

Low Latency Query Planning and Processing in Database Systems

Philipp Fent

Low Latency Query Planning and Processing in Database Systems

Philipp Fent

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:

Prof. Dr. Viktor Leis

Prüfer der Dissertation:

1. Prof. Dr. Thomas Neumann
2. Prof. Alfons Kemper, Ph.D.
3. Prof. Dr. Guido Moerkotte

Die Dissertation wurde am 26. Oktober 2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 21. Februar 2024 angenommen.

Abstract

Efficient data processing is one of the core techniques that enables modern data driven computer systems. Database systems are uniquely positioned to use increasing hardware capabilities to enable better data processing. Since database systems have a long history, modern systems need to reengineer the data processing pipelines to take full advantage of fast hardware that allows reducing the query response latency. To achieve low latency, systems not only need to increase the data processing throughput through new hardware, but also need to tune query planning for a low end-to-end latency.

In the first part of this thesis, we focus on parallel processing with shared state, how this affects the choosing of algorithms, and how a system can execute such tasks. We propose new evaluation strategies for groupjoins, which join and aggregate data without unnecessarily duplicating state. We also discuss the challenges of cardinality estimation for query planning with complex calculated expressions, and develop a novel approach to estimating the such expressions. Combined with the parallel processing in groupjoins, this improves database system performance for analytical queries that aggregate data.

Second, we address the latency challenges of query planning. Database systems first plan the query before they process the data, where the processing was the limiting factor for a long time. With faster hardware, query planning becomes a larger part of end-to-end latency, and we can observe bad performance of established planning techniques. In this thesis, we propose Indexed Algebra as a novel technique that uses a dynamic index structure to asymptotically reduce the time to plan a query.

This thesis concludes with a discussion of our results, and gives an outlook on future research directions. The results of this dissertation allow low latency data processing in more scenarios, which could be extended even further. Modern SSDs, for example, allow low latency processing even in out-of-memory scenarios, and we see potential to reduce queries latency by adaptive recompilation of query plans.

Zusammenfassung

Effiziente Datenverarbeitung ist eine der Schlüsseltechnologien für moderne datengetriebene Computersysteme. Datenbanksysteme sind dabei in einer einzigartigen Position, um die wachsenden Hardwareressourcen zu nutzen und die Datenverarbeitung zu verbessern. Aufgrund der langen Geschichte der Datenbanksysteme müssen moderne Systeme die traditionellen Datenverarbeitungsprozesse neu strukturieren, um die neue Hardware optimal zu nutzen und die Latenz der Anfragen zu verbessern. Um niedrige Latenzen zu erreichen, müssen Systeme nicht nur den Datendurchsatz durch neue Hardware beschleunigen, sondern auch die Anfrageplanung anpassen, um eine niedrige Gesamtlatenz zu erreichen.

Im ersten Teil dieser Arbeit diskutieren wir die parallele Datenverarbeitung mit gemeinsam bearbeitetem Zustand. Insbesondere wird untersucht, wie sich die Parallelität auf die Wahl der Algorithmen und die Anfragebearbeitung im System auswirkt. Wir entwerfen neue Auswertungsstrategien für Groupjoins, die Daten gleichzeitig verknüpfen und aggregieren, ohne unnötig Zustände zu duplizieren. Darüber hinaus diskutieren wir die Herausforderungen der Kardinalitätsschätzung für die Anfrageoptimierung bei komplexen berechneten Ausdrücken und entwickeln eine neue Technik zur Schätzung solcher Ausdrücke. Zusammen mit der parallelen Auswertung von Groupjoins verbessert dies die Geschwindigkeit von analytischen Anfragen, die Daten aggregieren.

Zweitens untersuchen wir die Latenz der Anfrageplanung. Datenbanksysteme planen zunächst die Auswertung der Anfrage, bevor sie die Daten nach diesem Plan verarbeiten, wobei die Verarbeitung lange Zeit der begrenzende Faktor war. Durch schnellere Hardware wird die Planung jedoch zu einem immer größeren Anteil der Gesamtlatenz, was die relativ schlechte Leistung der etablierten Planungstechniken aufzeigt. In dieser Arbeit entwickeln wir Indexed Algebra, eine dynamische Indexstruktur für relationale Algebra, als eine neue Technik, mit der wir die Zeit für die Planung einer Anfrage asymptotisch reduzieren.

Diese Arbeit schließt mit einer Diskussion unserer Ergebnisse und einem Ausblick auf zukünftige Forschung. Als Ergebnis dieser Dissertation können Daten in vielen Fällen mit geringerer Latenz verarbeitet werden, es jedoch weiterhin Verbesserungspotential gibt. Zum Beispiel ermöglichen moderne SSDs eine schnelle Datenverarbeitung von Daten, die nicht im Hauptspeicher liegen, und wir sehen Potenzial die Latenz von Anfragen durch adaptive Rekompilierung von Plänen zu reduzieren.

Preface

Excerpts of this thesis were published in advance.

Chapter 2 was published as:

[44] Philipp Fent, Altan Birlir, and Thomas Neumann. “Practical Planning and Execution of Groupjoin and Nested Aggregates”. In: *VLDB J.* 32 (2023), pp. 1165–1190

[47] Philipp Fent and Thomas Neumann. “A Practical Approach to Groupjoin and Nested Aggregates”. In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2383–2396.

Chapter 3 was published as:

[46] Philipp Fent, Guido Moerkotte, and Thomas Neumann. “Asymptotically Better Query Optimization Using Indexed Algebra”. In: *Proc. VLDB Endow.* 16.11 (2023), pp. 3018–3030.

Furthermore, the author of this thesis also contributed to the following publications, which are not part of this thesis:

[48] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. “Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory”. In: *ICDE*. IEEE, 2020, pp. 1477–1488.

[45] Philipp Fent, Michael Jungmair, Andreas Kipf, and Thomas Neumann. “START - Self-Tuning Adaptive Radix Tree”. In: *ICDE Workshops*. IEEE, 2020, pp. 147–153.

[43] Martin Eppert, Philipp Fent, and Thomas Neumann. “A Tailored Regression for Learned Indexes: Logarithmic Error Regression”. In: *aiDM@SIGMOD*. ACM, 2021, pp. 9–15.

[123] Magdalena Pröbstl, Philipp Fent, Maximilian E. Schüle, Moritzichert, Thomas Neumann, and Alfons Kemper. “One Buffer Manager to Rule Them All: Using Distributed Memory with Cache Coherence over RDMA”. in: *ADMS-*

@VLDB. 2021, pp. 17–26.

[101] Leonard von Merzljak, Philipp Fent, Thomas Neumann, and Jana Giceva. “What Are You Waiting For? Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth!” In: *ADMS@VLDB. 2022*, pp. 36–46.

[131] Tobias Schmidt, Philipp Fent, and Thomas Neumann. “Efficiently Compiling Dynamic Code for Adaptive Query Processing”. In: *ADMS@VLDB. 2022*, pp. 11–22.

[129] Adrian Riedl, Philipp Fent, Maximilian Bandle, and Thomas Neumann. “Exploiting Code Generation for Efficient LIKE Pattern Matching”. In: *ADMS@VLDB. 2023*.

Contents

Preface	i
1 Introduction	1
1.1 Parallel Processing with Shared State	3
1.2 Cardinality Estimation for Complex Calculated Expressions . .	4
1.3 Low-latency Query Optimization	5
1.4 Research Questions	6
2 Groupjoins	7
2.1 Introduction	7
2.2 Groupjoin for Nested Aggregates	9
2.2.1 Groupjoin	10
2.2.2 Correlated Subquery Unnesting	10
2.3 Parallel Execution of Groupjoins	11
2.3.1 Eager Right Groupjoin	12
2.3.2 Memoizing Groupjoin	15
2.3.3 Separating Join and Group-By	16
2.3.4 Using Indexes for Groupjoins	18
2.3.5 Choosing a Physical Implementation	21
2.4 Evaluation of Groupjoins	22
2.5 Aggregate Estimates	28
2.5.1 Skew Normal Distribution	29
2.5.2 Transformations on the Skew Normal	32
2.5.3 Aggregate Estimation	33
2.6 Evaluation of Estimates	35
2.7 Planning with Groupjoins	39
2.7.1 Eagerly Introducing Groupjoins	39
2.7.2 Additional Groupjoins in TPC-H	41
2.8 Eager Aggregation	43
2.8.1 Placement of Preaggregation	45
2.8.2 Cardinality Estimation Strategy for Preaggregation . . .	48

2.8.3	Integrating Eager Aggregation into an Optimizer	50
2.8.4	Evaluation of Eager Aggregation	52
2.9	Groupjoins in Detail	53
2.10	Related Work	58
2.11	Conclusion	59
3	Indexed Algebra	61
3.1	Introduction	61
3.2	Query Representation	63
3.2.1	Algebra: Operators, Expressions, and IUs	64
3.2.2	Efficiently Navigating Algebra	64
3.2.3	Reasoning about Column Sets	65
3.2.4	Reasoning by Path Traversal	66
3.3	Indexing the Algebra	67
3.3.1	Simple Tree Indexes	67
3.3.2	Path Labeling	69
3.4	Implementing Indexed Algebra	69
3.4.1	Link/Cut Trees	71
3.4.2	Efficient Operations using the Link/Cut Tree	72
3.5	Applications in Query Optimization	74
3.5.1	Determining Join Graph Edges	74
3.5.2	Detecting Dependent Joins	75
3.5.3	Tracking IU Nullability	76
3.5.4	Predicate Pushdown	77
3.5.5	Propagating Constants	78
3.5.6	Bounding Distinct Values Estimates	79
3.5.7	Placing Expression Evaluation	80
3.6	Beyond Indexed Algebra	82
3.6.1	Complex Expressions	82
3.6.2	Lazy Property Evaluation	83
3.6.3	DAG Structured Algebra	83
3.7	Efficiently Representing Column Sets	85
3.8	Evaluation	88
3.8.1	Efficiency on Query Complexity	89
3.8.2	Benchmarks	91
3.8.3	Interactive Workloads	93
3.8.4	Overall Results	94
3.9	Related Work	95
3.10	Conclusion	96
4	Conclusions	99

CONTENTS

v

Bibliography

103

List of Figures

1.1	Read throughput of Samsung storage drives.	2
2.1	Missing components for practical groupjoins. Our improvements to estimation and parallel execution enable efficient evaluation of queries with nested aggregates.	9
2.2	Preconditions to introduce a groupjoin.	10
2.3	General Unnesting: Decorrelation of dependent subqueries containing an aggregation can introduce a groupjoin.	11
2.4	Single Threaded Groupjoin Hash Table. Aggregates from either join side are materialized as hash table payload.	12
2.5	Eager Grouping. While the middle groupjoin eliminates the second hash table, the schematic eager aggregation on the right can additionally eliminate the result scan.	13
2.6	Memory consumption of TPC-H SF 10 orders - lineitem groupjoins.	22
2.7	Performance of the index groupjoin on TPC-H SF 10 orders - lineitem via the l_orderkey foreign key.	23
2.8	Comparison of fastest and cost model recommended implementation for a TPC-H orders - lineitem groupjoin.	24
2.9	Peak memory usage for a TPC-H orders - lineitem groupjoin. . .	24
2.10	Parallel scale-out of TPC-H SF10 groupjoin queries.	26
2.11	Data size scale-out of groupjoins in TPC-H.	27
2.12	Possible query plans for TPC-H Query 18. Depending on the σ filterselectivity, we use the customer relation as hash-join build or probe side, which roughly leads to a 10 % difference in performance.	28
2.13	Skew-Normal Fit. Histograms of several data sets, ranging from uniform synthetic to skewed real-world data sets. The overlaid red distribution is a fitted skew-normal distribution.	30
2.14	Skew-Normal Fit of Aggregates. The first column shows three different distributions of base column and group size. The next three columns compare simulated aggregates with a calculated fit of our skew-normal estimator.	34
2.15	Estimates for TPC-H Q18 style subqueries.	36

2.16	Estimation quality of aggregation queries. The box plots show the log-scale q-error of our estimates in comparison to the static selectivity of Hyper. Our skew-normal model reduces the geomean q-error by 46 % from 45.8 to 24.7.	38
2.17	Overall Impact on TPC-H and TPC-DS.	39
2.18	Physical planning of groupjoin operators depends on join ordering. A purely opportunistic approach misses non-trivial combinations.	40
2.19	Intermediary planned groupjoin for TPC-H Q2.	41
2.20	Final plan for TPC-H Q2.	42
2.21	Groupjoin query plan for TPC-H Q20.	43
2.22	Eager Aggregation of TPC-DS Q22.	44
2.23	Possible plans using the potential preaggregation placements.	46
2.24	Potential preaggregation placements.	47
2.25	Four alternative plans considered by the optimizer. All costs contain the cost of a preaggregation on the join.	51
2.26	Overall Impact of Eager Aggregation on TPC-DS.	53
2.27	Query plan for TPC-H Q3.	54
2.28	Operator Trace of TPC-H Q3.	55
2.29	Query plan for TPC-H Q13.	56
2.30	Query plan for TPC-H Q17.	56
2.31	Query plan for TPC-H Q18.	57
3.1	Relation algebra tree with subtle data flow. In this chapter, we optimize queries by efficiently analyzing data flow.	62
3.2	Our algebra representation interlinks operators, expressions, and IUs to allow efficient navigation.	65
3.3	Traditional predicate pushdown traverses the query tree operator by operator. Path-centric optimization can take a shortcut to the IU's source, but still needs to check if the path is sound, or if it contains e.g., an outer join.	66
3.4	A binary search tree keyed on the distance to the root (annotated in superscript) allows efficient path queries on static algebra trees.	68
3.5	Partial path indexes of Figure 3.4. Link/cut trees build dynamically balanced splay trees over paths through the algebra and connect subtrees via path-parent pointers.	70
3.6	Intrusive structure of link/cut indexed operators.	72
3.7	Algebra indexes efficiently determine if a path contains an outer join by propagating a marker through the balanced auxiliary tree.	74
3.8	The nullability of IUs can depend on outer joins and their position in the algebra.	77

3.9	SQL query with nested constants. We propagate and fold constants from inner queries to all uses to enable transitive pushdown.	79
3.10	The cardinality of the top aggregation Γ depends on the distinct values reaching it from the base table. Indexed Algebra maintains a path aggregate to efficiently maintain this information.	80
3.11	Materialization points allow evaluating expressions that reduce the size of the materialized data. We use Indexed Algebra's LCA and path operations to efficiently find suitable materializations points.	81
3.12	A DAG structured query plan. Operators can be referenced multiple times, but need to rename their IUs to avoid ambiguity.	84
3.13	Two copy-on-write column sets share the striped nodes and only duplicate a logarithmic number of nodes when inserting new IUs.	85
3.14	Microbenchmark of column set operations on a left-deep join tree with an increasing number of joins.	86
3.15	Query optimization time of synthetic join queries with many relations. Indexed Algebra has asymptotically better runtime for large queries.	90
3.16	Query optimization time of various benchmarks.	91
3.17	Comparison of total optimization times.	92
3.18	Evaluation of interactive workloads.	93
3.19	Improvements of total query optimization time. We compare the time to optimize queries using Indexed Algebra in contrast to column sets. The average improvements are: 12% for TPC-H, 29% for TPC-DS, and 10% for JOB.	94
3.20	Total time spent processing synthetic join queries with many relations. Umbra uses Indexed Algebra to efficiently optimize large queries.	95

List of Tables

2.1	Static count semantics. In separate operators count (*) aggregates might produce different results.	18
2.2	Example cost calculations of groupjoin implementations.	21
2.3	TPC-H Groupjoins. Cost model calculations with four TPC-H groupjoin queries on scale factor 1.	25
3.1	Complexity of operations on relational algebra.	69
3.2	Impact of Indexed Algebra on TPC-DS optimization.	90

CHAPTER 1

Introduction

Database systems are a central component for the access and management of data. Already in the seventies, systems to manage databases emerged, most prominently System R [4, 17]. These early systems spent most of the time shuffling data from disk to main memory, since they could only hold a couple of pages in-memory [56]. Nowadays, this picture has completely changed and processing mostly in-memory is possible and common [3]. Not only main memory grew, but CPUs also increasingly have large on-die caches, which can fit many lookup data structures, without accessing the relatively slow main memory.

This change in hardware capabilities has also impacted database systems design. Data processing algorithms need to be adapted to use the fast hardware that can process large amounts of data with low latency. Specifically, they need to be efficient not only for I/O, but also for main memory operations.

Not only can modern systems process much data in main memory or caches, but storage and I/O is also fast now. Fast storage, especially in the form of SSDs, is becoming mainstream and cheaply available. Figure 1.1 shows the performance trend of selected Samsung storage drives. This plot starts in 2010 with HDDs with a read throughput of around 100 MB/s, to current PCIe 4 NVMe SSDs that have a read throughput of around 7 GB/s. With already a handful of devices in parallel, I/O performance that rivals main memory speeds is cheaply achievable. With such fast data access, many analytic workloads do not need to be batch processed overnight anymore, but can be evaluated interactively with much lower latency. Even when data resides on storage, the processing bottleneck is not the I/O operation anymore, but it shifts from to efficiently processing the data.

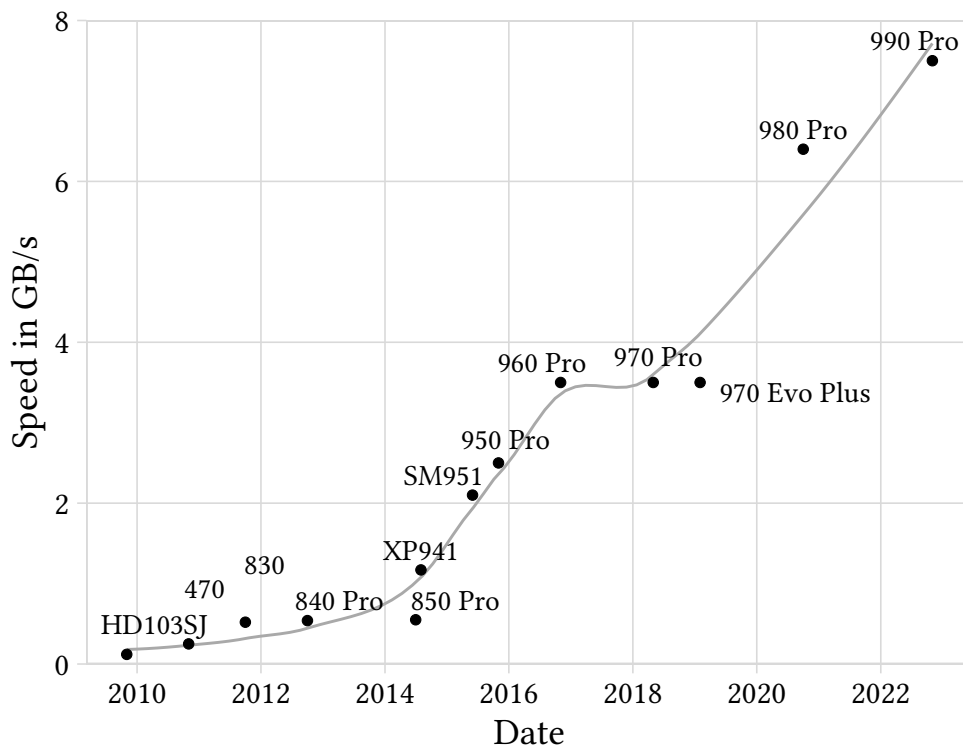


Figure 1.1: Read throughput of Samsung storage drives.

Modern database systems redesign the execution engine to process in-memory data more efficiently [109]. For modern engines, two major modern designs emerged: Vectorization, as pioneered by MonetDB [21, 68], or data-centric code generation, as pioneered by HyPer [72, 108]. Both approaches are successful in increasing the efficiency and significantly accelerating the in-memory performance [74]. For efficient base table scans, vectorization has some advantage, but for computationally-intensive queries with complex expressions, code generation and data-centric query execution generalizes better and produces more efficient execution pipelines. However, these new execution models only set the foundation for a new push for efficiency. In this thesis, we work on three problems that arise for modern execution engines and implement and evaluate them in the database system Umbra.

1. Parallel processing with shared state
2. Cardinality estimation for complex calculated expressions
3. Low-latency query optimization

Umbra already has several advanced techniques that enable low-latency query processing, including low-latency code generation [59, 75, 81, 84], query optimization [15, 16, 116], and specialized data processing operators [10, 36, 49, 82, 128, 133, 138, 152, 153]. Using this system, we have a state-of-the-art baseline that supports a wide variety of benchmarks and use-cases and thus allows to cover the multifaceted workloads of modern database systems. With an implementation in Umbra, this thesis is uniquely situated to demonstrate that these topics enable more efficient low-latency query planning and processing on modern hardware.

1.1 Parallel Processing with Shared State

In contrast to I/O bandwidth, processor performance did not increase as easily in the last ten years. For a long time denser CPU transistors lead to an increase in frequency and instructions per cycle (IPC). In contrast, modern CPU development switched to integrating more and more parallel execution units, but with relatively stagnating throughput per core [89]. As a result, large server CPUs now contain in the order of a hundred cores, which requires parallel data processing algorithms to use all cores.

In general, modern analytical data processing systems are compute constrained. As we saw in the last section, a large amount of data can fit into main memory or SSDs with huge I/O throughput. Modern many-core processors offer vast improvements in processing power to keep up with the increase in I/O bandwidth. Hence, modern data processing algorithms thus need to support efficient parallel execution. However, this is aggravated by Amdahl's law [2], which means that even small parts of an algorithm that needs single-threaded execution limits the scalability of the complete algorithm.

Strategies for parallel query execution, e.g., morsel-driven parallelism, work well when they can avoid the need for synchronization and thus potentially single-threaded phases. However, when the query execution needs data structures with shared state, e.g., for aggregations, this inherently leads to contention, which limits the parallel throughput.

The potential upside of parallel algorithms cannot be understated. When we have a fully parallel algorithm, the algorithm is in another quality class. To illustrate this, let us compare it with the traditional asymptotic complexity of algorithms. As a hypothetical problem, take an algorithm that processes data in $O(n)$, but can only process single-threaded, and a fully parallel algorithm that is in $O(n \log_2(n))$.

For the traditional asymptotic analysis, the single-threaded algorithm is superior. However, for practical applications where $\log_2(n) < \text{\#cores}$ the parallel

algorithm will be much faster. For the example, assume the constant factors of the algorithms are the same. Then, take consider some recent hardware, an AMD EPYC Genoa processors with 96 cores and storage of 16×8 TB SSDs. For the algorithms, n is bounded by the data size in bytes, i.e., $n \leq 2^{47}$. With these assumptions, we can estimate the speedup of the parallel execution as follows:

$$\frac{96}{\log_2(2^{47})} = \frac{96}{47} \approx 2$$

Parallel data processing algorithms thus promise a significant speedup, even if they have some overhead over sequential processing. However, algorithms often have data structures with shared state, where updating this state in parallel creates contention, which limits the parallel processing. In Chapter 2, we focus on groupjoins, which deduplicate state for hash joins and hash aggregations over the same key. However, this creates shared state in the hash table that needs to be updated in parallel. In this thesis, we develop novel techniques to adapt and improve groupjoin to scale to parallel processing that avoid the data hazards that go along with the traditional approach.

1.2 Cardinality Estimation for Complex Calculated Expressions

Finding the optimal query plan is important for the performance of analytical queries. Database systems use cardinality and selectivity estimations to do cost-based optimization of the query execution plan. For example, reducing the size of intermediate results as early as possible to avoid processing tuples that get filtered out later in the pipeline. To compare predicates and join conditions, database systems estimate the selectivity and result cardinality of these operations.

Usually, we can calculate these estimates based on statistics over the stored relations. E.g., maintain helper data structures like histograms or sketches. Additionally, keep track of a sample and evaluate predicates on that sample. This allows accurate estimations of most common predicates.

Where this stops working is for calculated expressions, e.g., for a sum over one or multiple columns. The usual statistics of distinct-counts or value-ranges only help marginally for this purpose. Instead, for such numerical operations, we need more type specific information, i.e., proper numerical statistics that allow to estimate numerical operations. In combination with distinct-count estimates, which allow estimating group sizes, we can also get estimates of calculated expressions.

For expressions that can't be estimated using the standard techniques, systems often fall back to naïve heuristics or even to magic constants. However, these fallbacks are based on the assumption of no prior knowledge of the underlying data. For example, for a comparison $a < b$, we assume that both are random variables on the entire domain. Then, an unbiased estimate for the selectivity, is 0.5, since of all values, both have equal probability of being that value. However, for most predicates the assumption that both sides are in the same domain is violated. Consider the predicate $a + b < 10$ in contrast to $a + b < 1\,000$. Most likely, their selectivity differs by over a magnitude, but with no prior knowledge of the domains, the estimation of these predicates will be severely off, which potentially disastrous consequences for query plan quality.

Thus, we need a model of the underlying data distribution, which allows us to estimate complex calculated expressions, and takes into account as much information as we can gather from the database. In Chapter 2, we describe a strategy to estimate the distribution of calculated numeric columns with a focus on the result of aggregates. These statistical estimations then allow cheaply estimating most expressions.

1.3 Low-latency Query Optimization

Both previous techniques work towards reducing the query execution time. With these techniques in place, database systems can process large amounts of data with lower result latency. However, the pure improvement in execution performance has diminishing returns the lower the processing time is, when other parts of the system take a relatively longer time. In Umbra, we first parse the query, then optimize the relational algebra, before we compile and execute it.

Overall, parsing SQL is quick, and compiling the execution plan to machine code can be sped up by existing techniques. Query optimization, on the other hand, can take a longer, and with sufficiently complex queries can take several hundred milliseconds. For low-latency processing, this is prohibitively expensive. While there are workarounds like prepared statements or plan caching, these are complex to deploy and do not generalize to ad-hoc queries, which actually need more efficient planning.

In Chapter 3, we re-think the query optimization approach to improve its efficiency for low-latency operation. We identify several cases where the traditional algorithms are expensive and replace them with a more efficient system that uses indexes of the algebra.

1.4 Research Questions

To summarize, we see three research questions that we attempt to answer in this dissertation:

1. *How can we design algorithms with shared state to benefit from parallel processing?*
2. *How to include calculated expressions and especially aggregates in query result cardinality estimation?*
3. *How can we redesign the query optimizer to improve latency?*

CHAPTER 2

Practical Planning and Execution of Groupjoin and Nested Aggregates

Parts of this chapter have been previously published in the Proceedings of the VLDB Endowment [47] and the International Journal on Very Large Data Bases [44]. With contributions from Altan Birlir.

2.1 Introduction

Joins and aggregations are the backbone of query engines. A common query pattern, which we observe in many benchmarks [20, 107] and industry applications [150], is a join with grouped aggregation on the same key:

```
SELECT cust.id, COUNT(*), SUM(s.value)
FROM customer cust, sales s
WHERE cust.id = s.c_id
GROUP BY cust.id
```

In a traditional implementation, we answer the query by building two hash tables on the same key, one for the hash join and one for the group-by. However, we can speed up this query by reusing the join's hash table to also store the aggregate values. This combined execution of join and group-by is called a *groupjoin* [103].

The primary reason to use a groupjoin is its performance. We spend less time building hash tables, use less memory, and improve the responsiveness of this query. However, groupjoins are also more capable than regular group-bys, as we can create the groups explicitly. Consider the following nested query, with subtly different semantics:

```

SELECT cust.id, cnt, s
FROM customer cust, (
  SELECT COUNT(*) AS cnt, SUM(s.value) AS s
  FROM sales s
  WHERE cust.id = s.c_id
)

```

Here, nested the query calculates a `COUNT(*)` over the inner table, which evaluates to zero when there are no join partners. Answering that query without nested-loop evaluation of the inner query is tricky, as a regular join plus group-by will produce wrong results for empty subqueries, which is known as the `COUNT` bug [106]. A groupjoin directly supports such queries by evaluating the static aggregate for the nested side of the join, taking the groups from the other side.

Despite their benefits, groupjoins are not widely in use. We identify two problems and propose solutions that make groupjoins more practical: First, existing algorithms for groupjoins do not scale well for parallel execution. Since groupjoin hash tables contain shared aggregation state, parallel updates of these need synchronization, and can cause heavy memory contention. Furthermore, current estimation techniques deal poorly with results of groupjoins from unnested aggregates.

The unnesting of inner aggregation subqueries is very profitable, since it eliminates nested-loops evaluation and improves the asymptotic complexity of the query. However, this causes the aggregates to be part of a bigger query tree, mangled between joins, predicates and other relational operators. Query optimization, specifically join ordering, depends on the quality of cardinality and selectivity estimates [94]. With unnested aggregates, the estimation includes group-by operations and aggregates, which are notoriously hard [50, 78]. Consider the following nested aggregate with a predicate:

```

SELECT ... GROUP BY x HAVING SUM(value) > 100

```

The result might have vastly different cardinality, depending on the selectivity, which in turn influences the optimal execution order of the query.

In this chapter, we work on techniques that make combined join and aggregation more efficient, e.g., with eager aggregation [130, 154] and hash table sharing via groupjoins [40, 103]. In addition, we propose a novel estimation framework for computed aggregate columns, which improves the plan quality with nested aggregates. We introduce this here as part of our work in groupjoins, but the estimation framework is useful for queries with regular group-by operators, too. We integrate our work in the high-performance compiling query engine of our research database system Umbra [113]. Figure 2.1 shows a high-level overview

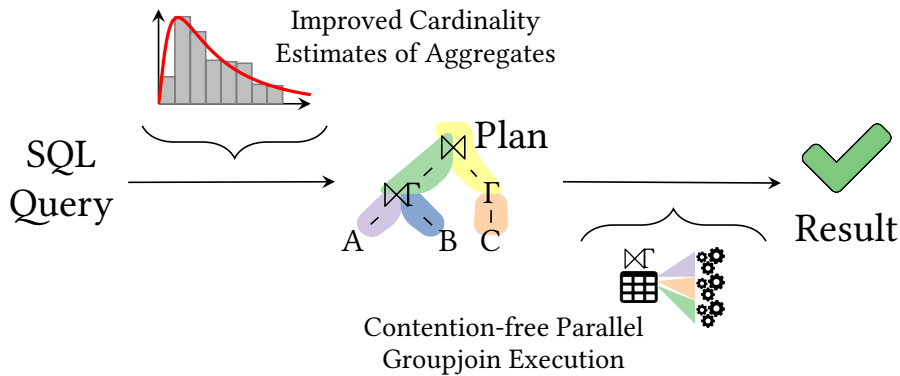


Figure 2.1: Missing components for practical groupjoins. Our improvements to estimation and parallel execution enable efficient evaluation of queries with nested aggregates.

of our query optimizer. On the way from an SQL query from a relational algebra query plan to the query result, we focus on efficiently evaluating nested aggregates with *computed column estimates* and *parallel groupjoin execution*.

The rest of this chapter is structured as follows: First, we introduce the groupjoin and its use in general unnesting in Section 2.2. Then, we discuss and evaluate four parallel groupjoin execution strategies in Section 2.3, and propose a cost model to choose the optimal execution strategy. We evaluate the execution strategies and our cost model based on the well-known TPC-H and TPC-DS benchmarks in Section 2.4. Afterwards, we introduce our estimations for computed columns in Section 2.5 and evaluate them in Section 2.6. Furthermore, in Section 2.7 and Section 2.8, we improve query plans by considering groupjoins for operator ordering and propose an eager aggregation strategy. Section 2.9 discusses the impact of our work on queries from TPC-H, before we discuss related work in Section 2.10, and conclude in Section 2.11.

2.2 Groupjoin for Nested Aggregates

Apart from better performance, the semantics of groupjoins are useful to compute nested aggregates. Due to the versatile subqueries in SQL, aggregates can appear in various places of the query plan. To efficiently calculate such aggregates, it is important to unnest and not evaluate them in nested-loops [14, 65, 114]. However, decorrelated aggregates need a careful implementation and are challenging for query planning.

2.2.1 Groupjoin

We define a groupjoin \bowtie [103] as an equi-join with separate aggregates over its binary inputs grouped by the join key.

$$R \bowtie_{a_1 = a_2} \text{agg } S := \{r \circ [g_r : G_R] \circ [g_s : G_S] \mid r \in R, \\ G_S = \text{agg}(\{s \mid s \in S \wedge r.a_1 = s.a_2\}), \\ G_R = \text{agg}(\{r \mid r \in R \wedge r.a_1 = s.a_2\})\}$$

We further require that $a_1 \rightarrow R$, i.e., that the join condition functionally determines R. With this definition, we compute the left-outer join between R and S on a key of R, and compute aggregates separately over the matching tuples. We also generalize groupjoins to inner-join semantics, which is beneficial to avoid duplicating tuples of R and building a duplicate hash table in more cases.

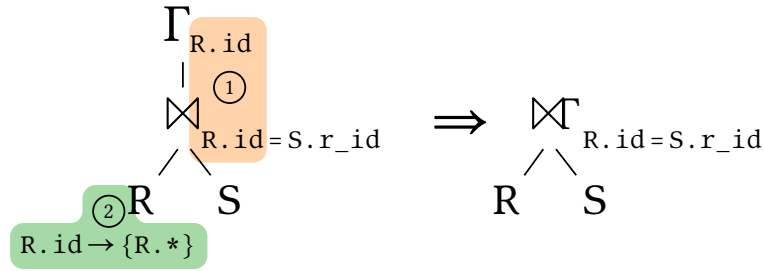


Figure 2.2: Preconditions to introduce a groupjoin.

The intuitive use-case for groupjoins is an optimization to fuse a join and a group-by operator, given that the preconditions shown in Figure 2.2 are satisfied, and we can separately evaluate the aggregates: ① The join and aggregation keys need to be equivalent, and ② these keys are superkeys w.r.t. functional dependencies of the left build side. In this case, introducing a groupjoin is usually considered to be a net win [27, 40] and can reduce the cost of those operators by up to 50 % by eliminating intermediate results.

2.2.2 Correlated Subquery Unnesting

The groupjoin also supports the challenging edge cases of whole table aggregates in a correlated subquery. Consider the correlated subquery from Section 2.1, where we calculate a whole-table $\text{COUNT}(\ast)$ on sales that is correlated with the outer query's customer. Conceptually, we need to calculate a whole table aggregate for each customer, but ideally want to introduce a more efficient join. However, using an outer join is tricky, since we cannot directly translate whole

table aggregates to the join result. A groupjoin can instead evaluate the left and right sides separately, where a careful initialization can produce equivalent results to whole table aggregates. For the $\text{COUNT}(\ast)$ example, we initialize empty groups (e.g., customers with no sales) as zero, and increment it with whole-table tuple counting logic¹.

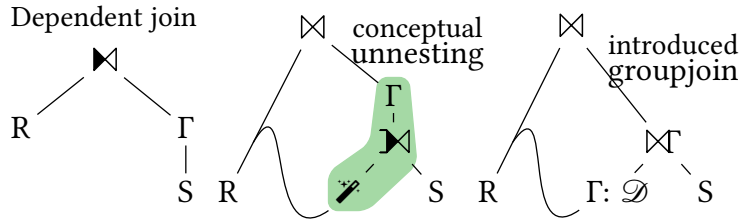


Figure 2.3: General Unnesting: Decorrelation of dependent subqueries containing an aggregation can introduce a groupjoin.

For the general case, we deliberately introduce a groupjoin to separately calculate the aggregates of the correlated subquery, filter unnecessary tuples, and avoid the COUNT bug [114]. Figure 2.3 shows this unnesting for two arbitrary tables R and S , with the dependent subquery-join \bowtie on top and a nested whole table aggregate Γ in the correlated right subtree. To decorrelate this aggregate, we first compute the magic set \mathcal{M} of relevant tuples for the correlated subquery [135]. To compute the set, we eliminate any duplicates of the outer-side join key with a group-by Γ and get the precise domain \mathcal{D} of potentially equivalent keys for which we need to calculate the inner aggregate. With this condensed set of outer keys, we satisfy both preconditions to introduce a groupjoin, which we use to keep the aggregation of the subquery side S separate.

In the following, we parallelize groupjoins with on-the-fly adaptive data segmentation into morsels and contention-avoiding relational operators that allow dynamic work-stealing.

2.3 Parallel Execution of Groupjoins

The parallel execution of common relational operators is widely studied and efficient parallel join and aggregation algorithms are used in many systems that can scale analytical workloads [25, 76, 115]. Groupjoins, which fuse join with aggregation hash tables, promise a significant speedup in comparison to separate operators and are necessary for general unnesting. However, parallel

¹ $\text{COUNT}(\ast)$ has some edge cases that are trivial in a groupjoin, but difficult in separate operators. See Section 2.3.3 for an extended discussion.

execution of groupjoins can be a bottleneck due to contention. While several publications have previously discussed groupjoins, they are now well over a decade old and single-threaded [23, 26, 99].

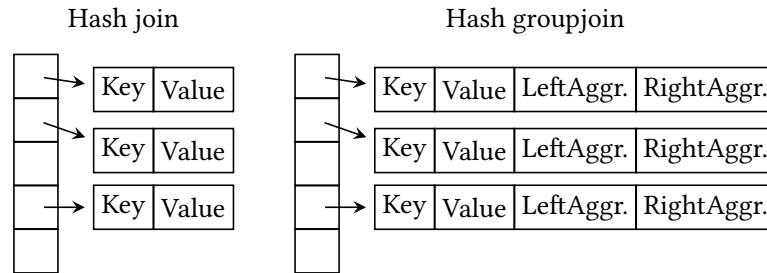


Figure 2.4: Single Threaded Groupjoin Hash Table. Aggregates from either join side are materialized as hash table payload.

Figure 2.4 shows a basic, single-threaded implementation of a groupjoin, and its similarity to a regular hash join. In this example, we use a hash table to store the hash table payload, which includes the accumulators for the aggregates of both sides. During the build phase, we initialize these as empty to support the semantics of static (whole table) aggregation.

In contrast to joins, the probe phase of groupjoins is not read-only, but needs synchronization of the aggregate updates when using more than one thread. The shared state of the aggregates poses a problem for parallel execution, and we need synchronization, e.g., with fine-grained locking, to avoid data races. Unsurprisingly, the synchronization overhead can quickly become a bottleneck, especially in the presence of heavy-hitters [122]. While updating the aggregates is generally a quick operation, and the critical section only spans a couple of instructions, all threads will compete for the same locks of the heavy-hitters. Even when eliding this lock and updating the aggregates with lock-free atomic instructions, memory contention, which is the root-cause for this bottleneck, still remains a problem and causes suboptimal performance.

In the following, we propose three execution strategies for groupjoins that avoid synchronization between threads. For each implementation, we discuss, in which scenarios it is an efficient implementation of a groupjoin. Based on these insights, we propose a cost-based strategy in Section 2.3.5, to choose the best physical plan, depending on the underlying data distribution.

2.3.1 Eager Right Groupjoin

One well-known technique of aggregation queries is eager aggregation [154]. A group-by can be pushed down, past a join, to reduce the number of input

tuples to the join. In the general case, this needs an additional group-by after the join, since the join might have a multiplying effect on the aggregate tuples. In this section, we apply eager aggregation to groupjoins: When we can speculate that almost every tuple finds a join partner, i.e., the relative-right selectivity $\sigma_S = |R \bowtie S| \cdot |S|^{-1}$ is close to 1, then eager aggregation will substantially reduce the number of tuples that need to be processed by the join.

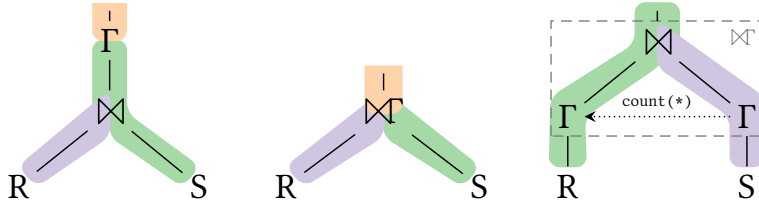


Figure 2.5: Eager Grouping. While the middle groupjoin eliminates the second hash table, the schematic eager aggregation on the right can additionally eliminate the result scan.

When eagerly aggregating in a groupjoin, we can exploit several facts that allow making eager aggregation very efficient: Precondition ② (cf. Section 2.2.1) of the groupjoin guarantees that the join and group key of the left-hand side functionally determine the left tuples. In other words, the left side does not contain duplicates and, thus, cannot have a multiplying effect on the right aggregation. As sketched in Figure 2.5, we can exploit this fact by first eagerly executing the right aggregation. If there are any aggregates on attributes of R, duplicate keys in S have a multiplying effect that duplicates the keys but do not change their value. We account for this effect with a `count(*)` aggregate on S, which we apply as multiplication factor of the unique tuples of R. In result, we elide the final group-by that would be needed for general eager aggregation as described by Yan and Larson [154], and replace the result scan with a single hash table probe.

We eliminate the result scan, and improve the pipeline behavior, by using the same precondition ②. A group-by is a *full pipeline breaker* [108], i.e., it materializes all incoming tuples and scans the result when the last tuple was processed. However, this flushes all data from CPU registers, or very hot cache, which makes pipeline breakers expensive. Algorithm 1 shows pseudocode to execute this operator, where each loop represents one pipeline. In the first loop, we eagerly aggregate the whole right side S into the aggregation hash table Γ_S . The second loop probes with the left side R and calculates the complete left aggregate in a single step with the probe result. Afterwards, the loop still is not

Algorithm 1: Example code generated to execute an eager right group-join with inner-join semantics.

```

initialize memory of  $\Gamma_s$ 
for each tuple  $s$  in  $S$ 
    aggregate  $s$  as  $a_s$  in hash table of  $\Gamma_s$ 
for each tuple  $r$  in  $R$ 
    if  $r$  has match  $a_s$  in  $\Gamma_s$  # inner groupjoin
         $a_r := \text{agg}(r * a_s.\text{count}(*))$ 
        output:  $r \circ a_r \circ a_s$ 

```

terminated, but can continue its pipeline with any next operations, in this case output.

In contrast to a lazily aggregated groupjoin, eager aggregation requires no explicit synchronization through locks. Our implementation reuses the implementation of regular aggregation, which first builds partitioned, thread-local aggregation hash tables [83, 90, 127]. A second step exchanges these partitions between threads and merges them into a partitioned global result hash table. Afterwards, the duplicate-free left side can exclusively read its matches in the hash table, which allows contention-free and full parallel execution.

While it can be executed very efficiently, eager aggregation is no one-size-fits-all solution. Depending on the relative right selectivity of the join part, i.e., how many groups of the right-hand side are not matched by the left, we might calculate many unneeded aggregates. Therefore, we deem it necessary to only use this eager aggregation, when a local cost-model predicts it to be beneficial.

The following cost function models the eager right groupjoin and closely follows the presented algorithm:

$$C_{\text{eager}} = |S| + |R \times S|$$

First, we build the eager hash aggregation in two passes over the data, which touches every tuple of S twice: $2|S|$. Then, we probe the hash table with the left-hand side $|R|$, and check the matching tuples $|R \times S|$, for equality. In our cost function C_{eager} , we exclude the initial passes over each input side $|R| + |S|$, which are required by any groupjoin implementation. Nevertheless, we include the result scanning phase of pipeline breakers, to differentiate operator-fused pipelines that do not need to materialize their result.

2.3.2 Memoizing Groupjoin

Eagerly aggregating is very beneficial, when almost every right tuple finds a join partner. The other extreme is also common, i.e., that many tuples are filtered by the join. In this case, we want to filter right-hand side tuples, before aggregating them. In the following, we present a groupjoin implementation that builds filtered thread-local aggregates and efficiently merges them to a groupjoin result.

The idea of this implementation is to optimistically use a shared global aggregation hash table for aggregates with few tuples, but aggregate heavy-hitters thread-locally. The global hash table resembles the sketched single-threaded groupjoin in Figure 2.4, where we first build a join hash table with the left-hand side R with additional space for the aggregates. For synchronization, we use an atomic set-on-first-use thread-id tag that assigns groups to the first thread that updates it. Additionally, when we probe the hash table with S , we memoize the payload pointer to avoid a duplicate lookup.

Algorithm 2: Memoizing groupjoin probe pipeline with ownership tagging.

```

1 Hashtable globalHt
2 // Omitted: Concurrent build of R hash table
3 thread_local localHt, tid
4 for each tuple s in S
5   hash := hash(s.key)
6   *p := globalHt.probe(hash, s.key)
7   if p not found
8     continue
9   owner := p->tid.atomic_load(relaxed)
10  inPlace := owner == tid;
11  // Is uninitialized?
12  if owner == 0
13    inPlace = p->tid.CAS(owner, tid)
14  if inPlace
15    p->aggregate(s)
16  else
17    localHt[hash, p].aggregate(s)

```

The intent behind this hybrid synchronization strategy is to avoid tiny thread-local groups with very few tuples, while still aggregating heavy-hitters thread-locally. With the thread-id tags, singleton groups, and groups that are

clustered on a single thread, directly use the result hash table, which reduces the size of local hash tables that would later need to be merged again. In effect, this reduces the partial aggregates to the number of threads n , compared to $n + 1$ for full thread-local preaggregation and merging into a global hash table.

Algorithm 2 shows pseudocode for the described groupjoin probe pipeline. The atomic operations here use a memory model akin to the C++ model [18]. For our optimistic synchronization, we use a single atomic compare-and-swap (CAS), which is the only operation that requires memory synchronization. The low-cost relaxed read of the current tag in line 9 does not need synchronization and could read stale data. For correctness, in the sense of being free of data races, this read is not required. However, it is a vital optimization for heavy-hitters, where the CAS synchronization would cause memory contention. Instead, after the initial CAS, any heavy-hitters will not take this branch again and all other operations are either non-atomic or relaxed. In result, this thread-local preaggregation is virtually contention free. Afterwards, when all input data was either aggregated locally or globally, we exchange the local partitions between threads.

In the thread-local aggregation, we reuse previously calculated intermediates. The local hash table lookup reuses the hash of the global hash table lookup, and, instead of comparing the full key for equality, we only check if the pointer of the probe result from line 6 matches. We also store just this pointer in the local tables, which we also use as a shortcut for merging the aggregates. When all probes from R are finished, we merge the thread-local groups by following this memoized probe pointer, which reduces the number of cache misses and avoids a second hash table lookup.

Compared to the eager right groupjoin, this memoizing approach favors small left sides with a selective join. Expressed more formally for our cost model, we use a build of the left hash table in two passes $2|R|$, probe once with the entire right side $|S|$, before checking the matching tuples $|R \times S|$ for equality. Then, we use these to build thread-local aggregates, before merging them into their memoized global bucket, $2|R \times S|$. Since this variant of the groupjoin is a full pipeline breaker, we additionally need to scan the entire $|R|$ hash table to start the next pipeline, while omitting unjoined results. In sum, we arrive at the following cost function:

$$C_{\text{memo}} = 2|R| + 3|R \times S|$$

2.3.3 Separating Join and Group-By

As laid out in Section 2.2, groupjoin has its own semantics that is useful for whole-table aggregates of unnested queries. An alternative to a dedicated operator would be to emulate this behavior with reused join and group-by operators,

which reduces the implementation overhead, but might build duplicate state in two hash tables.

This duplicate state was the reason that previous work [27, 40, 103] considered a groupjoin as unconditionally advantageous to a separate execution. However, a careful analysis of the involved operations shows that there exist cases where a groupjoin is more expensive than a separate inner join followed by a group-by. The intuition behind this somewhat counter-intuitive finding is that the groupjoin result set might be bigger than that of a separate group-by. That is, when the join is selective on the left build side R , then the join-reduced aggregate table will be significantly smaller than the join table. In this case, it is cheaper to probe a separate join table and build a densely populated aggregation table instead of reusing the relatively sparse matches in the join table. In the following, we show how a groupjoin can be rewritten as a join and group-by, while still preserving the static aggregation semantics to unnest arbitrary queries (cf. Section 2.2.2).

While for most groupjoins, the separation into a join and group-by is trivial, the ungrouped whole table aggregations that can appear in correlated subqueries require special care to preserve their semantics, especially with NULL values [26, 145]. We call this special case a *static groupjoin*. Consider the following example of a query that we process with such a static groupjoin:

```
SELECT r.id, cnt FROM R r, (
  SELECT COUNT(*) cnt
  FROM S s
  WHERE r.id IS s.r_id)
```

Our general unnesting resolves the correlated subquery with a groupjoin. The following shows the resulting plan in SQL-like syntax:

```
SELECT r.id, COUNT(S::*) FROM R r
STATIC LEFT GROUP JOIN S s
ON r.id IS s.r_id
```

The important distinction of the static groupjoin is between empty inner tables and NULL values. Table 2.1 shows three cases, where the aggregated count differs: A `count(*)` in a subquery counts any matching tuple, even when its value is NULL. Executing an outer join $R \bowtie S$, produces additional NULL values that need to be ignored by a count of S tuples. However, with a separate aggregation operator, a naïve count cannot distinguish between matches, where NULL IS NULL and padded tuples that did not have a join partner. Even evaluating the aggregates before the join would still require coalescing of NULL aggregates. To execute the join before aggregating, we ensure the correctness of the aggregates with a join marker that decides between ignored and NULL tuples:

Table 2.1: Static count semantics. In separate operators `count(*)` aggregates might produce different results.

$R \bowtie S$	<code>count(*)</code> subquery	<code>count(S)</code> after \bowtie	<code>count(*)</code> before \bowtie
(NULL, NULL)	1	0	1
(1, 1)	1	1	1
(2, NULL)	0	0	NULL

```

SELECT r.id, COUNT(s.joinMarker)
FROM R r LEFT OUTER JOIN (
  SELECT *, TRUE AS joinMarker FROM S
) ON r.id = s.r_id
GROUP BY r.id

```

Rewriting such a groupjoin as `LEFT JOIN` is usually not beneficial for performance, since it fixes the relative left selectivity to one. On the other hand, most groupjoins do not need an outer join, and might be cheaper executed in separate hash tables. For our cost model calculation, we first two-pass build a $2|R|$ hash table, then probe with $|S|$ and match $|R \bowtie S|$ right tuples. With the resulting tuples, we build a separate aggregation table, again in two passes $2|R \bowtie S|$, before we scan the $|R \bowtie S|$ matched aggregation groups. The drawback in comparison to the memoizing approach is that we do not know the size of the aggregation state beforehand. Therefore, we need to additionally check if the aggregate already exists, and dynamically allocate and initialize memory on demand. While this can reduce resource usage for unmatched keys in R , the fine-grained allocations are more expensive per match ($|R \bowtie S|$) than a bulk operation for all keys. In our simplified cost model, we express this as a fixed factor, which we measured empirically as $c = 30\%$ overhead. In total, we arrive at the following cost function:

$$C_{\text{sep}} = |R| + (3 + c)|R \bowtie S| + |R \times S|$$

2.3.4 Using Indexes for Groupjoins

The previous execution strategies are designed to work over arbitrary inputs. That means, we always need to build a data structure to aggregate values during execution. For join processing, one can often use indexes to access a base table relation on one side of the join [134]. Using indexes to support aggregations can also be beneficial when applied properly.

Some DBMSs already use index scans to filter and compute group-by aggregates pipelined. Similarly, we can avoid building a separate aggregation data

structure and use the index for a groupjoin implementation that aggregate the group with little overhead. The idea here is similar to the eager right groupjoins (cf. Section 2.3.1), where we probe the right side aggregation hash table. However, for already existing indexes, we do not have matching aggregates, but need to calculate them during the index probe. We can do this efficiently, since the left side is duplicate free, and we visit all elements of the right group during a regular index probe.

Using indexes is especially fitting for groupjoins, since we operate on a key of the left side. When we have a key and join with a base relation, this usually means that there exists a foreign key constraint with a corresponding index. Thus, we likely already maintain matching indexes for groupjoins and can use them for a more efficient execution.

Using already existing indexes for groupjoins has several advantages: By using the index to find matching join partners, we avoid accessing unrelated tuples, e.g., when the relative right selectivity σ_s is very low. Also, we need a minimal amount of working memory, since we only keep the aggregation state for the current index probe. As a consequence, e.g., when we calculate a single sum, we can keep the aggregate in a dedicated CPU register and get excellent performance.

Algorithm 3: Example code generated to execute an index groupjoin with inner-join semantics.

```

for each tuple  $r$  in  $R$ 
   $a_s := \emptyset$ 
  for each matching tuple  $s$  in  $S.index$ :
    aggregate  $s$  in  $a_s$ 
  if at least one match: # inner groupjoin
     $a_r := \text{agg}(r * a_s.count(*))$ 
    output:  $r \circ a_r \circ a_s$ 

```

Algorithm 3 shows pseudocode to execute such an index groupjoin. The outer loop represents the input pipeline, in this case a table scan, but we can also operate on arbitrary input tuples, e.g., from a join result. For each input tuple, we initialize an empty aggregate a_s , before we probe the index for matching tuples and combine them in this local aggregate. After probing the index, we report the result to the next operator, depending on if we had at least one join partner, or if we need to report static (cf. Section 2.3.3) results.

In contrast to the methods presented in the previous subsections, probing an index has some inherent limitations in parallel execution. While we can probe

the index in parallel with multiple threads, scanning the matching tuples in the index is harder to parallelize. This is especially problematic for heavy hitters, e.g., when a single left tuple finds millions of join partners on the right. Then, it is unattractive to aggregate this heavy hitter single-threaded.

In this case, it is advantageous to split the equality ranges into schedulable morsels and use multiple threads to aggregate in thread-local storage. This, however, loses the pipelining benefits of the index-based groupjoin operation and effectively executes the groupjoin separately. Defending against this case requires metadata in the index to detect the presence of such heavy hitters. When this is the case, we fall back to a separate execution as presented in Section 2.3.3.

The JCC-H [19] benchmark demonstrates these problems, where there are five populous orders that have very many lineitems. When we groupjoin these orders with the lineitem table, the index based execution is essentially single-threaded, while the memoizing and eager right execution execute on all available cores. In this scenario, the optimal method additionally depends on the available cores of the machine.

The cost calculation for this execution strategy differs significantly from the other strategies. Since we choose a different access path for the tuples of the right side S , the decision to use an index is strongly dependent on the relative performance of accessing data linearly during a table scan, or using the random-accesses of an index. This performance depends on many factors: What kind of index do we use? Traditional B-Trees [11, 55], or in-memory optimized indexes such as lock free hash indexes [32] or Adaptive Radix Trees [92, 93]. Additionally, the random-access performance also depends on the physical storage of the base table. For example, cloud-centric storage architectures using large files [29] have large read amplification for random lookups, and even storing tuples in main-memory optimized compressed Data Blocks [87] introduces some overhead.

We, therefore, exclude index groupjoins from our regular cost models. In our system Umbra, we instead use a two-stage optimization, where we first decide if we use the index, and if not choose one of the other execution strategies. In Umbra, we use an empirically determined $10\times$ overhead of accessing tuples via an index join with a B-Tree index and cached pages, compared to a full table scan.

When we have a suitable index and accessing it is cheaper than the full table scan, index groupjoins have excellent performance. In comparison to the other combined methods, we can avoid scanning the full right table. In separate join and group-by execution, we also use the index, but still break the execution pipeline by building the aggregation hash table. By avoiding this unnecessary hash table, index groupjoins are about 33 % faster than separate index joins.

Table 2.2: Example cost calculations of groupjoin implementations.

$ R $	$ S $	σ_R	σ_S	$ R \times S $	$ R \bowtie S $	C_{eager}	C_{memo}	C_{sep}
100	200	80%	80%	80	160	280	680	708
100	200	80%	10%	80	20	280	260	246
100	100	100%	10%	100	10	200	230	233
100	500	100%	5%	100	25	600	275	283

2.3.5 Choosing a Physical Implementation

To recap, we presented four parallel execution strategies for groupjoins. In Section 2.3.1, we presented an eagerly right aggregating groupjoin, in Section 2.3.2 we used a combined join and aggregation table with memoizing thread-local aggregations. Furthermore, we showed in Section 2.3.3, that we can rewrite arbitrary groupjoins as separate join and group-by. Lastly, we described how to use indexes for groupjoins in Section 2.3.4. All four implementations have different characteristics, which we formalized for the first three as a cost model to compare their relative performance:

$$\begin{aligned}
 C_{\text{eager}} &= |S| + |R \times S| \\
 C_{\text{memo}} &= 2|R| + 3|R \bowtie S| \\
 C_{\text{sep}} &= |R| + 3.3|R \bowtie S| + |R \times S|
 \end{aligned}$$

The base of all three cost functions consists of the underlying cardinalities $|R|$ and $|S|$, and the semijoin reduced cardinalities $|R \times S| = |R| \sigma_R$ and $|R \bowtie S| = |S| \sigma_S$. In Table 2.2 we go through some exemplary calculations of this cost model. As the examples show, the different implementations have significant differences in the total cost of execution, depending on how much a side is reduced with its relative selectivity σ .

When considering C_{eager} , the differences are especially pronounced. C_{memo} and C_{sep} are closer, since both approaches implement similar logic. Their largest difference is the static vs. dynamic memory allocation to compute the aggregates. Figure 2.6 shows the allocated memory in Umbra during the execution of a groupjoin with our three implementations. In the shown case, every input tuple finds a join partner, thus we need memory to store all tuples. Both fused approaches store them in one hash table, either statically allocated up front (memoizing), or dynamically during eager aggregation of S . In contrast, separate execution allocates a smaller initial hash table and dynamically builds the additional aggregation table. In this example, the fused storage uses about 1 GB peak memory, while separate execution consumes about 50 % more. However, depending on how many distinct aggregates we encounter (σ_R), the dynamic

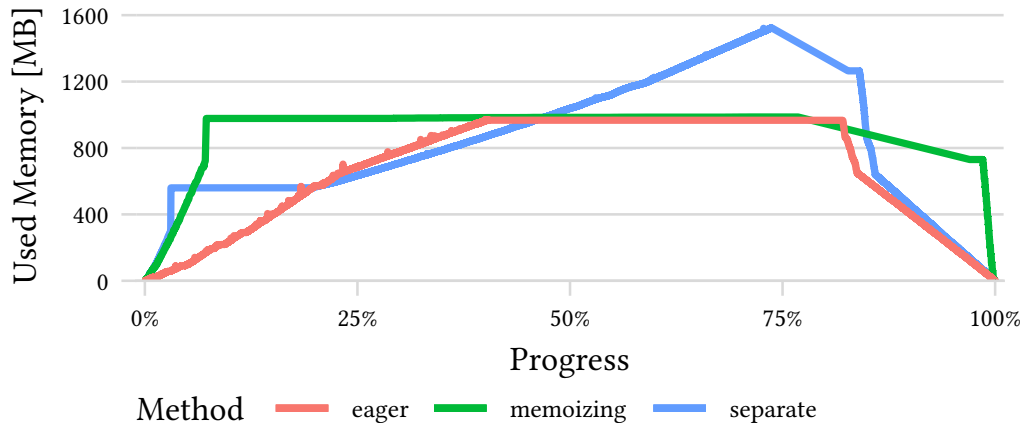


Figure 2.6: Memory consumption of TPC-H SF 10 orders - lineitem groupjoins.

allocation of the separate execution might also use less memory. In our cost model, we encode this difference as the simplified 30 % factor in C_{sep} . However, this factor depends on a few system characteristics, e.g., the cost to dynamically allocate memory and the momentary scarcity of it. Additionally, the number of aggregates also influences the hash table payload sizes.

Like any cost-based optimization, this approach relies on estimates of the underlying data. While this works well for base tables and joins, the quality can deteriorate with nested groupjoins and other aggregates.

2.4 Evaluation of Groupjoins

In this chapter, we present the experimental evaluation of the presented groupjoins in our research RDBMS Umbra [113]. We start with a study of the behavior of parallel groupjoin execution in the TPC-H benchmark, and if it corresponds to our presented cost model. As detailed in Section 2.3, groupjoins are commonly used in unnesting, but we also apply them when they can improve performance. For this evaluation, we consider the groupjoins in the well-known analytical benchmark TPC-H, compare the performance of our proposed implementations, and evaluate our cost model therein.

Hypothesis For TPC-H, the selectivity and relative sizes do not change when increasing the scale factor, thus our cost model presented should stay consistent relative to each variant. Since all three proposed algorithms are virtually lock and contention free, we expect no relative changes between algorithms under varying parallelism or data size.

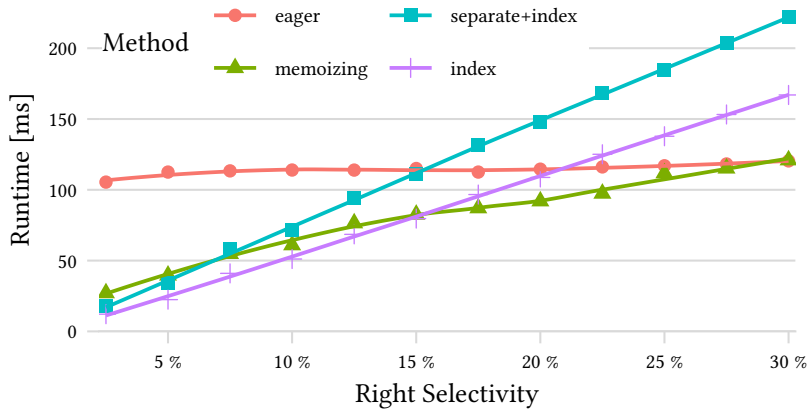


Figure 2.7: Performance of the index groupjoin on TPC-H SF 10 orders - lineitem via the `l_orderkey` foreign key.

Setup of Performance Measurements We run all benchmarks on a NUMA system with $2 \times$ Intel Xeon E5-2660v2 CPU with 10 cores each, $2 \times$ hyper-threads, and 256 GB RAM. To measure performance with warm caches, we repeat the executions 20 times and report the median value. The typical run-to-run median absolute deviation for this setup is 1 %.

In a first experiment, we evaluate the performance of our different groupjoin implementations under a varying right selectivity σ_s . Figure 2.7 shows the execution time of our different groupjoin implementations. In this experiment, we groupjoin the TPC-H orders and lineitem tables with a foreign key index on `l_orderkey`, which we use either in an index join or an index groupjoin. In a direct comparison, index groupjoins are strictly better than building a separate aggregation hash table when σ_s is small. However unsurprisingly, when we join with more tuples of lineitem, the memoizing and eager right approaches can be faster. Index groupjoins are faster for a right selectivity of up to $\sigma_s = 15\%$. Compared to this, Umbra’s $10 \times$ heuristic is rather conservative.

In the second experiment, we validate the quality of our cost model recommendations. This experiment compares the predicted cheapest to the actually measured fastest implementation. The setup is a micro benchmark on the TPC-H SF 10 data set with the same, single orders-lineitem groupjoin. To test the whole σ_R and σ_S parameter space, we prefilter each of the tables in 1 % increments via the primary key. Figure 2.8 shows the 10 000 combinations, and plots the measured fastest implementations in the left plot, in comparison to the cost model recommended ones on the right.

This experiment shows that our prediction is a good indicator of the actual fastest performance. As expected from the cost model, the most impactful

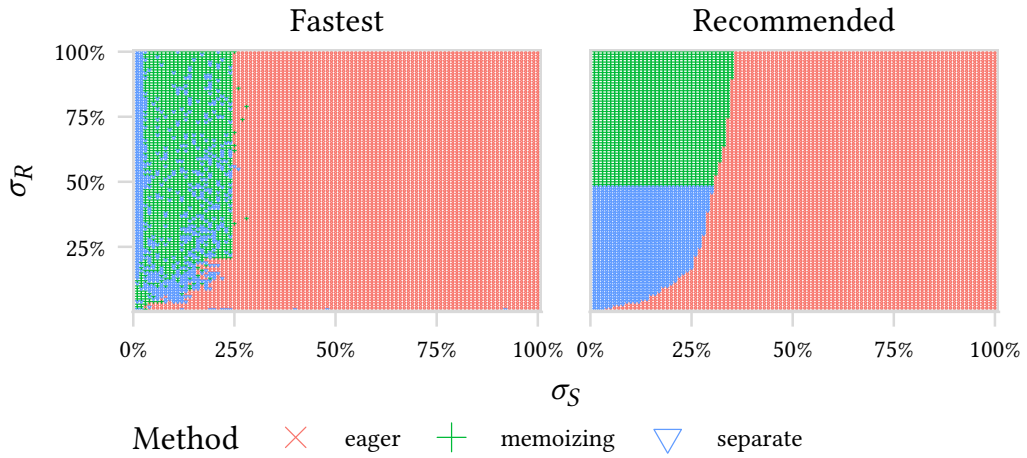


Figure 2.8: Comparison of fastest and cost model recommended implementation for a TPC-H orders - lineitem groupjoin.

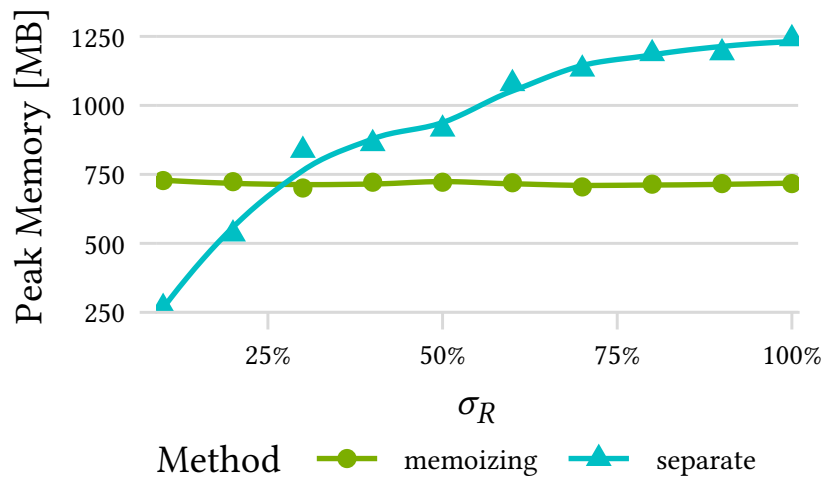


Figure 2.9: Peak memory usage for a TPC-H orders - lineitem groupjoin.

decision is whether we should aggregate eagerly. Our cost model recommends this for the upper two-thirds of the σ_S range, while the measurements indicate that the break-even point is already a bit lower. However, at this border the methods only have minor performance differences. To quantify this, we pairwise compare the performance of the measured fastest method with the, sometimes slower, cost model recommendation. Using the recommendations results in a mean absolute percentage error of only 1.7% over the best performance and a maximum absolute percentage error of 95%.

Table 2.3: TPC-H Groupjoins. Cost model calculations with four TPC-H groupjoin queries on scale factor 1.

Q	R	S	σ_R	σ_S	C_{eager}	C_{memo}	C_{sep}
3	147 k	3.24 M	54%	6.8%	3.32 M	956 k	954 k
13	150 k	1.48 M	63%	100%	1.58 M	4.75 M	5.14 M
17	204	6.00 M	100%	0.10%	6.00 M	19.0 k	20.9 k
18	57	6.00 M	100%	0.84%	6.00 M	152 k	167 k

However, the memoizing and separate execution strategies are generally closer in their measured runtime performance. We attribute this mostly to the optimized dynamic memory allocation in Umbra [37], since the peak memory usage differs much more. To quantify this, we measure the maximum amount of memory used to execute the groupjoin under a varying left selectivity σ_S in Figure 2.9.

In the next experiment, we limit the amount of parallelism and observe the query performance with a fixed groupjoin algorithm, and fixed TPC-H scale factor 10. The third experiment uses all threads, but varies the scale factor. Note that Umbra was already a system with state-of-the-art performance, even without our contributions. As baseline for TPC-H, the speedup of Umbra over MonetDB [21] is about 3.2 \times and about 101 \times over PostgreSQL [147].

Cost Model We first go through the cost model calculations for groupjoins in TPC-H, before evaluating if the model accurately predicts the performance of these queries. For this evaluation, we look at a total of four TPC-H queries using a groupjoin: Q3 is an organically occurring groupjoin, where we first join and then group-by the same key. Q13 has a similar groupjoin sequence, albeit in a nested query itself. In contrast, the groupjoins in Q17 and Q18 are the result of unnesting. We also provide an interactive query plan viewer for these queries online².

The cost model calculations for these joins in Table 2.3 show our predicted relative performance for these queries. Q3 has high selectivity of the right-hand side, which favors the lazily aggregating variants, and a moderate relative left selectivity, which puts separate processing at an advantage. When we look at Q13, the join is very unselective on the right side, which puts eager right aggregation at a clear advantage. Both unnested queries Q17 and Q18 only compute the groupjoin on a small and highly selective left side, which puts the hybrid memoizing groupjoin at a slight advantage.

²<https://umbra-db.com/interface/>

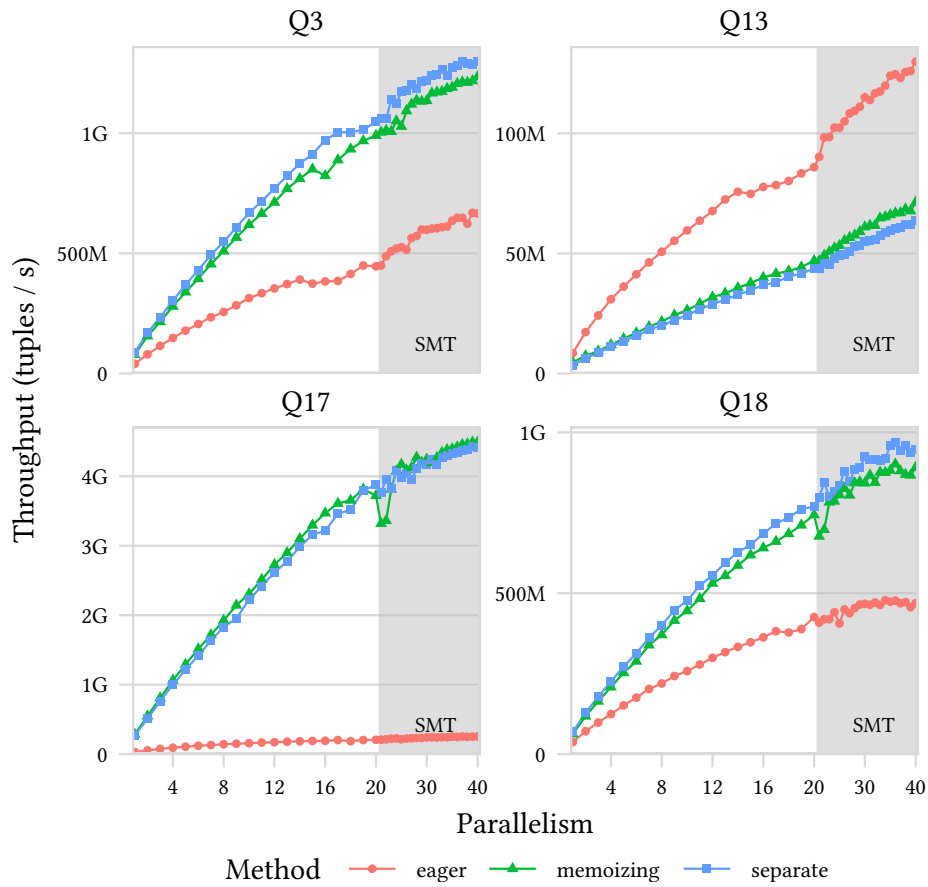


Figure 2.10: Parallel scale-out of TPC-H SF10 groupjoin queries.

In the following, we run two experiments of our algorithms under a varying parallelism and data scale to validate these claims and to show that the cost model calculations are robust under these parameters. In contrast to the cost calculations from Table 2.3, which only include the variable costs of the groupjoin implementation, our benchmarks measure the throughput of the whole query.

Figure 2.10 shows the relative performance of the different groupjoin implementations with increasing parallelism. We observe that, as expected, the relative performance between the algorithms stays the same. All three implementations show a linear speedup when increasing the parallelism, with a tamping down speedup on hyper-threads.

In Figure 2.11, we vary the amount of processed data via the scale factor and see a similar picture. Again, the relative performance stays unchanged and, apart from some effects when exceeding cache sizes, the overall throughput stays relatively constant. All in all, our cost model has proven to be robust in

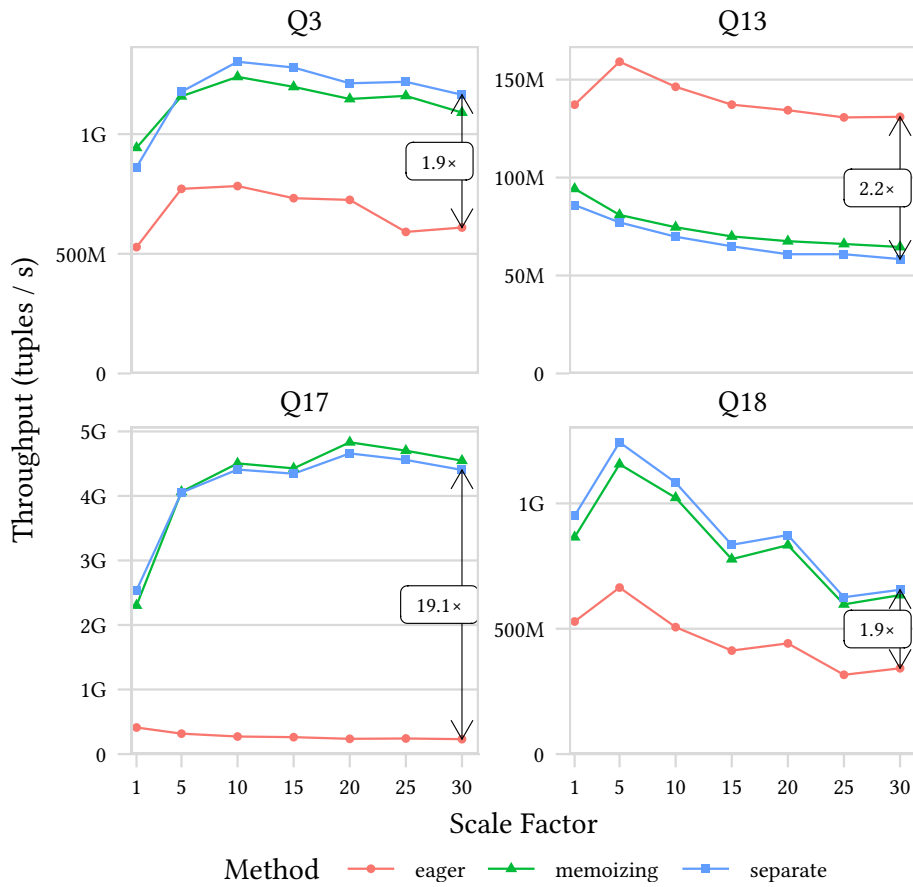


Figure 2.11: Data size scale-out of groupjoins in TPC-H.

regard to variable system parameters, and accurately predicts the most efficient groupjoin implementation.

Overall, eager aggregation can bring over $2\times$ improvement in Q13, but is over an order of magnitude slower in Q17. The other implementations are much closer to one another, mostly because we build the hash table with the duplicate free left side, which is orders of magnitude smaller than the right side. In comparison to processing the large right side, building the relatively small left hash table has only a minuscule impact on the overall query. Nevertheless, a proper model will find the best execution plan and significantly improve the efficiency.

Over the four queries in TPC-H that use a groupjoin, our cost model based approach achieves a geometric mean speedup of 20% over a baseline that executes join and group-by separately. We also ran a similar experiment over

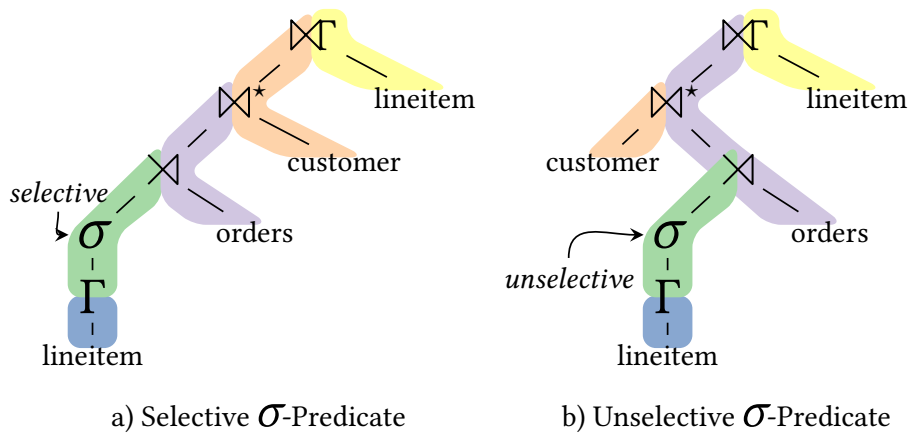


Figure 2.12: Possible query plans for TPC-H Query 18. Depending on the σ filterselectivity, we use the customer relation as hash-join build or probe side, which roughly leads to a 10 % difference in performance.

TPC-DS, where we see similar results: A total of 13 queries can use a groupjoin, with a geometric mean speedup of 5 %.

2.5 Aggregate Estimates

Good estimates for computed columns in nested aggregates are one of the missing links in cost-based query optimization. Cardinality and selectivity estimations for base table columns are well-known, and despite some problems, work quite well in practice [91, 115]. While statistics on singular columns fail to capture correlations, histograms, samples, and sketches provide a solid baseline, and recently developed techniques using machine-learning work towards multi-column estimates [38, 79]. However, estimates for computed columns such as aggregates are rarely used, which results in poor cost-model calculations, and suboptimal query plans.

We propose to extend existing approaches that work on base table columns by calculating statistics, which allow deducing computed column estimates. Our approach uses a lightweight statistical model that can be piggybacked onto regular sampling or histogram-based statistics. The key idea is to fit a skew-normal distribution to the underlying data using a method of moments estimator, which can be cheaply maintained on base tables, as well as for computations throughout the query tree. With this fitted distribution, we then efficiently estimate the selectivity of predicates on computed columns, and the resulting cardinality.

Surprisingly few systems consider the results of computed columns in cardinality estimation, which is rather surprising considering this is a part of standard SQL, which even has a dedicated `HAVING` syntax. After unnesting or in nested analytical views, it is common to have aggregates and predicates on aggregates embedded in lower parts of the algebra tree, where the resulting cardinality has consequences for the quality of query plans. One example is TPC-H Q18 shown in Figure 2.12, with the nested predicate `HAVING SUM(l_quantity) > 300`. The estimated selectivity for the filter σ in the green pipeline has significant impact on the query performance. Depending on the selectivity, the optimal query plan is either, a) when the predicate is very selective, or b) if it is not.

In the figure, we use the convention to build the join hash tables with the left and probe with the right side. For Q18, all data sources except the aggregate result are unfiltered base tables, where cardinality estimation is trivial. The challenging part for cardinality estimation is the join with the customer table \bowtie^* , which is marked with an asterisk. Since building a hash table is more expensive than probing it, we estimate which side is smaller. In Q18, we estimate if the σ filter condition produces less tuples than the entire customer table. The base table cardinalities differ by an order of magnitude (150 k customers and 1.6 M distinct orders for scale factor 1), so simple heuristics most likely mispredict these cardinalities. In preliminary experiments, this misprediction has roughly a 10 % performance penalty for the whole query. To avoid this and get closer to the real selectivity of 0.003 %, we need robust estimation of computed columns.

In the following, we present our novel computed column estimator, based on the method of moments for skew-normal distributions [120]. In result, we get orders of magnitude better estimates for filters on computed columns and in turn generate better query plans.

2.5.1 Skew Normal Distribution

Our key insight is that `HAVING` predicates are mostly on computed values based on columns of “natural” numerical quantities, e.g., price, balance, counts, ratings, durations, etc. In contrast to predicates on keys or identifiers, they are rarely compared for equality, but more commonly with range predicates, e.g., `≤` or `BETWEEN`. In the following, we propose an estimation model for computed columns that roughly follow a normal distribution, i.e., most values center around a mean, with relatively few outliers from that mean. Additionally, we model a limited amount of skewness in the underlying data to break the inherent symmetry of a pure Gaussian normal distribution. The resulting selectivity estimation framework then handles a wide variety of computed columns.

The centerpiece of our estimation framework is the skew-normal distribution, as proposed by Azzalini [5], which combines the normality assumption with

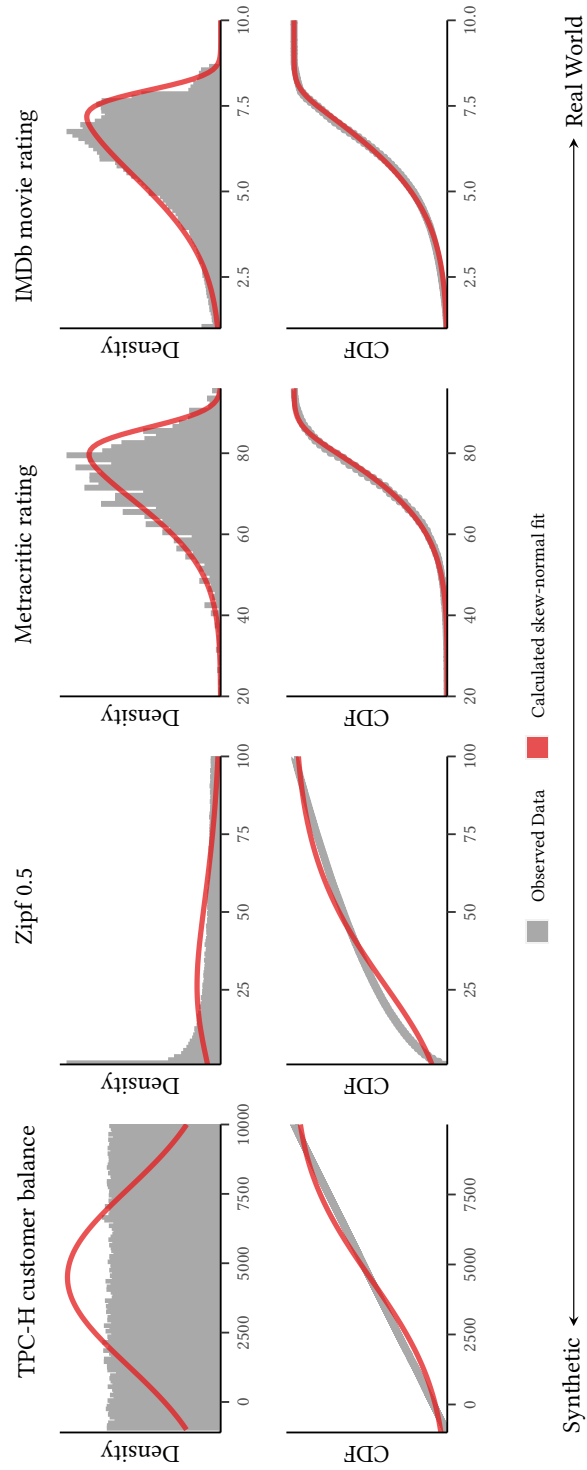


Figure 2.13: Skew-Normal Fit. Histograms of several data sets, ranging from uniform synthetic to skewed real-world data sets. The overlaid red distribution is a fitted skew-normal distribution.

a better fit for skewed distributions. For our estimation framework, the skew-normal is a good trade-off for a reasonably robust, yet computationally simple statistical model. The skew-normal $sn(\xi, \omega, \alpha)$ is closely related to the normal distribution $\mathcal{N}(\mu, \sigma^2)$, with an additional shape parameter α , that allows some asymmetry as skew. With the special case $sn(\mu, \sigma, 0) \sim \mathcal{N}(\mu, \sigma^2)$, the skew-normal represents a superset of the normal distribution.

To reason about computed columns, we first discuss how to fit the distribution to existing columns, before defining transformations that describe the calculation of new columns. For the base table fit, we use the *method of moments* as proposed by Pewsey [120], which uses the observed moments of a random sample. For our approach, we piggyback this calculation of the moments, in the form of descriptive statistics, onto regular table samples. To calculate these, we take a sample X of size n of each numerical column and calculate the statistics as follows:

$$\begin{aligned} \text{Mean: } \bar{x} &= \frac{\sum X}{n} \\ \text{Standard deviation: } \bar{\sigma} &= \sqrt{\frac{\sum X^2}{n} - \bar{x}^2} \\ \text{Skewness: } \bar{\gamma} &= \frac{\frac{\sum X^3}{n} - 3\bar{x}\frac{\sum X^2}{n} + 2\bar{x}^3}{\bar{\sigma}^3} \end{aligned}$$

Then, we transform the observed moments to the parameters of the skew-normal $sn(\xi, \omega, \alpha)$, as described by Azzalini [6, 7].

$$\begin{aligned} \xi &= \bar{x} - \omega \cdot m & \delta &= \sqrt{\pi/2} \cdot m \\ \omega &= \sqrt{\frac{\bar{\sigma}^2}{1 - m^2}} & \text{where } m &= \frac{o}{\sqrt{1 + o^2}} \\ \alpha &= \frac{\delta}{\sqrt{1 - \delta^2}} & o &= \pm \sqrt[3]{\frac{2|\bar{\gamma}|}{4 - \pi}} \end{aligned}$$

In Umbra, we default to a sample size of 1024 values, which we keep up-to-date using reservoir sampling [16]. Our sampling process also incrementally updates the observed moments, which means that we can keep online statistics that always track the up-to-date state of the database.

Figure 2.13 shows the calculated skew-normal fit over four data sets. The two left distributions are both generated, i.e., uniform random data from TPC-H and a sample of a moderately skewed Zipf distribution [57]. Both distributions on the right are from real-world data sets: Steam App statistics Metacritic

ratings [118] and IMDb movie ratings [94]. The figure shows the underlying data as gray histogram in the top row, and the empirical distribution function in the bottom row. Overlaid in red, we plot the PDF and the CDF, of our inferred skew-normal model.

Arguably, this method leads to a good fit of the underlying data. However, the synthetic data also pinpoints a fundamental limit of this approach. The skew-normal is unable to accurately capture the “squareness” of the uniform random data with its heavy tails, respectively “peakiness” of left edge of the Zipf distribution. More formally, the skew normal cannot fit the kurtosis—the fourth statistical moment. In addition, it can only fit a limited skewness within its parameter space ($\gamma^{\max} = \frac{\sqrt{2(4-\pi)}}{(\pi-2)^{3/2}} \approx 0.9953$ for $\alpha \rightarrow \infty$ [7]). Thus, our model can only truncate the skewness, while ideally, we would detect these edge cases and switch to a better fitting distribution using a hyperparameter model. While such a more advanced model would probably produce a better fit, the trade-off we take here has little overhead, while still fitting a CDF that produces a relatively low error for selectivity estimates of predicates.

This resulting statistical distribution $sn(\xi, \omega, \alpha)$ has several applications for our estimations. The main use-case is the estimation of \leq predicates, like the one in TPC-H Q18, which follows naturally from the CDF Φ_{sn} of the skew-normal:

$$\Pr[x \leq c] = \Phi_{sn}(c)$$

Estimating equality is only possible indirectly, since the probability distribution is continuous. As approximation, we evaluate a range predicate BETWEEN $\pm \epsilon$ with default $\epsilon = 0.5$ to get a bucket sized for one integer.

2.5.2 Transformations on the Skew Normal

To reason about computed columns, we first define arithmetic transformations on our statistics. Given two skew normal input distributions, we model binary arithmetic expressions to estimate predicates on computed columns. As an example, consider the following condition on an analytical query that filters for orders exceeding the customer’s current balance:

```
... WHERE
  part.price * ord.quantity > cust.balance
```

We estimate the resulting distribution of such algebraic expressions using $\circ \in \{+, -, *, /\}$ with our statistical model. We piecewise transform the input moments, before fitting a skew-normal distribution for the resulting computed

column:

$$\begin{aligned}\mu_{x \circ y} &= \mu_x \circ \mu_y \\ \sigma_{x \circ y}^2 &= E[(x \circ y)^2] - \mu_{x \circ y}^2 \\ \gamma_{x \circ y} &= \sigma_{x \circ y}^{-3}(E[(x \circ y)^3] - 3\mu_{x \circ y}\sigma_{x \circ y}^2 - \mu_{x \circ y}^3)\end{aligned}$$

2.5.3 Aggregate Estimation

We extend these statistical building blocks on binary expressions to reason about the statistical distributions of aggregated n-ary columns. Staying with a similar example as previously, consider a query that builds an analysis on the biggest customers that have at least a revenue of one million:

```
... GROUP BY cust.id HAVING
    SUM(part.price * ord.quantity) > 1000000
```

In the following, we go over the standard SQL aggregate functions, i.e., AVG, COUNT, MAX, MIN, and SUM, and discuss our estimates for these. Figure 2.14 shows three examples of differently skewed input columns X in green. We model the group sizes of these aggregates as i.i.d. random variables within the domain of the estimated distinct values of the grouping key [50]. This results in a binomial distribution of group sizes, which we again approximate using a skew-normal distribution, plotted in blue. For COUNT aggregates, this already estimates the result distribution. AVG aggregates are similarly independent of the group size and follow the same distribution as the input of the aggregation function.

More interesting are SUM aggregates, shown in the second column, which depend on both input statistical distributions: The distribution of the summed-up column, and that of the group size. We approximate the resulting computed column by a multiplication via the previously discussed transformations, and plot the resulting calculated estimate in red.

To cross-validate the fit of this model, we simulate the calculation of the aggregates and plot a histogram of the resulting data in gray. For MIN and MAX aggregates, as displayed in the following two columns, we additionally need to consider their extreme value property, which we model with a Gumbel extreme value distribution G [31]. Since the distributions of maximum and minimum are symmetrical, we only detail the MAX case here, but MIN behaves similarly with flipped signs.

Let X be a skew-normal distributed random variable with inverse CDF quantile function Φ_{sn}^{-1} . Then we use the theorem of Fisher-Tippett and Gnedenko [31]

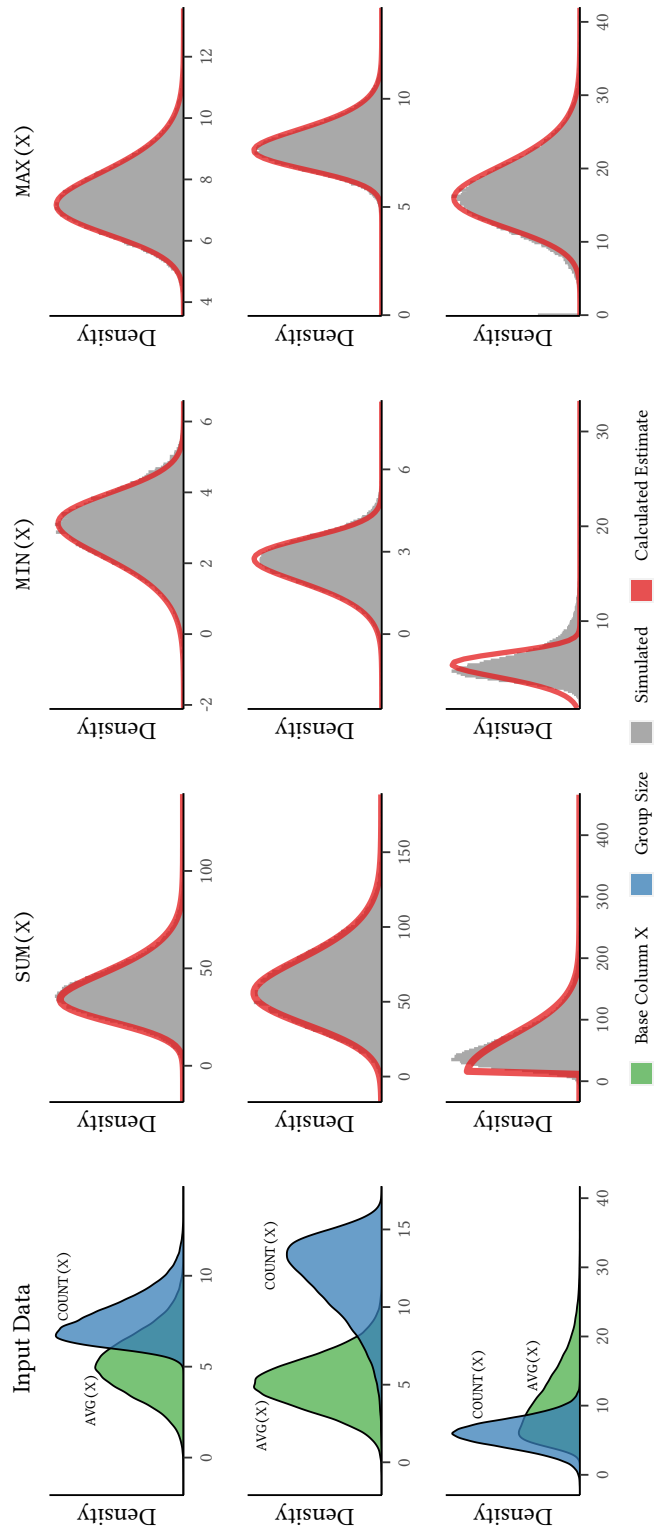


Figure 2.14: Skew-Normal Fit of Aggregates. The first column shows three different distributions of base column and group size. The next three columns compare simulated aggregates with a calculated fit of our skew-normal estimator.

to find parameters for the extreme value distribution $G(\mu, \sigma)$:

$$\begin{aligned}\mu &= \Phi_{sn}^{-1}\left(1 - \frac{1}{n}\right) \\ \sigma &= \Phi_{sn}^{-1}\left(1 - \frac{1}{n}e^{-1}\right) - \mu\end{aligned}$$

Then, we fit a skew-normal distribution to $G(\mu, \sigma)$ to make our model closed. The resulting models provide insight on the expected distribution of such computed columns. For our query optimization pipeline, this means that we can provide accurate input for subsequent cost-based join-ordering. Good estimates, in combination with low-contention parallel execution, then produce near-optimal query plans.

2.6 Evaluation of Estimates

In this chapter, we present the experimental evaluation of the quality of our aggregate estimations in our research RDBMS Umbra [113]. We determine how much impact improved aggregate estimates have with a comparison of the estimated cardinalities for predicates on aggregated columns. We compare our implementation to three other RDBMS, before isolating the effect of aggregate estimation.

In TPC-H, the only query with a nested aggregation is the Large Volume Customer Query Q18, with a fairly simple HAVING predicate. To focus on the quality of aggregate estimates, we only consider its subquery in this experiment:

```
SELECT l_orderkey
FROM lineitem
GROUP BY l_orderkey
HAVING SUM(l_quantity) > THRESHOLD
```

The subquery sums the quantity of items in an order and only selects the orders with the most numerous items. As described in the TPC-H specification, the threshold over which an order is considered large is a substitution parameter. In our experiment, we extend the range of this parameter to vary the predicate selectivity from 0 % to 100 % and also consider more challenging expressions.

Systems Comparison In the first part of our evaluation of aggregate estimates, we consider a total of four database management systems: Tableau Hyper via its Python API 0.0.15145, DBMS X, PostgreSQL 14.4, and our research system Umbra [113]. To get accurate cardinality estimates, we load the TPC-H scale factor 1 validation data into an empty database. Then, we ensure that the DBMS

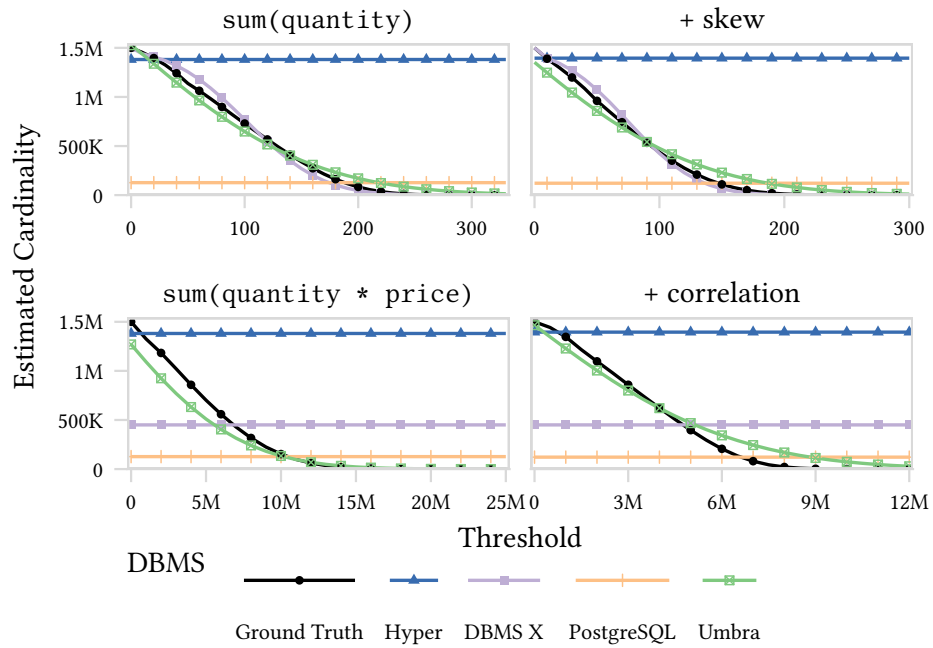


Figure 2.15: Estimates for TPC-H Q18 style subqueries.

has accurate statistics over this data by issuing control commands to regenerate statistics, if such commands exist. Afterwards, we execute the subquery with the substituted constant and extract the query plan. For this evaluation, we record the estimated, and the true cardinality in four different scenarios that are similar to the subquery of TPC-H Q18. First the regular Q18 aggregate over the uniform random base column $\text{sum}(\text{quantity})$, and with a skewed Zipf 0.5 quantity. As a slightly more complex aggregate, we use $\text{sum}(\text{quantity} * \text{price})$, again on the uniform random base columns, and with an additional anti-correlation ($\rho = -0.7$) between quantity and price (i.e., the higher the price, the lower the quantity). Note, that all these scenarios depend on group-size estimates, which we do not consider in the scope of our work, but refer to previous work [50].

Figure 2.15 shows this data, where the ground truth cardinality describes a decreasing curve that corresponds to higher thresholds. Our presented estimation framework in Umbra is close to the ground truth over the whole range of the threshold, even for complex predicates. In the simple scenarios with aggregates over a single column, DBMS X behaves similar. However, it does not publicly describe or document the underlying model. In addition, it falls back to estimating “magic constants” for expressions referencing more than one base column. That means, when the selectivity for a predicate cannot be determined, the systems just estimate a fixed fraction of the estimated input

cardinality. Indeed, Hyper estimates $1/2$ of its input estimate and PostgreSQL $1/3$.

Isolating the Impact of Aggregate Estimates We established that our estimates capture the cardinality of HAVING predicates. In the following, we isolate the impact of these aggregate estimations, and increase the complexity of queries and data sets. To eliminate other factors, we emulate the selectivity estimation with a fixed selectivity inside Umbra. This allows a more clear-cut evaluation of the impact of Umbra’s skew-normal model on the estimation.

This evaluation uses queries on two real-world data sets. In contrast to the generated TPC-H data, these are full of correlations and non-uniform data distributions. The first data set is the Internet Movie Database (IMDb), in a slightly modified form from the Join Order Benchmark (JOB)³: Since IMDb primarily stores facts as strings, we extract a separate table that contains the vote count and the user rating for movies, to allow statistics collection. On these columns, we define five additional aggregation queries that calculate statistics on the new numerical columns. Furthermore, we also consider aggregation queries derived from public workbooks in Tableau Public⁴. The query set is available online⁵.

To measure the quality of the estimates, we report the *q-error*. The *q-error* measures the factor that an estimate differs from the ground truth. It captures the relative difference to the real value and is symmetric and multiplicative. For example, a *q-error* of one means that the estimate accurately captured the true cardinality, and a *q-error* of 10 corresponds either an over- or underestimation by a factor of 10. With a bounded *q-error*, it is also possible to give a theoretical guarantee about the optimal query plan quality [105].

In Figure 2.16, we visualize the quality of our estimates from over 100 individual queries with predicates on aggregates. For the IMDb queries, we vary a replacement parameter of a having predicate, similar to the last experiment, to cover the whole range of 0 to 100 % true selectivity. From the public BI benchmark, we consider all queries that evaluate a predicate on more than one aggregation tuple. In total, this gives us 82 IMDb aggregation queries and 48 aggregation queries from the public BI benchmark. Each box in this plot shows the median and the first and third quartiles, with individual dots for outliers.

The quality of our estimates strongly depends on the calculated statistics. For Q35 to Q38, the estimates are close to the true cardinality, with occasional outliers on the tail edges of the distribution, i.e., when the predicate is very

³<https://homepages.cwi.nl/~boncz/job/imdb.tgz>

⁴https://github.com/cwida/public_bi_benchmark

⁵<https://db.in.tum.de/~fent/data/aggEst.tgz>

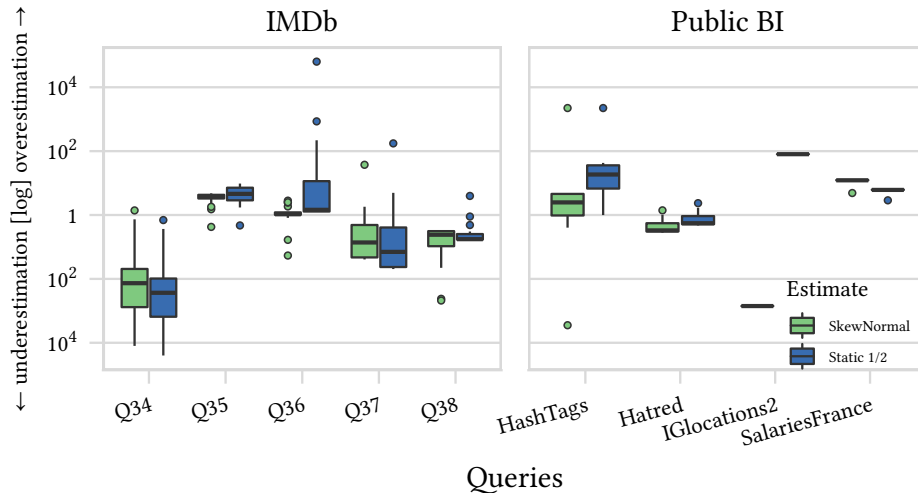


Figure 2.16: Estimation quality of aggregation queries. The box plots show the log-scale q-error of our estimates in comparison to the static selectivity of Hyper. Our skew-normal model reduces the geomean q-error by 46 % from 45.8 to 24.7.

selective. Our estimates, compared to a static selectivity estimation, capture the shape of the aggregates better and reduce over- as well as underestimation. Q34 shows one of the shortcomings of our approach, where a sum aggregate combines two distributions with heavy tails. In comparison to static selectivity estimation, our skew-normal model improves the error, but is limited by the quality of the baseline group size estimates. We found that for static estimation, we get the least error with the $1/2$ fraction that Hyper uses, which we compare in Figure 2.16. In comparison to this configuration, our skew-normal selectivity estimation is a clear advantage and reduces the geometric mean of the q-errors from 45.8 to 24.7, which eliminates the impact of bad selectivity estimation.

To summarize, computed column estimates improve the estimation quality of nested aggregates. In combination with efficient parallel groupjoins, this can have significant impact on query performance. Figure 2.17 shows a breakdown of the affected queries in TPC-H and TPC-DS. Most queries see a moderate speedup, with only one major slowdown in TPC-DS Q73. The slowdown arises due to a worse logical plan, where previous the magic-constant estimation had a lucky guess and canceled out an unrelated error in group-size estimation. Nevertheless, it is still valuable to improve estimates so that they can capture the behavior of nested aggregates. Over the affected queries, we get a geometric mean speedup of 23 % in TPC-H and 6 % in TPC-DS.

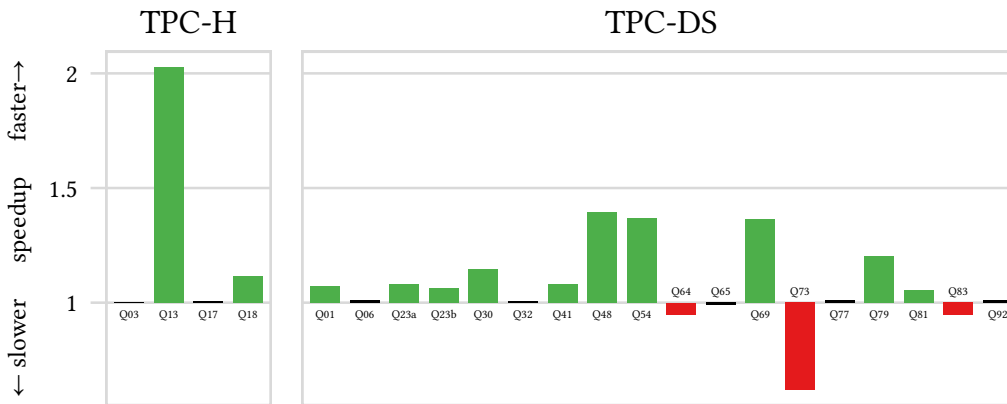


Figure 2.17: Overall Impact on TPC-H and TPC-DS.

2.7 Planning with Groupjoins

Our previous discussions of the groupjoin focussed on the implementation of an individual operator. By choosing an optimal execution strategy, we get cheaper physical execution plans. In the following, we use groupjoins to reason about the whole logical query plan and improve its overall quality.

So far, we only considered strictly better plans by opportunistically introducing groupjoins when our join reordering [116] produces a suitable plan. Specifically, this requires that the resulting plan resembles Figure 2.18a, and contains a sequence of matching group-by and join without intermediary operators. Figure 2.18b illustrates that this misses opportunities for plans with nested aggregates. Thus, planning of groupjoins early on produces better and more robust query plans. Instead of introducing groupjoins opportunistically, we look for and eagerly introduce groupjoins for nested aggregates. This improves the plan for this potential groupjoin, and helps to find a better overall plan by providing reordering possibilities. We first show, how we eagerly introduce groupjoins in our logical query plan, before we discuss two example queries from TPC-H that benefit from this optimization.

2.7.1 Eagerly Introducing Groupjoins

For join ordering, we build a query graph that connects relations by join predicates. For the initial construction of this graph, we consider any non inner-joins as relations [125]. As a consequence, group-by operations form boundaries of our reordering graphs.

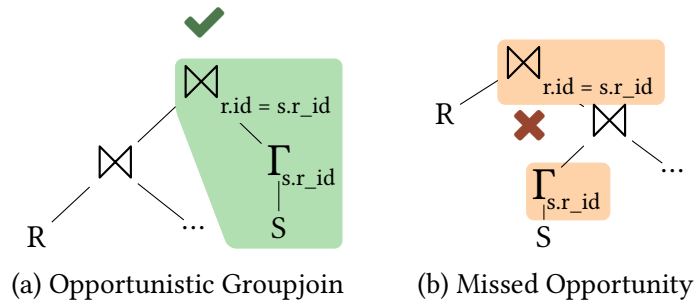


Figure 2.18: Physical planning of groupjoin operators depends on join ordering. A purely opportunistic approach misses non-trivial combinations.

Algorithm 4: Identifying Groupjoins.

```

input: JoinGraph (relations, predicates)
for each GroupBy  $\Gamma \in$  relations:
  ps  $\leftarrow$  predicates joining on  $\Gamma$ .key
  // Precondition ①
  if ps does not cover  $\Gamma$ .key completely:
    continue
  R  $\leftarrow$  opposite relation(s) of ps
  // Precondition ②
  if ps is not a superkey of R:
    continue
  push ( $R \bowtie_{ps} \Gamma$ .key) below  $\Gamma$ 

```

We identify potential groupjoins in this graph via Algorithm 4. The algorithm first identifies potential group-bys in the input relations and checks the preconditions to introduce a groupjoin, i.e., that we have a foreign key join with the group-by key (cf. Section 2.2.1). However, we do not immediately introduce a groupjoin, but only push this join into the group-by inputs. Swapping join and group-by allows optimizing the input even further [155]. If the join is selective, we might want to push it even further down. Keeping this conceptual groupjoin separated allows our standard reordering algorithm to determine the optimal plan. Otherwise, the join will end up at the top of the subtree, and we choose a fitting groupjoin according to our cost model (Section 2.3.5).

As a result, we get an improved intermediary plan with a conceptual groupjoin. While this plan does not necessarily result in the execution of a groupjoin, the resulting plan is strictly better than the initial plan.

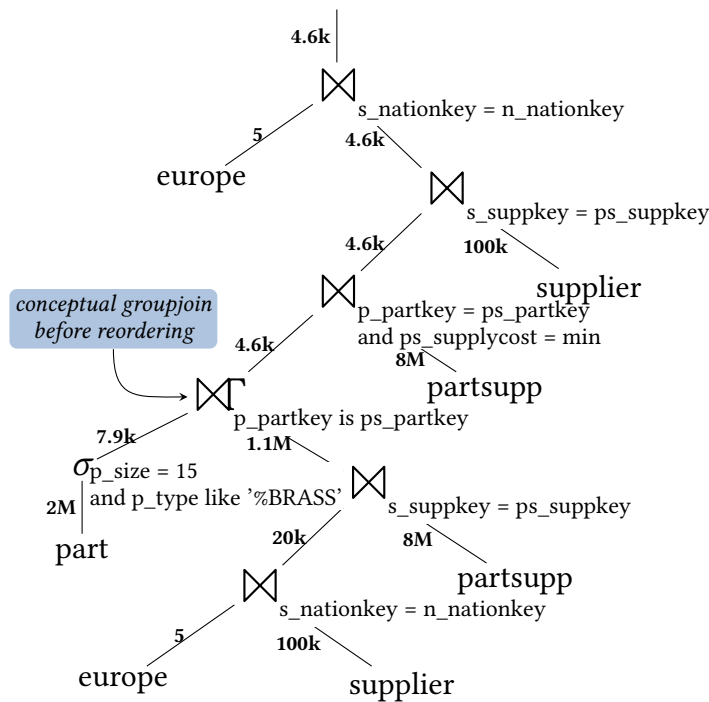


Figure 2.19: Intermediary planned groupjoin for TPC-H Q2.

2.7.2 Additional Groupjoins in TPC-H

In the following, we show that this eagerly introduction of groupjoins also practically results in better plans. In TPC-H, there are two queries with such groupjoins over nested aggregates: Q2 and Q20.

TPC-H Q2 finds the minimum cost supplier for certain parts using a dependent subquery. With basic decorrelation, we calculate the minimum supply cost for all parts, before joining with the specific parts. Figure 2.19 shows a first execution plan to evaluate this query. Note that we abbreviated the trivial join between region and nation as the CTE “europe”.

In this plan, Algorithm 4 detects a groupjoin for the nested aggregate of the join with part. Therefore, we conceptually introduce the groupjoin shown in Figure 2.19. The overall utility of physically executing this groupjoin is limited, but it acts as a useful stepping stone during query planning. We then push the join below the group-by and consider it during join reordering in the lower subtree.

In a first approximation, this technique generates plans that are somewhat similar to the common optimization technique of introducing semi join reductions to filter the partsupp relation earlier [34, 146]. However, since we directly

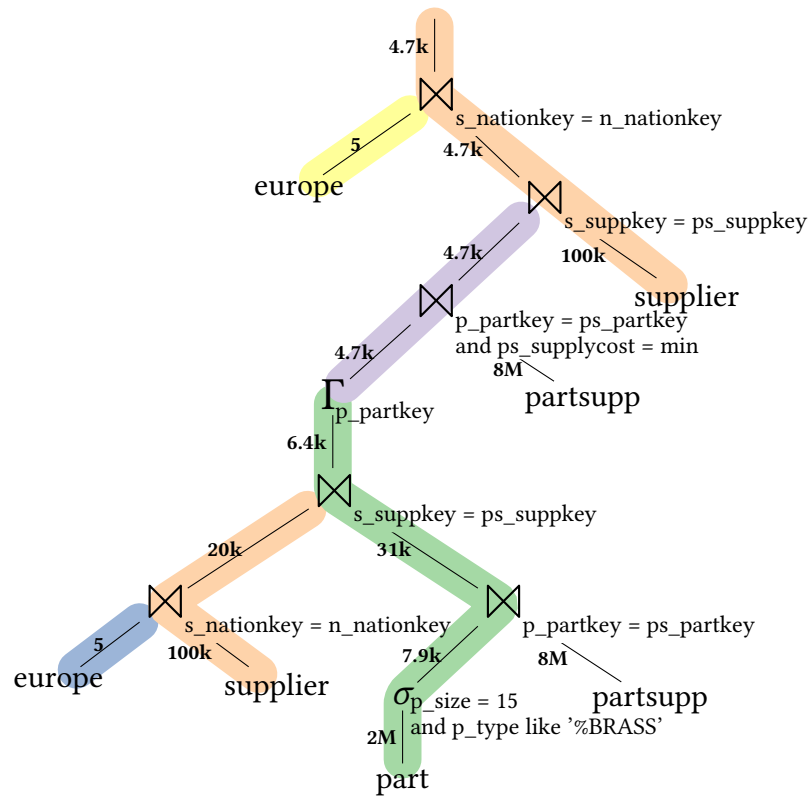


Figure 2.20: Final plan for TPC-H Q2.

join with the interesting parts, we can avoid duplicating the work of selecting the correct parts. Besides, in the final query plan shown in Figure 2.20, we do not build a hash table over the parts, but use index joins, as the filtered part table is about three orders of magnitude smaller than its join partner. Thus, we can avoid the costly full table scan of the largest table in this query, partsupp, which greatly improves its runtime.

TPC-H Q20 identifies suppliers that have an excess of parts, that it determines via an aggregation over lineitem in a dependent subquery. In this subquery, we join with the partsupp relation, which we unnest initially as an outer join with the aggregate. When we collect the join graph for this query, Algorithm 4 detects that this is a groupjoin and pushes the join with part and partsupp down the aggregation. Then, we estimate the predicate on the nested aggregate using the statistical method presented in Section 2.5, which reduces the estimation q-error from 2.4 with no statistics to 1.2. Then, our join optimizer determines the optimal join order for the input of the nested aggregate, where we join early with the parts we are interested in. As a consequence, we execute the join we

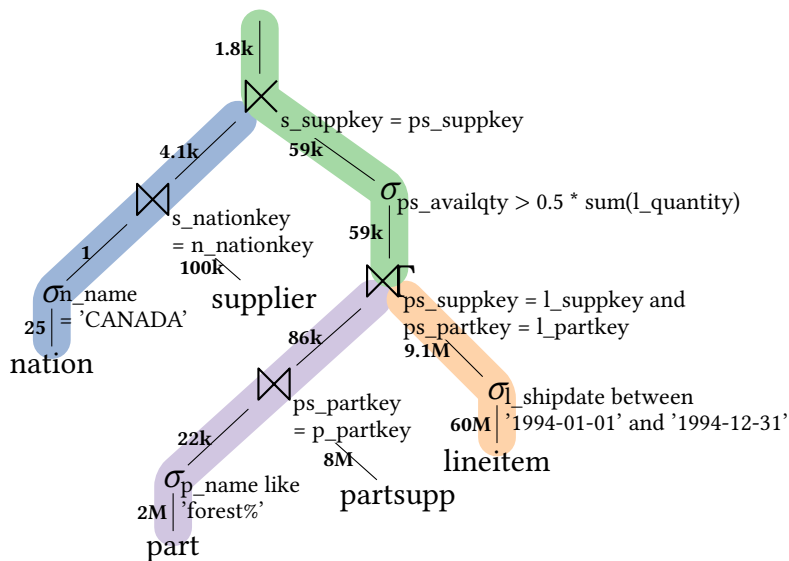


Figure 2.21: Groupjoin query plan for TPC-H Q20.

identified as a groupjoin last and introduce a groupjoin again, arriving at the plan shown in Figure 2.21.

Impact In both queries, this technique aids the performance of unnested aggregates. For correlated predicates in an aggregate subquery, we transform these into a group-by key and a join over this key. When this additionally is a foreign key, our presented transformation of conceptually introducing groupjoins before reordering additionally improves these plans. Note, that this is not limited to automatically decorrelated queries, but also if the query had been flattened manually. In addition to the performance improvement already shown in the Evaluation in Section 2.4 and 2.6, we get additional speedups. While the two example queries from TPC-H previously had no performance changes, the changes in our query planner now result in a 67 % speedup in Q2 and a 35 % speedup in Q20.

2.8 Eager Aggregation

In Section 2.7.1 we have explored pushing a group-by below a join in the context of groupjoins. In this section, we evaluate eager aggregations in general, pushing group-by operators further down a join tree. We propose a fast greedy approach to eager aggregation with a near-optimality guarantee, that results in significant

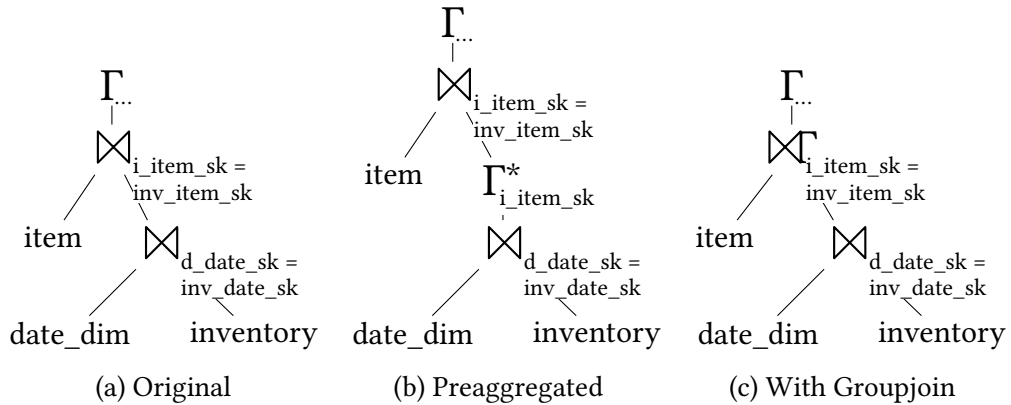


Figure 2.22: Eager Aggregation of TPC-DS Q22.

speedups to 11 TPC-DS queries with queries 2, 22, and 59 almost reaching a 3× speedup.

Pushing a group-by down a join reduces the amount of tuples going into the join by eliminating duplicates. However, duplicates are uncommon, and eager aggregations can be costly. Thus, an eager aggregation strategy needs to intelligently identify the beneficial cases in a cost based manner.

Our eager aggregation builds on the fundamentals by Yan and Larson [155]. Chaudhuri and Shim [24] introduce a greedy heuristic for introducing eager aggregation into a join tree, however, their heuristic lacks strong guarantees on the cost of the resulting plan. We introduce an improved greedy heuristic that is guaranteed to generate plans that are optimal in cost up to a constant factor. Eich et al. [40] incorporate eager aggregations into existing dynamic programming approaches to generate optimal plans, but their approach incurs an exponential increase in optimization time. Using our greedy heuristic, near optimality can be achieved without sacrificing optimization time. In the following we summarize the fundamentals and discuss the integration of our approach into a modern optimizer. We additionally describe a cardinality estimation technique for pushed down group-by operators that allows for more consistent and reliable estimations.

As motivating example, consider the query tree of the TPC-DS Query 22 in Figure 2.22a. This query first looks up the inventories of a certain date range, before finding their corresponding items, and aggregating some business metrics. However, in the end we are only interested in results *per* item. So, we are only interested in the aggregates on `i_item_sk` that we need to execute the following join, but not any particular sale or date. This means that we can preaggregate before the join with `item` and significantly reduce the costs of the join. Figure 2.22b shows the preaggregated query tree. Note that we first

group on $inv_date_sk=i_item_sk$ and then group again on other attributes of item. As the newly placed group-by and the join directly above have the same hash table key we merge them into an eager groupjoin in Figure 2.22c (cf. Section 2.3.1). However, this groupjoin has the slight twist that we relax Precondition ② and allow duplicates, since we aggregate them properly later.

With this strategy, we can reduce the incoming tuples into a join with an additional group-by, which we refer to as preaggregation Γ^* . To generalize this example, we first discuss when and how preaggregations can be placed to determine the space of possibilities for eager aggregation. Then we reduce this space to useful eager aggregations with a cardinality estimation strategy and a corresponding cost model. This model lends itself to a greedy approach, which we use to find near-optimal arrangements of eager aggregations.

2.8.1 Placement of Preaggregation

There are almost no limits to where we can apply eager aggregation within a query [61]. We show this by first discussing how to represent duplicates of tuples, and how eager aggregation allows us to switch between these representations at will. Then, we study an example of the steps that we take to apply eager aggregation while guaranteeing that the resulting query is equivalent to the original query.

Relational algebra, as extended by Grefen and de By [58], works on multisets (or bags) of tuples. A multiset can contain multiple entries of the same tuple, where we refer to the amount of duplicates of a tuple as its multiplicity. Multisets can be represented practically in two main ways:

1. All duplicates are stored and processed as separate copies, i.e., the expanded representation. This represents tuple a with three duplicates and a singleton b as $\{a, a, a, b\}$
2. The multiplicity is stored within the tuple as an additional attribute m , i.e., the (strongly) aggregated representation. We represent our example tuple as $\{a^3, b^1\}$, where the superscript denotes the tuple's multiplicity.

The expanded representation of multisets is widely used, easy to implement, and performant. However, it results in repeated work for duplicates, as the same computations have to be performed for different copies of the same values. The aggregated representation allows us to eliminate such inefficiencies, but is difficult to maintain. As different operators are applied and the projection onto the required attributes shrinks, duplicates can arise, unless tuples are constantly re-aggregated after every operator. We define the weakly aggregated

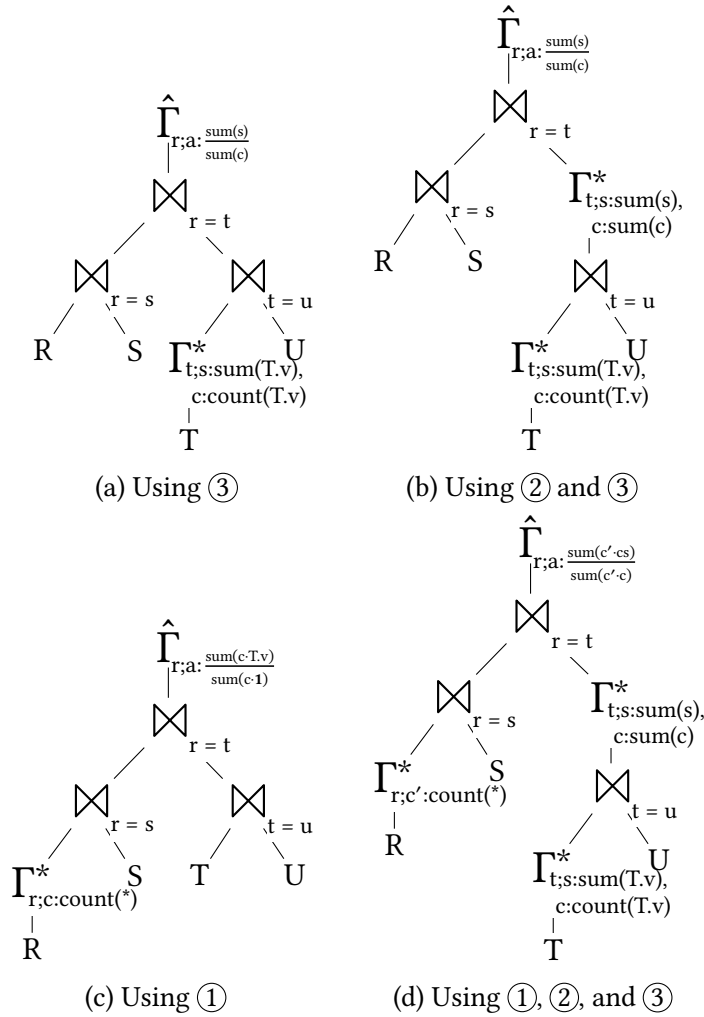


Figure 2.23: Possible plans using the potential preaggregation placements.

representation, that has both the desirable qualities of the strongly aggregated representation, but is easier to maintain. The weakly aggregated representation may contain multiple entries of the same tuple with different multiplicities. $\{a^3, b^1\}$, $\{a^2, a^1, b^1\}$, $\{a^1, a^1, a^1, b^1\}$ are all valid weakly aggregated representations of $\{a^3, b^1\}$. The multiplicity of a tuple a in a weakly aggregated representation is the sum of all the multiplicities of all the occurrences of a .

Eager aggregation allows us to intelligently switch from the expanded representation to the aggregated representation. To switch to the aggregated representation, we place a preaggregation operator [88] which can also be interpreted as a generalized projection [61]. Preaggregations should eliminate as many duplicates as possible, i.e., aggregate together as many tuples as possible, while

guaranteeing that the query result is correct. When placing a preaggregation, we transform the data into the strongly aggregated representation, then do small modifications above the preaggregation to efficiently maintain a weakly aggregated representation. While we use group-bys as our preaggregation operators, a partial preaggregation operator [88] that produces tuples in weakly aggregated representation may be used analogously. If the query result, or a specific operator requires the expanded representation as input, we can use an expand-operator [61] to transform the data back to that representation.

Preaggregations can be applied in any join tree. If the result of that join tree is to be aggregated with a group-by, it is more likely that preaggregations will be useful, as the existence of the group-by in the query implies that there likely will be duplicates in the join tree's result. Thus, we will focus on applying preaggregations below a group-by, or similarly, a non-duplicate sensitive operator such as distinct, or the set operations intersect, union, and except. In these cases, an expand-operator is not even needed, as the result of the join tree can be directly processed in the (weakly) aggregated representation.

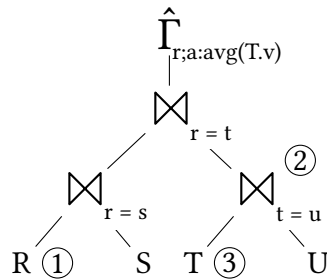


Figure 2.24: Potential preaggregation placements.

Consider the example join tree below a group-by shown in Figure 2.24. Suppose our optimizer determines that duplicates are likely at points ①, ②, and ③ during the execution. Thus, we want to place preaggregation operators there to eliminate duplicates and speedup the execution. To simplify, we assume that all aggregates of $\hat{\Gamma}$ are decomposable [40], which is true for our example with the *avg* function.

By introducing a preaggregation operator Γ^* , we eagerly aggregate the required attributes for $\hat{\Gamma}$ in the subtree below Γ^* . All non-preaggregated attributes, e.g., $\hat{\Gamma}$'s key and join predicates, form the key of the preaggregation. When an attribute is preaggregated, it is split into three operations:

1. Initial aggregation, which computes an aggregate on attributes that have not been preaggregated.
2. Merge aggregation, which aggregates on preaggregations.

3. Finalization, computing an expression on aggregations.

For $avg(v)$, the initial aggregation is $s : sum(v), c : count(v)$. In an intermediary step, we merge them: $s : sum(s), c : sum(c)$. We then calculate the final aggregate as $\frac{s}{c}$. In Figure 2.23a, we place a preaggregation at ③, where it calculates the initial aggregates, while $\hat{\Gamma}$ merges and finalizes them. Figure 2.23b shows an additional preaggregation placed in between at ②, which merges the aggregates from its input.

If we also preaggregate at ①, we need to compute and propagate the multiplicity with a $count(*)$ aggregate. The top aggregate in Figure 2.23c then uses this multiplicity for the finalization of its duplicate sensitive aggregates, by multiplying their inputs with the multiplicity.

We refer to the correction of aggregates with a multiplicity as a multiplication mapping, which we also utilize for computing groupjoins in Section 2.3.1. We maintain a weakly aggregated representation of the tuples by using these multiplications. Such a multiplication is needed when both sides of a join are preaggregated. Aggregates of the left side are multiplied with the multiplicity of the right side and aggregates of the right side are multiplied with the multiplicity of the left side. The multiplicities of both sides are multiplied, resulting in the output multiplicity. Note that the multiplicity for the empty side after an outer join is 1. A multiplication is also required when only one side of a join is preaggregated, but attributes from the other side will be later aggregated above the join. Figure 2.23d shows an example with all preaggregations applied. In summary:

- We first determine suitable aggregates of $\hat{\Gamma}$, and include any free attributes in the preaggregation key.
- Then, we decompose the aggregates of $\hat{\Gamma}$, depending on whether the input already is preaggregated.
- Lastly, we apply multiplication mappings to maintain correct weakly aggregated representation.

For a detailed listing of decompositions for various aggregates and eager aggregation transformations, we refer to the works by Gupta et al. [61] and Eich et al. [40].

2.8.2 Cardinality Estimation Strategy for Preaggregation

To find the optimal plan, join order optimizers estimate the cardinalities and costs of many plans, which means this process should be both fast and accurate.

As we want our join optimizer to consider preaggregations as well, we need fast and accurate estimations for preaggregation result cardinalities. Additionally, we need our estimations to be consistent. Comparing the costs of plans with preaggregations and without *should make sense*.

To estimate the cardinality of preaggregation, one can naïvely use the standard cardinality estimator of full group-by. This approach has multiple issues. Firstly, the estimators for a full group-by are generally more involved and slower than the join size estimators used within an optimizer. Secondly, optimizers have issues meaningfully comparing join and group-by estimations as estimators tend to underestimate joins and overestimate group-bys in many systems including but not limited to Umbra [113] and PostgreSQL. This is due to the contradictory way the independence assumption is generally applied to these two operators. The cardinality estimates of joins are usually based on the multiplication of individual selectivities of predicates, resulting in underestimations. The cardinality estimates of group-bys are usually based on the multiplication of the domain sizes of the key attributes, resulting in overestimations. These estimations are often “corrected” with a variety of heuristics, which do improve the results, but do not fundamentally change their shortcomings.

We neither want to reuse group-by estimators nor introduce a new cardinality estimator specifically for preaggregations. Regardless of how accurate such an estimator is, it will not be useful unless its estimations are sensibly comparable with join size estimations. Thus, we propose a simple strategy that relies on cardinality estimations of semi joins.

We know that $|X \bowtie_a \Gamma_a(Y)| = |X \times_a Y|$ for two relations X and Y . Thus, we want to pick an estimate for $|\Gamma_a(Y)|$ in such a way that the estimate of the upcoming join cardinality $|X \bowtie_a \Gamma_a(Y)|$ would be the same as the estimate for $|X \times_a Y|$. So, our estimate for $|\Gamma_a(Y)|$ is the size of Y' such that $|X \bowtie_a Y'| = |X \times_a Y|$. If we use simple estimators for join and semi join based on (relative) selectivities, this results in the equivalence $|\Gamma_a(Y)| = \frac{\sigma_X}{\sigma}$, where σ is the selectivity and σ_X is the relative left selectivity of the join. This estimate fits well into a join optimizer that also considers preaggregations, as it needs to compare the cardinalities of plans of different join orderings and preaggregations. By using a common system to estimate both joins and preaggregations, we avoid introducing inconsistent estimates, which would result in the optimizer making subpar decisions.

We have shown our estimate for the case when preaggregation and join share the same key. If the preaggregation’s key contains additional attributes from base relations which are not key sides of a key-foreign key join, these attributes may cause the number of distinct groups to increase. Let us consider

the general case in the presence of multiple joins and additional attributes.

- \mathcal{R} := the join tree of a subset of relations
- \mathcal{J} := upcoming joins of the form $S \bowtie \mathcal{R}$ where $S \notin \mathcal{R}$
- \mathcal{K} := the key attributes of the preaggregation
- $\mathcal{A}(R)$:= the attributes of a relation R
- $dv(R_i)$:= Estimate for $|\Gamma_{\mathcal{K} \cap \mathcal{A}(R_i)}(R_i)|$ for a relation R_i using an existing group-by estimator

Then our estimate v for $|\Gamma^*(\mathcal{R})|$ is:

$$v = \prod_{j \in \mathcal{J}} \frac{\sigma_S(j)}{\sigma(j)} \cdot \prod_{R_i \in \mathcal{R}} \begin{cases} 1, & \text{if } R_i \text{ is key side of a } j \in \mathcal{J} \\ dv(R_i), & \text{otherwise} \end{cases}$$

2.8.3 Integrating Eager Aggregation into an Optimizer

Eager aggregation can theoretically speed up queries significantly, as there is no upper bound on the number of duplicates in multiset relational algebra. However, in reality, a high percentage of joins are key-foreign key joins [39] which do not produce any duplicates and prevent many eager aggregations, as any preaggregation within the key side of the join needs to contain the key, which makes such a preaggregation useless. So we need an efficient intelligent optimizer that can recognize when eager aggregations will be useful, apply eager aggregation when needed for significant gains in performance, and avoid them when they are not needed. Additionally, with the placement of eager aggregations, the optimal join orderings for a query can change as an eager aggregation can significantly reduce the cardinalities of subtrees. This further indicates the need to deeply integrate eager aggregations into a join optimizer.

Chaudhuri and Shim [24] propose a conservative (constant additional time per generated plan) and greedy (locally evaluated costs) optimization technique for eager aggregation. This technique extends an existing optimizer with an additional step that places a preaggregation directly below a join, if it locally improves the cost for that join. This is guaranteed to globally improve costs, as placing a preaggregation only decreases cardinalities and, thus, decreases costs above the preaggregation. However, this technique does not guarantee to generate optimally preaggregated plans. Instead, it only improves the (imperfect) plans of the original optimizer that does not consider preaggregations. Note that the best plan without any preaggregation can have a significantly higher cost than the optimal plan with preaggregations.

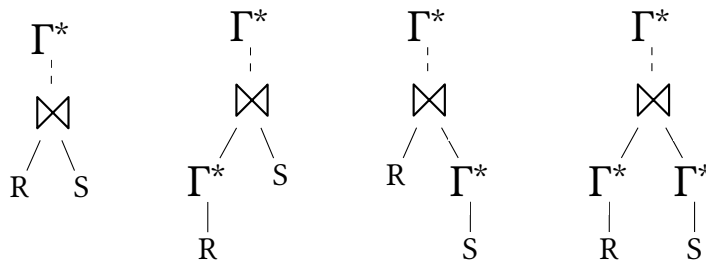


Figure 2.25: Four alternative plans considered by the optimizer. All costs contain the cost of a preaggregation on the join.

Let T^* be the globally optimal plan among all possible plans with and without preaggregations. Let J^* be the best plan without any preaggregations. In the worst case, the overhead denoted by $\frac{C(J^*)}{C(T^*)}$ can be on the order of $\mathcal{O}(n^k)$ where $C(T)$ is a cost function, relations have sizes $\mathcal{O}(n)$, and there are $\mathcal{O}(k)$ relations. So J^* can have an exponentially higher cost than T^* . The plans generated with Chaudhuri and Shim's greedy technique is guaranteed to have a cost lower than J^* . However, this is a really high upper bound for queries where eager aggregation may be extremely useful, such as queries with multiple N-to-M joins. We propose a slightly different conservative greedy technique which is guaranteed to generate plans with costs $\mathcal{O}(C(T^*))$, meaning that our technique generates plans which are at most a constant factor larger than the globally optimal plan.

We start with the optimal query plan T^* among all possible plans, including plans with preaggregations. We take this plan T^* and place additional preaggregations after every single join in a risk averse way. We call the new fully preaggregated plan $f(T^*)$. As T^* was optimal, placing these preaggregations increased the cost of the plan. However, as a preaggregation operator can not produce more tuples than it consumes, and assuming the cost of a preaggregation operator is linear in its input size, placing additional preaggregations can only increase costs by a constant factor. Thus, $f(T^*)$ is optimal up to a constant factor. This implies that any fully preaggregated plan $f(T')$ better than the fully preaggregated optimal plan $f(T^*)$ is also optimal up to a constant factor.

It is trivial to modify a join optimizer to find the best fully preaggregated plan as the structure of fully preaggregated plans are simple; every join is always followed by a preaggregation. The optimizer should simply place a preaggregation on top of every (sub-)plan it evaluates. As we have shown, such a plan is guaranteed to be optimal up to a constant factor as it would have a lower cost than $f(T^*)$, the fully preaggregated version of the optimal plan. However, a plan with preaggregations everywhere is suboptimal. Thus,

we can use a greedy approach to remove preaggregations and improve the generated plan even further. We iterate on every join, bottom-up, and remove the preaggregation on the join’s left and/or right, if this locally improves the cost for the subtree including the preaggregation above this join. As we also consider the preaggregation above the join, and as the preaggregation’s output size is not dependent on its input’s size, this operation is guaranteed to reduce costs globally as well. The four alternative plans that are considered are shown in Figure 2.25, where the join’s inputs are denoted by R and S . This approach guarantees the cost upper bound $\mathcal{O}(C(T^*))$.

In our database system Umbra, we have integrated this optimization strategy into the adaptive optimization framework [116] which can compute optimal join plans for small queries using DPHyp [104] and high quality plans for larger queries using LinDP++ [125] and iterative DP [86]. With the consideration of these 4 preaggregated alternatives when building operator trees, the adaptive optimizer is able to generate high quality plans with preaggregations for a wide variety of query sizes.

Note that our approach is equivalent to the Chaudhuri and Shim [24] approach for a simple preaggregation cost function that does not depend on the input size such as $C_{bad}(\Gamma^*(T)) = \mathcal{O}(|\Gamma^*(T)|) + C(T)$. Such a cost function is not desirable as it underestimates costs and results in preaggregations being placed too eagerly. For example, in TPC-DS SF10 Query 22 as shown in Figure 2.22a, an additional group-by above `inventory` is placed when C_{bad} is used. This does improve the join with `date_dim`, but not enough to be worth the group-by’s cost. Thus, we use a cost function of the form $C(\Gamma^*(T)) = |T| + \mathcal{O}(|\Gamma^*(T)|) + C(T)$.

A final optimization step after the generation of preaggregation operators is to pull up these preaggregations into the joins above when possible, thus generating eager groupjoins instead of a join and a group-by. This final optimization step can result in significant performance improvements as one less pipeline needs to be generated and processed.

2.8.4 Evaluation of Eager Aggregation

We have evaluated our eager aggregation strategy on the TPC-H and TPC-DS Benchmarks. Both these benchmarks contain many join trees below group-bys. However, most of their queries are not amenable to eager aggregation, as most of the joins below group-bys are key-foreign key joins which are unlikely to produce duplicates. Thus, the eager aggregation strategy does not change the plans generated for the TPC-H queries. However, eager aggregation results in significant improvements to some queries in TPC-DS.

For TPC-DS SF 10, our optimizer places preaggregations in 22 out of 103 queries with a geometric mean speedup of around 20 % for those 22 queries.

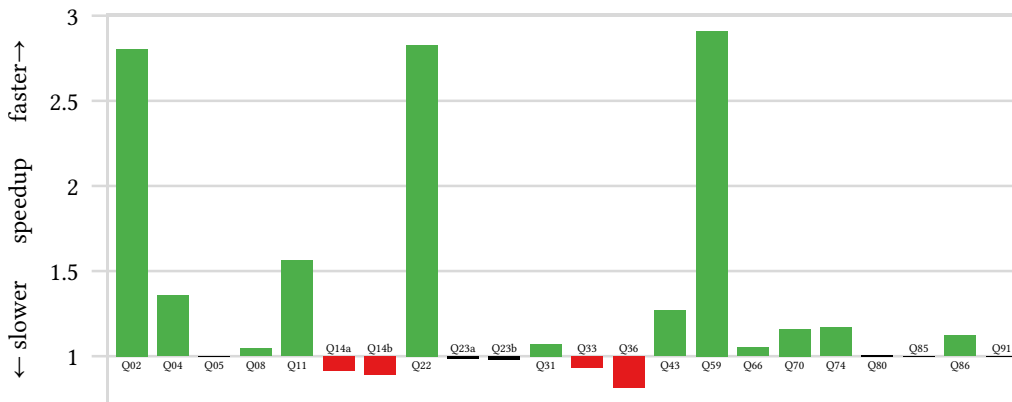


Figure 2.26: Overall Impact of Eager Aggregation on TPC-DS.

Figure 2.26 shows the relative speedups of individual queries. Q2, Q22, and Q59 with eager aggregation applied run more than 2.8 times as fast as their non-eagerly aggregated counterparts.

The queries with the biggest wins have a costly top group-by but simple preaggregations. Their group-bys contain multiple attributes, some of them strings, and may require additional processing for features such as ROLLUP. As join predicates primarily use integer keys, preaggregations below joins have integer keys as well. The costly attributes are functionally dependent on the integer keys; thus, they can be excluded from preaggregation keys. While our cost functions do not consider the costs of processing individual attributes, we tend to overestimate cardinalities of group-bys with larger keys, causing the optimizer to prefer simpler preaggregations.

2.9 Groupjoins in Detail

For a qualitative analysis on the impact of groupjoins, we now take a detailed look at the queries in TPC-H that benefit from using groupjoins. In the previous sections, we already saw improved query plans. In this section, we focus on the operator selection for physical execution. Choosing the right physical execution strategy improves performance considerably, but is still sensitive to imperfect cardinality estimates.

We visualize query plans in tree form and mark our code generation pipelines with colored regions on its branches. The lowest operator of a pipeline typically generates a loop that drives the query execution. Intermediary operators then execute the query logic, before the pipeline ends at a materialization point, e.g., building a join hash table. In our interactive online query plan demonstra-

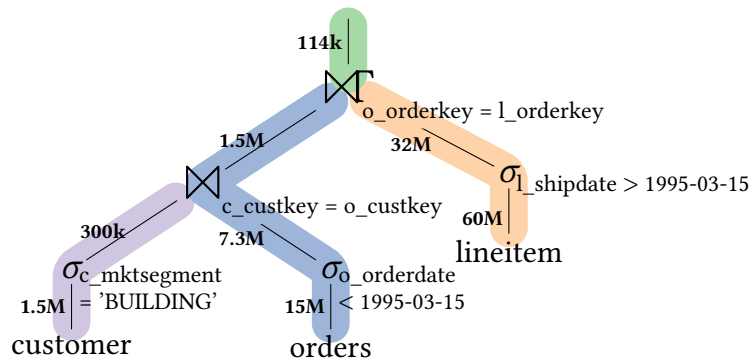


Figure 2.27: Query plan for TPC-H Q3.

tion⁶, we use the same notation and provide additional details of intermediary optimization and cardinality estimation.

When edges between a base table and a join are not part of a pipeline, i.e., not included in a colored region, we determined it advantageous to use an index to access the table. We additionally mark all edges between operators with the cardinality of tuples that are produced on the TPC-H scale factor 10, which allows us to reason about the quality of our produced plans.

TPC-H Q3 We execute this query with a groupjoin for the top level aggregate. During optimizing of the group-by, we determine the equivalence relation between `l_orderkey` and `o_orderkey` through the join condition. Thus, we can eliminate the functionally dependent keys `o_orderdate` and `o_shippriority`, and, since we now join and aggregate over the same key, we introduce a groupjoin.

Figure 2.27 shows Umbra’s execution plan for this query. In contrast to the cost model calculation in Table 2.3, our optimizer chooses to execute a suboptimal memoizing groupjoin. This is mainly caused by a misestimation of the relative left selectivity of the groupjoin. In this query, the filter predicates on the `lineitem` and order dates are correlated, but Umbra’s estimations do not consider correlations [79, 94]. In turn, only about a tenth of the groups in the memoizing hash table have a match, which wastes relatively much space.

The execution of this query spends most time in the groupjoin. Figure 2.28 shows a trace of the operator activity over the execution [13]. Note that to get a precise trace, we gathered it on a recent Ice Lake system and limited parallelism, so the total execution time is not comparable to our previous evaluation. In the plot, the relative frequency, combined with the execution time, indicates the cost of executing operators. When focussing on the groupjoin, we observe its presence in four phases. In its first phase, it works in a pipeline along the table

⁶<https://umbra-db.com/interface/>

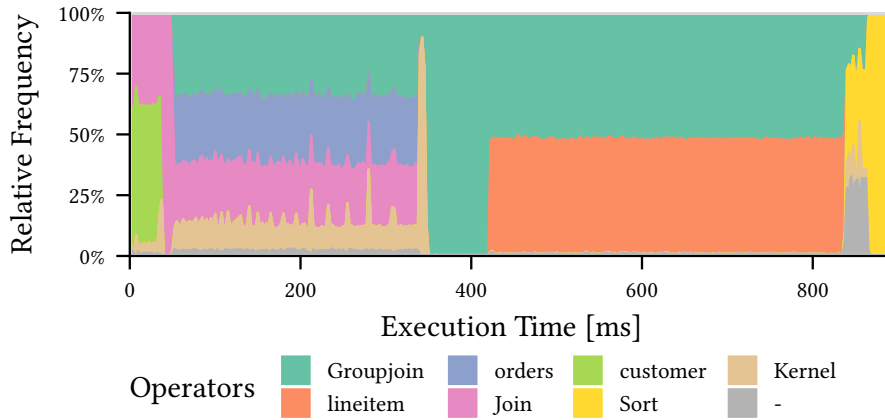


Figure 2.28: Operator Trace of TPC-H Q3.

scan of orders and the join with customer, and collects tuples in kernel allocated memory. In the second phase, it builds the hash table, before probing it with the tuples of lineitem, and scanning the results to sort and output them.

TPC-H Q13 Figure 2.29 shows the physical execution plan of this query. In it, we first calculate a nested aggregate, where we count how many orders each customer has made. However, we also want to consider customers that have no recorded orders, so this query needs proper outer join semantics in the groupjoin. Since we match all customers, an eager right aggregation strategy is very advantageous.

After probing the eagerly built right aggregation hash table with the customer keys, we continue with the next aggregation that calculates the histogram over the amount of orders. This way, we directly aggregate the customers in one of the few top level groups. This pipelining, as indicated by the marked colors, keeps tuples hot and usually directly in CPU registers. In effect, we execute this query with two small, hot aggregation loops.

TPC-H Q17 The query plan shown in Figure 2.30, again benefits from unnesting. While we do have a static whole subquery aggregate `avg(l_quantity)`, this aggregate does not require a complex unnesting that would require all empty aggregates of the domain. Instead, we directly use all parts and join them with a groupjoin with “inner join” semantics.

Since we only have two predicates directly on the part table, we have correct estimates, and we confidently know that the qualifying parts are four orders of magnitude fewer than lineitem. Thus, we use the foreign key index on `l_partkey` and calculate the groupjoin aggregate pipelined, and directly continue with

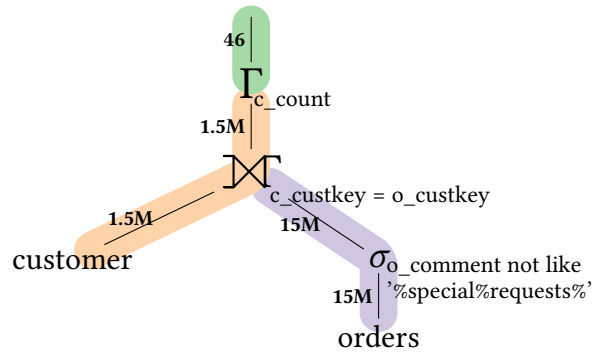


Figure 2.29: Query plan for TPC-H Q13.

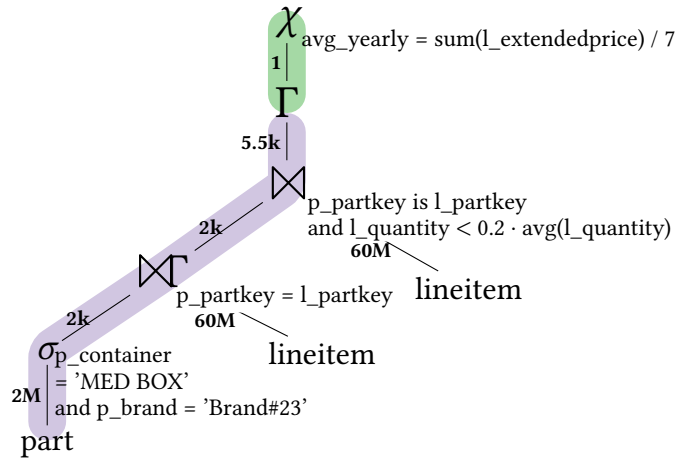


Figure 2.30: Query plan for TPC-H Q17.

another probe of the same index. We then sum up the price of the matching tuples in the top level, before finally calculating the yearly average.

On a tangent, this still does some duplicate work, since we probe `lineitem` twice and check the partkey equivalence condition twice. As a theoretical improvement, one could execute both, groupjoin and join, with a single index lookup using the partkey. We could first calculate the aggregate for this key, before resetting and scanning the current probe index cursor again, this time checking the quantity predicate.

TPC-H Q18 This query has several interesting challenges. As we already discussed in Section 2.5, estimating the having predicate on the nested aggregate ($\text{sum} > 300$) is challenging. Originally, we also had a semijoin with orders as the literal translation of the `IN` expression. We transform it to an easier to execute inner join in a subsequent optimization, since we join over the key of the nested

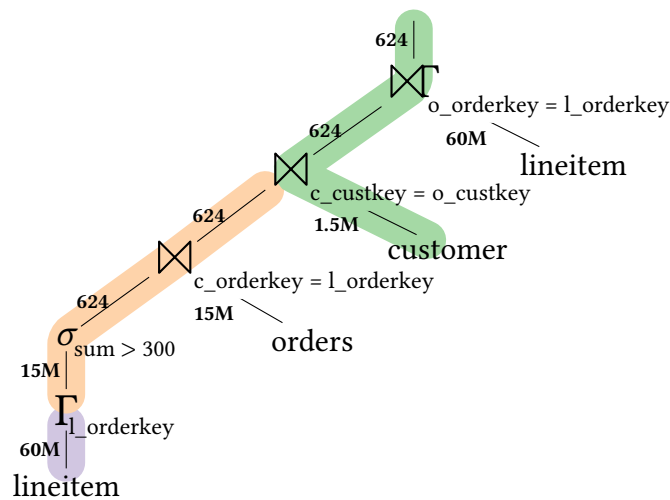


Figure 2.31: Query plan for TPC-H Q18.

aggregate, which results in the query plan shown in Figure 2.31. For the top level aggregate, we also use our knowledge of such functional dependencies of the key and drop the four functionally dependent keys, which allows the combined groupjoin with `lineitem`.

For physical execution, the theoretically best approach would be to use three index lookups for `orders`, `customer`, and `lineitem`. However, our estimates are still too uncertain, and we still estimate the left side relatively large. Umbra currently does not use in-memory optimized indexes like ART [92], but uses a more traditional disk-oriented B-tree index [11]. Since these index lookups are relatively expensive, we only use them when they are definitively faster than a full table scan. For the smaller tables, our current heuristics do not use the index, but for the large groupjoin with `lineitem` using the foreign key index is very advantageous.

Observations Similar to the improvements of Q2 and Q20 that we saw in Section 2.7.2, unnesting is again one of the key techniques for Q17. Since we first calculate the tiny domain, we can use a very efficient index lookup over `lineitem` to calculate the groupjoin result.

Using index lookups to answer queries is often very profitable, but one limiting factor for them are estimation errors. For example, in Q18, we use a suboptimal query plan that does not use the indexes of `orders` and `customer` due to our low confidence in the estimation of the nested aggregate. While our aggregate estimates from Section 2.5 already improve these, we still estimate

that several thousand sums qualify the condition, and we only choose to use the index for the largest join partner, `lineitem`.

Using the index for the groupjoin in Q17, results in a $6 \times$ speedup for scale factor 10. In Q18, the shown query plan has an 15% speedup compared to using a memoizing groupjoin. If we estimate the having predicate accurately and correctly identify that few tuples qualify, we would also use an index for `customer`, however the performance impact is negligible.

The tracking and inference of functional dependencies is another fundamental optimization for these queries. These allow us to minimize the grouping keys to only the candidate key over which we join with another table. In TPC-DS, we can use significantly fewer groupjoins, since the functional dependencies are not directly included in the schema. Although the DS schema defines primary keys, it additionally defines so-called business keys, which we cannot represent in the schema. Many aggregates then use the business keys, where we miss the functional dependency to minimize the keys and thus do not produce optimal execution plans.

2.10 Related Work

As outlined in Section 2.2, our work relies on well-known work on query unnesting, which enables aggregates to be embedded in the query tree [30, 42, 77, 114]. Subquery unnesting to flatten the query tree is well-known as one of the most important aspects of query optimization [20, 34]. Galindo-Legaria and Joshi [51] describe the comprehensive optimization of aggregation in Microsoft SQL Server. They describe the reordering of group-by and outer-join, where they use similar conditions to our groupjoin preconditions (cf. Section 2.2.1) and also discuss the problems with `COUNT` in static (scalar) aggregation. In contrast to our work on groupjoins, they keep join and aggregation separate, where a pushed-down group-by will still build a redundant hash table.

Bellamkonda et al. [14] describe the execution of correlated subqueries with window operations in Oracle. Hölsch et al. [66] use an extended form of relational algebra to reason about nested queries and are able to express more transformation on aggregations. To incorporate unnested aggregations in cost models, practical implementations, e.g., in DB2, use statistical views [41]. However, each query needs a matching view, which are relatively costly to create and maintain, and are usually only created where missing statistics lead to very poor plans.

In many real-world evaluations, join and aggregation are big contributors to the overall workload [73, 150]. Consequently, there is a large body of related work that optimizes hash joins [97, 115, 132] and hash aggregations [95, 122,

156]. Re-using hash partitions, and even whole hash tables is a well-known optimization [30, 71]. One often discussed question is, if hash tables should be partitioned or non-partitioned [10]. Our proposed approaches in Section 2.3 try to use a non-partitioned hash table to avoid materializing data, while using thread-local partitioning for heavy-hitters. Other recent work on the interaction of multiple operators focused on memory access patterns to better utilize the available hardware [27, 100]. We see this work as orthogonal, and these ideas can work hand-in-hand with parallel groupjoin execution.

2.11 Conclusion

In this chapter, we improved several aspects for an efficient evaluation of joins and aggregates in a general-purpose relational database management system. We improve important pieces of the query engine that previously have not worked well with nested aggregates. First, we presented a low overhead estimation of computed columns, which significantly improves the estimates that we use to find better query plans in the query optimizer. Our aggregate estimates result in a near 50 % reduction of estimation error, without any changes to the underlying sampling method.

Furthermore, we improved the execution of groupjoins, which commonly occur in nested and regular aggregation queries. Our contention-free parallel and index-based execution allows them to be more universally applicable. We also demonstrated where using a groupjoin is advantageous and presented a simple, yet effective cost model to plan the best execution strategy. With our improved groupjoin execution, we achieve significant speedups in several TPC-H queries.

Building on our improvements to groupjoins, we presented an eager aggregation technique that significantly improves execution plans with minimal regressions. It consists of a simple cardinality estimation strategy, and a novel greedy conservative optimization approach to introducing preaggregation operators. When integrated into the adaptive optimization framework, this technique introduces preaggregations in 22 TPC-DS queries, resulting in approximately 20 % geometric mean speedup with scale factor 10, with 3 queries reaching almost a $3 \times$ speedup.

CHAPTER 3

Asymptotically Better Query Optimization Using Indexed Algebra

Parts of this chapter have been previously published in the Proceedings of the VLDB Endowment [46].

3.1 Introduction

Optimizing the algebra plan of a query can take a significant portion of the overall runtime. The challenge for the optimizer here is that the data flow through the query, and its analysis can be astonishingly complex. Additionally, automatically generated queries with complex business logic amplify this problem [33, 96, 98, 144]. Query optimizers struggle to deal with such complex input, which is especially painful for small datasets where query optimization can be more expensive than query execution. Small data sizes are common during testing, but also in the real world, where, for example, Tableau reports that many workloads contain fewer than a million tuples [150]. As a result, query optimization usually operates on a budget, trading-off optimizations versus optimization time.

Some typical questions that come up during query optimization are: From which part of the plan does a value come from? What are the join predicates? Can we push a predicate down into the inputs? Consider for example the SQL query below:

```
SELECT *
FROM   A, B, C LEFT OUTER JOIN D ON C.u = D.u
WHERE  A.v = 5 AND A.w = B.w AND B.x = C.x
      AND C.y = 7 AND D.z = 8
```

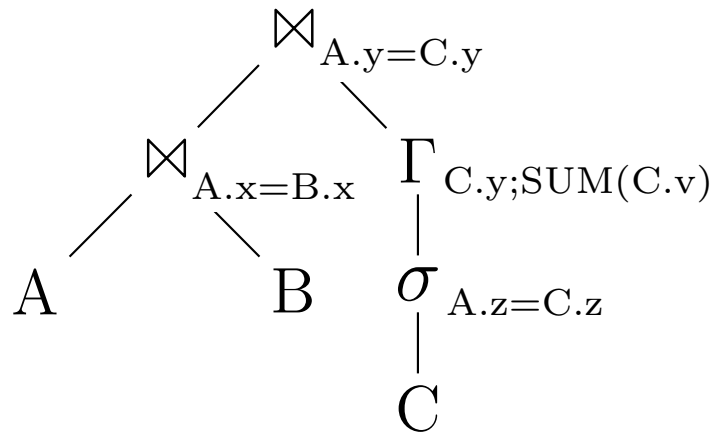


Figure 3.1: Relation algebra tree with subtle data flow. In this chapter, we optimize queries by efficiently analyzing data flow.

In this small example it is easy to see which attributes form join edges ($\{u, w, x\}$), which filters can be pushed down ($\{v, y\}$), and which not directly ($\{z\}$). In general, these questions are difficult because the FROM clause can contain arbitrary subqueries. The traditional solution to this problem is to keep track of the columns that are available in each step of the query in a set [52, 54] and to move predicates around step by step, checking the available columns in each transformation. But if we have a join tree of depth n , where each join produces at least one column, the construction time for these column sets grows with $\Omega(n^2)$, which is highly unattractive for large queries.

Even if we ignore the performance problems, this myopic look at individual operators is insufficient to express optimizations efficiently. In many cases we want to inspect the full data flow instead. Consider the small algebra tree shown in Figure 3.1. The top-most join predicate in this tree compares an attribute from its left input ($A.y$) with an attribute from its right input ($C.y$). While this data flow direction might be easy to see for such a small example, it is non-obvious when there are dozens of operators between the base tables and the predicate. And note that the example tree contains a non-trivial data flow that is not obvious at a first glance: The selection operator on the lower right uses an attribute that is produced in a different part of the operator tree, which effectively makes the top-most operator a *dependent join*. Evaluating such a join is highly inefficient, since it requires a nested loop join execution. The query optimizer has to detect these dependent joins and can then rewrite the query to remove the correlation between parts of the join tree [114]. While we can detect these dependent joins by reasoning over the columns available after each operator, this again leads to highly unattractive quadratic (or even cubic) algorithms.

What we need instead is a framework to reason about complex data flows without inspecting individual operators. For example, we want to be able to correctly push down a predicate deep into a query tree in one holistic operation, skipping all intermediate operators, and we want to detect dependent joins without traversing the operator tree. This not only makes query optimization more efficient from an asymptotic perspective, it also makes the optimizer more pleasant to write, as we can ask data flow questions about the query itself instead of traversing the operator tree all the time.

In this chapter we show how to answer these data flow questions that arise during query optimization with an *Indexed Algebra*. To avoid the frequent tree traversals, we maintain an auxiliary index of the algebraic query representation. This index answers data flow questions in $O(\log n)$ time and supports efficient query transformation. We implement this index structure as a link/cut tree [142] that builds a balanced search structure for the paths through our algebra. Using our Indexed Algebra to answer the data flow questions during query optimization is dramatically more efficient than traditional approaches, while additionally leading to concise and elegant formulations of optimization rules.

Indexed Algebra has the biggest advantage for complex queries, but also improves relatively harmless queries like the ones from TPC-H. Before developing Indexed Algebra, our research system Umbra [113] spent a significant time reasoning over columns. For example, in the intricate nested TPC-H Q2, which our system parses as 21 operators, Umbra spent more than 20% of the optimization time in naïve implementations answering data flow questions. Indexed Algebra helps to significantly improve this, while helping even more for complex queries.

The rest of this chapter is structured as follows: We first discuss the algebraic representation of queries and common operations on the algebra in Section 3.2. We then introduce the idea of indexing paths through the algebra in Section 3.3, and how to efficiently support dynamic updates to that index. Then, we propose several optimizations using Indexed Algebra in Section 3.5. Furthermore, we discuss other techniques like property caching in Section 3.6. The benefits of these techniques are demonstrated in Section 3.8, and related work is discussed in Section 3.9.

3.2 Query Representation

In this section, we introduce an algebraic representation that represents a query execution plan. We first discuss the general components of this algebra, before we show how we navigate and reason about this algebra.

3.2.1 Algebra: Operators, Expressions, and IUs

Operators. Like most relational systems, we represent a query in some form of extended relational algebra. *Operators* like joins \bowtie , selections σ , group by Γ , or table scans, make up the base structure of the algebra. The operators process a multiset of tuples from one or several inputs that have a clear parent-child relationship, where data conceptually flows from children inputs to the parent outputs. For now, we assume that an operator outputs its tuples to a single output operator, so that the query forms a tree of operators. We generalize this operator tree to a DAG in Section 3.6.3, and first concentrate on simpler tree structured queries.

Expressions. Attached to the operators are *expressions* that process scalar values instead of tuples, e.g., individual columns. Like operators, expressions are tree structured and can be arbitrary nested, however, the expression tree is anchored at exactly one operator. In contrast to relational operators, expressions are wide spread in general purpose programming languages, that also deal with reasoning and optimization of expressions. In Section 3.6.1, we detail relational algebra specific implementations that deal with expressions.

IUs. While our expressions consist of e.g., comparisons, arithmetic or constants, a relational algebra expression can also reference columns of data in relations. This way, an operator can execute, e.g., $x > 42$ as a filter, or $x = y$ as a join predicate. In our algebra, we generalize the notion of a column to an *information unit* (IU) [102]. IUs describe scalar values that can be referenced multiple times, e.g., as a cached expression for common subexpression elimination to avoid repeated computation.

IUs represent the data flow through a query. An IU has a single source operator, typically a table scan, that produces the values. Other operators then pass IUs through the algebra until they reach the consuming expressions. This makes IUs a key part of logical query planning, where we initially do not consider physical implementation details, like if an IU is pipelined and passed through a register or materialized in a hash table. Query optimization then brings the data flow of IUs in a form that is efficient to execute.

3.2.2 Efficiently Navigating Algebra

The relational algebra uses these components of operators, expressions, and IUs. While this is already enough to model and execute queries, optimizing this algebra is more complex. Simple analyses can be operator-local, e.g., to reorder predicates or to fold constant expressions. However, most optimizations require a structural analysis of the data flow through the relational algebra. For example, to drop unused IUs, we need to know which expressions, if any, use

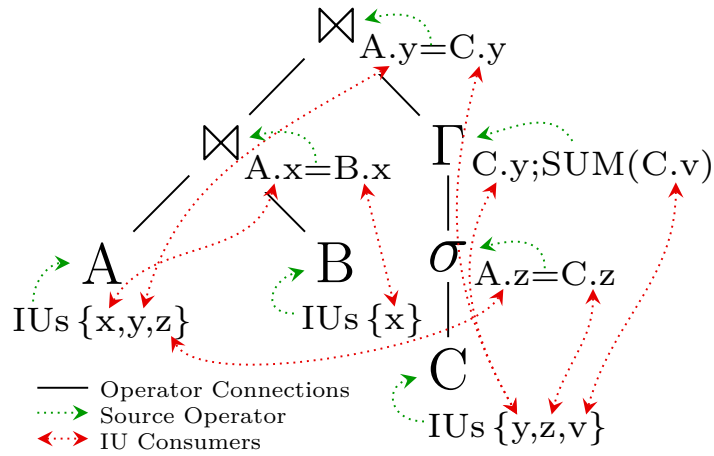


Figure 3.2: Our algebra representation interlinks operators, expressions, and IUs to allow efficient navigation.

them. Likewise, to move around select σ or map χ operators, we need to consider the source of any consumed IU.

For these analyses, we set up explicit links between the components of our algebra that allow to navigate it efficiently. In Figure 3.2, we visualize these links as dotted arrows between the components in the example algebra tree of the introduction. Operators are connected via their parent-child relationships, which allow traversing the algebra tree. Expressions also have hierarchical links and are additionally connected to their using operator, e.g., the top most comparison $A.y = C.y$ is linked to its join \Join , which transitively applies to the IU references in the expression. Similarly, IUs are linked to their source operator and are additionally connected to all references of that IU throughout the algebra.

With this setup of links, we can now traverse the tree to build reasoning grounds for optimizations. However, traversing this tree for each analysis, each optimization, each transformation, and each operator seems costly. Therefore, we want to build data structures that allow us to avoid duplicate traversal.

3.2.3 Reasoning about Column Sets

A traditional technique to reason about the query structure are column sets. For this method, we say that an operator produces a set of IUs, and an expression consumes a set of IUs for evaluation. This way, we can implement all kinds of optimizations, e.g., pushing predicates down a join, where we need to determine, which of the joins inputs produces the required IUs of the predicate.

Computing the required IUs of an operator also requires traversing the query tree. We can limit the algebra traversals by caching the produced IUs per

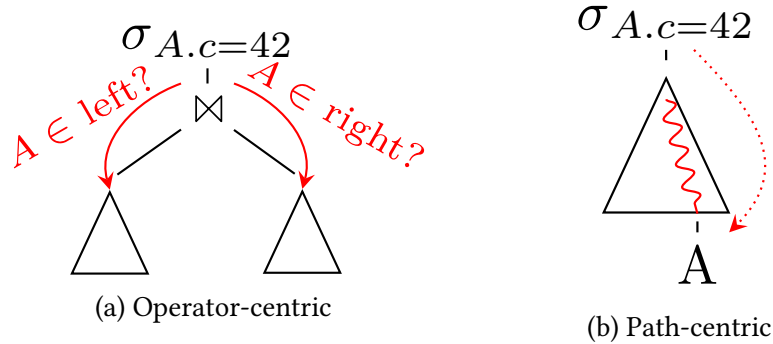


Figure 3.3: Traditional predicate pushdown traverses the query tree operator by operator. Path-centric optimization can take a shortcut to the IU's source, but still needs to check if the path is sound, or if it contains e.g., an outer join.

operator, which allows us to calculate all sets in a single pass over the algebra. Unfortunately, we now use dynamic memory per operator to cache the column sets. For an algebra tree of n operators, we also need to cache n column sets. For most queries, the size of the set of produced IUs also grows, since regular joins produce the union of their input columns, which makes the size of these sets in $O(n)$. Consequently, our cache stores $n \cdot O(n) = O(n^2)$ produced IUs.

While reasoning about column sets is quite efficient, building the column sets is already expensive. This caching cost is additionally amplified by wide tables containing dozens of IUs. Also, column sets lead to complex code since transformations of the algebra might need to invalidate the cached column sets, potentially triggering a quadratic recalculation. As we will see in the evaluation in Section 3.8, the maintenance of these sets is too expensive to be worthwhile.

3.2.4 Reasoning by Path Traversal

We argue that instead of reasoning about column sets, we should reason about the paths that IUs take through the algebra. Another way to look at these optimizations is if we argue locally for each operator (i.e., with column sets), or if we argue about the whole algebra tree. Figure 3.3 contrasts these two approaches for predicate push down. In the operator-centric approach, we reason from the perspective of a join, where we can either push the predicate to its left or right inputs. On the other side, in path-centric reasoning, we use the link between IU reference and the producing operator to directly get to the IU source. Then, we traverse the algebra bottom-up to get the push down path and to detect operators like outer joins, through which we cannot directly push predicates.

For path-centric reasoning, we identify two common basic operations: *Finding the root* of an operator tree and intersecting data flows by finding the *lowest common ancestor*. We can find the root of an operator tree by traversing tree bottom up, which allows, e.g., to find the (sub)tree that contains the source of an IU. The lowest common ancestor of two operators is the place where the two paths from these operators to the root intersect, and is useful, e.g., as the earliest possible place to evaluate an expression consuming data from two IU sources. Based on these operations, one can implement many optimizations, which we discuss in Section 3.5.

The main advantage of this path-centric reasoning is that it requires no dynamic per operator state. Avoiding this state entirely avoids the quadratic space requirements, and additionally simplifies the transformation logic, since it no longer needs to invalidate the column set caches. Still, deep query trees are a problem. The tree traversal for each analysis still can lead to quadratic runtime. Nevertheless, we found that it is still more efficient than building column sets.

While better than column sets, walking the raw algebra tree is still quite naïve. In the following, we propose Indexed Algebra to make path-centric queries efficient. Our index structure maintains a balanced path structure to make traversal fast, while amortizing the balancing of the index during algebra transformations. In effect, we get the simplicity of path traversal with the analysis performance of set operations.

3.3 Indexing the Algebra

To make query optimization analysis with path traversal efficient, Indexed Algebra uses an embedded index structure that makes direct traversal unnecessary. To build this index, we first start with the simpler problem of indexing static algebra trees in this section, before we generalize our Indexed Algebra to support dynamic updates with link/cut trees.

3.3.1 Simple Tree Indexes

For the path queries that we want to support efficiently, simple path traversals are only suitable for basic optimization. Consider, for example, again predicate push down. Placing predicates that use a single IU requires only traversing a single path through the algebra. However, if the predicate uses multiple IUs, i.e., it is actually a join condition, finding the appropriate place in the tree is more challenging. For this case, we need to determine the operator where all paths from IU sources converge, i.e., we need to find the *lowest common ancestor* (LCA) of the involved IUs.

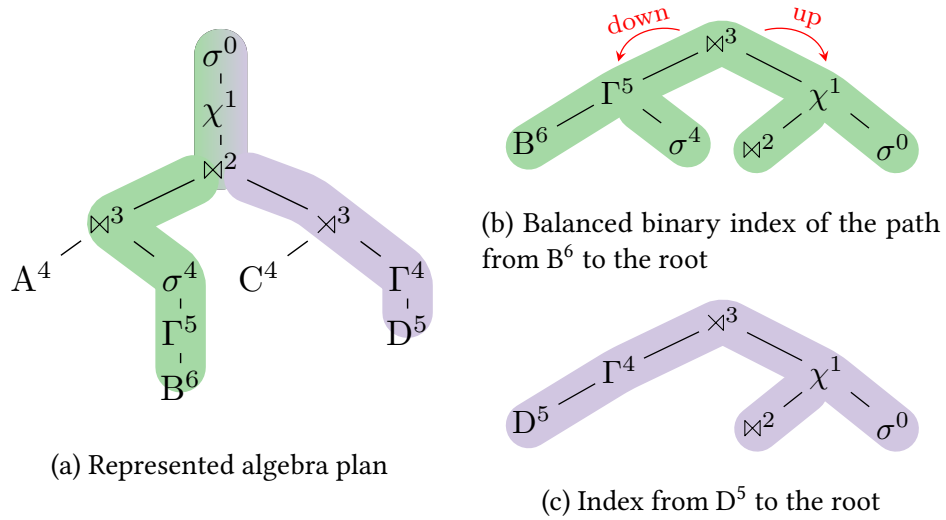


Figure 3.4: A binary search tree keyed on the distance to the root (annotated in superscript) allows efficient path queries on static algebra trees.

To answer such queries efficiently, Indexed Algebra builds balanced binary search trees for paths through the relational algebra. These auxiliary search trees are keyed by the distance to the root of the algebra tree. The balance significantly shortens the $O(n)$ path from a base relation to the root. Figure 3.4 shows an example of such an algebra tree with the distance to the root annotated as superscript. Consider the $O(n)$ path from σ^0 to B^6 , marked on the left side of Figure 3.4a. Figure 3.4b shows the $O(\log n)$ balanced index that allows to take a shortcut to traverse the path.

As an example analysis, assume that the predicate σ^0 references a column of B and a column of D. To place this predicate, the query optimizer now needs to answer the path-query to find the LCA of B and D. We can find this join by traversing the search trees of both B and D upwards until we reach a common operator. In our example in Figure 3.4b and 3.4c, we traverse to parent nodes until we reach the root node of the auxiliary tree, then follow right \uparrow child pointers until we find the subtree of common ancestors rooted at χ^1 . Since this subtree contains multiple nodes, but we want to find the lowest common ancestor, we still need to find the lowest operator in this subtree. By following the left \downarrow child pointers of χ^1 , we find \bowtie^2 as the LCA of A and D.

With the help of these indexes, we can now reason efficiently without explicitly traversing the algebra plan, and also answer more complex path queries. Since we only traverse balanced auxiliary plans, the complexity of these improves from linear to logarithmic, and we can find the LCA in $O(\log n)$ [63].

Table 3.1: Complexity of operations on relational algebra.

Rel. Algebra	Transformation	Traversal
w/o index	$\mathcal{O}(1)$	$\mathcal{O}(n)$
static index	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
path labeling	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Indexed Algebra	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Still, building these indexes is potentially expensive. Since we build an index from each leaf operator to the root, we again end up with quadratic complexity.

3.3.2 Path Labeling

Another well known technique to index hierarchical data are *labeling schemes*, e.g., pre/post encoding [60] or OrdPath [117]. In OrdPath, we label each node with its path from the root, with a special labeling algorithm that allows insertions without the need to relabel other nodes. Using these path labels, queries are efficient, i.e., we can answer most queries in $O(1)$, and LCA queries in $O(\log n)$.

However, the OrdPath scheme was originally developed for XML queries, which need to preserve document order. For relational algebra, we can relax this and label nodes not with ordinals, but with pointers, which makes accessing nodes referenced in the path more efficient. The fundamental downside of this approach is that relocating a whole subtree requires a relabeling of the whole subtree. As an example, when we push down a predicate from the root of the tree to a table scan, we need to update the OrdPath labels of all operators on this path to remove the predicate. Unfortunately, this means that the complexity for algebra transformations is $O(n)$.

3.4 Implementing Indexed Algebra

Static algebra indexes have the problem of efficient updates of the indexed algebra. While they allow efficient traversal over the algebra for queries, transforming and building the tree is now significantly more costly than in the raw algebra without an index. We now improve the transformation by using the amortization technique of link/cut trees proposed by Sleator and Tarjan [142, 143]. Table 3.1 summarizes the complexity of these different approaches to reason about relational algebra. While not indexing allows efficient transformations, traversing the algebra is expensive. Static indexes improve the traversal,

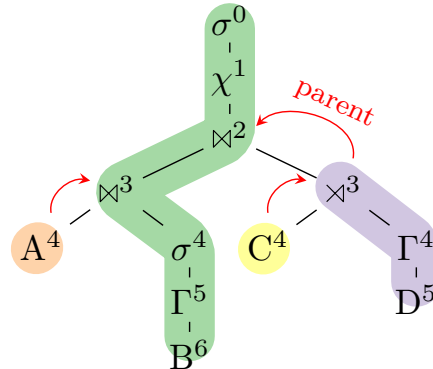



Figure 3.5: Partial path indexes of Figure 3.4. Link/cut trees build dynamically balanced splay trees over paths through the algebra and connect subtrees via path-parent pointers.

but make updates costly. With Indexed Algebra, we achieve both logarithmic updates and traversal.

On a high level, link/cut trees do not maintain balanced indexes for all paths from leaves to the algebra root, but build them dynamically when needed. They do this by maintaining splay trees for (partial) paths, which are connected by path-parent pointers. We show an example in Figure 3.5, where only the path from B^6 has a complete splay index to the root. When we want to operate on another path, e.g., from D^5 , we first need to transform the path-parent  edge to a balanced edge.

Heavy path decomposition [142] improves the cost to build the indexes. A key observation of this technique is the inherent redundancy within these path indexes: In the example in Figure 3.4 this manifests as the duplicate ancestor subtree at χ^1 , which appears in both path indexes from B^6 and D^5 . Heavy path decomposition can eliminate this redundancy by only indexing this subtree once in the largest index, decomposing the path in one heavy and one light path. The second, light, path index can now reuse this part of the path, essentially by just pointing to the already constructed subtree in the first index.

Heavy path decomposition of the paths eliminates the redundancy in their indexes. We only build balanced search indexes over heavy edges, and only store additional to the index of the parent heavy path that is connected by a light edge. Now the traversal from a leaf to the root of the algebra uses potentially multiple connected indexes. However, since we now only index each algebra operator exactly once, we also bound the complexity to build the indexes. The

most expensive part of constructing this tree is now keeping the paths sorted, effectively making the construction $O(n \log(n))$.

For now, we only indexed static algebra. Considering the goal of optimizing the algebra, which means transforming it to a more efficient form, updating the index structure is also important. For example, when we eliminate an operator, e.g., by merging a filter σ with a subsequent join \bowtie , we need to re-balance the index. Since such a modification changes the lengths of paths through the tree, we might need to split the previously longest path, and merge with another path. To amortize this potentially expensive operation, we will only keep the path indexes approximately balanced, which amortizes the $O(n)$ merging of paths. With this approximately balanced index, our query optimization process, where we often add, remove, and reorder individual operators, will also be efficient.

3.4.1 Link/Cut Trees

Implementing a link/cut tree is relatively little effort, e.g., a public implementation of Sleator fits in about 100 lines of code [141]. In this implementation, efficient *link* operations connect two trees, and *cut* operations split a subtree from its parent. During all operations, the link/cut tree keeps the accessed path roughly balanced. The operation that enables the simple implementation of the other function is *expose*. Exposing a node brings its path to the root in a form that is suitable for queries, and keeps the nodes balanced enough for amortized logarithmic behavior by organizing them in a self-balancing splay tree [143].

Link/cut trees use a heavy path decomposition that manages the path to the root of the subtree in a roughly balanced splay tree. In addition, they relax the condition that the heavy path is also the longest path in the tree. To differentiate, we call these paths *preferred* paths instead. Since we do not need to ensure the longest paths anymore, we can implement the link and cut operations efficiently by connecting the balanced with the unbalanced light edges, which we now call *path-parent* edges.

To amortize the maintenance of enough balance for logarithmic traversal, link/cut trees introduce the concept of exposing a node. The expose operation is the foundation of all other operations, which always first expose a node before executing their own logic. Exposing transforms the path from the current node to the root to a preferred path. This transform all its path-parent edges to balanced edges in the search tree, guaranteeing logarithmic time for all other operations.

To maintain the balance in the search tree, we implement it as a splay tree. In expose, we, thus, *splay* the node to the root, which maintains the approximate balance. The link and cut operations also expose the involved nodes, but then only connect or disconnect the subtrees. In contrast to the static approach,

Figure 3.6: Intrusive structure of link/cut indexed operators.

```

struct Operator {
    // Path-parent and children of the link/cut index
    Operator *lcDown, *lcUp, *lcParent;
    // Parent operator in relation algebra
    Operator *parent;
    // Algebra children are in subclasses
}

```

linking and cutting allows efficient transformations, but only roughly balance the involved search trees. However, since we also expose (and thus splay) nodes during link and cut, the balance is eventually maintained, effectively amortizing the amortizes the tree-mergers over a series of operations. Since expose is a logarithmic time operation, algebra transformations also take logarithmic time.

As described by Sleator and Tarjan [143], the amortized maintenance of balanced paths in link/cut trees pairs nicely with the amortized maintenance of balance in splay trees. In addition, splay trees move frequently accessed nodes near the root. For our application of reasoning over relation algebra, it harmonizes with the locality of reference.

For the integration of the link/cut index onto the operators, we employ the intrusive pointer structure shown in Figure 3.6. As described in Section 3.2, our algebra representation maintains an upwards reference to each operator's parent in the regular relational algebra semantics. The `lcDown` and `lcUp` pointers connect to form the auxiliary, roughly balanced splay tree that allows efficient path traversal. To support the dynamic properties, the `lcParent` links a disjoint splay-subtree to its path-parent tree with a non-preferred edge. These edges will be transformed by the first expose of that node on the first root-to-subtree path access.

3.4.2 Efficient Operations using the Link/Cut Tree

Operations on the link/cut tree work similar as with static index trees, except for the additional *expose* operations. The expose operation builds the basis of all other operations on the link/cut tree by bringing the operator to the root of the splay tree through rotations and ensuring its path to the algebra root is preferred (i.e., connected via `lcDown`/`lcUp`, and not via `lcParent` path-parent pointers). In the following, we describe the core path traversal operations that we use for algebra optimization.

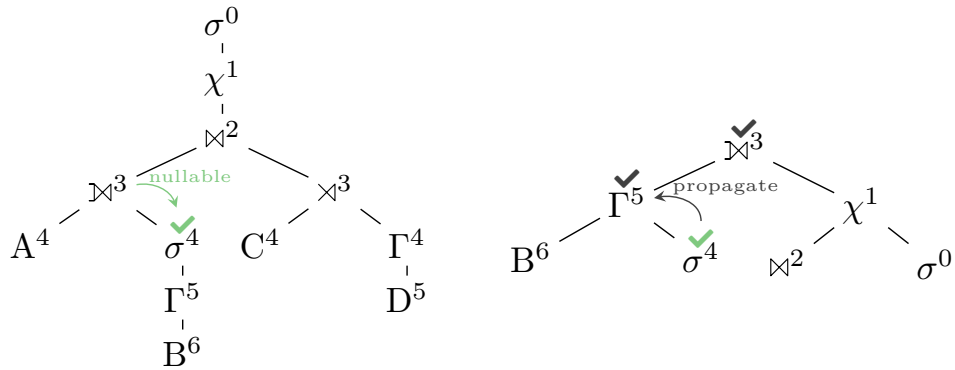
Find Root: Finding the root of an algebra tree is useful to detect if two operators are in the same tree, or if they are separated, e.g., by a common table expression. Additionally, we also use it during join ordering to detect if subtrees are already connected via a transitive join edge. Finding the root requires a path index to the root, i.e., exposing that node. With an exposed operator, its path index is fully connected to the root and roughly balanced. Finding the root of the algebra tree now means following upwards index pointers. Since these are organized via a balanced index, this operation is efficient and reaching the root takes $O(\log n)$ steps.

Lowest Common Ancestor: Finding the lowest common ancestor $lca(A, B)$ in the link cut tree differs from static indexes. We first expose A , which connects its path to the root. This path now necessarily also contains the LCA. Then, we also expose B , now connecting B 's path to the root. The lowest intersection of these paths then mark the LCA.

Path Aggregates: Finally, one advanced technique allows to efficiently answer queries about the paths between operators in a tree [80]. The idea here is to maintain the answers for calculations over a path in the index as *path aggregates*, where we calculate the aggregate from left and right index tree aggregates. This adds a constant maintenance overhead to the balancing during expose, but amortizes path queries over expose operations, which allows answering queries about a path of length n in $\mathcal{O}(\log n)$ time.

One example of a useful path aggregate is determining if a path between two operators contains an outer join \bowtie . Since outer joins make IUs nullable, it depends on if the data flow to a consumer crosses an outer join, if the column is actually nullable or not. Figure 3.7a shows the already familiar indexing example, where the left join \bowtie^3 now has left outer semantics. This outer join \bowtie^3 now marks its right, potentially nullable, child σ^4 , indicating that IUs passing both, the child and the join, can be null due to a missing join partner. In the balanced path index in Figure 3.7b, we propagate this marker, marking a node when either of its children has a marker. Transitively, the marker at the root of the path index then indicates an outer join somewhere on the path from B^6 to σ^0 . With a slight tweak, i.e., storing pointers instead of markers, we can also quickly determine the causing outer join. Similarly, we can also determine the lowest or highest outer join when we preferably propagate the outer join pointer from either the upwards or downwards direction.

With these three base operations, we can implement a plethora of useful optimizations. Since these operations take only logarithmic amortized time, these optimizations are also very effective. In the following, we describe how we apply these operations for query optimization, and how this makes the optimizations effective.



(a) Outer joins mark their children as nullable.

(b) Path indexes propagate nullable markers.

Figure 3.7: Algebra indexes efficiently determine if a path contains an outer join by propagating a marker through the balanced auxiliary tree.

3.5 Applications in Query Optimization

Indexed algebra not only enables efficient operations on relational algebra, but also allows elegant formulations of many query optimization techniques. Since our indexes ensure amortized $O(\log n)$ operations, we can formulate many optimizations that consider each operator individually without risking quadratic runtime. In the following, we discuss some query optimization problems, and how Indexed Algebra helps to efficiently apply them. This ties together the connections between operators and expressions of Section 3.2 with the indexing approach of Section 3.3. We start with simple, yet effective optimization techniques using path traversals and LCA queries, before we demonstrate the versatility of path aggregates.

3.5.1 Determining Join Graph Edges

Join ordering algorithms to find the optimal execution order of joins usually operate on a join graph [116]. To construct this graph, we collect all subtrees (e.g., nested operators like group-by Γ , or base relations) that are connected via joins as graph nodes. In addition, we also collect all join conditions from join \bowtie nodes or selections σ . To determine the edges of this join graph, we need match to match the consumed IUs in the conditions to nodes in the graph. The difficulty here is that IUs in the join condition might have their source arbitrary deep in a nested operator of the graph's leaf nodes.

To avoid building explicit column sets for each node of the join graph, we use the efficient `findRoot` operation of our Indexed Algebra. Algorithm 5 formulates

the base construction of this join graph. Note, that the following algorithms rely on the connections between expressions, IUs, and their source and consuming operators, as introduced in Section 3.2.2.

Algorithm 5: Determining join graph edges with Indexed Algebra.

```

1 nodes := The joined subtrees
2 for R ∈ nodes do
   | // Split up the join edges between subtrees
3   | cut(R)
   | // Collect the join edges
4 edges = ∅
5 for (IUref(a), IUref(b)) ∈ conditions do
6   | edges += (findRoot(a.source), findRoot(b.source))
7 ...
   | // After join ordering, rebuild tree with link()

```

In this algorithm, we first cut the joined subtrees from the overall algebra tree. This effectively makes each leaf node of the join graph a separate tree with a distinct root node. When we now consider each condition, we identify each consumed IU of that expression. From these IUs, we can determine its source operator, which needs to be in one of the subtrees that make up the nodes of our join graph. The `findRoot()` operation now finds precisely these nodes. Subsequently, for each pair of referenced IUs within the conditions, we can add a join edge between the nodes we find this way.

During this join graph construction, the Indexed Algebra is implicitly maintained by `link`, `cut`, and `findRoot` operations. This allows us to efficiently construct the join graph, even for complex input subtrees.

3.5.2 Detecting Dependent Joins

As already discussed in Section 3.1, detecting and eliminating dependent joins is one of the most important query optimization steps. Our algorithm to efficiently construct the join graph also relied on the absence of dependent join attributes. However, recognizing dependent joins is not trivial and needs some data flow analysis.

To recognize dependent IU references, we need to detect the situation where a consumed IU's source is not in the input relations of the operator containing that reference (e.g., the `A.z` in Figure 3.1). A possible implementation to detect these IUs would be to temporarily cut each operator consuming an IU and determine, if the root of the subtree containing the IU source is the consuming

operator. This gives us a binary check if that reference is dependent, but we still do not know, which join to transform to eliminate the dependency.

Algorithm 6: Recognizing dependent joins.

```

1 for IUref(a) ∈ query do
2   | lca ← findLCA(IUref.operator, a.source)
3   | if lca ≠ IUref.operator then
4   |   | mark lca as dependent join

```

To simultaneously recognize the dependent join, we implement our dependency detection slightly differently. Algorithm 6 shows our implementation using Indexed Algebra and LCA operations. We again check all IU references, but now find the LCA of the IU's source operator and the reference's containing operator. For regular references, the reference is in an ancestor operator of the source, and thus equal to the LCA. Otherwise, the reference is dependent. Here, not only detects the dependent reference, but also directly determine the dependent join.

To efficiently process the query and make subsequent optimizations simpler, we eliminate all correlations as one of the first optimizations. To unnest this query, we use the transformation rules presented in earlier work [114], eliminating the need for expensive nested loop joins. In this optimization, Indexed Algebra again helps to skip large unrelated parts of the query that are in between IU source and its references.

3.5.3 Tracking IU Nullability

For query optimizers, *null* columns are especially unpleasant to deal with [111]. In our system, we therefore try to eliminate null values already in base table scans, where we can filter many tuples all at once [87]. When we can prove that an IU is not null, subsequent operations (e.g., a join condition) can efficiently compare values and ignore null bits. However, outer joins complicate this optimization significantly, since they can conditionally produce null values.

Figure 3.8 shows an example of a query with an outer join \bowtie between base relations A and B, where B can become null after the join. Below the join, we have a predicate σ that filters on a condition of B.x, which allows us to filter null values already at the scan of B. Subsequently, we can skip the null check for the predicate and any subsequent reference of that IU, and also optimize expressions based on the assumption that the IU is not null. Unfortunately, a naïve application would incorrectly transform the upper coalesce expression.

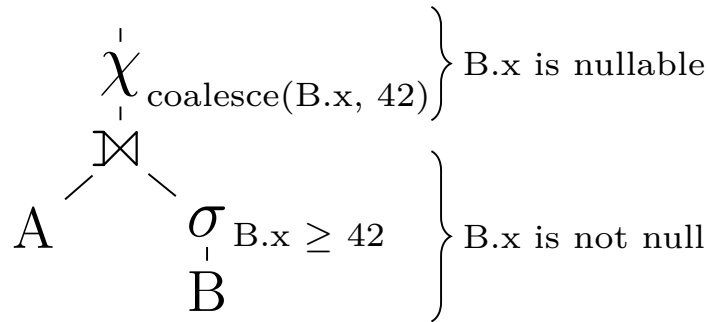


Figure 3.8: The nullability of IUs can depend on outer joins and their position in the algebra.

Instead, we need to analyze the query further and prove the absence of outer joins.

Algorithm 7: Determine if the path between B and χ contains an outer join that makes B's columns nullable.

```

// Temporarily cut to limit the path  $B \rightarrow^* \chi$ 
1 cut( $\chi$ )
// Calculate the aggregate for B
2 expose(B)
3 result  $\leftarrow$  B.nullable marker
// Restore the full path index
4 link( $\chi$ ,  $\chi$ .parent)

```

To analyze this situation, we use Indexed Algebra's efficient path aggregates (cf. Figure 3.7), as shown in Algorithm 7. One catch here is, that aggregates by default propagate through the whole preferred-path. This means that aggregates for non-root algebra nodes do not directly contain the analysis result we are interested in, i.e., if the path from B to χ contains an outer join. Instead, we temporarily *cut* the χ subtree from the overall algebra tree, so that the operator of the reference becomes a temporary root node. Afterwards, we *expose* the source operator B to propagate our aggregate through that path and get the answer. Since all three operations, cut, expose, and linking the temporary cut again, are $O(\log n)$, these operations are surprisingly efficient.

3.5.4 Predicate Pushdown

Predicate pushdown is a ubiquitous optimization that needs data flow analysis. In a simple form, we push a predicate σ on the path towards the source of

its referenced IUs. In a more complex optimization, we also want to capture transitive predicates that are conjunctive. So, when we encounter a join, we want to infer new predicates, e.g., $\sigma_{x=42}(\bowtie_{x<y}) \implies \sigma_{42<y}$.

A problem with the operator-by-operator detection of transitive edges is that views or common table expressions can also introduce predicates in arbitrary deep subtrees. To infer predicates from all join edges, even above the initial predicate, the optimizer would need to first bubble predicates up, then push them down again. Transitively, this can transform join edges ($x < y$) to predicates that can be pushed further down ($42 < y$). To infer all transitive edges, we need to iteratively bubble up and push down all predicates until we find no new transitive predicates. Since transitive chains can be arbitrary long, traversing tree operator-by-operator for each step is inefficient. Indexed Algebra instead uses two distinct techniques to propagate transitive predicates:

- Predicate push down without considering transitivity
- Upward constant propagation generating new predicates

For the push down, we use the path-centric logic from Figure 3.3 that skips intermediary operators. With Indexed Algebra, we can avoid directly traversing the algebra by finding the LCA of the referenced IUs in the predicate. Certain operators, e.g., outer joins \bowtie , require more logic, so we need a way to recognize these in the path. To detect these operators, we, again, use path aggregates that signal the presence of these operators on paths through the algebra. With these aggregates, we can effectively skip unrelated operators and push any predicate in $O(\log n)$ to the next interesting operator. With these pushed down predicates, we can now infer additional constants that we can propagate upwards.

3.5.5 Propagating Constants

Our predicate push down ensured that IUs are filtered as early as possible, ideally at base relations. After evaluating this predicate, we know that it holds in any downstream operator, which allows propagating it transitively. For some predicates like $<$, we can infer additional bounds for other comparisons, but for equality comparisons, we directly replace the IU references with constants.

For example, consider the SQL query shown in Figure 3.9. This query shows a factored subquery with the nested equality predicate $\sigma_{\text{year}=2022}$. Constant propagation would now replace all references of the year, i.e., the produced IU of both the orders subquery and the complete query, with the constant value. Then, we fold the resulting expression, which results in a simple predicate that we can push down to the deliveries table. Note that this is not limited to just constants, but this can be generalized to arbitrary predicates. For our example, the inferred

Figure 3.9: SQL query with nested constants. We propagate and fold constants from inner queries to all uses to enable transitive pushdown.

```
with years_orders as
  (select * from orders where o_year = 2022)
select * from deliveries d, years_orders o
where d.order_id = o.id and d.d_year <= o.o_year + 1
```

bounds of `o_year` can be propagated again using the same technique, and we can deduce that any other comparisons must also maintain similar bounds.

A core assumption we have for constant propagation is that IUs have the same value during execution. As in the last sections, outer joins \bowtie are the exception to this rule. When there is an outer join on the path between IU and its reference, we cannot directly propagate a constant to that reference. Constant propagation uses the familiar path aggregates that track tuple nullability from Section 3.5.3 to detect this case and only propagate predicates when no outer join can introduce nulls.

In combination with predicate pushdown, the constant propagation transitively introduces predicates through the whole query. Again, Indexed Algebra ensures that the path traversals are cheap, and we can efficiently reach all relevant IUs.

3.5.6 Bounding Distinct Values Estimates

So far, we mainly discussed query optimization rules that are almost always beneficial. For cost based optimizations, e.g., join ordering, it is essential to have good cardinality estimations [44, 91]. By making cardinality estimations path sensitive with Indexed Algebra, we can improve the estimation bounds.

Cardinality estimation often uses distinct value counts, e.g., to estimate the size of aggregations that eliminate duplicates from keys. Usually, base table estimates derive these counts by calculating statistics, e.g., in the form of HyperLogLog sketches [50]. However, when we leave base tables and process predicates or joins, changes in cardinality do not translate easily to changes in distinct values. Due to these limitations, many systems just use base table estimates, which gives suboptimal estimates compared to a more precise tracking of distinct values through the algebra.

Figure 3.10 shows a simplified query on the TPC-H schema, where we can get better estimates by considering the path from source to root. Consider the marked path from `nation` to the group by Γ where we first have a filtering

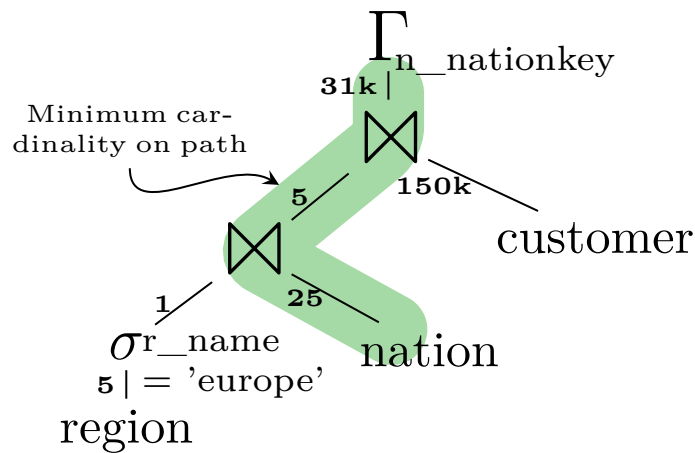


Figure 3.10: The cardinality of the top aggregation Γ depends on the distinct values reaching it from the base table. Indexed Algebra maintains a path aggregate to efficiently maintain this information.

join with region, then a growing join with customers. If we just consider the distinct values at the base table (25), we lose the information about the filtering intermediate join. Inspecting the whole path between the operators to find the minimum path cardinality (5) captures a more precise upper bound for the distinct values.

To calculate this minimum path cardinality efficiently, we again use path aggregates. The path aggregates maintain the minimum cardinality of the path to the root by selecting the minimum aggregate from $lcUp$ and $lcDown$. To estimate the minimum path cardinality to an arbitrary operator, we *cut* its subtree and *expose* the source operator. In our example, we cut the parent of the aggregate Γ to make it the root, and expose the nation table scan to ensure its path to the root is preferred. Then, we get the min cardinality of 5, which gives us a precise estimate without the need to traverse the operator tree.

3.5.7 Placing Expression Evaluation

We also optimize the amount of data that we store in intermediary operators that need to materialize. These *pipeline breakers* are costly, since they break the data centric execution and allocate memory to store tuples that are read later. The most common pipeline breaker is the build side of a hash join that stores tuples in a hash table. Reducing the size of the tuples stored in this materialized state is advantageous, since it also increases cache locality.

For this optimization, we evaluate expressions at materialization points when this leads to a smaller materialized state. In this state, we need to store any

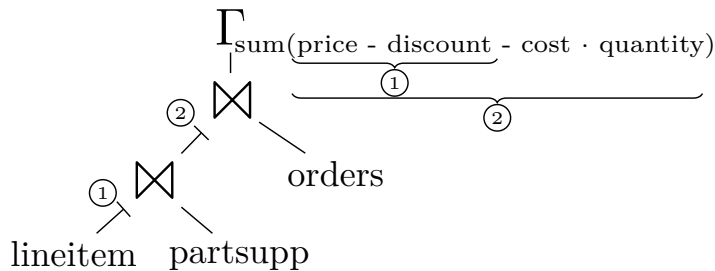


Figure 3.11: Materialization points allow evaluating expressions that reduce the size of the materialized data. We use Indexed Algebra’s LCA and path operations to efficiently find suitable materializations points.

produced IUs of the input that might be consumed by subsequent expressions. By evaluating expressions before this materialization, we can reduce the state, which reduces the memory consumption, and the overall cost of the query execution. While evaluating expression might incur some overhead, a smaller materialized state is usually more beneficial.

As an example, consider the algebra tree shown in Figure 3.11 that approximately follows the TPC-H schema. We explicitly annotate the pipeline breaking build side of the joins with \top , which gives us two locations to evaluate expressions. We can evaluate the first subexpression $\text{price} - \text{discount}$ at the lower build side ①, which allows us to materialize the single result value instead of the two partial values. At the second hash table build ②, we have all required IUs to evaluate the whole expression. Thus, we can avoid materializing its four referenced IUs, and instead only store the single result value.

To reorder the evaluation of an expression, we need to consider its constraints in the algebra tree. We can only evaluate it when all its referenced IUs are available, which we can find by computing the LCA of the producing operators. Then, we can place it anywhere between the LCA position and its actual usage, at a suitable pipeline breaker. To also efficiently find pipeline breakers, we implement an aggregated property of Indexed Algebra, similar to the markers of Section 3.5.4. This way, we can efficiently find a suitable evaluation place for each subexpression of a complex expression that reduces the overall materialized state.

In summary, we presented several optimizations that benefit from Indexed Algebra. By using path-centric optimization techniques, we obtain concise and efficient algorithms.

3.6 Beyond Indexed Algebra

Indexed Algebra allows to efficiently answer data flow questions that arise during query optimization. In the following, we discuss further implementation techniques that are less related to data flow.

3.6.1 Complex Expressions

Expressions in the relational algebra are often a significant challenge, and many systems optimize them, e.g., by compiling them to efficient machine code [121]. In practice, expressions can be very large, which makes optimizing and compiling them non-trivial. For example, we observed queries with a predicate of several thousand disjunctions. A naïve approach to such queries can result in large optimization states that exceed machine limits.

One problem with such large expressions is that recursive algorithms analyzing it reach stack-size limitations [112]. As with relational operators, we organize expressions in a tree structure that unfortunately is not balanced. This means that its depth can scale linearly with the size of the input expression. However, the stack space for recursive calls is limited and usually not very large.

We implement a technique to avoid the stack limits by recognizing large stack depths before an actual overflow. To recognize this situation, we can inspect the stack pointer, i.e., `rbp` on `x86`, and abort the query. This way, we avoid crashing the DBMS, but we degrade its usefulness, since now users need to work around the system's limitations. When we exceed the stack size in Umbra, we instead switch to a different stack, which allows processing such queries, albeit with some overhead during optimization. Note that since Umbra compiles queries, this is a one-time instead of a per-tuple overhead.

We also address the underlying issue of the large, deeply nested structure of expressions. One problem we identify is the representation of such predicates as *binary* boolean expressions. Instead, we increase the fan-out of our boolean expressions by representing them as *Nary* expressions, which, e.g., results in a single disjunctive expression with arbitrary many boolean inputs. In practice, this means that we get shallow expression trees, where optimizing scales nicely even for large input predicates. Note that this does not avoid any deeply nested expression, but we found that this significantly improves the common case.

Having such large expressions in mind, we also eagerly fold constants. Folding constants early not only reduces the size of the expression trees, but also reduces the load of subsequent code generation. In certain cases, compilers like LLVM use super-linear algorithms to compile code [110], which becomes painful when dealing with large generated expressions.

3.6.2 Lazy Property Evaluation

For individual operators, some properties are especially expensive to evaluate. For these properties, we only want to calculate them once we need them, and when they are unlikely to change again. In our implementation, we identified the most expensive properties:

- The estimated cardinality
- The functional dependencies

For cardinality estimation, loading data structures for statistics and evaluating predicates on samples of a base table is relatively expensive. Ideally, we only want to estimate base table cardinalities once, when we pushed down all predicates that we can evaluate on that table. If we push down additional predicates after this cardinality estimation, we would need to invalidate previous estimations, which would cause duplicate work. Thus, we should lazily estimate cardinalities on demand after predicate push down.

However, we potentially access the cardinality multiple times during join ordering, where we treat filtered base tables as leaf nodes of the join graph. To estimate the cardinality only once, we additionally cache it for each operator. This unfortunately has the downside that we need cache invalidation logic when we alter operators in a way that affects the cardinality.

Similarly, calculating and maintaining functional dependencies between IUs can enable query optimizations, such as minimizing group-by keys. However, computing functional dependencies and equivalent IUs is relatively expensive, when only a handful of optimizations actually require functional dependencies. This lazy evaluation is mainly useful with complex expressions. For example in TPC-H, we see an overall improvement of 6%. However, the impact is larger for queries with complex predicates, e.g., TPC-H Q19, where we get a 23% improvement by avoiding unnecessary work.

3.6.3 DAG Structured Algebra

So far, we only looked at tree structured algebra plans, where each operator has a single parent. However, relational algebra can also have multiple outputs in the form of *common table expressions* (CTEs, i.e., WITH clauses or views in SQL). Additionally, we also want to be able to share intermediate results for push-based execution. Therefore, we introduce non-tree edges, which makes our algebra DAG structured in the general case.

For regular queries, we expect DAG edges to be significantly fewer than the tree edges of our regular algebra. To minimize the DAG edges, we inline shared

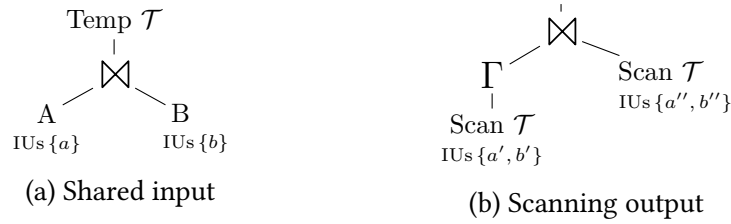


Figure 3.12: A DAG structured query plan. Operators can be referenced multiple times, but need to rename their IUs to avoid ambiguity.

table scans to avoid unnecessary intermediate materialization, which makes them part of the algebra tree. CTEs can also be read multiple times, but a simple inlining transformation can similarly transform these DAG edges. Inlining is not always advisable, but helps to transform DAG to tree edges if a CTE is only read once.

To support DAG edges, it first seems that analyzing data flows crossing DAG edges complicates the reasoning over dependencies. When an IU crosses such a DAG edge, its data now takes two paths through the execution plan. However, to avoid ambiguity in the subsequent query, we also need to have distinct names for these IUs. Renaming is also a pragmatic solution for the data flow analysis. Within an algebra tree, we can use Indexed Algebra for efficient reasoning, but when the data flow crosses a DAG edge, we also switch the IU we reason about.

In our implementation, we use a special operator to cross DAG edges that share an input node. This parent operator of a shared node creates new IUs that are distinct from the input IUs. We call this operator a *PipelineBreakerScan* (PBS) that maintains the shared input in a referenced-counted state and explicitly maps from IUs in its input to IUs in its output. The PBS also generalizes over its input, which can be an arbitrary operator that can be scanned multiple times, which are usually pipeline breakers.

Figure 3.12 shows an example query plan that contains DAG edges. The scanned input in Figure 3.12a joins two base relations A and B, and materializes them in a Temp operator. Crossing the DAG edges, Figure 3.12b has two PBSs that rename all inputs to two sets of distinct IUs. The scanned output still references the IUs a and b , but we take special care to consider them part of the scanned Temp operator and not part of the disconnected output. This way, data flow questions reasoning about a are contained in one tree, while a' refers to the same data across the DAG edge.

As a result, we generalize our algebra to DAG structured queries with the simple renaming abstraction. The resulting overall algebra now has DAG edges between algebra trees. Within these trees, we use Indexed Algebra that allows

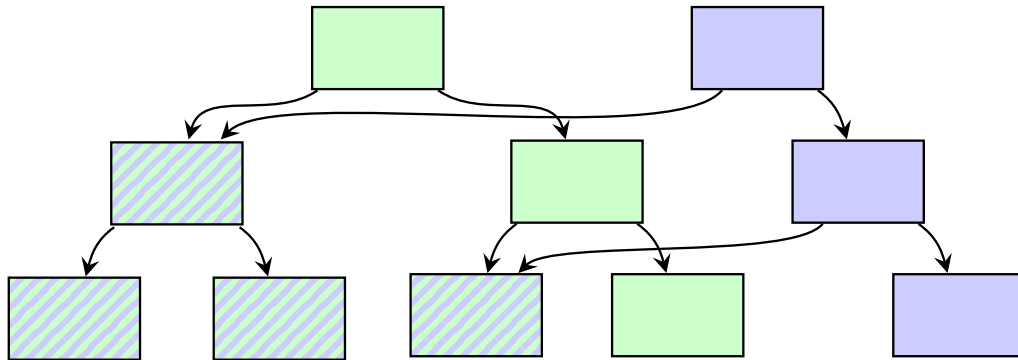


Figure 3.13: Two copy-on-write column sets share the striped nodes and only duplicate a logarithmic number of nodes when inserting new IUs.

efficient reasoning. Thus, we can lift all techniques discussed before from trees to DAGs.

3.7 Efficiently Representing Column Sets

In the last sections, we proposed restructuring the relational algebra and rewriting the query optimization algorithms to achieve an asymptotic speedup of the optimizer. However, this is relatively invasive and requires many changes to the implementation of the database system, which hinders the adoption in other systems. An alternative to this relational algebra restructuring would be making the sets of columns, i.e., *IU sets*, more efficient, and avoid the $O(n^2)$ behavior. The maintenance of sets of available IUs is a common operation during the semantic analysis of a query plan, however, for most operators the set of available IUs is similar to the input to that operator, and contain duplicate IUs. Intuitively, operators like $\text{map } \chi$ pass-through their available columns, but they still need to maintain a distinct set to add their additional computed columns. As discussed in Section 3.1, this leads to a worst-case quadratic amount of IUs in all sets, even though we only have a linear amount of total distinct columns. In the following, we discuss an efficient representation of column sets that avoids unnecessarily duplicating the input, while still providing efficient set operations.

The fundamental problem of these IU sets is the duplication of the input state. This duplication is necessary, since we want to cache the IU sets for reach operator to avoid expensive recomputation when we reason about the IUs multiple times. Theoretically, computing the new set of columns only requires to apply the column delta of an operator, but since we need to preserve the input set, a naïve implementation of IU sets needs to make a copy of the input. A solution to this dichotomy, efficient delta updates and preserving input, are

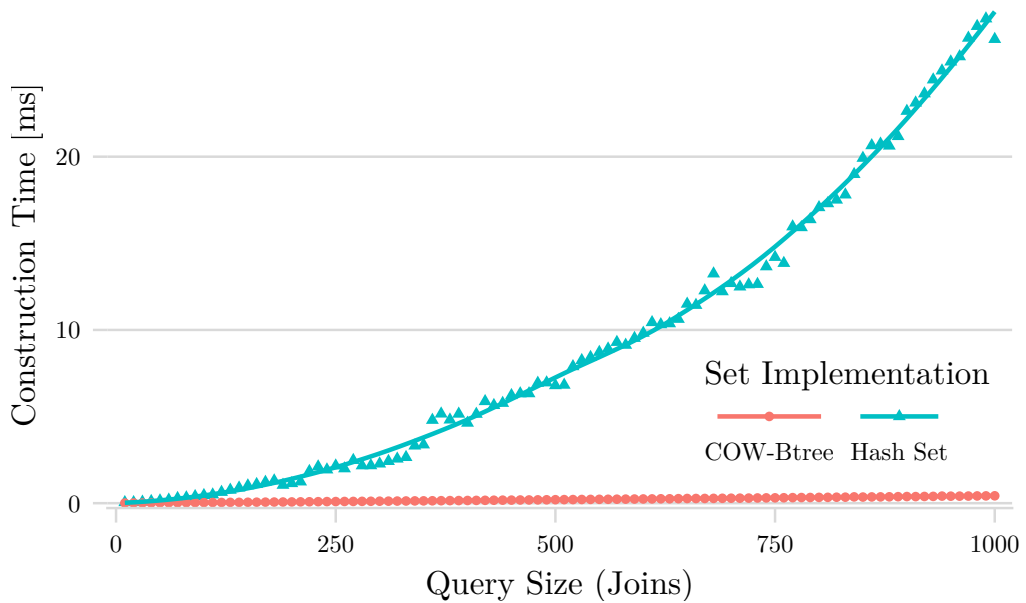


Figure 3.14: Microbenchmark of column set operations on a left-deep join tree with an increasing number of joins.

persistent data structures [35] that keep older states unmodified when updating the set with a delta. This allows cheaply caching the input IU set, while still being able to insert the delta efficiently.

Traditional persistent data structures are not unproblematic for performance sensitive code, since they are usually pointer heavy, which results in high per-entry memory overhead and bad cache locality. In this section, we implement copy-on-write B-Trees, which reuse B-Tree pages when they are unchanged. Additionally, the shared B-Tree nodes stores a reference count that indicates if a node is shared between sets, or if we can avoid to copy when updating the node and can modify it in place. Figure 3.13 shows a simplified example of two sets that differ in the rightmost IUs, but still share all left children.

When inserting IUs at the right of these nodes, we can update these exclusively used pages in-place without copying nodes. Only when a page is shared between sets, we need to copy nodes on the path to the modification point and decrease the reference count. We therefore want to preferably insert into non-shared edges to get minimal copy overhead. Our implementation achieves this by ordering IUs consecutively by creation, so that IUs of the same operator share the same node, and all IUs of an operator are inserted at the same node.

Potential Performance Benefit. The sharing of parts of the IU sets can significantly decrease the number of duplicate IUs in sets, which in turn improves performance and reduces memory use. We show the effect of this optimization

in a micro benchmark in Figure 3.14. In this experiment, we compare the performance of calculating all available columns after a join, which can, e.g., be used to find the push-down direction for predicates. For this benchmark, we perform the union of two input sets with our copy-on-write (COW) B-Trees and compare this with a state-of-the-art hash-based set implementation, i.e., `tsl::hopscotch_set` [64]. In this experiment, we construct a left-deep join tree under an increasing number of relations with four columns each. This resembles the process used to construct unoptimized relational algebra from parsed SQL.

In this microbenchmark, we can observe quadratic construction time, which already starts to show in moderately sized queries. With 100 joins, our copy-on-write implementation is about $14\times$ faster than the hash set that needs to copy its inputs, which grows to a $63\times$ speedup and for 1000 joins. The tables in this test case are even relatively narrow, which is often only the case after pruning unused columns. For comparison, TPC-H has on average 8 and TPC-DS 17 columns per table.

A possible optimization opportunity is to only include referenced columns in the column sets, without the need for explicit projections. This, of course, only works when we already computed which columns are in-use and which are unnecessary, for which we already need column sets to begin with. However, it is quite common that some columns are accessed infrequently, especially for wide tables with hundreds of columns. After the initial semantic analysis pass, where it is unnecessary to cache column sets, we can detect completely unused columns that are never referenced with a pointer structure similar to Section 3.2.2. This reduces the size of column sets significantly, but can only avoid the quadratic behavior in well-behaved queries, and has no effect, e.g., for a simple `SELECT *`. Thus, while pruning appears to help with column set size, its worst-case behavior is still quadratic. Indexed Algebra, in contrast, improves the performance for the general case.

Discussion. Copy on write works well when we only have operators that pass through their inputs. However, new IUs, i.e., calculated results with maps χ or markers in mark joins, limit the effectiveness of the shared nodes. When an algebra transformation changes the produced IUs of an operator, some part of the IU set containing the produced IUs of that operator need to be recalculated. How big this recalculated part is depends on the position of the affected IUs in the B-Tree nodes and their fill degree. Since our implementation orders IUs by their creation, any operator that filters or adds IUs of a whole table thus can operate on a contiguous range of IUs in this set. Additionally, reordering operators, e.g., for join order optimization, does not affect the order of the IUs in the set.

However, for some operations might still invalidate a large number of IU sets, which makes the asymptotic behavior still worse than using Indexed Algebra to index the paths through the algebra. Consider constant propagation, where we replace an IU with a constant and erase it from all IU sets. The dependent sets are potentially all sets on the path to the algebra root, i.e., $\mathcal{O}(n)$ sets. Naïvely, constructing each set requires $\mathcal{O}(n \log(n))$ operations, thus transforming the algebra has $\mathcal{O}(n) \cdot \mathcal{O}(n \log(n)) = \mathcal{O}(n^2 \log(n))$ complexity. However, when we assume that we can share $\mathcal{O}(n)$ IUs in each set and this brings each set's construction cost down to $\mathcal{O}(\log(n))$, the complexity for each transformation is still $\mathcal{O}(n \log(n))$, which is significantly worse than our approach based on link/cut path indexes that supports transformations in $\mathcal{O}(\log(n))$.

Finally, column sets do not help for optimizations that need to traverse a path through the algebra tree, e.g., the tracking of IU nullability due to outer joins (Section 3.5.3) or determining the minimum cardinality of a path (Section 3.5.6). In these cases, we still traverse the algebra. Thus, maintaining IU sets for algebra optimization does not improve the asymptotic optimization complexity. A more efficient representation of column helps the case where we need to build these sets once, e.g., to calculate the materialized IUs in a hash join. However, updating these sets in a dynamically changing algebra tree is the most expensive way to reason about algebra that we looked at. In the following, we use a standard hash set implementation to give a more honest evaluation compared to the status-quo.

3.8 Evaluation

We now evaluate the impact of our work on the performance of the query optimization engine in our research RDBMS Umbra. We compare our implementation of Indexed Algebra to our implementation using path traversal, and additionally evaluate an approach using column sets. We start with an evaluation that shows the asymptotic improvements for large queries, before we show the impact on popular benchmarks.

Our query optimizer produces state-of-the-art query plans that do not differ between any of the presented analysis approaches. To show the quality of our produced plans, we provide an online interactive query plan viewer for the evaluated benchmarks¹.

Setup of Performance Measurements: We run all measurements on a system with an Intel Xeon W-2145 CPU with 8 cores, 2× hyper-threads, and 32 GB RAM. Since we measure the relatively small amount of optimization time in the benchmarks, we ensure consistent results by repeating every measurement

¹<https://umbra-db.com/interface/>

1000 times and reporting the average time. Compared to query execution, the small optimization time seem negligible, but as in most systems, query planning in Umbra is single threaded and every millisecond in query planning blocks potentially hundreds of cores for parallel query execution.

As benchmarks, we use TPC-H [20], TPC-DS [107], and the Join Order Benchmark (JOB) [91]. The complexity of the queries varies significantly between the benchmarks, with TPC-H having the least complex queries. In our implementation, the TPC-H queries involve on average 9 and a maximum of 20 operators, where JOB has an average of 18 and a maximum of 36, while TPC-DS is the most complex with an average of 20 and a maximum of 71 operators.

Reports from industry, however, feature orders of magnitude more complex queries. SAP [33, 98] for example reports that their core data services contain over 100 views that reference more than 100 tables, with the largest view referencing over 4000 tables. Similarly, we hear reports from Tableau [150] and VMware [144] about auto generated queries that are dozens of pages of SQL.

To test the asymptotic optimization runtime for such large queries, we cannot compare queries with a fixed amount of operators. Instead, we use a synthetic workload, where we gradually increase the involved operators. In the following, we use a synthetic join workload that is inspired by a SQLite's `sqllogictest`². In this workload, we gradually increase the number of involved base relations, which allows simulating large algebra trees.

In addition, query optimization has many parts that are unaffected by the size of the algebra. Table 3.2 shows a break-down of the average optimization times over Umbra's optimization passes in TPC-DS. For most passes, Indexed Algebra has no significant benefit, since these passes do not reason about the algebra itself, but use mostly operator local information, e.g., for constant folding during expression simplification. For these passes, Indexed Algebra adds some overhead to maintain the indexes, which only become relevant for the algebra centric optimizations. For these optimizations, unnesting and predicate pushdown, Indexed Algebra has the biggest impact. In the following, we concentrate on unnesting, since that is the optimization that sees the biggest improvement of using an index, and, since we need to check each IU reference if it is correlated, also scales with the size of the algebra tree.

3.8.1 Efficiency on Query Complexity

In a first experiment, we evaluate the impact of query complexity on optimization runtime. However, the implementation of the traditional operator-centric optimizations using column sets differs significantly from our proposed path-centric

²<https://www.sqlite.org/sqllogictest/>

Table 3.2: Impact of Indexed Algebra on TPC-DS optimization.

Optimization Pass	Avg. Time [μ s]		
	Column Sets	Indexed Algebra	Speedup
Simplify Expressions	10.5	11.0	0.95
Unnesting	78.8	10.9	7.26
Predicate Pushdown	66.9	58.8	1.18
Cardinality Estimation	96.9	97.4	1.00
Join Ordering	63.5	65.7	0.97
Physical Planning	20.7	23.1	0.91
Total	339.9	269.5	1.28

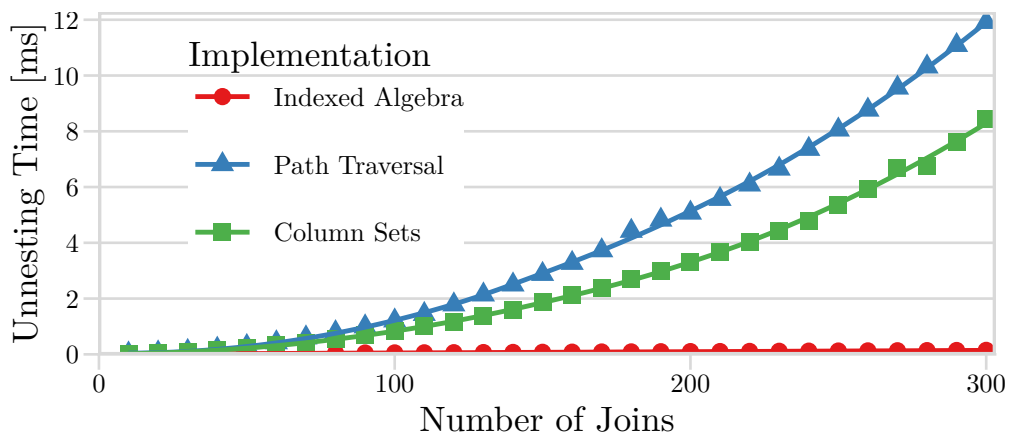


Figure 3.15: Query optimization time of synthetic join queries with many relations. Indexed Algebra has asymptotically better runtime for large queries.

optimization (cf. Figure 3.3). For this evaluation, we implement all optimizations with the path-centric optimizations, and only calculate column sets once for the unnesting logic of Section 3.5.2. For path-centric optimization, we compare a naïve path traversal with Indexed Algebra.

For this experiment, we expect Indexed Algebra to have an advantage growing with the query complexity. Path traversal and column sets both have quadratic behavior, while Indexed Algebra runs in $O(n \log n)$. Between column sets and path traversal, we expect no dramatic difference.

Figure 3.15 shows the results measuring the time to optimize queries of increasing complexity. As a first micro benchmark, we only consider the time to execute the optimization to decorrelate any nested expressions. On the x-axis, the figure shows the increasing number of joins between 10 and 300 relations. As expected, the two traditional implementation approaches, path traversal and

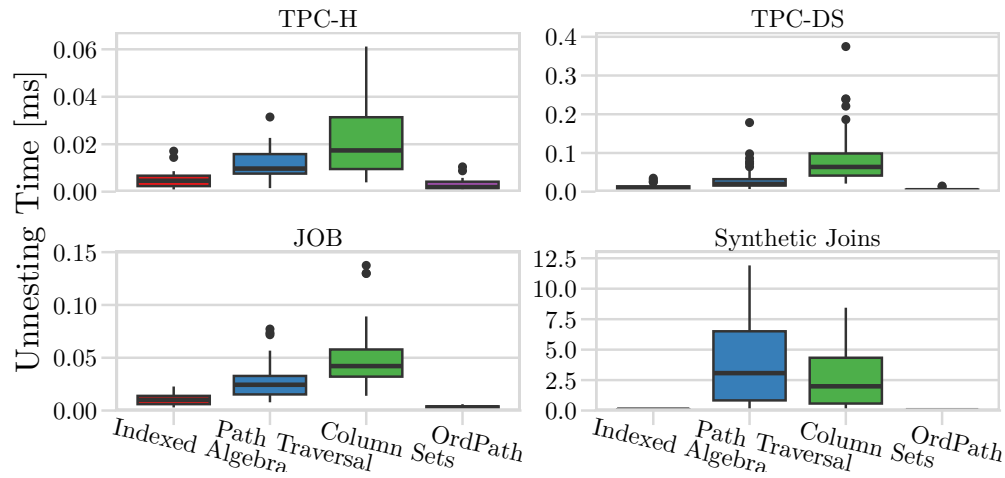


Figure 3.16: Query optimization time of various benchmarks.

column sets, scale badly with them taking several milliseconds to perform the single optimization pass.

With increasing query complexity, the two traditional approaches show clear super linear execution time. For ten relations, path traversal takes 20 μ s, while Indexed algebra is 4 \times faster and only takes 5 μ s. With 300 relations, path traversal takes over 12 ms, where Indexed Algebra only takes 0.14 ms, over 85 \times faster. This means that using Indexing algebra, we can cope with very large queries. We tested even larger join sizes with 1k and 10k joins, where Umbra takes 0.16 and 13 seconds to optimize the query. For such large joins, join reordering becomes a bottleneck, where, e.g., join linearization shows quadratic runtime [116].

This shows the advantage of Indexed Algebra over the traditional approaches for complex queries. While this is not as pronounced for smaller queries, it is still a significant advantage there.

3.8.2 Benchmarks

While we saw a definitive improvement for synthetic joins, as a workload, it is rather simplistic. The synthetic workload only has base relations and join operators, where the benchmark queries of TPC-H, TPC-DS and JOB better capture the real world applications that also contain business logic in aggregates and more complex expressions. For this experiment, we continue to measure unnesting time, which applies to all operators and expressions since the SQL standard allows correlated attributes at almost any point of a query.

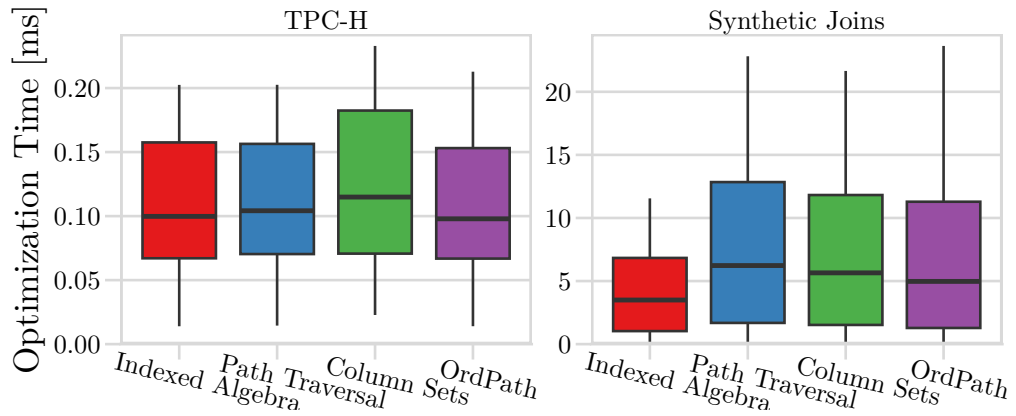


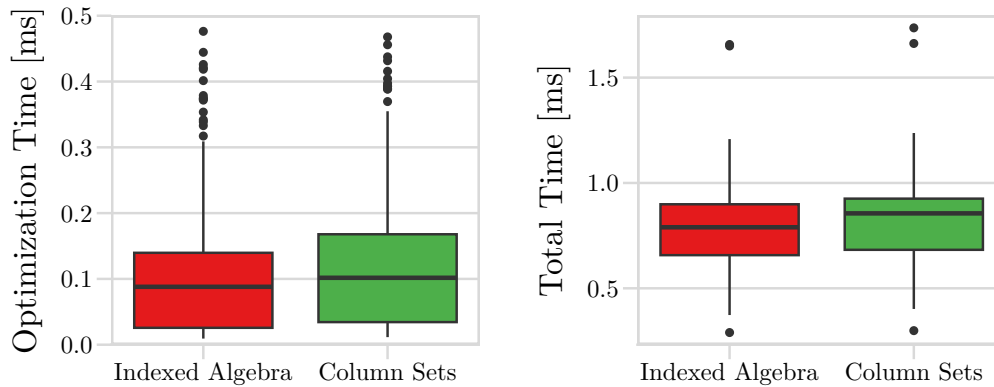
Figure 3.17: Comparison of total optimization times.

In this experiment, we expect fewer gains than with the complex synthetic queries. Since the queries in this experiment contain significantly fewer operators on average, the quadratic behavior is not as dramatic. However, as we already saw in the last benchmark, Indexed algebra should still be several times faster.

Figure 3.16 shows a box plot of the results, which roughly follow our expectations. In TPC-H and JOB, Indexed Algebra is more than $4\times$ faster than using column sets, while it is on average $7\times$ faster in TPC-DS. In addition, Indexed Algebra significantly improves the situation for the outliers: Unnesting TPC-DS Q64 takes over $350\ \mu\text{s}$ with column sets, while Indexed Algebra only takes about $23\ \mu\text{s}$. Similarly, TPC-H Q8 takes over $80\ \mu\text{s}$ with column sets, where Indexed Algebra takes $8.5\ \mu\text{s}$.

In comparison to the synthetic queries, the performance of path traversal and column sets is unexpected. On average, path traversal is faster for this workload of real queries. We suspect that this might be caused by memory allocation: The relations in real-world queries have many more columns than in the synthetic workload, which results in large dynamically allocated sets. Since these sets also scale quadratic with the query size, the ballooning memory results in poor cache locality, which path traversal avoids.

In contrast, OrdPath seems to be the best implementation, when just considering unnesting. However, unnesting mostly reads the query tree and favors optimization strategies that allow cheap path queries. When we also consider the total optimization time that includes transformations, this picture changes: Figure 3.17 shows the total optimization time, including query transformations. For the small queries in TPC-H, OrdPath is competitive with Indexed Alge-



(a) Optimization time for over 1000 queries from Tableau Public workbooks.

(b) Total end-to-end time for a comparably sized TPC-H workload.

Figure 3.18: Evaluation of interactive workloads.

bra, but for larger synthetic join queries the asymptotically worse updates to OrdPath’s path labels become costly.

To summarize, Index Algebra not only has sizable improvements for huge queries, but also significantly improves optimization of relatively small real world queries. In the following, we investigate how well the improvements of this specific task translate to the complete query optimization process.

3.8.3 Interactive Workloads

Benchmarks capture a narrow use-case with mostly static queries. Query optimization is more challenging for workbooks, which are popular tools for complex, interactive data analytics [149]. The data these workbooks run on is typically relatively small, but the queries can be quite complex. For the following evaluation, we use real-world queries from Tableau Public, which we convert to standard SQL and CSV files³.

For our evaluation of interactive workloads in Figure 3.18a, we consider over 1000 queries from nine complex Tableau Public workbooks. The data queried in these workbooks is relatively small, and consequently, the median execution time for these queries is relatively short with 230 μ s. In contrast, the median optimization time of these queries with Umbra’s traditional column set implementation 105 μ s. Using Indexed Algebra brings the median total optimization time down to 89 μ s, which is an 18% speedup on real world queries.

As an additional data point on small data, we use TPC-H on a small scale factor 0.01. The small scale factor captures the optimization challenges of the

³<https://github.com/tum-db/tableaupublic>

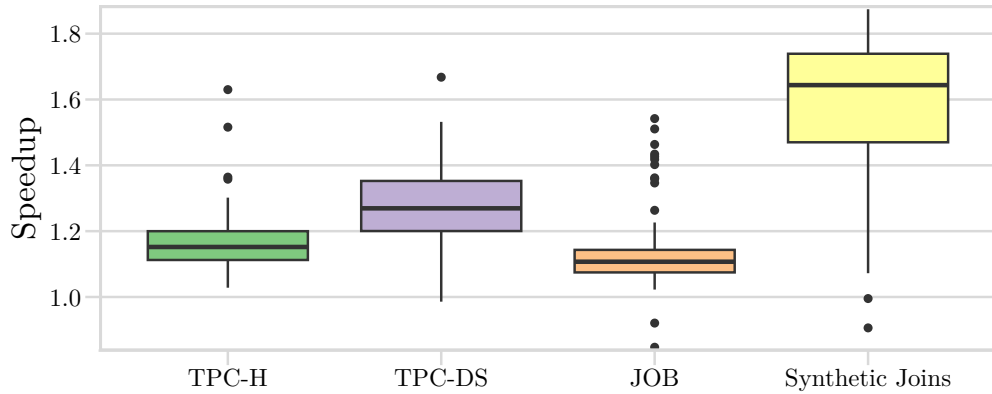


Figure 3.19: Improvements of total query optimization time. We compare the time to optimize queries using Indexed Algebra in contrast to column sets. The average improvements are: 12% for TPC-H, 29% for TPC-DS, and 10% for JOB.

interactive nature of such workbooks, and still captures the data size of the 75th percentile of Tableau Public workbooks [150]. In this configuration, the TPC-H queries have a median execution time of 369 μ s. The median total processing time for column sets is 856 μ s, using Indexed Algebra reduces this to 790 μ s. Figure 3.18b shows this as a box plot. In total, Indexed Algebra can reduce the end-to-end latency of this workload by 8%.

3.8.4 Overall Results

Query optimization comprises many optimizations, where the unnesting is only one partial optimization. Many other analyses in other optimization passes can be similarly costly, but are not as dependent on the query structure as unnesting, which diminishes the improvement of Indexed Algebra. As discussed in Section 3.6.2, cardinality estimation using sample evaluation is expensive, and unaffected by Indexed Algebra. To quantify the overall improvement, we measure the speedup of using Indexed Algebra over the total query optimization time.

Over all optimizations, we expect less speedup than we saw for query unnesting in the last sections. Still, many optimizations besides unnesting also depend on the query structure, so we should still measurable a significant improvement with Indexed Algebra. Especially for complex queries, where the quadratic scaling of traditional methods has the most impact, we expect good improvements.

Figure 3.19 shows the aggregated speedup of query optimization time over the measured benchmarks. As expected, the speedup is less than for the specific

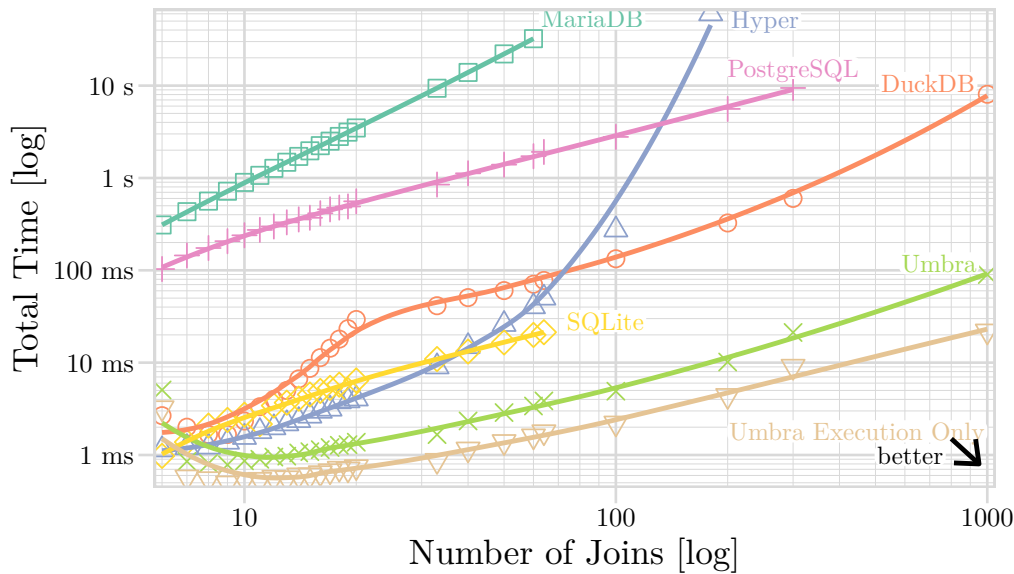


Figure 3.20: Total time spent processing synthetic join queries with many relations. Umbra uses Indexed Algebra to efficiently optimize large queries.

optimization of the last measurements, but we still see some significant speedups for outliers, e.g., TPC-H Q8 and TPC-DS Q64, which improve by over 50%.

We conclude with a systems comparison of the total processing time for synthetic join queries in Figure 3.20. This experiment shows MariaDB 10.9.4, DuckDB 0.6.1, PostgreSQL 14.6, Hyper 0.0.16377, and SQLite 3.40.1 over an average of three runs for join sizes of up to 1000 joins. As workload, we adapted the synthetic joins that we already used in the last experiments, but with base tables and a result of 1000 tuples so that the query execution is not trivial. We measure all systems with an increasing number of joins, until we hit the limits of the systems, e.g., a parse error for SQLite, or excessive processing time. While the excessive runtime could have multiple causes, inefficient planning or execution, Umbra still achieves sub-second processing times even for more than a thousand joins using Indexed Algebra.

3.9 Related Work

After the classical approach in System R [134], Goetz Graefe pioneered the implementation of optimizations on relational algebra with the EXODUS [54], Volcano [53], and Cascades [52] systems. Modern optimizers like Calcite [12] or Orca [144] still use the same concepts. These systems all rely on operator centric optimizations that transform the plan with predefined rules. Indexed

Algebra works on path-centric algorithms instead, which allows more efficient plan transformations.

A newer development is the development of advanced query compilers. Query compilers nowadays build upon data centric code generation [108, 136], which translates query plans into an intermediate language that a compiler like LLVM can optimize and transform to machine code. Subsequent work in this area advanced the used intermediate representations (IR) to fit the needs of query processing systems [69, 75, 137, 148]. Indexed Algebra optimizes the logical plan from a high level, where IRs focus on the lowering to machine code for the physical query plan. This also allows powerful inter-operator optimizations such as operator fusion. However, a series of operators forming a pipeline become a function or loop that is a boundary where imperative compilers cannot easily optimize. In contrast, Indexed Algebra specializes for data flow questions inherent to query languages, where we can introduce optimizations across pipelines. In summary, we see these approaches as complementary: IRs allow powerful low-level optimizations on individual expressions, while Indexed Algebra optimizes the high-level plan.

Another related work is TreeToaster [9], which builds efficient pattern matching based on the incremental view maintenance engine DBToaster [1]. TreeToaster recognizes that pattern matching is a bottleneck, and makes pattern matching on dynamic algebra trees efficient. In contrast, our work reengineers the pattern matching to operate on paths instead of individual operators.

3.10 Conclusion

In this chapter we introduced Indexed Algebra as an efficient solution to optimize relational algebra. Traditional techniques to query optimization did not scale for complex queries, often showing quadratic runtime with increasing operators. While complex queries previously took a long time to optimize, our technique helps to reduce the average optimization time and makes processing the most complex queries viable.

Indexed Algebra tames the quadratic complexity of query optimization by building an index structure of the data flow paths through the query. In combination with our proposed path-centric query optimization, this reduces the time spent optimizing. With Indexed Algebra, the runtime of both queries and transformations of the algebra is logarithmic in the number of operators. In total, this query optimization can implement optimizations looking at all operators in $O(n \log n)$.

Furthermore, we have shown that path-centric query optimization not only allows efficient, but also expressive implementation of query optimization. For

path-centric optimization, especially the least common ancestor operation is a convenient way to directly find interesting points where the data flow intersects. This approach also does not require additional data structures that we would need to maintain and update, which significantly reduces the implementation effort. In effect, this allows Indexed Algebra to offer the best of both worlds, efficiency and ease of use.

Even for moderately complex queries as we find in TPC-H, Indexed Algebra improves the total optimization time by up to 1.8×. Due to the asymptotically better runtime, complex queries show an even larger improvement. For such complex queries, Indexed Algebra allows to efficiently optimize queries that previously were considered impossible to optimize.

CHAPTER 4

Conclusions

In this thesis, we observed several challenges for low latency query processing and query execution, and developed strategies to improve modern database systems. First, we introduced a strategy for parallel execution with optimistically shared state that avoids contention between threads. In addition, we propose an improved cardinality estimation algorithm for calculated columns based on statistical distributions to guide the query optimizer. Lastly, we enable low latency query optimization using an index over paths through the relational algebra to accelerate common query optimization tasks.

These techniques enable new business use-cases that require low-latency query processing. For example, many users have a need for hybrid OLTP and OLAP [67, 119] processing. Our contributions bring traditionally long-running analytical queries closer to the latency needed for transaction processing [70]. In turn, this also leads to a lower time-to-insight, i.e., when one can move from batch processing analytics overnight to running them on real-time data. Lastly, an improved latency also is beneficial for interactive visualizations [85], which need low latency to not feel sluggish and allow smooth updates of the visualization based on query results when changing the analysis. Low-latency query execution also sparks interest from industry, where, anecdotally, most analytical queries process only a few hundred megabytes of data. With such workloads, the line between the traditionally distinct worlds of OLTP and OLAP systems are blurry, since modern database systems can execute OLAP queries with OLTP latency.

Outlook. While this thesis improves low latency data processing for modern database systems, we still see further opportunities for improvements, which were partially explored in related publications that the author of this thesis was

part of, but are not part of this thesis. These still have the potential to further improve execution latency.

One path to more capable low latency processing is to make I/O more efficient by enabling cheap I/O parallelism that hides I/O access latency [101]. While modern I/O devices, either storage devices in the form of SSDs or network interfaces, offer very high throughput that is comparable to main memory, but still have relatively high latency for single reads [62]. With multiple outstanding read requests, it is possible to hide the latency and overlap processing of already available data with the latency time for the I/O. This is especially relevant when we need many I/Os for processing, e.g., for index joins [124]. However, traditional methods to achieve such I/O parallelism, e.g., by oversubscribing the CPU with thousands of execution threads, create massive CPU overhead, which then limits the useful processing capacity and leads to higher query latency [28]. One way to make this more efficient is to use coroutines, and an asynchronous I/O interface like `io_uring`. Some challenges for this approach remain, e.g., an efficient approach for userspace scheduling of coroutines, or to make locking coroutine aware and non-blocking [22].

Other approaches to reduce query latency include *adaptive query processing* [8, 126], which can dynamically switch query execution plans to adapt to hardware or predicate misestimations. Keeping the latency for the plan adaption low is challenging for code-generating execution engines, which need low-latency query re-compilation [131]. To enable this efficiently, we can reuse already compiled-code to speed up recompilation, e.g., when selectivity during execution is different from query planning, or to quickly generate different query plans in prepared statements with parameters. Using compiler techniques, we can generate code that can be cheaply reordered or optionally executed without needing to recompile the whole query, but with a much cheaper code copying approach. As a result, this approach further reduces the latency of many queries.

Finally, we see a need for efficient text processing in database systems [129]. Many data processing operations are already efficient, e.g., hash based data structures with small integer payloads. However, many real-world workloads contain text, which is substantially slower to process, and the typical column-based vectorization approaches that speed up regular queries are not applicable to strings. However, using code generation, one can specialize the processing code of strings when an operation applies to many tuples, e.g., pattern matching within these strings. As a result, the tailored code for string processing can make better use of hardware-specific features such as SIMD [139, 140, 151], which significantly improves the latency of these operations.

In conclusion, this thesis continues the ongoing research efforts to reduce data processing latency. We contribute several new and improved algorithms

that improve query planning and processing, which enables better data analytics and improves the interactivity and approachability of data science.

Bibliography

- [1] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. “DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views”. In: *Proc. VLDB Endow.* 5.10 (2012), pp. 968–979.
- [2] Gene M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *AFIPS Spring Joint Computing Conference*. Vol. 30. AFIPS Conference Proceedings. AFIPS / ACM / Thomson Book Company, Washington D.C., 1967, pp. 483–485.
- [3] Raja Appuswamy, Goetz Graefe, Renata Borovica-Gajic, and Anastasia Ailamaki. “The five-minute rule 30 years later and its impact on the storage hierarchy”. In: *Commun. ACM* 62.11 (2019), pp. 114–120.
- [4] Morton M. Astrahan et al. “System R: Relational Approach to Database Management”. In: *ACM Trans. Database Syst.* 1.2 (1976), pp. 97–137.
- [5] Adelchi Azzalini. “A class of distributions which includes the normal ones”. In: *Scandinavian journal of statistics* (1985), pp. 171–178.
- [6] Adelchi Azzalini. *The R package sn: The skew-normal and related distributions such as the skew-t and the SUN (version 2.1.1)*. [tp://azzalini.stat.unipd.it/SN/](http://azzalini.stat.unipd.it/SN/). 2023.
- [7] Adelchi Azzalini. *The skew-normal and related families*. Vol. 3. Cambridge University Press, 2013.
- [8] Shivnath Babu and Pedro Bizarro. “Adaptive Query Processing in the Looking Glass”. In: *CIDR*. www.cidrdb.org, 2005, pp. 238–249.
- [9] Darshana Balakrishnan, Carl Nuessle, Oliver Kennedy, and Lukasz Ziarek. “TreeToaster: Towards an IVM-Optimized Compiler”. In: *SIGMOD Conference*. ACM, 2021, pp. 155–167.
- [10] Maximilian Bandle, Jana Giceva, and Thomas Neumann. “To Partition, or Not to Partition, That is the Join Question in a Real System”. In: *SIGMOD*. ACM, 2021, pp. 168–180.

- [11] Rudolf Bayer and Edward M. McCreight. “Organization and Maintenance of Large Ordered Indexes”. In: *SIGFIDET Workshop*. ACM, 1970, pp. 107–141.
- [12] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. “Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources”. In: *SIGMOD*. ACM, 2018, pp. 221–230.
- [13] Alexander Beischl, Timo Kersten, Maximilian Bandle, Jana Giceva, and Thomas Neumann. “Profiling dataflow systems on multiple abstraction levels”. In: *EuroSys*. ACM, 2021, pp. 474–489.
- [14] Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski, Angela Amor, Mohamed Zait, and Chun Chieh Lin. “Enhanced Subquery Optimizations in Oracle”. In: *Proc. VLDB Endow. 2.2 (2009)*, pp. 1366–1377.
- [15] Altan Birler. “Scalable Reservoir Sampling on Many-Core CPUs”. In: *SIGMOD Conference*. ACM, 2019, pp. 1817–1819.
- [16] Altan Birler, Bernhard Radke, and Thomas Neumann. “Concurrent online sampling for all, for free”. In: *DaMoN*. ACM, 2020, 5:1–5:8.
- [17] Mike W. Blasgen et al. “System R: An Architectural Overview”. In: *IBM Syst. J.* 20.1 (1981), pp. 41–62.
- [18] Hans-Juergen Boehm and Sarita V. Adve. “Foundations of the C++ concurrency memory model”. In: *PLDI*. ACM, 2008, pp. 68–78.
- [19] Peter A. Boncz, Angelos-Christos G. Anadiotis, and Steffen Kläbe. “JCC-H: Adding Join Crossing Correlations with Skew to TPC-H”. In: *TPCTC*. Vol. 10661. Lecture Notes in Computer Science. Springer, 2017, pp. 103–119.
- [20] Peter A. Boncz, Thomas Neumann, and Orri Erling. “TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark”. In: *TPCTC*. Vol. 8391. Lecture Notes in Computer Science. Springer, 2013, pp. 61–76.
- [21] Peter A. Boncz, Marcin Zukowski, and Niels Nes. “MonetDB/X100: Hyper-Pipelining Query Execution”. In: *CIDR*. 2005.
- [22] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. “Scalable and robust latches for database systems”. In: *DaMoN*. ACM, 2020, 2:1–2:8.

- [23] Damianos Chatziantoniou, Michael O. Akinde, Theodore Johnson, and Samuel Kim. “The MD-join: An Operator for Complex OLAP”. In: *ICDE*. IEEE, 2001, pp. 524–533.
- [24] Surajit Chaudhuri and Kyuseok Shim. “Including Group-By in Query Optimization”. In: *VLDB*. 1994, pp. 354–366.
- [25] John Cieslewicz and Kenneth A. Ross. “Adaptive Aggregation on Chip Multiprocessors”. In: *VLDB*. ACM, 2007, pp. 339–350.
- [26] Sophie Cluet and Guido Moerkotte. “Efficient Evaluation of Aggregates on Bulk Types”. In: *DBPL*. Electronic Workshops in Computing. Springer, 1995, p. 8.
- [27] Andrew Crotty, Alex Galakatos, and Tim Kraska. “Getting Swole: Generating Access-Aware Code with Predicate Pullups”. In: *ICDE*. IEEE, 2020, pp. 1273–1284.
- [28] Andrew Crotty, Viktor Leis, and Andrew Pavlo. “Are You Sure You Want to Use MMAP in Your Database Management System?” In: *CIDR*. www.cidrdb.org, 2022.
- [29] Benoît Dageville et al. “The Snowflake Elastic Data Warehouse”. In: *SIGMOD*. ACM, 2016, pp. 215–226.
- [30] Umeshwar Dayal. “Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers”. In: *VLDB*. 1987, pp. 197–208.
- [31] Laurens De Haan and Ana Ferreira. *Extreme value theory: an introduction*. Springer Science & Business Media, 2007.
- [32] Cristian Diaconu et al. “Hekaton: SQL server’s memory-optimized OLTP engine”. In: *SIGMOD*. ACM, 2013, pp. 1243–1254.
- [33] Nicolas Dieu, Adrian Dragusanu, Françoise Fabret, François Llirbat, and Eric Simon. “1,000 Tables Under the From”. In: *Proc. VLDB Endow.* 2.2 (2009), pp. 1450–1461.
- [34] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. “Quantifying TPC-H Choke Points and Their Optimizations”. In: *Proc. VLDB Endow.* 13.8 (2020), pp. 1206–1220.
- [35] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. “Making Data Structures Persistent”. In: *J. Comput. Syst. Sci.* 38.1 (1989), pp. 86–124.
- [36] Dominik Durner, Viktor Leis, and Thomas Neumann. “JSON Tiles: Fast Analytics on Semi-Structured Data”. In: *SIGMOD Conference*. ACM, 2021, pp. 445–458.

- [37] Dominik Durner, Viktor Leis, and Thomas Neumann. “On the Impact of Memory Allocation on High-Performance Query Processing”. In: *DaMoN*. ACM, 2019, 21:1–21:3.
- [38] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. “Selectivity Estimation for Range Predicates using Lightweight Models”. In: *Proc. VLDB Endow.* 12.9 (2019), pp. 1044–1057.
- [39] Marius Eich. “Extending Dynamic-Programming-Based Plan Generators: Beyond Pure Enumeration”. PhD thesis. University of Mannheim, Germany, 2017.
- [40] Marius Eich, Pit Fender, and Guido Moerkotte. “Efficient generation of query plans containing group-by, join, and groupjoin”. In: *VLDB J.* 27.5 (2018), pp. 617–641.
- [41] Amr El-Helw, Ihab F. Ilyas, and Calisto Zuzarte. “StatAdvisor: Recommending Statistical Views”. In: *Proc. VLDB Endow.* 2.2 (2009), pp. 1306–1317.
- [42] Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, and Milind Joshi. “Execution strategies for SQL subqueries”. In: *SIGMOD*. ACM, 2007, pp. 993–1004.
- [43] Martin Eppert, Philipp Fent, and Thomas Neumann. “A Tailored Regression for Learned Indexes: Logarithmic Error Regression”. In: *aiDM@SIGMOD*. ACM, 2021, pp. 9–15.
- [44] Philipp Fent, Altan Birler, and Thomas Neumann. “Practical Planning and Execution of Groupjoin and Nested Aggregates”. In: *VLDB J.* 32 (2023), pp. 1165–1190.
- [45] Philipp Fent, Michael Jungmair, Andreas Kipf, and Thomas Neumann. “START - Self-Tuning Adaptive Radix Tree”. In: *ICDE Workshops*. IEEE, 2020, pp. 147–153.
- [46] Philipp Fent, Guido Moerkotte, and Thomas Neumann. “Asymptotically Better Query Optimization Using Indexed Algebra”. In: *Proc. VLDB Endow.* 16.11 (2023), pp. 3018–3030.
- [47] Philipp Fent and Thomas Neumann. “A Practical Approach to Groupjoin and Nested Aggregates”. In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2383–2396.
- [48] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. “Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory”. In: *ICDE*. IEEE, 2020, pp. 1477–1488.

- [49] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. “Adopting Worst-Case Optimal Joins in Relational Database Systems”. In: *Proc. VLDB Endow.* 13.11 (2020), pp. 1891–1904.
- [50] Michael J. Freitag and Thomas Neumann. “Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates”. In: *CIDR*. 2019.
- [51] César A. Galindo-Legaria and Milind Joshi. “Orthogonal Optimization of Subqueries and Aggregation”. In: *SIGMOD*. ACM, 2001, pp. 571–581.
- [52] Goetz Graefe. “The Cascades Framework for Query Optimization”. In: *IEEE Data Eng. Bull.* 18.3 (1995), pp. 19–29.
- [53] Goetz Graefe. “Volcano - An Extensible and Parallel Query Evaluation System”. In: *IEEE Trans. Knowl. Data Eng.* 6.1 (1994), pp. 120–135.
- [54] Goetz Graefe and David J. DeWitt. “The EXODUS Optimizer Generator”. In: *SIGMOD*. ACM Press, 1987, pp. 160–172.
- [55] Goetz Graefe and Harumi A. Kuno. “Modern B-tree techniques”. In: *ICDE*. IEEE, 2011, pp. 1370–1373.
- [56] Jim Gray and Franco Putzolu. “The 5 Minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time”. In: *SIGMOD Rec.* 16.3 (1987), pp. 395–398.
- [57] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. “Quickly Generating Billion-Record Synthetic Databases”. In: *SIGMOD*. 1994.
- [58] Paul W. P. J. Grefen and Rolf A. de By. “A Multi-Set Extended Relational Algebra - A Formal Approach to a Practical Issue”. In: *ICDE*. IEEE, 1994, pp. 80–88.
- [59] Ferdinand Gruber, Maximilian Bandle, Alexis Engelke, Thomas Neumann, and Jana Giceva. “Bringing Compiling Databases to RISC Architectures”. In: *Proc. VLDB Endow.* 16.6 (2023), pp. 1222–1234.
- [60] Torsten Grust. “Accelerating XPath location steps”. In: *SIGMOD Conference*. ACM, 2002, pp. 109–120.
- [61] Ashish Gupta, Venky Harinarayan, and Dallan Quass. “Aggregate-Query Processing in Data Warehousing Environments”. In: *VLDB*. 1995, pp. 358–369.
- [62] Gabriel Haas, Michael Haubenschild, and Viktor Leis. “Exploiting Directly-Attached NVMe Arrays in DBMS”. In: *CIDR*. www.cidrdb.org, 2020.

- [63] Dov Harel and Robert Endre Tarjan. “Fast Algorithms for Finding Nearest Common Ancestors”. In: *SIAM J. Comput.* 13.2 (1984), pp. 338–355.
- [64] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. “Hopscotch Hashing”. In: *DISC*. Vol. 5218. Lecture Notes in Computer Science. Springer, 2008, pp. 350–364.
- [65] Denis Hirn and Torsten Grust. “PgCuckoo: Laying Plan Eggs in PostgreSQL’s Nest”. In: *SIGMOD*. ACM, 2019, pp. 1929–1932.
- [66] Jürgen Hölsch, Michael Grossniklaus, and Marc H. Scholl. “Optimization of Nested Queries using the NF2 Algebra”. In: *SIGMOD*. ACM, 2016, pp. 1765–1780.
- [67] Dongxu Huang et al. “TiDB: A Raft-based HTAP Database”. In: *Proc. VLDB Endow.* 13.12 (2020), pp. 3072–3084.
- [68] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. “MonetDB: Two Decades of Research in Column-oriented Database Architectures”. In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 40–45.
- [69] Michael Jungmair, André Kohn, and Jana Giceva. “Designing an Open Framework for Query Optimization and Compilation”. In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2389–2401.
- [70] Robert Kallman et al. “H-store: a high-performance, distributed main memory transaction processing system”. In: *Proc. VLDB Endow.* 1.2 (2008), pp. 1496–1499.
- [71] Alfons Kemper, Donald Kossmann, and Christian Wiesner. “Generalised Hash Teams for Join and Group-by”. In: *VLDB*. 1999, pp. 30–41.
- [72] Alfons Kemper and Thomas Neumann. “HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots”. In: *ICDE*. IEEE Computer Society, 2011, pp. 195–206.
- [73] Martin Kersten, Panagiotis Koutsourakis, Niels Nes, and Ying Zhan. “Bridging the Chasm between Science and Reality”. In: *CIDR*. 2021.
- [74] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. “Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask”. In: *Proc. VLDB Endow.* 11.13 (2018), pp. 2209–2222.
- [75] Timo Kersten, Viktor Leis, and Thomas Neumann. “Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra”. In: *VLDB J.* 30.5 (2021), pp. 883–905.

- [76] Changkyu Kim et al. “Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs”. In: *Proc. VLDB Endow.* 2.2 (2009), pp. 1378–1389.
- [77] Won Kim. “On Optimizing an SQL-like Nested Query”. In: *ACM Trans. Database Syst.* 7.3 (1982), pp. 443–469.
- [78] Andreas Kipf, Michael Freitag, Dimitri Vorona, Peter Boncz, Thomas Neumann, and Alfons Kemper. “Estimating Filtered Group-By Queries is Hard: Deep Learning to the Rescue”. In: *AIDB*. 2019.
- [79] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. “Learned Cardinalities: Estimating Correlated Joins with Deep Learning”. In: *CIDR*. 2019.
- [80] Philip N. Klein and Shay Mozes. *Optimization Algorithms for Planar Graphs*. <https://planarity.org>. 2021.
- [81] André Kohn, Viktor Leis, and Thomas Neumann. “Adaptive Execution of Compiled Queries”. In: *ICDE*. IEEE Computer Society, 2018, pp. 197–208.
- [82] André Kohn, Viktor Leis, and Thomas Neumann. “Building Advanced SQL Analytics From Low-Level Plan Operators”. In: *SIGMOD Conference*. ACM, 2021, pp. 1001–1013.
- [83] André Kohn, Viktor Leis, and Thomas Neumann. “Building Advanced SQL Analytics From Low-Level Plan Operators”. In: *Proc. VLDB Endow.* 14 (2021).
- [84] André Kohn, Viktor Leis, and Thomas Neumann. “Making Compiling Query Engines Practical”. In: *IEEE Trans. Knowl. Data Eng.* 33.2 (2021), pp. 597–612.
- [85] André Kohn, Dominik Moritz, and Thomas Neumann. “DashQL - Complete Analysis Workflows with SQL”. In: *CoRR* abs/2306.03714 (2023).
- [86] Donald Kossmann and Konrad Stocker. “Iterative dynamic programming: a new class of query optimization algorithms”. In: *ACM Trans. Database Syst.* 25.1 (2000), pp. 43–82.
- [87] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. “Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation”. In: *SIGMOD*. ACM, 2016, pp. 311–326.
- [88] Per-Åke Larson. “Data Reduction by Partial Preaggregation”. In: *ICDE*. IEEE, 2002, pp. 706–715.

- [89] Viktor Leis. *The Great CPU Stagnation*.
<https://databasearchitects.blogspot.com/2023/04/the-great-cpu-stagnation.html>. 2023.
- [90] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. “Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age”. In: *SIGMOD*. ACM, 2014, pp. 743–754.
- [91] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. “How Good Are Query Optimizers, Really?” In: *Proc. VLDB Endow.* 9.3 (2015), pp. 204–215.
- [92] Viktor Leis, Alfons Kemper, and Thomas Neumann. “The adaptive radix tree: ARTful indexing for main-memory databases”. In: *ICDE*. IEEE, 2013, pp. 38–49.
- [93] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. “The ART of practical synchronization”. In: *DaMoN*. ACM, 2016, 3:1–3:8.
- [94] Viktor Leis et al. “Query optimization through the looking glass, and what we found running the Join Order Benchmark”. In: *VLDB J.* 27.5 (2018), pp. 643–668.
- [95] Feilong Liu, Ario Salmasi, Spyros Blanas, and Anastasios Sidiropoulos. “Chasing Similarity: Distribution-aware Aggregation Scheduling”. In: *Proc. VLDB Endow.* 12.3 (2018), pp. 292–306.
- [96] Zhenghua Lyu et al. “Greenplum: A Hybrid Database for Transactional and Analytical Workloads”. In: *SIGMOD*. ACM, 2021, pp. 2530–2542.
- [97] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. “Many-query join: efficient shared execution of relational joins on modern hardware”. In: *VLDB J.* 27.5 (2018), pp. 669–692.
- [98] Norman May, Alexander Böhm, and Wolfgang Lehner. “SAP HANA - The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads”. In: *BTW*. Vol. P-265. LNI. GI, 2017, pp. 545–563.
- [99] Norman May and Guido Moerkotte. “Main Memory Implementations for Binary Grouping”. In: *XSym*. Vol. 3671. Lecture Notes in Computer Science. Springer, 2005, pp. 162–176.
- [100] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. “Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last”. In: *Proc. VLDB Endow.* 11.1 (2017), pp. 1–13.

- [101] Leonard von Merzljak, Philipp Fent, Thomas Neumann, and Jana Giceva. “What Are You Waiting For? Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth!” In: *ADMS@VLDB. 2022*, pp. 36–46.
- [102] Guido Moerkotte. *Building Query Compilers*. <http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf>. 2020.
- [103] Guido Moerkotte and Thomas Neumann. “Accelerating Queries with Group-By and Join by Groupjoin”. In: *PVLDB* 4.11 (2011), pp. 843–851.
- [104] Guido Moerkotte and Thomas Neumann. “Faster Join Enumeration for Complex Queries”. In: *ICDE. IEEE*, 2008, pp. 1430–1432.
- [105] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. “Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors”. In: *Proc. VLDB Endow.* 2.1 (2009), pp. 982–993.
- [106] M. Muralikrishna. “Improved Unnesting Algorithms for Join Aggregate SQL Queries”. In: *VLDB. 1992*, pp. 91–102.
- [107] Raghunath Othayoth Nambiar and Meikel Poess. “The Making of TPC-DS”. In: *VLDB. ACM*, 2006, pp. 1049–1058.
- [108] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware”. In: *Proc. VLDB Endow.* 4.9 (2011), pp. 539–550.
- [109] Thomas Neumann. “Engineering High-Performance Database Engines”. In: *Proc. VLDB Endow.* 7.13 (2014), pp. 1734–1741.
- [110] Thomas Neumann. *Linear Time Liveness Analysis*. <https://databasearchitects.blogspot.com/2020/04/linear-time-liveness-analysis.html>. 2020.
- [111] Thomas Neumann. “Reasoning in the Presence of NULLs”. In: *ICDE. IEEE Computer Society*, 2018, pp. 1682–1683.
- [112] Thomas Neumann. *Taming Deep Recursion*. <https://databasearchitects.blogspot.com/2020/11/taming-deep-recursion.html>. 2020.
- [113] Thomas Neumann and Michael J. Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *CIDR. 2020*.
- [114] Thomas Neumann and Alfons Kemper. “Unnesting Arbitrary Queries”. In: *BTW. Vol. P-241. LNI. GI*, 2015, pp. 383–402.
- [115] Thomas Neumann, Viktor Leis, and Alfons Kemper. “The Complete Story of Joins (in HyPer)”. In: *BTW. Vol. P-265. LNI. GI*, 2017, pp. 31–50.

- [116] Thomas Neumann and Bernhard Radke. “Adaptive Optimization of Very Large Join Queries”. In: *SIGMOD*. ACM, 2018, pp. 677–692.
- [117] Patrick E. O’Neil, Elizabeth J. O’Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. “ORDPATHs: Insert-Friendly XML Node Labels”. In: *SIGMOD Conference*. ACM, 2004, pp. 903–908.
- [118] Mark O’Neill, Elham Vaziripour, Justin Wu, and Daniel Zappala. “Condensing Steam: Distilling the Diversity of Gamer Behavior”. In: *Internet Measurement Conference*. ACM, 2016, pp. 81–95.
- [119] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. “Hybrid Transactional/Analytical Processing: A Survey”. In: *SIGMOD Conference*. ACM, 2017, pp. 1771–1775.
- [120] Arthur Pewsey. “Problems of inference for Azzalini’s skewnormal distribution”. In: *Journal of applied statistics* 27.7 (2000), pp. 859–870.
- [121] Ravindra Pindikura. *Gandiva Initiative: Improving SQL Performance by 70x*. <https://www.dremio.com/gandiva-performance-improvements-production-query/>. 2018.
- [122] Orestis Polychroniou and Kenneth A. Ross. “High throughput heavy hitter aggregation for modern SIMD processors”. In: *DaMoN*. ACM, 2013, p. 6.
- [123] Magdalena Pröbstl, Philipp Fent, Maximilian E. Schüle, Moritz Sichert, Thomas Neumann, and Alfons Kemper. “One Buffer Manager to Rule Them All: Using Distributed Memory with Cache Coherence over RDMA”. In: *ADMS@VLDB*. 2021, pp. 17–26.
- [124] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. “Interleaving with Coroutines: A Practical Approach for Robust Index Joins”. In: *Proc. VLDB Endow.* 11.2 (2017), pp. 230–242.
- [125] Bernhard Radke and Thomas Neumann. “LinDP++: Generalizing Linearized DP to Crossproducts and Non-Inner Joins”. In: *BTW*. Vol. P-289. LNI. GI, 2019, pp. 57–76.
- [126] Bogdan Raducanu, Peter A. Boncz, and Marcin Zukowski. “Micro adaptivity in Vectorwise”. In: *SIGMOD Conference*. ACM, 2013, pp. 1231–1242.
- [127] Vijayshankar Raman et al. “DB2 with BLU Acceleration: So Much More than Just a Column Store”. In: *Proc. VLDB Endow.* 6.11 (2013), pp. 1080–1091.

- [128] Maximilian Reif and Thomas Neumann. “A Scalable and Generic Approach to Range Joins”. In: *Proc. VLDB Endow.* 15.11 (2022), pp. 3018–3030.
- [129] Adrian Riedl, Philipp Fent, Maximilian Bandle, and Thomas Neumann. “Exploiting Code Generation for Efficient LIKE Pattern Matching”. In: *ADMS@VLDB.* 2023.
- [130] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. “A Layered Aggregate Engine for Analytics Workloads”. In: *SIGMOD.* ACM, 2019, pp. 1642–1659.
- [131] Tobias Schmidt, Philipp Fent, and Thomas Neumann. “Efficiently Compiling Dynamic Code for Adaptive Query Processing”. In: *ADMS@VLDB.* 2022, pp. 11–22.
- [132] Stefan Schuh, Xiao Chen, and Jens Dittrich. “An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory”. In: *SIGMOD.* ACM, 2016, pp. 1961–1976.
- [133] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. “ArrayQL for Linear Algebra within Umbra”. In: *SSDBM.* ACM, 2021, pp. 193–196.
- [134] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. “Access Path Selection in a Relational Database Management System”. In: *SIGMOD.* ACM, 1979, pp. 23–34.
- [135] Praveen Seshadri et al. “Cost-Based Optimization for Magic: Algebra and Implementation”. In: *SIGMOD.* ACM Press, 1996, pp. 435–446.
- [136] Hesam Shahrokhi and Amir Shaikhha. “Building a Compiled Query Engine in Python”. In: *CC.* ACM, 2023, pp. 180–190.
- [137] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. “How to Architect a Query Compiler”. In: *SIGMOD.* ACM, 2016, pp. 1907–1922.
- [138] Moritz Sichert and Thomas Neumann. “User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases”. In: *Proc. VLDB Endow.* 15.5 (2022), pp. 1119–1131.
- [139] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. “Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures”. In: *SIGMOD Conference.* ACM, 2017, pp. 403–415.

- [140] Evangelia A. Sitaridi, Orestis Polychroniou, and Kenneth A. Ross. “SIMD-accelerated regular expression matching”. In: *DaMoN*. ACM, 2016, 8:1–8:7.
- [141] Daniel Dominic Sleator. *Submission #860934 - Codeforces*. <https://codeforces.com/contest/117/submission/860934>. 2011.
- [142] Daniel Dominic Sleator and Robert Endre Tarjan. “A Data Structure for Dynamic Trees”. In: *J. Comput. Syst. Sci.* 26.3 (1983), pp. 362–391.
- [143] Daniel Dominic Sleator and Robert Endre Tarjan. “Self-Adjusting Binary Search Trees”. In: *J. ACM* 32.3 (1985), pp. 652–686.
- [144] Mohamed A. Soliman et al. “Orca: a modular query optimizer architecture for big data”. In: *SIGMOD*. ACM, 2014, pp. 337–348.
- [145] Hennie J. Steenhagen. “Optimization of Object Query Languages”. PhD thesis. University of Twente, 1995.
- [146] Konrad Stocker, Donald Kossmann, Reinhard Braumandl, and Alfons Kemper. “Integrating Semi-Join-Reducers into State of the Art Query Processors”. In: *ICDE*. IEEE, 2001, pp. 575–584.
- [147] Michael Stonebraker and Lawrence A. Rowe. “The Design of Postgres”. In: *SIGMOD*. ACM Press, 1986, pp. 340–355.
- [148] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. “How to Architect a Query Compiler, Revisited”. In: *SIGMOD Conference*. ACM, 2018, pp. 307–322.
- [149] Adrian Vogelsgesang, Tobias Mühlbauer, Viktor Leis, Thomas Neumann, and Alfons Kemper. “Domain Query Optimization: Adapting the General-Purpose Database System Hyper for Tableau Workloads”. In: *BTW*. Vol. P-289. LNI. Gesellschaft für Informatik, Bonn, 2019, pp. 313–333.
- [150] Adrian Vogelsgesang et al. “Get Real: How Benchmarks Fail to Represent the Real World”. In: *DBTest@SIGMOD*. ACM, 2018, 1:1–1:6.
- [151] Xiang Wang et al. “Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs”. In: *NSDI*. USENIX Association, 2019, pp. 631–648.
- [152] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. “On-Demand State Separation for Cloud Data Warehousing”. In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2966–2979.
- [153] Christian Winter, Tobias Schmidt, Thomas Neumann, and Alfons Kemper. “Meet Me Halfway: Split Maintenance of Continuous Views”. In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2620–2633.

- [154] Weipeng P. Yan and Per-Åke Larson. “Eager Aggregation and Lazy Aggregation”. In: *VLDB*. 1995, pp. 345–357.
- [155] Weipeng P. Yan and Per-Åke Larson. “Performing Group-By before Join”. In: *ICDE*. IEEE, 1994, pp. 89–100.
- [156] Zuyu Zhang, Harshad Deshmukh, and Jignesh M. Patel. “Data Partitioning for In-Memory Systems: Myths, Challenges, and Opportunities”. In: *CIDR*. 2019.