Low Latency Query Planning and Processing

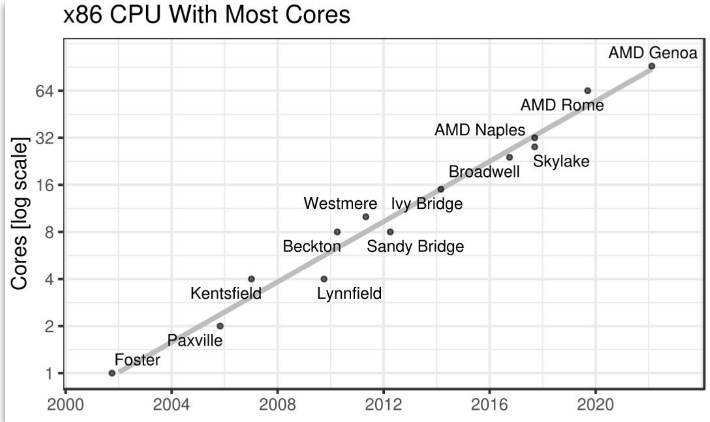
Philipp Fent

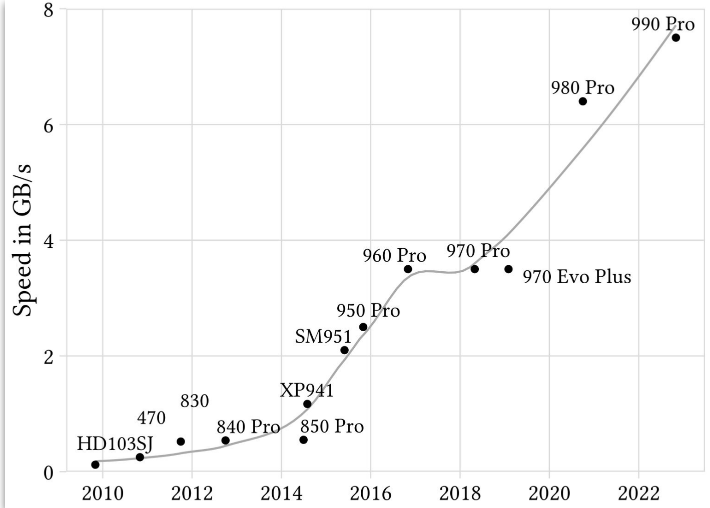
Hardware gets fast

- Large main memory
- Fast SSDs
- Many core machines

Low latency queries

- Still bound by CPU capabilities
- Algorithmic changes





Algorithmic challenges

Query processing

- Intra-query parallelism
- Shared state

Query planning

- Cardinality estimation
- Algebra optimization

A Practical Approach to Groupjoin and Nested Aggregates

Philipp Fent
Technische Universität München
fent@in.tum.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

ABSTRACT

Groupjoins, the combined execution of a join and a subsequent group by, are common in analytical queries, and occur in about 1/8 of the queries in TPC-H and TPC-DS. While they were originally invented to improve performance, efficient parallel execution of groupjoins can be limited by contention, which limits their usefulness in a many-core system. Having an efficient implementation of groupjoins is highly desirable, as groupjoins are not only used to fuse group by and join but are also introduced by the unnesting component of the query optimizer to avoid nested-loops evaluation of aggregates. Furthermore, the query optimizer needs be able to reason over the result of aggregation in order to schedule it correctly. Traditional selectivity and cardinality estimations quickly reach their limits when faced with computed columns from nested aggregates, which leads to poor cost estimations and thus, suboptimal query plans.

In this paper, we present techniques to efficiently estimate, plan, and execute groupjoins and nested aggregates. We propose two
imates to predict the result distribu-
l groupjoin execution for a scalable
sulting system has significantly bet-
free evaluation of groupjoins, which
ries up to a factor of 2.

n. A Practical Approach to Groupjoin
(11): 2383 - 2396, 2021.

Figure 1: Missing components for practical groupjoins. Our improvements to estimation and parallel execution enable efficient evaluation of queries with nested aggregates.

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
```

The VLDB Journal (2023) 32:1165–1190
https://doi.org/10.1007/s00778-022-00765-x

SPECIAL ISSUE PAPER

Practical planning and execution of groupjoin and nested aggregates

Philipp Fent[1] · Altan Birler[1] · Thomas Neumann[1]

Received: 1 April 2022 / Revised: 14 September 2022 / Accepted: 25 September 2022 / Published online: 22 October 2022

Abstract

Groupjoins combine execution of a *join* and a subsequent *group-by*. They are common in analytical queries and occur in about 1/8 of the queries in TPC-H and TPC-DS. While they were originally invented to improve performance, efficient parallel execution of groupjoins can be limited by contention in many-core systems. Efficient implementations of groupjoins are highly desirable, as groupjoins are not only used to fuse group-by and join, but are also useful to efficiently execute nested aggregates. For these, the query optimizer needs to reason over the result of aggregation to optimally schedule it. Traditional systems quickly reach their limits of selectivity and cardinality estimations over computed columns and often treat group-by as an optimization barrier. In this paper, we present techniques to efficiently estimate, plan, and execute groupjoins and nested aggregates. We propose four novel techniques, *aggregate estimates* to predict the result distributions of aggregates, *parallel groupjoin execution* for scalable execution of groupjoins, *index groupjoins*, and a greedy *eager aggregation* optimization technique that introduces nested preaggregations to significantly improve execution plans. The resulting system has improved estimates, better execution plans, and a contention-free evaluation of groupjoins, which speeds up TPC-H and TPC-DS queries significantly.

Keywords Query optimization · Query processing · Parallel processing

1 Introduction

Joins and aggregations are the backbone of query engines. A common query pattern, which we observe in many benchmarks [10,59] and industry applications [77], 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* [56].

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
   GROUP BY cust.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 [58]. A groupjoin directly supports such

---

✉ Philipp Fent
fent@in.tum.de

Altan Birler
altan.birler@tum.de

Thomas Neumann
neumann@in.tum.de

[1] Technische Universität München, Garching, Germany

Asymptotically Better Query Optimization Using Indexed Algebra

Philipp Fent
Technische Universität München
fent@in.tum.de

Guido Moerkotte
Universität Mannheim
moerkotte@uni-mannheim.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

ABSTRACT

Query optimization is essential for the efficient execution of queries. The necessary analysis, if we can and should apply optimizations and transform the query plan, is already challenging. Traditional techniques focus on the availability of columns at individual operators, which does not scale for analysis of data flow through the query. Tracking available columns per operator takes quadratic space, which can result in multi-second optimization time for deep algebra trees. Instead, we need to re-think the naïve algebra representation to efficiently support data flow analysis.

In this paper, we introduce *Indexed Algebra*, a novel representation of relational algebra that makes common optimization tasks efficient. Indexed Algebra enables efficient reasoning with an auxiliary index structure based on link/cut trees that support dynamic updates and queries in $O(\log n)$. This approach not only improves the asymptotic complexity, but also allows elegant and concise formulations for the data flow questions needed for query optimization. While large queries see theoretically unbounded improvements, Indexed Algebra also improves optimization time of the relatively harmless queries of TPC-H and TPC-DS by more than 1.8×.

PVLDB Reference Format:
Philipp Fent, Guido Moerkotte, and Thomas Neumann. Asymptotically Better Query Optimization Using Indexed Algebra. PVLDB, 16(11): 3018 - 3030, 2023.
doi:10.14778/3611479.3611505

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tum-db/indexed-algebra.

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 [5, 18, 19, 37]. 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 [39]. As a result, query optimization usually operates on a budget, trading-off optimizations versus optimization time.

Figure 1: Relation algebra tree with subtle data flow. In this paper, we optimize queries by efficiently analyzing data flow.

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
```

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 [9, 10] 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.

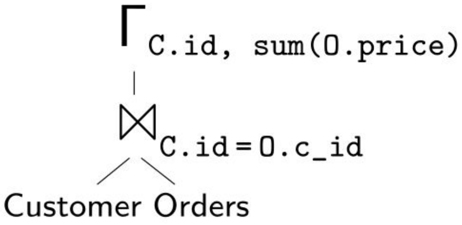
Even if we ignore the performance problems, this myopic look at individual operators is insufficient to express optimizations efficiently. In many cases we want to inspect the full data flow instead. Consider the small algebra tree shown in Figure 1. The top-most join predicate in this tree compares 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 [27]. While we can detect these dependent joins by

---

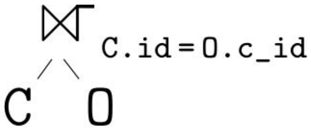
Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611505

Groupjoin – Idea

- Combined execution of compatible join and aggregation
- Q: “Total sales per customer”



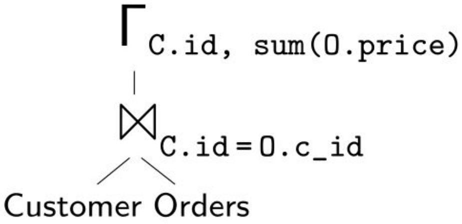
```
join = Hashtable()
for c in customer:
   join[c.id] = c

group = Hashtable()
for o in orders:
   if join.contains(o.c_id):
      group[c_id].sum += o.price
```



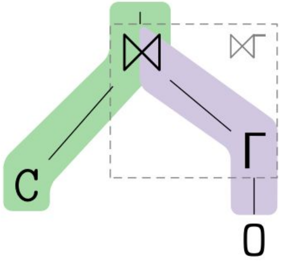
```
groupjoin = Hashtable()
for c in customer:
   groupjoin[c.id] = c

for o in orders:
   if groupjoin.contains(o.c_id):
      groupjoin[c_id].sum += o.price
```

Groupjoin – Avoiding contention

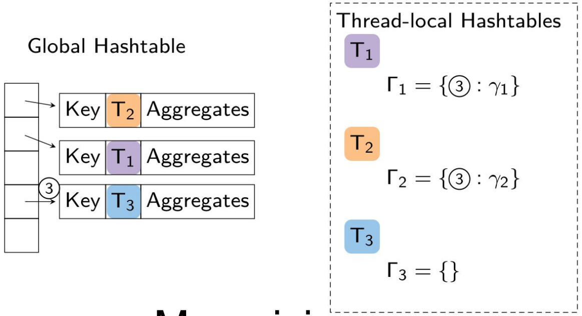
- Shared hash table unsuitable for multithreaded execution
- Four execution strategies for parallel groupjoin:



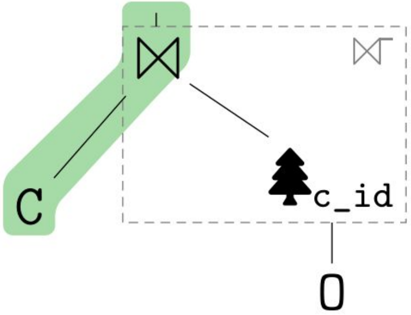
Separate



Eager



Memoizing



Index

Nested Aggregates

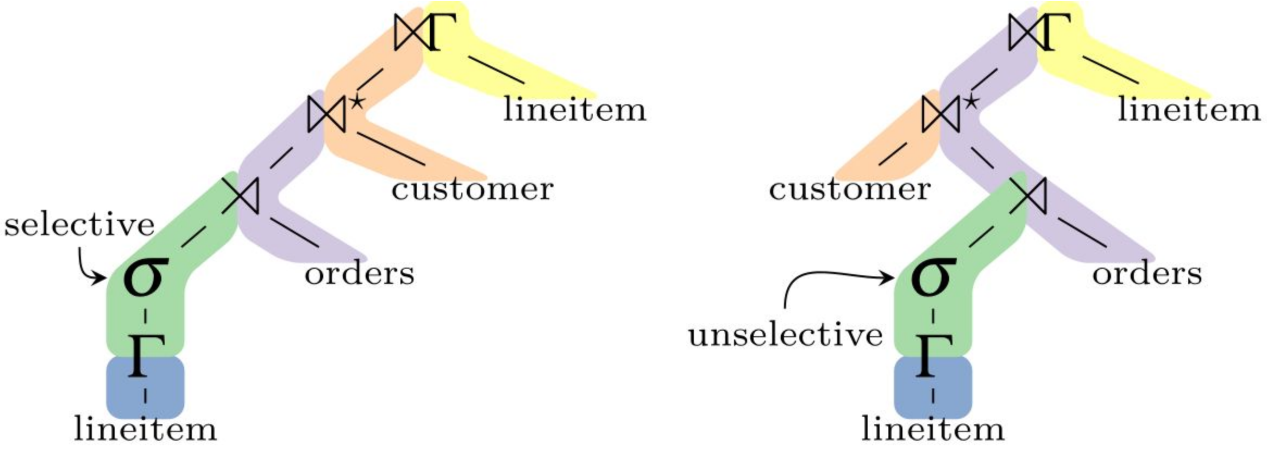
- Common in analytical queries
- `HAVING` predicates are hard to estimate

Q: “Large orders”

```
select l_orderkey
from lineitem
group by l_orderkey
having sum(l_quantity) > 300
```

Nested Aggregates

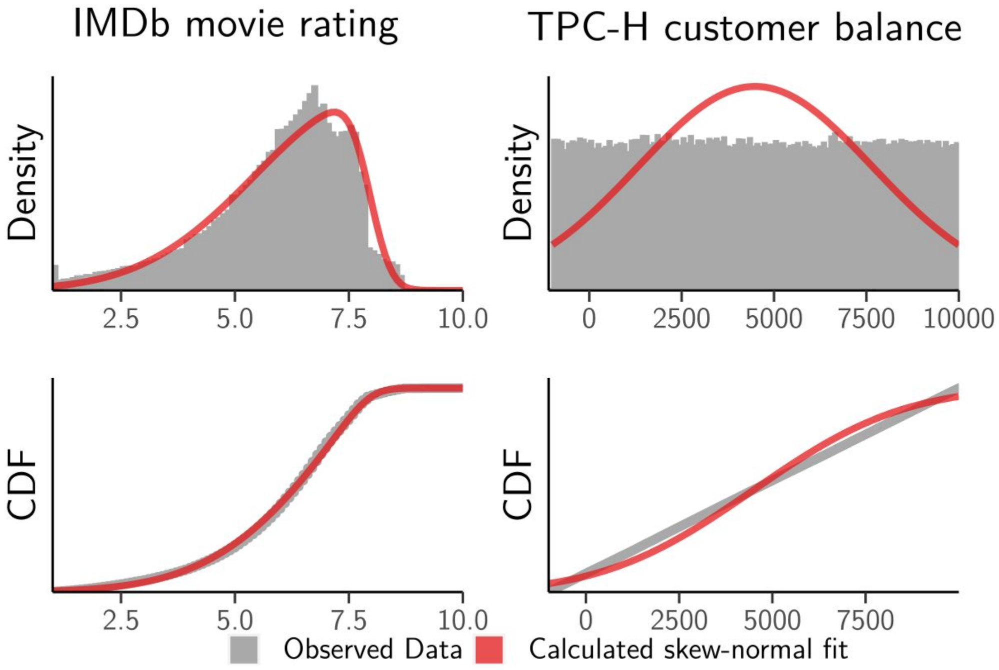
- Common in analytical queries
- `HAVING` predicates are hard to estimate
- But have significant impact on execution plans



a) Selective σ-Predicate b) Unselective σ-Predicate

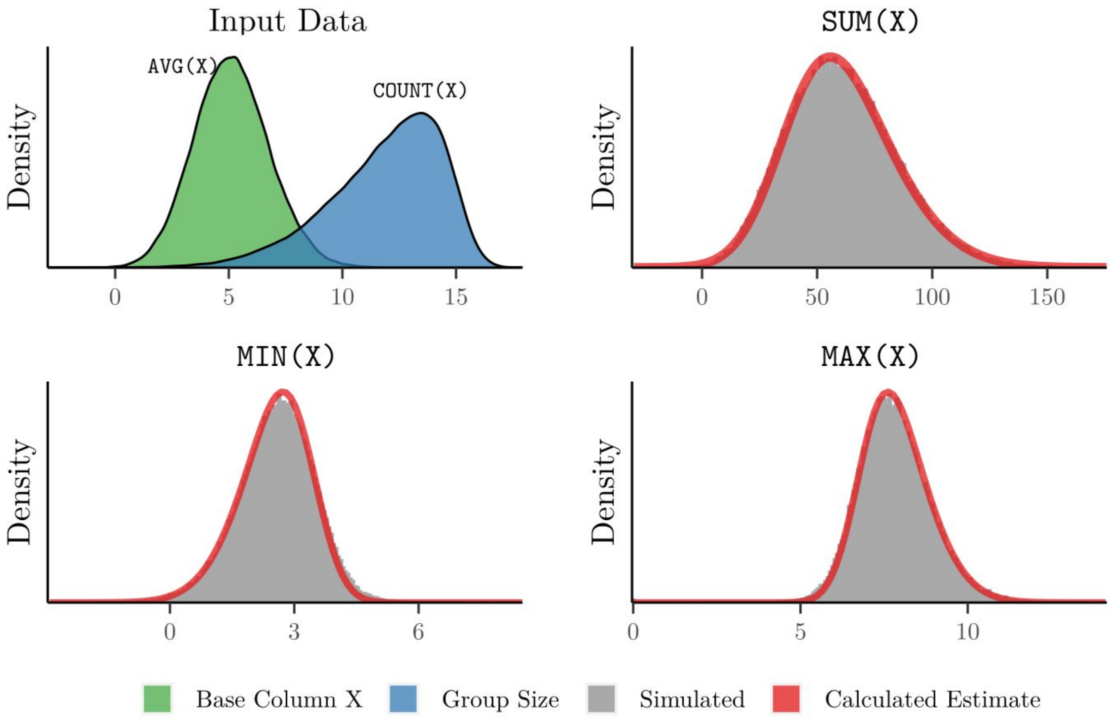
Estimating Aggregates

- Numerical columns ~ $\mathbf{N}(\mu,\sigma^2)$
- Cheap and generalizes nicely, but inherently symmetric



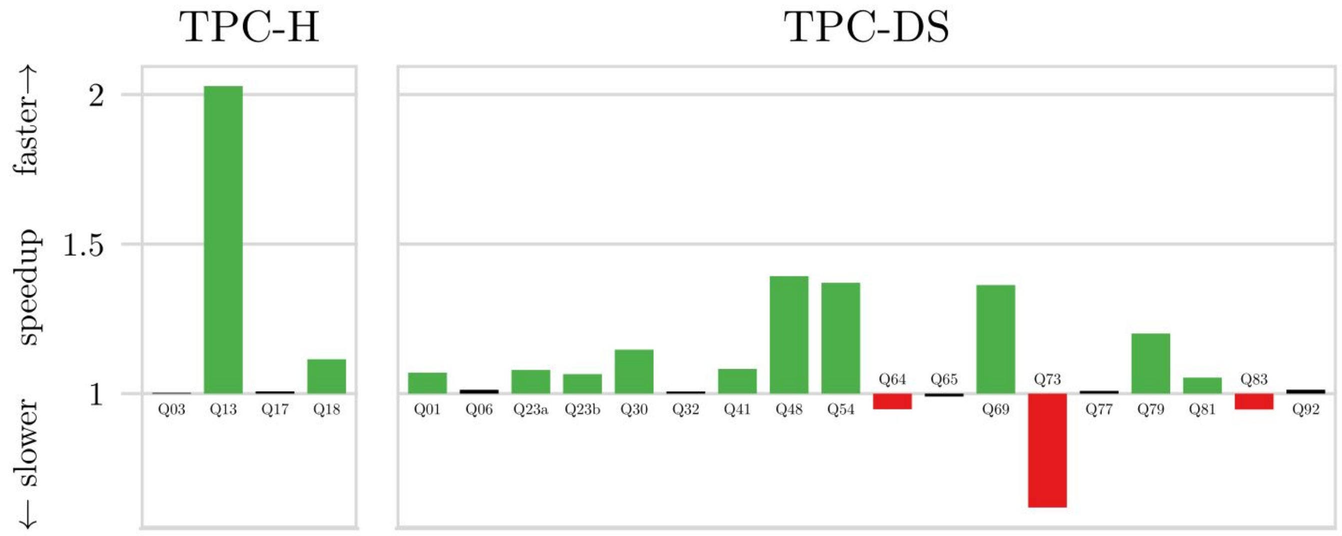
Estimating Aggregates

- Using a skew-normal distribution



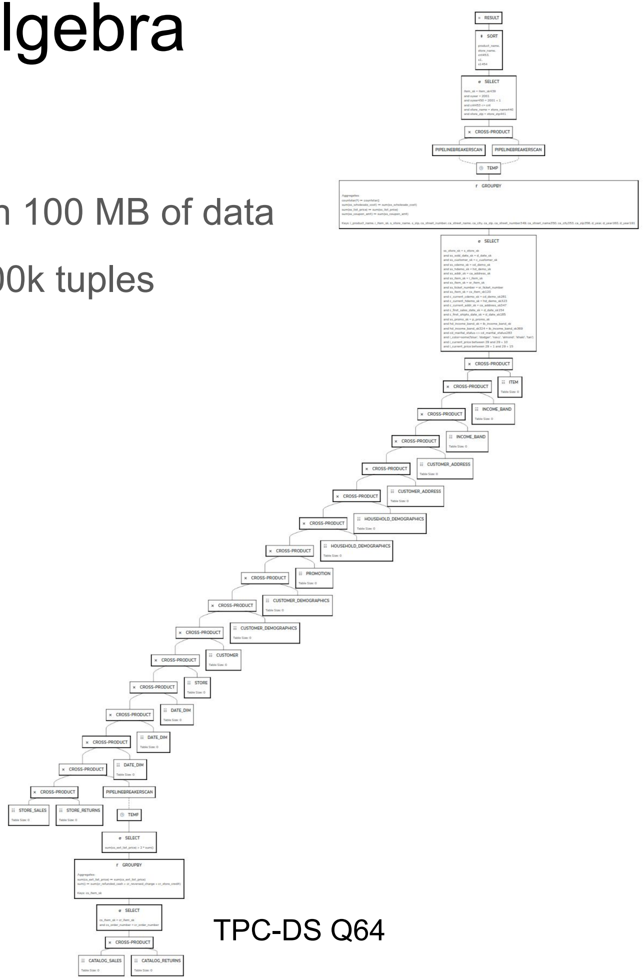
Practical Groupjoins and Nested Aggregates

- Effects ⅛ of queries
- +23% in TPC-H, +6% in TPC-DS



Query Optimization with Indexed Algebra

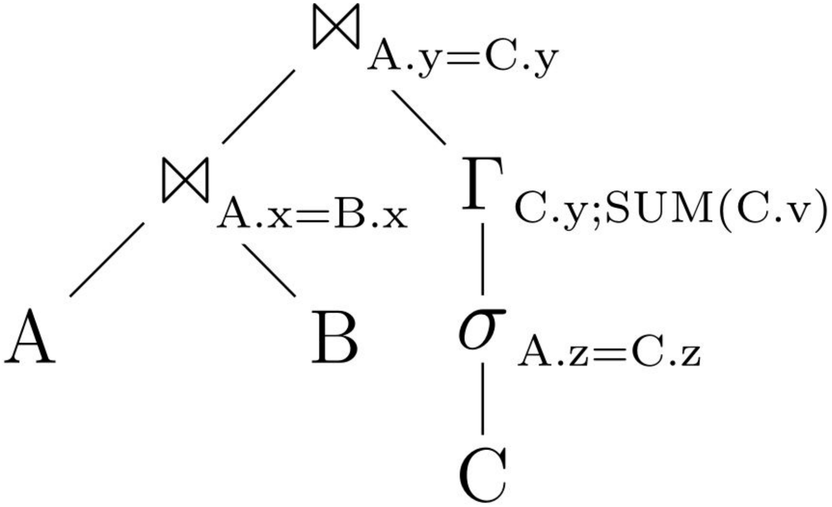
- Complex queries on small workloads
    - BigQuery: 90% of queries processed less than 100 MB of data
    - Tableau Public: 90% of workbooks are less than 100k tuples
- TPC-H
    - Scale 1: 0.8 ms optimization, 20 ms execution
    - Scale 0.01: 0.8 ms optimization, 0.2 ms execution
- Optimization time scales super-linear with query complexity



TPC-DS Q64

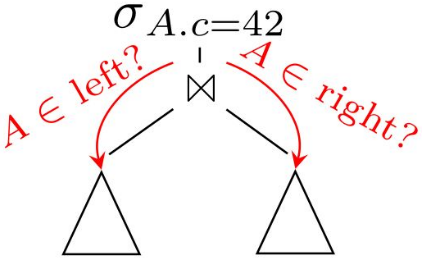
Algebra

- Relational algebra trees
    - Operators
    - Expressions
    - Columns / IUs
- Analyze data-flow for optimization
    - Which path?
    - Modifications?
    - Materialized?

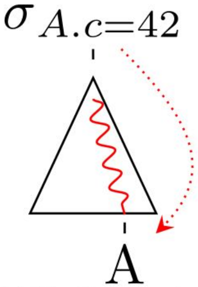


Optimization

- Reason about the algebra to derive optimization possibilities
- Top-down, operator at a time
  - Needs $O(n^2)$ column sets
- Path-centric
  - Still $O(n)$ length
  - With indexing: $O(\log n)$



(a) Operator-centric



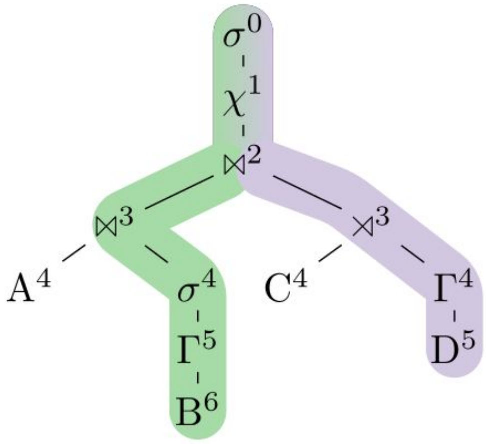
(b) Path-centric

Indexing Algebra

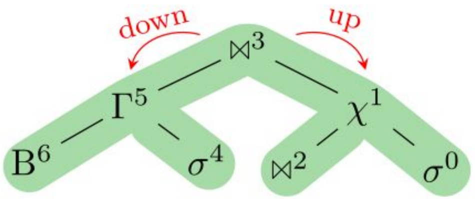
- Index paths through the algebra

  ➡ Faster path traversal

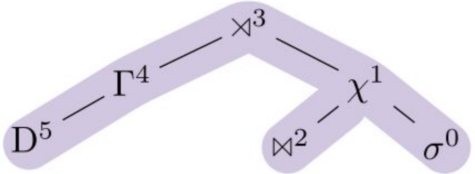
- Binary search trees on path depth
- Paths from root overlap
- **Link/cut trees** support that efficiently



(a) Represented algebra plan



(b) Balanced binary index of the path from B^6 to the root



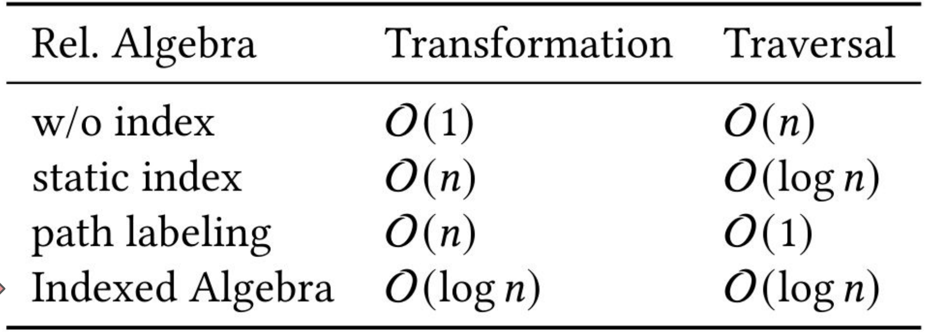
(c) Index from D^5 to the root

Indexing Algebra

- Index paths through the algebra

  ➡ Faster path traversal

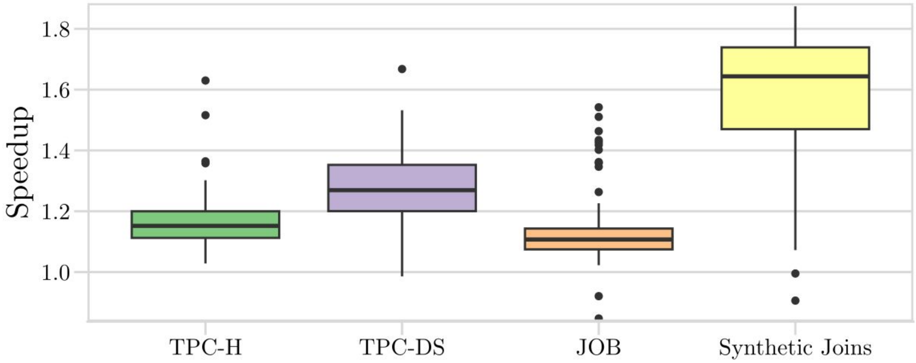
- Binary search trees on path depth
- Paths from root might overlap
- **Link/cut trees** support that efficiently



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	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Indexed Algebra Performance

- Significant overall improvements
- 10 - 30% faster optimization
- 8% better *end-to-end* latency in Tableau Public



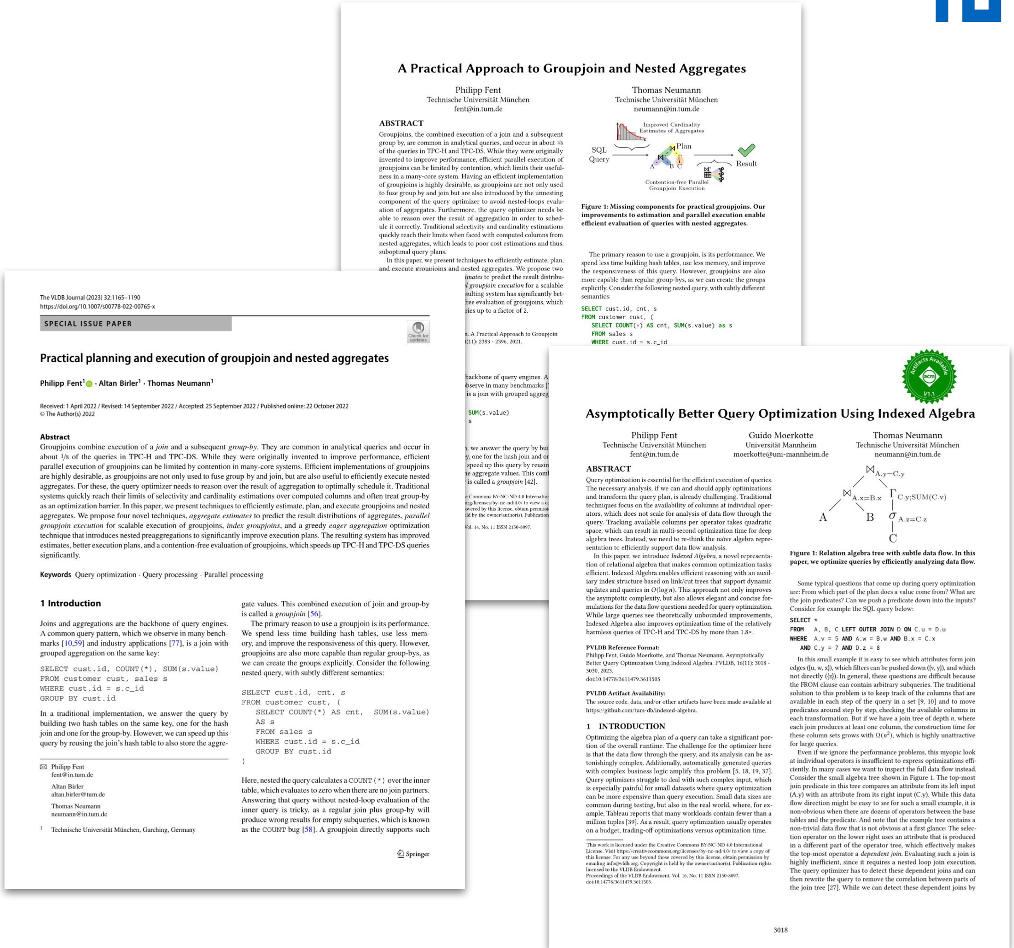
Conclusion

Query processing

- ✅ Intra-query parallelism
- ✅ Shared state

Query planning

- ✅ Cardinality estimation
- ✅ Algebra operations



A Practical Approach to Groupjoin and Nested Aggregates

Philipp Fent
Technische Universität München
fent@in.tum.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

ABSTRACT

Groupjoins, the combined execution of a join and a subsequent group by, are common in analytical queries, and occur in about 1/8 of the queries in TPC-H and TPC-DS. While they were originally invented to improve performance, efficient parallel execution of groupjoins can be limited by contention, which limits their usefulness in a many-core system. Having an efficient implementation of groupjoins is highly desirable, as groupjoins are not only used to fuse group by and join but are also introduced by the unnesting component of the query optimizer to avoid nested-loops evaluation of aggregates. Furthermore, the query optimizer needs be able to reason over the result of aggregation in order to schedule it correctly. Traditional selectivity and cardinality estimations quickly reach their limits when faced with computed columns from nested aggregates, which leads to poor cost estimations and thus, suboptimal query plans.

In this paper, we present techniques to efficiently estimate, plan, and execute groupjoins and nested aggregates. We propose two
imates to predict the result distribu-
l groupjoin execution for a scalable
sulting system has significantly bet-
free evaluation of groupjoins, which
ries up to a factor of 2.

n. A Practical Approach to Groupjoin
(11): 2383 - 2396, 2021.

Figure 1: Missing components for practical groupjoins. Our improvements to estimation and parallel execution enable efficient evaluation of queries with nested aggregates.

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
```

backbone of query engines. A
bserve in many benchmarks [
is a join with grouped aggreg

```
SUM(s.value)
s
```

, we answer the query by bui
y, one for the hash join and o
speed up this query by reusin
e aggregate values. This com
is called a *groupjoin* [42].

Vol. 14, No. 11 ISSN 2150-8097.

The VLDB Journal (2023) 32:1165–1190
https://doi.org/10.1007/s00778-022-00765-x

SPECIAL ISSUE PAPER

Practical planning and execution of groupjoin and nested aggregates

Philipp Fent[1] · **Altan Birler**[1] · **Thomas Neumann**[1]

Received: 1 April 2022 / Revised: 14 September 2022 / Accepted: 25 September 2022 / Published online: 22 October 2022

Abstract

Groupjoins combine execution of a *join* and a subsequent *group-by*. They are common in analytical queries and occur in about 1/8 of the queries in TPC-H and TPC-DS. While they were originally invented to improve performance, efficient parallel execution of groupjoins can be limited by contention in many-core systems. Efficient implementations of groupjoins are highly desirable, as groupjoins are not only used to fuse group-by and join, but are also useful to efficiently execute nested aggregates. For these, the query optimizer needs to reason over the result of aggregation to optimally schedule it. Traditional systems quickly reach their limits of selectivity and cardinality estimations over computed columns and often treat group-by as an optimization barrier. In this paper, we present techniques to efficiently estimate, plan, and execute groupjoins and nested aggregates. We propose four novel techniques, *aggregate estimates* to predict the result distributions of aggregates, *parallel groupjoin execution* for scalable execution of groupjoins, *index groupjoins*, and a greedy *eager aggregation* optimization technique that introduces nested preaggregations to significantly improve execution plans. The resulting system has improved estimates, better execution plans, and a contention-free evaluation of groupjoins, which speeds up TPC-H and TPC-DS queries significantly.

Keywords Query optimization · Query processing · Parallel processing

1 Introduction

Joins and aggregations are the backbone of query engines. A common query pattern, which we observe in many benchmarks [10,59] and industry applications [77], 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* [56].

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
   GROUP BY cust.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 [58]. A groupjoin directly supports such

---

✉ Philipp Fent
fent@in.tum.de

Altan Birler
altan.birler@tum.de

Thomas Neumann
neumann@in.tum.de

[1] Technische Universität München, Garching, Germany

Asymptotically Better Query Optimization Using Indexed Algebra

Philipp Fent
Technische Universität München
fent@in.tum.de

Guido Moerkotte
Universität Mannheim
moerkotte@uni-mannheim.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

ABSTRACT

Query optimization is essential for the efficient execution of queries. The necessary analysis, if we can and should apply optimizations and transform the query plan, is already challenging. Traditional techniques focus on the availability of columns at individual operators, which does not scale for analysis of data flow through the query. Tracking available columns per operator takes quadratic space, which can result in multi-second optimization time for deep algebra trees. Instead, we need to re-think the naïve algebra representation to efficiently support data flow analysis.

In this paper, we introduce *Indexed Algebra*, a novel representation of relational algebra that makes common optimization tasks efficient. Indexed Algebra enables efficient reasoning with an auxiliary index structure based on link/cut trees that support dynamic updates and queries in $O(\log n)$. This approach not only improves the asymptotic complexity, but also allows elegant and concise formulations for the data flow questions needed for query optimization. While large queries see theoretically unbounded improvements, Indexed Algebra also improves optimization time of the relatively harmless queries of TPC-H and TPC-DS by more than 1.8×.

PVLDB Reference Format:
Philipp Fent, Guido Moerkotte, and Thomas Neumann. Asymptotically Better Query Optimization Using Indexed Algebra. PVLDB, 16(11): 3018 - 3030, 2023.
doi:10.14778/3611479.3611505

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tum-db/indexed-algebra.

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 [5, 18, 19, 37]. 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 [39]. As a result, query optimization usually operates on a budget, trading-off optimizations versus optimization time.

Figure 1: Relation algebra tree with subtle data flow. In this paper, we optimize queries by efficiently analyzing data flow.

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
```

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 [9, 10] 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 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 [27]. While we can detect these dependent joins by

---

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611505