Figure 3.5: TPC-H Performance of Distributed Predicate Transfer (SF100, 4 nodes).

speedup over the both baselines. Specifically, PT outperforms NoPT and QS on Q5 by $20\times$ and $9\times$, respectively, and achieves speedups of $25\times$ and $20\times$ on Q17, respectively. Figure 3.5b presents the corresponding data exchange volume. PT reduces the amount of data exchanged by $2.7\times$ over NoPT, and by $2.5\times$ over QS. 13 queries show a reduction in network traffic over the both baselines. Specifically, PT can reduce network traffic by $99\times$ on Q17 over both NoPT and QS, and achieves a $24\times$ and $36\times$ 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 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 $3\times$. However, for queries like Q3, Q7, Q9, and Q18, QS slightly underperforms NoPT because the pre-filtering within left-deep join subtrees

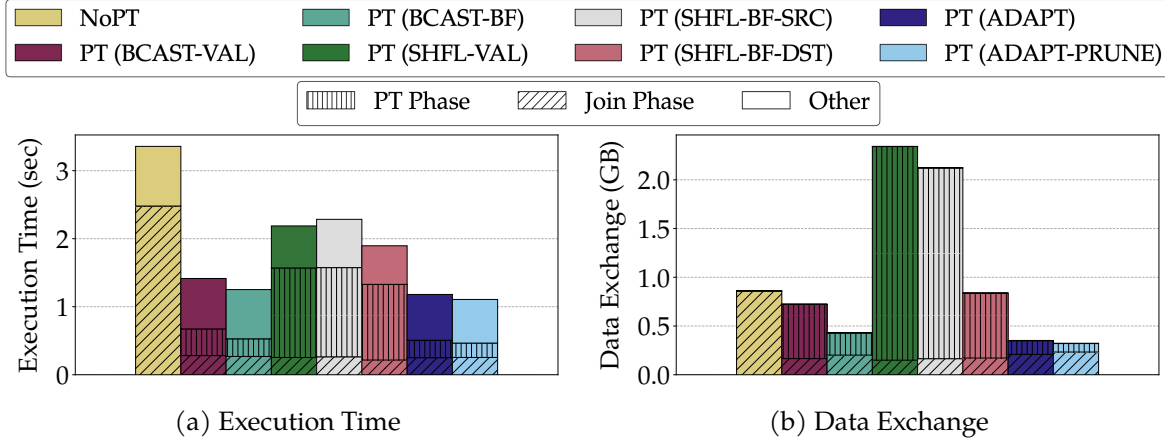


Figure 3.6: TPC-H Performance of Different Distributed Predicate Transfer Algorithms — GeoMean across all queries (SF100, 4 nodes).

eliminates only a small subset of unnecessary rows. On average, QS reduces the join input table size by $2.6\times$, while PT achieves a reduction of $17\times$.

To evaluate the overhead introduced by the predicate transfer algorithm, we present a performance breakdown in Figure 3.5. As Figure 3.5a 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 3.5b 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.

3.5.3 Distributed Predicate Transfer Algorithms

In this subsection, we compare performance among different variants of distributed predicate transfer algorithms (Section 3.5.3.1), and evaluate the effectiveness of both the cost-based adaptive execution (Section 3.5.3.2) and the pruning technique (Section 3.5.3.3). We also compare different design variants of BCAST-BF (Section 3.5.3.4).

3.5.3.1 Distributed Predicate Transfer Algorithms

Figure 3.6 shows the average performance of different distributed PT algorithms on TPC-H. As Figure 3.6a shows, all proposed distributed PT algorithms outperform NoPT, between the largest speedup of $2.7\times$ 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 3.6b. BCAST-VAL and BCAST-BF reduce amount of data exchange by 16% and $2\times$ respectively. Network traffic of SHFL-BF-DST is close to NoPT. SHFL-VAL and SHFL-BF-SRC increase the amount of data exchange by $2.7\times$ and $2.5\times$ 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.

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 3.6. 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 3.5.3.2 and Section 3.5.3.3, respectively.

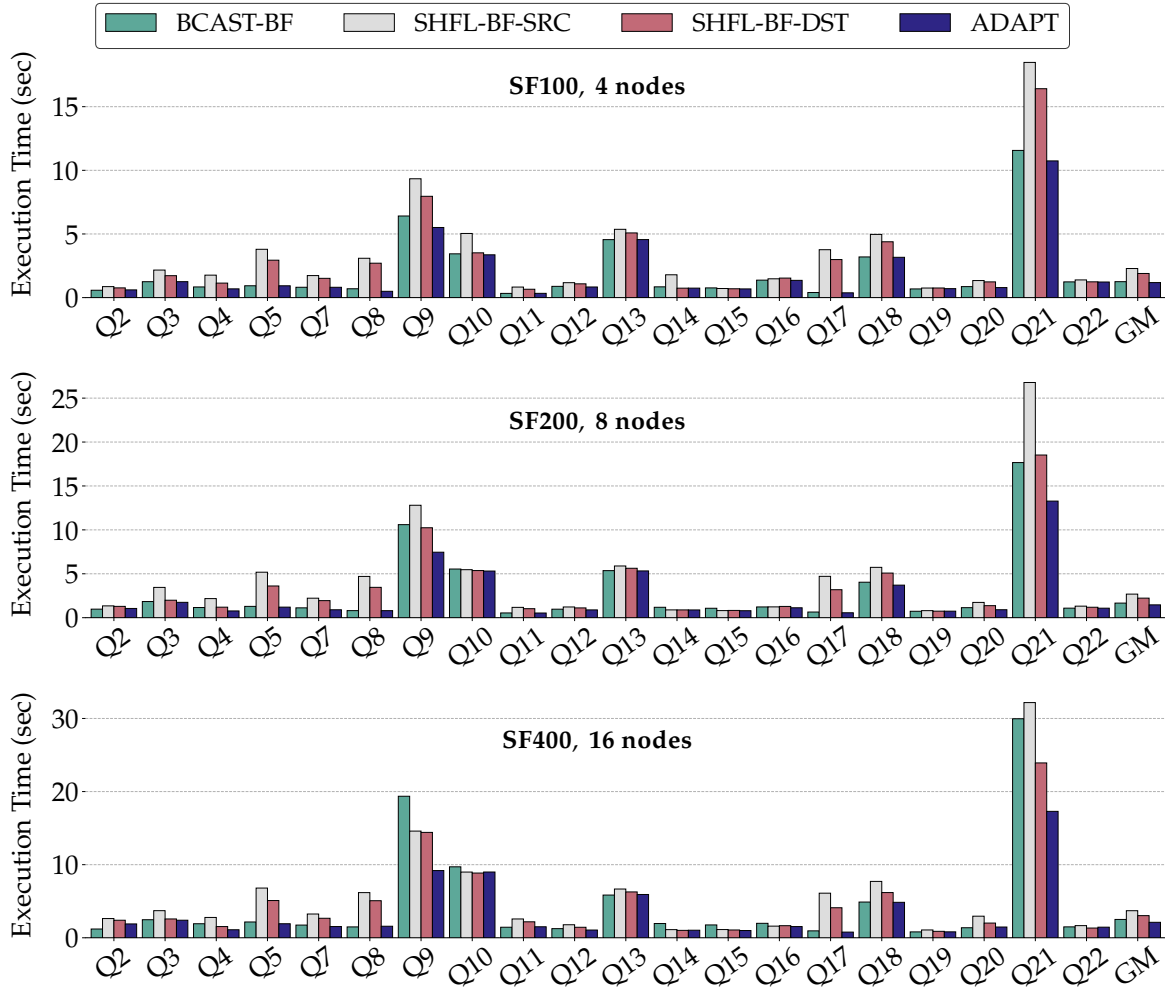


Figure 3.7: TPC-H Performance of Distributed Predicate Transfer Algorithm and Cost-Based Adaptive Execution on Individual Queries.

3.5.3.2 Cost-Based Adaptive Execution

Figure 3.7 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 3.7. 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).

Cost-based adaptive execution consistently outperforms all static algorithm vari-

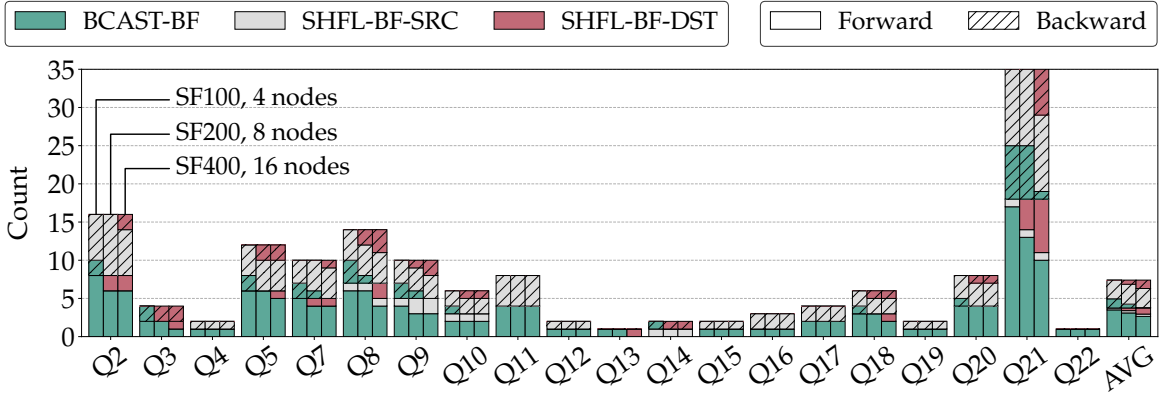


Figure 3.8: Breakdown of Selected Distributed Predicate Transfer Algorithms in Cost-Based Adaptive Execution on TPC-H.

ants 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 3.8. 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.

3.5.3.3 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 the number of pruned transfer steps on each query in Figure 3.9. 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 3.9b 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 3.9b 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 unnecessary.

One key property to satisfy for pruning is join key containment. Table 3.4 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

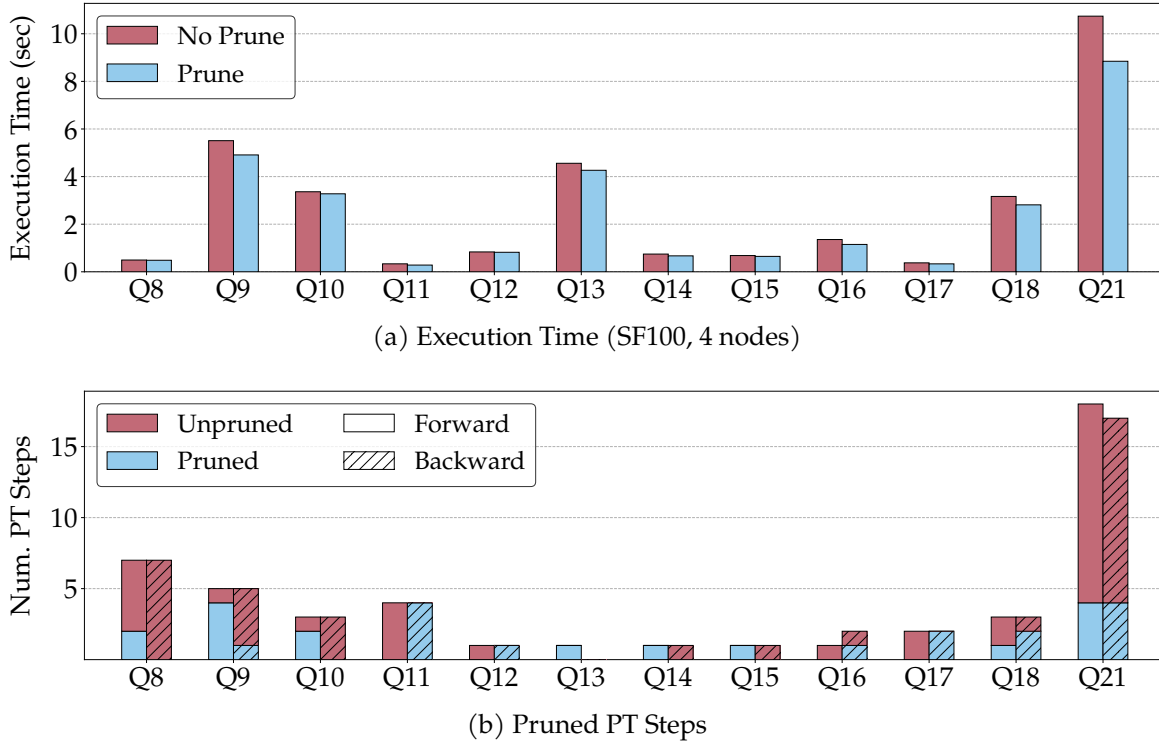


Figure 3.9: Amount of Transfer Step Pruning Occurred on TPC-H.

Table 3.4: Discovery of Join Key Containment on TPC-H.

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

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 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.

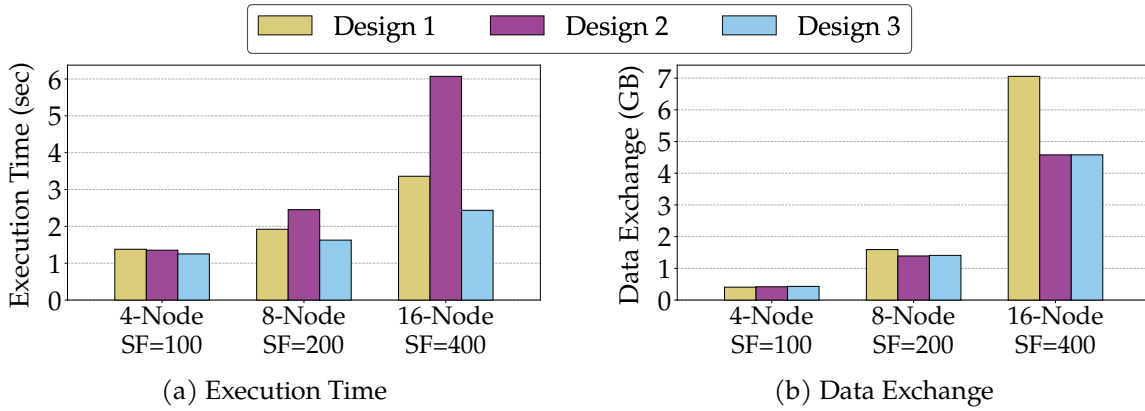


Figure 3.10: TPC-H Performance of BCAST-BF Designs — GeoMean of performance across all measured queries.

3.5.3.4 Design Variants of BCAST-BF

In this part, we evaluate variants of BCAST-BF proposed in Section 3.1.1.2. We conduct experiments under multiple scales of the cluster and data set. The results are shown in Figure 3.10.

As Figure 3.10a 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 $\times$ respectively; when the cluster is small (i.e., 4 nodes), it slightly outperforms the other two designs.

Figure 3.10b 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.

Table 3.5: 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

3.5.4 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 3.5 and 3.6, 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

Table 3.6: 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 $\rightarrow$ customer	PRUNED	/	/	/	/	/
	customer $\rightarrow$ orders	PRUNED	/	/	/	/	/
	orders $\rightarrow$ lineitem	BCAST-BF	148M	9%	8M	5	123
Backward	lineitem $\rightarrow$ orders	SHFL-BF-SRC	6M	86%	42M	19	38
	orders $\rightarrow$ customer	BCAST-BF	15M	26%	8M	6	46
	customer $\rightarrow$ nation	SHFL-BF-SRC	25	100%	9M	4	1

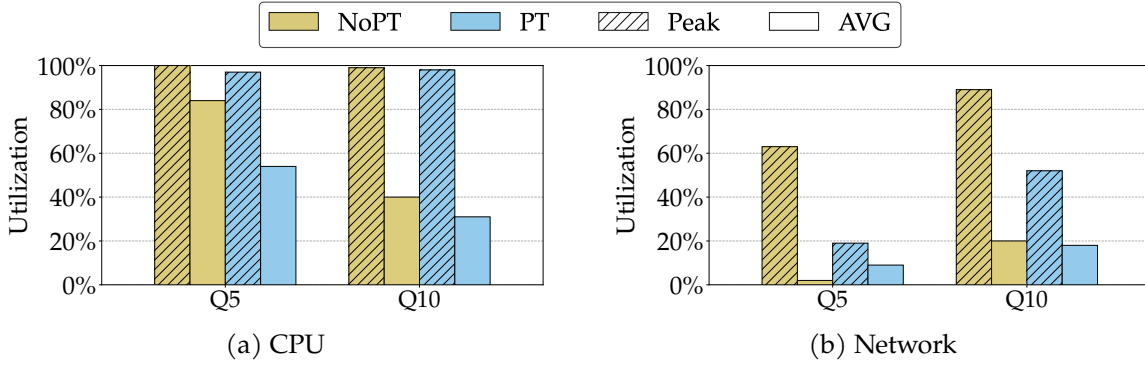


Figure 3.11: Resource Utilization on Representative Queries of TPC-H (SF100, 4 nodes).

128ms, which accounts for 4% of the query 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 3.11. In both cases, CPU utilization is higher than network utilization. In Q5, we observed that with PT, CPU utilization decreases while network utilization increases. This occurs because PT significantly reduces the

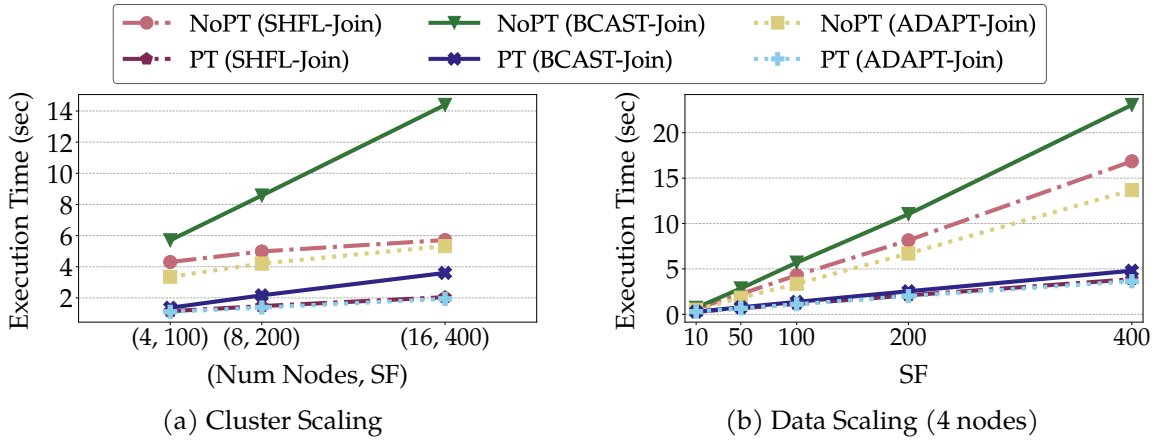


Figure 3.12: Scalability of Distributed Predicate Transfer on TPC-H — GeoMean of performance across all measured queries.

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.

3.5.5 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 3.12a 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 3.12b 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 $3\times$, $3.3\times$, and $3.8\times$ 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 $2.8\times$, 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.

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

Figure 3.13a compares query execution time of PT against NoPT and QS. On average, PT outperforms NoPT by $3.3\times$ 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 $10\times$ of the joining table sizes, whereas QS also reduces the input size by $2\times$.

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

Figure 3.13b presents data exchange volumes across different distributed pre-filtering solutions, in log-scale. Overall, PT reduces network traffic by $4\times$ and $3.1\times$ compared to NoPT and QS on single-block queries, and by 43% and $2\times$ 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,

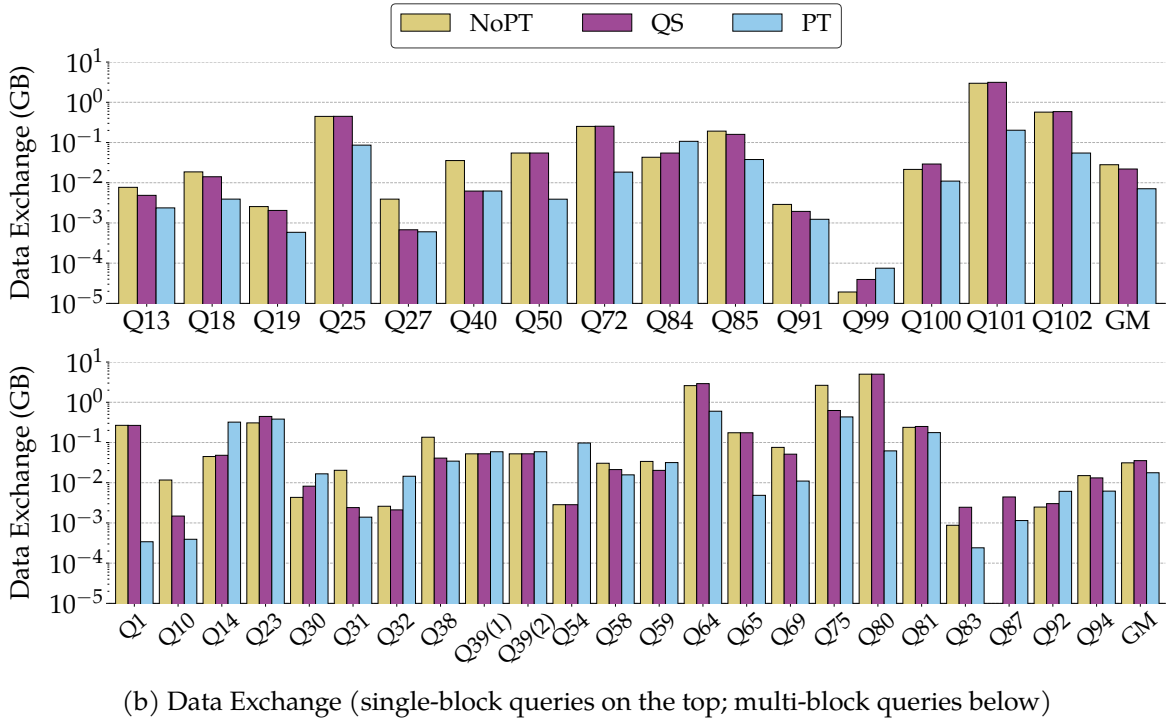
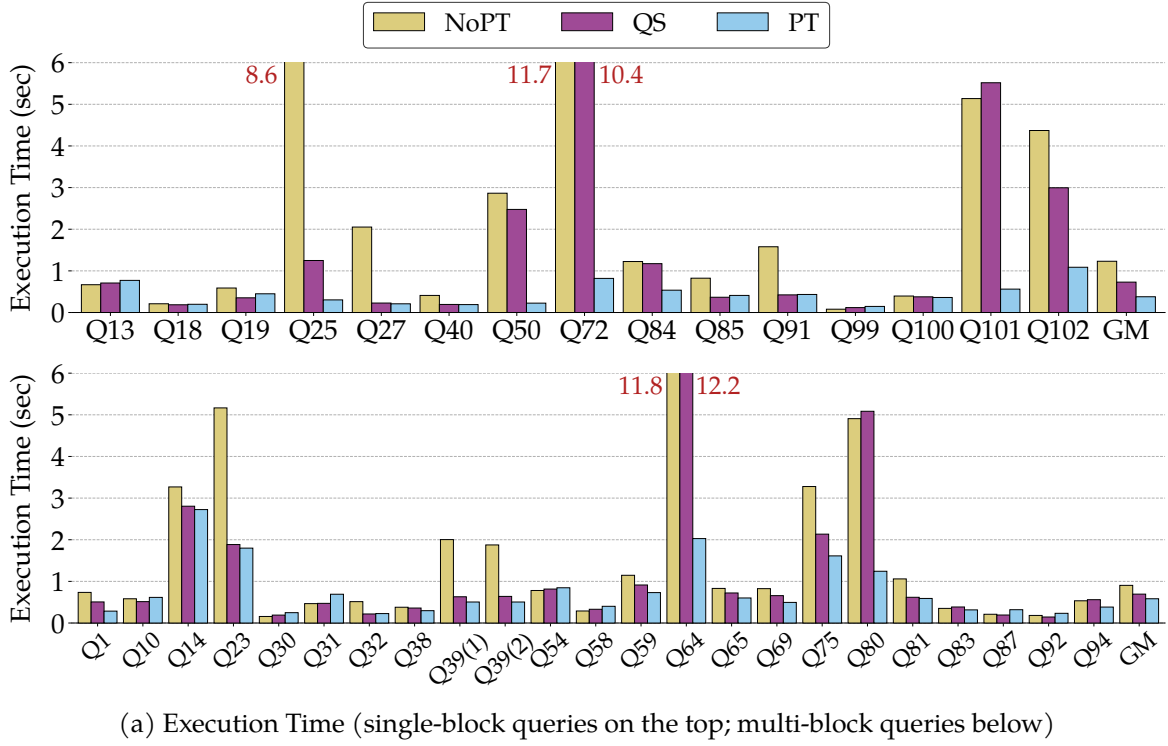


Figure 3.13: DSB Performance of Distributed Predicate Transfer (SF100, 4 nodes).

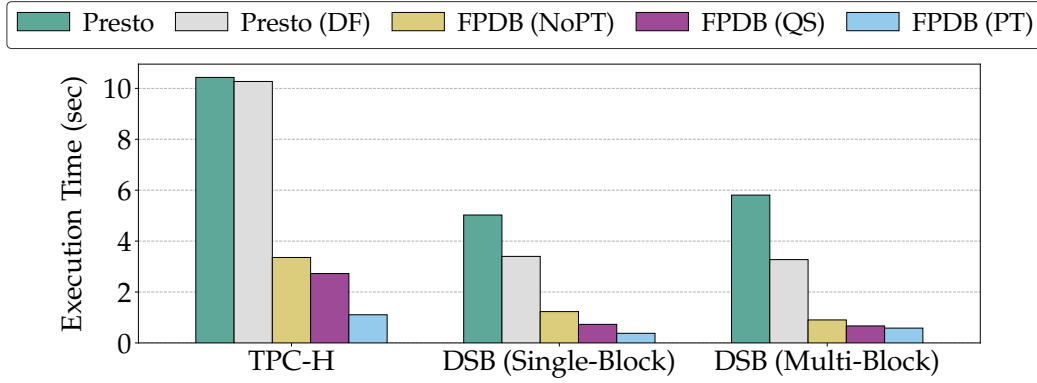


Figure 3.14: Performance Comparison of FPDB against Presto — GeoMean across all measured queries (SF100, 4 nodes); Presto (DF) represents Presto execution with “dynamic filtering” enabled.

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.

3.5.7 Comparison to Other Existing Systems

We further compare the performance against Presto [178, 184], 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 [13], 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” [3] which we also incorporate in the evaluation.

As shown in Figure 3.14, NoPT of FPDB outperforms vanilla Presto by $3.1\times$, $4.1\times$, and $6.4\times$ 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 $2.8\times$ across all benchmarks.

With PT, FPDB delivers even greater performance gains over Presto. Specifically, PT of FPDB surpasses vanilla Presto by $9.4\times$, $13.1\times$, and $10\times$ on TPC-H, DSB single-block queries, and DSB multi-block queries, respectively. Besides, it outperforms Presto with dynamic filtering by $9.3\times$, $9\times$, and $5.6\times$ 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.

3.6 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 [168, 79, 135] 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 [68, 149, 150] leverage SIP [72] to perform query rewriting, such that irrelevant rows are filtered out early. QuickStep [160] and LIP [224] generalized Bloom Joins to star schemas, and adaptively reorders the Bloom filters when applying to the fact table. Bitvector Filters [97] explored the plan space of right-deep trees to optimize star and snowflake queries. [76] 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) [127] 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 [74] is a well-known technique for efficiently pre-filtering the joining tables. The Yannakakis algorithm [212] performs semi-join reduction prior to the actual join, ensuring optimal theoretical pre-filtering for acyclic queries. Predicate transfer [211] 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 [74, 75, 85, 141,

84, 164, 86, 200, 182, 126, 109, 88]. 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.

[216, 192] 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 [163] 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 [106] are the most prominent algorithms in modern distributed query engines [64, 14, 37, 121]. [71] compared radix hash join and sort-merge join algorithms and discussed their implementation at a large scale. [174, 80] optimized distributed joins over skewed data partitions. SquirrelJoin [173] studied distributed joins on heterogeneous network topologies. [146] proposed optimized data placement to accelerate distributed 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.

3.7 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 chapter, 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 $3\times$, and reduce the amount of data exchange by over $2.7\times$.

Part II

Innovations in the Infrastructure Stack — Cloud- and GPU-Native Execution Engines

Chapter 4

Hybrid Pushdown and Caching in Cloud-Native Databases

Database management systems (DBMSs) are gradually moving from on-premises to the cloud for higher elasticity and lower cost. Modern cloud DBMSs adopt a *storage-disaggregation architecture* that divides computation and storage into separate layers of servers connected through the network, simplifying provisioning and enabling independent scaling of resources. However, disaggregation requires rethinking a fundamental principle of distributed DBMSs: “move computation to data rather than data to computation”. Compared to the traditional shared-nothing architecture, which embodies that principle and stores data on local disks, the network in the disaggregation architecture typically has lower bandwidth than local disks, making it a potential performance bottleneck.

Two solutions have been explored to mitigate this network bottleneck: *caching* and *computation pushdown*. Both solutions can reduce the amount of data transferred between the two layers. Caching keeps the hot data in the computation layer. Examples include Snowflake [92, 199] and Presto with Alluxio cache service [13]. The Redshift [112] layer in Redshift Spectrum [21] can also be considered as a cache with user-controlled contents. With computation pushdown, filtering and aggregation are performed close to the storage with only the results returned. Examples include Oracle Exadata [203], IBM Netezza [101], AWS Redshift Spectrum [21], AWS Aqua [70], and PushdownDB [217]. The fundamental reasons that caching and pushdown have performance benefits are that local memory and storage have higher bandwidth than the network and that the internal bandwidth within the storage layer is also higher than that of the network.

Existing DBMSs consider caching and computation pushdown as *orthogonal*. Most systems implement only one of them. Some systems, such as Exadata [203], Netezza [101], Redshift Spectrum [21], and Presto [13] consider the two techniques as independent: query operators can either access cached data (i.e., full tables) or push down computation on remote data, but not both.

In this work, we argue that caching and computation pushdown are *not* orthogonal techniques, and that the rigid dichotomy of existing systems leaves potential performance benefits unexploited. We propose *FlexPushdownDB* (FPDB in short), an OLAP cloud DBMS prototype that combines the benefits of caching and pushdown.

FPDB introduces the concept of *separable operators*, which combine local computation on cached segments and pushdown on the segments in the cloud storage. This hybrid execution can leverage cached data at a fine granularity. While not all relational operators are separable, some of the most commonly-used ones are, including filtering, projection, aggregation. We introduce a *merge* operator to combine the outputs from caching and pushdown.

Separable operators open up new possibilities for caching. Traditional cache replacement policies assume that each miss requires loading the missing data block to the cache, which incurs a constant cost if the blocks have the same size. In FPDB, however, this assumption is no longer true because we can push down computation instead of loading data. The cost of a miss now depends on how amenable the block is to pushdown — for misses that can be accelerated with pushdown (e.g., high-selectivity filters), the cost of a miss is lower. We develop a new benefit-based caching framework and a new caching policy called *Weighted-LFU*, which incorporates caching and pushdown into a unified cost model to predict the best cache eviction decisions. We compare Weighted-LFU with popular conventional policies, including *least-recently used* (LRU), *least-frequently used* (LFU), and an optimal policy called *Belady* [73] that assumes the availability of future access patterns. Our evaluation shows that *Weighted-LFU* outperforms all these policies.

In summary, this work makes the following key contributions:

- We develop a fine-grained hybrid execution mode for cloud DBMSs to combine the benefits of caching and pushdown in a storage-disaggregation architecture. The hybrid mode outperforms both *Caching-only* and *Pushdown-only* architectures by $2.2\times$ on the Star Schema Benchmark (SSB) [156].

- We study the performance effect of various caching management policies. We developed a novel *Weighted-LFU* replacement policy that is specifically optimized for the disaggregation architecture. *Weighted-LFU* outperforms the second-best policy (i.e., LFU) by 37% and Belady by 47%.
- We present the detailed design and implementation of FPDB, an open-source C++-based cloud-native OLAP DBMS¹. We believe it can benefit the community given the lack of cloud-DBMS prototypes.

The rest of the chapter is organized as follows. Section 4.1 presents the background and motivation of our design. After showing an overview of FPDB in Section 4.2, Sections 4.3, 4.4, and 4.5 discuss the hybrid execution model, the caching policy, and the detailed implementation of FPDB, respectively. Section 4.6 evaluates the performance of FPDB over baseline architectures and policies. Finally, Section 4.7 discusses related work and Section 4.8 summarizes this chapter. This chapter is adapted from previously published work [208].

4.1 Background and Motivation

This section describes the background on the storage-disaggregation architecture (Section 4.1.1) and the new challenges, as well as why existing systems are sub-optimal in solving the problem (Section 4.1.2).

4.1.1 Storage-disaggregation Architecture

According to the conventional wisdom, *shared-nothing* (Figure 4.1a) is the best architecture for high-performance distributed data warehousing systems, where servers with local memory and disks are connected through a network. While a cloud DBMS can also adopt a shared-nothing architecture, many cloud-native databases choose to disaggregate the computation and storage layers (Figure 4.1c). This brings benefits of lower cost, simpler fault tolerance, and higher hardware utilization. Many cloud-native databases have been developed recently following such an architecture, including Aurora [197, 198], Redshift Spectrum [21], Athena [19], Presto [178, 184], Hive [188], SparkSQL [64], Snowflake [92], and Vertica EON mode [136, 194].

¹https://github.com/cloud-olap/FlexPushdownDB/tree/vldb_21

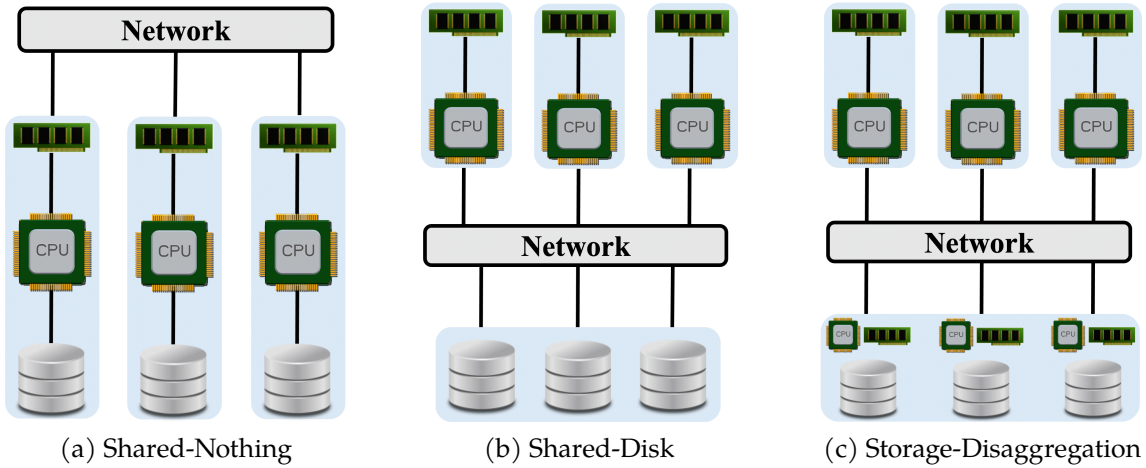


Figure 4.1: Distributed Database Architectures.

Besides the separation of computation and storage, the disaggregation architecture also supports *limited computation within the storage layer*. The computation can happen either on the storage nodes (e.g., Aurora [197]) or in a different layer close to the storage nodes (e.g., Redshift Spectrum [21] and Aqua [70]). This allows a database to push filtering and simple aggregation operations into the storage layer, leading to performance improvement and potential cost reduction as shown in previous work [217].

While the storage-disaggregation architecture sounds similar to the well-known *shared-disk* architecture (Figure 4.1b), they actually have significant differences. In a shared-disk architecture, the disks are typically centralized, making it hard to scale out the system. The disaggregation architecture, by contrast, can scale the storage layer horizontally (both the computation and the capacity) just like the computation layer. The disaggregation architecture also provides non-trivial computation in the storage layer, while disks are passive storage devices in the shared-disk architecture.

4.1.2 Challenges in Disaggregated DBMSs

The network connecting the computation and storage layers is a major performance bottleneck in the storage-disaggregation architecture. The network bandwidth is significantly lower than the disk IO bandwidth within a single server, leading to lower performance than a shared-nothing DBMS [186]. Existing cloud DBMSs use two solutions to alleviate this problem: *caching* and *computation pushdown*. Figure 4.2

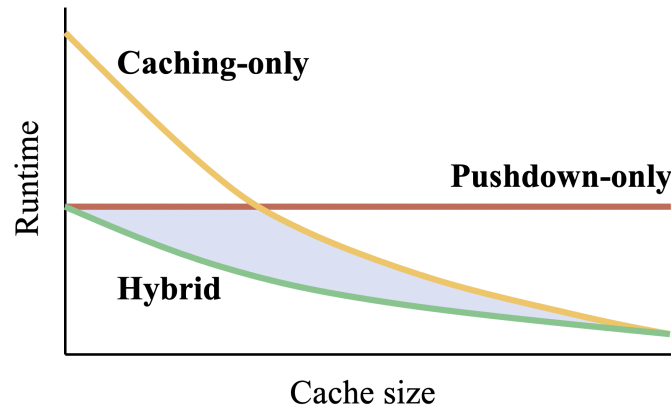


Figure 4.2: Performance trade-off between caching, computation pushdown, and an ideal hybrid approach.

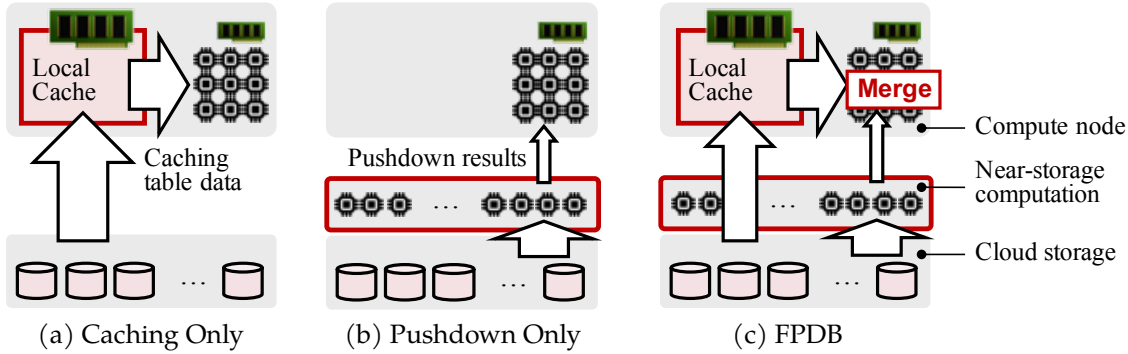


Figure 4.3: **System Architectures** — High level architectural comparison between *Caching-only*, *Pushdown-only*, and FPDB.

shows the high-level performance trade-off between the two approaches, which we describe below.

Solution 1: Caching. These systems keep hot data in the local memory or disks of the computation nodes. The high-level architecture is shown in Figure 4.3a. Cache hits require only local data accesses and are thus much faster than cache misses, which require loading data over the network. As shown in Figure 4.2, as the cache size increases, the query execution time decreases due to a higher hit ratio. Snowflake [92] and Presto [13] support caching with the LRU replacement policy. Redshift Spectrum [21] does not support dynamic caching but allows users to specify what tables to keep in the Redshift layer for fast IO. We are not aware of any thorough study of more advanced caching policies in these systems.

Solution 2: Computation pushdown. The idea here is to perform certain DBMS

operators close to where the data resides; the high-level architecture is shown in 4.3b. Computation pushdown has been widely explored in the context of database machines [190, 104, 101, 203], Smart Disk/SSD [99, 204, 105, 134, 111, 207], processing-in-memory (PIM) [108, 129], and cloud databases [21, 70, 147, 217]. They are all based on the observation that the internal bandwidth in the storage is higher than the external bandwidth between storage and computation and that pushdown can reduce data transfer; the problem is particularly severe in cloud databases due to disaggregation. As shown in Figure 4.2, the performance of computation pushdown is independent of the cache size — the DBMS does not exploit caching.

When the cache size is small, pushdown outperforms caching due to reduced network traffic; when the cache size is sufficiently large, caching performs better due to a higher cache hit ratio. Ideally, a system should adopt a hybrid design that combines the benefits of both worlds — caching a subset of hot data and pushdown computation for the rest, which is shown as the bottom line in Figure 4.2.

Existing systems do not offer a fully hybrid design. While some systems support both caching and pushdown, they select the operation mode at the table granularity. The storage layer keeps additional “external” tables that can be queried using computation pushdown. But no system, to the best of our knowledge, can dynamically cache external tables at a fine granularity (e.g., subsets of columns) without an explicit load command. A key challenge here is to split a general query to exploit both techniques at a fine granularity. Another key challenge is to design a caching policy that is aware of computation pushdown. FPDB addresses both challenges.

4.2 FPDB Overview

4.2.1 System Architecture

Figure 4.3c shows the high-level architecture of FPDB in contrast to existing architectures of *Caching-only* (Figure 4.3a) and *Pushdown-only* (Figure 4.3b). Similar to *Caching-only*, FPDB stores the hot input data in the local cache (i.e., main memory or disk) to take advantage of fast IO. Similar to *Pushdown-only*, FPDB keeps the cold input data in the external cloud storage and pushes down filters to reduce network traffic. FPDB contains the following two main modules to enable such hybrid query

execution:

Hybrid query executor. The query executor takes in a logical query plan from the optimizer and transforms it into a *separable query plan* based on the content in the cache. The separable query plan processes the cached data in the compute node and pushes down filters and aggregation to the storage layer for uncached data. The two portions are then merged and fed to downstream operators. Section 4.3 will discuss the details of how operators are separated and merged.

Cache manager. The cache manager determines what data should be cached in the computation node. The cache eviction policy takes into account the existence of computation pushdown to exploit further performance improvement. For each query, the cache manager updates the metadata (e.g., access frequency) for the accessed data and determines whether admission and/or eviction should occur.

4.2.2 Design Choices

Designing FPDB requires making two high-level design decisions: *what to cache* and *at which granularity*, which we discuss below.

Caching Table Data or Query Results. Two types of data can potentially be cached in FPDB: *table data* and *query results*. Table data can be either the raw input files or a subset of rows/columns of the input tables. Query results can be either the final or intermediate results of a query, which can be considered as materialized views.

We consider the caching of table data and results as two orthogonal techniques, with their own opportunities and challenges. In FPDB, we explore caching only the table data since it is a simpler problem. The extension to result caching is left to future work.

Storage and Caching Granularity. FPDB stores tables in a distributed cloud storage service. Tables are horizontally partitioned based on certain attributes (e.g., primary key, sorted field, timestamp, etc.). Each *partition* is stored as a file in the cloud storage and contains all the columns for the corresponding subset of rows.

The basic caching unit in FPDB is a *segment*, which contains data for a particular column in a table partition (i.e., a column for a subset of rows). A segment is uniquely identified by the *segment key*, which contains three parts: the table name, the partition name, and the column name. The data format of a segment (e.g., Apache Arrow)

can be potentially different from the data format of the raw table file (e.g., CSV or Parquet).

4.3 Hybrid Query Executor

At a high level, the query executor converts the logical query plan into a *separable query plan* by dispatching *separable operators* into both local and pushdown processing; the results are then combined through a *merge operator*. This section describes these modules and then illustrates how the system works through an example query.

4.3.1 Separable Operators

We call an operator *separable* if it can be executed using segments in both the cache and the cloud storage, with the results combined as the final output. Not all the operators are separable (e.g., a join). Below we analyze the separability of several common operators.

Projection. Projection is a separable operator. If only a subset of segments in the queried columns are cached, the executor can load the remaining segments from the storage layer. The results can be then combined and fed to the downstream operator.

Filtering scan. Whether a filtering scan is separable depends on the cache contents. Ideally, the executor can process some partitions in the cache, push down filtering for the remaining partitions to the storage, and then merge the results, thus separating the execution. However, the situation can be complex when multiple columns are selected but not all of them are part of the filtering predicate.

Consider a scan query that returns two sets of attributes A and B from a table but the filtering predicate is applied on only attribute set A . For a particular table partition, if all segments in both A and B are cached, the partition can be processed using the data in the cache. However, if only a subset of segments in A are cached, the executor must either load the missing segments or push down the scan of the partition to the storage. Finally, if all segments in A but only a subset of segments in B are cached (call it B'), the processing can be partially separated — the executor filters A and B' using cached data, and pushes down the filtering for $(B - B')$.

Base table aggregation. Pushdown computation can perform aggregation on certain columns of a table. These operators can be naturally separated: a partition is aggre-

gated locally if all involved segments are cached; otherwise the aggregation is pushed down to the storage. The output can then be merged. For aggregation functions like average, both local and remote sides return the sum and count based on which the average is calculated.

Hash join. A join cannot be completely pushed down to the storage layer due to limitations of the computation model that a storage layer supports. Prior work [217] has shown that a bloom hash join can be partially pushed down as a regular predicate on the outer relation in a join. Given this observation, we conclude that the *building phase* in hash join is not separable — the columns of interest in the inner relation must be loaded to the compute node. The *probing phase* is separable: cached segments of the outer relation can be processed locally, while uncached segments can be filtered using the bloom filter generated based on the inner relation.

Sort. Theoretically, sort is separable — a remote segment can be sorted through pushdown and the segments are then merged in the computation node. Such techniques have been explored in the context of near-storage processing [123] using FPGA. However, since the cloud storage today does not support sorting (e.g., S3 Select), the separation of sorting is not supported in FPDB.

4.3.2 Separable Query Plan

A query plan is separable if it contains separable operators. Figure 4.4 shows examples of a conventional query plan without pushdown (Figure 4.4a) and a separable query plan (Figure 4.4b).

A conventional query plan reads all the data from the computation node’s local cache (i.e., buffer pool). For a miss, the data is loaded from the storage layer into the cache before query processing. A separable query plan, by contrast, splits its *separable operators* and processes them using both the cached data and pushdown computation. How the separation occurs depends on the current content in the cache, as described in Section 4.3.1.

For good performance and scalability, the merge operator in FPDB is implemented across multiple parallel threads. Specifically, each operator in FPDB is implemented using multiple worker threads and each worker thread is assigned multiple segments of data. The segments assigned to a particular worker might be entirely cached, entirely remote, or a mixture of both. For threads with a mixture of data sources,

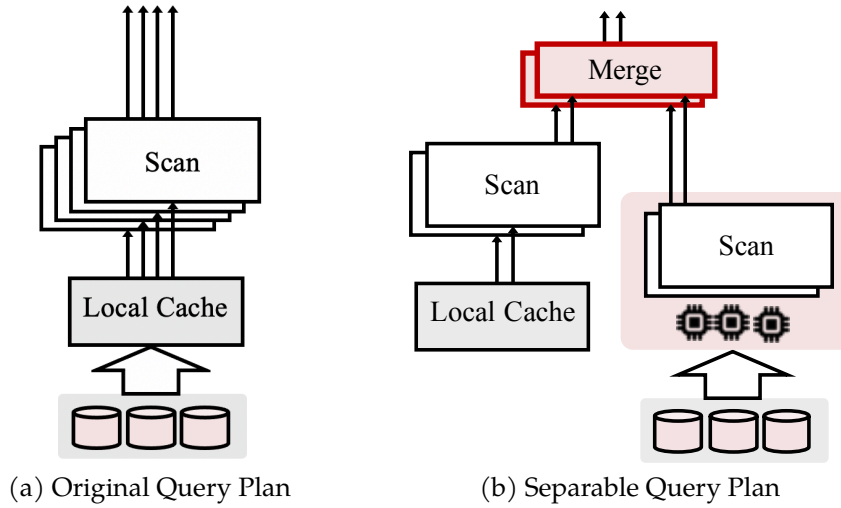


Figure 4.4: Example of a Separable Query Plan — The hybrid query plan contains a parallel *merge* operator that combines the results from cache and computation pushdown.

the results must be first merged locally into a unified data structure. The data across different threads does not need to be explicitly merged — they are directly forwarded to the downstream operators (e.g., joins) following the original parallel query plan.

4.3.3 Example Query Execution

```
SELECT R.B, sum(S.D)
FROM R, S
WHERE R.A = S.C AND R.B > 10 AND S.D > 20
GROUP BY R.B
```

Listing 4.1: Example query joining relations R and S.

We use the query above as an example to further demonstrate how the hybrid query executor works; the plan of the query is shown in Figure 4.5. The example database contains two relations R and S with the assumption that $|R| < |S|$, and each relation has two partitions (as shown in the cloud storage in Figure 4.5). Relation R has two attributes A and B, and relation S has two attributes C and D. Four segments are cached locally, as shown in the Local Cache module in Figure 4.5.

To execute the query using hash join, the DBMS first scans R to build the hash table and scans S to probe the hash table. The output of the join is fed to the group-by

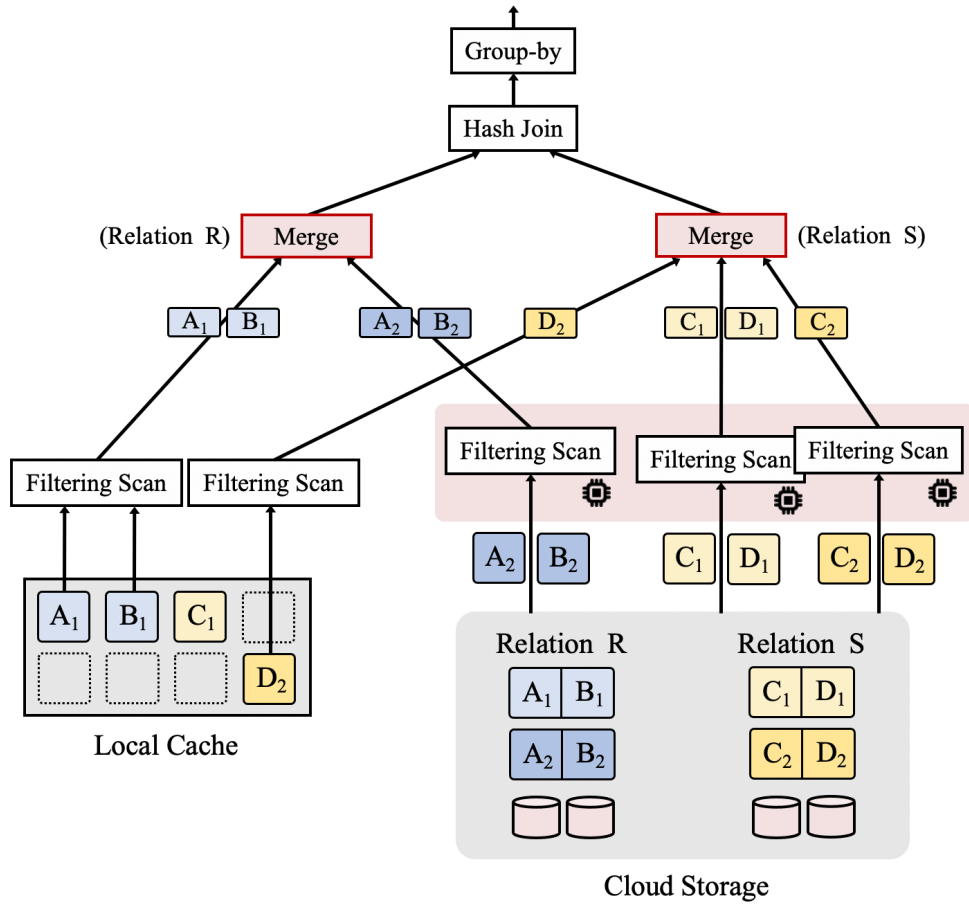


Figure 4.5: Separable Query Plan — For the query in Listing 4.1.

operator. Four partitions are involved in the join, i.e., partitions 1 and 2 in relations *R* and *S*, respectively. Depending on what segments are cached, the partition can be scanned locally, remotely, or in a hybrid mode.

Scanning relation *R*. For the first partition in *R*, both segments (i.e., *A*₁ and *B*₁) are cached. Therefore, the executor reads them from the local cache and no pushdown is involved. For the second partition in *R*, neither segment (i.e., *A*₂ and *B*₂) is cached, thus the filter is pushed down to the storage layer, which returns the subset of rows in *A*₂ and *B*₂ that satisfy the predicate. Finally, the local and remote results are combined through a merge operator.

Scanning relation *S*. For the first partition in *S*, only segment *C*₁ is cached, but the filter predicate of relation *S* is on attribute *D*, so the filtering scan cannot be processed locally and must be pushed down to the storage, which returns the filtered segments *C*₁ and *D*₁. For the second partition in *S*, only segment *D*₂ is cached. Since the filter

predicate is on D , the DBMS can directly read from the cache to process D_2 . Since the scan should also return attribute C_2 , the DBMS can push down the filter to the storage to load C_2 . Note that it is also possible to process this partition by pushing down the processing of both segments C_2 and D_2 , namely, ignoring the cached D_2 . This alternative design avoids evaluating the predicate twice (i.e., for the cached data and remote data) but incurs more network traffic. FPDB adopts the former option.

In the discussion so far, only the filtering scan is executed in a hybrid mode. As described in Section 4.3.1, the probe table in a hash join can also be partially pushed down. For the example query in particular, the DBMS can scan relation R first, builds a *bloom filter* on attribute $R.A$, and consider this bloom filter as an extra predicate when scanning relation S ; namely, the predicates on S then become $S.D > 20$ AND $\text{bloom_filter}(S.C)$. Note the bloom filter can only be constructed after the entire column of attribute $R.A$ is loaded. Therefore, when pushing down the probe table of the hash join, the scan of relation S can start only after the scan of relation R completes. In contrast, both scans can be executed in parallel when the bloom filter is not pushed down.

4.3.4 Execution Plan Selection

FPDB currently uses heuristics to generate separable query plans. It takes an initial plan from the query optimizer, and splits the execution of separable operators based on the current cache content. Specifically, an operator on a partition is always processed based on cached segments whenever the accessed data is cached. Otherwise, we try to pushdown the processing of the partition as much as we can. If neither works (e.g., the operator is not separable), we fall back to the pullup solution and load the missing segments from the storage layer. Note that the heuristics we adopt can generate only one separable plan given an input query plan. We adopt these heuristics based on the following two assumptions:

- Local processing on cached data is more efficient than pushdown processing in the storage layer.
- Pushdown processing is more efficient than fetching all the accessed segments and then processing locally.

The two conditions can hold in many cloud setups with storage disaggregation. The computation power within the storage layer is still limited compared to the local computation nodes and the network between the compute and storage layers has lower bandwidth than the aggregated disk IO bandwidth within the storage layer. Evaluation in Section 4.6 will demonstrate the effectiveness of the heuristics in improving system performance.

Although our heuristics are simple and effective, we do note that they may not always lead to a globally optimal separable query plan, because the two assumptions may not hold universally. For example, pushdown may outperform due to its massive parallelism even if all the data is cached. An important follow-up work is to develop a *pushdown-aware* query optimizer that can deliver better performance than the simple heuristics. We leave such exploration to future work.

4.4 Cache Manager

The cache manager decides what table segments should be fetched into the cache and what segments should be evicted, as well as when cache replacement should happen. We notice a key architectural difference in FPDB that makes conventional cache replacement policies sub-optimal: in a conventional cache-only system, cache misses require loading data from storage to cache. If cached segments are of equal size, each cache miss incurs the same cost. In FPDB, however, we can push down computation instead of loading data. Some segments may be more amenable to pushdown than others, which affects the benefit of caching. In other words, segments that cannot be accelerated through pushdown should be considered for caching with higher weight; and segments that can already be significantly accelerated through pushdown can be cached with lower weight — the extra benefit of caching beyond pushdown is relatively smaller. We develop a *Weighted-LFU* (WLFU) cache replacement policy for FPDB based on this observation.

In the rest of this section, we describe the integration of the cache manager and the hybrid executor in Section 4.4.1 and present a generic *benefit-based caching framework* in Section 4.4.2. We describe how conventional replacement policies (i.e., LRU, LFU, and Belady) fit into this framework in Section 4.4.3 and present the proposed Weighted-LFU policy in Section 4.4.4.

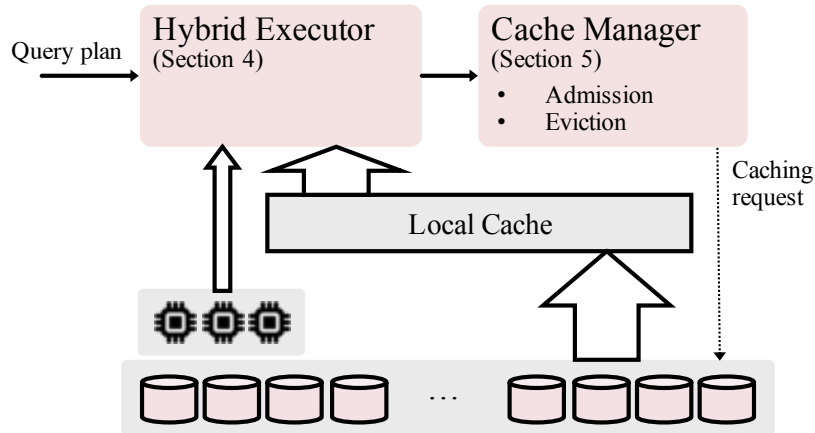


Figure 4.6: Integration of Executor and Cache Manager — The cache manager determines what segments should stay in cache.

4.4.1 Integration with Hybrid Executor

Figure 4.6 demonstrates how the cache manager is integrated with the hybrid executor in FPDB. The hybrid executor takes a query plan as input and sends information about the accessed segments to the cache manager. The cache manager updates its local data structures, determines which segments should be admitted or evicted, and loads segments from the cloud storage into the cache.

For cache hits, the hybrid executor processes the query using the cached segments. Cache misses include two cases: First, if the caching policy decides *not* to load the segment into the cache, then FPDB exploits computation pushdown to process the segment. Otherwise, if the caching policy decides to cache the segment, the DBMS can either wait for the cache load or push down the computation. The tradeoff here is between query latency and network traffic. FPDB adopts the former to minimize network traffic.

4.4.2 Benefit-Based Caching Framework

The cache manager determines which segments to admit or evict. The key design decisions here are the *admission* and *eviction* policies, which vary depending on the algorithm being implemented.

Ideally, a cache should contain the segments that can benefit the most from caching. We use the benefit value, $s.benefit$, to represent the benefit that segment s can get from staying in the cache. The key differences among caching policies are the different

ways of determining the benefit value. For example, the benefit value represents recency and frequency information for LRU and LFU policies, respectively. The details on determining the benefit value will be discussed in Sections 4.4.3 and 4.4.4.

Algorithm 3: Benefit-Based Caching Framework

Input: list of accessed segments S_{in}

```

1 foreach  $s$  in  $S_{in}$  do
2   | update the benefit of  $s$ 
3 while  $s \leftarrow$  get missing segment in  $S_{in}$  with highest benefit do
4   | if cache has enough space to hold  $s$  then
5   |   | issue request to load segment  $s$ 
6   |   |  $S_{in}.remove(s)$ 
7   | else
8   |   |  $t \leftarrow$  get segment in cache with lowest benefit
9   |   | if  $t.benefit < s.benefit$  then
10  |   |   | evict segment  $t$  from cache
11  |   | else
12  |   |   | break

```

Algorithm 3 shows how the benefit-based caching framework works in FPDB. When a set of segments are accessed, their benefit values are updated (lines 1–2) to reflect the new accesses. Then for each missing segment s in the order of decreasing benefit, we try to cache the segment if there is still space in the cache (lines 3–6). If the space is insufficient, we make space by evicting the segment with the least benefit (i.e., segment t) from the cache until t has a higher benefit value than the newly updated segments (lines 8–12), at which point the caching algorithm stops loading new segments.

Note that our framework makes a few simplifying assumptions regarding the caching policy. For example, we assume an eager admission policy where a segment is immediately admitted when it can be. The algorithm is also greedy since we consider segments for caching strictly in the decreasing order of benefit values. Alternative designs that break these assumptions should be orthogonal to the rest of the paper and beyond the scope of this work.

4.4.3 Traditional Replacement Policies

We explore three traditional caching policies: *least-recently-used* (LRU), *least-frequently-used* (LFU), and an optimal policy *Belady*.

Least-Recently Used (LRU). LRU is the most widely used cache replacement policy. A new cache miss always replaces the cached segment(s) that are least-recently used. LRU assumes that more recently used data are also likely to be used again in the future, thus having a higher priority to be cached. In our framework, this means the benefit value of a segment is simply the timestamp of its last access. Segment size does not play a role in our LRU algorithm.

Least-Frequently Used (LFU). In LFU, the *frequency counter* of a segment is incremented by $1/(segment.size)$ whenever it is accessed. The counter captures the size-normalized access frequency of the segment. If a missing segment's counter is larger than the smallest counter of cached segments, the least-frequently used segment in the cache is replaced. In our framework, the benefit value of segment s is simply the size-normalized frequency counter.

Belady Replacement Policy. Belady [73] is an optimal replacement policy that assumes availability of future information. Specifically, upon a miss, the algorithm replaces the segment that will not be accessed again for the longest time in the future. The benefit value of a segment s is therefore $1/(the\ number\ of\ queries\ until\ the\ segment\ is\ accessed\ again)$.

The Belady algorithm has been proven optimal when all the segments have the same size and all cache misses have the same cost. Since these assumptions are not always true in FPDB (due to variable segment size and pushdown), Belady is no longer optimal in our setting. Even so, we use it as a baseline to gain insights about different replacement policies.

4.4.4 Weighted-LFU Replacement Policy

As discussed at the beginning of this section, the hybrid caching and pushdown design in FPDB changes a fundamental assumption of cache replacement — cache misses for different segments incur different costs. Specifically, consider two segments, A and B , where A is accessed slightly more frequently than B , so that an LFU policy prefers caching A . However, it can be the case that segment A can benefit from computation pushdown so that a cache miss is not very expensive, while segment B

is always accessed with no predicate hence cannot benefit from pushdown. In this case, it might be more beneficial if the DBMS prefers B over A when considering caching.

In the benefit-based caching framework, we can incorporate the above intuition by taking pushdown into consideration when calculating the benefit values. Specifically, we modify the existing LFU policy to achieve this goal. Instead of incrementing the frequency counter by $1/segment.size$ for each access to a segment, we increment the counter by a *weight*, the value of which depends on whether the segment can be pushed down and if so, what cost pushdown is. Intuitively, the more costly the pushdown is, the more benefit we get from caching, hence the higher weight.

While there are many different ways to calculate the weight, in FPDB, we choose a straightforward formulation to represent a weight by the estimated total amount of work (measured in time) of pushdown computation, which is modeled by three components: time of network transfer, time of data scanning, and time of computation, as shown in Equation 4.1. The total time is divided by the segment size to indicate the size-normalized benefit of caching.

$$w(s) = \frac{total_work(s)}{size(s)} = \frac{t_{net}(s) + t_{scan}(s) + t_{compute}(s)}{size(s)} \quad (4.1)$$

We estimate the time of each component using the following simple equations (Equations 4.2–4.4).

$$t_{net}(s) = \frac{selectivity(s) \times size(s)}{BW_{net}} \quad (4.2)$$

$$t_{scan}(s) = \frac{N_{tuples}(s) \times size(tuple)}{BW_{scan}} \quad (4.3)$$

$$t_{compute}(s) = \frac{N_{tuples}(s) \times N_{predicates}}{BW_{compute}} \quad (4.4)$$

The equations above assume the data within the cloud storage is in row-oriented format (e.g., CSV) — they can be easily accommodated for column-format data (e.g., Parquet) by adjusting data scan amount to the size of columns accessed instead of the whole object. The time of each component is essentially the total amount of data transfer or computation divided by the corresponding processing bandwidth. Most of the parameters in the numerators can be statically determined from the

statistics (e.g., $size(s)$, $size(tuple)$, $N_{tuples}(s)$) or from the query (e.g., $N_{predicates}$), except for $selectivity(s)$ which can be derived after the corresponding segment has been processed by the executor.

For the bandwidth numbers in the denominators, we run simple synthetic queries that exercise the corresponding components to estimate their values. This process is performed only once before all the experiments are conducted.

4.5 Implementation

Given that not many open-source cloud-native DBMSs exist, we decided to implement a new prototype, FPDB, in C++ and make the code publicly available to the community. In this section, we describe different aspects of our system.

4.5.1 Cloud Environment

FPDB adopts a storage-disaggregation architecture. We choose the widely used AWS Simple Storage Service (S3) [22] as the cloud storage service. We also use S3 Select [120], a feature where limited computation can be pushed down onto S3, including projection, filtering scan, and base table aggregation. FPDB currently supports a single computation node.

FPDB requests data from S3 through AWS C++ SDK. We configure rate limits, timeouts, and connections in AWS *ClientConfiguration* high enough for better performance. Besides, FPDB does not use HTTPS/SSL which incurs extra overhead, as we expect analytics workloads would typically be run in a secure environment.

4.5.2 Data Format

FPDB supports table data in the cloud storage in both CSV and Parquet [30] format. Within the database engine and the cache, we use Apache Arrow [24] to manage data; table data is converted to Arrow within the scan operator. Arrow is a language-agnostic columnar data format designed for in-memory data processing. In the executor, we encapsulated Arrow’s central type, the *Array*, to represent a data segment. Arrays are one or more contiguous memory blocks holding elements of a particular type. The number of blocks required depends on several parameters such as the element type, whether null values are permitted, and so on. Using the same data

format for the processing engine and the cache eliminates the overhead of an extra data type conversion.

4.5.3 In-Memory Parallel Processing

Below we describe the details of FPDB’s in-memory parallel processing engine in a bottom-up fashion.

Expression Evaluation. FPDB uses Gandiva [151] for efficient expression evaluation. Gandiva compiles and evaluates expressions on data in Arrow format. It uses LLVM to perform vectorized processing and just-in-time compilation to accelerate expression evaluation and exploit CPU SIMD instructions for parallelism.

While Gandiva supports multi-threaded expression *execution* we found multi-threaded expression *compilation* troublesome. An attempt to compile multiple expressions simultaneously often failed in LLVM code generation or produced incomplete expressions. Therefore, we use serial compilation only.

Actor-Based Parallel Processing. FPDB supports parallel query execution using the C++ Actor Framework (CAF) [83]. CAF is a lightweight and efficient implementation of the Actor model [60] similar to those found in Erlang [66] or Akka [18]. Queries are composed of a number of operator types arranged in a tree. Scan operators form the leaves and a single collate operator forms the root. Operators communicate via message passing from producers to consumers, i.e., messages flow from leaves to the root.

Each operator type is instantiated into multiple actors. For the scan operator, an actor is created for each input table partition. For other operators, the number of actors is determined by the number of CPU cores. By default, CAF multiplexes the execution of these actors over all CPU cores on the host machine.

When query execution starts, a query coordinator instructs each operator to begin execution, in parallel, and then tracks each operator’s completion status. Operators perform their work, which may depend on the work of upstream operators, and send output messages to consumers, in a pipelined fashion. On completion of all operators the coordinator delivers the query result, from the collate operator, to the client executing the query.

Finally, both parallel hash join and parallel aggregation are implemented in FPDB. With actor-based parallel processing, FPDB can achieve a CPU utilization of more

than 96%.

Caching. We perform caching using the main memory (RAM) of the compute nodes. The cache in FPDB is also implemented as an actor with cache contents managed by the caching policy. The cache communicates with other operators through message passing to admit and evict data segments.

4.6 Evaluation

In this section, we evaluate the performance of FPDB by focusing on the following key questions:

- How does the hybrid architecture perform compared to baseline architectures?
- How does Weighted-LFU perform compared to traditional cache replacement policies?
- What is the impact of the network bandwidth?
- What is the resource usage and monetary cost of FPDB?

4.6.1 Experimental Setup

Server configuration. We conduct all the experiments on AWS EC2 compute-optimized instances. To thoroughly evaluate the performance, we pick two instance types with different network bandwidth, c5a.8xlarge (which costs \$1.52 per hour in US-West-1 pricing), with 32 vCPU, 64 GB memory, and a 10 Gbps network bandwidth, and c5n.9xlarge (which costs \$2.43 per hour in US-West-1 pricing), with 36 vCPU, 96 GB memory, and a 50 Gbps network bandwidth. Unless otherwise specified, we use the c5a.8xlarge instance as default. The server runs the Ubuntu 20.04 operating system.

Benchmark. We use the Star Schema Benchmark (SSB) [156] with a scale factor 100. Each table is partitioned into objects of roughly 150 MB when using CSV format, which is large enough to avoid generating many scan operators and the associated scheduling overhead. Each request to S3 is responsible for 15 MB data, which is small enough to leverage parallelism when loading data from the storage layer. Parquet data is derived from CSV data. The SSB dataset has five tables — four *dimension*

tables and one *fact table*. The fact table is much larger than the dimension tables, and contains foreign keys which refer to primary keys in the dimension tables.

We implement a *random query generator* based on SSB queries. A query is generated based on a *query template* with parameters in the filter predicates randomly picked from a specified range of values (or a set of values for categorical data). We incorporate *skewness* into the benchmark by picking the values following a *Zipfian* [110] distribution with tunable skewness that is controlled by a parameter θ . Skewness is applied to the fact table (i.e. *lineorder*) such that more recent records are accessed more frequently. A larger θ indicates higher skewness. The default value of θ is 2.0, where about 80% queries access the 20% most recent data (i.e. hot data). We set the default cache size to 8 GB which is enough to cache the hot data under the default skewness.

Measurement. Each experiment executes a batch of queries sequentially. The experiment contains a *warmup phase* and an *execution phase*. The warmup phase contains 50 queries sequentially executed; we have conducted separate experiments and found that 50 is sufficient to warm up the cache. The execution phase contains 50 sequentially executed queries on which we report performance.

Architectures for Comparison. The following system architectures are implemented in FPDB for performance comparison:

- *Pullup*: Data is never cached in the computation node. All accesses load table data from S3. This design is used in Hive [188], SparkSQL [64], and Presto [178, 184].
- *Caching-only*: Data is cached in the computation node with no pushdown supported. This design has been adopted in Snowflake [92] and Presto with Alluxio caching [13].
- *Pushdown-only*: Filtering scan is always pushed down to the storage layer whenever possible. Data is never cached in the computation node. This design is used in PushdownDB [217] and Presto with S3 Select enabled.
- *Hybrid*: The hybrid caching and pushdown architecture proposed in this work.

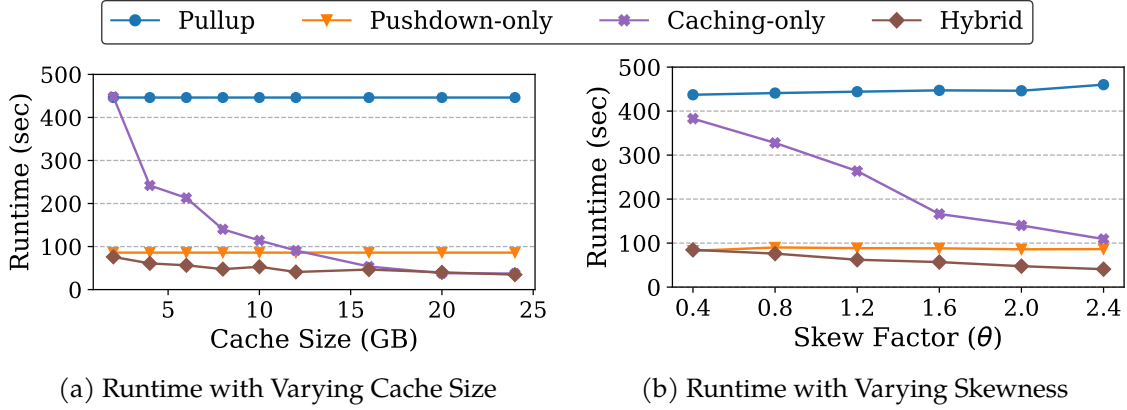


Figure 4.7: Performance Comparison (c5a.8xlarge) — The runtime with different (a) cache sizes and (b) access skewness.

4.6.2 Performance Evaluation

This subsection evaluates the performance of FPDB. We compare and analyze the query execution time of different caching and pushdown architectures. Then we evaluate the effectiveness of the Weighted-LFU caching policy. Finally, we study the impact of network bandwidth between the compute and storage layers.

4.6.2.1 Caching and Pushdown Architectures

We start with comparing the performance of the hybrid architecture with baseline architectures. We use LFU as the default cache replacement policy for *Caching-only* and *Hybrid*. We report performance varying two parameters: the cache size and the access skewness in the workload (i.e., θ), on both c5a.8xlarge and c5n.9xlarge instances.

Overall Performance. Figure 4.7 shows the runtime comparison on the c5a.8xlarge instance. Figure 4.7a compares the performance between different caching/pushdown architectures when the cache size changes. First, we observe that the performance of *Pullup* and *Pushdown-only* are not affected by the cache size; this is because data is never cached in either architecture. *Pushdown-only* outperforms *Pullup* by $5.2\times$, because computation pushdown can significantly reduce the network traffic. The *Caching-only* architecture, in contrast, can take advantage of a bigger cache for higher performance. When the cache is small, its performance is close to *Pullup*. When the cache size is bigger than 12 GB, it outperforms *Pushdown-only* due to the sufficiently high cache hit ratio.

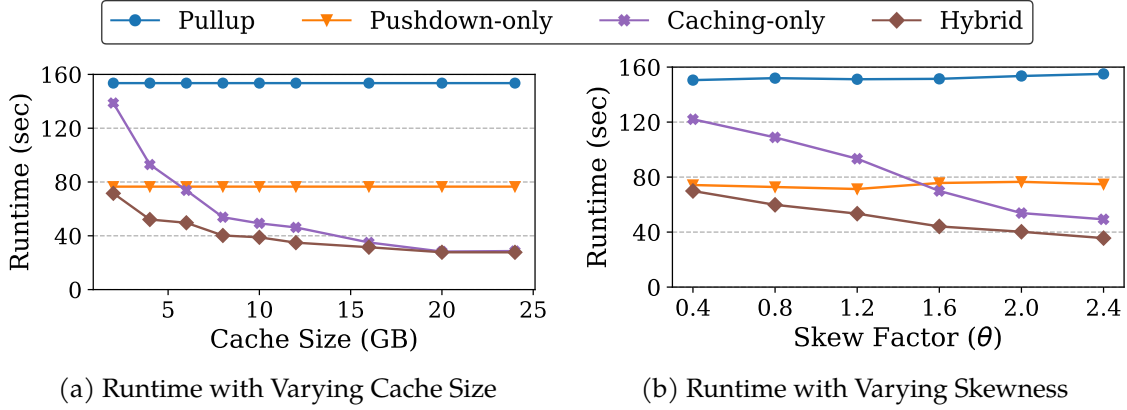


Figure 4.8: Performance Comparison (c5n.9xlarge) — The runtime with different (a) cache sizes and (b) access skewness.

Finally, the performance of *Hybrid* is consistently better than all other baselines. When the cache is small, FPDB behaves like *Pushdown-only*; when the cache is large enough to hold the working set, FPDB behaves like *Caching-only*. For cache sizes in between, *Hybrid* can exploit both caching and pushdown to achieve the best of both worlds. At the default cache size (i.e., 8 GB), *Hybrid* outperforms *Pushdown-only* by 80%, *Caching-only* by 3.0 $\times$, and *Pullup* by 9.4 $\times$. At the crossing point of *Pushdown-only* and *Caching-only* (i.e., roughly 12 GB), *Hybrid* outperforms both by 2.2 $\times$.

Figure 4.7b shows the performance of different architectures as the access skewness increases. *Pullup* and *Pushdown-only* are not sensitive to changing skewness. Both *Caching-only* and *Hybrid* see improving performance for higher skew, due to a higher cache hit ratio since there is less hot data.

We further evaluate FPDB on the c5n.9xlarge instance which has a 50 Gbps network. Figure 4.8a compares the performance of different architectures with different cache sizes. The general trends are similar to c5a.8xlarge, and *Hybrid* consistently outperforms all baselines. With a higher network bandwidth, *Pushdown-only* is 2 $\times$ faster than *Pullup*, which is lower than the speedup on the c5a.8xlarge instance because loading segments missing from the cache is faster. The crossing point of *Pushdown-only* and *Caching-only* shifts towards the left to roughly 6 GB, at which point *Hybrid* outperforms both baselines by 51%.

The performance results with increasing access skew on the c5n.9xlarge instance are shown in Figure 4.8b. The general trend is also similar to c5a.8xlarge.

From the results above, we observe that different hardware configurations can shift the relative performance of pushdown and caching, but the hybrid design always

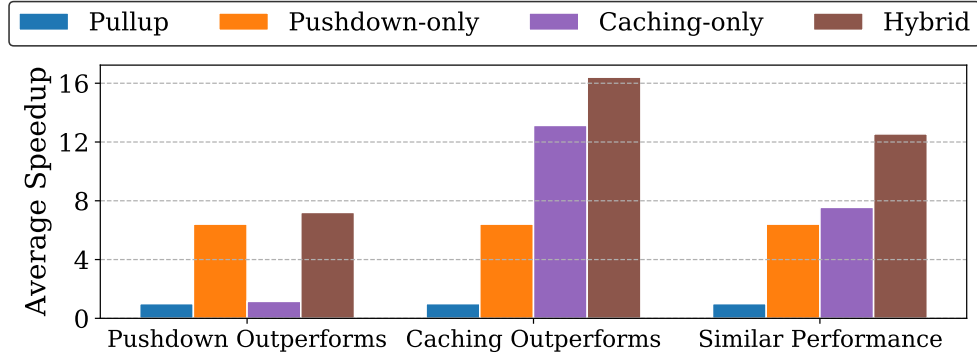


Figure 4.9: Per-Query Speedup — The average speedup of each representative case in different architectures.

outperforms both baselines.

Per-Query Analysis. We dive deeper into the behavior of the system by inspecting the behavior of individual queries, and observe that they can be categorized into three representative cases: (1) caching has better performance, (2) pushdown has better performance, and (3) both have similar performance. For each category, we compute the average speedup of different architectures compared to *Pullup*. The results on c5a.8xlarge are shown in Figure 4.9. Although not shown here, the results on c5n.9xlarge have a similar trend.

In all three cases, the performance of *Hybrid* can match the best of the three baseline architectures. When pushdown (or caching) achieves a higher speedup, *Hybrid* slightly outperforms *Pushdown-only* (or *Caching-only*). When the two techniques have similar performance, *Hybrid* can outperform either baseline significantly. The performance results in Figure 4.7 and Figure 4.8 are an aggregated effect of these three categories of queries.

Comparison against Existing Solutions.

To further validate the performance of our system, we compare FPDB with Presto, a production cloud database. We use Presto v0.240 which supports computation pushdown through S3 Select and caching through Alluxio cache service [13]. For Alluxio, we cache data in main memory, which is consistent with FPDB. We conduct the experiment under the default workload on an m5a.8xlarge instance² on CSV data.

The result is shown in Table 4.1. FPDB outperforms Presto by 25% in *Pullup* and

²We ran Presto on AWS Elastic MapReduce (EMR), which at the time of this work did not support c5a and c5n instances in US-West-1. m5a has the most similar configuration.

Table 4.1: Performance Comparision between Presto and FPDB — The runtime (in seconds) of different architectures in both systems.

Architecture	Pullup	Pushdown-only	Caching-only	Hybrid
Presto	588.7	271.3	536.3	-
FPDB	472.1	111.2	225.7	80.8

$2.4\times$ in *Pushdown-only*, which implies that query processing inside FPDB is efficient. In *Caching-only* FPDB is $2.4\times$ faster than Presto with Alluxio caching. A few reasons explain this performance gain: First, Alluxio caches data at block granularity, which is more coarse-grained than FPDB. Second, Alluxio manages cached data through its file system, incurring higher overhead than FPDB, which manages cached data directly using heap memory. We further note that only FPDB supports *Hybrid* query execution. The Alluxio caching layer is unaware of the pushdown capability of S3 while loading data, thus only one technique can be used.

4.6.2.2 Parquet Performance

In this experiment, we investigate the performance of FPDB on the Parquet data. The challenge, however, is that current S3 Select has poor performance on Parquet (potentially due to inefficient Parquet readers), leading to even worse performance than *Pullup*. We studied a few other cloud-storage systems but they either have the same problem [43, 34] or do not support Parquet pushdown [2].

Therefore, we evaluate the performance on the Parquet data using our storage prototype — FPDB-Store (details are discussed later in Section 5.4.2), which uses Arrow’s Parquet reader to efficiently decompress the Parquet data into Arrow’s in-memory columnar format. The pushdown results are sent to the compute layer via Arrow Flight [31] without any format conversions — uncompressed data is transferred (S3 Select also transfers uncompressed CSV data).

We perform the experiments on both c5a.8xlarge and c5n.9xlarge instances and report results in Figure 4.10. We add the performance on the CSV data for reference. We observe that the performance on Parquet is always better than the performance on CSV, since projection pushdown is free in the Parquet format, leading to network traffic and disk I/O reduction. For *Pullup* and *Caching-only*, by comparing Figure 4.10a and 4.10b, we observe that the gain of Parquet is more prominent when the network bandwidth is low, in which case a more severe bottleneck is being addressed. The

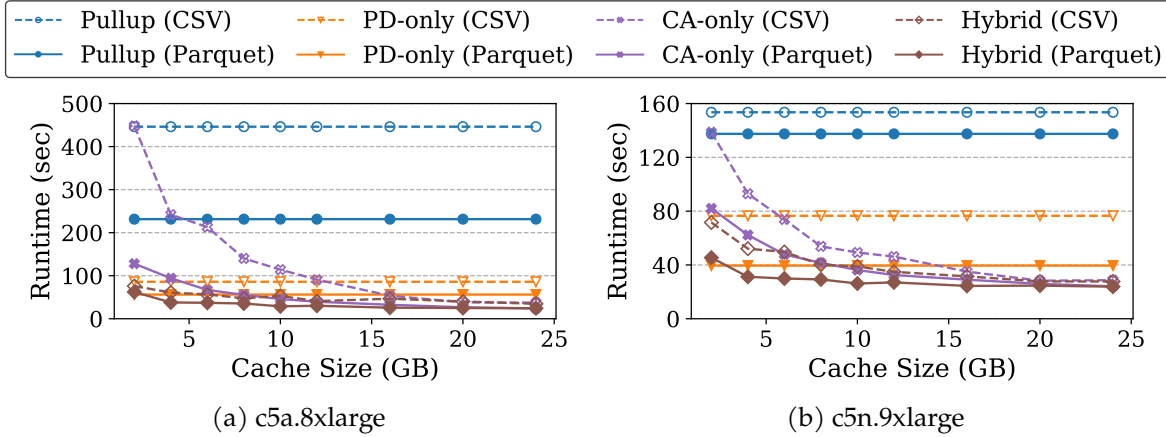


Figure 4.10: Performance of different architectures with different cache sizes on the Parquet data (*Pushdown-only* as *PD-only*, *Caching-only* as *CA-only*). Results on the CSV data (Figure 4.7a, 4.8a) are added for reference.

performance improvement of *Pushdown-only* on Parquet is similar on both instances, since pushdown results are relatively small, making the network not a major bottleneck anymore — joins are dominating the performance instead. When using the Parquet format, *Hybrid* still achieves the best performance among all the architectures. At the crossing point of *Pushdown-only* and *Caching-only*, *Hybrid* outperforms both by 59% and 30%, respectively.

We provide the following intuition as to why *Hybrid*'s advantage remains in Parquet data. In essence, the performance gain of pushdown mainly comes from three aspects: (1) network traffic reduction from *projection pushdown*; (2) network traffic reduction from *selection pushdown*; (3) parsing and filtering data with massive parallelism. With Parquet, all architectures have the benefit of (1), but only pushdown processing has the advantages of (2) and (3). For example, under the default cache size (i.e. 8 GB), *Hybrid* reduces network traffic by 50% over *Caching-only* on the Parquet data (as opposed to 93% on CSV).

4.6.2.3 Weighted-LFU Caching Policy

In this experiment we study the cache replacement policy proposed in this chapter, Weighted-LFU (cf. Section 4.4.4). Figure 4.11 compares it with conventional cache replacement policies. We have conducted a separate experiment which shows that LRU has worse performance than LFU and Belady, and thus exclude it from this experiment.

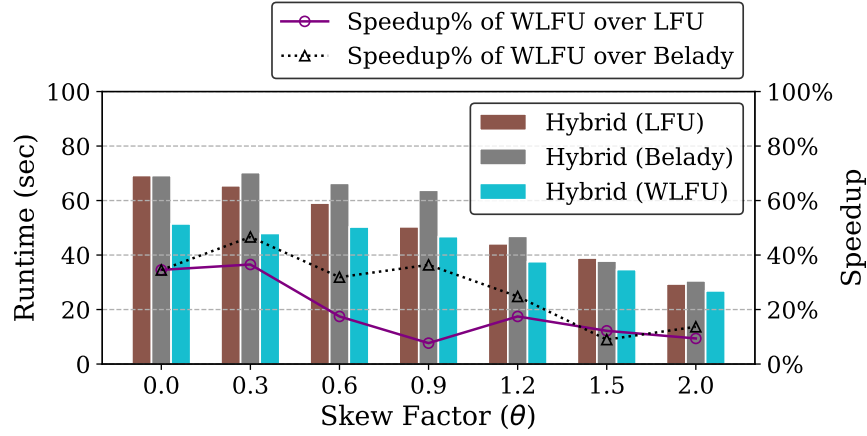
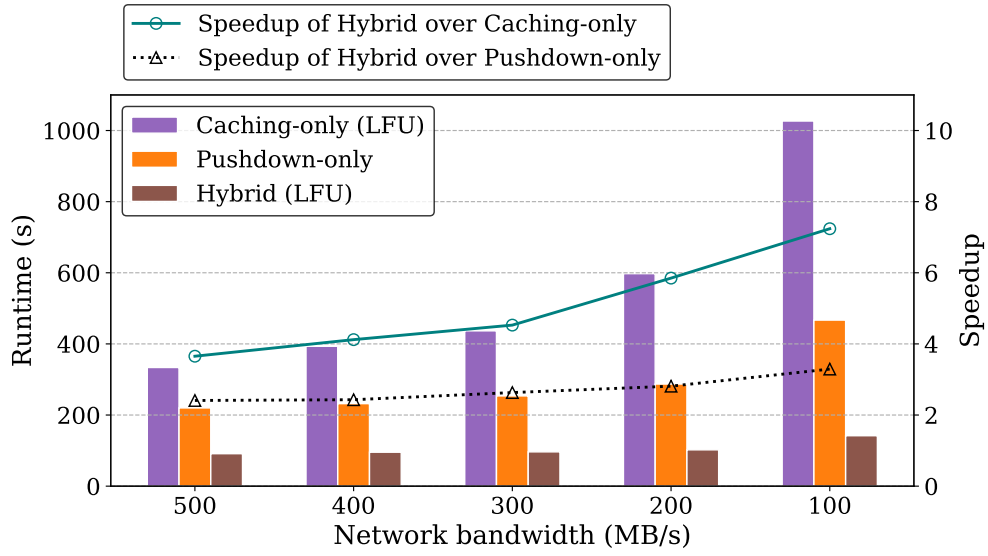


Figure 4.11: Weighted-LFU Cache Replacement Policy — Runtime comparison of Weighted-LFU (WLFU) and baseline replacement policies (i.e., LFU and Belady) with varying access skewness.

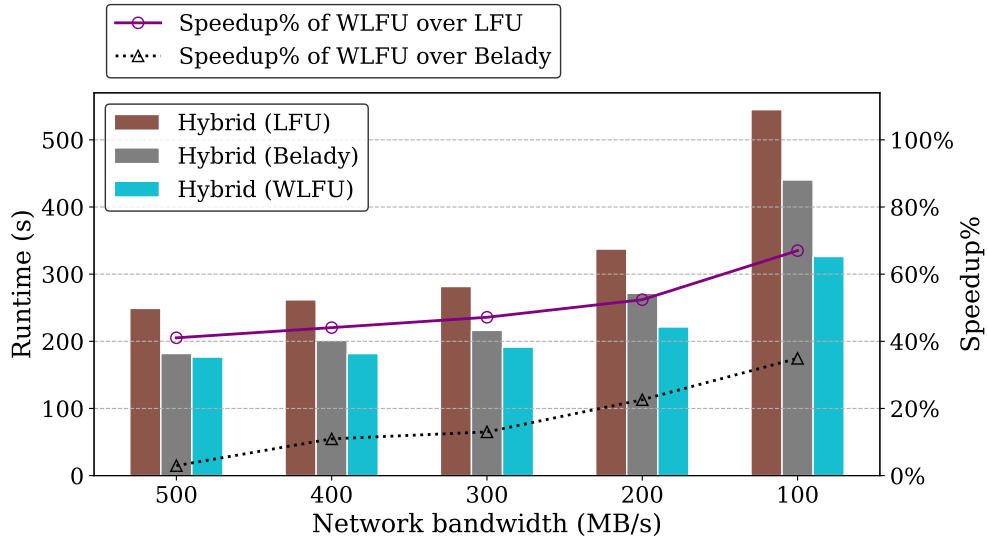
In the default SSB queries, predicates on different attributes have similar selectivity, making the pushdown cost of different segments similar. To measure the effectiveness of Weighted-LFU, we change the SSB queries to incorporate different pushdown costs, by varying the selectivity of filter predicates. Specifically, we change predicates on some attributes to equality predicates which have very high selectivity (e.g. `lo_quantity = 10`) while keeping others range predicates (e.g. `lo_discount < 3` or `lo_discount > 6`).

As Figure 4.11 shows, WLFU consistently outperforms the baseline LFU and Belady. The biggest speedup happens when $\theta = 0.3$ (i.e., low access skewness), where WLFU outperforms LFU and Belady by 37% and 47%, respectively. We further measure network traffic incurred and find WLFU reduces network traffic by 66% and 78%, compared to the baseline LFU and Belady respectively. Recall that the optimization goal of WLFU is to reduce network traffic; this demonstrates that the algorithm achieves the goal as expected. Interestingly, Belady underperforms the baseline LFU and incurs more network traffic, because Belady keeps prefetching entire segments for future queries, which takes little advantage of computation pushdown.

As θ increases, the performance benefit of WLFU decreases. When θ is small, there is little access skewness, so segments with higher pushdown cost are cached, leading to the higher effectiveness of WLFU. When θ is large, the access skewness overwhelms the difference of pushdown cost among segments. In this case, almost all the hot segments are cached, leading to reduced performance gains for WLFU.



(a) Architectures



(b) Caching Policies

Figure 4.12: Network Bandwidth — The runtime of different (a) architectures and (b) cache replacement policies as the network bandwidth varies.

4.6.2.4 Impact of the Network Bandwidth

A major bottleneck in the storage-disaggregation architecture is the network bandwidth between the computation and storage layers, thus in this experiment we study its impact on the performance. We evaluate on the c5n.9xlarge instance and vary the network bandwidth by artificially setting *readRateLimiter* of the S3 client.

The Hybrid Architecture. We first study the effect of the network bandwidth on different architectures. *Pullup* always has the worst performance so we do not include it in this experiment. We use the same workload as Section 4.6.2.1 with the default cache size and skewness, and report the results in Figure 4.12a. *Hybrid* suffers the least from the limitation of the network bandwidth, since it incurs the least network traffic. With throttled network, the more network traffic involved, the higher latency during network transfer, so the performance benefit of *Hybrid* keeps increasing. With 2 Gbps network bandwidth, *Hybrid* achieves $8.7\times$ speedup over *Caching-only*, and $2.9\times$ speedup over *Pushdown-only*.

Weighted-LFU Caching Policy. Next we investigate the impact of the network bandwidth on different caching policies. We use the same workload as Section 4.6.2.3 with the default cache size and small θ (i.e., $\theta = 0$). As described in Section 4.6.2.3, with a large θ , segments with a higher access frequency are cached, so both LFU and WLFU cache hot data, leading to similar performance; i.e. WLFU outperforms LFU more with a small θ . In this experiment we set θ to 0. We compare Weighted-LFU, LFU, and Belady caching policies in the *Hybrid* architecture, and report the result in Figure 4.12b. Among different caching policies, WLFU suffers the least from the limitation of the network bandwidth. With the network gradually being throttled, the performance benefit of WLFU over LFU becomes more significant, due to the amplification of the benefit from the reduction of network traffic. When the network bandwidth is limited to 2 Gbps, WLFU outperforms baseline LFU and Belady by $2.3\times$ and $2.9\times$, respectively.

4.6.3 Resource Usage and Cost

In this subsection, we investigate the resource usage and cost of different architectures. We note that a specific pricing model (e.g., AWS S3 Select) may involve various non-technical factors, and thus may not be the best metric in measuring the efficiency of a system. Therefore, we build a *resource usage model* purely based on the usage of CPU and network resources, then we explain how the resource usage model is related to various pricing models.

We consider two kinds of resources: CPU and network. Table 4.2 compares the network usage of different architectures; these numbers are directly measured at the compute node. *Hybrid* reduces the network traffic by 79% over *Pushdown-only* due to

Table 4.2: Network Usage (GB) of different architectures.

Architecture	Pullup	Pushdown-only	Caching-only	Hybrid
Usage	460.9	37.1	112.6	7.9

Table 4.3: CPU Usage (with dedicated compute servers) — CPU time (in minutes) of different architectures (normalized to the time of 1 vCPU).

Architecture	Pullup	Pushdown-only	Caching-only	Hybrid
Compute	249.6	48.5	70.3	23.2
Storage	0.0	31.1	0.0	7.4
Total	249.6	79.6	70.3	30.6

Table 4.4: CPU Usage (with multi-tenant compute servers) — CPU time (in minutes) of different architectures (normalized to the time of 1 vCPU).

Architecture	Pullup	Pushdown-only	Caching-only	Hybrid
Compute	41.8	15.9	15.7	11.5
Storage	0.0	31.1	0.0	7.4
Total	41.8	47.0	15.7	18.9

caching and data reuse, and 93% over *Caching-only* because *Hybrid* can push down filtering to the storage layer.

In a cloud database, the compute servers can be either dedicated to a user or shared across multiple tenants (e.g., a serverless model). We estimate the CPU usage in both cases. For the compute server, we monitor the CPU usage using *vmstat*. Since we cannot directly measure the CPU usage for S3 storage servers, we estimate it by running the same pushdown task in the compute node and measuring the CPU usage. With a dedicated server, users pay for the entire server for the duration of query execution; idle but reserved CPUs are considered “used” during query execution. Table 4.3 shows the CPU usage. *Pullup* and *Caching-only* have no CPU usage of storage since there is no pushdown. *Hybrid* incurs the least CPU usage due to its low runtime.

In the multi-tenant case, users pay only for the CPU-time actually consumed by the query. The CPU usage is shown in Table 4.4. *Hybrid* reduces the CPU usage by 60% compared to *Pushdown-only*. However, *Hybrid* increases the total CPU usage by 20% over *Caching-only* because of the redundant work performed at both the compute and storage sides during pushdown, including evaluating the same filter predicate (cf.

Section 4.3.3) and data parsing. The extra CPU resource consumption of pushdown is relatively small.

As a conclusion, *Hybrid* achieves higher performance using comparable CPU resource and much less network resource.

How the resource usage model translates to monetary cost depends on the use case of the system. We identify the following three common use cases:

1. Users deploy the cloud storage service on their own infrastructures. Examples include Ceph [34] and MinIO [43]. Private clouds fall into this category.
2. Users adopt public cloud storage like S3, computation occurs in **a different data center** from the storage.
3. Users adopt public cloud storage like S3, computation occurs in **the same data center** as the storage.

In case 1, users are already paying for the storage infrastructure. Pushdown is simply exploiting otherwise underutilized resources within the storage layer and does not introduce extra cost.

In case 2, most cloud service providers charge significantly more when the data is transferred outside of a region. For example, AWS charges \$0.09/GB for data transferred outside the cloud and \$0.02/GB for data transferred across data centers. In this case, the traffic reduction of *Hybrid* leads to significant cost savings.

In case 3, the cost depends on the pricing model of computation pushdown. We observe that S3 Select charges significantly more for storage-layer computation than regular EC2 computation. As a result, *Hybrid* is more expensive than *Caching-only* in S3 Select today. We believe future systems will offer a more fair pricing model, making pushdown computation more attractive in terms of cost.

4.7 Related Work

In this section, we further discuss related work that has not been discussed earlier in this chapter.

Cloud Databases. Modern cloud databases adopt an architecture with storage disaggregation. This includes conventional data warehousing systems adapting to the cloud (e.g., Vertica [136] Eon [194] mode) as well as databases natively developed

for the cloud (e.g., Snowflake [92, 199], Redshift [112], Redshift Spectrum [21], Athena [19]).

Besides data warehousing systems, transactional databases also benefit from storage-disaggregation. AWS Aurora [197, 198] is an OLTP database deployed on a custom-designed cloud storage layer where functionalities including log replay and garbage collection are offloaded to the storage layer. The disaggregation of computation and storage also allows each component to easily adapt the workload requirements dynamically.

Computation Pushdown. The technique of computation pushdown has been explored in multiple research areas, both within and beyond database systems, including in-cloud databases, database machines, Smart Disks/SSD, and processing-in-memory (PIM).

Many in-cloud databases push certain computation tasks near the data source, such as AWS Redshift Spectrum [21], S3 Select [120], and PushdownDB [217]; these systems use software techniques to implement pushdown functionalities. Moreover, AWS Advanced Query Accelerator (AQUA) [70] announced to use special hardware accelerators (i.e., AWS Nitro chips [33]) to implement pushdown functions with faster speed and lower energy consumption.

Database machines have emerged since the 1970s. Many of them push computation to storage via special hardware. The Intelligent Database Machines (IDM) [190] pushes most DBMS functionalities to the backend machine which sits closer to disks. In Grace [104], multiple filter processors are connected to disks to perform selection and projection. IBM Netezza data warehouse appliances [101] enable pushdown of selection, projection, and compression to disks through FPGA-enabled near-storage processors. Finally, Exadata Cells in the storage layer of Oracle Exadata Database Machine [203] support operations including selection, projection, bloom join, and indexing. To the best of our knowledge, none of these systems support fine-grained hybrid caching and pushdown like FPDB does.

Smart Disks/SSD is another line of research exploiting computation pushdown. Projects including Active Disks [170] and IDISKS [128] have investigated the idea of pushing computation to magnetic storage devices. Summarizer [134] and Biscuit [111] are near-data processing architectures supporting selection pushdown to the SSD processor. Both filtering and aggregation are pushed to smart SSDs [99] and near-storage FPGAs [204, 105] to accelerate relational analytical processing. AQUO-

MAN [207] supports pushing down most SQL operators including multi-way joins to SSDs. Again, none of these systems support fine-grained hybrid caching/pushdown like FPDB does.

Moreover, some recent work explored *processing-in-memory* (PIM) to push computation into DRAM or NVM. Modern 3D-stacked DRAM implemented a logic layer underneath DRAM cell arrays within the same chip [91], avoiding unnecessary data movement between memory and the CPU [108]. Kepe et al. [129] presented an experimental study focusing on selection in PIM.

Client-server Architectures. Many distributed databases adopt a client-server architecture. These systems can move the data to the client where the query is initiated (i.e., caching); alternatively, the query can be moved to the servers where the data resides (i.e., pushdown). Plenty of research focused on query optimization by combining both. Franklin et al. [102] introduced Hybrid Shipping which executes some query operators at the client-side, where the query is invoked, and some at the server-side, where data is stored. Garlic [172] extended this approach by pushing query operators to heterogeneous data stores. MOCHA [171] supports pushing both query operators and their code to remote sites where these operators are not even implemented. The key difference between FPDB and these systems is that FPDB is built for a storage-disaggregation architecture in a cloud database, where the computation power inside the cloud storage is more limited than the servers in a client-server architecture.

4.8 Conclusion

In this chapter, we presented FlexPushdownDB, a cloud-native OLAP database that combines the benefits of caching and computation pushdown in a storage-disaggregation architecture through fine-grained hybrid execution. We explored the design space in a hybrid query executor and designed a novel *Weighted-LFU* cache replacement policy specifically optimized for the disaggregation architecture. Evaluation on the Star Schema Benchmark demonstrated that the hybrid execution can outperform both *Caching-only* and *Pushdown-only* architectures by $2.2\times$.

Chapter 5

Enhance Pushdown Computation for Cloud-Native Databases

As discussed in Chapter 4, modern OLAP databases are aggressively moving to the cloud for high elasticity and low cost. These systems adopt a storage-disaggregation architecture which manages computation and storage into separate layers. The network between the two layers has become a major performance bottleneck due to its relatively lower bandwidth and higher latency [186]. *Computation pushdown* is a promising solution to mitigate the network bottleneck, where some computation logic is sent and evaluated close to the storage, resulting in less data returned to the computation layer [217, 208, 120, 21, 178].

In existing cloud OLAP DBMSs, the query execution engine decides whether to push down computation during the query optimization phase. For example, Presto [178] enables pushdown for all filter operators to S3 by setting a flag in the configuration file (`"hive.s3select-pushdown.enabled=true"`). However, pushdown in existing systems does not consider the current storage layer's computational capacity and load at the time when the query is executed. If the storage-layer computation resource is scarce (e.g., due to multi-tenancy), computation pushdown may hurt performance for a particular query. Unfortunately, it is difficult and sometimes impossible to predict the storage-side computational load ahead of query execution. Another limitation of existing pushdown systems is the *lack of a general principle that determines which operators are amenable for pushdown*. Existing systems empirically consider only simple operators such as selection, projection, and aggregation due to their simplicity and effective traffic reduction. We believe a larger set of operators can benefit from pushdown and a principle should exist to decide which operators

should be considered.

In this work, we address both limitations above. To address the first limitation, we propose an adaptive query processing approach that adapts the query plan while the query is executed to consider the current load on storage nodes. We first explore the design space and analyze the theoretical bound — what is the optimal division of the computation tasks between pushdown and non-pushdown to achieve the best overall performance. Then we design *Adaptive pushdown* as a new mechanism to avoid throttling the storage-layer computation during pushdown. Instead of having the database engine making the pushdown decisions, *Adaptive pushdown* lets the storage layer to decide whether to execute an incoming pushdown request, or to push the request back to the computation layer. When a pushback happens, the computation layer reads the raw data from the storage layer and processes the task locally. Intuitively, pushdown requests should be executed at the storage layer when sufficient computation resources exist, and pushed back when the storage layer is saturated. We will demonstrate that the proposed mechanism can perform close to the theoretical bound. Evaluation results show that *Adaptive pushdown* outperforms both *No pushdown* and *Eager pushdown* baselines by up to $1.9\times$ on the TPC-H [54] benchmark.

To address the second limitation, we derive a general principle to identify pushdown-amenable computational tasks, through summarizing common patterns of pushdown capabilities in existing systems. First, pushdown tasks should be *local* — the pushdown computation should access data only within a single storage node and not incur data transfer within the storage layer. Second, pushdown tasks should be *bounded* — a pushdown task should require at most linear CPU and memory resources with respect to the accessed data size. This principle preserves the benefits of storage-disaggregation, and simplifies resource isolation and security in a multi-tenant environment. Following the principle above, we further identify two operators that can benefit from pushdown to the storage layer — *selection bitmap* and *distributed data shuffle*. Evaluation results show that the two new pushdown operators can further accelerate end-to-end query processing on TPC-H by up to $3.0\times$ and $1.7\times$ respectively.

In summary, this work makes the following key contributions:

- We develop *Adaptive pushdown* to leverage storage-layer computation dynamically, which utilizes a pushback mechanism to determine whether a pushdown operator should be executed in the storage.

- We infer a general principle to determine whether a database operator is amenable to pushdown from existing cloud OLAP DBMSs, and identify two more common operators that can benefit from being offloaded to the storage layer — *distributed data shuffle* and *selection bitmap*.
- We implement and integrate *Adaptive pushdown* and new pushdown operators into FlexPushdownDB [208] (*FPDB* in short), an open-source C++-based cloud-native OLAP DBMS¹.

The rest of this chapter is organized as follows. Section 5.1 introduces the background of pushdown in existing cloud OLAP DBMSs and illustrates the motivations. Section 5.2 demonstrates adaptive pushdown and analyzes its theoretical optimum. In Section 5.3, we present a general principle of pushdown that is summarized from the implementation of existing cloud OLAP DBMSs, and identify new pushdown operators. Section 5.4 reveals additional implementation details. Section 5.5 evaluates the performance of both adaptive pushdown and new pushdown operators using FPDB. Finally, Section 5.6 discusses the related work and Section 5.7 concludes this work. This chapter is adapted from previously published work [210].

5.1 Background and Motivation

As described in Section 4.1, the concept of computation pushdown [120, 197, 198, 21, 70, 2, 178, 217, 208, 48] has been widely adopted by modern cloud-native databases to mitigate the network bottleneck between the compute and storage layers in disaggregated architectures. However, there are still restrictions and unexplored opportunities. In the rest of this section, we demonstrate two major limitations to motivate the work in this chapter.

Limitation 1: Static Pushdown Decisions Made at Planning Time. Existing cloud OLAP DBMSs make pushdown decisions during the query optimization phase — the query plan is split into two pieces, with the pushdown portion executed in the storage layer, and the rest executed in the computation layer. Once a pushdown decision is made, it cannot be changed at runtime — a pushdown task will always be executed at storage. In other words, pushdown decisions are made *eagerly*, where the entire pushable subquery plan is offloaded to the storage.

¹<https://github.com/cloud-olap/FlexPushdownDB/tree/vldb24>

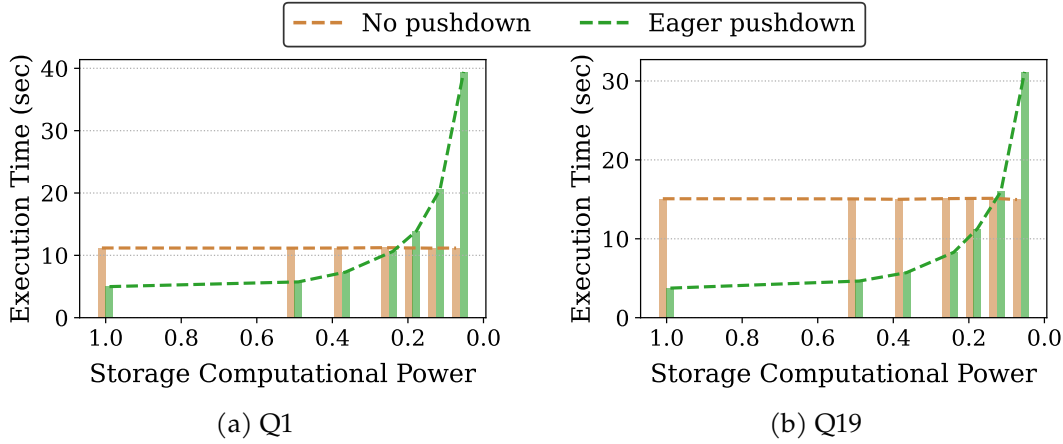


Figure 5.1: Performance of *No pushdown* and *Eager pushdown* on Sample Queries — Q1 and Q19 in TPC-H.

Eagerly pushing all computation tasks that can be offloaded may not always benefit query processing, since the storage layer is in a shared multi-tenant environment, where the amount of computation resource for each request may vary. To build a deeper understanding, we prototype a S3-like object storage layer within FPDB, and measure the performance of pushdown using standard benchmark queries (TPC-H [54], see Section 5.5.1 for detailed setups) in different storage-layer resource utilization conditions. Figure 5.1 presents the results of two sample queries (Q1 and Q19). *No pushdown* is included as a baseline for comparison.

For both queries, the performance of *No pushdown* is independent of the available storage-layer computational power (we regard full storage-layer computational power as all CPU cores at the storage node are available for pushdown tasks, see Section 5.5.2). *Eager pushdown* outperforms *No pushdown* when the storage has abundant computational resource, since it offloads computation tasks to the storage layer which reduces data transfer. However, when storage-layer computational resource is insufficient, query execution starts to suffer from the slowdown of pushdown execution. When the storage layer is saturated, *Eager pushdown* underperforms *No pushdown*, and pushdown execution becomes the major performance bottleneck in the end-to-end query processing.

An ideal solution should consider the storage-layer computational resource utilization status, and adjust pushdown decisions at runtime *adaptively*. Intuitively, when the storage system is idle or under light load, more computation tasks should be placed and executed at the storage layer to speed up query processing, which

behaves close to *Eager pushdown*. Conversely, when the storage system is under heavy load, the DBMS should be more inclined to execute operators at the local compute nodes instead of offloading them to the storage, to avoid throttling storage-layer computation, which leads to a behavior similar to *No pushdown*. We will present the proposed adaptive pushdown mechanism in Section 5.2.

Limitation 2: Lack of Systematic Analysis on Pushdown Operators. Existing systems design and implement pushdown features empirically, which end up picking a customized set of pushdown operators respectively. For example, simple pushdown functionalities like selection, projection, and scalar aggregation are supported by almost all existing pushdown systems, grouped aggregation is favored by Redshift Spectrum [21], and pushdown of bloom filters is introduced in PushdownDB [217].

We aim to conduct a comprehensive analysis of the design space, to identify key characteristics that contribute to the suitability of a pushdown operator. We closely examine the behaviors of existing OLAP DBMSs that offer pushdown support and categorize pushdown operators based on their key features. By deriving a shared pattern from these observations, we can establish a general principle that will potentially facilitate the discovery of new pushdown operators (Section 5.3).

5.2 Adaptive Pushdown

With computation pushdown, typically a query plan is split into two parts: the pushdown portion is executed in the storage layer, and the remaining part is executed in the computation layer. Existing cloud OLAP DBMSs execute the query plan with pushdown *eagerly*, where the entire pushdown portion of the query plan is placed and executed in storage. However, as discussed in Section 5.1, this may be sub-optimal since the storage layer may have insufficient computational resource at runtime (e.g., due to other pushdown requests), leading to performance degradation.

We propose that the DBMS should perform computation pushdown in storage *adaptively*, by taking into account the storage-layer resource utilization status. The high-level architecture of adaptive pushdown is shown in Figure 5.2. Instead of blindly executing all pushdown tasks in the storage layer, only a portion of tasks that the pushdown engine can sustain are offloaded. For the rest computation tasks, only the accessed raw data is returned to the compute nodes and no pushdown occurs. The two portions of computation results are then combined at the computation layer.

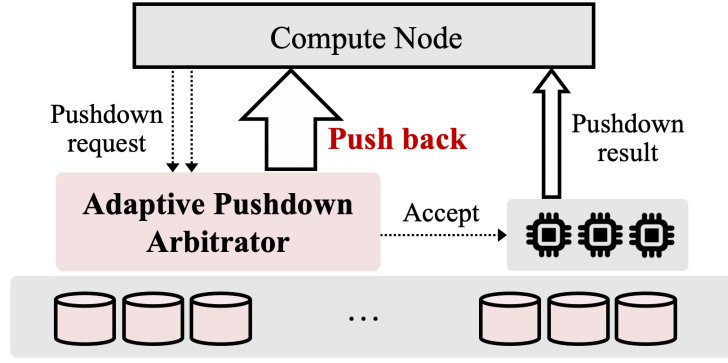


Figure 5.2: **Architecture of Adaptive pushdown** — The *Adaptive Pushdown Arbitrator* determines whether to accept a pushdown request for execution or push it back.

One major challenge of adaptive pushdown is that it is hard for the compute nodes to collect accurate statistics from the storage layer to decide whether a pushable task should be offloaded. Furthermore, even if such information can be accurately collected at planning time, the resource utilization in the storage layer may change at runtime. In this section, we develop a *pushback* mechanism to let the storage layer instead of the computation layer to make the pushdown decision at runtime.

5.2.1 Theoretical Analysis

We begin by analyzing the theoretical optimal bound of adaptive pushdown — the optimal division of the computation tasks between pushdown and non-pushdown to achieve the best overall performance. Specifically, we analyze the workload under the following assumptions.

- The workload (which may consist of multiple concurrent queries) contains N pushdown requests submitted to the storage layer in parallel.
- Each pushdown request consumes the same amount of computational resource bw_{pd} when admitted, or the same amount of network resource bw_{pb} when pushed back. The total available CPU and network bandwidth in the storage are BW_{cpu} and BW_{net} .
- The overall execution time of the pushable query plan portion is T_{pd} with pushdown enabled, and T_{npd} with pushdown disabled, where $\frac{T_{npd}}{T_{pd}} = k$. Intuitively, k determines the maximum speedup that any pushdown technique can possibly achieve.

We use the following terms to describe the pushdown decisions made at the storage layer.

- Among N pushdown requests arriving at the storage, n requests are admitted and $N - n$ requests are pushed back.
- The admitted pushdown requests result in an overall execution time of T_{pd_part} , and requests that are pushed back take T_{pb_part} to finish.

Since the admitted pushdown requests at the storage layer and the pushback requests at the compute layer are executed in parallel, the overall execution time of the pushable portion in the query plan can be formulated as follows (Equation 5.1).

$$T = \max\{T_{pd_part}, T_{pb_part}\} \quad (5.1)$$

In the optimal case, the storage layer would have a global view of all requests that it will receive ahead of the query execution. Therefore, an optimal split of the pushdown requests for admission and pushback can be constructed. Intuitively, with more pushdown requests admitted at the storage, T_{pd_part} becomes larger, and T_{pb_part} gets smaller, and vice versa. The overall execution time obtains the minimum when these two terms are equal, namely:

$$T_{opt} = T_{pd_part} = T_{pb_part} \quad (5.2)$$

We assume the pushdown tasks are bounded by CPU computation and pushback tasks are bounded by network. Therefore, Equation 5.2 can be further expanded as follows (Equation 5.3).

$$\frac{n \cdot bw_{pd}}{BW_{cpu}} = \frac{(N - n) \cdot bw_{pb}}{BW_{net}} \quad (5.3)$$

T_{pd} and T_{npd} can be expressed as Equation 5.4.

$$T_{pd} = \frac{N \cdot bw_{pd}}{BW_{cpu}}, \quad T_{npd} = \frac{N \cdot bw_{pb}}{BW_{net}} \quad (5.4)$$

Since we know $\frac{T_{npd}}{T_{pd}} = k$ by assumptions, we plug it into Equation 5.4 and obtain Equation 5.5.

$$\frac{kN \cdot bw_{pd}}{BW_{cpu}} = \frac{N \cdot bw_{pb}}{BW_{net}} \quad (5.5)$$

Combining Equation 5.5 and Equation 5.3, we can express n as follows (Equation 5.6).

$$n = \frac{k}{k+1}N \quad (5.6)$$

Additionally, the optimal execution time can be expressed as follows (Equation 5.7).

$$T_{opt} = \frac{k}{k+1}T_{pd} = \frac{1}{k+1}T_{npd} \quad (5.7)$$

Intuitively, a larger k means a higher speedup through pushdown, leading to more tasks executed in the storage (n being larger). Even when the pushdown layer processes tasks slower than the compute layer ($k < 1$), it can still accelerate the system by offloading some computation. When the pushdown layer is extremely slow or does not exist ($k = 0$), all tasks are pushed back to the compute layer and no pushdown occurs. Note the optimum (Equation 5.6) can only be approximately satisfied in practice, since the number of pushdown and pushback tasks can only be integers. For example, assume a query submits ten requests to storage, and the optimal division of pushdown and pushback tasks is 7.7 versus 2.3. In practice we have to round them to the closest integers, i.e., execute eight requests at the storage and push back the rest two requests.

5.2.2 Pushback Mechanism

In our design, the compute nodes always try to offload all pushdown tasks as if the storage has abundant computational resource. When the storage server receives a pushdown request, the *Adaptive Pushdown Arbitrator* (see Figure 5.2) determines whether the pushdown request should be accepted and executed. If the storage server is busy, the pushdown request is rejected and the computation task is *pushed back*, in which case the raw data is returned and processed at the compute node as if pushdown did not happen. The query plan at the computation layer is then dynamically adjusted to accommodate the pushback.

Algorithm 4: Pushback Mechanism at the Storage Layer

State: wait queue: Q_{wait}
 pushdown execution slots: $S_{exec-pd}$
 pushback execution slots: $S_{exec-pb}$
Assume: all incoming requests first enter Q_{wait}

```

1 while  $Q_{wait}$  is not empty do
2    $req = Q_{wait}.front()$ 
3    $t_{pd} = estimate\_pushdown\_time(req)$ 
4    $t_{pb} = estimate\_pushback\_time(req)$ 
5   if  $t_{pd} < t_{pb}$  then
6      $success = try\_pushdown(req, S_{exec-pd})$  or
7      $try\_pushback(req, S_{exec-pb})$ 
8   else
9      $success = try\_pushback(req, S_{exec-pb})$  or
10     $try\_pushdown(req, S_{exec-pd})$ 
11   if  $success$  then
12      $Q_{wait}.dequeue()$ 
13   else
14     break

```

Algorithm 4 illustrates the pushback mechanism deployed in the storage layer. It is invoked when a new request arrives or a running request completes. The state maintained in the pushdown node includes a *wait queue* (Q_{wait}), which is used to buffer excess pushdown requests when the server is under heavy load, and a finite set of *execution slots* ($S_{exec-pd}$, $S_{exec-pb}$) for both pushdown and pushback executions, to help isolate performance among different executions and avoid throttling. We assume all incoming requests first enter Q_{wait} .

For each request in the wait queue (line 1–2), we begin by estimating the execution time for both pushdown and pushback (lines 3–4), which are classified as the faster path and slower path respectively through comparison (line 5). The adaptive pushdown arbitrator first tries to assign the request to the faster path (line 6 and line 9). If the assignment is not successful due to resource contention, i.e., the corresponding execution slots are full, the arbitrator then tries to assign the request to the slower path (line 7 and line 10). If at least one assignment is successful, the request is removed from the wait queue and admitted for execution, and we start evaluating the next request in the wait queue. The process pauses when both computation and network

resources are saturated (line 14), which behaves as a back-pressure mechanism to limit data processing rates. The intuition here is that the storage server should balance the resource utilization between CPU and network adaptively. In practice, we observe Algorithm 4 is lightweight and the overhead is negligible.

Time estimation (line 3–4) follows the same methodology adopted by Weighted-LFU cache replacement policy (Section 4.4.4), where the pushdown time (t_{pd}) consists of three components: data scanning, computation, and network transfer of pushdown results. Similarly, the pushback time (t_{pb}) contains data scanning, network transfer of raw data, and compute-layer computation. Note data scanning is included in both pushdown time and pushback time, and will cancel each other when compared in Algorithm 4 (line 5).

We further simplify the time estimation by ignoring compute-layer computation in pushback time, based on two observations. First, in a storage-disaggregated architecture, usually raw data transfer dominates the pushback time so the computation component has little effect (in Section 5.3.1 we observe most existing pushdown operators are bounded). Second, the storage layer is unaware of the computation bandwidth of the compute nodes, which can vary across different users. This may bring some inaccuracies into the estimation, but overall we observe a near-optimal result on how aggressive pushdown should be and better performance. Section 5.5.2 will show the pushback mechanism closely aligns with the theoretical analysis in Section 5.2.1.

5.2.3 Awareness of Pushdown Amenability

The wait queue deployed in Algorithm 4 manages the arriving pushdown requests in a FIFO order — whether the request will be accepted is purely based on current resource utilization status. However, some requests may benefit more on pushdown compared to other requests (e.g., the request has a selective filter but incurs little computation). Intuitively, these pushdown-amenable requests should be given a higher priority to be executed in the pushdown path than the requests that cannot benefit a lot by pushdown.

Consider a scenario where the wait queue contains two requests: r_1 with $t_{pd} = 3$ and $t_{pb} = 4$, followed by r_2 with $t'_{pd} = 1$ and $t'_{pb} = 4$. The two requests have the same estimated pushback time but differs on the estimated pushdown time. Assume at a moment one request can be admitted for pushdown execution and the other needs to

be pushed back. Algorithm 4 would first evaluate r_1 and places it into the pushdown path, then evaluate r_2 with it pushed back. However, a better solution would be to push back r_1 instead of r_2 , since r_2 incurs a lower execution time by pushdown.

Given a request, we define *Pushdown Amenability* (PA) as the potential benefit of pushdown compared to pushback, which can be expressed as Equation 5.8.

$$PA = t_{pb} - t_{pd} \quad (5.8)$$

At runtime, the arbitrator keeps the wait queue sorted by the PA value of the requests. Pushdown execution always consumes the request with the highest PA value, and pushback execution does the reverse. In the example shown above, $PA(r_1) = 1$ and $PA(r_2) = 3$, which means the storage server would consider executing r_2 in the storage and pushes r_1 back. Compared to Algorithm 4, the total amount of consumed CPU and network resources are potentially decreased.

5.3 Analysis of Pushdown Operators

In this section, we aim to obtain a comprehensive understanding of the effects of computation pushdown to cloud OLAP DBMSs. We collect the supported pushdown operators in existing cloud DBMSs, and by observing the common patterns, we derive a general principle about whether an operator is amenable to pushdown (Section 5.3.1). Following the principle, we further identify two operators that can benefit from pushdown to the storage, in particular, *selection bitmap* and *distributed data shuffle* (Section 5.3.2).

5.3.1 Key Characteristics

Table 5.1 summarizes the pushdown support in existing cloud OLAP DBMSs. Selection, projection, and aggregation are mostly considered due to the wide usage and ease of development (e.g., AQUA, S3 Select, Presto, etc.). Beyond this, different systems adopt a customized set of pushdown operators. Grouped aggregation can be pushed to the storage by Redshift Spectrum. PushdownDB supports pushdown of grouped aggregation, top-K, and Bloom filters using existing APIs of S3 Select. However, due to the intrinsic restrictions of S3 Select interfaces, the implementations are not efficient. First, pushdown of grouped aggregation has to be processed in two phases, resulting

Table 5.1: Supported Pushdown Operators in Existing Cloud OLAP DBMSs — FPDB refers to the work [208] discussed in Chapter 4, which further supports hybrid pushdown and caching query execution. Marked with *: Pushdown of grouped aggregation and Bloom filters are not efficiently supported by PushdownDB, join pushdown in PolarDB-X requires both tables co-located on the join key, and top-K pushdown in PolarDB-MySQL requires indexes on the sort key.

Operator	Selection	Projection	Scalar Agg	Grouped Agg	Bloom Filter	Top-K	Sort	Join	Merge
Redshift Spectrum	✓	✓	✓	✓					
AQUA	✓	✓	✓						
S3 Select	✓	✓	✓						
Presto	✓	✓	✓						
PushdownDB	✓	✓	✓	✓*	✓*	✓			
FPDB	✓	✓	✓						
Azure Data Lake Query Acceleration	✓	✓	✓						
Snowflake Query Acceleration Service	✓	✓	✓						
PolarDB-X	✓	✓	✓	✓		✓	✓	✓*	
PolarDB-MySQL	✓	✓			✓	✓*			

two rounds of data exchange. Second, Bloom filters are required to be serialized explicitly into strings with 0s and 1s, which is space- and computation-inefficient. FlexPushdownDB ([208]) improves the efficiency of pushdown computation through a fine-grained hybrid execution framework with caching. PolarDB-X [48] enables pushdown for a specific join type which requires both tables co-partitioned on the join key. PolarDB-MySQL [47] supports pushing Bloom filter and top-K operations to the underlying storage, but does not support aggregation pushdown.

We conclude the following key characteristics from Table 5.1 that contribute to the suitability of a pushdown operator.

Key Characteristics of Pushdown. *The required storage-layer computation is local and bounded.*

5.3.1.1 Characteristic 1: Locality

Locality means the computation tasks placed at the storage do not incur any network traffic across the storage servers — the traffic occurs only between the compute and storage layers.

Analysis of Operators. Popular pushdown operators like selection, projection, and scalar aggregation comply with locality, since no network traffic is incurred across storage nodes. The same rationale also applies to operators including grouped aggregation, Bloom filter, top-K, sort, where the computation functionality can be performed on each individual data object.

General join does not embrace locality, unless the two joining relations are co-partitioned using the join key (PolarDB-X). Otherwise the data needs redistributed across the network. Another example of non-local operator is *Merge*, which combines the output of multiple upstream operators (e.g., select, project, aggregate, sort, etc.). Merge requires data exchange within the storage layer since data objects are typically spread across multiple storage servers. As Table 5.1 shows, none of the existing systems pushes merge to the storage.

Potential Advantages. We believe locality is an important characteristic for pushdown operators for three reasons. First, a pushdown environment must support multi-tenancy. Forbidding data exchange across storage servers can reduce the performance variations in pushdown tasks (e.g., due to network interference and queueing). Second, cloud storage must be encrypted using protocols like TLS [169] during data transfer. Local operators avoid the complexity of encryption and decryption across storage servers (e.g., distributing private keys). Third, local pushdown keeps the design of the storage layer simple, since otherwise it needs to support the server-side functionalities of client-server APIs.

5.3.1.2 Characteristic 2: Boundedness

Bounded implies the pushdown tasks require at most linear amount of CPU and memory resources over the accessed data size.

Analysis of Operators. Selection, projection, and scalar aggregation are linearly

bounded since the CPU consumption is linear to the size of the processed data, and the memory consumption is a constant. Grouped aggregation consumes linear CPU and linear memory capacity. Bloom filter can be regarded as a special regular filter which is thus also linearly bounded. Top-K is typically implemented using a max or min heap which consumes $O(K)$ memory and $O(N \log K)$ execution time, where N is the size of the input data. The variable K is a constant and typically much smaller than N , making the time complexity also linear in N .

The computation complexity of the sort operation is not linearly bounded. The boundedness of the join operator depends on the cardinality of its output. For example, in a key-foreign key join, the output size is bounded by the larger input table, which also bounds the memory usage and computational complexity. Non-equi-joins are typically computed using a nested loop, resulting in complexity that grows beyond linear.

Potential Advantages. Supporting only *bounded* operators preserves the key benefits of storage-disaggregation, where the storage service scales only based on the volume of the stored data, regardless of the computational resource consumption of the workloads. If pushdown operators are not bounded (linearly), the computational resource consumption may grow beyond the scale of the store data, which requires the storage servers to balance between storage and computational needs, which defeats the purpose of storage-disaggregation.

5.3.2 Proposed Pushdown Operators

Following the key characteristics derived from existing systems, we identify two commonly used operators in modern distributed query processing that can also benefit from pushdown to storage, but have not been deeply investigated previously. First, we observe that the hybrid pushdown and caching execution (Section 4.3) may waste some cached segments when they cannot cover the filter predicates entirely. To tackle this, we leverage late materialization to push down *selection bitmap* as the form of the filter predicate to better use the cached data. Second, we noticed that existing pushdown operator are not specifically designed for distributed execution. Therefore, we propose to offload *distributed data shuffle* to the storage layer to reduce network traffic.

5.3.2.1 Selection Bitmap

Late materialization is adopted by columnar OLAP engines broadly, as demonstrated by various previous studies [183, 163, 142, 55, 167]. Selection bitmaps are one common technique embracing late materialization, and DBMSs frequently filter columnar data using selection bitmaps. For instance, when selecting a column based on a filter predicate on another column, the predicate column is read in first to generate a selection bitmap. This bitmap is then used to filter the selection column in later execution stages.

Storage-disaggregation opens up new design space to better utilize late materialization, where bitmap construction and bitmap apply may not happen in the same layer, since modern columnar query engines cache individual columns within the compute nodes [65, 92, 8, 38]. Selection bitmap pushdown essentially is a variant of regular filtering pushdown, and hence is both local and bounded. In the following discussion, we will delve deeper into this larger design space by demonstrating with two prevalent cases, and then formulate a comprehensive solution that can be applied more broadly.

Case 1: Selection Bitmap from the Storage Layer. Selection bitmaps can be transferred from the storage to the compute layer when they can only be created at storage — when the storage evaluates pushdown computation (i.e., filter predicates), it actively constructs a bitmap which is later leveraged by the compute nodes to filter columns in the local cache.

W.l.o.g., we use the filtering query below to demonstrate how selection bitmap pushdown works, which essentially evaluates a set of filter predicates on attribute set B and returns both columns of both attribute sets A and B. In more general cases, the filtering query can be a subquery in more complicated queries, for example, producing a input joining relation.

```
SELECT A, B FROM R
WHERE [predicates on B]
```

Listing 5.1: A General Filtering Query (A and B refer two attribute sets).

Assume columns A are stored in the local cache, as Figure 5.3 shows. Conventionally, the DBMS needs to load the missing columns B to evaluate the filter (Figure 5.3a) (it is also possible that the DBMS pushes down the entire scan and loads both filtered columns A and B). The hybrid solution proposed in [208] also requires the predicate

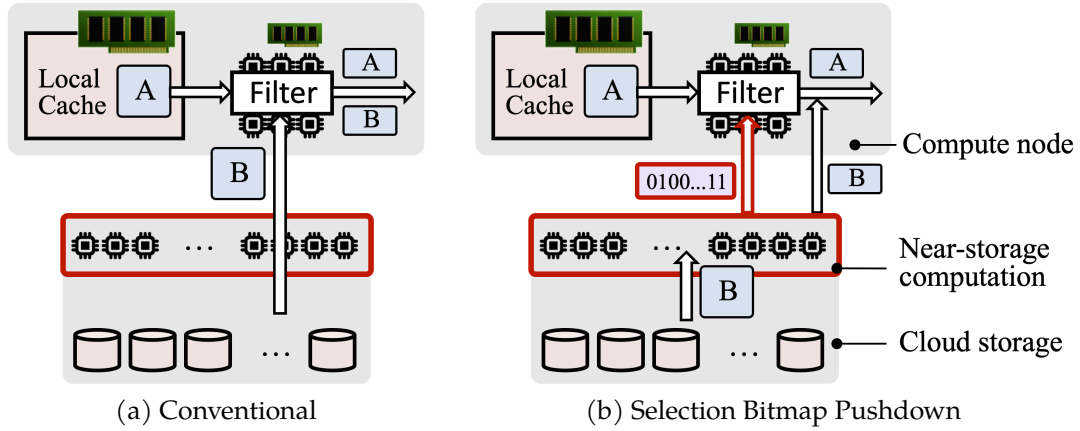


Figure 5.3: Selection Bitmap Pushdown (from the Storage Layer) — Selection bitmap constructed at storage can be used to filter cached data at the compute layer.

column B loaded to the compute node. Figure 5.3b illustrates the case when selection bitmap pushdown is enabled. The storage layer first constructs a selection bitmap by evaluating the filter predicates on columns B. Then the bitmap is sent to the compute node, where columns A are loaded from the cache and filtered by directly applying the bitmap. At the same time, filtered columns B are returned to the compute node, forming the final result together with filtered columns A.

Case 2: Selection Bitmap from the compute layer. When the selection bitmap can be constructed in the compute layer, it can be sent to the storage layer to do filtering without touching the predicate columns from the disks.

Specifically, assume columns B are cached when executing Listing 5.1. As Figure 5.4 shows, conventionally, the storage layer must scan both columns A and B to get the result back to the compute layer (Figure 5.4a). Selection bitmap pushdown can reduce the number of columns scanned at the storage layer. As Figure 5.4b demonstrates, before sending the pushdown request, the compute node evaluates the filter predicate on B from the local cache, and constructs a bitmap representing the filter result. The bitmap is then sent to storage such that the storage server can perform filtering without loading columns B from disks. Moreover, the CPU cycles used to evaluate filter predicates are eliminated at the storage layer. Our prior work [208] in Section 4.3 adopts the conventional approach since it does not support transferring bitmaps between the compute and storage layers.

Complete Fine-Grained Solution. Selection bitmap pushdown from the storage layer

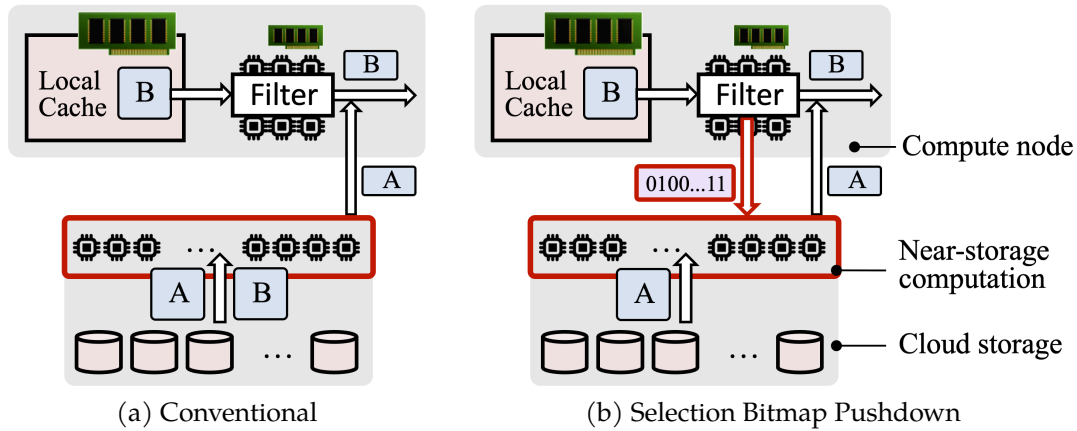


Figure 5.4: Selection Bitmap Pushdown (from the Compute Layer) — Storage can use the compute-layer selection bitmap to perform filtering without touching the predicate columns.

and the compute layer complements each other, as one of them is always applicable depending on what data columns are cached. In practice, filter predicates are often composed of multiple sub-predicates connected by ‘AND’ or ‘OR’, and these sub-predicates may themselves be composite. In these situations, a fine-grained execution framework can be used to combine the benefits of both storage-side and compute-side selection bitmaps. For example, on evaluating the predicate “(A or B) and C”, assume columns A and B are cached. We can assign the sub-predicate “A or B” to be evaluated at the compute layer, and leave the rest “C” to the storage layer. Selection bitmaps are constructed in both layers and then exchanged, allowing for the formation of a complete selection bitmap that corresponds to the input filter predicate, through inexpensive bitwise operations. Subsequently, the complete selection bitmap is used to filter both the cached columns at the compute layer and the uncached columns at the storage layer.

5.3.2.2 Distributed Data Shuffle

Shuffle is a commonly used operator to redistribute data among compute nodes, for example, when the downstream operator is an equi-join and the two joining relations need to be redistributed based on the join key. In a traditional shared-nothing architecture, shuffle moves data from one node to another based on a partition function (hash-based, range-based, etc), so each data record experiences one network transfer. However, in a storage-disaggregation architecture, as Figure 5.5a shows,

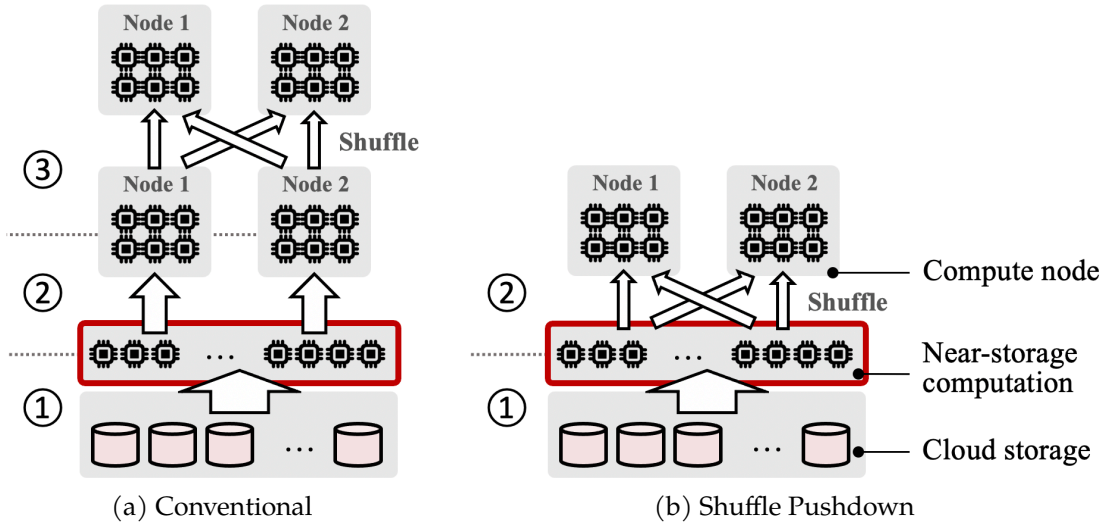


Figure 5.5: Distributed Data Shuffle Pushdown — Data is directly redistributed to the target compute node from the storage layer.

existing systems (e.g., Presto) involve two network transfers — load data from the storage (Step 2) which is pre-filtered within the storage (Step 1, e.g., selection and projection pushdown), and then exchange data across compute nodes (Step 3).

Figure 5.5b presents the proposed design where we push the shuffle operators into the storage layer. In this case, the processing of the pushdown tasks is initiated on the storage servers without shuffling (Step 1). Before returning the pushdown results to the compute layer, the data is partitioned and directly forwarded to the appropriate target computation nodes (Step 2). Essentially, the new shuffle design merges steps 2 and 3 in the previous design into a single step, which reduces two network transfers to one.

The compute layer needs to send some key parameters to the storage layer for it to conduct the shuffle operation. These include a partition function, the partition key, and the identifiers of the target compute nodes (e.g., IP addresses and keys in FPDB, for FaaS platforms this requires more complicated designs which are beyond the scope of this paper) which the shuffled results are returned to. In our implementation, pushdown requests are sent per data partition, and the parameters used in shuffle processing are attached to each pushdown request when sent to the storage layer. The shuffle operation for a data partition is initiated once its upstream operators are finished (e.g. scan, filter).

There are two approaches for a storage server to transfer shuffled data to target

compute nodes: (1) actively pushing the data to the compute nodes, or (2) buffering the shuffled results locally and waiting for compute nodes to request. We chose the latter approach since the target compute nodes may not be immediately ready to receive the data when the storage server issues the transmission. However, the storage server has limited memory space and should not indefinitely write shuffled results to the local buffer. The storage server will set an upper bound on its local buffer size; when the buffer is full, the shuffle process will throttle until the buffer is drained by the target computation node.

Pushdown of distributed data shuffle is *local* since it does not incur network traffic across the storage servers — data is solely transferred from storage to the compute layer. It is also *bounded* because essentially it involves scanning of the input data and assigning records to their corresponding partitions, which consumes CPU and memory resources linearly.

Interact with Cached Data. It is non-trivial to exploit the data in the cache while performing shuffle pushdown, since the shuffle operation changes the data distribution, which means the cached data may not be directly applicable to downstream operators.

The most straightforward method is to ignore the cached data when pushing shuffle to the storage layer — the entire table is redistributed to the compute nodes from the storage. A better solution is to perform the same shuffle function to the cached columns within the compute cluster, and only brings back shuffle results on uncached columns from the storage. The advantages are as follows. First, in an n -node cluster, a portion of roughly $\frac{1}{n}$ network traffic of redistributed data can be saved (i.e., redistributed to the same node), assuming the raw data is initially uniformly distributed into the n nodes and the shuffle function evenly partitions the data across the cluster. Second, the network bandwidth within the compute cluster is usually higher than the bandwidth between the computation and storage layer. Finally, reading cached data is more efficient than loading data from the storage devices.

However, it may not be directly applicable to perform the shuffle function over the cached data, which depends on the existence of the shuffle columns in the cache. To tackle this, we can resort to the similar insight of selection bitmap pushdown. When processing the partitioning function, the storage layer can generate a *position vector*, representing the compute node that each row should be redistributed to. For a n -node

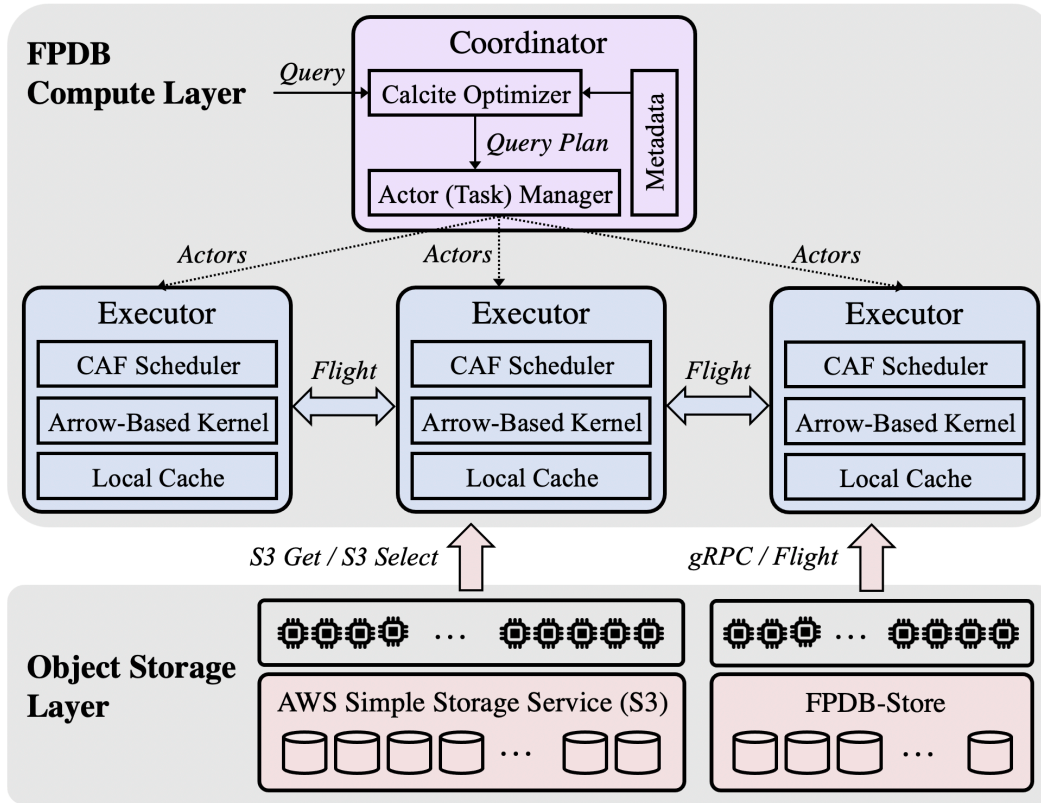


Figure 5.6: Distributed Query Execution in FPDB.

compute cluster, each position value requires only $\log_2 n$ bits, making the position vector a lightweight data structure.

Interact with the Upstream Operator. The rationale aforementioned can be generalized when the shuffle operation is not the direct downstream operator of data scan — it may be performed on intermediate results of computation like filtering and aggregation. The input data produced by the upstream operator can be divided into local portion — produced within the compute layer, and remote portion — produced within the storage layer (i.e. pushdown), which can also abstractly be regarded as cached data and uncached data respectively.

5.4 Implementation

In this work, we extend the previous FPDB implementation (Section 4.5) into a distributed, cloud-native query execution engine. Figure 5.6 illustrates the distributed query execution framework of FPDB. As discussed in Section 4.5, FPDB adopts a

storage-disaggregation architecture in which data is persistently stored in an object storage layer, such as Amazon Simple Storage Service (S3) [22]. FPDB supports querying both row-oriented formats (e.g., CSV) and columnar formats (e.g., Parquet [30]), and fully supports major analytical benchmarks, including SSB [156], TPC-H [54], JOB [139], and DSB [96].

5.4.1 FPDB Compute Layer

FPDB runs on a cluster of AWS EC2 [20] virtual machines, which contains a coordinator node and several executor nodes. When a query arrives, the coordinator is responsible for inspecting the catalog and metadata, parsing the query, and optimizing the query plan. FPDB integrates Apache Calcite [25] as an extensible cost-based query optimizer.

Actor-Based Parallel Processing. As described in Section 4.5.3, FPDB supports distributed parallel query execution using C++ Actor Framework (CAF [83]). Physical operators are wrapped into actors, which are remotely spawned at the executor nodes by the Actor Manager. A query is composed of a number of actors arranged in a tree. Within each individual executor node, the CAF scheduler multiplexes active actors over all CPU cores on the host machine.

Actors communicate via message passing from producers to consumers — messages flow from leaves to the root. FPDB manages actor behaviors using data messages and control messages. Data messages represent the execution result of the producer, as well as the data input of the consumer, which are sent in a pipelined fashion. Data messages are placed in the shared memory when the producer and consumer actors are in the same node. For data transfer among different nodes, FPDB leverages Arrow Flight [31], which achieves a higher network throughput over CAF’s builtin cross-node communication. Arrow Flight enables wire-speed, zero-copy, and serialization-free data transfer by sending data in Arrow IPC format [32], which can be processed directly by FPDB’s execution runtime. Control messages are used to instruct actors to begin execution, track their completion status, and collect operators’ execution metrics. Query execution begins on leaf operators (i.e., scan), and the results are gathered on the *collate* operator on completion.

FPDB’s local cache is also implemented as an actor with cache contents managed by the cache replacement policy. The cache actor communicates with other operator

actors through message passing to admit and evict data segments. FPDB performs caching using the main memory (RAM) of the executor nodes.

Arrow-Based Runtime. As mentioned in Section 4.5.2, within the database engine and the cache, we use Apache Arrow [24] to manage data; table data is converted to Arrow within the scan operator. Using the same data format for the processing engine and the cache eliminates the overhead of extra data conversions.

Arrow provides efficient compute kernels (aggregation, sorting, join, etc.) using optimization techniques including prefetching for cache-efficiency and vectorization via CPU SIMD instructions. Besides, FPDB uses Gandiva [151] for efficient expression evaluation, which is built on top of Arrow and further enhanced by exploiting LLVM and just-in-time expression compilation.

5.4.2 FPDB Storage Layer

FPDB supports querying data from object storage including AWS S3 and *FPDB-Store*, a S3-like storage prototype with pushdown capabilities discussed in Section 5.2 and Section 5.3.

When querying data from S3, FPDB pushes down computational tasks through S3 Select [120]; raw data retrieval is performed by issuing S3 GetObject [23] requests. To offers more complex storage-layer computation (e.g., shuffle, selection bitmap, Bloom filter, etc.), we developed *FPDB-Store*, an open-source storage layer with pushdown support.

Existing pushdown platforms (Table 5.1) are deployed in a separate layer with compute resources (e.g., Redshift Spectrum, Snowflake Query Acceleration Service, Azure Data Lake Query Acceleration, S3 Select), or by leveraging the computation power within the storage servers (e.g., PolarDB-X, PolarDB-MySQL). *FPDB-Store* emulates the functionality of a pushdown layer by deploying onto EC2 storage-optimized instances. Data objects are stored on file systems located on locally attached SSDs, and fetched by the compute nodes via gRPC [41]. Arrow Flight is utilized to send pushdown requests to the storage layer and get results back to the compute layer, to avoid costly data format conversions. Each pushdown request contains a serialized query plan instead of a plain SQL dialect [120, 2], to avoid redundant query parsing and planning in the storage. *FPDB-Store* reuses the same distributed MPP engine of FPDB when processing pushdown tasks.

For pushdown operators with bitmaps (e.g., selection bitmap, Bloom filter), FPDB wraps bitmaps into single-column Arrow tables for low serialization overhead. FPDB also avoids two-phase processing [217] when pushing down grouped aggregation to achieve lower latency. In adaptive pushdown, when the storage decides to push back a request, a special Flight error will be returned to the computation layer, where the corresponding compute node will issue a another gRPC call to retrieve the compressed raw data.

5.5 Evaluation

In this section, we evaluate the performance of adaptive pushdown and new pushdown operators using standard OLAP benchmarks.

5.5.1 Experimental Setup

Hardware Configuration. We conduct all the experiments on AWS EC2 virtual machines. We use r5.4xlarge instances (which costs \$1.008 per hour in US-East-2 pricing) with 16 vCPU, 128 GB memory, and up to 10 Gbps network bandwidth in the computation layer. The storage layer is set up on r5d.4xlarge instances (which costs \$1.152 per hour in US-East-2 pricing), with 16 vCPU, 128 GB memory, up to 10 Gbps network bandwidth, and two 300 GB local NVMe SSDs. All servers run the Ubuntu 20.04 operating system. For experiments with distributed data shuffle pushdown, we configure both the compute layer and the storage layer to 4-node clusters. Otherwise, we configure both layers to a single instance.

Benchmark. We use the widely adopted data analytics benchmark, *TPC-H* [54]. TPC-H contains 22 queries in total. We use a scale factor of 100 (100 GB data set when uncompressed) for experiments with distributed data shuffle pushdown, and a scale factor of 50 (50 GB data set when uncompressed) otherwise. Each table is sharded into partitions of roughly 150 MB when using CSV format. Table partitions are converted to Parquet [30] format and evenly distributed to local disks of the storage instances. All the experiments are conducted using the Parquet data.

Measurement. We measure the execution time and other relevant metrics (e.g., network traffic, number of admitted requests, etc.) for each experiment. For each query, we run three times and record the average execution time. Since selection

bitmap pushdown needs to interact with the local cache of FPDB, we conduct separate experiments to evaluate its performance. Specifically, each experiment consists of a warm-up query that populates the data into the cache, and an execution query where the performance is measured.

5.5.2 Evaluating Adaptive Pushdown

In this section, we evaluate the performance of *Adaptive pushdown* (Section 5.2) in different storage-layer computational resource conditions. This can be affected by both the CPU power of the storage server, and its computational resource usage. We emulate the storage-layer computational resource status by varying the number of available CPU cores for pushdown tasks (with a storage computational power of 100% meaning that all CPU cores are available). Since FPDB is built on top of the actor framework [83], we achieve this by configuring the maximal number of threads that can be exploited by the actor system scheduler. At each time we measure the performance of a single query.

Baselines. We compare the proposed *Adaptive pushdown* mechanism against the following baseline designs within FPDB.

- *No pushdown*: Pushdown is disabled. Raw data is directly loaded from the storage layer. This design is used in existing systems without pushdown capabilities.
- *Eager pushdown*: Pushdown is enable and occurs whenever the computation task can be offloaded to the storage layer. This design is used in existing systems with pushdown capabilities (e.g., Redshift Spectrum, Presto, etc.).

5.5.2.1 Overall Performance

Figure 5.7 compares the execution time of *No pushdown*, *Eager pushdown*, and *Adaptive pushdown* under different computational resource utilization status in the storage. Results are normalized to *No pushdown*. When the computational resource at storage is abundant for pushdown execution (i.e., storage computational power is higher than 0.5), *Eager pushdown* outperforms *No pushdown* and is only slightly affected by the computational power at storage. As the storage computational power decreases,

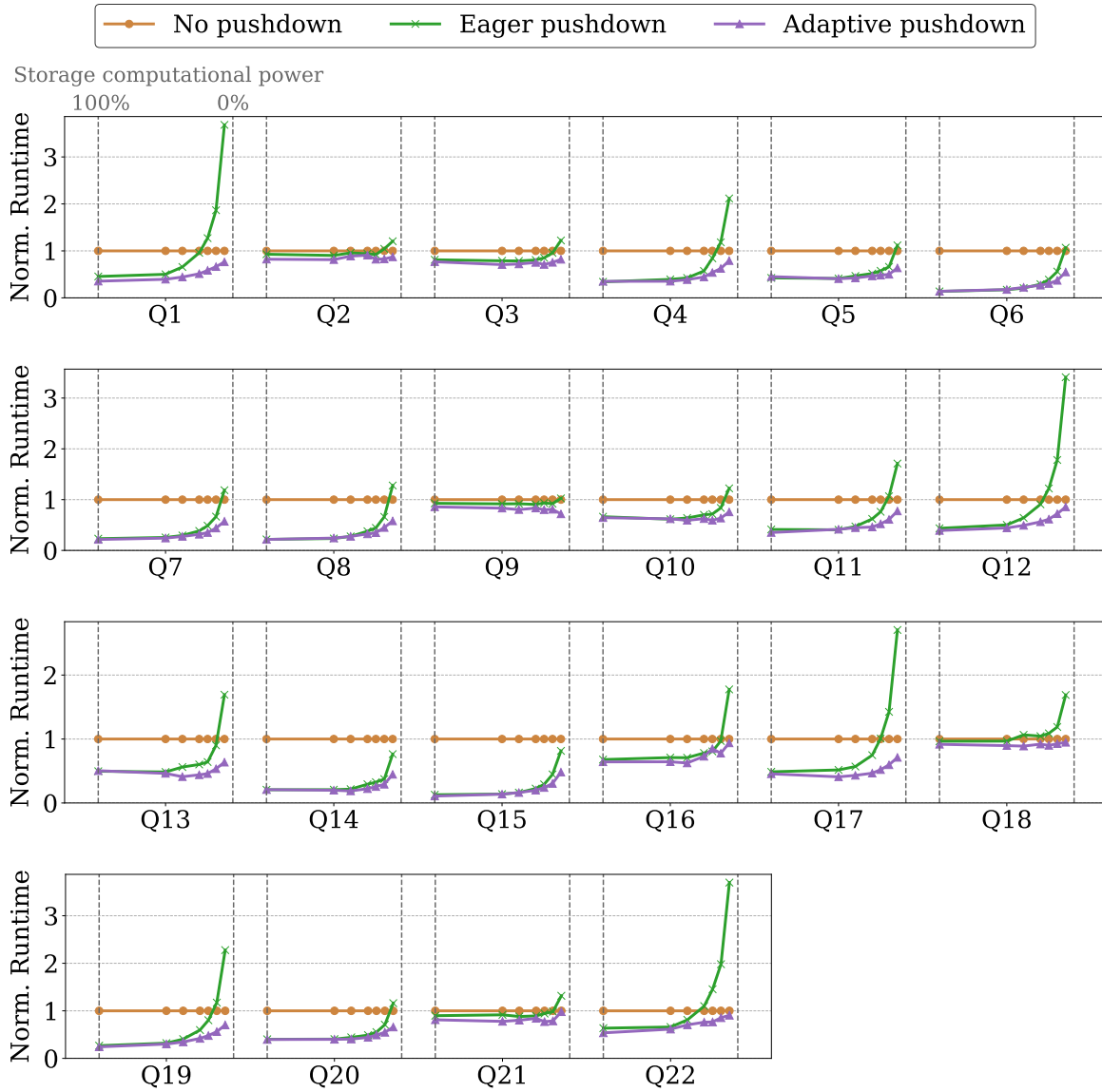


Figure 5.7: Performance Evaluation of Adaptive Pushdown on TPC-H (normalized to *No pushdown*).

pushdown execution gets throttled and gradually becomes the major bottleneck, making *Eager pushdown* underperform *No pushdown* when the storage-layer computational resource is scarce.

The performance of *Adaptive pushdown* is consistently better than both baselines. Specifically, when the storage computational power is high, it performs similarly to *Eager pushdown*, and when the storage server is tiny or under heavy burden, its performance degrades less than *Eager pushdown* and can still slightly outperform *No pushdown*. In situations where the storage computational power falls between these two extremes, *Adaptive pushdown* achieves the best of both worlds. When the performance of *No pushdown* and *Eager pushdown* breaks even, *Adaptive pushdown* outperforms both baselines by $1.5\times$ on average, and a speedup of $1.9\times$ is observed on queries including Q1, Q6, Q8, Q17, and Q19.

With pushdown enabled, the sensitivity on the storage-layer CPU utilization varies among different queries. Most queries (15 of all) expose a high sensitivity when executed with pushdown. For example, the performance of *Eager pushdown* of Q1, Q12, Q19, and Q22 is greatly impact by the storage-layer computational power, and starts to degrade even when the storage-layer CPU resource is not scarce. In these queries, the performance improvement of *Adaptive pushdown* is prominent since the pushable portion of the query plan dominates the end-to-end execution time, and *Adaptive pushdown* mitigates the issue of resource contention at storage when the pushable subquery plan is processed. For the remaining queries, the performance of *Eager pushdown* is not very sensitive to the storage-layer computational power (e.g., Q2, Q3, and Q18), where the execution time is dominated by non-pushable operators. In these queries, *Adaptive pushdown* only shows its superiority when the available computational resource at storage is extremely low. For example, *Adaptive pushdown* in Q2 outperforms both baselines by $1.2\times$ when the storage-layer computational power is 25%.

5.5.2.2 Case Study

To get a deeper understanding of the performance benefits, we conduct detailed analysis on two representative queries. We pick Q14 as the query that benefits significantly from computation pushdown, and Q12 as the query that benefits little from pushdown.

We measure the number of admitted pushdown requests at the storage layer

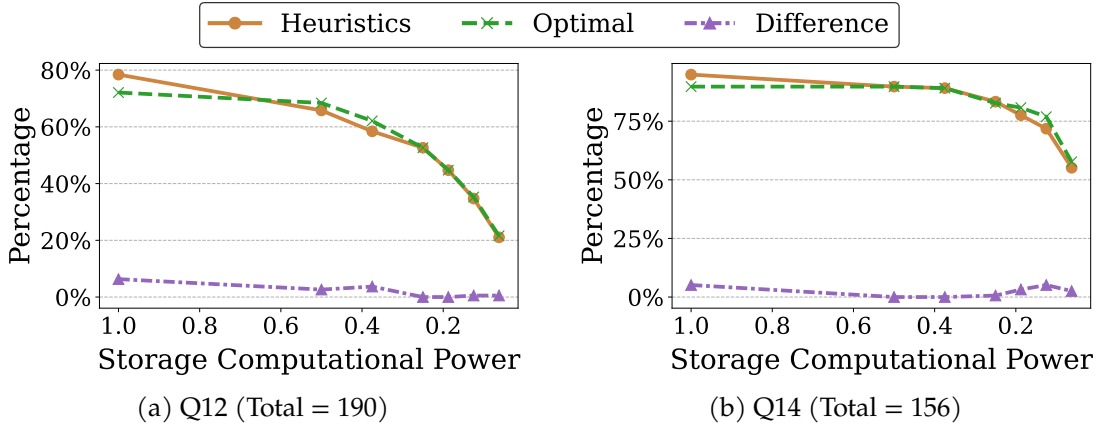


Figure 5.8: Percentage of the Number of Admitted Pushdown Requests over the Total Number of Requests.

in each experiment, and compute the ratio against the total number of requests. Figure 5.8 shows the results of both the heuristics used in the pushback mechanism (Algorithm 4), and the theoretical optimum (Equation 5.6). For both queries, with the storage computational power decreasing, fewer pushdown requests are admitted to be executed on the storage server, and more requests are pushed back to the compute layer. Compared to Q12, pushbacks in Q14 are less frequent since it achieves a higher maximal pushdown speedup, such that more tasks are executed in the storage.

Overall we observe a very small difference between the heuristics and optimum (1% on Q12 and 2% on Q14), and in some cases pushback heuristics achieve the optimal bound (e.g., for Q12 when the storage computational power is less than 25%). This demonstrates that the pushback mechanism is able to find a proper division of the computation tasks between pushdown and non-pushdown.

Figure 5.9 compares the total amount of data transfer between the storage and compute layers among different pushdown strategies. The data transfer amount of *No pushdown* and *Eager pushdown* both remain consistent, and *Eager pushdown* reduces data transfer up to an order of magnitude. The incurred network traffic of *Adaptive pushdown* is sensitive to the storage-layer computational power, since it adaptively adjusts the ratio between assigned pushdown and pushback tasks, such that both CPU and network resources at the storage server can be fully utilized.

Figure 5.10 shows the performance breakdown on the two representative queries. We present three cases where the storage-layer computational power is high, medium, and low, respectively. The execution time of the non-pushable portion of the query

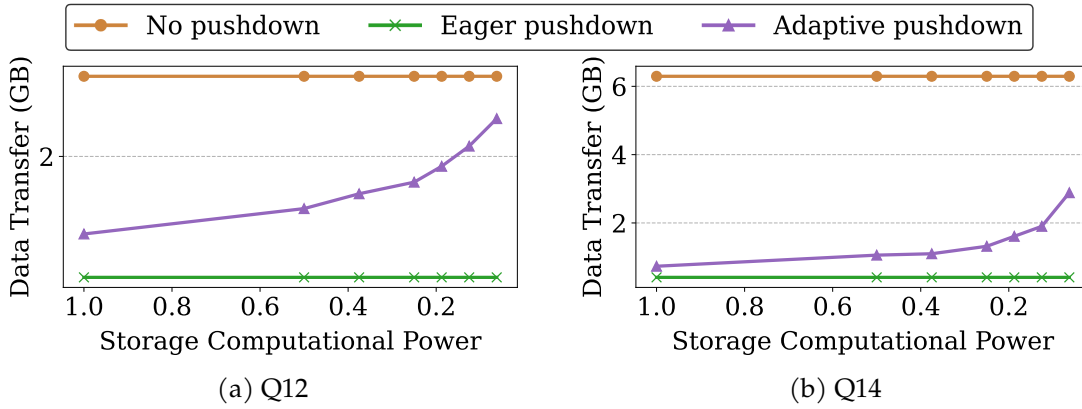


Figure 5.9: Total Amount of Data Transfer through the network (in a unit of gigabytes) of the Two Representative Queries (Q12 and Q14) among Different Pushdown Strategies.

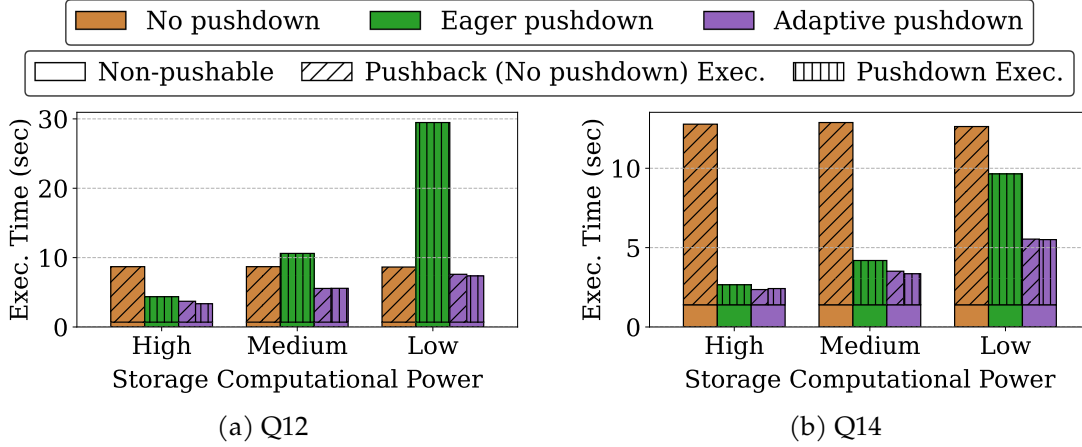


Figure 5.10: Performance Breakdown on Two Representative Queries (Q12 and Q14).

plan remains stable in all cases. Compared to fetching raw data from the storage in *No pushdown*, the overhead of pushback executions in *Adaptive pushdown* is consistently smaller since less data is returned. Compared to *Eager pushdown*, *Adaptive pushdown* executes pushdown tasks more efficiently since fewer tasks are actually admitted by the storage. We further observe in *Adaptive pushdown* that, the performance of pushdown and pushback executions are close, which means our algorithm is able to properly divide pushdown and pushback tasks to balance the usage of CPU and network resources.

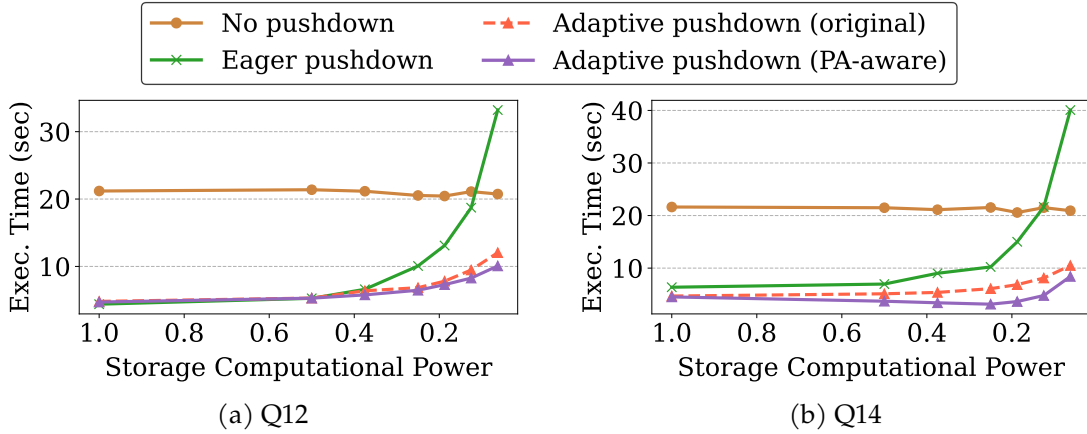


Figure 5.11: Evaluation of Awareness of Pushdown-Amenability in Concurrent Executions.

5.5.2.3 Awareness of Pushdown Amenability

Next we evaluate *Adaptive pushdown* when the pushdown requests have different pushdown amenability. Within a single query, pushdown amenability of different requests is similar, since the TPC-H dataset embraces a uniform distribution among different data partitions. Therefore, in this experiment we execute the two representative queries (Q12 and Q14) concurrently to ingest heterogeneity of pushdown amenability.

Figure 5.11 compares the original *Adaptive pushdown* and *Adaptive pushdown* with awareness of pushdown amenability. We also add *No pushdown* and *Eager pushdown* baselines for reference. Both original *Adaptive pushdown* and *Adaptive pushdown* which is aware of the pushdown amenability (*PA-aware*) outperform the two baselines. Compared to the original *Adaptive pushdown*, *PA-aware Adaptive pushdown* further improves the performance of both concurrent queries, where Q12 is accelerated by up to $1.2\times$ and Q14 is improved by up to $1.9\times$. An interesting observation on Q14 is that the performance is sometimes even improved with lower storage computation power (e.g., 0.3). This is because the performance gap between the two concurrent queries are increased, such that the contention on the non-pushable portion in the compute layer is mitigated — the slower query (Q12) has not entered the non-pushable portion when the faster query (Q14) has completed.

To understand the achieved performance improvement, we trace the number of admitted pushdown requests for both queries respectively, which is shown in Figure 5.12. Overall, we observe a decrease for the number of admitted pushdown

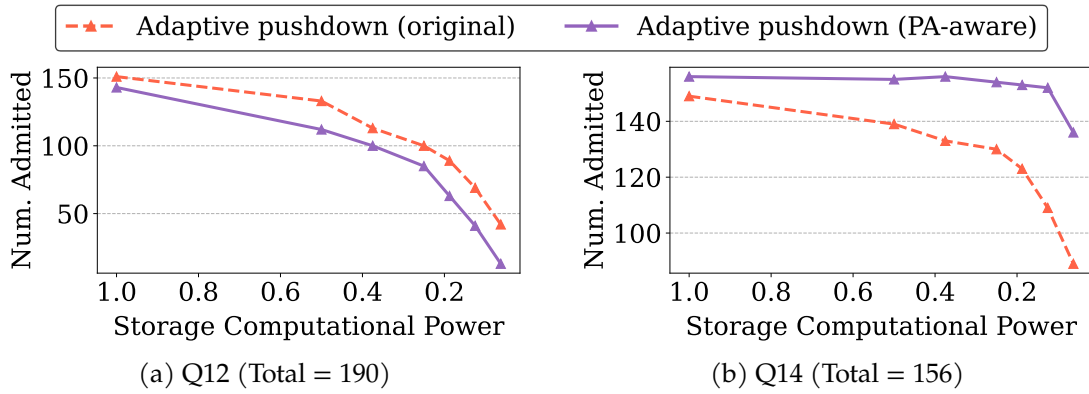


Figure 5.12: Number of Admitted Pushdown Requests at storage in Concurrent Executions.

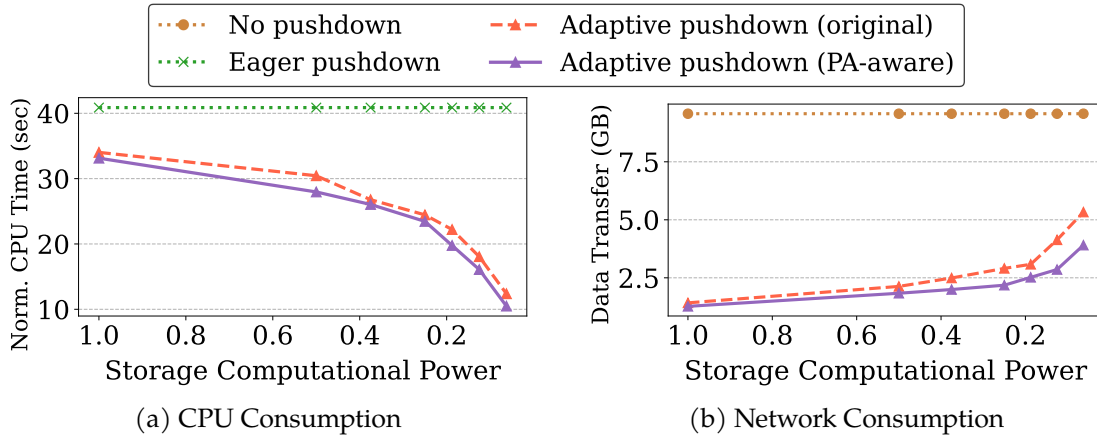


Figure 5.13: Total Resource Consumption of the Storage Layer in Concurrent Executions (CPU consumption is measured by the total CPU time that is normalized to the time of 1 vCPU, and network consumption is measured by the total amount of data transfer, in a unit of gigabytes).

requests on Q12 but an increase on Q14. This is because the requests of Q14 have a potentially larger pushdown benefit, and they are prioritized to be executed at the storage layer. Correspondingly, more requests of Q12 are pushed back to the compute layer. It is interesting to note that the performance of Q12 does not degrade but is even slightly improved, where the reasons are two-fold. First, the difference of the execution time between the pushback path and pushdown path on Q12 is not significant, so a small number of more pushback execution do not hurt the performance. Second, since the requests of Q14 are executed more efficiently, the time spent in the wait queue for the requests of Q12 decreases.

We further investigate the resource consumption of CPU and network, which is

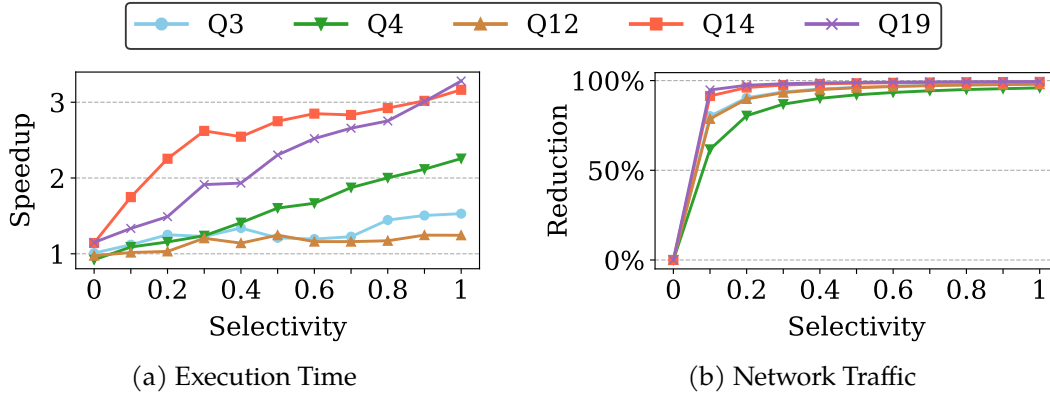


Figure 5.14: Performance Impact of Selection Bitmap Pushdown (Results are normalized to *baseline pushdown*) — The selection bitmap is constructed at the storage layer.

demonstrated in Figure 5.13. PA-aware *Adaptive pushdown* reduces CPU consumption by up to 15%, and network consumption by up to 31%, compared to the original *Adaptive pushdown*. The reduction is more significant when storage-layer computational power is lower, since more requests are pushed back to the compute layer, and PA-aware *Adaptive pushdown* is able to capture the most beneficial requests that should be pushed back.

5.5.3 Evaluating Proposed Pushdown Operators

In this section, we evaluate the performance of new pushdown operators, namely, selection bitmap and distributed data shuffle (Section 5.3.2). We implement pushdown operators in existing systems (selection, projection, aggregation, bloom filter, etc.) as the baseline (i.e., *baseline pushdown*).

5.5.3.1 Evaluating Selection Bitmap Pushdown

We first evaluate the performance of selection bitmap pushdown on several representative benchmark queries: Q3, Q4, Q12, Q14, and Q19 (other queries observe similar results). Part of the queries contain lightweight non-pushdown portion (e.g., Q4, Q14, and Q19), while the rest contain heavy or complex non-pushdown logics (Q3 and Q12). In each experiment, we vary the selectivity of the filter predicate associated with the fact table *Lineitem*.

Selection Bitmap from the Storage Layer. We first simulate the scenario where the selection bitmap can only be generated in the storage layer. We achieve this by

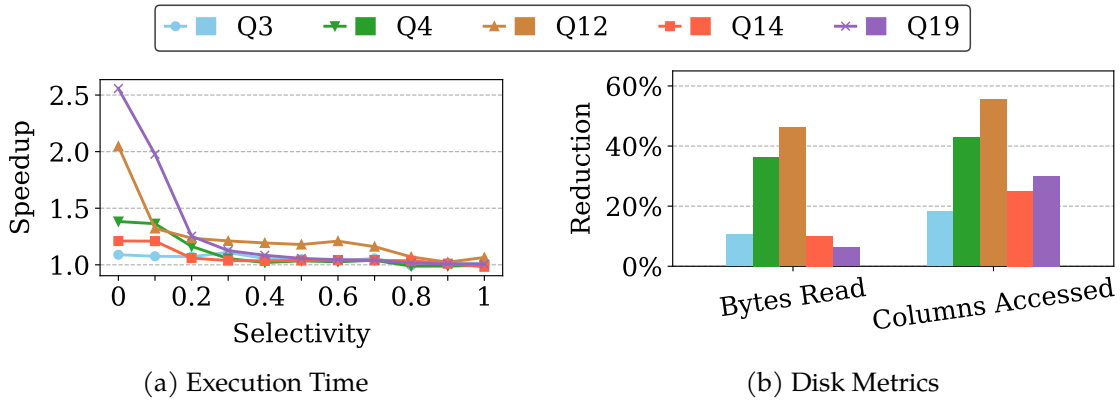


Figure 5.15: Performance Impact of Selection Bitmap Pushdown (Results are normalized to *baseline pushdown*) — The selection bitmap is constructed at the compute layer.

warming up the cache using the projected (i.e., output) columns of the filter operator. The results are depicted in Figure 5.14.

As Figure 5.14a illustrates, all queries show an improvement in performance compared to the baseline. Selection bitmap pushdown is most effective in Q14 and Q19. When the filter predicate is non-selective (e.g., 0.9), these two queries perform over $3.0\times$ better than baseline pushdown since transferring selection bitmaps instead of data columns reduces network traffic significantly (over 90% of data transfer is saved, as shown in Figure 5.14b). Q4 also observes a speedup of $2.3\times$.

When the filter predicate is highly selective, the speedup is less obvious. This is because baseline pushdown returns less data to the compute layer, such that the reduction of data transfer is less significant. Despite this, query execution still gets accelerated. For instance, when the selectivity is 0.1, the speedups of Q14 and Q19 are $1.8\times$ and $1.3\times$ respectively.

The performance gain on Q3 and Q12 is less substantial compared to the other queries. These queries contain more complex operators at the downstream of pushdown (e.g., more intricate joins and aggregations), leading to a diluted performance benefit.

Selection Bitmap from the Compute Layer. We next emulate the case where the compute-layer selection bitmap can be used to accelerate pushdown execution in the storage layer. The cache is warmed up using the predicate columns of the filter operator. Results are displayed in Figure 5.15.

As demonstrated in Figure 5.15a, all queries benefit from selection bitmap push-

down when the filter predicate is selective. For example, when the selectivity approaches 0, pushdown of selection bitmaps outperforms the baseline by $2.0\times$ and $2.6\times$ on Q12 and Q19, respectively. When the filter predicate becomes less selective, the performance gain decreases, since more data is transferred to the compute nodes, which dominates the query execution time.

We also analyze disk metrics by measuring the number of bytes read and the number of columns accessed from the disks, which are illustrated in Figure 5.15b. The amount of data scanning is reduced by 36% and 46% on Q4 and Q14 respectively, and by approximately 10% on the rest queries. Additionally, the number of columns accessed of the Parquet data decreases between 18% and 56%. The reduction of data scanning is less substantial compared to column access reduction because the columns that can be skipped via selection bitmap pushdown are typically highly compressed, as for example `l_shipmode`, which only has 7 unique values, and `l_quantity`, whose values are within a small range between 1 and 50. Conversely, columns that must be transferred are often join keys or have a decimal type, which usually have a low compression ratio.

5.5.3.2 Evaluating Distributed Data Shuffle Pushdown

Next we evaluate the performance of distributed data shuffle pushdown over TPC-H, as Figure 5.16 shows. The execution time is normalized to the *no pushdown* baseline. Across all queries, shuffle pushdown results in an average of $1.3\times$ performance improvement over baseline pushdown, and $1.8\times$ over no pushdown.

Among all 22 queries, we observe the performance improvement on 20 of them, with 15 queries are accelerated by over $1.2\times$, 10 queries accelerated by over $1.3\times$, and 6 queries accelerated by over $1.5\times$. Q7, Q8, and Q17 benefit from shuffle pushdown most significantly, which are improved by more than $1.7\times$. In these queries, the filter predicates associated with the base tables are not selective, where a major part of the table data still needs to be fetched. Shuffle pushdown is able to eliminate the redistribution of the scanned base table data, which occupies a large portion of the overall execution time. As a result, we observe more than half of the data exchange across the compute nodes is saved compared to baseline pushdown.

Conversely, several queries do not benefit a lot from shuffle pushdown (Q6, Q15, Q19, etc.). These queries typically have highly selective filter predicates on base tables, such that the amount of data transferred from the storage layer is not significant, and

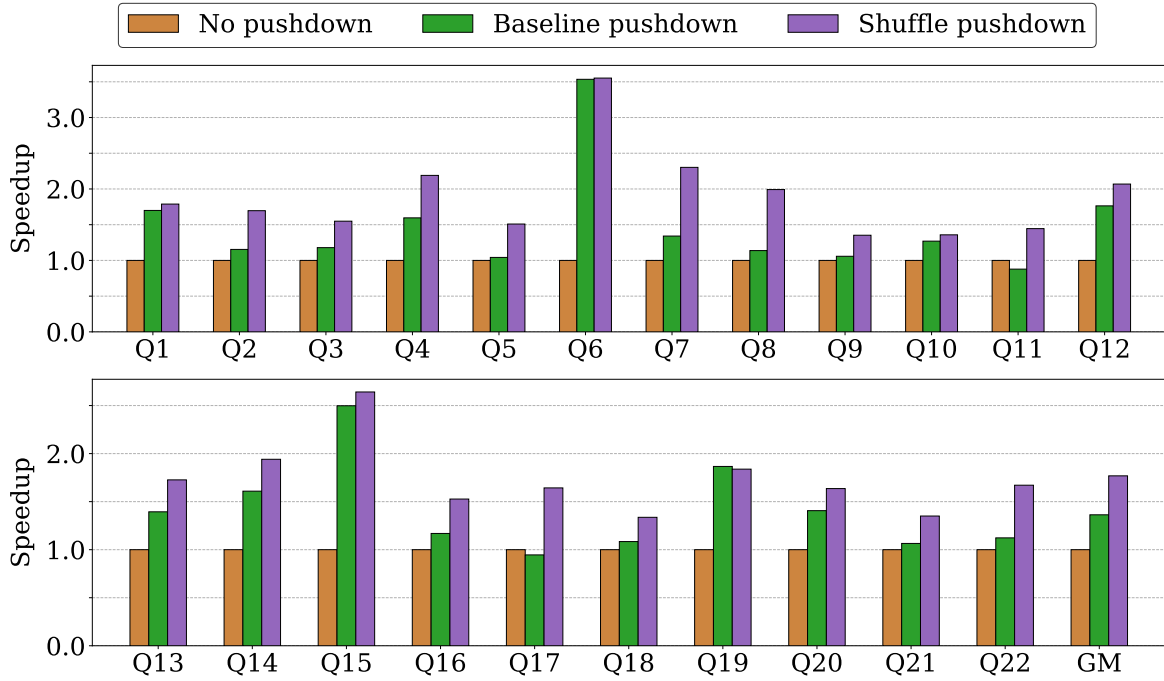


Figure 5.16: Performance Evaluation of Distributed Data Shuffle Pushdown on TPC-H (normalized to *no pushdown*).

the overhead of data exchange across the compute nodes is negligible.

We further investigate the incurred network traffic in different pushdown configurations. On average shuffle pushdown reduces the consumed network resource by 38%. Specifically, shuffle pushdown reduces the data exchange across the compute layer by 84% on average, while the network traffic between the compute and storage layers is unaffected. Out of all 22 queries, the incurred data exchange across the compute layer is reduced by over 50% on 20 queries, by over 90% on 16 queries, and by over 99% on 7 queries.

5.6 Related Work

In this section, we further discuss related work that has not been discussed earlier in this chapter and Section 4.7.

Workload Management. General workload management techniques have been intensively studied to make efficient use of system resources in addition to achieving any performance objectives [155, 220, 56], and widely deployed in industrial commercial systems [5, 16, 195]. For example, SQL Server Resource Governor [16] provides

multi-tenancy and resource isolation on single instances of SQL Server that serve multiple client workloads. Moreover, YARN [195] decouples resource management from the programming model for Hadoop’s [27] compute platform. However, existing workload managers fall short of considering the broader optimization space in a storage-disaggregated architecture, where a pushdown task can either be executed and pushed back. In this work, pushdown tasks are managed through adaptive pushdown to improve resource efficiency.

Adaptive Query Processing. There is a rich literature on the topic of adaptive query processing [95, 117, 67], which adjusts the query execution dynamically based on more accurate runtime statistics. Examples include re-optimization techniques [125, 206] and memory adaptive operators [158, 159, 221]. Adaptive query processing is also integrated into modern commercial systems such as Spark [15] and SQL Server [11]. However, optimizing adaptive processing in the pushdown context exposes new design constraints, including limited computational power in the storage, limited network bandwidth and the coordination between compute and storage layers, which are addressed in this work.

5.7 Conclusion

This paper presents adaptive pushdown, a technique that leverages a pushback mechanism to determine whether a pushdown operator should be executed in the storage adaptively based on the storage-layer resource utilization. We additionally conduct a systematical analysis of existing pushdown operators and propose two new operators that can benefit from offloaded to storage. Evaluation shows that adaptive pushdown and the new pushdown operators lead to up to $1.9\times$ and $3\times$ speedup on TPC-H benchmark respectively.

Chapter 6

Sirius: A Drop-In GPU-Native SQL Engine

In addition to the architectural shift toward cloud-native databases discussed in the previous chapters, the infrastructure stack for large-scale analytics is also evolving toward GPU-native execution. Modern GPUs now surpass CPUs in both compute throughput and memory bandwidth, making them an increasingly attractive platform for compute-intensive analytical workloads such as machine learning. Table 6.1 compares recent CPU and GPU architectures, highlighting the contrast in core count and memory bandwidth. Given the inherently parallel nature of relational data analytics, GPUs are a perfect fit for future analytical databases.

Despite their great potential, however, GPU-based SQL engines [10, 42, 62, 36, 51] have not yet seen mainstream adoption. Prior efforts, both academic and commercial, have been constrained by the following four key challenges and thereby remain niche solutions.

- **Limited GPU memory capacity:** Up to 288 GB in a latest GPU device in contrast

Table 6.1: Comparison of CPU and GPU Instances

	Amazon c6a.metal (AMD EPYC CPU)	GH200 (NVIDIA GPU)
Core Count	192 (vCPUs)	14,000+ (CUDA cores)
Memory BW	~400 GB/s	3,000 GB/s (HBM)
Memory Size	384 GB	96 GB (HBM)
Rental Cost	\$7.344/h (AWS)	\$1.5/h (Lambda Labs)

to several TBs in CPUs.

- **Data movement bottleneck:** Traditionally, PCIe has relatively low bandwidth, creating a bottleneck when transferring data between GPU and the rest of the system.
- **Expensive GPU hardware:** High-end GPUs remain significantly more expensive than CPUs and are often in short supply.
- **Engineering cost:** Building a full SQL engine optimized for GPUs from the ground up is a major engineering challenge.

In this work, we argue that the hardware and software foundations are finally in place today to overcome these challenges and enable scalable GPU data analytics. On the hardware side, GPU memory capacity doubles almost every generation, starting from Volta (32 GB), to Ampere (80 GB), Hopper (192 GB), and most recently Blackwell (288 GB). Better interconnects such as PCIe Gen6, NVLink-C2C [46], and GPU Direct [40] significantly reduce the data movement overhead between GPU and other system components. This allows a GPU to process data beyond on-device memory with very fast speed, enabling TBs of analytics even on a single GPU and more with distribution. Meanwhile, GPUs are increasingly affordable and accessible, especially for older generations which are already sufficient for data analytics workloads.

On the software side, the GPU ecosystem has significantly matured. Libraries such as `libcudf` [36] provides high-performance primitives like joins and aggregations that a GPU SQL engine can build upon. The rise of composable data systems [162]—driven by open standards like Substrait [52] for query representation, Apache Arrow and Parquet for columnar data formats—enable greater interoperability. A modern GPU engine can reuse existing components, such as SQL parser and query optimizer, to drastically reduce the complexity of building the system from the ground up.

To demonstrate the feasibility of GPU-native databases, we have built Sirius¹, a prototype open-source SQL engine that uses GPU as the primary execution device and delivers drop-in acceleration across diverse data systems. Sirius integrates seamlessly with existing database systems via the standard Substrait query representation. For example, plugging Sirius into DuckDB causes minimal change to the user-facing interface. Instead of executing the query using DuckDB’s CPU engine, the Substrait

¹<https://github.com/sirius-db/sirius>

plan is routed to Sirius for GPU-native execution. Sirius builds on GPU libraries such as `libcudf` [36], `RMM` [50], and `NCCL` [44], reusing optimized implementations of core relational operators like joins, filters, aggregations, and data shuffle. Thanks to its modular design, Sirius also allows developers to easily switch the operator implementation between these GPU libraries and custom CUDA kernels. Our detailed evaluation shows that Sirius delivers $8.3\times$ and $7.4\times$ better cost efficiency on TPC-H and ClickBench, respectively, when used as a drop-in accelerator for DuckDB [166] (single-node), and up to $12.5\times$ speedup as a drop-in accelerator for Apache Doris [26] (distributed).

The rest of the chapter is organized as follows. Section 6.1 reviews recent software and hardware developments. Section 6.2 describes the design, implementation, and future roadmap of Sirius. Section 6.3 presents the performance evaluation of Sirius. Finally, Section 4.8 concludes the chapter. This chapter is adapted from previously published work [214].

Acknowledgement. This project was co-led by Bobbi Yogatama and me. Bobbi contributed to the primary design of Sirius and its integration with DuckDB, while I contributed to the design of the distributed exchange service and the integration of Sirius with Apache Doris.

6.1 GPU Era for Data Analytics: Why Now

This section discusses recent trends in both hardware and software that pave the way towards scalable GPU SQL analytics.

6.1.1 Hardware Trend

A key barrier to GPU’s wider adoption in data analytics has been GPU’s limited memory capacity, as well as the limited communication bandwidth between GPU and other system components, such as main memory, storage, and network. In a traditional GPU database, data must fit in GPU memory to fully realize acceleration benefits, which severely limits their use cases. Fortunately, these limitations are diminishing with recent hardware advancements, as illustrated in Figure 6.1 and discussed below.

Bigger Memory at Faster Speed. The capacity of GPU device memory has been

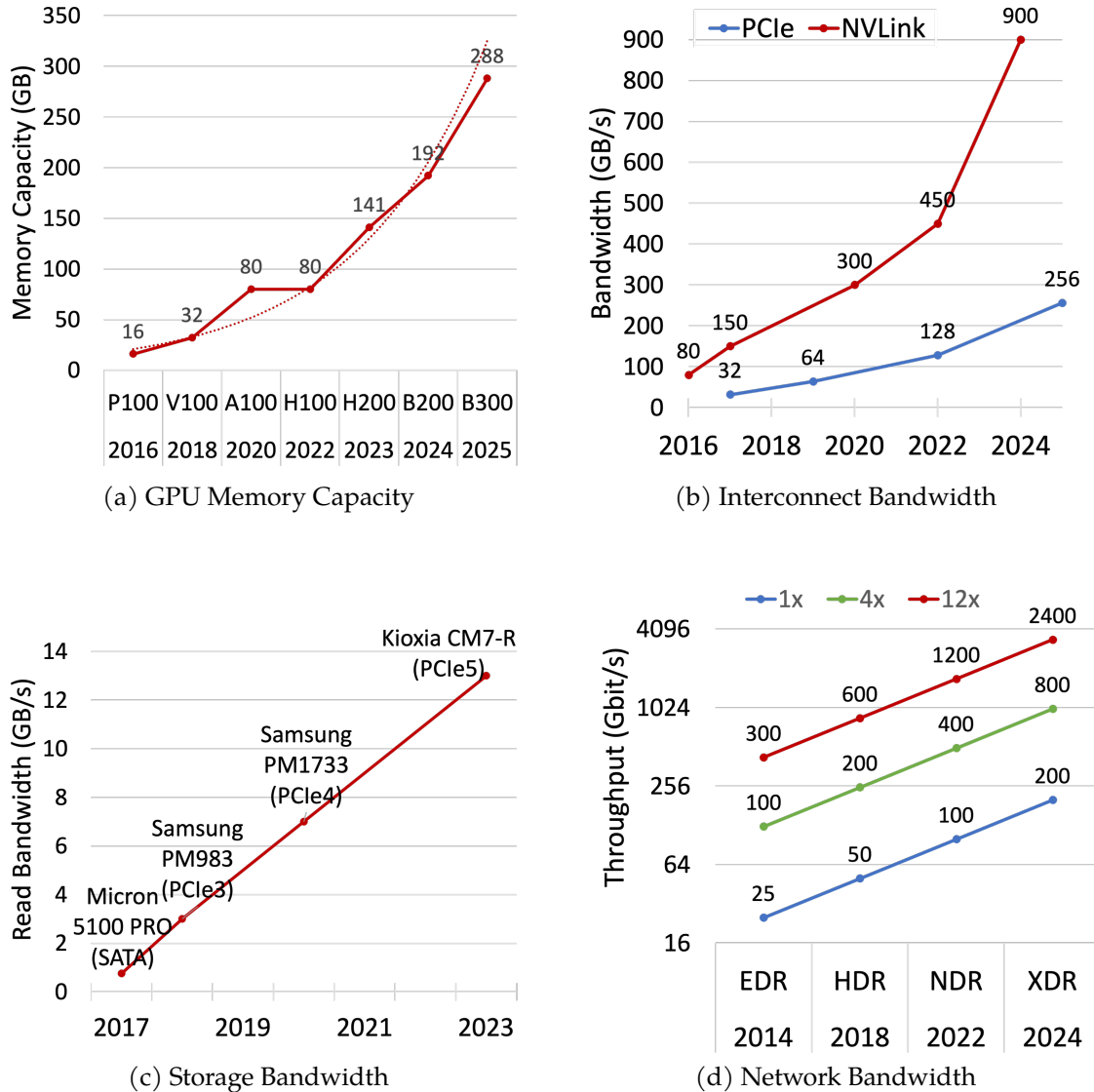


Figure 6.1: Recent hardware trends

growing rapidly. While the largest GPU memory was merely 16 GB ten years ago, a modern NVIDIA B300 Ultra or AMD MI350X has 288 GB device memory, which is sufficient for small to medium-sized analytics workloads.

PCIe, the interconnect between GPU and CPU, has traditionally been a severe performance bottleneck due to its relatively low bandwidth. However, PCIe bandwidth has been doubling every two years with PCIe Gen6 offering 128 GB/s, which is comparable to CPU memory bandwidth. The NVLink-C2C [46] technology takes this even further, enabling 900 GB/s unidirectional bandwidth between CPU and

GPU. In a GH200 superchip, the Hopper GPU can access host memory at more than 400 GB/s, which is more than the maximum bandwidth between the CPU and main memory. These faster interconnects allow a GPU analytics engine to support data larger than the GPU device memory, enabling high-speed access to terabytes of main memory and beyond.

Faster Network and Storage. Besides faster memory speed, modern GPUs also have increasingly better support for network and storage, allowing a GPU engine to go beyond in-memory data processing. Technologies like GPUDirect Storage and GPUDirect RDMA allow GPUs to access data directly from storage or network with minimal CPU involvement at very high speed. Similar to interconnect technology, storage and network bandwidth have also grown rapidly in recent years, and are expected to continue improving (see Figure 6.1c and 6.1d). Moreover, the recent S3 over RDMA technique further enables high-speed object store for GPUs, which can achieve storage access speed at 200 GB/s [189]. With significantly higher computational power, GPUs are better positioned than CPUs to fully utilize these high-bandwidth resources.

Declining GPU Cost. While GPUs of the latest generation remain in high demand with high cost, older generations are becoming increasingly accessible in the cloud at substantially lower cost. For instance, the GH200 instance shown in Table 6.1 has an on-demand price of \$3.2/hour in Lambda Labs, which is cheaper than many high-end CPU instances. The on-demand H100 prices in many GPU cloud providers has seen significant price drop from \$8/hour in March 2023 to around \$3/hour in 2025 [45, 9]. Even in major cloud provider such as AWS, in 2025 alone, the A100, H100, and H200 on-demand price have dropped by 33%, 44%, and 25% respectively [218].

6.1.2 Software Trend

Another barrier in building GPU DBMSes is the engineering cost and expertise required to build the software stack from the ground up. Fortunately, the barrier has been significantly lowered with the following two software trends.

Composable Data Systems. Modern open-source data systems are embracing composability [162] (e.g. Velox [161]), where different system components can be independently developed and interoperate in a data system. A new database can reuse existing components such as the language frontend, query optimizer, or execution

engine instead of reinventing the wheel. A crucial component in these systems is Substrait [52], a unified IR for physical and logical representations of query plans. The trend of composable data systems significantly lowers the barrier to build a GPU database—only the execution engine needs to be developed from the ground up tailored for GPU but other system components could be reused.

Maturing GPU Libraries. Open-source GPU libraries optimized for data processing have emerged and are rapidly reaching maturity in recent years. For example, libcudf [36] provides GPU-efficient implementation for relational operators such as joins, aggregations, and sorting. RMM [50] complements this by offering a high-performance GPU memory allocator, serving as a foundation for GPU-centric buffer managers. On the distributed side, the NCCL [44] provides high-speed communication between GPU devices over various interconnect, such as PCIe, NVLink, and Ethernet/RDMA.

6.2 Sirius Design and Architecture

Motivated by the hardware and software trends, we have designed Sirius, a GPU-native SQL engine that provides drop-in acceleration across a variety of existing data systems. In this section, we will discuss the design overview and system architecture of Sirius.

6.2.1 Design Overview

Sirius was designed with the following two key design principles:

GPU-Native Execution. Recent hardware trends show that the overhead of data movement between GPUs and other system components, especially CPUs, is diminishing. As the GPU memory barrier is breaking down, GPU-only query execution is becoming increasingly competitive with fine-tuned CPU–GPU co-execution.

Taking this into account, we design Sirius as a *GPU-native engine*: it treats GPUs as the primary execution engine, aiming to run the entire query plan, from scan to result, on the GPU. Sirius differs from systems that retrofit GPU acceleration onto traditional CPU-optimized engines while maintaining backward compatibility [51, 133], or hybrid systems that split execution between CPU and GPU [215, 124, 213, 89]. Instead, it builds its execution model, buffer management, and exchange mechanisms

entirely around full GPU residency. The CPU serves only as a fallback path when certain features are not supported on GPUs. This architecture can reduce the design complexity compared to a retrofitted or heterogeneous design, thereby improving portability and performance.

We believe GPU-native design is a promising direction. History shows that when a major infrastructure shift occurs, such as the rise of the cloud, systems built natively for the new paradigm (e.g., Snowflake) ultimately surpass retrofits of prior-generation architectures. Similarly, as GPUs emerge as a central computational platform for data analytics, GPU-native systems like Sirius have the potential to outperform hybrid or retrofitted approaches.

Drop-In Acceleration. Modern software stacks are gradually moving toward composable architectures [162], where components such as query optimizers, storage format, and query execution engine are decoupled. Sirius embraces this modular design to significantly reduce the development cost and the barrier of adoption.

By consuming a universal query plan format (e.g., Substrait [52]) and connects it with existing GPU-accelerated libraries (e.g., libcudf), Sirius can serve as a *drop-in accelerator* across a variety of data systems with minimal or no modification to the host systems. When using Sirius, the users can benefit from GPU acceleration without having to modify their user interface or migrate to a different DBMS.

6.2.2 System Architecture

In this Section, we will discuss the system architecture of Sirius as shown in Figure 6.2.

6.2.2.1 Host Database Layer

This layer includes the query parser and optimizer of external data systems that Sirius accelerates. These host systems generate query plans in a standardized format such as Substrait which Sirius can consume. Currently, Sirius supports DuckDB and Apache Doris as external hosts, and more will be supported in the future.

For distributed query execution, Sirius leverages the host database’s coordinator to manage the control plane. This includes identifying active nodes via heartbeat, scheduling plan fragments across nodes, determining partitioning schemes, and maintaining global metadata. For data exchange, however, it is handled by Sirius built-in exchange service layer explained in Section 6.2.2.4.

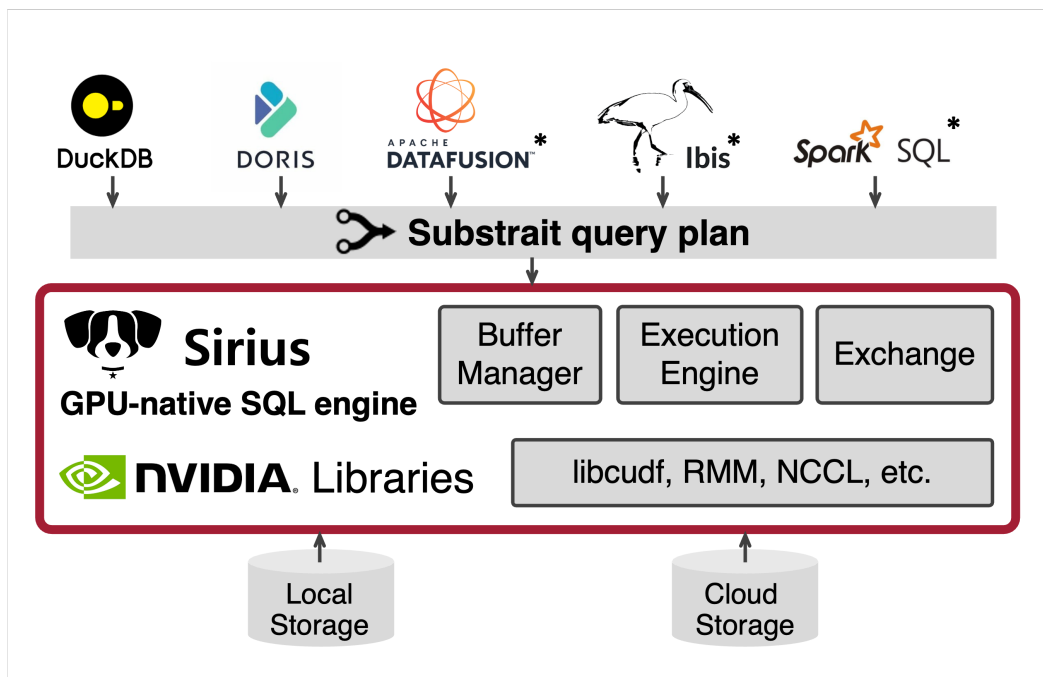


Figure 6.2: Sirius Architecture

DuckDB supports Substrait and provides an extension framework, enabling Sirius to be integrated with *zero modification* to DuckDB’s codebase. Apache Doris, however, lacks an extension framework and does not natively export Substrait plans; therefore, we introduce minimal modifications to its codebase to translate its internal query plans into the Substrait format.

6.2.2.2 Query Execution Engine

The query execution engine in Sirius executes the Substrait plan generated by the optimizer. Sirius adopts a **pipeline execution model**, in which the query plan is divided into pipelines. Each task, at the granularity of a pipeline, is enqueued into a global task queue. Idle CPU threads pull tasks from this queue and execute the pipelines by invoking each operator. This model is widely-used in modern data systems such as DuckDB, Hyper, and Velox.

Within each pipeline, Sirius adopts the **push-based execution model**. The query executor acts as a coordinator, maintaining the state of query execution (e.g., operator’s input and output). Instead of having operators *pull* data from their predecessors—as done in systems like Velox—the executor *pushes* data into operators. This design keeps the operators stateless, simplifying their implementation. This model is

adopted by modern systems such as DuckDB.

Most physical operators in Sirius are implemented using the `libcudf` library, with a few exceptions for specialized functions such as predicate pushdown, and materialization. Thanks to its modular design, Sirius allows developers to easily switch the operator implementation between `libcudf` and custom CUDA kernels. Sirius also includes a graceful fallback mechanism to the host database systems in the case of an error or missing features in Sirius.

6.2.2.3 Buffer Manager

The buffer manager is responsible for managing memory in Sirius. It divides the memory into the following two separate regions:

Data Caching. This region is used to store cached data in Sirius either in device memory or pinned host memory. To minimize the overhead of dynamic memory allocation during query execution, the caching region is pre-allocated in advance.

Data Processing. This region is in device memory which stores intermediate results during query execution (e.g., hash tables, intermediate results, etc.). Sirius uses the RMM [50] (RAPIDS Memory Manager) pool allocator to manage this region efficiently.

Moreover, the buffer manager is responsible for converting between different columnar formats used throughout the system: the internal Sirius columnar format, the `libcudf` columnar format, and the columnar format used by the host database. Both Sirius and `libcudf` derive their columnar format from Apache Arrow, which allows for zero-copy conversion via pointer passing. The only exception is when converting row indices between Sirius and `libcudf`, as they use different types; Sirius uses `uint64_t` whereas `libcudf` uses `int32_t`. Conversion between Sirius and the host database format also requires deep copies, but this occurs only during the cold run (initial data load) and when returning results to the user.

Currently, Sirius relies on the host database to read data from disk. Once data is read, the buffer manager automatically caches it into the pre-allocated caching region for future reuse. More discussion on disk support and spilling will be discussed in Section 7.2.2.

6.2.2.4 Exchange Service Layer

The exchange service layer orchestrates distributed query execution across multiple nodes. In single-node deployments, this layer can be bypassed entirely. However, in multi-node settings, it plays a critical role in coordinating data exchange between nodes.

In Sirius, exchange is modeled as dedicated physical operators. Sirius supports common exchange patterns such as broadcast, shuffle, merge, and multi-cast — all implemented using NCCL primitives.

In distributed execution, the query plan is divided into multiple fragments, each of which is executed locally on a participating node. Data exchange occurs between fragments — after one fragment completes, its output is transmitted across the compute nodes and consumed as input by the subsequent fragments. Sirius internally maintains a runtime registry of exchanged intermediate data as temporary tables. These temporary tables are automatically deregistered once the corresponding fragments finish execution.

6.2.3 Query Lifecycle in Sirius

In this section, we will describe the end-to-end query lifecycle when Sirius is used as a drop-in replacement for DuckDB (in single-node settings) and Apache Doris (in distributed environments).

Single-Node. Users begin by issuing SQL queries through the DuckDB CLI or Python API. DuckDB then parses and optimizes the query, producing a DuckDB-specific logical plan. In the vanilla DuckDB workflow, DuckDB-native execution engine will execute this plan and return the result to the user. When accelerated by Sirius, however, DuckDB delegates the execution to Sirius by passing the query plan to its GPU-accelerated execution engine (Section 6.2.2.2) and returns the result to the user.

Distributed. Figure 6.3 illustrates the query lifecycle for vanilla Apache Doris (Figure 6.3a) and GPU-accelerated Doris powered by Sirius (Figure 6.3b). In both cases, users submit SQL queries via the Doris SQL interface. The Doris coordinator generates a distributed query plan and dispatches plan fragments to Doris compute nodes. In standard Doris workflow, the compute nodes will execute the fragments locally, exchange intermediate data via Doris' native exchange service, and return the results to the coordinator.

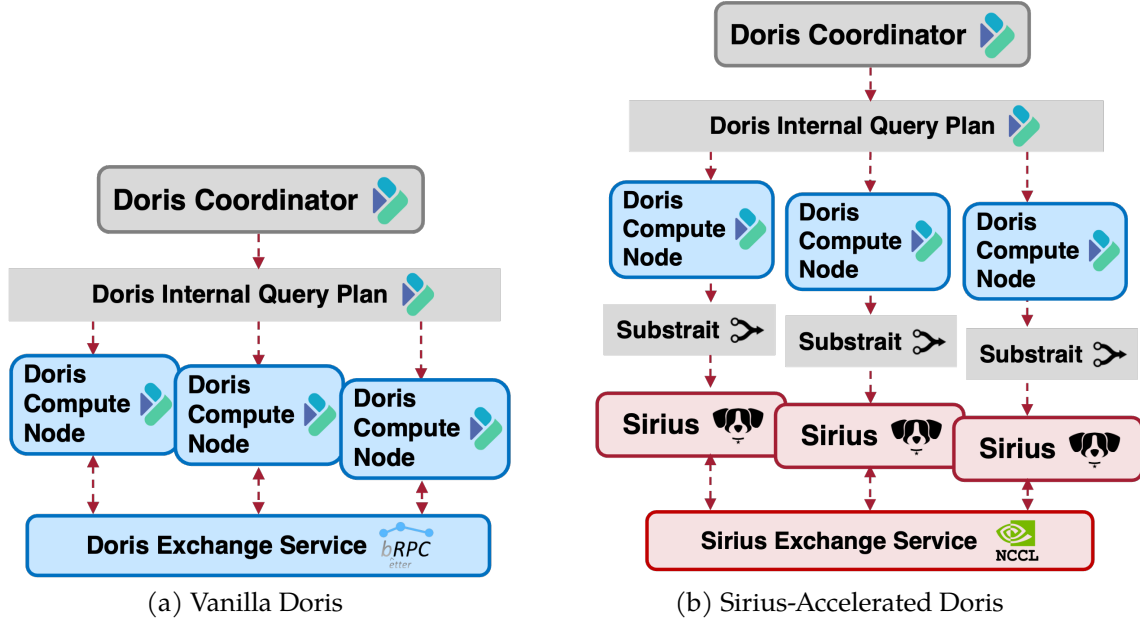


Figure 6.3: Distributed Query Lifecycle in Vanilla Doris and Sirius-Accelerated Doris.

When accelerated by Sirius, instead of executing the plan fragments directly, each compute node translates its assigned fragment into Subtrait and forwards it to the local Sirius execution engine (Section 6.2.2.2). To exchange intermediate results across nodes, Sirius uses its exchange service layer described in Section 6.2.2.4. Once execution completes, Sirius returns the results to Doris compute node, which then forward the results to the coordinator.

6.3 Evaluation

In this section, we will evaluate the performance of Sirius as a drop-in accelerator for DuckDB (single-node) and Apache Doris (distributed).

6.3.1 Experiment Setup

Hardware Configurations. We ran Sirius with two different setups:

- **NVIDIA GH200:** This machine features NVIDIA Grace CPU and Hopper GPU connected through NVLink C2C with 900 GB/s bidirectional bandwidth. The Hopper GPU has a 92 GB of HBM3 memory with a read/write bandwidth of

3 TB/s. The Grace CPU has 72 Neoverse Armv9 cores with 480 GB of LPDDR5X memory.

- **4 × NVIDIA A100:** This cluster consists of 4 identical nodes, where each node has one NVIDIA A100 GPU and 64-cores Intel Xeon Gold 6526Y CPUs. The A100 GPU has 40 GB of HBM3 memory with a read/write bandwidth of 1550 GB/s. The GPU and CPU are connected via PCIe4 with 25.6 GB/s bidirectional bandwidth. Every node support InfiniBand 4x NDR (Next-Generation Rate), providing network bandwidth of up to 400 Gbps.

Benchmark. We use the *TPC-H* [54] benchmark and *ClickBench* [35] in our experiment, which has been widely used both in academia and industry.

Measurement. We dedicate 50% of each GPU memory for data caching, and the other half for data processing. *The numbers reported are the hot runs when the data is already cached in the GPU memory.* Throughout the experiment, we will evaluate the performance of the following query engines:

- **DuckDB:** DuckDB [166] is a single node open-source embedded CPU-based analytical database.
- **ClickHouse:** ClickHouse [176] is a distributed open-source CPU-based online analytical database.
- **Umbra:** Umbra [153] is a single-node RDBMS designed to process data at line-speed for both in-memory or storage resident data.
- **Apache Doris:** Doris [26] is an open source high-performance and real-time distributed CPU-based data warehouse.
- **Sirius:** Our GPU-native SQL engine that serves as drop-in accelerator for DuckDB (single node) and Apache Doris (distributed).

6.3.2 TPC-H Single-Node Performance

In this Section, we evaluate Sirius performance running on a single node as a drop-in accelerator for DuckDB. We ran 22 TPC-H queries with a scale factor of 100 and compare the performance against DuckDB and ClickHouse. To ensure cost-normality, we ran DuckDB and ClickHouse on a CPU instance (c8i.8xlarge@AWS) that has the

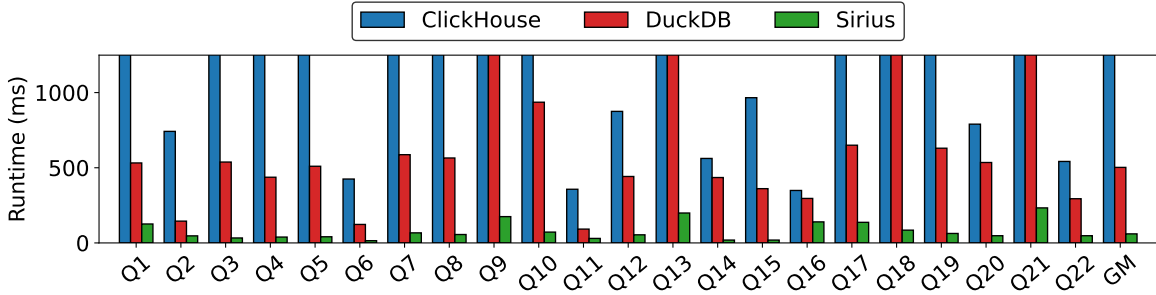


Figure 6.4: TPC-H End-to-end Query Performance on a Single Node.

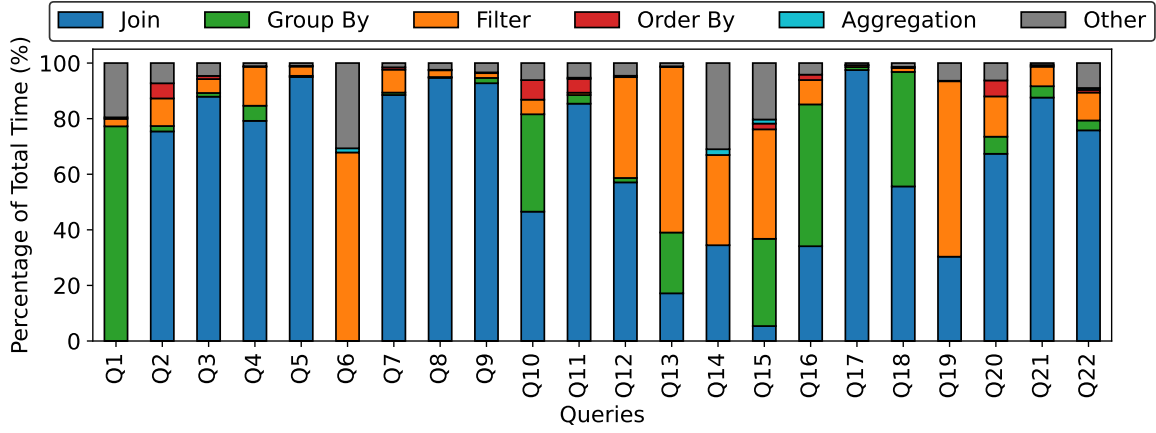


Figure 6.5: TPC-H Performance Breakdown.

same hourly rental cost (\$1.5/h) as the GPU instance (GH200@LambdaLabs) used to run Sirius.

End-to-End Performance Comparison. Figure 6.4 shows the end-to-end performance of various query engines on the TPC-H benchmark. Compared to DuckDB, Sirius achieves a $8.3\times$ speedup with the same hardware rental cost. Sirius leverages DuckDB’s optimized logical plans but replaces its backend with GPUs, demonstrating the significant performance advantage of GPU acceleration.

Since ClickHouse does not support correlated subqueries [176], we rewrite queries containing subquery correlation for compatibility. Compared to ClickHouse, Sirius is $38\times$ faster with the same hardware rental cost. In ClickHouse, the default parallel hash join algorithm triggers out-of-memory errors for Queries 5, 9, and 21. Therefore, we use merge join for these queries to ensure successful execution. We observed that ClickHouse is not optimized for join-heavy workloads, leading to suboptimal performance in many TPC-H queries (e.g., Q2, Q5, Q9, Q10, Q21, etc).

Table 6.2: TPC-H End-to-End Query Performance in the Distributed Setting.

	Doris	ClickHouse	Sirius	Breakdown in Sirius		
				Compute	Exchange	Other
Q1	1193	393	97	33	3	61
Q3	838	12785	341	43	233	75
Q6	199	294	84	36	1	47

Performance Breakdown. Figure 6.5 presents a performance breakdown across different TPC-H queries in Sirius. The results indicate that join operations dominate query execution time in most cases, particularly in join-heavy queries (Q2–Q5, Q7–Q8, Q20–Q22).

Group-by also account for a substantial portion in several queries, notably for Q1, Q10, Q16, and Q18. This overhead is especially visible in queries involving group-by on string keys (Q10, Q18) or with a small number of distinct groups (Q1). For string keys, libcudf defaults to a sort-based group-by strategy, which is less performant than hash-based group-by. For group-by operations with a small number of groups, GPU suffers from memory contention.

Filter operation contributes to a majority of the query execution time in filter-heavy queries such as Q6 and Q19. They also contribute significantly in Q13, where a complex string-matching expression with very low selectivity is used. In contrast, aggregation and order-by operations do not dominate the end-to-end execution time for any of the TPC-H queries. This is because the input size for these operations is typically very small.

6.3.3 TPC-H Distributed Performance

In this section, we ran a subset of TPC-H queries with a scale factor of 100 (i.e., Q1, Q3, and Q6) that are currently supported by the distributed mode of Sirius and compare the performance of Sirius against Doris and ClickHouse using a cluster of $4 \times$ A100 GPUs.

End-to-End Performance Comparison. Table 6.2 presents the performance comparison of different query engines in a distributed environment. Compared to Doris, Sirius achieves a speedup of $12.5\times$, $2.5\times$, and $2.4\times$ in Q1, Q3, and Q6, respectively.

Sirius leverages the exact same distributed query plan from Doris, but replaces its execution engine with GPUs, demonstrating the performance advantage of GPUs in distributed query execution.

Sirius outperforms ClickHouse by $4\times$, $37.5\times$, and $3.5\times$ in Q1, Q3, and Q6, respectively. When joins are not involved (Q1, Q6), ClickHouse is able to outperform Doris. However, its performance degrades significantly when distributed joins are involved (i.e., Q3), resulting in substantial slowdown compared to Doris and Sirius.

Performance Breakdown. We further present a breakdown of Sirius’ performance in Table 6.2, specifically, into the categories of local GPU computation, data exchange, and the rest portion during execution (e.g., query optimization, plan dispatching, etc.).

We noticed that GPU execution is not the primary performance bottleneck in the current implementation of distributed Sirius, indicating ample opportunities for further optimizations. For example, in Q1 and Q6, a significant portion of query execution time is spent in Doris’ query optimizer and coordinator, which run on the CPU. This overhead does not scale with the data size and is expected to be less significant for larger dataset. In Q3, data exchange is the main bottleneck as the Doris’ distributed query plan shuffles both the orders and `lineitem` tables. We have identified several opportunities to further improve the shuffle performance in Sirius but leave a deeper exploration to future work.

6.3.4 ClickBench Performance

In this section, we evaluate Sirius as a drop-in accelerator for DuckDB and compare its performance against the top two systems on the ClickBench [35] leaderboard (Umbra and DuckDB²). We ran Sirius on a GH200 instance from Lambda Labs (\$1.5/hour). The competing systems ran on CPU instances — AWS c8g.metal-48xl (\$7.6/hour) and AWS c7a.metal-48xl (\$9.8/hour). Because the three systems use differently priced hardware, we report the normalized total cost of ownership (TCO) relative to Sirius.

Figure 6.6 shows the normalized TCO of Umbra and DuckDB relative to Sirius (fixed at 1). Sirius achieves the lowest TCO on all queries, benefiting from highly efficient GPU execution. In particular, queries q4, q5, and q18 exhibit substantial performance gains on common operators such as filtering, projection, and aggregation.

²As of December 5, 2025; results are subject to change

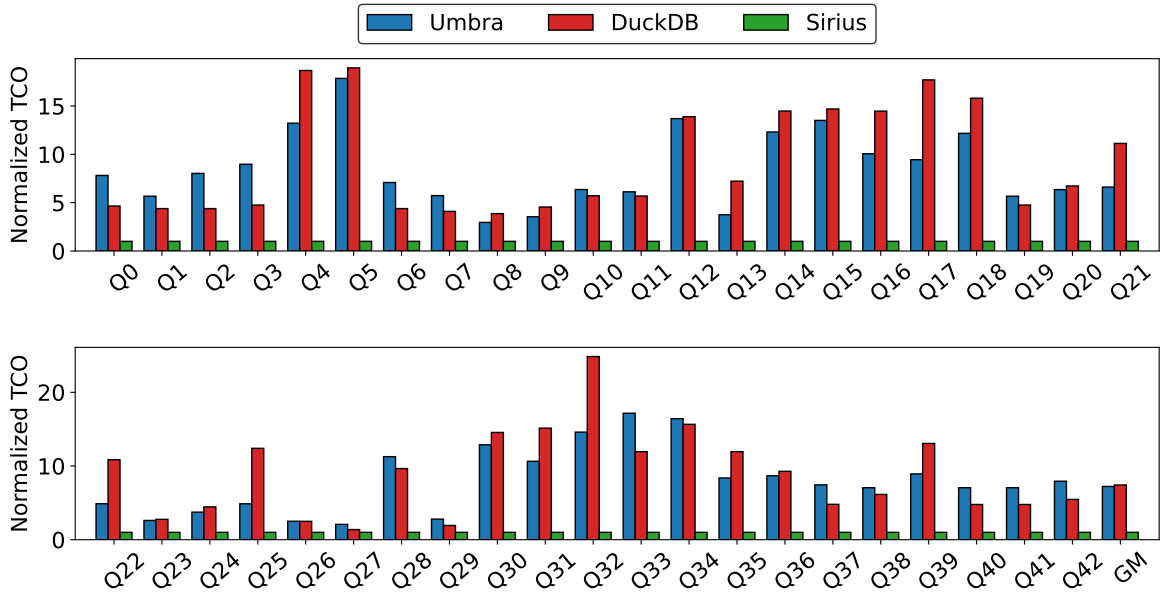


Figure 6.6: Normalized Total Cost of Ownership (TCO) Relative to Sirius in Click-Bench.

Some queries, however, highlight optimization opportunities. For example, q23 is bottlenecked by the contains operation on string columns; q24 and q26 by top-N operators; and q27 by aggregations over very large inputs. We plan to improve these operators in future releases of Sirius.

Overall, Sirius delivers at least a 7.4 $\times$ improvement in TCO when running Click-Bench as a drop-in accelerator for DuckDB. It achieves the lowest relative runtime on the ClickBench leaderboard (11% faster than Umbra, the top performer), while running on significantly cheaper hardware (\$1.5/hour vs. \$9.8/hour), setting a new performance record for the benchmark.

6.4 Conclusion

In this chapter, we argue that the GPU era for data analytics has arrived and supports this claim by examining recent trends in hardware and software development. Motivated by these trends, we introduce Sirius, an open-source GPU-native SQL engine that provides drop-in acceleration across a variety of existing data systems. Our evaluation on TPC-H and ClickBench shows that Sirius delivers 7.4–8.3 $\times$ better TCO than state-of-the-art systems in a single-node setting, and up to 12.5 $\times$ speedup

in distributed setting.

Chapter 7

Future Directions

In this chapter, we discuss potential future research directions that build upon the work presented in the previous chapters, spanning both the algorithmic stack (Section 7.1) and the infrastructure stack (Section 7.2).

7.1 Future Directions in the Algorithmic Stack

This section discusses potential future research directions building upon predicate transfer (Section 7.1.1), as well as additional optimization opportunities beyond acyclic queries (Section 7.1.2).

7.1.1 Extensions and Optimizations of Predicate Transfer

As presented in Chapters 2 and 3, predicate transfer is a novel and efficient pre-filtering framework for join queries. We identify several potential research questions that build upon this work, as outlined below.

Cost-Based Optimization and Adaptivity. The current predicate transfer framework relies on empirical heuristics, such as `Small2Large` (Section 2.2) and `LargestRoot` (Section 2.3) to produce predicate transfer schedules. Ideally, a cost model should be developed and used to explore the space of possible predicate transfer schedules. The cost of a transfer schedule should account for both the computational cost of Bloom filter operations, as well as the network cost incurred when execution occurs in a distributed environment. In addition, the model should consider the expected filtering selectivity on input join tables.

Overall, such a cost model naturally leads to a multi-objective optimization problem, in which the goal is to jointly optimize the effectiveness and efficiency of pre-filtering. For instance, cost-based algorithms could identify the fastest predicate transfer schedule that achieves a desired level of filtering effectiveness or, alternatively, the most effective predicate transfer schedule that can be executed within a given time budget. The cost model presented in Section 3.2 takes an initial step in this direction by determining the appropriate distributed execution strategy for each individual transfer step within a fixed predicate transfer schedule.

Based on the developed cost model, finer-grained adaptivity can be achieved at the level of individual transfer steps. Section 3.3 presents a heuristic-based approach for pruning transfer steps that provide no filtering benefit. This approach could be further refined by redefining the benefit of a transfer step in terms of the filtering selectivity it contributes relative to its execution cost — for example, determining whether it is worthwhile to incur an additional scan to pre-filter a particular input table. Notably, the decision to prune a transfer step may also affect both the cost and the potential contribution of subsequent transfer steps. By exploring the global pruning space, it may be possible to derive an optimal sequence of pruning decisions.

Out-of-Core Execution. The current design of predicate transfer assumes that input join tables, as well as the Bloom filters constructed from them, fit entirely in main memory. However, in large-scale SQL analytics, input tables may significantly exceed available memory, requiring data to spill to disk during execution. Supporting out-of-core execution therefore introduces several challenges for predicate transfer.

First, it remains an open question how to efficiently perform Bloom filter construction and probing when the input table does not fit in memory, or even when the Bloom filter itself exceeds memory capacity. Naively spilling Bloom filters or input data to disk can negate the performance benefits of pre-filtering. One potential direction is to leverage radix partitioning on the join keys, enabling the construction and probing of Bloom filters one partition at a time.

Second, out-of-core execution raises important decisions about materialization strategies during Bloom filter probing. For example, if the input table is larger than memory but the resulting pre-filtered table fits in memory, early materialization may be preferable so that subsequent scans can be served from memory. Conversely, if the input table is small, late materialization, which produces a selection vector and deferring materialization until the end of predicate transfer, may be more efficient.

Finally, out-of-core execution interacts closely with the cost model discussed earlier and highlights the potential benefits of predicate transfer. By aggressively reducing input sizes early in the query plan, predicate transfer can significantly shrink intermediate results and prevent or reduce spilling in downstream operators, yielding substantial performance gains in memory-constrained environments. Although predicate transfer may incur additional disk I/O when pre-filtered tables themselves do not fit in memory, this overhead is often amortized by the reduced cost of subsequent joins and scans. Accurately capturing this trade-off, including the cost of additional disk accesses in the aforementioned cost model, is essential for selecting predicate transfer schedules that maximize overall performance.

Coarse-Grained Predicate Transfer. The predicate transfer algorithm discussed so far performs pre-filtering at the tuple level. We believe that coarser-grained predicate transfer could yield additional performance gains by reducing operational costs. Coarse-grained filters, such as min-max filters, are widely used [1, 7, 58, 152, 136, 29, 12, 180], as they allow skipping entire row groups whose values fall outside the min-max range. Importantly, coarse-grained filters not only improve the computational efficiency of pre-filtering but also reduce I/O during table scans. A key research question is how to efficiently combine coarse-grained filters with tuple-level filters, like Bloom filters, in the predicate transfer framework.

One possible design is to introduce a separate predicate transfer phase using min-max filters instead of Bloom filters to prune unnecessary row groups. These operations are inexpensive because they operate only on small metadata. A more effective approach is to interleave coarse-grained and Bloom filter predicate transfers. For example, after executing a Bloom filter transfer step, the resulting filtered tables can provide tighter min-max bounds, which can then be used to further restrict the relevant ranges of other join tables, such that more row groups can be skipped in the subsequent table scans. Furthermore, the cost model discussed previously can be extended to account for this mixed approach, enabling a more thorough exploration of the predicate transfer space and potentially identifying more efficient schedules.

Predicate Transfer on Cyclic Queries. The theoretical foundation of predicate transfer, the Yannakakis algorithm, provides full table-reduction guarantees only for acyclic join queries. While predicate transfer can yield performance improvements on cyclic queries as well (e.g., TPC-H Q5), these benefits are not supported by the theoretical guarantees established for acyclic queries. As a result, the current predicate trans-

fer algorithm on cyclic queries may fail to achieve maximal pre-filtering, leaving additional opportunities for further table reduction.

Designing efficient pre-filtering techniques with strong theoretical guarantees for cyclic join queries therefore remains an open research direction. For example, in cycle join queries such as triangle or rectangle queries, predicates could be propagated in both clockwise and counterclockwise directions such that each table receives filtering information from every other table. In contrast, the tree-structured propagation used by the Yannakakis algorithm cannot achieve such complete information flow. Moreover, predicate transfer could potentially be combined with modern worst-case optimal join techniques (Section 7.1.2) to further improve efficiency on cyclic queries.

Exploiting Additional Filtering Power. Broadly speaking, predicate transfer propagates predicates derived from size-reducing operators to pre-filter join inputs. In the current framework, filters and equi-joins are the primary sources of such predicates, as they are both common and naturally supported by Bloom filter-based techniques. However, other query operators can also significantly reduce input sizes and thus represent additional opportunities for predicate transfer.

One such class of operators includes non-equi (θ) joins. While Bloom filters are well suited for equality predicates, more general join conditions may benefit from alternative approximate filtering structures, such as range-based summaries or sketch-based filters, to enable efficient pre-filtering under arbitrary predicates. Designing lightweight and effective filtering structures for these cases remains an open research direction.

Similar to joins, set operations such as intersect and except/minus can also reduce the size of intermediate results. For example, intersect behaves like an inner join, retaining only the rows that are common across all input tables. Except or minus behaves like an anti-join, removing tuples present in the subtracted table from the original input.

Another opportunity arises from exploiting limit and order-by clauses. For example, when the sort keys originate from the same base table, the limit condition can be applied after an initial predicate transfer phase to further reduce the size of the pre-filtered table. A subsequent predicate transfer phase can then be initiated from this reduced table, propagating additional filtering power to other join inputs. Due to false positives introduced during predicate transfer, the subsequent join phase may produce fewer rows than required by the limit clause. This can be addressed

by iteratively applying predicate transfer and join phases on successive windows of rows according to the order-by clause, continuing until the required number of output rows are produced.

Partitioning-Aware Distributed Predicate Transfer. The current distributed predicate transfer technique does not assume any particular partitioning scheme for the input join tables (Section 3.1). In practice, however, tables may be partitioned in various ways beyond simple random partitioning: for example, they may be hash-partitioned or range-partitioned on specific columns, or fully replicated on all or a subset of nodes. Moreover, the size of each partition may vary across nodes. Exploiting this underlying partitioning information offers an opportunity to further optimize distributed predicate transfer, while enabling more accurate cost estimations (Section 3.2).

Predicate Transfer with Pushdown Execution in Cloud Databases. As discussed in Section 5.3, both table-local filter predicates and Bloom filter operations are well suited for pushdown execution, and support for these operators already exists in modern cloud database systems [210, 47]. In storage-disaggregated architectures, where compute and storage are decoupled, this creates an opportunity to push not only simple filters but also the predicate transfer phase closer to the data.

Concretely, table-local filters as well as Bloom filter construction and probing can be executed within the storage or near-storage compute layer. While table-local predicates reduce input sizes independently, pushdown with predicate transfer goes further by applying cross-table pre-filtering, producing results that are often substantially smaller before being sent to the compute layer. Compared to traditional predicate pushdown, this approach further reduces the volume of data moved across the network.

Since table-local filtering and Bloom filter operations are lightweight and primarily scan-based, offloading these phases is unlikely to overwhelm storage-side compute resources. Once the predicate transfer phase completes at the storage layer, the heavier join processing is executed normally in the compute layer, benefiting from the already reduced input sizes.

Practical Implementation in Existing Systems. Predicate transfer was initially implemented in FlexPushdownDB [208], a research OLAP database prototype. Later efforts attempted to integrate predicate transfer into DuckDB [222, 165], but these changes were not merged due to their large scope and potential impact on other sys-

tem modules. A more practical approach for modern OLAP databases is to implement predicate transfer as an extension, a mechanism supported by many open-source DBMSs, including DuckDB [39] and PostgreSQL [49]. Using extensions, predicate transfer can be added without modifying the core engine, enabling experiments, rapid deployment, and easier maintenance, while still providing the benefits of pre-filtering for query execution.

7.1.2 Accelerate Cyclic Queries with Per-Split Query Plans

As mentioned in Section 1.1.3, while the seminal Yannakakis algorithm and predicate transfer provide strong guarantees for acyclic queries, minimizing intermediate join results for cyclic queries remains an open challenge, both theoretically and in practical systems. While worst-case optimal join (WCOJ) algorithms [196, 154, 175] guarantee runtimes bounded (i.e., the AGM bound) by the maximum possible output size, they have seen limited adoption in real-world systems, where decades of engineering have made binary joins highly optimized and dominant.

In contrast to WCOJ, *partition-based strategies* [130, 93, 94, 119] address data skew explicitly by separating heavy and light values in join-key domains and evaluating them using different binary-join subplans, still achieving the same strong runtime guarantees (i.e., the AGM bound). Building on this line of work, one key future research direction is to partition-based strategies practical. We propose our initial high level design, *SplitJoin*, which introduces *split* as a first-class operator in query processing. Intuitively, the split operator partitions an input table R into a light table (R_L) and a heavy table (R_H), which are then processed with potentially different query plans.

In our design, the SplitJoin framework extends the query optimizer with an additional component, termed the *split planner*. The split planner operates in two phases: a *split phase* and a *join phase*. By integrating this component into the optimizer, query planning becomes aware not only of join orders and access paths, but also of alternative split strategies and their impact on execution.

During the split phase, a basic strategy is to partition an unsplit relation on one of its join attributes into a heavy and a light portion, thereby transforming the original database instance into two subinstances. This process is applied recursively until all relations have been considered. In the subsequent join phase, the optimizer generates different query plans for each resulting subinstance.

In practice, several design choices and optimization heuristics are required to make the framework effective. For example, when two relations join on an attribute, they can be split simultaneously on that attribute using the same threshold, since splitting one relation naturally induces a compatible partitioning of the other. Additional heuristics include selecting appropriate split thresholds and determining which attribute sets to split on; a detailed discussion of these choices can be found in [115].

In the join phase, for each database subinstance produced by the split phase, we follow a key guiding principle: *prioritize joins on light attributes whenever possible*. Accordingly, after selecting an initial relation, the optimizer performs as many light joins as possible. If no further light joins are available, the process restarts from a different relation and repeats the same strategy. When two intermediate results become connected, they are merged. The process continues until a single intermediate result remains, at which point the query output is produced. Our analysis in [115] provides formal proofs showing that this join-ordering heuristic satisfies the AGM bound.

Our preliminary experimental results demonstrate a substantial performance advantage of SplitJoin on representative social network workloads. When integrated into DuckDB [166], SplitJoin successfully completes 14 additional queries that the original DuckDB optimizer fails to finish due to timeouts or out-of-memory errors (43 vs. 29). For the queries that can finish in both cases, SplitJoin reduces runtime by an average of $2.1\times$ (up to $13.6\times$) and decreases the size of the largest intermediate results by an average of $7.9\times$ (up to $74\times$). When integrated into Umbra [153], SplitJoin completes 10 additional queries that the native optimizer cannot finish (45 vs. 35). For queries that complete in both cases, SplitJoin achieves an average speedup of $1.3\times$ (up to $6.1\times$) and reduces the maximum intermediate result size by $1.2\times$ on average (up to $2.1\times$). A more detailed and comprehensive experimental evaluation is provided in [115].

We believe that this first step toward split-based strategies makes such optimizations more practical, opening new opportunities for adaptive cost models and broader integration into modern query engines.

7.2 Future Directions in the Infrastructure Stack

In this section, we outline potential future research directions for innovations in the infrastructure stack, spanning both cloud-native OLAP databases (Section 7.2.1) and GPU-native query engines (Section 7.2.2).

7.2.1 Future Optimizations for Cloud-Native OLAP Databases

Multi-Cloud and Multi-Data-Source Databases. Most existing disaggregated database architectures are designed for deployment within a single cloud environment. A natural next step is to extend these architectures to support multi-cloud settings, including deployments spanning multiple public clouds, hybrid public-private clouds, or even geographically distributed data centers. In such environments, data may be physically located across different administrative domains, where cross-cloud or cross-data-center communication incurs significantly higher latency and bandwidth costs. Efficient query execution in these settings requires new optimization strategies that explicitly account for heterogeneous network characteristics, data locality, and cloud-specific pricing models when deciding where to execute operators and how to move data.

Another closely related direction is supporting multi-data-source query processing, where a single query may access data from diverse sources such as object or file storage systems, relational databases, data warehouses, and specialized analytics engines. Different data sources expose different execution capabilities and have varying performance characteristics and data access costs. Effectively optimizing such queries requires the query optimizer to reason about which operators should be pushed down to each data source, which data should be materialized or transferred, and how intermediate results should be combined across systems. Developing cost models and execution frameworks that jointly consider data-source heterogeneity, pushdown capabilities, and data movement costs remains an important open research problem.

Memory Disaggregation. Recent systems like [81] extend storage disaggregation by further decoupling memory into a shared remote pool that can be provisioned and scaled independently from compute, improving resource utilization and reducing memory over-provisioning. Advances in high-speed interconnects, such as RDMA and emerging CXL fabrics, can help mitigate the latency of remote memory access.

Memory disaggregation introduces new optimization opportunities for query processing, as remote memory access lies on the critical execution path and directly impacts operator performance and buffer management. Query engines must carefully decide where to place base data, intermediate results, and auxiliary structures (e.g., hash tables and indexes) across local memory, remote memory, and storage. Similar to computation pushdown in storage-disaggregated systems, disaggregated memory also enables near-memory computation. Fully exploiting these capabilities across the combined storage and memory tiers remains an open research problem.

Toward a Unified Near-Data Execution Layer. The pushdown layer discussed so far primarily targets relational operators, such as filter predicates. In practice, however, a wider range of database functionalities can be incorporated into this layer and executed close to the data. Examples include lakehouse metadata management [63, 28], HTAP-oriented delta merging and version reconciliation [148], query optimization [52], materialized view maintenance, etc. Exposing these capabilities through a unified near-data execution interface enables shared optimization, reduces data movement, and fosters tighter integration between query processing and data management in disaggregated architectures.

7.2.2 Toward Practical and Scalable GPU-Native Query Engines

Sirius (Chapter 6) represents our initial step toward practical and scalable GPU-native query engines. Below, we outline several future directions for further advancing the design and implementation of such systems.

Out-of-Core Execution. Currently, Sirius assumes that input data fits entirely within the GPU caching region and that all intermediate results can be accommodated in the GPU processing region. To support larger-than-memory workloads, the system must be extended to operate out of core.

First, Sirius currently processes each table as a whole. To enable out-of-core execution, tables need to be partitioned into fixed-size data batches (e.g., on the order of a few gigabytes), allowing operators to process data incrementally at the batch level rather than requiring the entire input to reside in GPU memory.

Second, the buffer manager must be extended to manage data movement across multiple memory tiers, including GPU memory, CPU pinned memory, and disk. This involves efficiently transferring batches between tiers, overlapping data transfers with

computation, and evicting or prefetching batches based on access patterns. Such mechanisms would allow Sirius to sustain GPU utilization while respecting memory constraints.

Distributed and Multi-GPU Execution. The current distributed execution mode of Sirius provides more limited SQL coverage than its single-node counterpart, as shown in Section 6.3. For example, certain aggregate functions, such as AVG, are not yet supported. Future work includes expanding SQL coverage for distributed execution [62] and incorporating additional system-level features, such as fault tolerance.

In addition, we plan to extend Sirius to support multiple GPUs per node [213], enabling it to scale to larger workloads and better utilize heterogeneous hardware resources. This requires addressing challenges in workload partitioning, GPU-aware scheduling, and inter-GPU data movement.

Finally, the current exchange service in Sirius performs data exchange across nodes synchronously, introducing execution barriers between pipeline stages. With batch-level, pipelined execution, the exchange operators should also be redesigned to operate asynchronously, allowing computation and communication to overlap. This would improve resource utilization and reduce end-to-end query latency in distributed, multi-GPU deployments.

Cloud Storage and GPU-Native Scan. At present, Sirius relies on DuckDB’s internal scan executors to read disk-resident data. Future designs should extend Sirius to support additional storage backends, including cloud object stores, as well as open data formats such as Parquet.

Another important direction is the design of GPU-native scan operators. By leveraging technologies such as GPUDirect Storage and GPUDirect RDMA, Sirius can directly load compressed raw data from storage media into GPU memory, bypassing unnecessary CPU involvement. Compute-intensive tasks such as decoding and decompression can then be executed natively and asynchronously on the GPU, overlapping with data transfers from storage. Such a design is essential for approaching peak I/O bandwidth and maximizing GPU utilization in end-to-end query execution.

Expanding SQL Coverage. Sirius already supports most fundamental SQL operators and data types. A natural next step is to broaden this coverage by adding support for more complex and expressive data types, such as LISTS, ARRAYS, JSON, and other nested types. In addition, we plan to implement advanced SQL operators in

Sirius, including ASOF joins, window functions, recursive queries, vector search, and user-defined functions (UDFs).

Expanding Host Database Coverage. Sirius currently serves as a drop-in accelerator for both DuckDB and Apache Doris. Going forward, we plan to support other substrait-compatible systems, such as DataFusion, Ibis, and SparkSQL.

Performance Optimizations. As Sirius is still in an early stage of development, there remains substantial room for performance improvement across multiple layers of the execution engine. One important direction is to introduce fully asynchronous execution, enabling computation to overlap with data transfers across GPUs, the network, and storage, thereby improving end-to-end throughput. At the operator level, several SQL operators offer significant optimization potential on GPUs. Examples include more efficient implementations of top-N selection, grouped aggregation on string columns, and computationally expensive aggregate functions such as “count(distinct)”. These operators often exhibit irregular memory access patterns and non-uniform workloads, making them particularly challenging for GPU-specific optimizations. In addition, integrating lightweight and GPU-friendly compression schemes [179, 59] can reduce data movement and memory footprint, especially when combined with GPU-native decoding and decompression. Finally, recent advances in algorithmic runtime query optimizations, such as predicate transfer (Section 2 and 3), offer complementary opportunities to reduce input sizes early in execution and further improve overall performance when integrated into GPU-native execution pipelines.

Chapter 8

Conclusion

Large-scale data analytics has become a cornerstone of modern organizations. However, the rapid expansion of data collection is making analytics increasingly challenging. Modern techniques are incorporating innovations across the stack, from query execution algorithms to cloud and hardware infrastructure, each addressing performance from a different angle. In this thesis, we investigate performance optimizations for large-scale SQL analytics across these two key stacks.

At the algorithmic stack, we develop data reduction techniques that significantly shrink intermediate results in large complex join queries. By reducing the volume of data processed during query execution, these techniques directly accelerate expensive join operations and improve performance robustness.

In Chapter 2, we present predicate transfer, a novel pre-filtering technique for multi-join queries built on the theoretical foundation of the Yannakakis algorithm. Predicate transfer generalizes Bloom join by propagating table-local predicates to pre-filter multiple related tables, eliminating a large number of non-contributing input rows early and significantly reducing the size of intermediate join results, which leads to substantial performance improvements. Going one step further, we propose a heuristic variant for generating predicate transfer schedules that guarantee full reduction of the input tables, resulting in robust query performance across random join orders while preserving the benefits of aggressive pre-filtering. This robustness makes predicate transfer a promising approach for simplifying join optimization logic in future query optimizers.

In Chapter 3, we extend predicate transfer to distributed environments, making it practical for modern large-scale SQL analytics. Building on distributed join techniques, we present algorithms for both broadcast-based and shuffle-based distributed

predicate transfer. Based on these designs, we develop an adaptive model that dynamically selects the optimal strategy for each predicate transfer step to minimize network traffic. Furthermore, we propose a pruning algorithm that eliminates unnecessary predicate transfer steps, reducing transfer overhead without sacrificing pre-filtering effectiveness. Together, these techniques significantly accelerate complex join processing in distributed settings.

In addition, we make an initial effort to study complementary techniques for complex cyclic join queries (Section 7.1.2) by bringing theoretically grounded partition-based join algorithms with strong guarantees into practical systems. Starting from algorithms with provable theoretical properties, we propose several heuristics to account for real-world data distributions and workload characteristics. Our preliminary results demonstrate significant performance improvements and a higher query success rate than existing systems on social network workloads and cyclic query benchmarks.

At the infrastructure layer, we explore optimizations from two perspectives: cloud-native DBMSs and GPU-native query execution engines. In Chapter 4, we investigate techniques to alleviate the network bottleneck between the compute and storage layers in cloud-native databases. We present FlexPushdownDB, which leverages a fine-grained hybrid execution framework that combines the benefits of computation pushdown and data caching, significantly reducing data movement between the compute and storage layers. In addition, we revisit traditional cache replacement policies and propose a new policy tailored to the pushdown context, which unifies caching and pushdown within a single cost model to guide optimal cache eviction decisions. In Chapter 5, we further enhance computation pushdown techniques by proposing an adaptive pushdown framework that prevents storage-layer computation from becoming a bottleneck in multi-tenant environments. We also derive a general principle for identifying pushdown-amenable computational tasks and demonstrate its effectiveness by enabling selection bitmap operations and distributed data shuffle at the storage layer. Together, these optimizations achieve higher performance with substantially reduced data movement and lower resource consumption across both compute and storage layers for cloud-native DBMSs.

In Chapter 6, we present Sirius, our prototype for GPU-native SQL analytics. Leveraging recent hardware trends, including larger GPU memory capacities, faster interconnects and I/O, and declining hardware costs, we achieve significantly lower

total cost of ownership compared to state-of-the-art systems. Building on modern software trends such as composable data systems and mature GPU-accelerated libraries, Sirius enables drop-in acceleration for existing database engines with minimal modifications to the database software stack. We demonstrate the feasibility and performance benefits of our approach on both production-grade single-node and distributed query engines.

Taken together, the techniques and systems presented in this thesis aim to contribute foundational insights toward the development of next-generation query engines for large-scale SQL analytics. By advancing both algorithmic optimizations for data reduction and infrastructure-aware execution strategies, this work addresses key performance challenges in modern analytical workloads. We hope these ideas will inspire further research on robust, adaptive, and scalable query processing, and guide the development of future analytical database systems that can fully leverage emerging hardware platforms and cloud-native architectures.

references

- [1] 2017. Oracle Database Guide: Using Zone Maps. https://docs.oracle.com/database/121/DWHSG/zone_maps.htm.
- [2] 2020. Azure Data Lake Storage query acceleration. <https://docs.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-query-acceleration/>.
- [3] 2020. Presto Release 0.241.
- [4] 2021. Best Practices for Optimizing Your DBT and Snowflake Deployment. <https://www.snowflake.com/wp-content/uploads/2021/10/Best-Practices-for-Optimizing-Your-dbt-and-Snowflake-Deployment.pdf>.
- [5] 2021. DB2 Workload Manager. <https://www.ibm.com/docs/en/db2/10.1.0?topic=manager-db2-workload>.
- [6] 2022. TPC-H Benchmark on Doris. <https://doris.apache.org/zh-CN/blog/tpch/>.
- [7] 2023. Data Skipping Index in Spark. <https://docs.databricks.com/en/archive/spark-3.x-migration/dataskipping-index.html>.
- [8] 2024. Optimize Performance with Caching on Databricks. <https://docs.databricks.com/en/optimizations/disk-cache.html>.
- [9] 2024. The Missing Guide to the H100 GPU Market. <https://blog.lepton.ai/the-missing-guide-to-the-h100-gpu-market-91ebfed34516>.
- [10] 2025. HeavyAI. <https://github.com/heavyai>.

- [11] 2025. Intelligent Query Processing in SQL Databases. <https://learn.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing>.
- [12] 2025. Parquet Thrift Format. <https://github.com/apache/parquet-format/blob/master/src/main/thrift/parquet.thrift>.
- [13] 2025. Presto Documentation: Alluxio SDK Cache.
- [14] 2025. Presto Documentation: Properties. <https://prestodb.io/docs/current/admin/properties.html>.
- [15] 2025. Spark Documentation — Adaptive Query Execution. <https://spark.apache.org/docs/latest/sql-performance-tuning.html>.
- [16] 2025. SQL Server Documentation — Resource Governor. <https://learn.microsoft.com/en-us/sql/relational-databases/resource-governor/resource-governor>.
- [17] 2025. TPC-H Benchmarking on StarRocks. https://docs.starrocks.io/docs/benchmarking/TPC-H_Benchmarking/.
- [18] 2026. Akka. <https://akka.io/>.
- [19] 2026. Amazon Athena — Serverless Interactive Query Service. <https://aws.amazon.com/athena/>.
- [20] 2026. Amazon EC2.
- [21] 2026. Amazon Redshift. <https://aws.amazon.com/redshift/>.
- [22] 2026. Amazon S3. <https://aws.amazon.com/s3/>.
- [23] 2026. Amazon S3 documentation — GetObject. https://docs.aws.amazon.com/AmazonS3/latest/API/API_GetObject.html.
- [24] 2026. Apache Arrow. <https://arrow.apache.org/>.
- [25] 2026. Apache Calcite. <https://calcite.apache.org/>.
- [26] 2026. Apache Doris. <https://doris.apache.org/>.

- [27] 2026. Apache Hadoop.
- [28] 2026. Apache Iceberg. <https://iceberg.apache.org/>.
- [29] 2026. Apache ORC Spec. v1. <https://orc.apache.org/specification/ORCv1/>.
- [30] 2026. Apache Parquet. <https://parquet.apache.org/>.
- [31] 2026. Arrow Flight RPC. <https://arrow.apache.org/docs/format/Flight.html>.
- [32] 2026. Arrow IPC Format. <https://arrow.apache.org/docs/format/Columnar.html>.
- [33] 2026. AWS Nitro System. <https://aws.amazon.com/ec2/nitro/>.
- [34] 2026. Ceph. <https://ceph.io/>.
- [35] 2026. ClickBench: A Benchmark For Analytical Databases. <https://github.com/ClickHouse/ClickBench>.
- [36] 2026. cuDF. <https://github.com/rapidsai/cudf>.
- [37] 2026. Data Distribution and Redistribution for AWS Redshift. https://docs.aws.amazon.com/redshift/latest/dg/c_data_redistribution.html.
- [38] 2026. Dremio SQL Query Engine. <https://www.dremio.com/platform/sql-query-engine>.
- [39] 2026. DuckDB Extensions. <https://duckdb.org/docs/stable/extensions/overview>.
- [40] 2026. GPU Direct. <https://developer.nvidia.com/gpudirect>.
- [41] 2026. gRPC. <https://grpc.io>.
- [42] 2026. Kinetica. <https://kinetica.com/>.
- [43] 2026. MinIO. <https://min.io/>.
- [44] 2026. NCCL. <https://developer.nvidia.com/nccl>.

- [45] 2026. NVIDIA H100 Price Guide 2026: GPU Costs, Cloud Pricing & Buy vs Rent. <https://docs.jarvislabs.ai/blog/h100-price>.
- [46] 2026. NVLink-C2C. <https://www.nvidia.com/en-us/data-center/nvlink-c2c/>.
- [47] 2026. PolarDB for MySQL. <https://www.alibabacloud.com/en/product/polardb-for-mysql>.
- [48] 2026. PolarDB-X. <https://www.alibabacloud.com/product/polardb-x>.
- [49] 2026. PostgreSQL Extensions. <https://www.postgresql.org/download/products/6-postgresql-extensions/>.
- [50] 2026. Rapids Memory Manager. <https://github.com/rapidsai/rmm>.
- [51] 2026. Spark RAPIDS. <https://nvidia.github.io/spark-rapids/>.
- [52] 2026. Substrait. <https://substrait.io/>.
- [53] 2026. TPC-DS Benchmark. <http://www.tpc.org/tpcds/>.
- [54] 2026. TPC-H Benchmark. <http://www.tpc.org/tpch/>.
- [55] Abadi, Daniel J., Samuel R. Madden, and Nabil Hachem. 2008. Column-Stores vs. Row-Stores: How Different Are They Really? In *Proceedings of the 2008 acm sigmod international conference on management of data*, 967–980.
- [56] Aboulnaga, Ashraf, and Shivnath Babu. 2013. Workload Management for Big Data Analytics. In *Proceedings of the 2013 acm sigmod international conference on management of data*, 929–932.
- [57] Adam Wright. 2025. Worldwide IDC Global DataSphere Forecast Update, 2025–2029: Regional Analysis. Tech. Rep., International Data Corporation (IDC).
- [58] Afroozeh, Ali, Bogdan Ghit, and Greg Rahn. 2020. Faster SQL Queries on Delta Lake with Dynamic File Pruning. <https://www.databricks.com/blog/2020/04/30/faster-sql-queries-on-delta-lake-with-dynamic-file-pruning.html>.

- [59] Afroozeh, Azim, Lotte Feliuss, and Peter Boncz. 2024. Accelerating GPU Data Processing using FastLanes Compression. In *Proceedings of the 20th international workshop on data management on new hardware*. DaMoN '24, New York, NY, USA: Association for Computing Machinery.
- [60] Agha, Gul. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press.
- [61] Almeida, Paulo Sérgio, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable Bloom Filters. *Information Processing Letters* 101(6):255–261.
- [62] Aramburú, Felipe, William Malpica, Kaouther Abrougui, Amin Aramoon, Romulo Auccapuclla, Claude Brisson, Matthijs Brobbel, Colby Farrell, Pradeep Garigipati, Joost Hoozemans, et al. 2025. Theseus: A Distributed and Scalable GPU-Accelerated Query Processing Platform Optimized for Efficient Data Movement. *arXiv preprint arXiv:2508.05029*.
- [63] Armbrust, Michael, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13(12):3411–3424.
- [64] Armbrust, Michael, 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.
- [65] Armenatzoglou, Nikos, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinsky, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-Invented. In *Proceedings of the 2022 international conference on management of data*, 2205–2217.
- [66] Armstrong, Joe. 1996. Erlang — a Survey of the Language and its Industrial Applications. In *Proc. inap*, vol. 96.

- [67] Avnur, Ron, and Joseph M Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 acm sigmod international conference on management of data*, 261–272.
- [68] Bancilhon, Francois, 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.
- [69] Barber, R., 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):353–364.
- [70] Barr, Jeff. 2021. AQUA (Advanced Query Accelerator) — A Speed Boost for Your Amazon Redshift Queries. <https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/>.
- [71] Barthels, Claude, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed Join Algorithms on Thousands of Cores. *Proceedings of the VLDB Endowment* 10(5):517–528.
- [72] Beeri, C., 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.
- [73] Belady, L. A. 1966. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM System Journal* 5(2):78–101.
- [74] Bernstein, Philip A., and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28(1):25–40.
- [75] Bernstein, Philip A., 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):602–625.
- [76] Birler, Altan, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proceedings of the VLDB Endowment* 17(11):3215–3228.

- [77] Blanas, Spyros, 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.
- [78] Bloom, Burton H. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM* 13(7):422–426.
- [79] Bratbergsengen, Kjell. 1984. Hashing Methods and Relational Algebra Operations. In *Proceedings of the 10th international conference on very large data bases*, 323–333.
- [80] Bruno, Nicolas, YongChul Kwon, and Ming-Chuan Wu. 2014. Advanced Join Strategies for Large-Scale Distributed Computation. *Proc. VLDB Endow.* 7(13): 1484–1495.
- [81] Cao, Wei, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 international conference on management of data*, 2477–2489.
- [82] Chainani, Naresh, Berni Schiefer, and Neeraja Rentachintala. 2020. Improved Speed and Scalability in Amazon Redshift. <https://aws.amazon.com/blogs/big-data/improved-speed-and-scalability-in-amazon-redshift/>.
- [83] Charousset, Dominik, Raphael Hiesgen, and Thomas C. Schmidt. 2016. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures* 45(C):105–131.
- [84] Chen, and Li. 1984. Improvement Algorithms for Semijoin Query Processing Programs in Distributed Database Systems. *IEEE Transactions on Computers* C-33(11):959–967.
- [85] Chen, M.-S., and P.S. Yu. 1993. Combining Joint and Semi-Join Operations for Distributed Query Processing. *IEEE Transactions on Knowledge and Data Engineering* 5(3):534–542.
- [86] Chen, Ming-Syan, 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):152–165.

- [87] Chen, Shimin, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving Hash Join Performance through Prefetching. *ACM Trans. Database Syst.* 32(3):17–es.
- [88] Chiu, Dah Ming, 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.
- [89] Chrysogelos, Periklis, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating Heterogeneous CPU-GPU Parallelism in JIT Compiled Engines. *Proc. VLDB Endow.* 12(5):544–556.
- [90] Colgan, Maria. 2023. Getting Started with Oracle Database In-Memory Part IV — Joins In The IM Column Store. <https://blogs.oracle.com/in-memory/post/getting-started-with-oracle-database-in-memory-part-iv-joins-in-the-im-column-store>.
- [91] Consortium, Hybrid Memory Cube. 2014. HMC Specification 2.1.
- [92] Dageville, Benoit, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Sigmod*, 215–226.
- [93] Deep, Shaleen, Xiao Hu, and Paraschos Koutris. 2020. Fast Join Project Query Evaluation using Matrix Multiplication. In *Proceedings of the 2020 acm sigmod international conference on management of data*, 1213–1223. SIGMOD ’20, New York, NY, USA: Association for Computing Machinery.
- [94] Deep, Shaleen, Hangdong Zhao, Austen Z. Fan, and Paraschos Koutris. 2024. Output-Sensitive Conjunctive Query Evaluation. *Proc. ACM Manag. Data* 2(5).
- [95] Deshpande, Amol, Zachary Ives, Vijayshankar Raman, et al. 2007. Adaptive Query Processing. *Foundations and Trends® in Databases* 1(1):1–140.
- [96] Ding, Bailu, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14(13):3376–3388.

- [97] Ding, Bailu, 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.
- [98] Ding, Jialin, 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.
- [99] Do, Jaeyoung, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Sigmod*, 1221–1230.
- [100] Fortune Business Insights. 2025. Big Data Analytics Market to Grow at a CAGR of 13.5 over 2024 to 2032.
- [101] Francisco, Phil. 2011. The Netezza Data Appliance Architecture.
- [102] Franklin, Michael J., Björn Thór Jónsson, and Donald Kossmann. 1996. Performance Tradeoffs for Client-Server Query Processing. *SIGMOD Record* 25(2): 149–160.
- [103] Freitag, Michael, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proceedings of the VLDB Endowment* 13(12):1891–1904.
- [104] Fushimi, Shinya, Masaru Kitsuregawa, and Hidehiko Tanaka. 1986. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In *Vldb*, 209–219.
- [105] Gao, Mingyu, and Christos Kozyrakis. 2016. HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing. In *HPCA*, 126–137.
- [106] Garcia-Molina, Hector, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book*. 2nd ed. Upper Saddle River, NJ: Prentice Hall.
- [107] Gekk, Maxim, Wenchen Fan, Hyukjin Kwon, Serge Rielau, Yingyi Bu, Xiao Li, and Reynold Xin. 2022. Introducing Apache Spark™ 3.3 for Databricks Runtime 11.0. <https://www.databricks.com/blog/2022/06/15/introducing-apache-spark-3-3-for-databricks-runtime-11-0.html>.

- [108] Ghose, Saugata, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. 2018. Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions. *arXiv preprint arXiv:1802.00320*.
- [109] Gouda, Mohamed G, 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.
- [110] Gray, Jim, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. *SIGMOD Record* 23(2):243–252.
- [111] Gu, Boncheol, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanhoo Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *Isca*, 153–165.
- [112] Gupta, Anurag, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Sigmod*, 1917–1923.
- [113] Haritsa, Jayant R. 2019. Robust Query Processing: Mission Possible. In *2019 ieee 35th international conference on data engineering (icde)*, 2072–2075.
- [114] Harmouch, Hazar, and Felix Naumann. 2017. Cardinality Estimation: An Experimental Survey. *Proc. VLDB Endow.* 11(4):499–512.
- [115] He, Yujun, Hangdong Zhao, Simon Frisk, Yifei Yang, Kevin Kristensen, Paraschos Koutris, and Xiangyao Yu. 2025. One Join Order Does Not Fit All: Reducing Intermediate Results with Per-Split Query Plans. *arXiv preprint arXiv:2510.25684*.
- [116] Hellerstein, Joseph M. 1998. Optimization Techniques for Queries with Expensive Methods. *ACM Trans. Database Syst.* 23(2):113–157.
- [117] Hellerstein, Joseph M., Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Samuel Madden, Vijayshankar Raman, and Mehul A.

- Shah. 2000. Adaptive Query Processing: Technology in Evolution. *IEEE Data Eng. Bull.* 23(2):7–18.
- [118] Hellerstein, Joseph M., 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.
 - [119] Hu, Xiao. 2024. Fast Matrix Multiplication for Query Processing. *Proc. ACM Manag. Data* 2(2).
 - [120] Hunt, Randall. 2017. S3 Select and Glacier Select – Retrieving Subsets of Objects. <https://aws.amazon.com/blogs/aws/s3-glacier-select/>.
 - [121] Jarett, Leigha, and Seth Hollyman. 2021. BigQuery Admin Reference Guide: Query Optimization. <https://cloud.google.com/blog/topics/developers-practitioners/bigquery-admin-reference-guide-query-optimization>.
 - [122] Joglekar, Manas R., Rohan Puttagunta, and Christopher Ré. 2016. AJAR: Aggregations and Joins over Annotated Relations. In *Proceedings of the 35th acm sigmod-sigact-sigai symposium on principles of database systems*, 91–106. PODS '16, New York, NY, USA: Association for Computing Machinery.
 - [123] Jun, Sang-Woo, Shuotao Xu, and Arvind. 2017. Terabyte Sort on FPGA-accelerated Flash Storage. In *Ieee international symposium on field-programmable custom computing machines (fccm)*, 17–24.
 - [124] Kabić, Marko, Shriram Chandran, and Gustavo Alonso. 2025. Maximus: A Modular Accelerated Query Engine for Data Analytics on Heterogeneous Systems. *Proc. ACM Manag. Data* 3(3).
 - [125] Kabra, Navin, and David J DeWitt. 1998. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *Proceedings of the 1998 acm sigmod international conference on management of data*, 106–117.
 - [126] Kambayashi, Yahiko, 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.

- [127] Kandula, Srikanth, Laurel Orr, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates through Joins in Big-Data Clusters. *Proc. VLDB Endow.* 13(3):252–265.
- [128] Keeton, Kimberly, David A Patterson, and Joseph M Hellerstein. 1998. A Case for Intelligent Disks (IDISks). *SIGMOD Record* 27(3):42–52.
- [129] Kepe, Tiago R., Eduardo C. de Almeida, and Marco A. Z. Alves. 2019. Database Processing-in-Memory: An Experimental Study. *VLDB* 13(3):334–347.
- [130] Khamis, Mahmoud Abo, Hung Q. Ngo, and Dan Suciu. 2025. PANDA: Query Evaluation in Submodular Width. *TheoretCS* 4(12):1–42.
- [131] Kim, Changkyu, 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):1378–1389.
- [132] Kim, Kyoungmin, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-Depth Study. In *Proceedings of the 2022 international conference on management of data*, 1214–1227.
- [133] Kimball, Gregory. 2025. Extending Velox — GPU Acceleration with cuDF. <https://velox-lib.io/blog/extending-velox-with-cudf/>.
- [134] Koo, Gunjae, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading Communication with Computing Near Storage. In *Micro*, 219–231.
- [135] Koutris, Paraschos. 2011. Bloom Filters in Distributed Query Execution. *Principles of DBMS - University of Washington*.
- [136] Lamb, Andrew, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *VLDB* 5(12):1790–1801.
- [137] Lan, Hai, Zhifeng Bao, and Yuwei Peng. 2021. A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. *Data Science and Engineering* 6:86–101.

- [138] Lehmann, Claude, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. *Proceedings of the VLDB Endowment* 17(7):1565–1577.
- [139] Leis, Viktor, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9(3):204–215.
- [140] Levy, Alon Y, Inderpal Singh Mumick, and Yehoshua Sagiv. 1994. Query Optimization by Predicate Move-Around. In *Vldb*, 96–107.
- [141] Lin, Xuemin, 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.
- [142] Lin, Yuting, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. 2011. Llama: Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework. In *Proceedings of the 2011 acm sigmod international conference on management of data*, 961–972.
- [143] Lohman, Guy. 2014. Is Query Optimization a “Solved” Problem. In *Proc. workshop on database query optimization*, vol. 13, 10.
- [144] Luo, Lailong, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. 2018. Optimizing Bloom Filter: Challenges, Solutions, and Comparisons. *IEEE Communications Surveys & Tutorials* 21(2):1912–1949.
- [145] Maier, David. 1983. *The Theory of Relational Databases*.
- [146] Mehta, Manish, and David J DeWitt. 1997. Data Placement in Shared-Nothing Parallel Database Systems. *The VLDB journal* 6(1):53–72.
- [147] Melnik, Sergey, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *VLDB* 3(1-2):330–339.
- [148] Milkai, Elena, Xiangyao Yu, and Jignesh M Patel. 2025. Hermes: Off-the-Shelf Real-Time Transactional Analytics. *Proceedings of the VLDB Endowment* 18(8): 2334–2347.

- [149] Mumick, I. S., 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.
- [150] Mumick, Inderpal Singh, 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.
- [151] Nadeau, Jacques. 2018. Gandiva: An LLVM-Based Arrow Expression Compiler. <https://arrow.apache.org/blog/2018/12/05/gandiva-donation/>.
- [152] Nanda, Arup. 2011. Smart Scans Meet Storage Indexes. <https://blogs.oracle.com/connect/post/smart-scans-meet-storage-indexes>.
- [153] Neumann, Thomas, and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on innovative data systems research*.
- [154] Ngo, Hung Q, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-Case Optimal Join Algorithms. *Journal of the ACM (JACM)* 65(3):1–40.
- [155] Niu, Baoning, Patrick Martin, and Wendy Powley. 2009. Towards Autonomic Workload Management in DBMSs. *Journal of Database Management (JDM)* 20(3): 1–17.
- [156] O’Neil, Patrick, Elizabeth O’Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Technology conference on performance evaluation and benchmarking*, 237–252.
- [157] Pagh, Anna, Rasmus Pagh, and S Srinivasa Rao. 2005. An Optimal Bloom Filter Replacement. In *Soda*, 823–829.
- [158] Pang, Hwee Hwa, Michael J Carey, and Miron Livny. 1993. Memory-Adaptive External Sorting. Tech. Rep., University of Wisconsin-Madison Department of Computer Sciences.
- [159] ———. 1993. Partially Preemptible Hash Joins. In *Proceedings of the 1993 acm sigmod international conference on management of data*, 59–68.

- [160] Patel, Jignesh M., 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):663–676.
- [161] Pedreira, Pedro, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta’s Unified Execution Engine. *Proc. VLDB Endow.* 15(12):3372–3384.
- [162] Pedreira, Pedro, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16(10): 2679–2685.
- [163] Polychroniou, Orestis, 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.
- [164] Pramanik, Sakti, 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.
- [165] Qiao, Yiming. 2025. Implement optimized robust predicate transfer to improve duckdb robustness. <https://github.com/duckdb/duckdb/pull/17326>.
- [166] Raasveldt, Mark, and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 international conference on management of data*, 1981–1984.
- [167] Raman, Vijayshankar, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proc. VLDB Endow.* 6(11):1080–1091.
- [168] Ramesh, Sukriti, 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. Springer.
- [169] Rescorla, Eric. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446.
 - [170] Riedel, Erik, Christos Faloutsos, Garth A Gibson, and David Nagle. 2001. Active Disks for Large-Scale Data Processing. *Computer* 34(6):68–74.
 - [171] Rodríguez-Martínez, Manuel, and Nick Roussopoulos. 2000. MOCHA: A Self-Extensible Database Middleware System for Distributed Data Sources. In *Sigmod*, 213–224.
 - [172] Roth, Mary Tork, and Peter M. Schwarz. 1997. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *Vldb*, 266–275.
 - [173] Rupperecht, Lukas, William Culhane, and Peter Pietzuch. 2017. SquirrelJoin: Network-Aware Distributed Join Processing with Lazy Partitioning. *Proc. VLDB Endow.* 10(11):1250–1261.
 - [174] Rödiger, Wolf, 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.
 - [175] Salihoglu, Semih. 2023. Kuzu: A Database Management System For "Beyond Relational" Workloads. *SIGMOD Rec.* 52(3):39–40.
 - [176] Schulze, Robert, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.* 17(12):3731–3744.
 - [177] Selinger, P. Griffiths, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 acm sigmod international conference on management of data*, 23–34.
 - [178] Sethi, Raghav, 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.
- [179] Shanbhag, Anil, Bobbi Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based Lightweight Integer Compression in GPU. In *Proceedings of the 2022 ACM SIGMOD international conference on management of data*.
- [180] Ślęzak, Dominik, Jakub Wróblewski, Victoria Eastwood, and Piotr Synak. 2008. Brighthouse: an analytic data warehouse for ad-hoc queries. *Proceedings of the VLDB Endowment* 1(2):1337–1345.
- [181] Steinbrunn, Michael, Guido Moerkotte, and Alfons Kemper. 1997. Heuristic and Randomized Optimization for the Join Ordering Problem. *The VLDB journal* 6:191–208.
- [182] Stocker, K., D. Kossmann, R. Braumand, 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.
- [183] Stonebraker, Mike, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O’Neil, Pat O’Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st international conference on very large data bases*, 553–564.
- [184] Sun, Yutian, Tim Meehan, Rebecca Schlusel, 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 Shankdhar, 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).
- [185] Swami, Arun, and Anoop Gupta. 1988. Optimization of Large Join Queries. In *Proceedings of the 1988 ACM SIGMOD international conference on management of data*, 8–17.

- [186] Tan, Junjay, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing A Cloud DBMS: Architectures and Tradeoffs. *VLDB* 12(12): 2170–2182.
- [187] Tarkoma, Sasu, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2011. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Surveys & Tutorials* 14(1):131–155.
- [188] Thusoo, Ashish, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive — A Petabyte Scale Data Warehouse Using Hadoop. In *Icde*, 996–1005.
- [189] Toor, Jon. 2024. Clouidian Shatters AI Storage Barriers with Direct GPU-to-Object Storage Access. <https://cloudian.com/blog/cloudian-delivers-groundbreaking-performance-with-nvidia-gpudirect-support>.
- [190] Ubell, Michael. 1985. The Intelligent Database Machine (IDM). In *Query processing in database systems*, 237–247.
- [191] Ullman, Jeffrey D. 1988. *Principles of Database and Knowledge-Base Systems, Vol. I*. Computer Science Press, Inc.
- [192] Valduriez, Patrick. 1982. Semi-Join Algorithms for Multiprocessor Systems. In *Proceedings of the 1982 acm sigmod international conference on management of data*, 225–233.
- [193] Vance, Bennet, and David Maier. 1996. Rapid Bushy Join-Order Optimization with Cartesian Products. In *Proceedings of the 1996 acm sigmod international conference on management of data*, 35–46.
- [194] Vandiver, Ben, Shreya Prasad, Pratibha Rana, Eden Zik, Amin Saeidi, Pratyush Parimal, Styliani Pantela, and Jaimin Dave. 2018. Eon Mode: Bringing the Vertica Columnar Database to the Cloud. In *Sigmod*, 797–809.
- [195] Vavilapalli, Vinod Kumar, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O’Malley, Sanjay Radia, Benjamin

- Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual symposium on cloud computing*, 1–16.
- [196] Veldhuizen, Todd L. 2014. Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. international conference on database theory*.
- [197] Verbitski, Alexandre, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Sigmod*, 1041–1052.
- [198] Verbitski, Alexandre, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, et al. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *Sigmod*, 789–796.
- [199] Vuppalapati, Midhul, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an Elastic Query Engine on Disaggregated Storage. In *Nsdi*, 449–462.
- [200] Wang, C., 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.
- [201] Wang, Xiaoying, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready for Learned Cardinality Estimation? *Proceedings of the VLDB Endowment* 14(9).
- [202] Wang, Yisu Remy, Max Willsey, and Dan Suciu. 2023. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proceedings of the ACM on Management of Data* 1(2):1–23.
- [203] Weiss, Ronald. 2012. A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server. *Oracle White Paper*.
- [204] Woods, Louis, Zsolt István, and Gustavo Alonso. 2014. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *VLDB* 7(11): 963–974.

- [205] Wu, Jiacheng, and Dan Suciu. 2025. HoneyComb: A Parallel Worst-Case Optimal Join on Multicores. In *SIGMOD '25: International conference on management of data, berlin, germany, june 22 - 27, 2025*. ACM.
- [206] Wu, Wentao, Jeffrey F Naughton, and Harneet Singh. 2016. Sampling-Based Query Re-Optimization.
- [207] Xu, Shuotao, Thomas Bourgeat, Tianhao Huang, Hojun Kim, Sungjin Lee, and Arvind Arvind. 2020. AQUOMAN: An Analytic-Query Offloading Machine. In *Micro*, 386–399.
- [208] Yang, Yifei, 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. *Proc. VLDB Endow.* 14(11): 2101–2113.
- [209] Yang, Yifei, and Xiangyao Yu. 2025. Accelerate Distributed Joins with Predicate Transfer. *Proc. ACM Manag. Data* 3(3).
- [210] Yang, Yifei, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2024. FlexpushdownDB: Rethinking Computation Pushdown for Cloud OLAP DBMSs. *The VLDB Journal* 33(5):1643–1670.
- [211] Yang, Yifei, 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)*.
- [212] Yannakakis, Mihalis. 1981. Algorithms for Acyclic Database Schemes. In *Proceedings of the seventh international conference on very large data bases - volume 7*, 82–94. VLDB '81, VLDB Endowment.
- [213] Yogatama, Bobbi, Weiwei Gong, and Xiangyao Yu. 2024. Scaling your Hybrid CPU-GPU DBMS to Multiple GPUs. *Proc. VLDB Endow.* 17(13):4709–4722.
- [214] Yogatama, Bobbi, Yifei Yang, Kevin Kristensen, Devesh Sarda, Abigale Kim, Adrian Cockcroft, Yu Teng, Joshua Patterson, Gregory Kimball, Wes McKinney, et al. 2026. Rethinking Analytical Processing in the GPU Era. In *Proceedings of the conference on innovative data systems research (cidr)*.

- [215] Yogatama, Bobbi W, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating Data Placement and Query Execution in Heterogeneous CPU-GPU DBMS. *Proceedings of the VLDB Endowment* 15(11):2491–2503.
- [216] Yu, C. T., 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.
- [217] Yu, Xiangyao, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS using S3 Computation. In *Icde*, 1802–1805.
- [218] Yun, Channy. 2025. Announcing up to 45% Price Reduction for Amazon EC2 NVIDIA GPU-Accelerated Instances. <https://aws.amazon.com/blogs/aws/announcing-up-to-45-price-reduction-for-amazon-ec2-nvidia-gpu-accelerated-instances/>.
- [219] Zhang, Haozhe, Christoph Mayer, Mahmoud Abo Khamis, Dan Olteanu, and Dan Suciu. 2025. LpBound: Pessimistic Cardinality Estimation Using lp-Norms of Degree Sequences. *Proc. ACM Manag. Data* 3(3).
- [220] Zhang, Mingyi, Patrick Martin, Wendy Powley, and Jianjun Chen. 2018. Workload Management in Database Management Systems: A Taxonomy. *IEEE Transactions on Knowledge and Data Engineering* 30(7):1386–1402.
- [221] Zhang, Weiye, and Per-Ake Larson. 1997. Dynamic Memory Adjustment for External Mergesort. In *Vldb*, vol. 97, 25–29.
- [222] Zhao, Junyi, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. *Proc. ACM Manag. Data* 3(3).
- [223] ———. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. *arXiv preprint arXiv:2502.15181*.
- [224] Zhu, Jianqiao, 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):889–900.