

# CedarDB

Philipp Fent  
philipp@cedar db.com



# Overview



- TUM Startup
  - Started at TUM with Umbra
  - Cutting-edge database research
  - Query compilation
  - Disk-based with in-memory performance

## CSRankings: Computer Science Rankings

CSRankings is a metrics-based ranking of top computer science institutions around the world. **Click on a triangle (▶)** to expand areas or institutions. **Click on a name** to go to a faculty member's home page. **Click on a chart icon** (the ) after a name or institution) to see the distribution of their publication areas as a **bar chart** (the ). **Click on a Google Scholar icon** (the ) to see publications, and **click on the DBLP logo** (the ) to go to a DBLP entry. *Applying to grad school? Read this first. Do you find CSRankings useful? Sponsor CSRankings on GitHub.*

Rank institutions in  by publications from  to

### All Areas [\[off | on\]](#)

#### AI [\[off | on\]](#)

- ▶ Artificial intelligence
- ▶ Computer vision
- ▶ Machine learning
- ▶ Natural language processing
- ▶ The Web & information retrieval

#### Systems [\[off | on\]](#)

- ▶ Computer architecture
- ▶ Computer networks
- ▶ Computer security
- ▶ Databases
- ▶ Design automation
- ▶ Embedded & real-time systems

### # Institution

1	▶ TU Munich 
2	▶ HKUST 
3	▶ Tsinghua University 
4	▶ University of Waterloo 
5	▶ National University of Singapore 
6	▶ Duke University 
7	▶ Chinese University of Hong Kong 
8	▶ Nanyang Technological University 
9	▶ Univ. of California - San Diego 
10	▶ Univ. of California - Berkeley 
11	▶ Peking University 

# Overview



- TUM Startup

- Started at TUM with Umbra
- Cutting-edge database research
- Query compilation
- Disk-based with in-memory performance

- “PostgreSQL for analytics”

- PostgreSQL protocol and client compatibility
- Simultaneous high-performance analytics and operations on the same data
- Full utilization of modern hardware capabilities (e.g. massive parallelism, RAM capacity)
- Transparently and gracefully scales **beyond main memory**
- **Several orders of magnitude** speedup over existing systems

## CSRankings: Computer Science Rankings

CSRankings is a metrics-based ranking of top computer science institutions around the world. [Click on a triangle \(▶\)](#) to expand areas or institutions. [Click on a name](#) to go to a faculty member's home page. [Click on a chart icon](#) (the ) after a name or institution) to see the distribution of their publication areas as a [bar chart](#) (▼). [Click on a Google Scholar icon](#) (🔍) to see publications, and [click on the DBLP logo](#) (📄) to go to a DBLP entry. *Applying to grad school? Read this first. Do you find CSRankings useful? Sponsor CSRankings on GitHub.*

Rank institutions in  by publications from  to

### All Areas [\[off | on\]](#)

#### AI [\[off | on\]](#)

- ▶ Artificial intelligence
- ▶ Computer vision
- ▶ Machine learning
- ▶ Natural language processing
- ▶ The Web & information retrieval

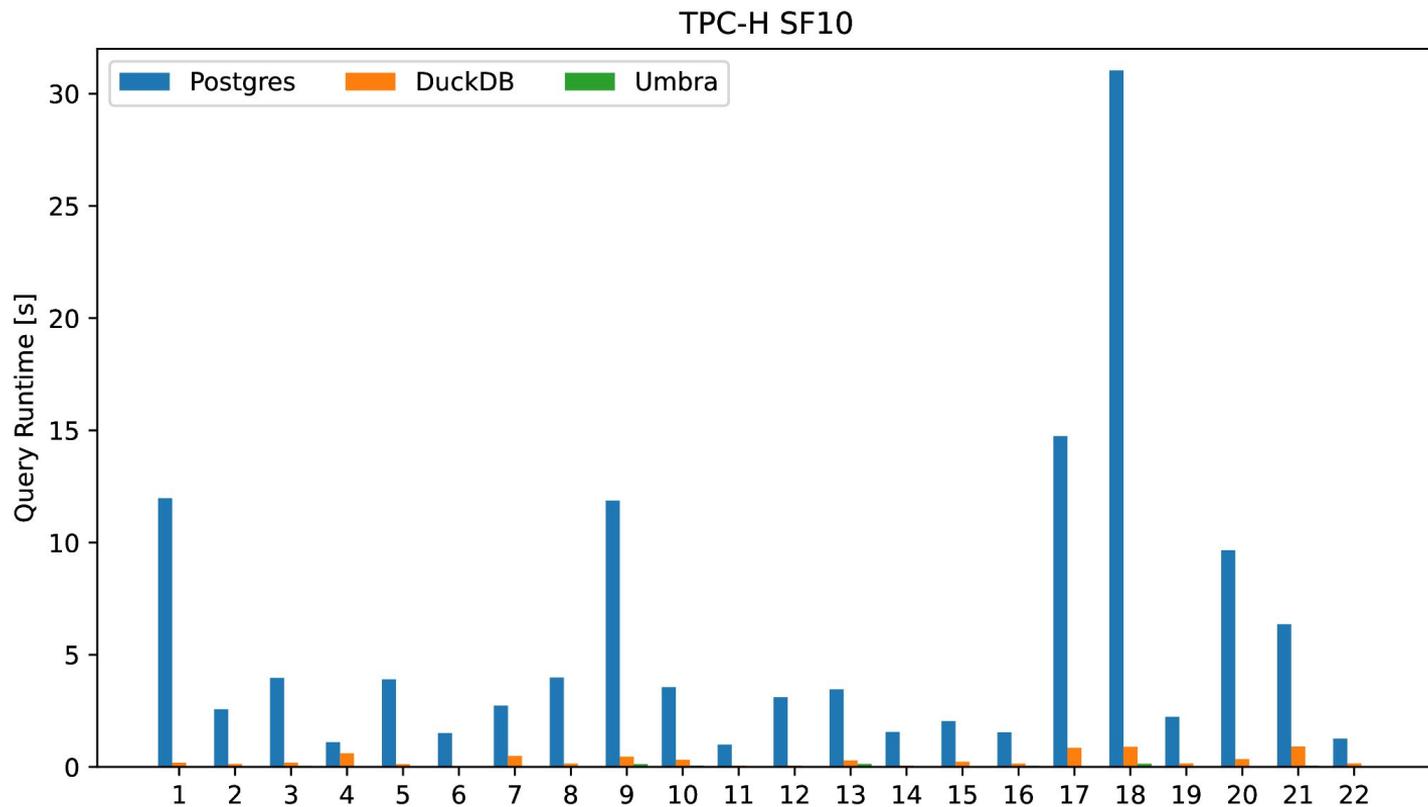
#### Systems [\[off | on\]](#)

- ▶ Computer architecture
- ▶ Computer networks
- ▶ Computer security
- ▶ Databases
- ▶ Design automation
- ▶ Embedded & real-time systems

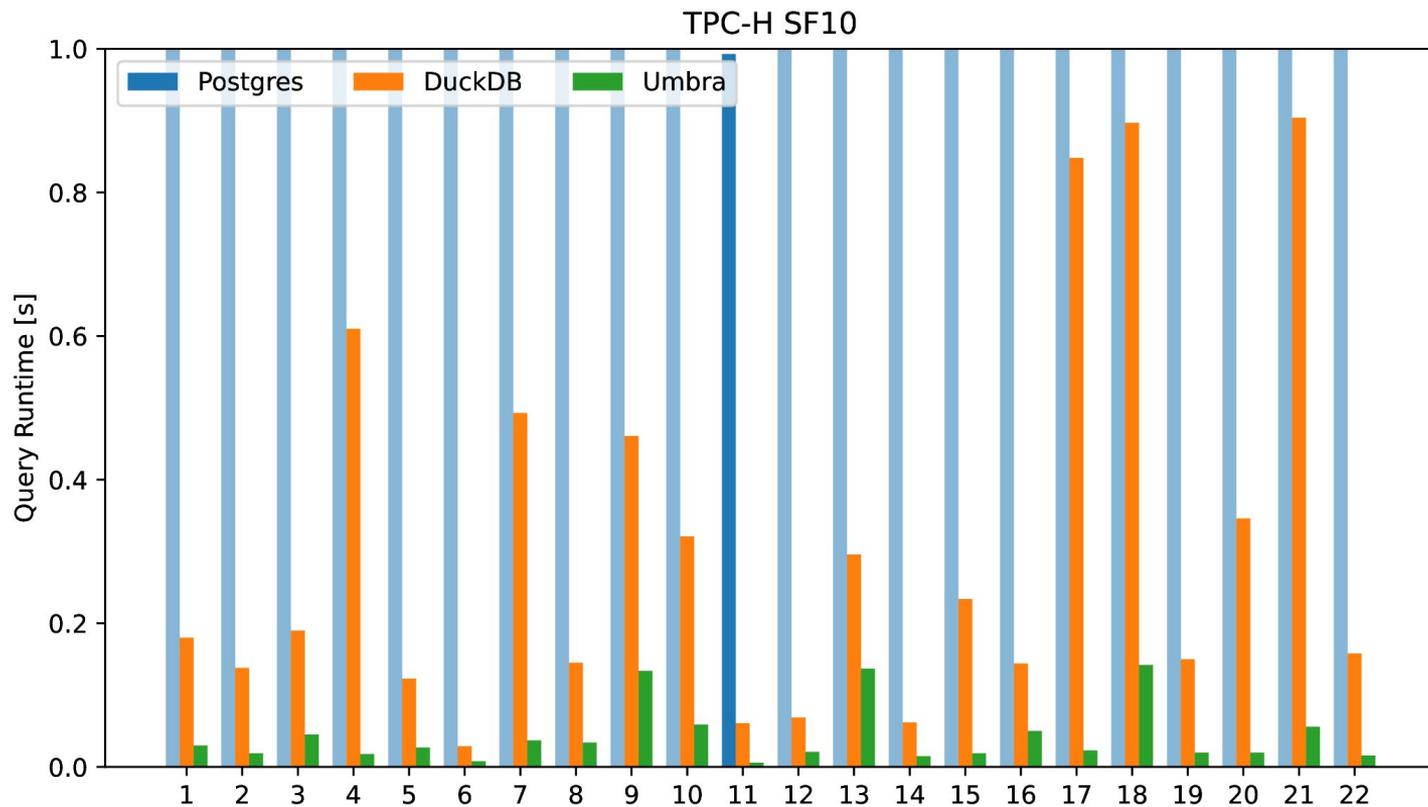
### # Institution

1	▶ TU Munich 
2	▶ HKUST 
3	▶ Tsinghua University 
4	▶ University of Waterloo 
5	▶ National University of Singapore 
6	▶ Duke University 
7	▶ Chinese University of Hong Kong 
8	▶ Nanyang Technological University 
9	▶ Univ. of California - San Diego 
10	▶ Univ. of California - Berkeley 
11	▶ Peking University 

# Performance



# Performance



# Overview

- “PostgreSQL for analytics”
  - PostgreSQL protocol and client compatibility
  - Simultaneous high-performance analytics and operations on the same data
  - Full utilization of modern hardware capabilities (e.g. massive parallelism, RAM capacity)
  - Transparently and gracefully scales **beyond main memory**
  - **Several orders of magnitude** speedup over existing systems
  
- Free Community Edition for Linux / Docker:  
`curl https://get.cedardb.com | bash`



# Agenda

- Recap: DBMS Components
- Relational Algebra Optimization
- Storage: B-Tree deep dive

Overview over cutting-edge database research  
 Research papers referenced like this ->

## UmbrA: A Disk-Based System with In-Memory Performance

Thomas Neumann, Michael Freitag  
 Technische Universität München  
 {neumann, freitagm}@in.tum.de

### ABSTRACT

The increases in main-memory sizes over the last decade have made pure in-memory database systems feasible, and in-memory systems offer unprecedented performance. However, DRAM is still relatively expensive, and the growth of main-memory sizes has slowed down. In contrast, the prices for SSDs have fallen substantially in the last years, and their read bandwidth has increased to gigabytes per second. This makes it attractive to combine a large in-memory buffer with fast SSDs as storage devices, combining the excellent performance for the in-memory working set with the scalability of a disk-based system.

In this paper we present the UmbrA system, an evolution of the pure in-memory HyPer system towards a disk-based, or rather SSD-based, system. We show that by introducing a novel low-overhead buffer manager with variable-size pages we can achieve comparable performance to an in-memory database system for the cached working set, while handling accesses to uncached data gracefully. We discuss the changes and techniques that were necessary to handle the out-of-memory case gracefully and with low overhead, offering insights into the design of a memory optimized disk-based system.

### 1. INTRODUCTION

Hardware trends have greatly affected the development and evolution of database management systems over time. Historically, most of the data was stored on (rotating) disks, and only small fractions of the data could be kept in RAM in a buffer pool. As main memory sizes grew significantly, up to terabytes of RAM, this perspective changed as large fractions of the data or even all data could now be kept in memory. In comparison to disk-based systems, this offered a huge performance advantage and led to the development of pure in-memory database systems [4, 2] including our own system HyPer [9]. These systems make use of RAM-only storage and offer outstanding performance, but tend to fail or degrade heavily if the data does not fit into memory.

Moreover, we currently observe two hardware trends that cast strong doubt on the viability of pure in-memory systems. First, RAM sizes are not increasing significantly any more. Ten years

ago, one could conceivably buy a commodity server with 1 TB of memory for a reasonable price. Today, affordable main memory sizes might have increased to 2 TB, but going beyond that disproportionately increases the costs. As costs usually have to be kept under control though, this has caused the growth of main memory sizes in servers to subside in the recent years.

On the other hand, SSDs have achieved astonishing improvements over the past years. A modern 2 TB M.2 SSD can read with about 3.5 GB/s, while costing only \$500. In comparison, 2 TB of server DRAM costs about \$20 000, i.e. a factor of 40 more. By placing multiple SSDs into one machine we can get excellent read bandwidths at a fraction of the cost of a pure DRAM solution. Because of this, Lomet argues that pure in-memory DBMSs are uneconomical [15]. They offer the best possible performance, of course, but they do not scale beyond a certain size and are far too expensive for most use cases. Combining large main memory buffers with fast SSDs, in contrast, is an attractive alternative as the cost is much lower and performance can be nearly as good.

We wholeheartedly agree with this notion, and present our novel UmbrA system which simultaneously features the best of both worlds: Genuine in-memory performance on the cached working set, and transparent scaling beyond main memory where required. UmbrA is the spiritual successor of our pure in-memory system HyPer, and completely eliminates the restrictions of HyPer on data sizes. As we will show in this paper, we achieve this without sacrificing any performance in the process. UmbrA is a fully functional general-purpose DBMS that is actively developed further by our group. All techniques presented in this paper have been implemented and evaluated within this working system. While UmbrA and HyPer share several design choices like a compiling query execution engine, UmbrA deviates in many important aspects due to the necessities of external memory usage. In the following, we present key components of the system and highlight the changes that were necessary to support arbitrary data sizes without losing performance for the common case that the entire working set fits into main memory.

A key ingredient for achieving this is a novel buffer manager that combines low-overhead buffering with *variable-size pages*. Compared to a traditional disk-based system, in-memory systems have the major advantage that they can do away with buffering, which both eliminates overhead and greatly simplifies the code. For disk-based systems, common wisdom dictates to use a buffer manager with *fixed-size pages*. However, while this simplifies the buffer manager itself, it makes using the buffer manager exceedingly difficult. For example, large strings or lookup tables for dictionary compression often cannot easily be stored in a single fixed-size page, and both complex and expensive mechanisms are thus required also over the database system in order to handle large objects. We ar-

This article is published under a Creative Commons Attribution License (<http://creativecommons.org/licenses/by/2.0/>), which permits distribution and reproduction in any medium as well as allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2020, 10th Annual Conference on Innovative Data Systems Research (CIDR 2020, January 12-15, 2020, Amsterdam, Netherlands).



CedarDB

# Agenda

- **Recap: DBMS Components**
- Relational Algebra Optimization
- Storage: B-Tree deep dive

# Recap: DBMS Components

- SQL parsing
- Relational Algebra Plan
- Physical Execution Plan
- Storage Access



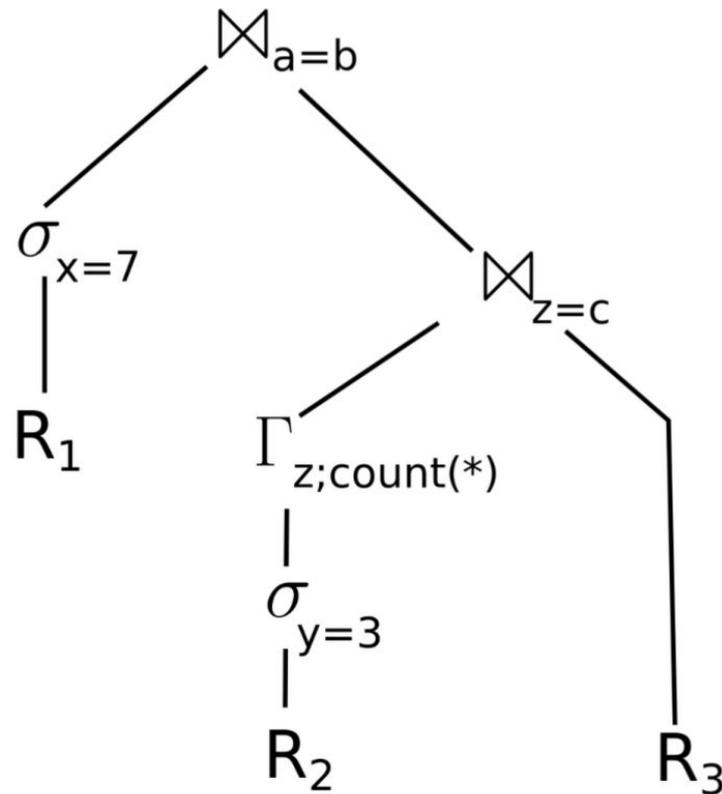
# SQL Parsing

```
SELECT *  
FROM R1, R3, (  
    SELECT R2.z, count(*)  
    FROM R2  
    WHERE R2.y = 3  
    GROUP BY R2.z  
) R2  
WHERE R1.x = 7  
AND R1.a = R3.b  
AND R2.z = R3.c
```

# SQL Parsing

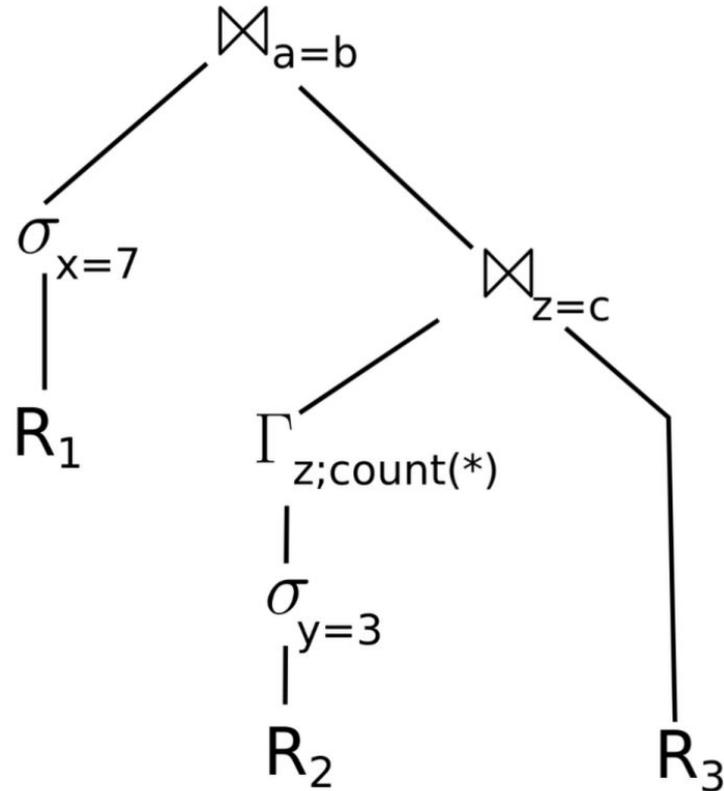
```

SELECT *
FROM R1, R3, (
    SELECT R2.z, count(*)
    FROM R2
    WHERE R2.y = 3
    GROUP BY R2.z
) R2
WHERE R1.x = 7
AND R1.a = R3.b
AND R2.z = R3.c
    
```



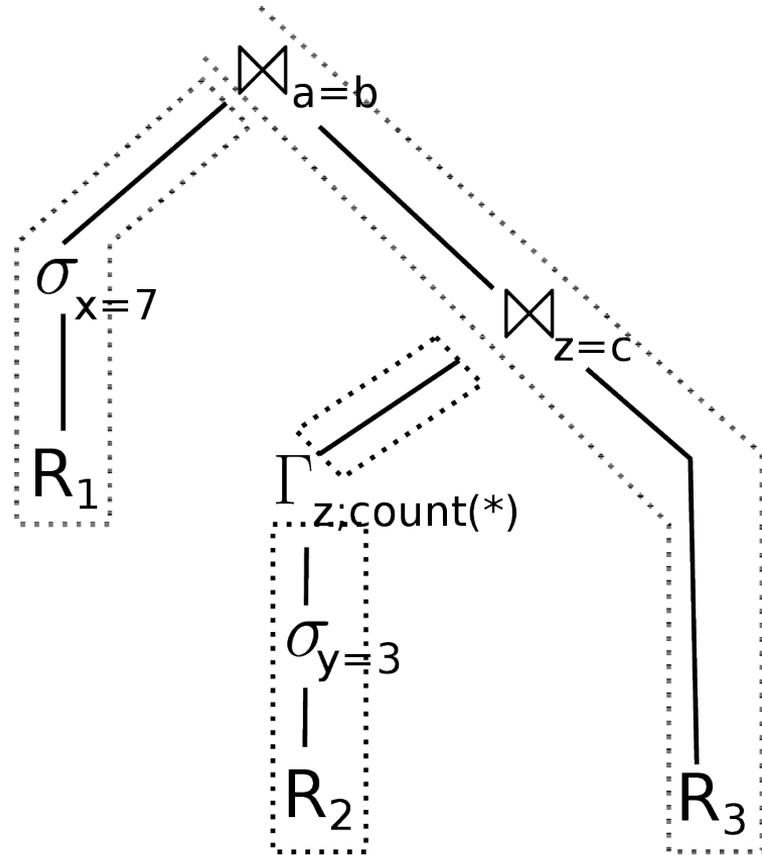
# Relational Algebra Plan

- Set-Oriented Query Processing
- Allows abstract optimization
- Crucial for efficient execution



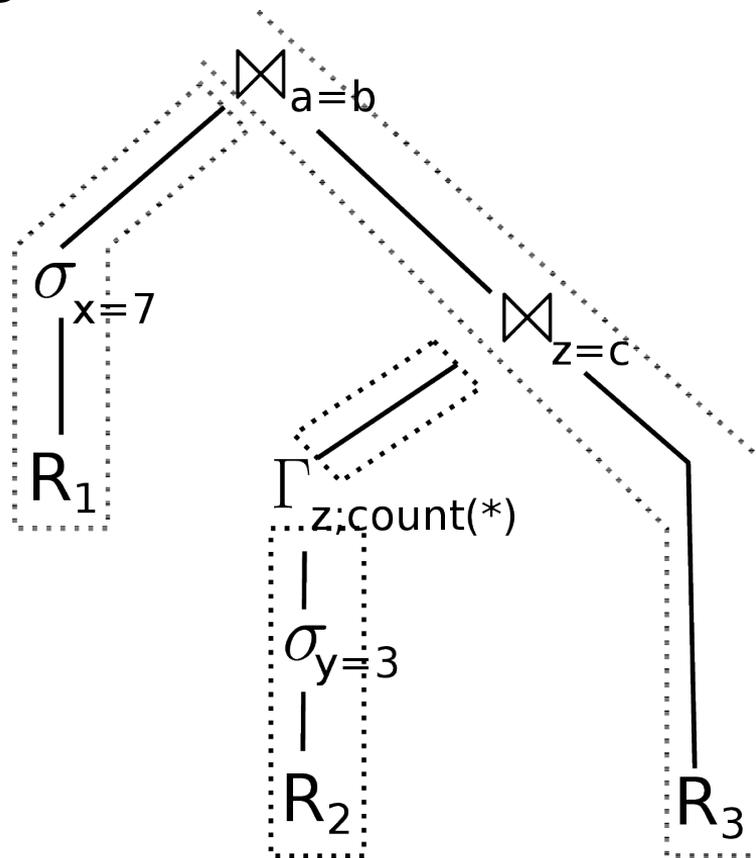
# Physical Execution Plan

- Physical access paths
  - Index or table scan



# Physical Execution in CedarDB

- Pipelined execution
  - Keeps values in registers
  - Minimizes materialization



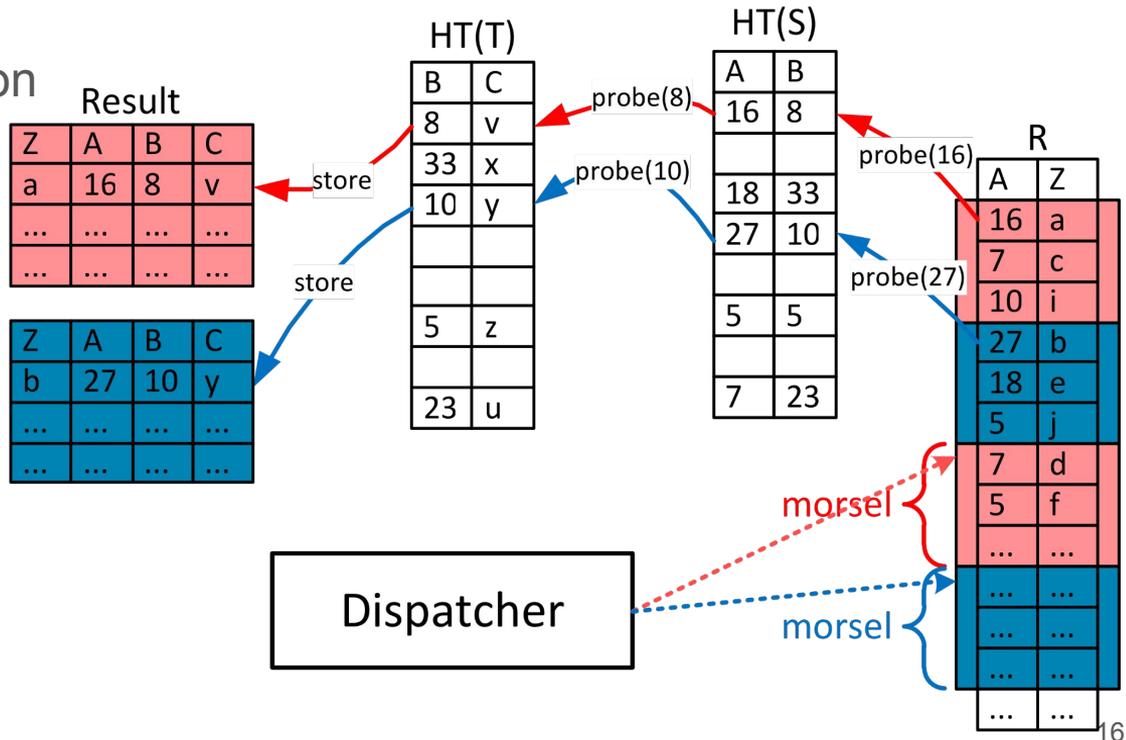
# Physical Execution in CedarDB

- 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
```

# Physical Execution in CedarDB

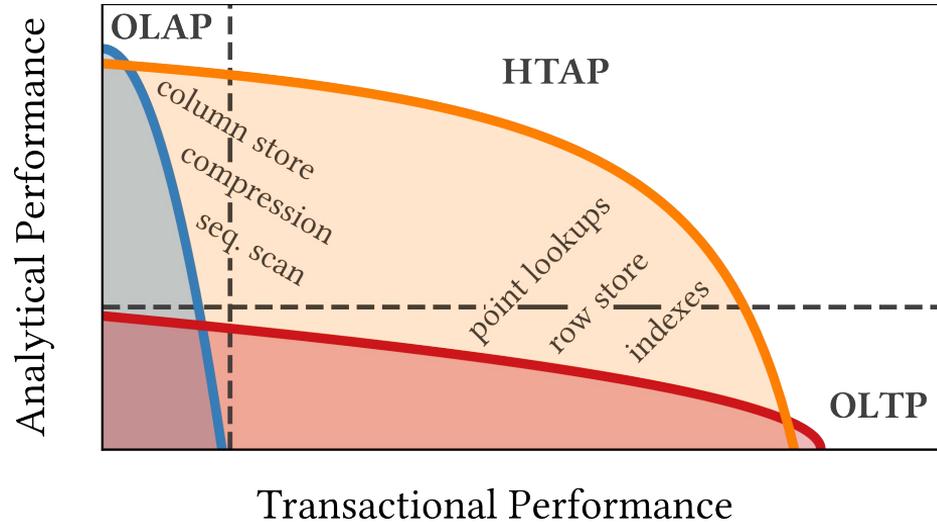
- Pipelined execution
- Data-centric code generation
- Fully parallel algorithms
  - Allows scaling
  - Benefits from new hardware



# Storage Access

- Storage on disk
- Row vs. column stores
- Hybrid storage for transactions and analytics
  - Fast scans
  - Fast point lookups
  - Fast writes
  - Index structures

➔ B-Trees





CedarDB

# Agenda

- Recap: DBMS Components
- **Relational Algebra Optimization**
- Storage: B-Tree deep dive



CedarDB

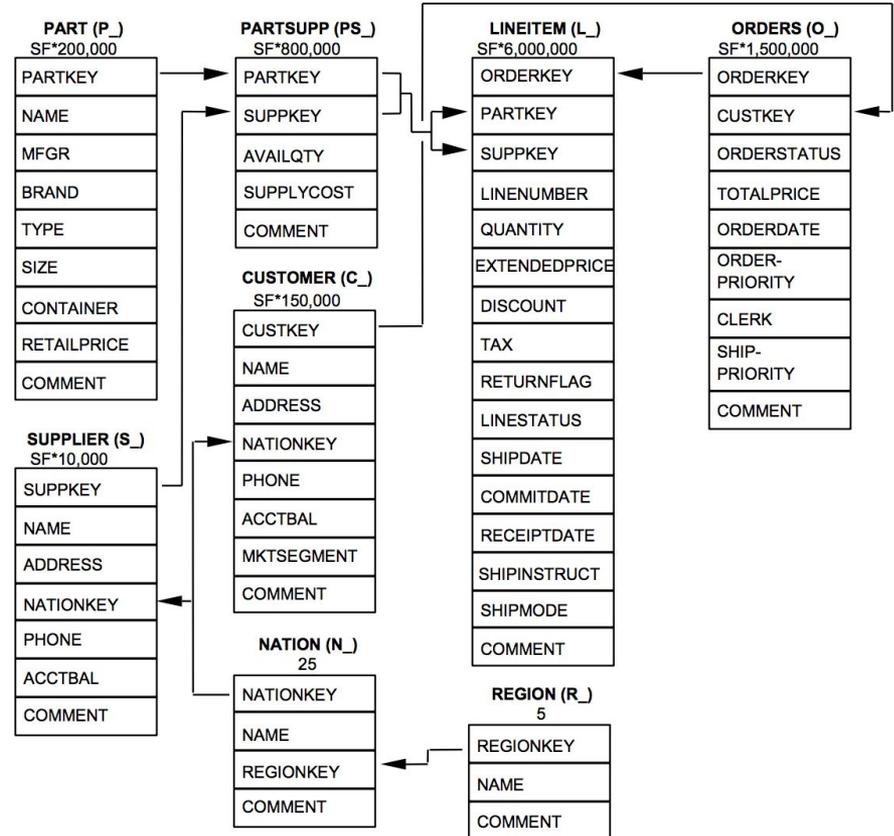
# Query Optimization

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

# Running Example: TPC-H Q17

- How much average yearly revenue would be lost if orders were no longer filled for small quantities of certain parts?  
This may reduce overhead expenses by concentrating sales on larger shipments.

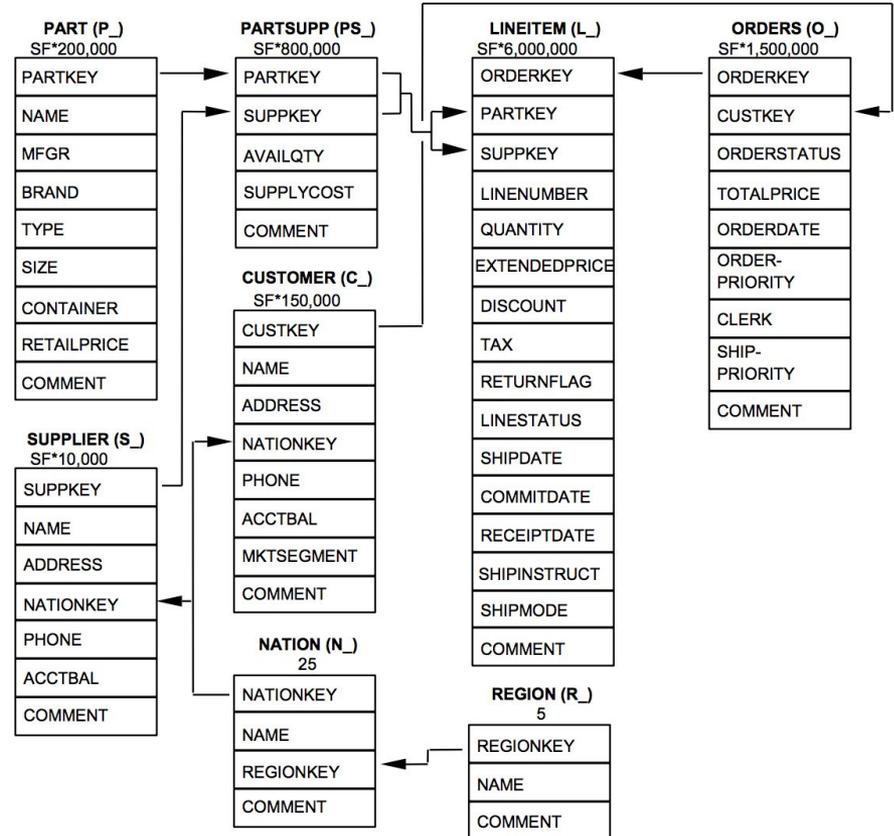
Figure 2: The TPC-H Schema



# Running Example: TPC-H Q17

```
-- TPC-H Query 17
select sum(l_extendedprice)
       / 7.0 as avg_yearly
from lineitem, part
where p_partkey = l_partkey
and p_brand = 'Brand#23'
and p_container = 'MED BOX'
and l_quantity < (
  select 0.2 * avg(l_quantity)
  from lineitem
  where l_partkey = p_partkey
)
```

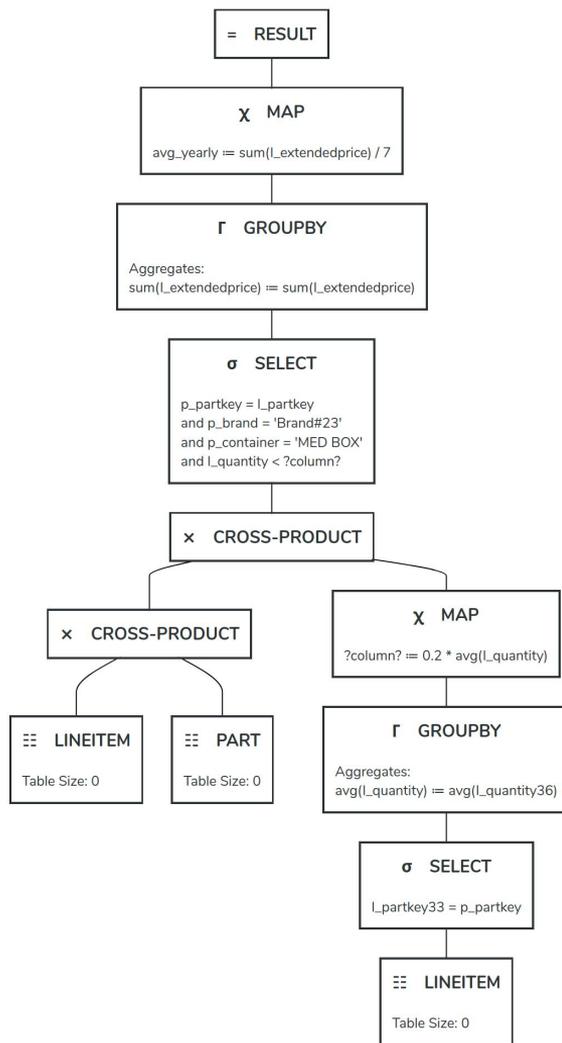
Figure 2: The TPC-H Schema





# 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

Cost-based  
Optimization

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

Rule-based  
Canonicalization



# 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

## 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.

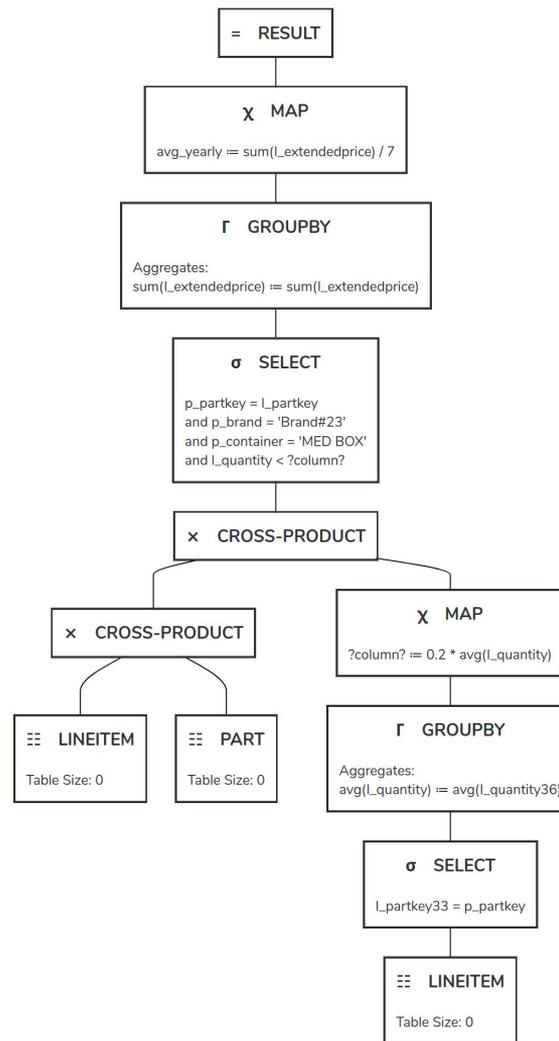
- Scalar subqueries
- EXISTS
- IN / ANY / ALL

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

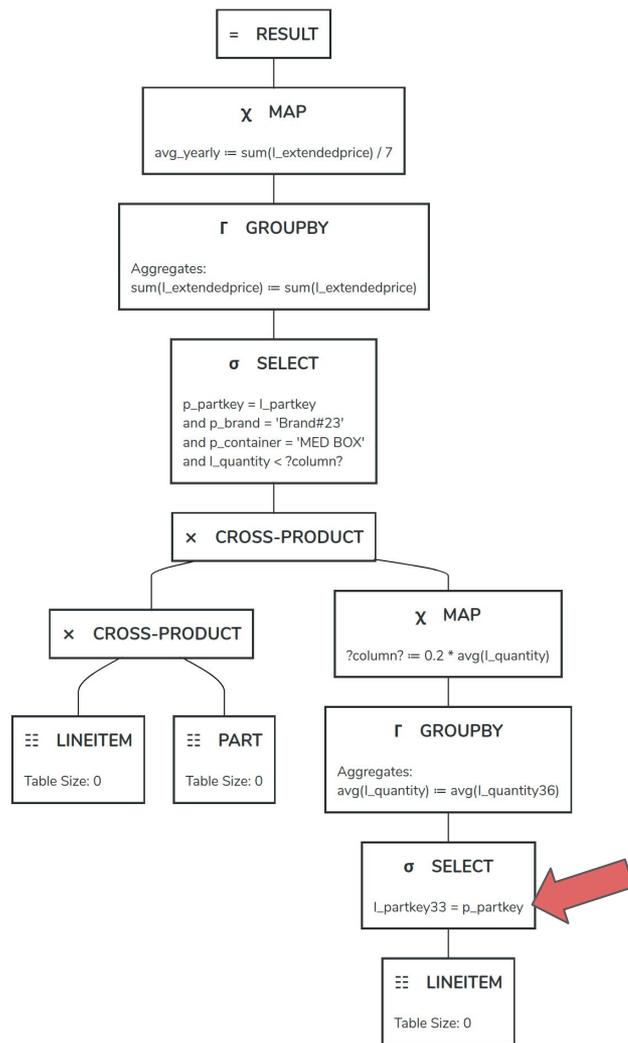
- Unnesting Arbitrary Queries
  - $O(n^2)$





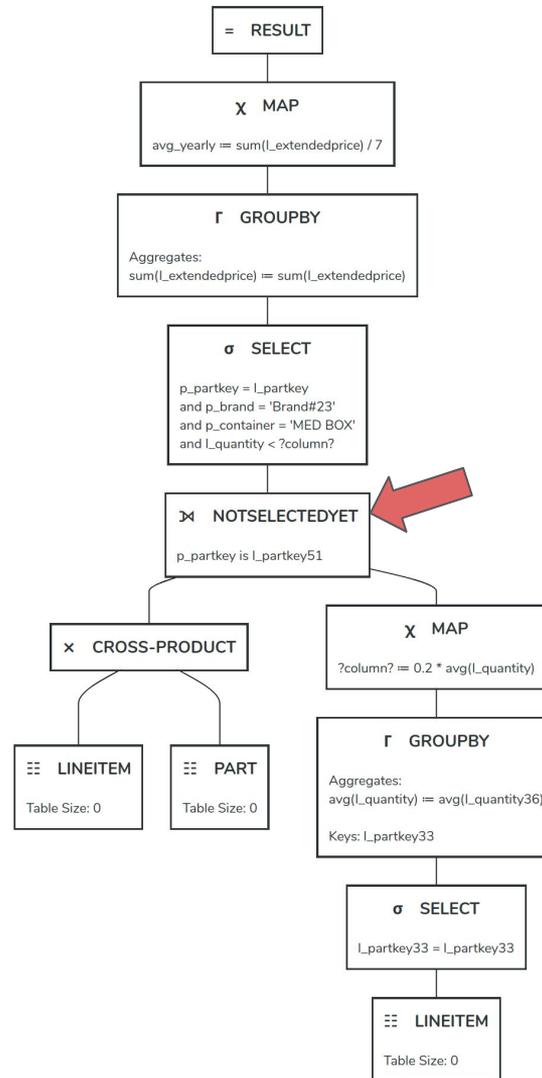
# Query Unnesting

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



# Query Unnesting

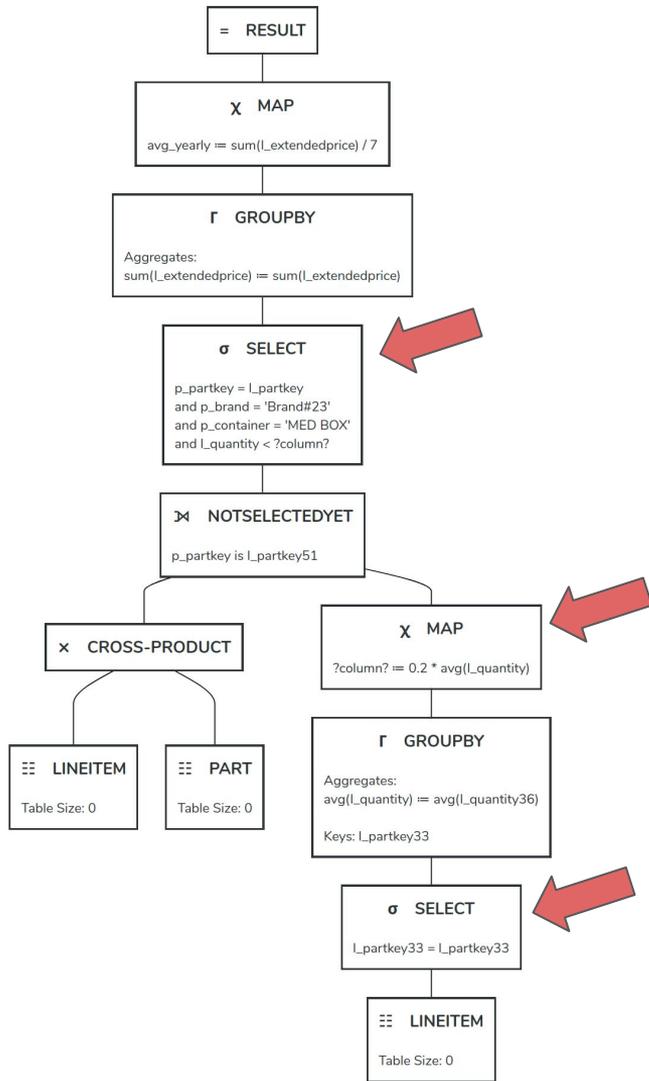
- Unnesting Arbitrary Queries
  - $O(n^2) \rightarrow O(n)$
  - Huge improvement





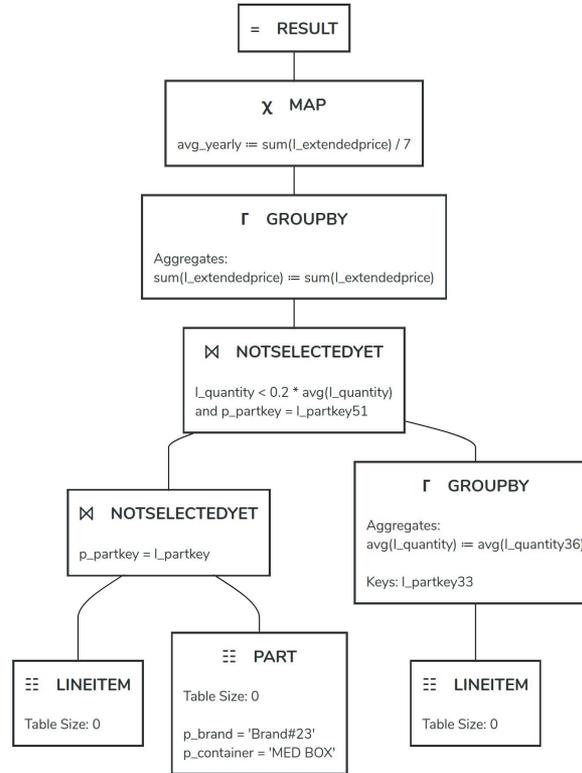
# Predicate Pushdown

- Place predicates at scan
- Propagate & fold constants



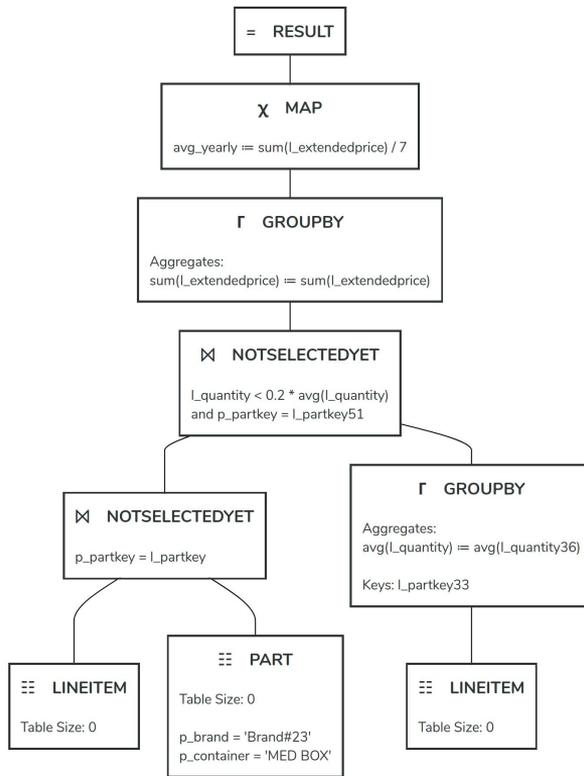
# Predicate Pushdown

- Place predicates at scan
- Propagate & fold constants



# Predicate Pushdown

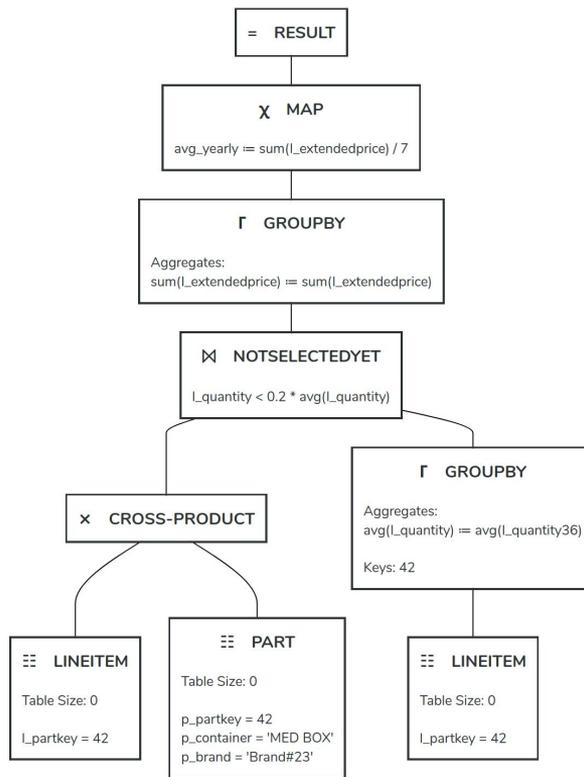
- Place predicates at scan
- Propagate & fold constants



+ 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
```



CedarDB

# 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  $\sigma$  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	<b>12.7 ms</b>



# 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
```



CedarDB

# Join Ordering

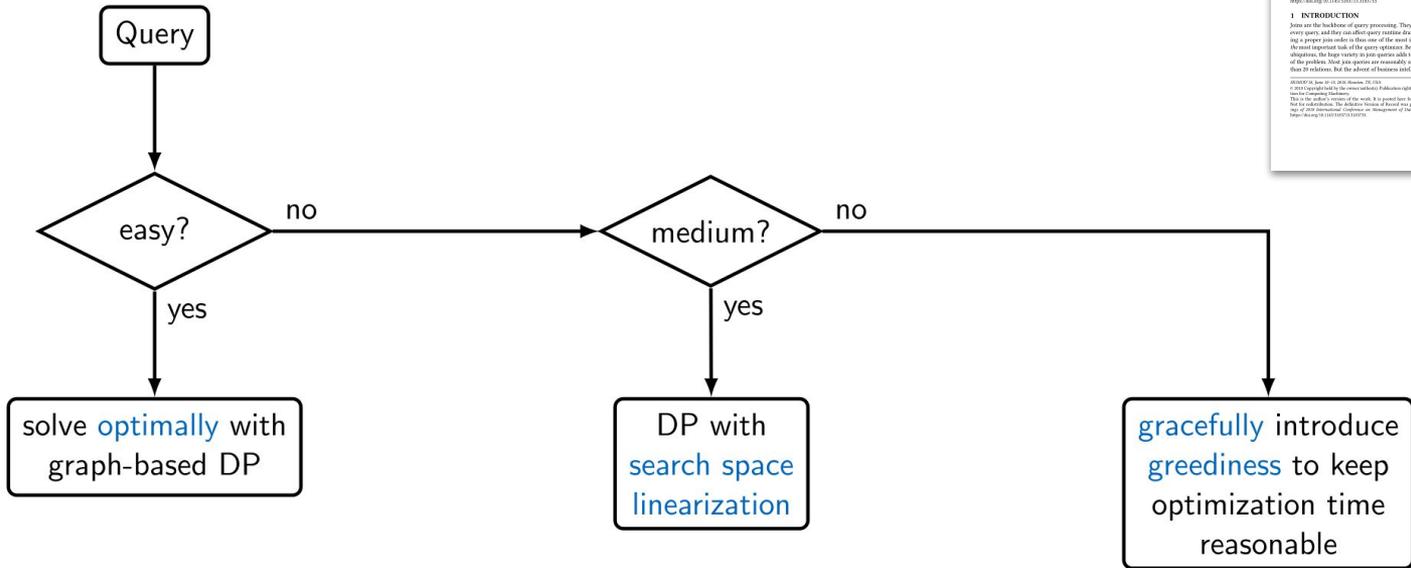
- Mostly Hash Joins
  - Indexes don't allow bushy plans -> less useful



# CedarDB

## Join Ordering

- Mostly Hash Joins



**Adaptive Optimization of Very Large Join Queries**

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

Berndrad 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 success in the area of join queries. While most systems are reasonably good, join queries with up to a hundred relations are still the main concern, since the distribution of query trees has an exponential long tail. In this paper, we study queries that we are aware of across more than 4000 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 here. This paper introduces an adaptive optimization framework that is able to solve most common join queries exactly, while simultaneously solving up queries with thousands of joins. A key component there is a novel search space linearization technique that leads to more optimized execution plans for large classes of queries. Finally, too, we describe implementation techniques that are necessary to make join ordering algorithms in these extremely large queries.

Business intelligence systems often deal with large data sets that the join algebra approach proposed here performs excellent over a large spectrum of query sizes, and provides optimal or near-optimal solutions for most common queries.

**CCS CONCEPTS**

Information systems → Query optimization.

**ACM Reference Format:**

Thomas Neumann and Berndrad Radke. 2018. Adaptive Optimization of Very Large Join Queries. In Proceedings of 2018 International Conference on Management of Data, Munich, Germany, June 18–21, 2018. ACM, New York, NY, USA, 14 pages.

<https://doi.org/10.1145/3205330.3205373>

**1 INTRODUCTION**

Join is the backbone of query processing. This task is usually every query, and this can often occur *repeatedly*. Clearly, it is a very important task. In the area of join ordering, it is the most important task of the query optimizer besides joins being collapsed. We have recently in our queries with the complexity of the problem, but join queries are reasonably small, joining just three or relations. But the advent of business intelligence tools has

**Figure 1: Normalized Cost Distribution of Random Plans for a Data Warehouse Query with 16 Relations.**

led to generally all of the queries that can easily touch a hundred relations, and a database system must be able to handle these. In fact, even moderately sized queries with, say, 30 relations can be handled what can be optimized exactly. In such cases, optimizers have to search optimally and may be limited to finding a long execution plan. Figure 1 shows the distribution of the costs resulting from the best plan of 100,000 random plans for a data warehouse table query with 16 relations. The cost of most plans are at least 10% higher than the cheapest plan found. At that scale, a heuristic is hard to be certain in that area of the cost space.

This however is by far not the end of the spectrum. There is an amazingly long tail of queries. The largest random plans that we are aware of include, after view expansion, 4700 relations [15]. Additionally, these single queries [1] are orders even in the I/O cost (the one largest has 3200 relations and only a handful have over 1000), but queries with several hundred relations are not that uncommon in the wild. The public query log from the Tableau Public data visualization tool shows a similar distribution, with most queries being small, but long tail of queries with more than 100 relations resulting up to 500 in single join queries. Note that especially in such long-tail scenarios, the workload can become, up front and most queries are small only ones. Other techniques to improve query performance or reduce their complexity (e.g., materialization, over time) are not always be desirable or applicable.

Large tables queries have a search space that is not even close to be do-able. For example, one dynamic programming to handle arbitrary join queries with 100 relations and multiple operators, algorithms for large queries [10] are designed to handle arbitrary queries, but they are generally strategy when the query programming and the flow is generally strategy when the query programming is large. Other systems do not handle this. One interesting possibility is “adding off a diff”, i.e., we get good plans up to a certain point, and significantly worse results one query get slightly larger, which is highly motivating. The query optimizer should try to make the problem smaller, and adaptive time-based result quality if optimality can no longer be guaranteed.



# Join Ordering

- Mostly Hash Joins
- Distinct count estimates with Pat Selinger's equations:  
column1 = column2  
$$F = 1 / \text{MAX}(\text{ICARD}(\text{column1}), \text{ICARD}(\text{column2}))$$
- HyperLogLog intersections
- Mean & stddev approximations for `l_quantity < 0.2 * avg(l_quantity)`



# 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/10.1145/3448616.3452895>

### 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 [14], 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 Vogelsang et al. [45] underlines the importance of filter operations and reveals: Most data is stored in string format, which enables arbitrary complex expressions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).  
SIGMOD '21, June 18–27, 2021, Virtual Event, China  
ACM ISBN 978-1-4503-8343-1/21/06...\$15.00  
<https://doi.org/10.1145/3448616.3452895>

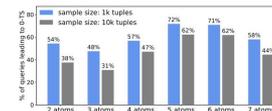


Figure 1: Relative number of queries over tables with at least 1M tuples that lead to empty samples (0-TS) with regard to the number of filter predicates (atoms) and the sample size.

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



# 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 Reif, Tobias Schmidt, Alfons Kemper, Thomas Neumann  
Technische Universität München  
(reif@tgm, 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 unstable ordered index structures on their input. Systems that support worst-case optimal joins often focus on a specific problem domain, such as real-time graph analytic queries, where expensive preprocessing allows them to mark time 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 hybrid hybrid transactional and analytical workload. 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

of workloads. Nevertheless, it is well-known pathological cases in which our binary join outperforms performance [10,11]. The use of binary joins is the foundation of interest can become much larger than the actual cost. Unfortunately, this situation is generally complex analytical setups where joins both relations are commensurate. For instance, on the TPCH schema would be to look for some order that could have been derived in all of which generate large intermediate join results that incur this issue we also provide a hybrid query optimizer for existing hybrid join join plus frequently exhibit distinct or even fail to produce any result at all [12]. Consequently, there has been a long-standing desire to avoid enumerating and plotting intermediate results [10,11].

### ABSTRACT

Analytical database systems provide great insights into large datasets and are an excellent tool for data exploration and analysis. A central pillar of query processing is the efficient evaluation of equi-joins, typically with linear-time algorithms (e.g. hash join). However, for many use-cases with location and temporal data, non-equi-joins like range joins, occur in queries. Without optimizations, this typically results in nested loop evaluation with quadratic complexity. This leads to unacceptable query execution times. Different mitigations have been proposed in the past, like partitioning or sorting the data. While these allow for handling certain classes of queries they tend to be restricted in the kind of queries they can support. And, perhaps even more importantly, they do not play nicely with additional equality predicates that typically occur within a query and that have to be considered, too.

In this work, we present a hybrid hash-based, multi-dimension range join that supports a very wide range of queries, and that can exploit additional equality constraints. This approach allows us to handle range classes of queries very efficiently, with negligible memory overhead, and as is suitable as a general-purpose solution for range queries in database systems. The join algorithm is fully parallel, both during the build and the probe phase, and scales to large problem instances and high core counts.

We demonstrate the feasibility of this approach by integrating it into our database system Umbra and performing extensive experiments with both large real-world data sets and with synthetic benchmarks used for sensitivity analysis. In our experiments, it outperforms hand-tuned Spark code and all other database systems that we have tested.

**PVLDB Reference Format:**  
Maximilian Reif and Thomas Neumann. A Scalable and Generic Approach to Range Joins. PVLDB, 15(11):3018–3031, 2022.  
doi:10.14758/pvlldb.v15n11.355149

**PVLDB Artifact Availability:**  
The source code, data, and/or other artifacts have been made available at <https://github.com/in.tum.de/research-projects/reproducible>.

**1 INTRODUCTION**

Over the last years, we observed two major trends in data processing. The amount of data collected is vastly growing, and data analysis techniques are becoming more and more refined. Database systems provide an excellent base for managing these very large data sets.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any reuse or distribution, you must bear the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. Published by the VLDB Endowment. Proceedings of the VLDB Endowment, Vol. 15, No. 11, 2022. © 2022. doi:10.14758/pvlldb.v15n11.355149

## A Practical Approach to Groupjoin and Nested Aggregates

Philipp Feat  
Technische Universität München  
feat@tum.de

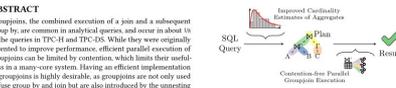


Figure 1: Missing components for practical groupjoin. Our improvements to estimation and parallel execution enable efficient evaluation of queries with nested aggregates.

The primary reasons 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, groupjoin are also more complex than regular groupjoins, 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(sales) AS s
  FROM sales s
  WHERE cust_id = s.cust_id
) s;
```

Here, nested 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 groupby will produce wrong results for empty subgroups, which is known as the COUNT bug [14]. A groupjoin directly supports such queries by evaluating the static aggregate for the nested side of the join, taking the group from the other side.

Despite these benefits, groupjoins are not widely in use. We identify two reasons and propose solutions that make groupjoins more practical. First, existing algorithms for groupjoins do not allow for parallel execution. Since the groupjoin hash table contains shared aggregate states, parallel updates of these need synchronization, and one can cause heavy memory contention. Furthermore, current estimation techniques deal poorly with results of groupjoins from nested aggregates.

The unsetting of inner aggregation subqueries is very problematic, as it eliminates nested-loop evaluation and improves the asymptotic complexity of the query. However, this causes the aggregates to be part of a bigger query tree, tangled between joins, predicates and other relational operators. Query optimization, specifically join ordering, depends on the quality of cardinality

```
select
  from Flights f1, Flights f2
  where
  f1.orig = f2.dest 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
```

In this case, the join has two join conditions. The equivalence predicate  $f1.dest = f2.orig$  and the range predicate  $f2.takeoff$  between  $f1.landing + 45$  minutes and  $f1.landing + 3$  hours. Thus, the join could be considered an equi-join with a range residual or a range-join with an additional equivalence predicate. Other examples for range-joins are: The matching of vehicle sensor data to vehicle sides (defined by a time frame) or the mapping of IP addresses to subnets [17]. Moreover, there are applications, which require the evaluation of multiple range predicates, so-called multi-dimensional range joins. Examples are: Finding return trips in taxi ride data (Destinations A, B) or combining bid sightings and weather reports [21] based on location and time data. Additional equivalence predicates, as in the flight example, are also very common and should be incorporated into a range-join algorithm.

# 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



CedarDB

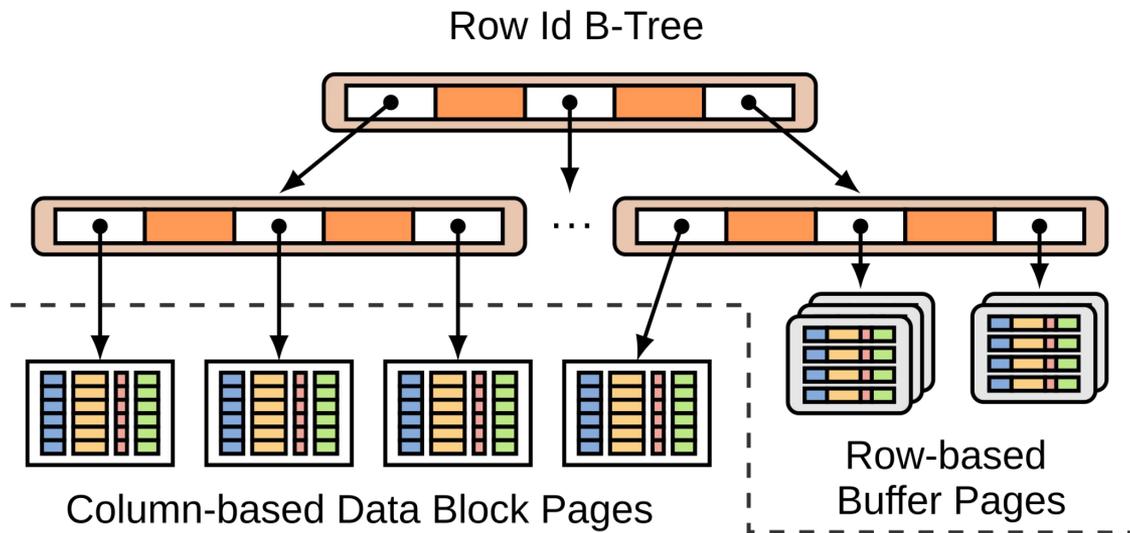
# Agenda

- Recap: DBMS Components
- Relational Algebra Optimization
- **Storage: B-Tree deep dive**

# B-Trees

Hybrid storage engine:

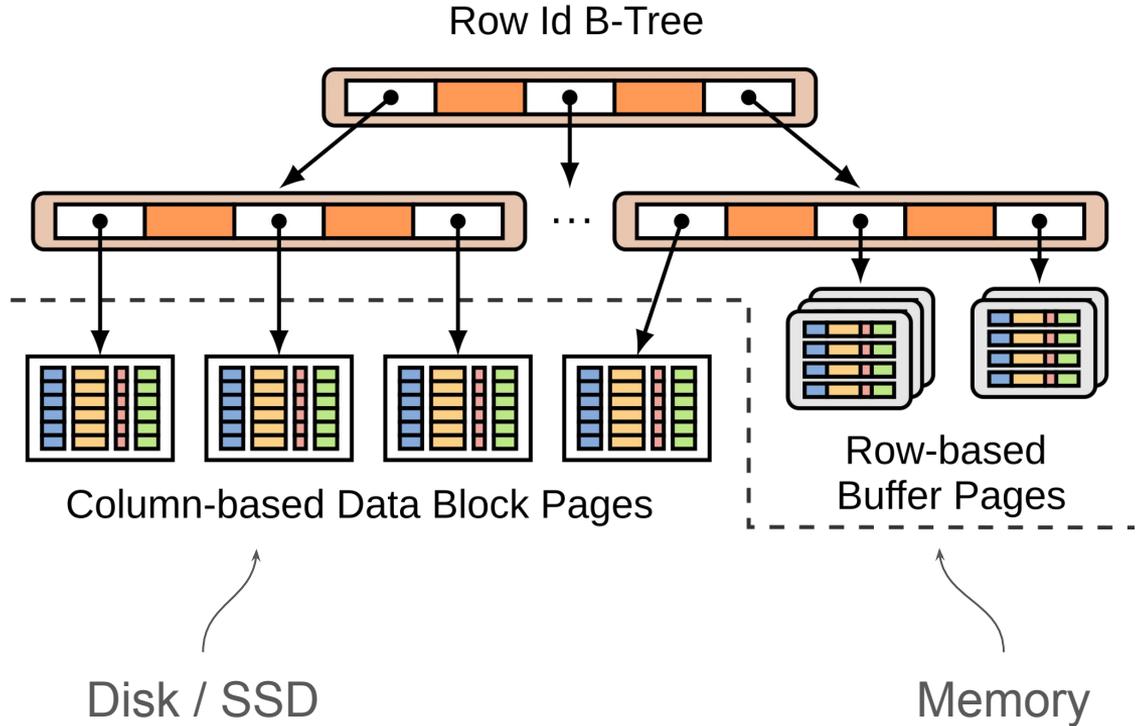
- Row oriented
- Columnar storage
- Hybrid structure
- Best of both worlds
- Hot writes at end



# B-Trees

Hybrid storage engine:

- Row oriented
- Columnar storage
- Hybrid structure
- Best of both worlds
- Hot writes at end

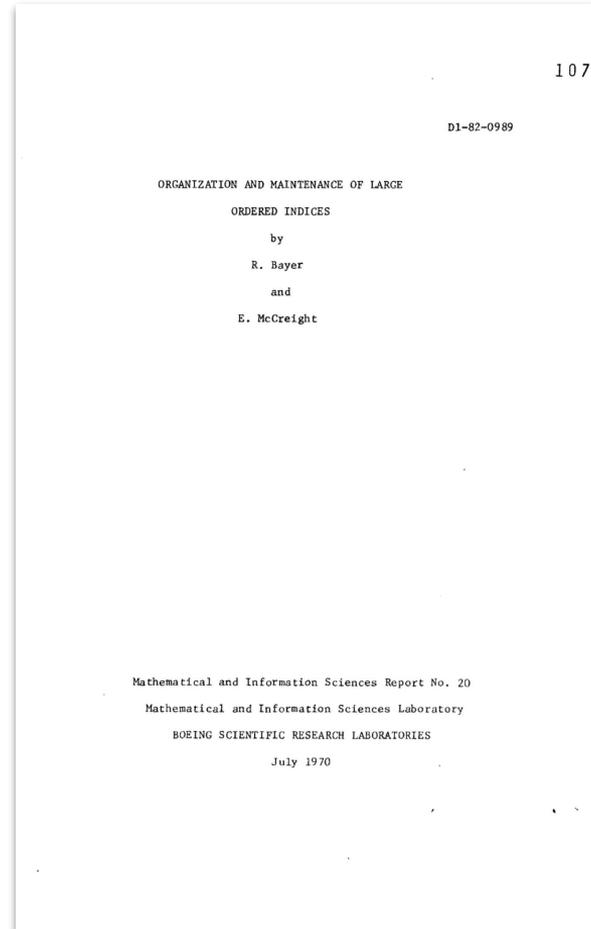




CedarDB

# B-Trees

- Universally used
  - XFS, Btrfs, APFS, & many DBMS
- 50 years old tech
- New storage engine in CedarDB

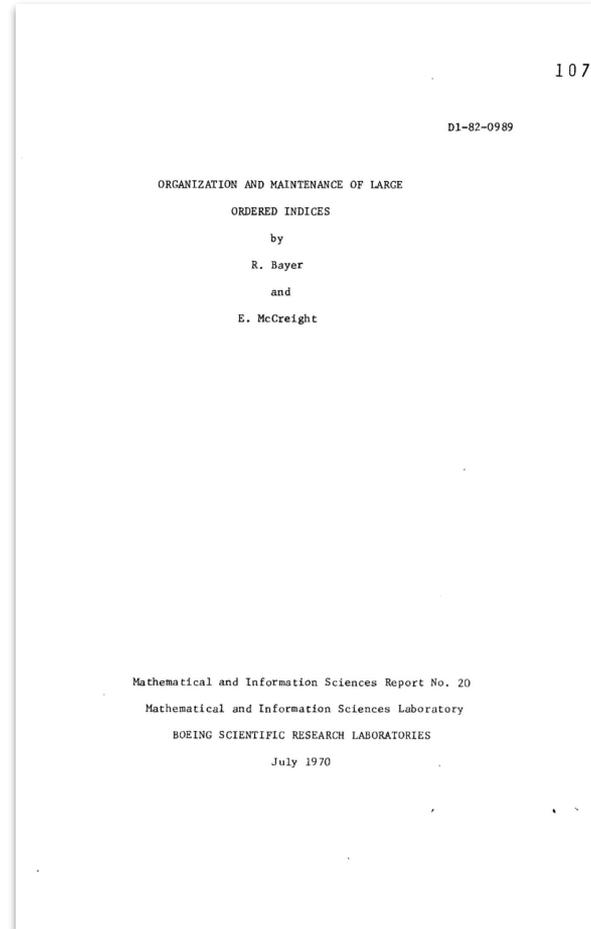




CedarDB

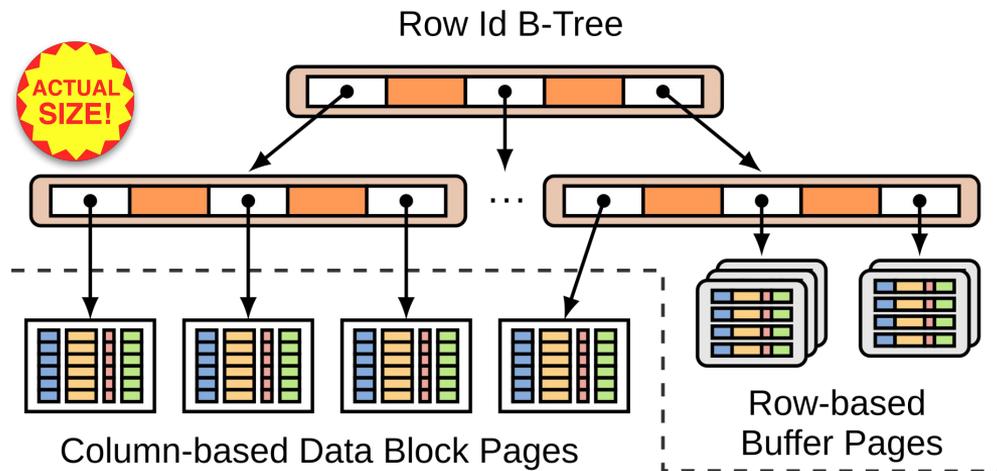
# B-Trees

- Universally used
  - XFS, Btrfs, APFS, & many DBMS
- 50 years old tech
- New storage engine in CedarDB
- Still appropriate for modern hardware



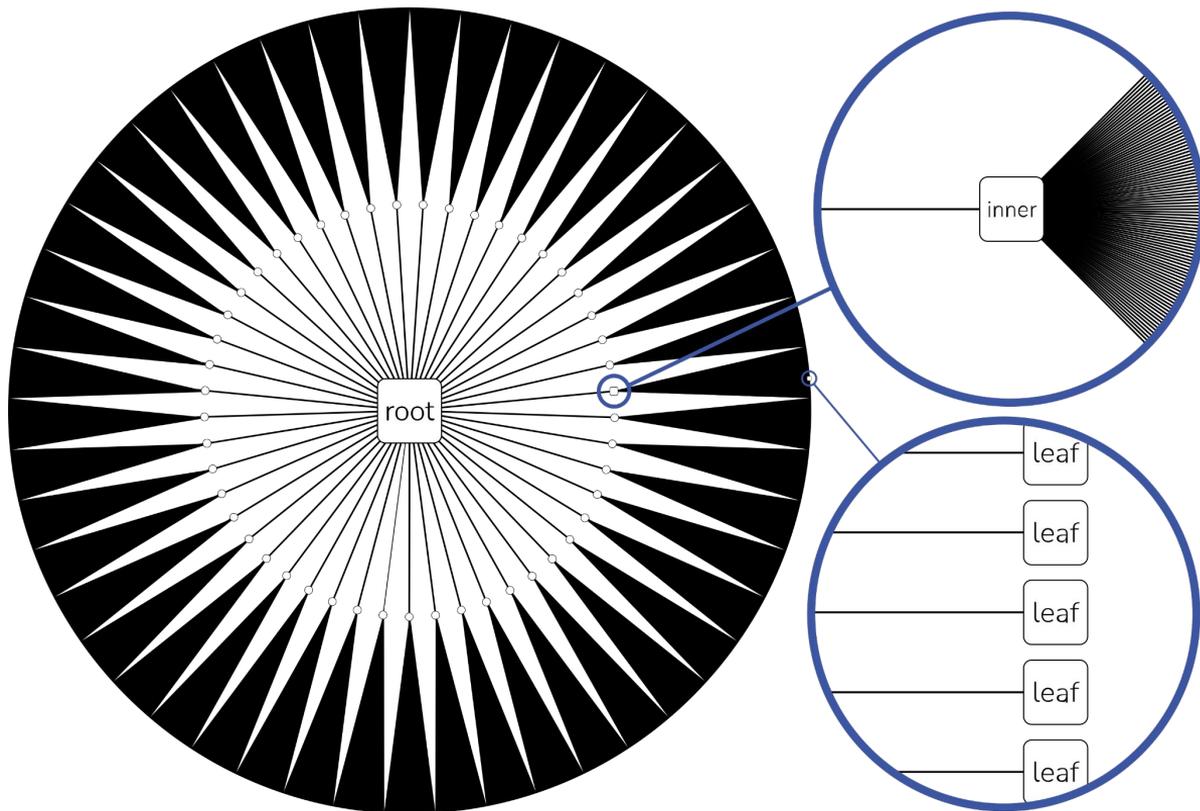
# B-Trees

- Example dataset:  
ClickBench hits
- 70GB, 100M rows
- 3 levels
  - Fanout:  
 $50 * 1500 * 1500$



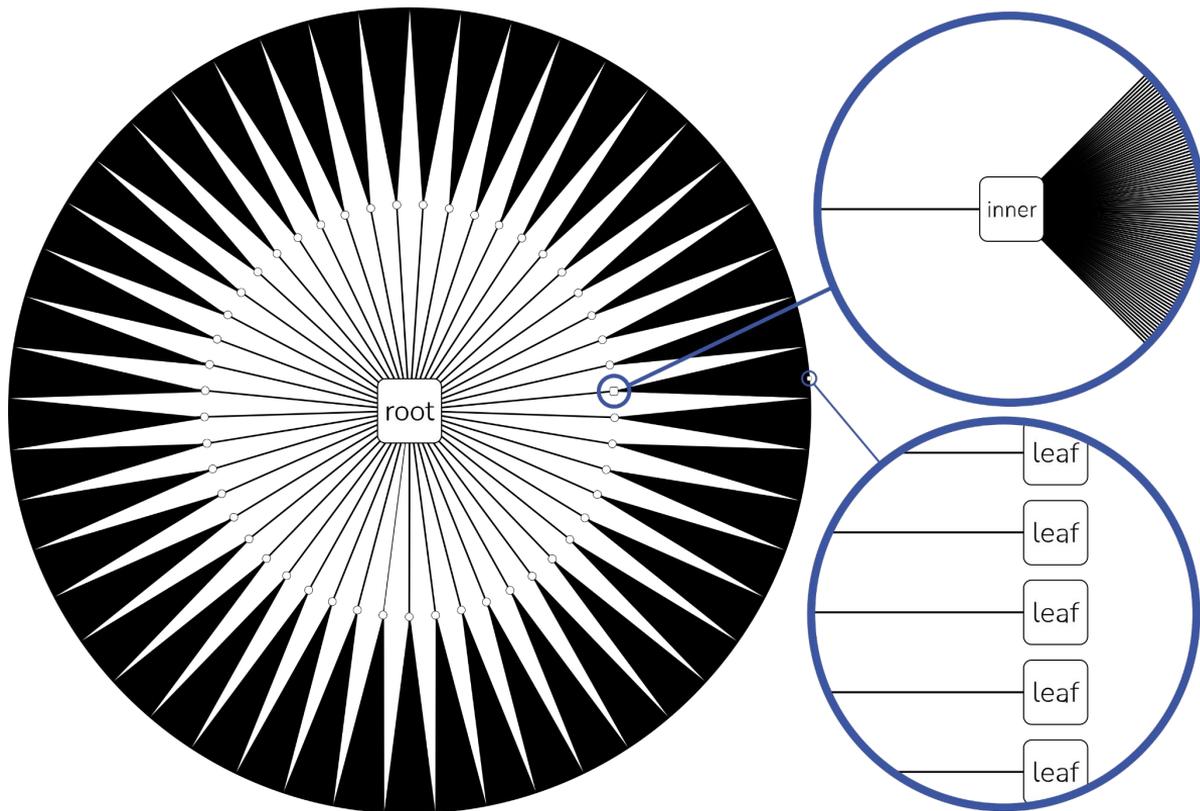
# B-Trees

- Example dataset:  
ClickBench hits
- 70GB, 100M rows
- 3 levels
  - Fanout:  
 $50 * 1500 * 1500$



# B-Trees

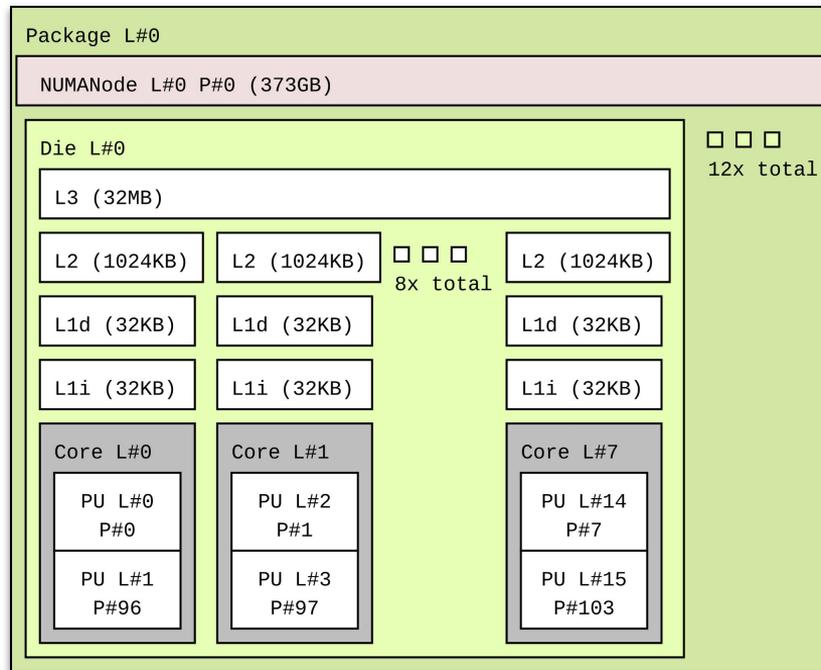
- 100M rows
- 66,689 leafs
- 49 inner
- 1 root



# On Modern Hardware

## Cache efficiency

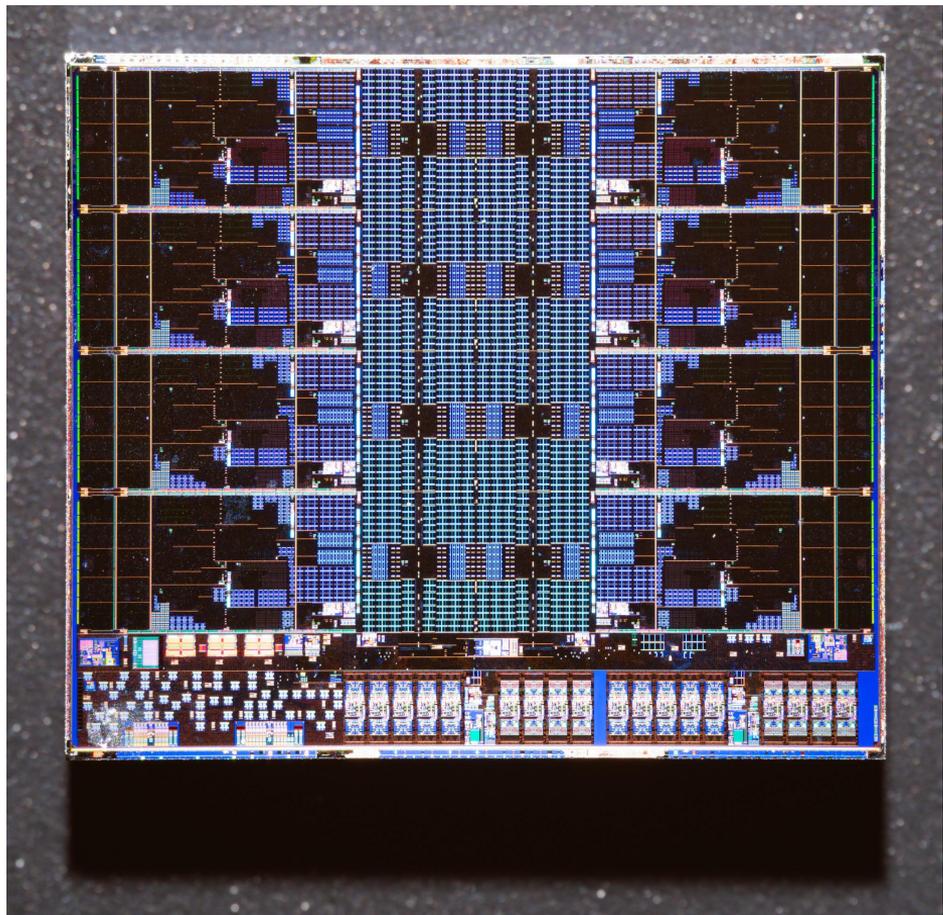
- 64 KB root
- 3.1 MB inner
- 4 GB leafs



# On Modern Hardware

## Cache efficiency

- 64 KB root
- 3.1 MB inner
- 4 GB leafs

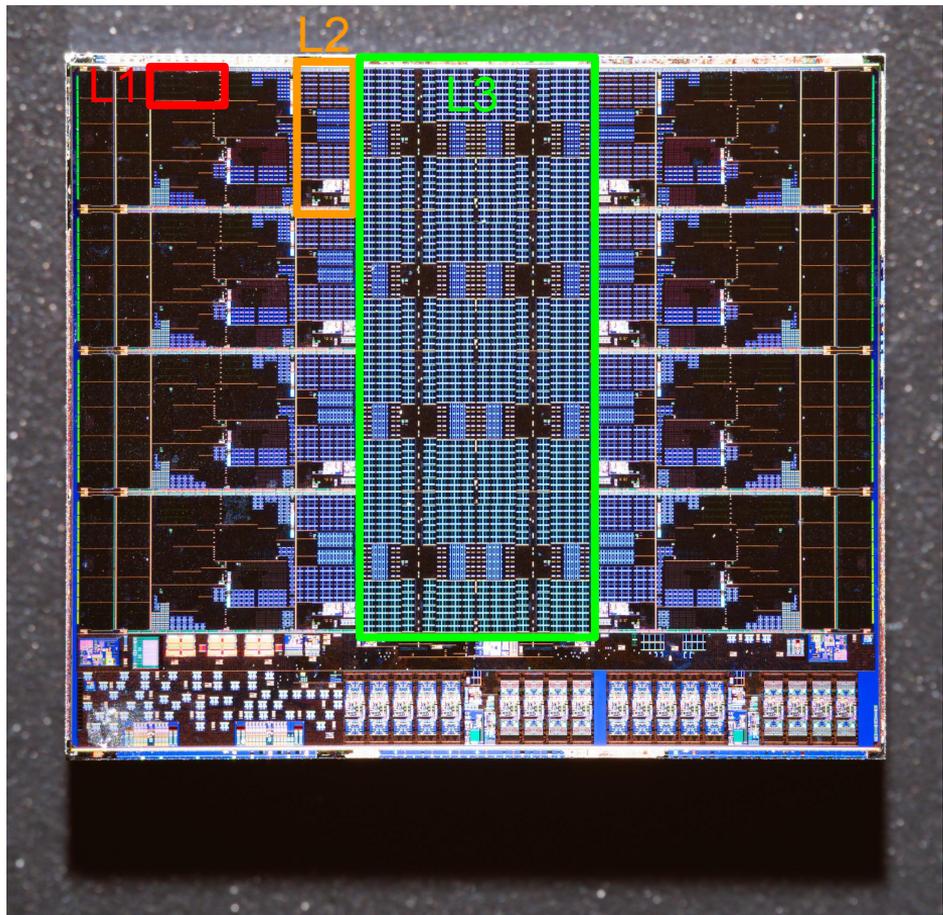


# On Modern Hardware

## Cache efficiency

- 64 KB root
- 3.1 MB inner
- 4 GB leafs

Inner nodes cached  
➔ almost no latency





CedarDB

# Lock Coupling on Modern Hardware

Problem:

Synchronization over 100s of cores



CedarDB

# Lock Coupling on Modern Hardware

Problem:

Synchronization over 100s of cores

- Lock coupling
- All accesses through root node

# Lock Coupling on Modern Hardware

Problem:  
Synchronization over 100s of cores

1. lock 
2. access 



- Lock coupling
- All accesses through root node

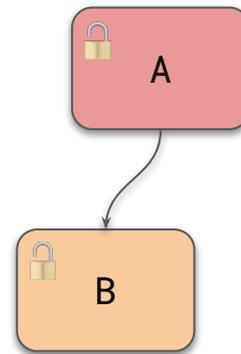
# Lock Coupling on Modern Hardware

Problem:

Synchronization over 100s of cores

- Lock coupling
- All accesses through root node

1. lock A
2. access A
3. lock B
4. unlock A
5. access B



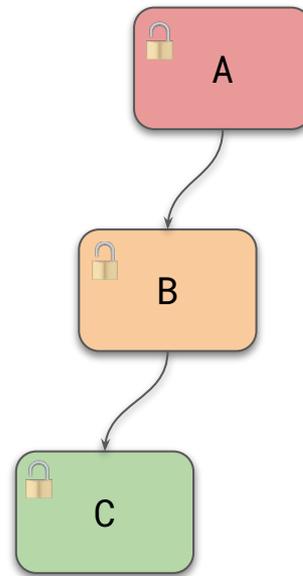
# Lock Coupling on Modern Hardware

Problem:

Synchronization over 100s of cores

- Lock coupling
- All accesses through root node

1. lock  A
2. access  A
3. lock  B
4. unlock  A
5. access  B
6. lock  C
7. unlock  B
8. access  C
9. unlock  C

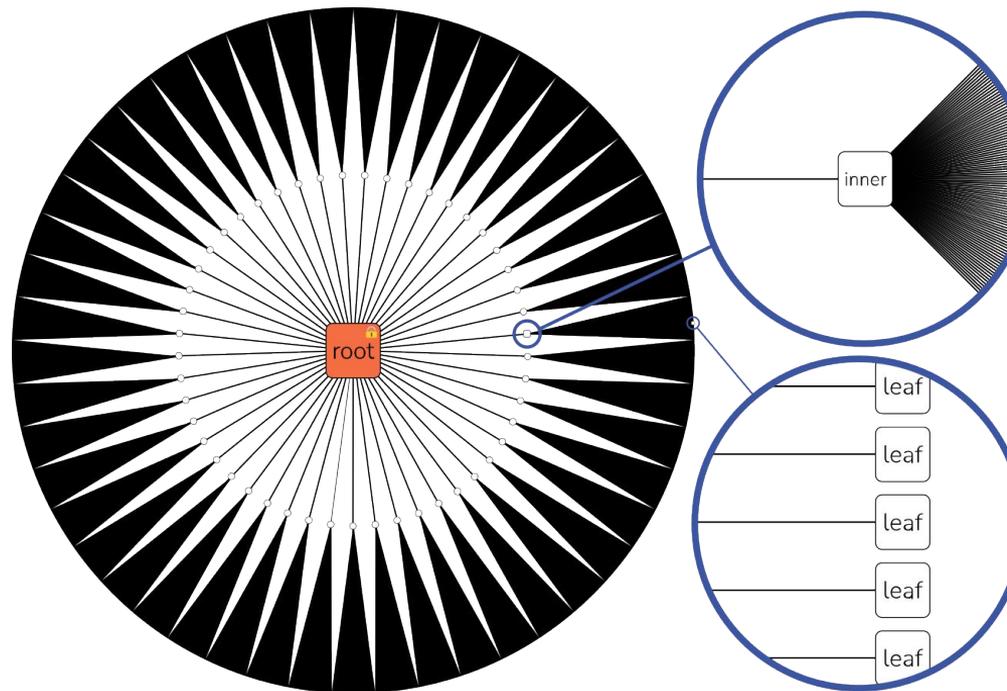


# Lock Coupling on Modern Hardware

Problem:

Synchronization over 100s of cores

- Lock coupling
- All accesses through root node

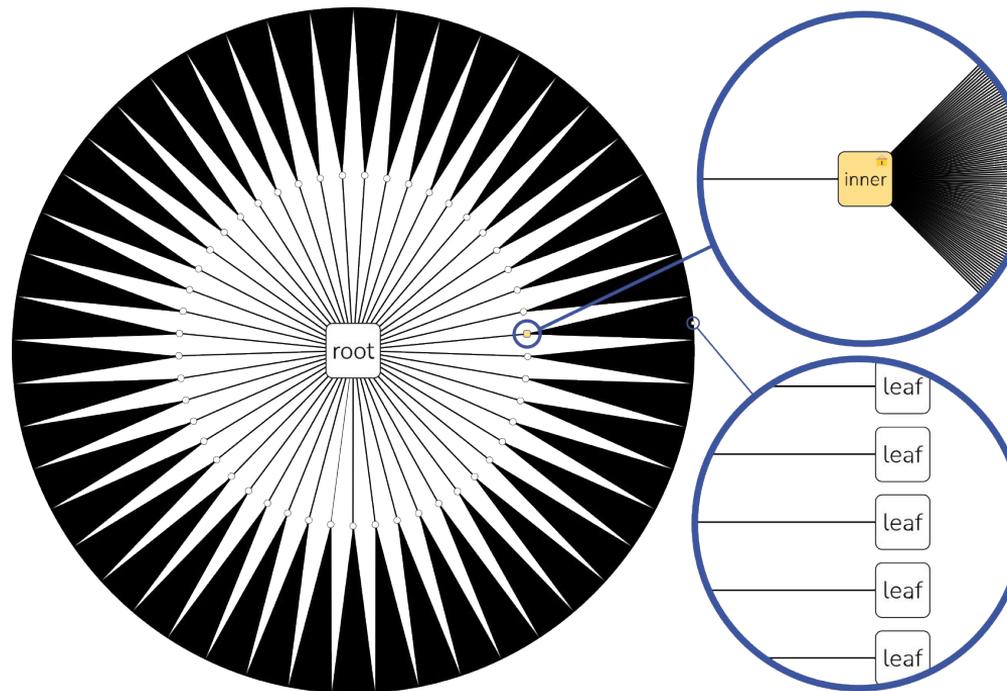


# Lock Coupling on Modern Hardware

Problem:

Synchronization over 100s of cores

- Lock coupling
- All accesses through root node

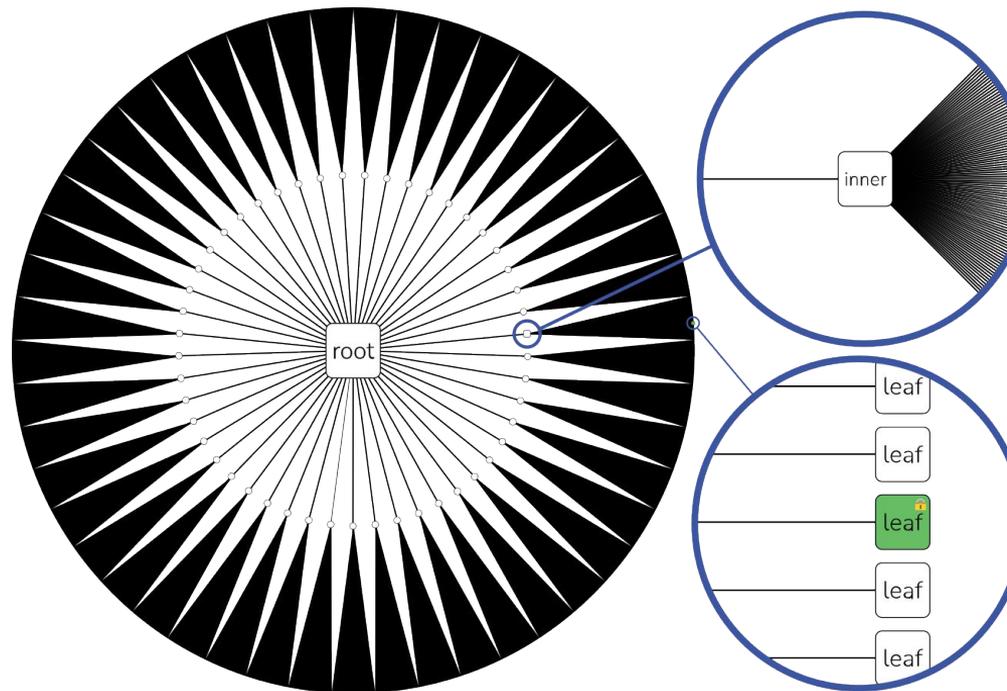


# Lock Coupling on Modern Hardware

Problem:

Synchronization over 100s of cores

- Lock coupling
- All accesses through root node
- Leafs are fine-grained
- Root is bottleneck

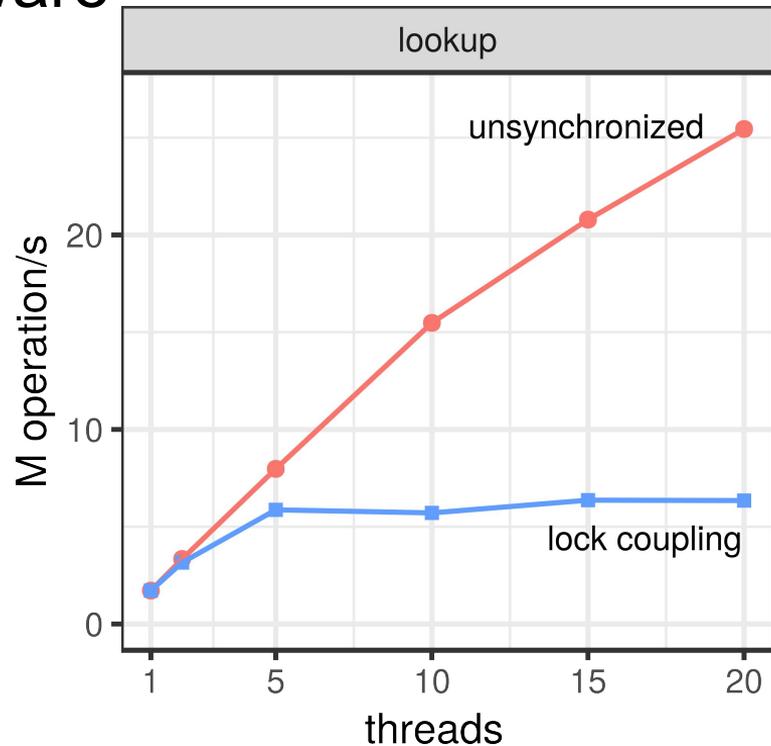


# Lock Coupling on Modern Hardware

Problem:

Synchronization over 100s of cores

- Lock coupling
- All accesses through root node
- Leafs are fine-grained
- Root is bottleneck
- Reference counting for shared locks **does not scale**
- Every lock is an **atomic write**

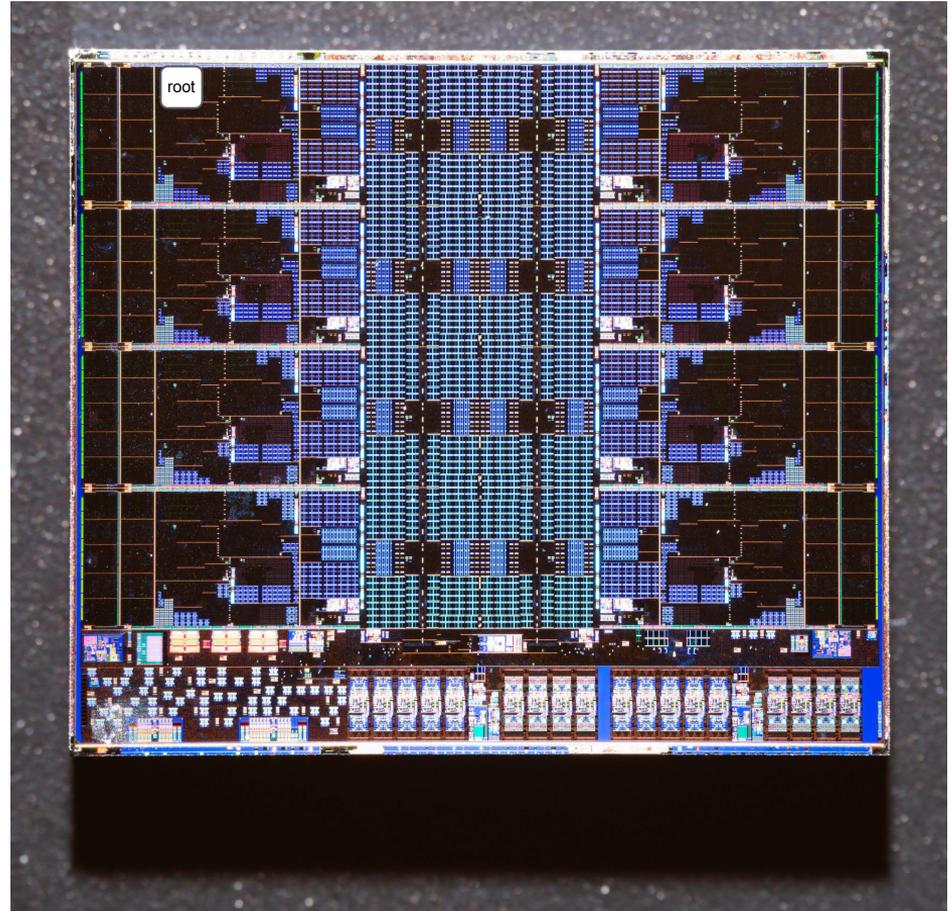


# Lock Coupling

Problem:

Synchronization over 100s of cores

- Lock coupling
- All accesses through root node
- Leafs are fine-grained
- Root is bottleneck
- Reference counting for shared locks **does not scale**
- Every lock is an **atomic write**

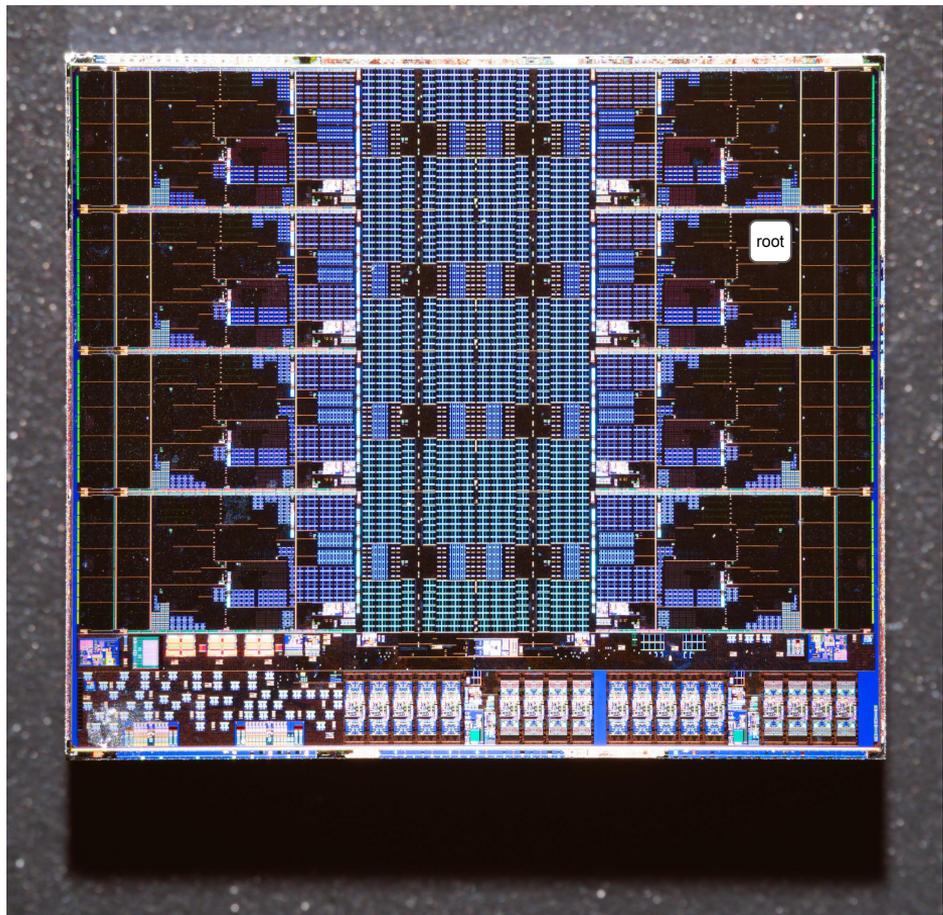


# Lock Coupling

Problem:

Synchronization over 100s of cores

- Lock coupling
- All accesses through root node
- Leafs are fine-grained
- Root is bottleneck
- Reference counting for shared locks **does not scale**
- Every lock is an **atomic write**

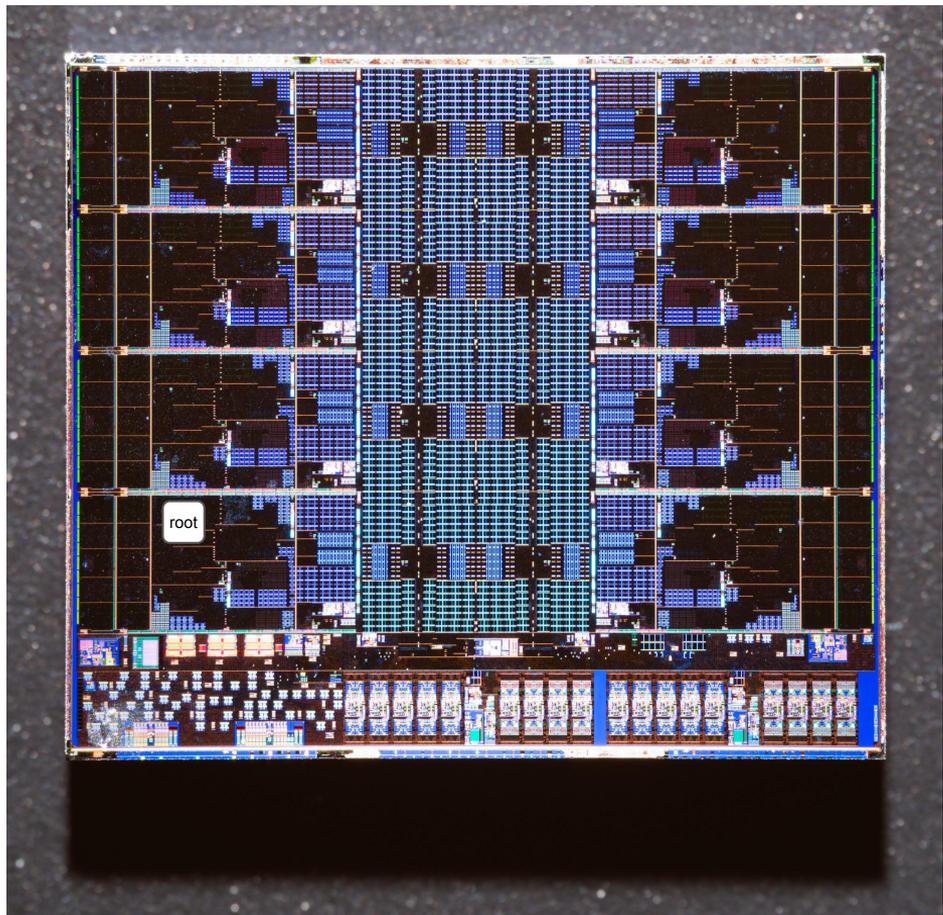


# Lock Coupling

Problem:

Synchronization over 100s of cores

- Lock coupling
- All accesses through root node
- Leafs are fine-grained
- Root is bottleneck
- Reference counting for shared locks **does not scale**
- Every lock is an **atomic write**





CedarDB

# Optimistic Lock Coupling

Idea: Ask forgiveness, not permission

# Optimistic Lock Coupling

Idea: Ask forgiveness, not permission

- Root changes rarely
- Just read unsynchronized, but verify that we didn't read wrong data



# Optimistic Lock Coupling

Idea: Ask forgiveness, not permission

- Root changes rarely
- Just read unsynchronized, but verify that we didn't read wrong data

➔ Versioning, writers increment version



CedarDB

# Optimistic Lock Coupling

Idea: Ask forgiveness, not permission

- Root changes rarely
- Just read unsynchronized, but verify that we didn't read wrong data

➔ Versioning, writers increment version

Also known as:

Seqlocks ~ Linux Kernel 2003



# Optimistic Lock Coupling

Idea: Ask forgiveness, not permission

- Root changes rarely
- Just read unsynchronized, but verify that we didn't read wrong data

➔ Versioning, writers increment version

Also known as:

Seqlocks ~ Linux Kernel 2003

## Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method

Viktor Leis, Michael Haubenschild\*, Thomas Neumann  
Technische Universität München    Tableau Software\*  
{leis,neumann}@in.tum.de    mhaubenschild@tableau.com\*

### Abstract

*As the number of cores on commodity processors continues to increase, scalability becomes more and more crucial for overall performance. Scalable and efficient concurrent data structures are particularly important, as these are often the building blocks of parallel algorithms. Unfortunately, traditional synchronization techniques based on fine-grained locking have been shown to be unscalable on modern multi-core CPUs. Lock-free data structures, on the other hand, are extremely difficult to design and often incur significant overhead.*

*In this work, we make the case for Optimistic Lock Coupling as a practical alternative to both traditional locking and the lock-free approach. We show that Optimistic Lock Coupling is highly scalable and almost as simple to implement as traditional lock coupling. Another important advantage is that it is easily applicable to most tree-like data structures. We therefore argue that Optimistic Lock Coupling, rather than a complex and error-prone custom synchronization protocol, should be the default choice for performance-critical data structures.*

### 1 Introduction

Today, Intel's commodity server processors have up to 28 cores and its upcoming microarchitecture will have up to 48 cores per socket [6]. Similarly, AMD currently stands at 32 cores and this number is expected to double in the next generation [20]. Since both platforms support simultaneous multithreading (also known as hyperthreading), affordable commodity servers (with up to two sockets) will soon routinely have between 100 and 200 hardware threads.

With such a high degree of hardware parallelism, efficient data processing crucially depends on how well concurrent data structures scale. Internally, database systems use a plethora of data structures like table heaps, internal work queues, and, most importantly, index structures. Any of these can easily become a scalability (and therefore overall performance) bottleneck on many-core CPUs.

Traditionally, database systems synchronize internal data structures using fine-grained reader/writer locks<sup>1</sup>. Unfortunately, while fine-grained locking makes lock contention unlikely, it still results in bad scalability because

Copyright 2019 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

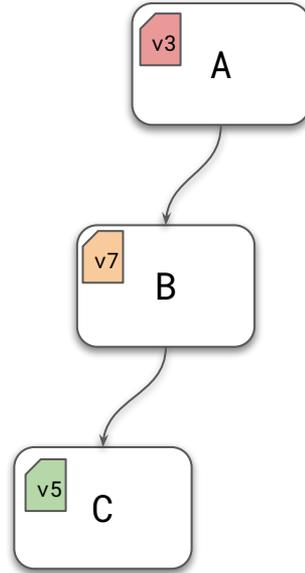
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

<sup>1</sup>In this work, we focus on data structure synchronization rather than high-level transaction semantics and therefore use the term *lock* for what would typically be called *latch* in the database literature. We thus follow common computer science (rather than database) terminology.

# Optimistic Lock Coupling

Lock Coupling:

1. lock A
2. access A
3. lock B
4. unlock A
5. access B
6. lock C
7. unlock B
8. access C
9. unlock C



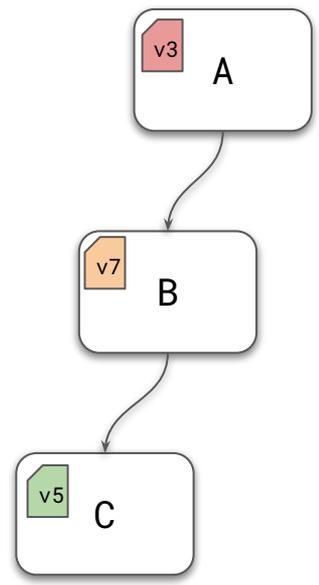
Optimistic:

1. read v3
2. access A
3. read v7
4. validate v3
5. access B
6. read v5
7. validate v7
8. access C
9. validate v5

# Optimistic Lock Coupling

Lock Coupling:

1. lock A
2. access A
3. lock B
4. unlock A
5. access B
6. lock C
7. unlock B
8. access C
9. unlock C



Optimistic:

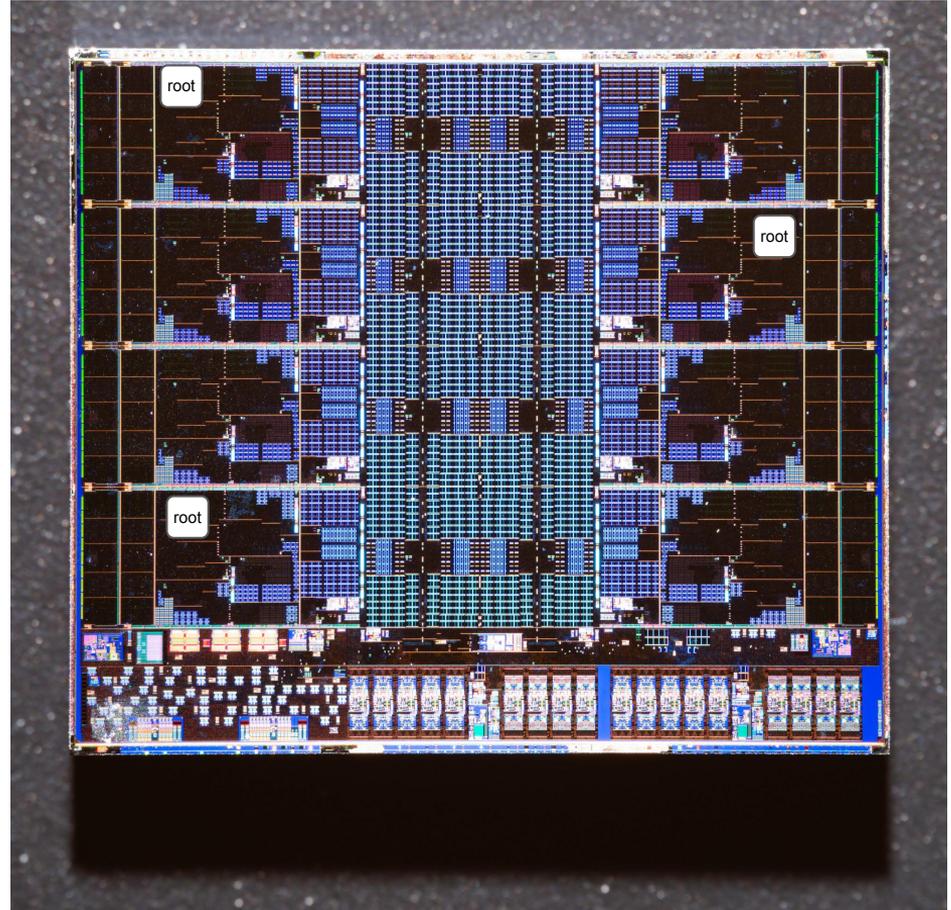
1. read v3
2. access A
3. read v7
4. validate v3
5. access B
6. read v5
7. validate v7
8. access C
9. validate v5

6 atomic writes

read only

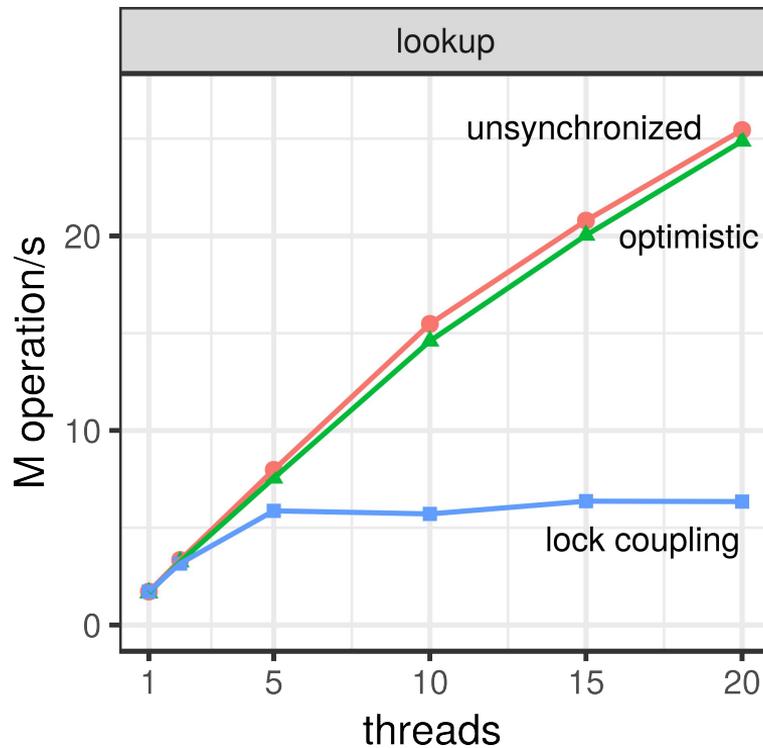
# Optimistic Lock Coupling

- Shared data
- No contention
- Less memory traffic



# Optimistic Lock Coupling

Much better scalability!

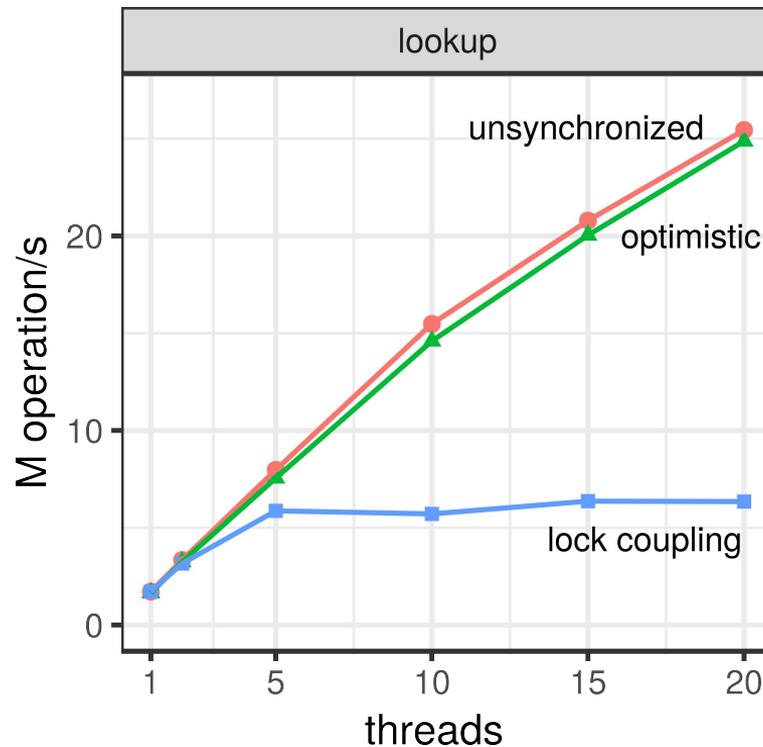


# Optimistic Lock Coupling

Much better scalability!

- Practically lock free
- Practically cache oblivious

But: Still rarely used



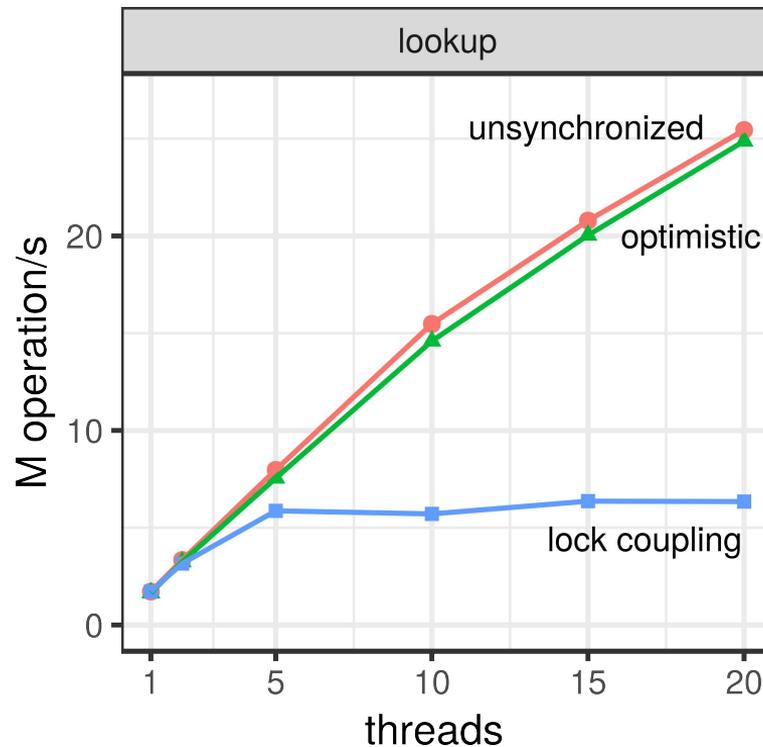
# Optimistic Lock Coupling

Much better scalability!

- Practically lock free
- Practically cache oblivious

But: Still rarely used

- Conceptually simple  
(for a lock free data structure)
- But the devil is in the details





Try it now:

Free Community Edition for Linux / Docker:

```
curl https://get.cedardb.com | bash
```



philipp@cedardb.com