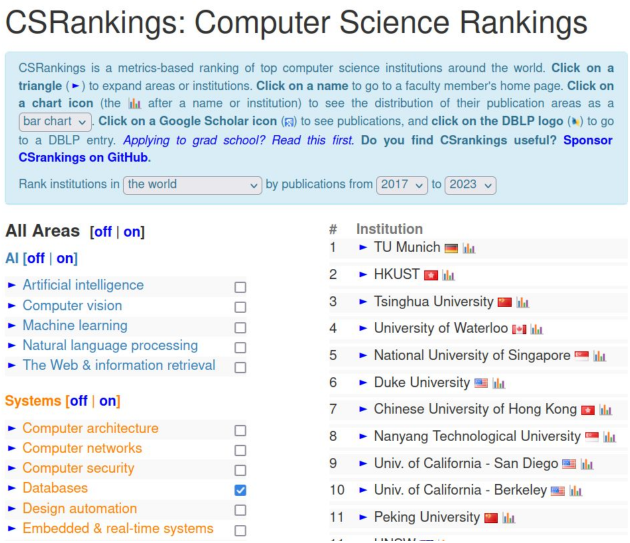
UMBRA

Philipp Fent
Technical University of Munich
fent@in.tum.de

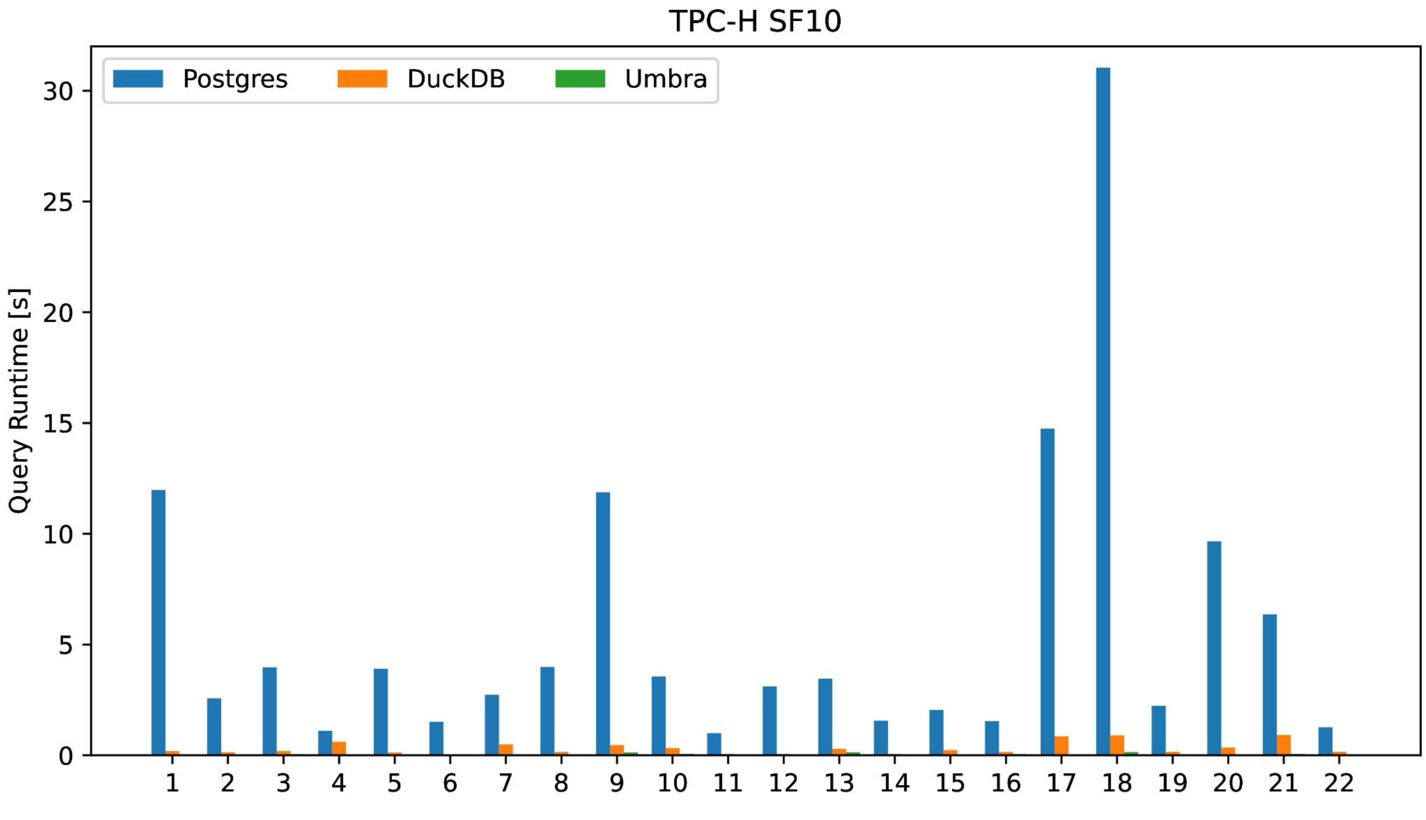
Umbra

- TUM's first DBMS acquired by Salesforce
- Rewrite from scratch
- Cutting-edge database research
- Disk-based with in-memory performance

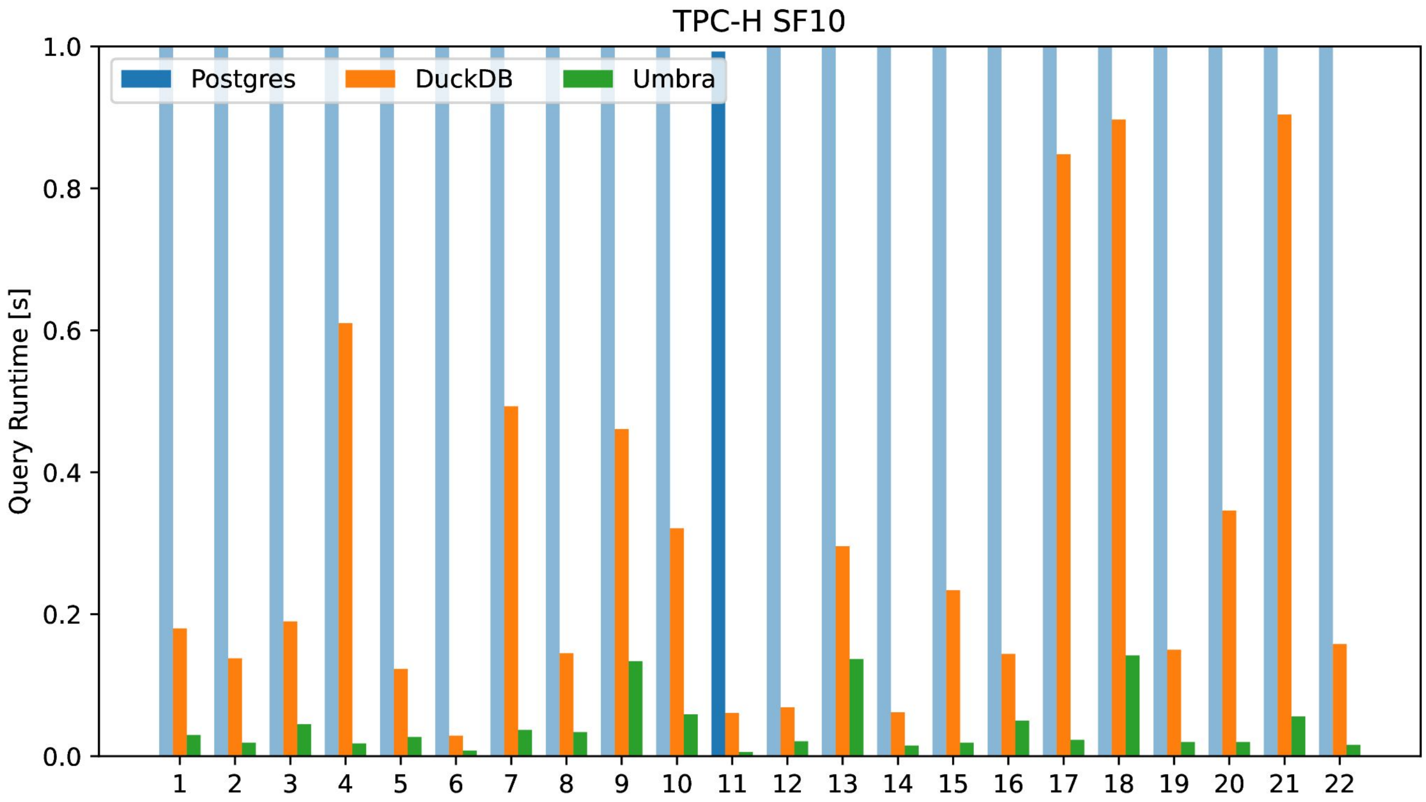
HyPer



Performance



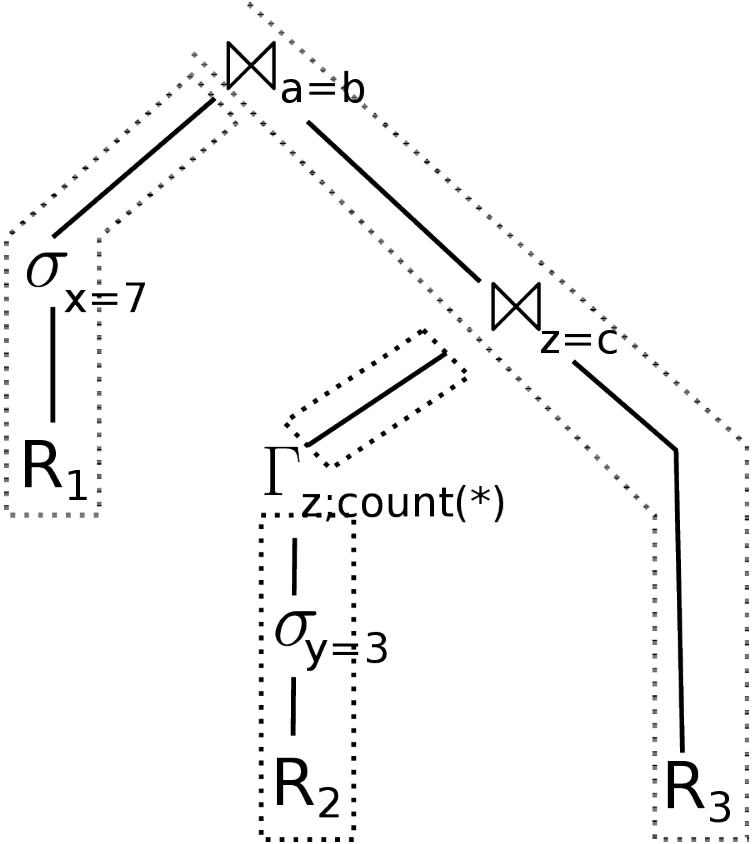
Performance



What makes Umbra fast?

What makes Umbra fast?

- Pipelined execution
    - Keeps values in registers
    - Minimizes materialization



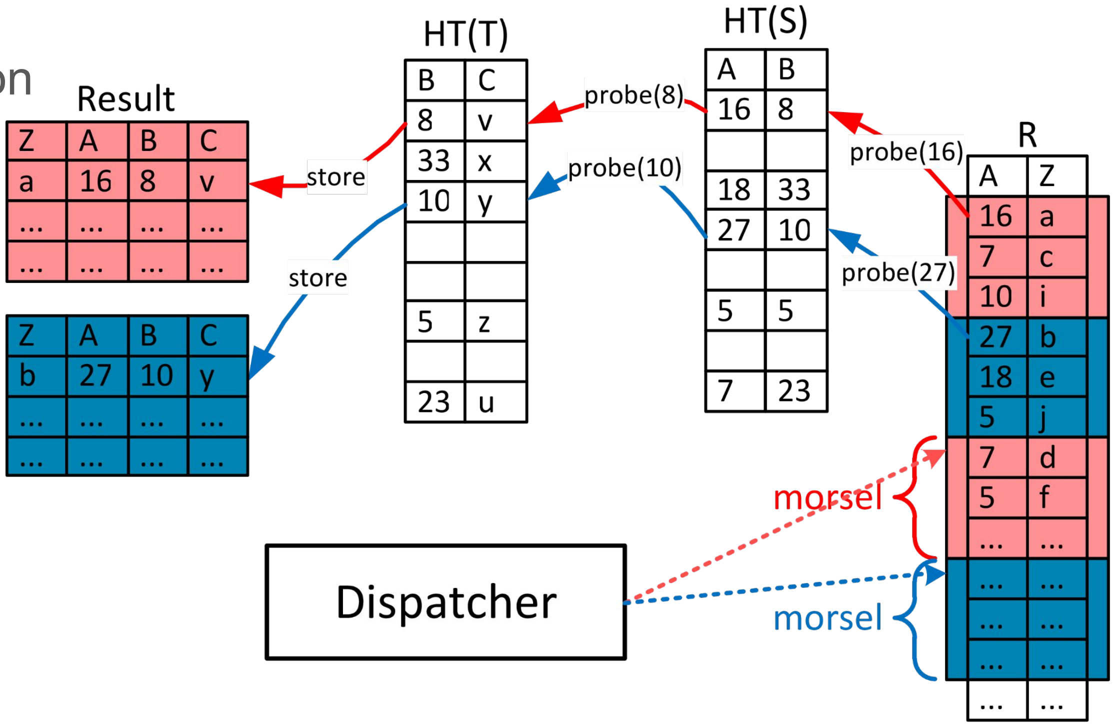
What makes Umbra fast?

- Pipelined execution
- Data-centric code generation
    - Efficient code for complex expressions

```
%1 = zext i64 %int1;                                      Zero extend to 64 bit
%2 = zext i64 %int2;
%3 = rotr i64 %2, 32;                                              Rotate right
%v = or i64 %1, %3;                                       Combine int1 and int2
%5 = crc32 i64 6763793487589347598, %v;                             First crc32
%6 = crc32 i64 4593845798347983834, %v;                            Second crc32
%7 = rotr i64 %6, 32;                                         Shift second part
%8 = xor i64 %5, %7;                                         Combine hash parts
%hash = mul i64 %8, 11400714819323198485;                             Mix parts
```

What makes Umbra fast?

- Pipelined execution
- Data-centric code generation
- Fully parallel algorithms
  - Allows scaling
  - Benefits from new hardware



What makes Umbra fast?

- Pipelined execution
- Data-centric code generation
- Fully parallel algorithms
- **State-of-the-art query optimizer**

What makes Umbra fast?

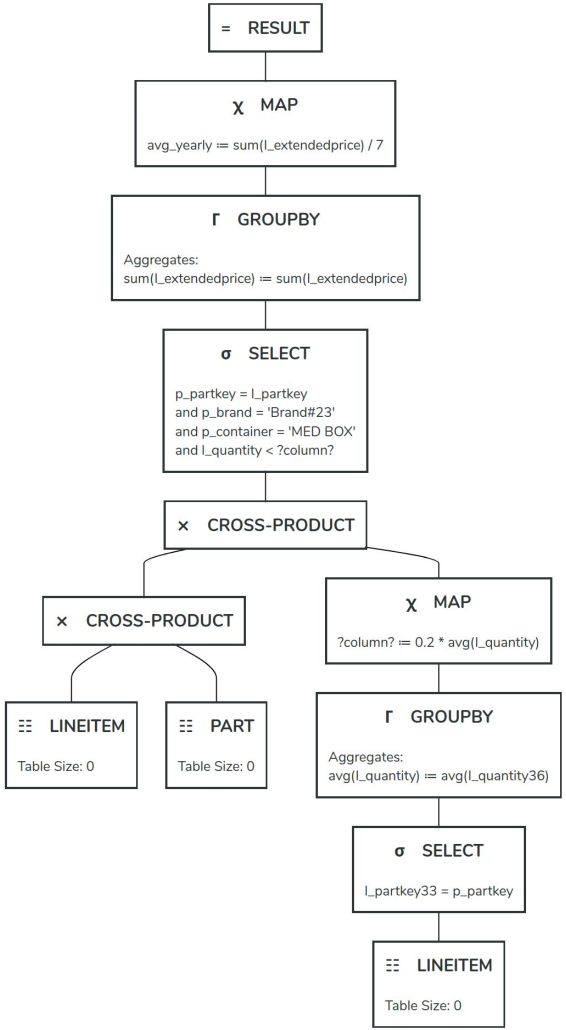
- Pipelined execution
- Data-centric code generation
- Fully parallel algorithms
- **State-of-the-art query optimizer**

Research system with all custom advanced parts

We’re commercializing soon!

Query Optimization

- PostgreSQL grammar
- Parsed into relational algebra
  - Example: TPC-H Q17
  - https://umbra-db.com/interface/



Query Optimization

- PostgreSQL grammar
- Parsed into relational algebra
- Optimizer passes over algebra

1: Unoptimized Plan

2: Expression Simplification

3: Unnesting

4: Predicate Pushdown

5: Initial Join Tree

6: Sideway Information Passing

7: Operator Reordering

8: Early Probing

9: Common Subtree Elimination

10: Physical Operator Mapping

Query Optimization

- PostgreSQL grammar
- Parsed into relational algebra
- Optimizer passes over algebra

Rule-based Canonicalization

1: Unoptimized Plan
2: Expression Simplification
3: Unnesting
4: Predicate Pushdown
5: Initial Join Tree

Cost-based Optimization

6: Sideway Information Passing
7: Operator Reordering
8: Early Probing
9: Common Subtree Elimination
10: Physical Operator Mapping

Expression Simplification

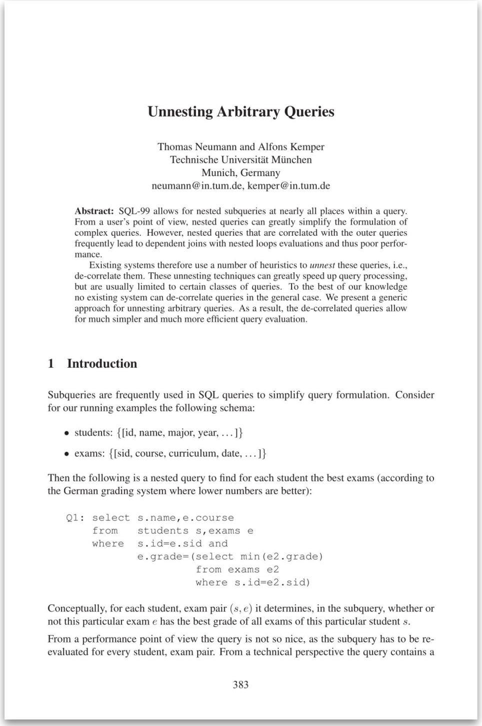
- Fold constants
- Canonicalize expressions

```
    o_orderdate >= date '1994-01-01'
and o_orderdate <  date '1994-01-01' + interval '1' year

                         ==

o_orderdate between date '1994-01-01' and date '1994-12-31'
```

- Execute in evaluation engine

Query Unnesting & Decorrelation

- Unnesting Arbitrary Queries



Unnesting Arbitrary Queries

Thomas Neumann and Alfons Kemper
Technische Universität München
Munich, Germany
neumann@in.tum.de, kemper@in.tum.de

Abstract: SQL-99 allows for nested subqueries at nearly all places within a query. From a user's point of view, nested queries can greatly simplify the formulation of complex queries. However, nested queries that are correlated with the outer queries frequently lead to dependent joins with nested loops evaluations and thus poor performance.

Existing systems therefore use a number of heuristics to *unnest* these queries, i.e., de-correlate them. These unnesting techniques can greatly speed up query processing, but are usually limited to certain classes of queries. To the best of our knowledge no existing system can de-correlate queries in the general case. We present a generic approach for unnesting arbitrary queries. As a result, the de-correlated queries allow for much simpler and much more efficient query evaluation.

1 Introduction

Subqueries are frequently used in SQL queries to simplify query formulation. Consider for our running examples the following schema:

- students: {[id, name, major, year, ...]}
- exams: {[sid, course, curriculum, date, ...]}

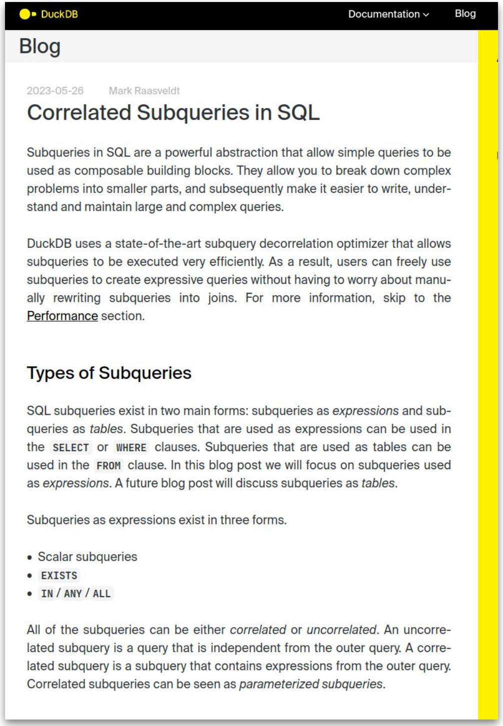
Then the following is a nested query to find for each student the best exams (according to the German grading system where lower numbers are better):

```
Q1: select s.name,e.course
    from   students s,exams e
    where  s.id=e.sid and
           e.grade=(select min(e2.grade)
                    from exams e2
                    where s.id=e2.sid)
```

Conceptually, for each student, exam pair (s, e) it determines, in the subquery, whether or not this particular exam e has the best grade of all exams of this particular student s.

From a performance point of view the query is not so nice, as the subquery has to be re-evaluated for every student, exam pair. From a technical perspective the query contains a

383



DuckDB Documentation Blog

Blog

2023-05-26 Mark Raasveldt

Correlated Subqueries in SQL

Subqueries in SQL are a powerful abstraction that allow simple queries to be used as composable building blocks. They allow you to break down complex problems into smaller parts, and subsequently make it easier to write, understand and maintain large and complex queries.

DuckDB uses a state-of-the-art subquery decorrelation optimizer that allows subqueries to be executed very efficiently. As a result, users can freely use subqueries to create expressive queries without having to worry about manually rewriting subqueries into joins. For more information, skip to the Performance section.

Types of Subqueries

SQL subqueries exist in two main forms: subqueries as *expressions* and subqueries as *tables*. Subqueries that are used as expressions can be used in the `SELECT` or `WHERE` clauses. Subqueries that are used as tables can be used in the `FROM` clause. In this blog post we will focus on subqueries used as *expressions*. A future blog post will discuss subqueries as *tables*.

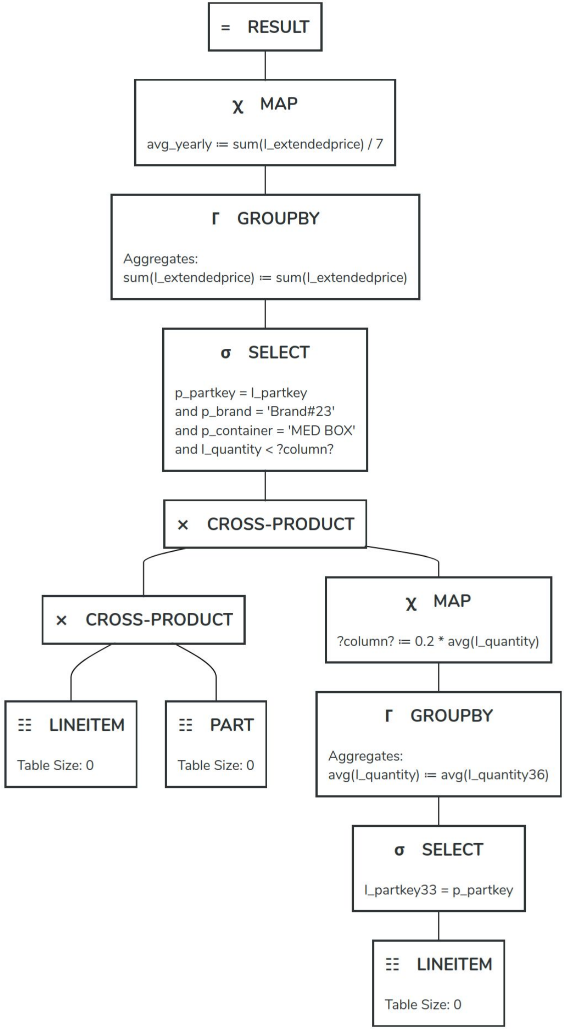
Subqueries as expressions exist in three forms.

- Scalar subqueries
- `EXISTS`
- `IN` / `ANY` / `ALL`

All of the subqueries can be either *correlated* or *uncorrelated*. An uncorrelated subquery is a query that is independent from the outer query. A correlated subquery is a subquery that contains expressions from the outer query. Correlated subqueries can be seen as *parameterized subqueries*.

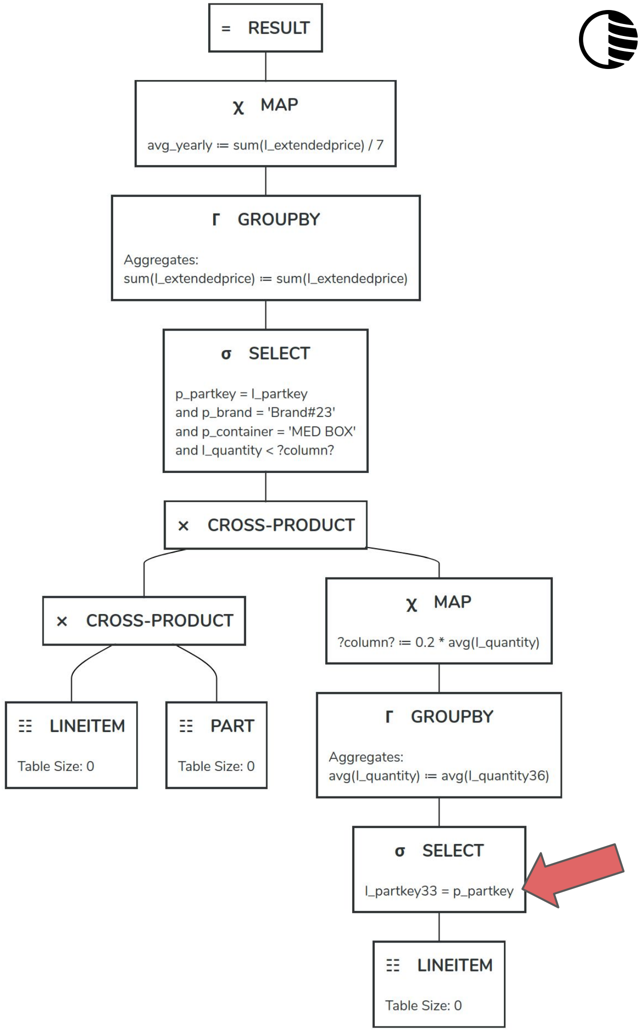
Query Unnesting

- Unnesting Arbitrary Queries
  - O(n²)



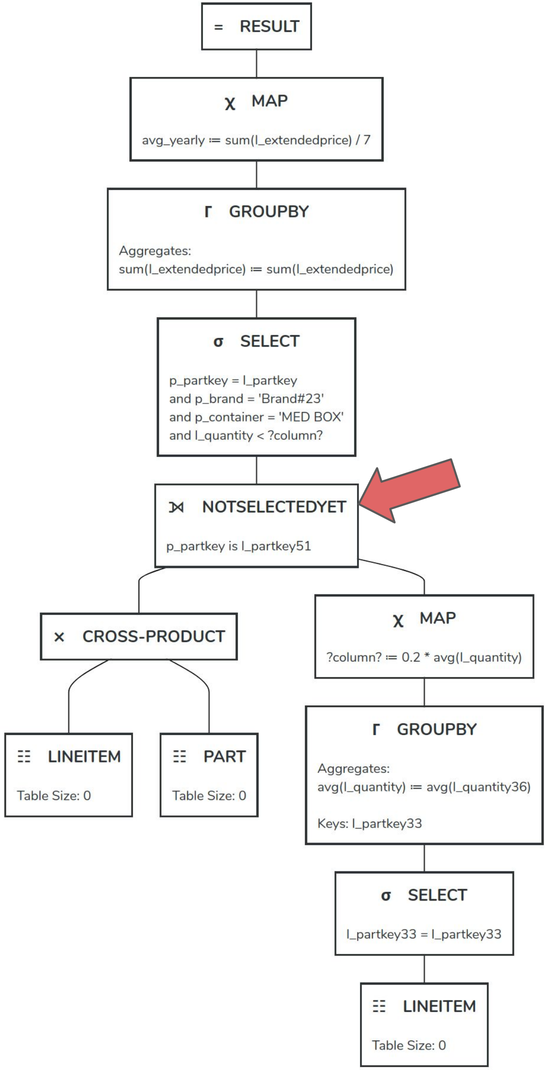
Query Unnesting

- Unnesting Arbitrary Queries
  - $O(n^2)$



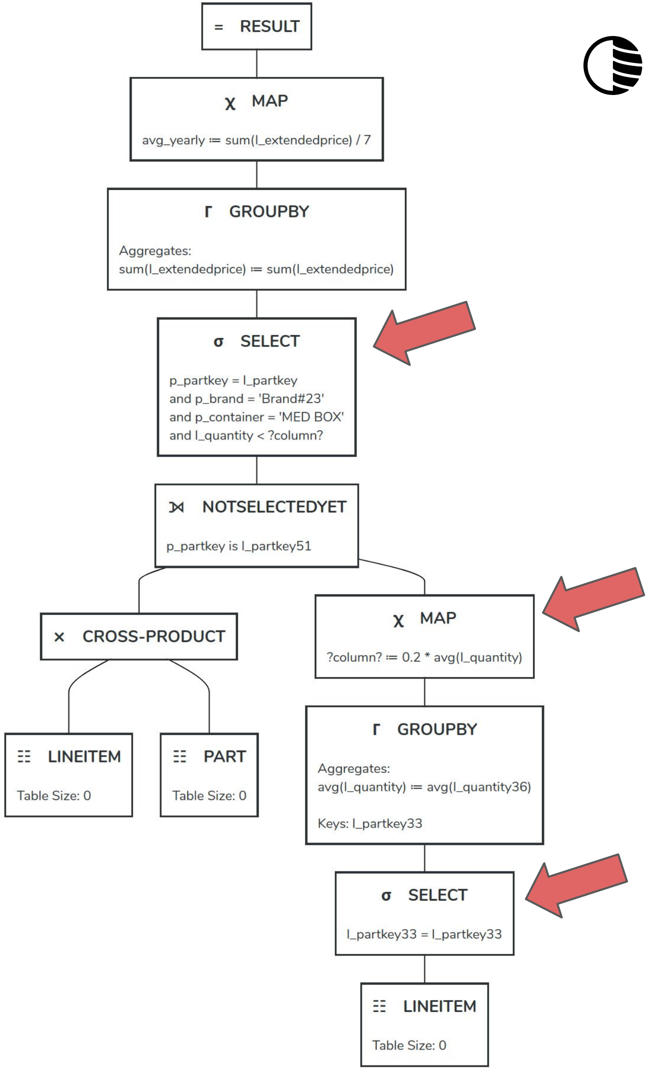
Query Unnesting

- Unnesting Arbitrary Queries
  - $O(n^2) \rightarrow O(n)$
  - Huge improvement



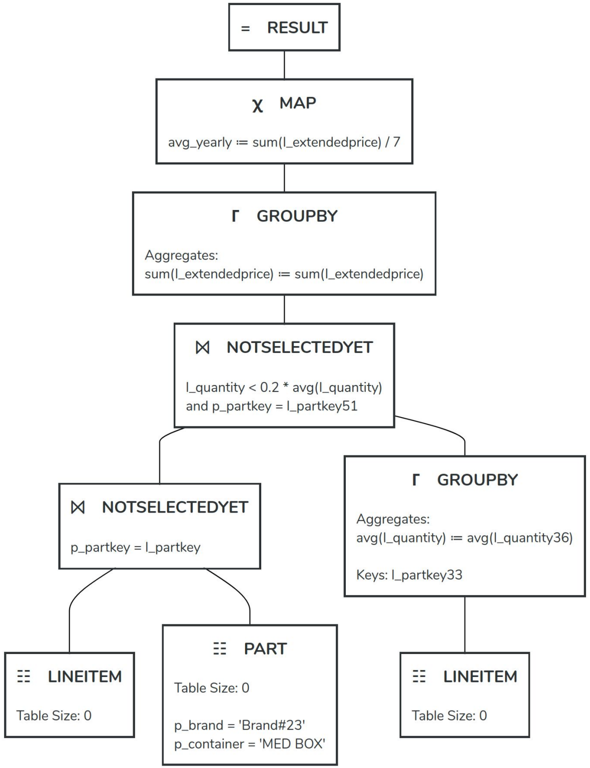
Predicate Pushdown

- Place predicates at scan
- Propagate & fold constants



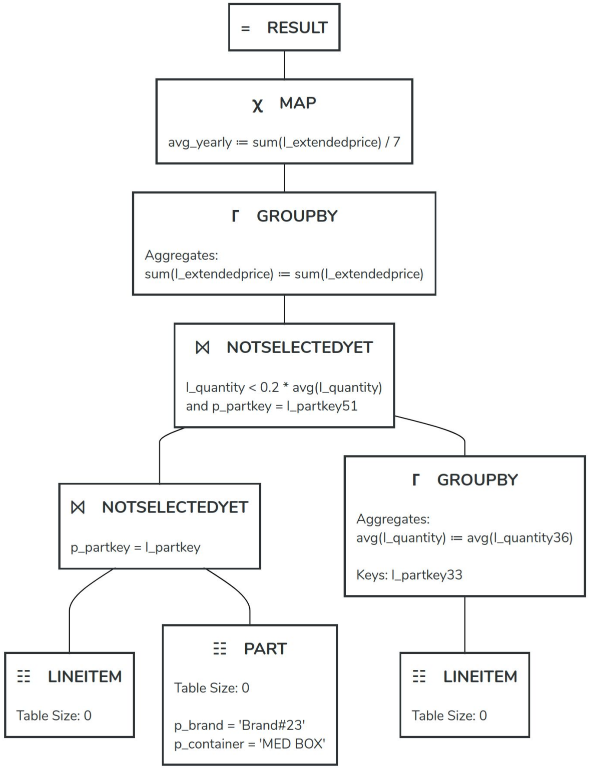
Predicate Pushdown

- Place predicates at scan
- Propagate & fold constants



Predicate Pushdown

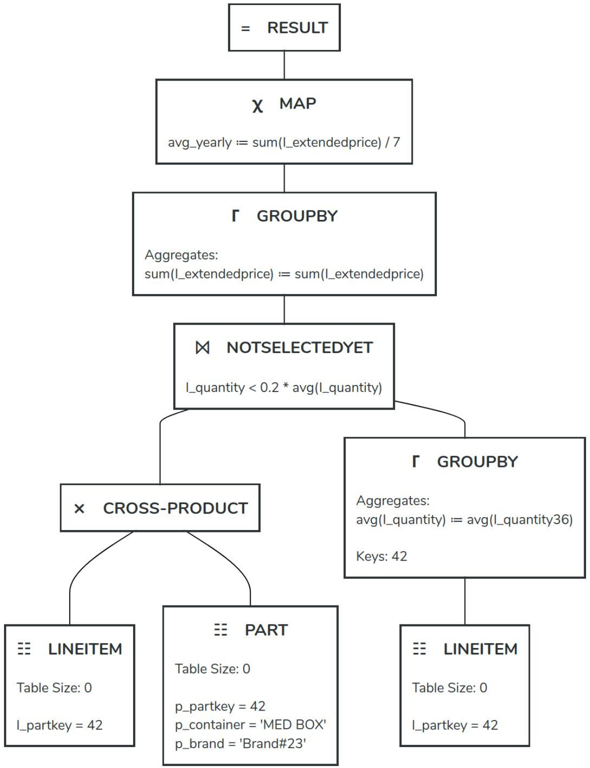
- Place predicates at scan
- Propagate & fold constants




```
where p_partkey = 42
```

Predicate Pushdown

- Place predicates at scan
- Propagate & fold constants



Initial Join Tree

- Push joins through aggregates
- Expand transitive join conditions

```
    c_nationkey = s_nationkey
and s_nationkey = n_nationkey

              ==

    c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and c_nationkey = n_nationkey
```

Initial Join Tree

- Push joins through aggregates
- Expand transitive join conditions
- Drop unnecessary joins

```
select sum(o_totalprice)
  from customer, orders
 where c_custkey = o_custkey

             ==

select sum(o_totalprice)
  from orders
```

Cost-Based Optimization

- Heuristics vs. statistics

Cost-Based Optimization

- Heuristics vs. statistics
- Statistics in Umbra:
  - Samples
  - Distinct counts
  - Numerical statistics (mean, variance) for aggregates
  - Functional dependencies

⇒ Estimate execution cost

Sample Evaluation

- Maintain uniform reservoir sample
- Evaluate scan predicates σ on sample
- Execute in evaluation engine
- Surprisingly accurate
  - 1024 tuples ~ 0.1% error

```
select count(*)
  from lineitem
 where l_commitdate < l_receiptdate
   and l_shipdate < l_commitdate
```

Sample Evaluation

```
for l in lineitem:
  if not l_shipdate < l_commitdate:
    continue   -- 51% taken
  if not l_commitdate < l_receiptdate:
    continue   -- 75% taken

  counter++
```

Variant (A) : Separate branches

```
for l in lineitem:
  if not l_commitdate < l_receiptdate:
    continue   -- 37% taken
  if not l_shipdate < l_commitdate:
    continue   -- 81% taken

  counter++
```

Variant (B) : Separate branches

```
for l in lineitem:
  if not (l_shipdate < l_commitdate
     and l_commitdate < l_receiptdate):
    continue   -- 88% taken

  counter++
```

Variant (C) : Combined branch

Sample Evaluation

```
for l in lineitem:
  if not l_shipdate < l_commitdate:
    continue  -- 51% taken
  if not l_commitdate < l_receiptdate:
    continue  -- 75% taken

  counter++
```

Variant A: Separate branches

```
for l in lineitem:
  if not l_commitdate < l_receiptdate:
    continue  -- 37% taken
  if not l_shipdate < l_commitdate:
    continue  -- 81% taken

  counter++
```

Variant B: Separate branches

```
for l in lineitem:
  if not (l_shipdate < l_commitdate
     and l_commitdate < l_receiptdate):
    continue  -- 88% taken

  counter++
```

Variant C: Combined branch

Variant	branch-misses	instructions	loads	exec. time
A	0.63 / tpl	7.62 / tpl	2.85 / tpl	18.4 ms
B	0.58 / tpl	7.91 / tpl	3.00 / tpl	17.7 ms
C	0.13 / tpl	11.67 / tpl	3.37 / tpl	**12.7 ms**

Sample Evaluation

- Estimate (correlated) predicates with confidence
- Any combination of predicates
- Tricky when 0 / 1024 tuples qualify
- Can do better for conjunctions

Small Selectivities Matter: Lifting the Burden of Empty Samples

Axel Hertzschuch
Technische Universität Dresden
axel.hertzschuch@tu-dresden.de

Guido Moerkotte
University of Mannheim
moerkotte@uni-mannheim.de

Wolfgang Lehner
Technische Universität Dresden
wolfgang.lehner@tu-dresden.de

Norman May
SAP SE
norman.may@sap.com

Florian Wolf
SAP SE
florian.wolf01@sap.com

Lars Fricke
SAP SE
lars.fricke@sap.com

ABSTRACT

Every year more and more advanced approaches to cardinality estimation are published, using learned models or other data and workload specific synopses. In contrast, the majority of commercial in-memory systems still relies on sampling. It is arguably the most general and easiest estimator to implement. While most methods do not seem to improve much over sampling-based estimators in the presence of non-selective queries, sampling struggles with highly selective queries due to limitations of the sample size. Especially in situations where no sample tuple qualifies, optimizers fall back to basic heuristics that ignore attribute correlations and lead to large estimation errors. In this work, we present a novel approach, dealing with these *0-Tuple Situations*. It is ready to use in any DBMS capable of sampling, showing a negligible impact on optimization time. Our experiments on real world and synthetic data sets demonstrate up to two orders of magnitude reduced estimation errors. Enumerating single filter predicates according to our estimates reveals 1.3 to 1.8 times faster query responses for complex filters.

ACM Reference Format:
Axel Hertzschuch, Guido Moerkotte, Wolfgang Lehner, Norman May, Florian Wolf, and Lars Fricke. 2021. Small Selectivities Matter: Lifting the Burden of Empty Samples. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452805

1 INTRODUCTION

Good cardinality estimates guide query optimizers towards decent execution plans and lower the risk of disastrous plans [25, 28]. Although many approaches were published on cardinality estimation, e.g., using histograms [18], sampling [11], or machine learning [13], it is still considered a grand challenge [28]. Especially analytical workloads remain challenging as they often comprise a multitude of correlated filter predicates. The comprehensive analysis of 60k real-world BI data repositories by Vogelsgesang et al. [45] underlines the importance of filter operations and reveals: Most data is stored in string format, which enables arbitrary complex expressions.

SIGMOD '21, June 18–27, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452805

Figure 1: Relative number of queries over tables with at least 1M tuples that lead to empty samples (0-TS) with regard to the number of filter predicates (atoms) and the sample size.

Sampling is an ad-hoc approach that captures correlations among arbitrary numbers and types of predicates. Therefore, it is commonly used in commercial systems [25, 26, 36, 40] and has been combined with histograms [35] and machine learning [23, 47]. However, it is not a panacea. Although sampling might be reasonably fast for in-memory systems due to the efficient random access [17], the number of sample tuples often is very limited. Traditionally, we randomly draw a fixed number of tuples from a table and divide the number of qualifying sample tuples by the total number of sample tuples. Instead of drawing the sample at query time, some approaches exploit materialized views [24] or use reservoir sampling [7, 44]. Given a sufficient number of qualifying tuples, these sample-based estimates are precise and give probabilistic error guarantees [32]. However, complex predicates frequently lead to situations where no sample tuple qualifies. According to Kipf et al. [22] we call these *0-Tuple Situations* (0-TS). To assess the frequency at which 0-TS occur, we analyze the *Public Bi Benchmark* [2], a real-world, user-generated workload. Considering base tables with at least 1M tuples, Figure 1 illustrates the relative number of queries that result in 0-TS when using two standard sized random samples. Interestingly, and contrary to the intuition of being a corner case, this analysis of a real-life workload reveals that up to 72% of the queries with complex filters lead to empty samples. In these situations, query optimizers rely on basic heuristics, e.g., using *Attribute Value Independence* (AVI), that lead to large estimation errors and potentially poor execution plans [33, 37]. To illustrate this deficiency, suppose we sample from a table containing *brands*, *models* and *colors* of cars. Even if no sample tuple qualifies for a given filter, there is little justification to assume independence between all attributes as the *model* usually determines the *brand*.

Surprisingly, no previous work we are aware of considers correlations in 0-TS. This paper therefore presents a novel approach that – given a sample – derives more precise selectivity estimates

Sample Evaluation

- Calculate matches-bitsets
- Combine them to optimize ordering
  - TPC-H Q12:

```
where l_shipmode in ('MAIL', 'SHIP')
  and l_commitdate < l_receiptdate
  and l_shipdate < l_commitdate
  and l_receiptdate between date '1994-01-01'
                          and date '1994-12-31'
```

```
  0100’0011’1010’0100’1110’1011’1011’1100’1010’1010’1011’0000’1011’0011’1100’0000
& 0000’1111’0000’1111’0000’1111’0000’1111’0000’1111’0000’1111’0000’1111’0000’1111
& 1111’0000’1111’0000’1111’0000’1111’0000’1111’0000’1111’0000’1111’0000’1111’0000
& 1010’0110’1110’1110’1000’0011’0111’0101’0110’1111’1001’1101’1110’0011’1000’0001
```

Early Execution

- Size of sample > table size
- Allows a third round of constant propagation
    - Especially for small fact tables

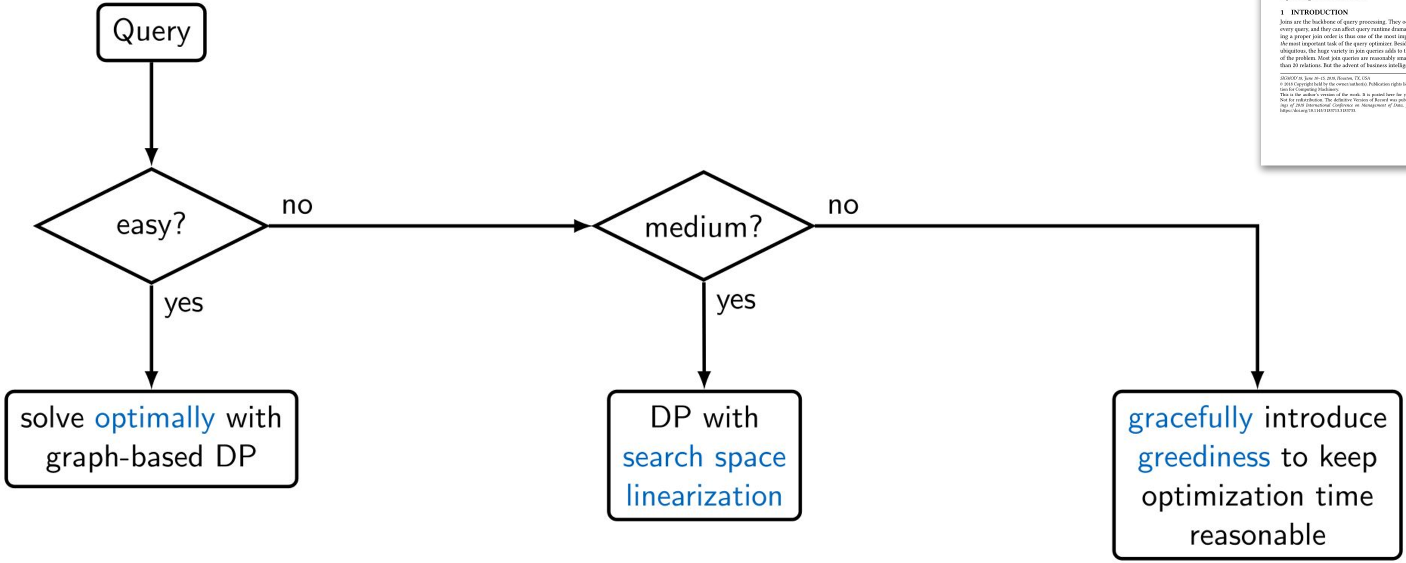
```
select r_regionkey
  from region
 where r_name = 'Europe'

   ==

select 3
```

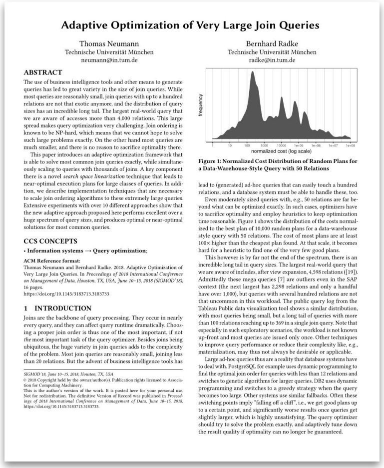
Join Ordering

- Hash Joins rule
  - Indexes don’t allow bushy plans -> less useful

Join Ordering

- Hash Joins rule
    - Indexes don’t allow bushy plans -> less useful





Adaptive Optimization of Very Large Join Queries

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

Bernhard Radke
Technische Universität München
radke@in.tum.de

ABSTRACT

The use of business intelligence tools and other means to generate queries has led to great variety in the size of join queries. While most queries are reasonably small, join queries with up to a hundred relations are not that exotic anymore, and the distribution of query sizes has an incredible long tail. The largest real-world query that we are aware of accesses more than 4,000 relations. This large spread makes query optimization very challenging. Join ordering is known to be NP-hard, which means that we cannot hope to solve such large problems exactly. On the other hand most queries are much smaller, and there is no reason to sacrifice optimality there.

This paper introduces an adaptive optimization framework that is able to solve most common join queries exactly, while simultaneously scaling to queries with thousands of joins. A key component there is a novel *search space linearization* technique that leads to near-optimal execution plans for large classes of queries. In addition, we describe implementation techniques that are necessary to scale join ordering algorithms to these extremely large queries. Extensive experiments with over 10 different approaches show that the new adaptive approach proposed here performs excellent over a huge spectrum of query sizes, and produces optimal or near-optimal solutions for most common queries.

CCS CONCEPTS

- **Information systems → Query optimization**;

ACM Reference format:
Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of 2018 International Conference on Management of Data, Houston, TX, USA, June 10–15, 2018 (SIGMOD’18),* 16 pages.
https://doi.org/10.1145/3183713.3183733

1 INTRODUCTION

Joins are the backbone of query processing. They occur in nearly every query, and they can affect query runtime dramatically. Choosing a proper join order is thus one of the most important, if not *the* most important task of the query optimizer. Besides joins being ubiquitous, the huge variety in join queries adds to the complexity of the problem. Most join queries are reasonably small, joining less than 20 relations. But the advent of business intelligence tools has lead to (generated) ad-hoc queries that can easily touch a hundred relations, and a database system must be able to handle these, too.

Even moderately sized queries with, e.g., 50 relations are far beyond what can be optimized exactly. In such cases, optimizers have to sacrifice optimality and employ heuristics to keep optimization time reasonable. Figure 1 shows the distribution of the costs normalized to the best plan of 10,000 random plans for a data-warehouse style query with 50 relations. The cost of most plans are at least 100× higher than the cheapest plan found. At that scale, it becomes hard for a heuristic to find one of the very few good plans.

This however is by far not the end of the spectrum, there is an incredible long tail in query sizes. The largest real-world query that we are aware of includes, after view expansion, 4,598 relations ([19]). Admittedly these mega queries [7] are outliers even in the SAP context (the next largest has 2,298 relations and only a handful have over 1,000), but queries with several hundred relations are not that uncommon in this workload. The public query log from the Tableau Public data visualization tool shows a similar distribution, with most queries being small, but a long tail of queries with more than 100 relations reaching up to 369 in a single join query. Note that especially in such exploratory scenarios, the workload is not known up-front and most queries are issued only once. Other techniques to improve query performance or reduce their complexity like, e.g., materialization, may thus not always be desirable or applicable.

Large ad-hoc queries thus are a reality that database systems have to deal with. PostgreSQL for example uses dynamic programming to find the optimal join order for queries with less than 12 relations and switches to genetic algorithms for larger queries. DB2 uses dynamic programming and switches to a greedy strategy when the query becomes too large. Other systems use similar fallbacks. Often these switching points imply “falling off a cliff”, i.e., we get good plans up to a certain point, and significantly worse results once queries get slightly larger, which is highly unsatisfying. The query optimizer should try to solve the problem exactly, and adaptively tune down the result quality if optimality can no longer be guaranteed.

frequency
normalized cost (log scale)
1 10 100 1000 10000 1e+05 1e+06 1e+07 1e+08

Figure 1: Normalized Cost Distribution of Random Plans for a Data-Warehouse-Style Query with 50 Relations

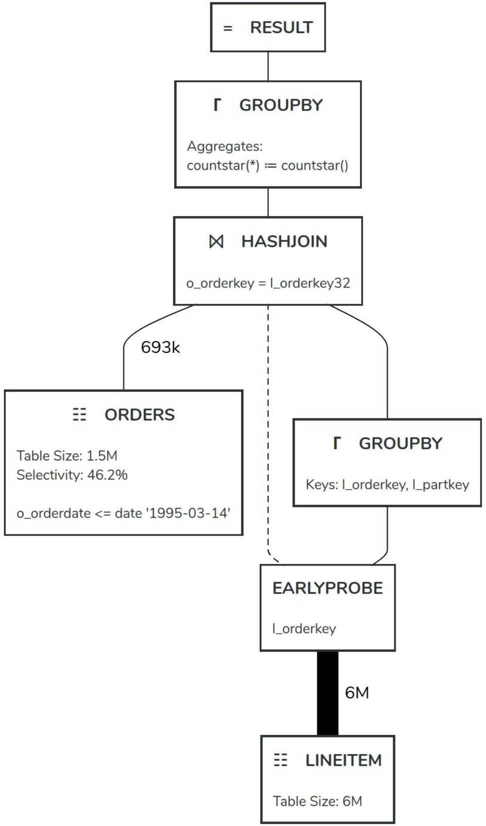
SIGMOD’18, June 10–15, 2018, Houston, TX, USA

Join Ordering

- Hash Joins rule
  - Indexes don’t allow bushy plans -> less useful
- Distinct count estimates with Pat Sellinger’s equations
- HyperLogLog intersections
- Mean & stddev approximations for `l_quantity < 0.2 * avg(l_quantity)`

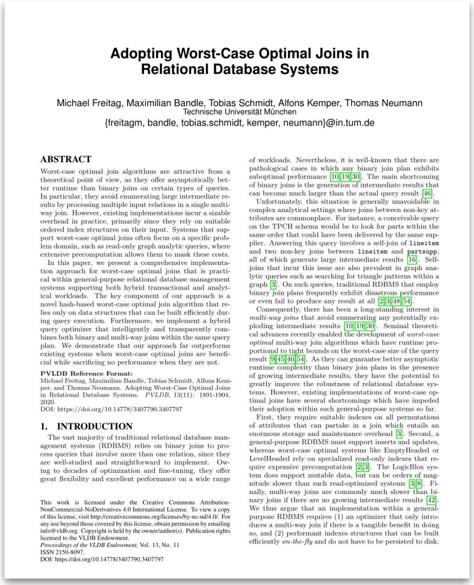
Early Probing

- Semijoin reduction
- Reuses existing hash tables
- Can use bloom filters if beneficial



Physical Optimization

- Indexes
- Worst-case optimal join



Adopting Worst-Case Optimal Joins in Relational Database Systems

Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, Thomas Neumann
Technische Universität München
{freitagm, bandle, tobias.schmidt, kemper, neumann}@in.tum.de

ABSTRACT

Worst-case optimal join algorithms are attractive from a theoretical point of view, as they offer asymptotically better runtime than binary joins on certain types of queries. In particular, they avoid enumerating large intermediate results by processing multiple input relations in a single multi-way join. However, existing implementations incur a sizable overhead in practice, primarily since they rely on suitable ordered index structures on their input. Systems that support worst-case optimal joins often focus on a specific problem domain, such as read-only graph analytic queries, where extensive precomputation allows them to mask these costs.

In this paper, we present a comprehensive implementation approach for worst-case optimal joins that is practical within general-purpose relational database management systems supporting both hybrid transactional and analytical workloads. The key component of our approach is a novel hash-based worst-case optimal join algorithm that relies only on data structures that can be built efficiently during query execution. Furthermore, we implement a hybrid query optimizer that intelligently and transparently combines both binary and multi-way joins within the same query plan. We demonstrate that our approach far outperforms existing systems when worst-case optimal joins are beneficial while sacrificing no performance when they are not.

PVLDB Reference Format:
Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting Worst-Case Optimal Joins in Relational Database Systems. *PVLDB*, 13(11): 1891-1904, 2020.
DOI: https://doi.org/10.14778/3407790.3407797

1. INTRODUCTION

The vast majority of traditional relational database management systems (RDBMS) relies on binary joins to process queries that involve more than one relation, since they are well-studied and straightforward to implement. Owing to decades of optimization and fine-tuning, they offer great flexibility and excellent performance on a wide range of workloads. Nevertheless, it is well-known that there are pathological cases in which any binary join plan exhibits suboptimal performance [10, 19, 30]. The main shortcoming of binary joins is the generation of intermediate results that can become much larger than the actual query result [46].

Unfortunately, this situation is generally unavoidable in complex analytical settings where joins between non-key attributes are commonplace. For instance, a conceivable query on the TPCH schema would be to look for parts within the same order that could have been delivered by the same supplier. Answering this query involves a self-join of `lineitem` and two non-key joins between `lineitem` and `partsupp`, all of which generate large intermediate results [16]. Self-joins that incur this issue are also prevalent in graph analytic queries such as searching for triangle patterns within a graph [3]. On such queries, traditional RDBMS that employ binary join plans frequently exhibit disastrous performance or even fail to produce any result at all [2, 3, 48, 54].

Consequently, there has been a long-standing interest in *multi-way joins* that avoid enumerating any potentially exploding intermediate results [10, 19, 30]. Seminal theoretical advances recently enabled the development of *worst-case optimal* multi-way join algorithms which have runtime proportional to tight bounds on the worst-case size of the query result [9, 45, 46, 54]. As they can guarantee better asymptotic runtime complexity than binary join plans in the presence of growing intermediate results, they have the potential to greatly improve the robustness of relational database systems. However, existing implementations of worst-case optimal joins have several shortcomings which have impeded their adoption within such general-purpose systems so far.

First, they require suitable indexes on all permutations of attributes that can partake in a join which entails an enormous storage and maintenance overhead [3]. Second, a general-purpose RDBMS must support inserts and updates, whereas worst-case optimal systems like EmptyHeaded or LevelHeaded rely on specialized read-only indexes that require expensive precomputation [2, 3]. The LogicBlox system does support mutable data, but can be orders of magnitude slower than such read-optimized systems [3, 8]. Finally, multi-way joins are commonly much slower than binary joins if there are no growing intermediate results [42]. We thus argue that an implementation within a general-purpose RDBMS requires (1) an optimizer that only introduces a multi-way join if there is a tangible benefit in doing so, and (2) performant indexes structures that can be built efficiently *on-the-fly* and do not have to be persisted to disk.

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407797

Physical Optimization

- Indexes
- Worst-case optimal join
- Groupjoin

Physical Optimization

- Indexes
- Worst-case optimal join
- Groupjoin
- Range join
- Join micro-optimizations
  - Multiset semantics
  - Allocation sizes

Adopting Worst-Case Optimal Joins in Relational Database Systems

Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, Thomas Neumann
Technische Universität München
{freitagm, bandle, tobias.schmidt, kemper, neumann}@in.tum.de

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

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

```
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
)
```

A Scalable and Generic Approach to Range Joins

Maximilian Reif
Technical University of Munich
reif@in.tum.de

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

Figure 1: Flight routing with stop-over

```
select *,
from flights f1, flights f2
where
f1.orig = 'MUC' and f2.dest = 'SYD' and
f1.dest = f2.orig and
f2.takeoff between
   f1.landing + '45 minutes' and
   f1.landing + '3 hours'
order by f1.price + f2.price limit 10
```

Recap

- Query compilation & optimization
  - Optimizer passes
  - Rule-based canonicalization
  - Cost-based optimization
- Cutting-edge research
  - Join ordering
  - Cardinality estimation
  - Integrated in a running system

1: Unoptimized Plan
2: Expression Simplification
3: Unnesting
4: Predicate Pushdown
5: Initial Join Tree
6: Sideway Information Passing
7: Operator Reordering
8: Early Probing
9: Common Subtree Elimination
10: Physical Operator Mapping

Conclusion

- Low latency analytical queries
- Also works excellent for transactional and graph workloads

We are commercializing

Reach out:

fent@in.tum.de

TUM Open Source Project

LingoDB