CedarDB

CedarDB

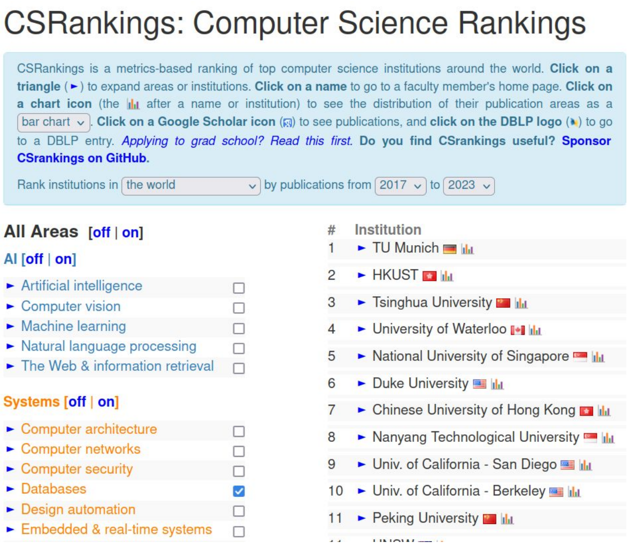
Philipp Fent
philipp@cedardb.com

CedarDB

Overview

UMBRA

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

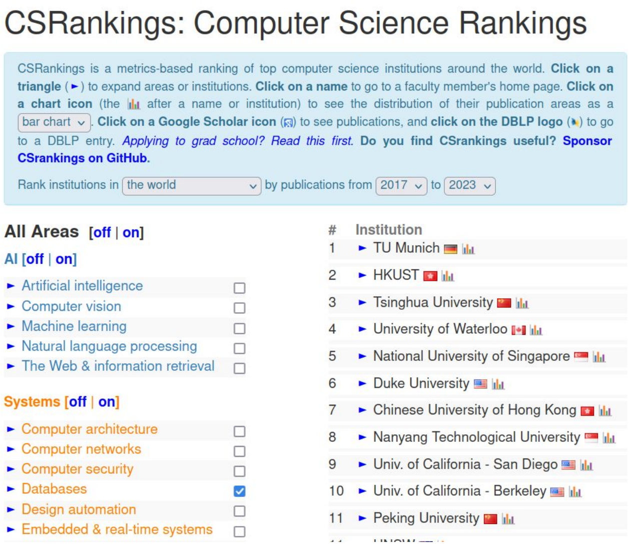


CedarDB

Overview

UMBRA

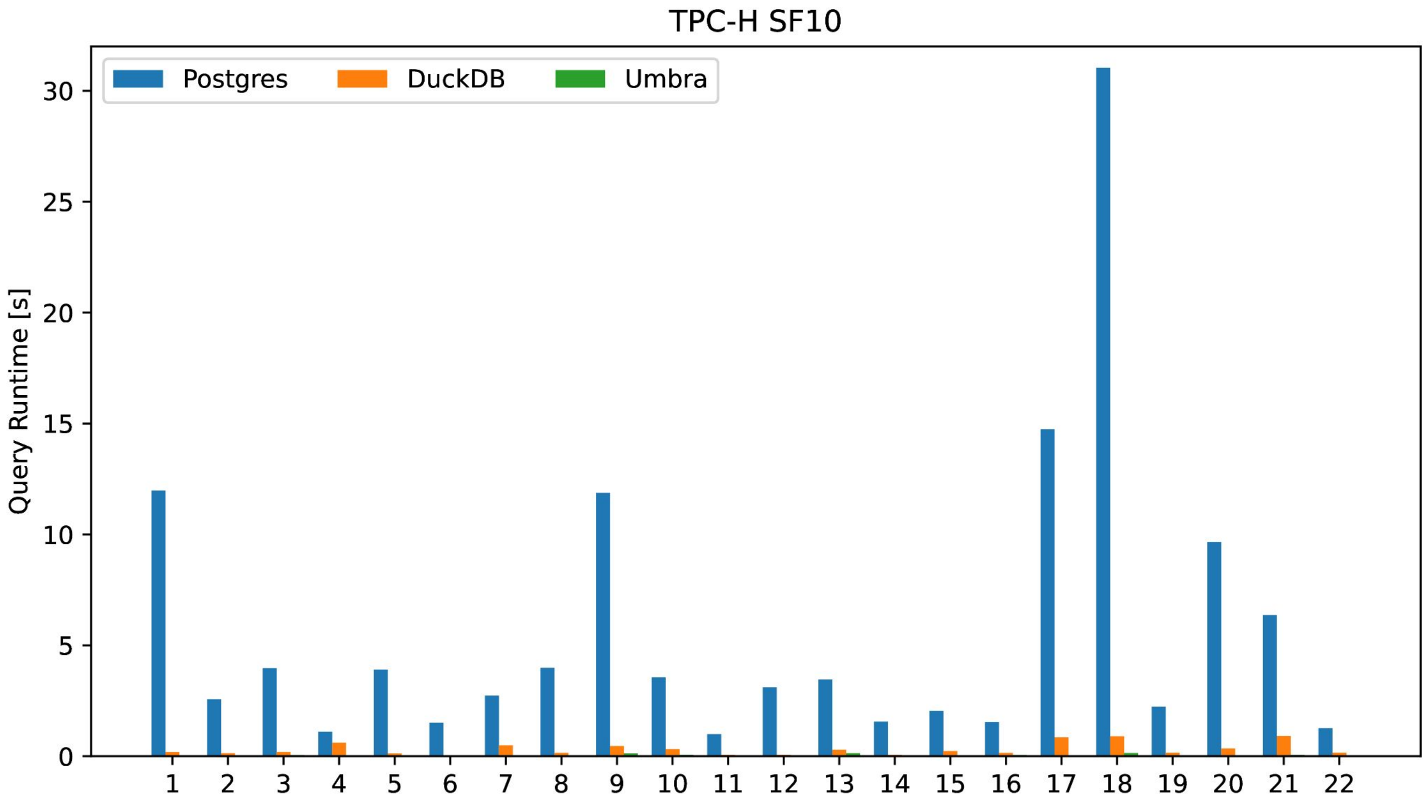
- TUM Startup
  - Started at TUM with Umbra
  - Cutting-edge database research
  - Query compilation
  - Disk-based with in-memory performance



- "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

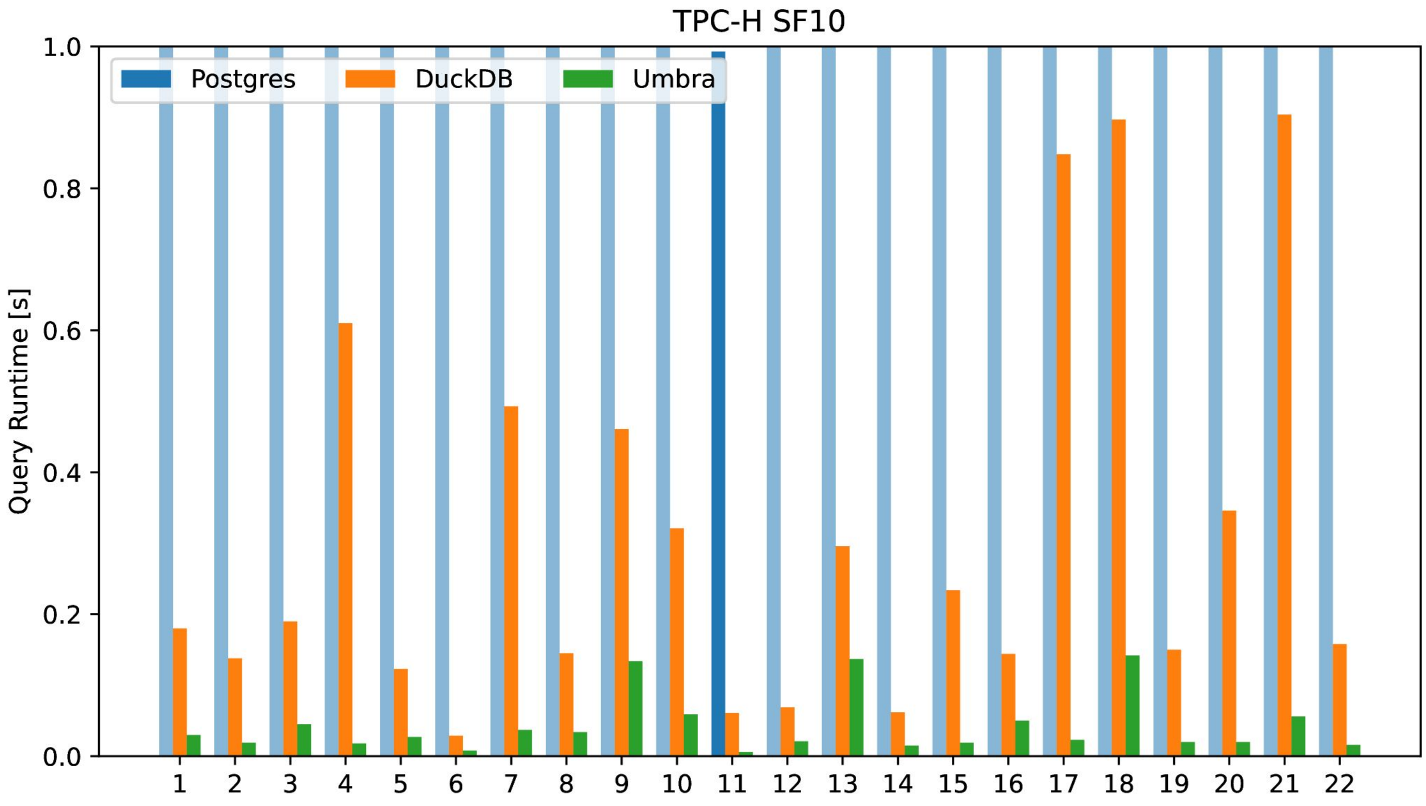
CedarDB

Performance



CedarDB

Performance



CedarDB

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
```

CedarDB

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, 5], 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 all over the database system in order to handle large objects. We ar-

10th Annual Conference on Innovative Data Systems Research (CIDR '20). January 12-15, 2020, Amsterdam, Netherlands.

Agenda

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

CedarDB

Recap: DBMS Components

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

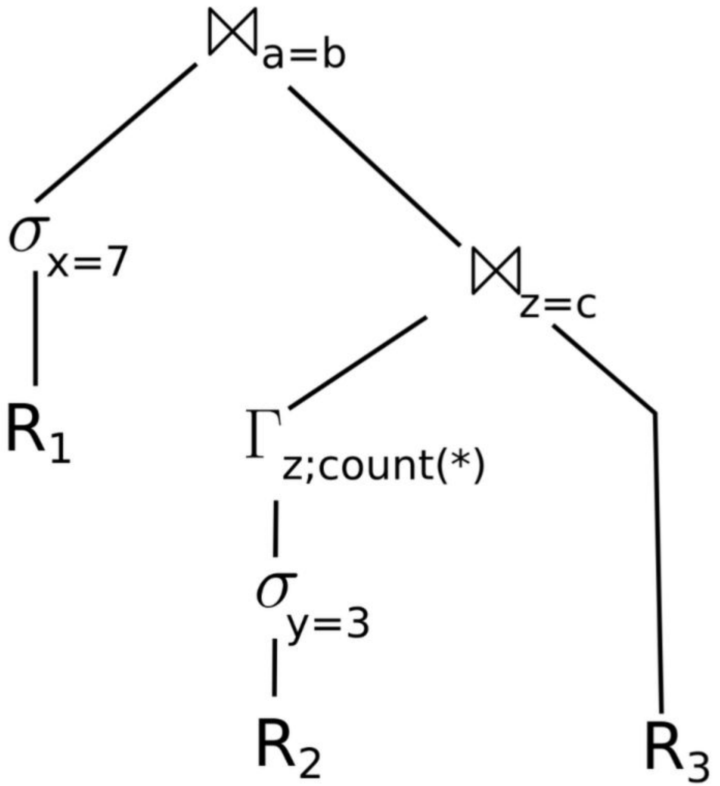
CedarDB

SQL Parsing

```sql
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

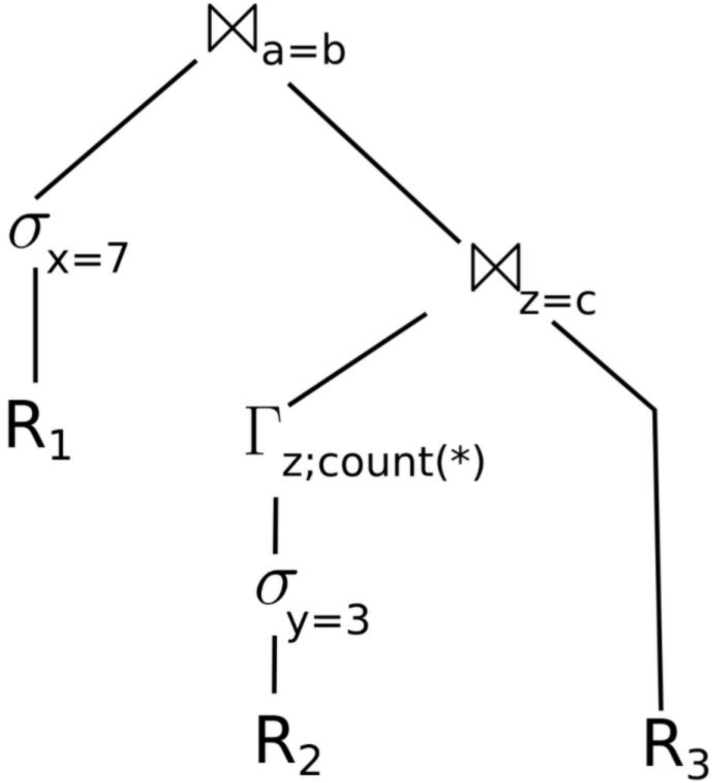
```sql
SELECT *
FROM R1,R3, (
     SELECT R2.z, count(*)
     FROM R2
     WHERE R2.y = 3
     GROUP BY R2.z
) R2
WHERE R1.x = 7
AND R1.a = R3.b
AND R2.z = R3.c
```



CedarDB

Relational Algebra Plan

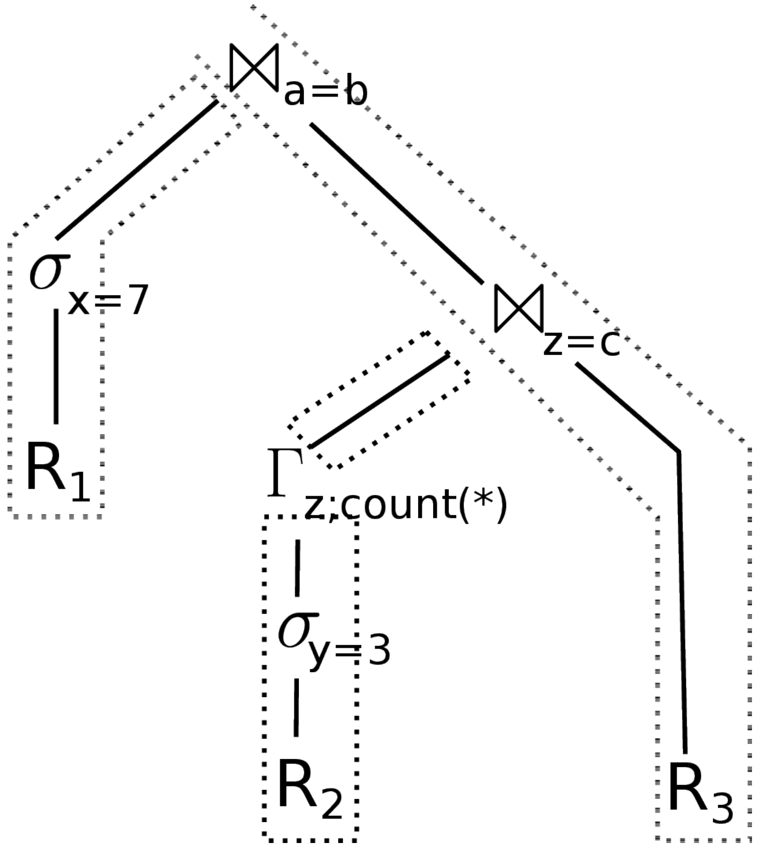
- Set-Oriented Query Processing
- Allows abstract optimization
- Crucial for efficient execution



CedarDB

Physical Execution Plan

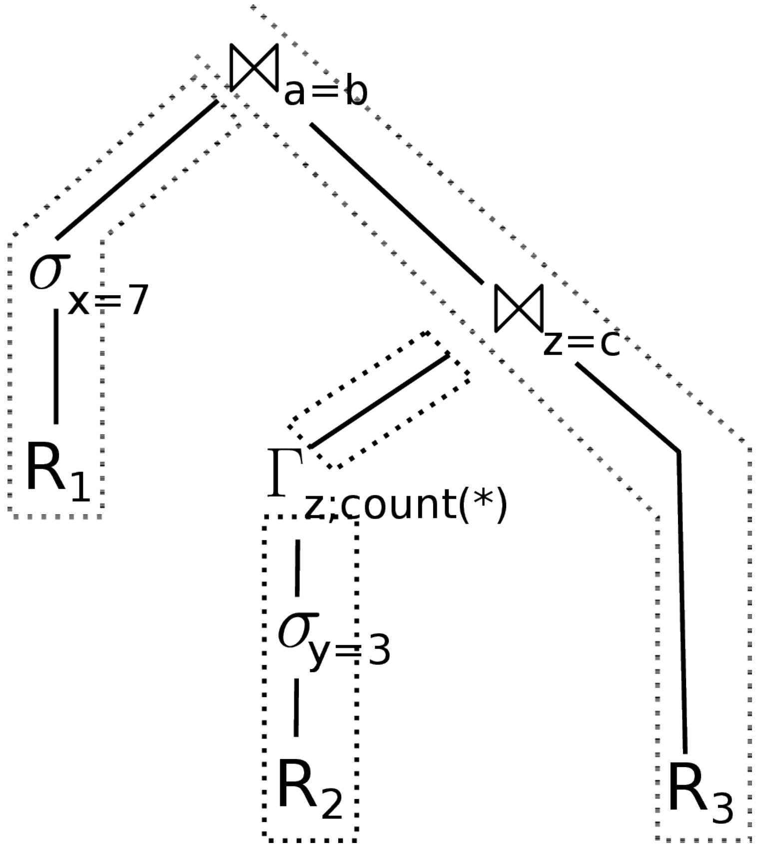
- Physical access paths
  - Index or table scan



CedarDB

Physical Execution in CedarDB

- Pipelined execution
  - Keeps values in registers
  - Minimizes materialization



Physical Execution in CedarDB

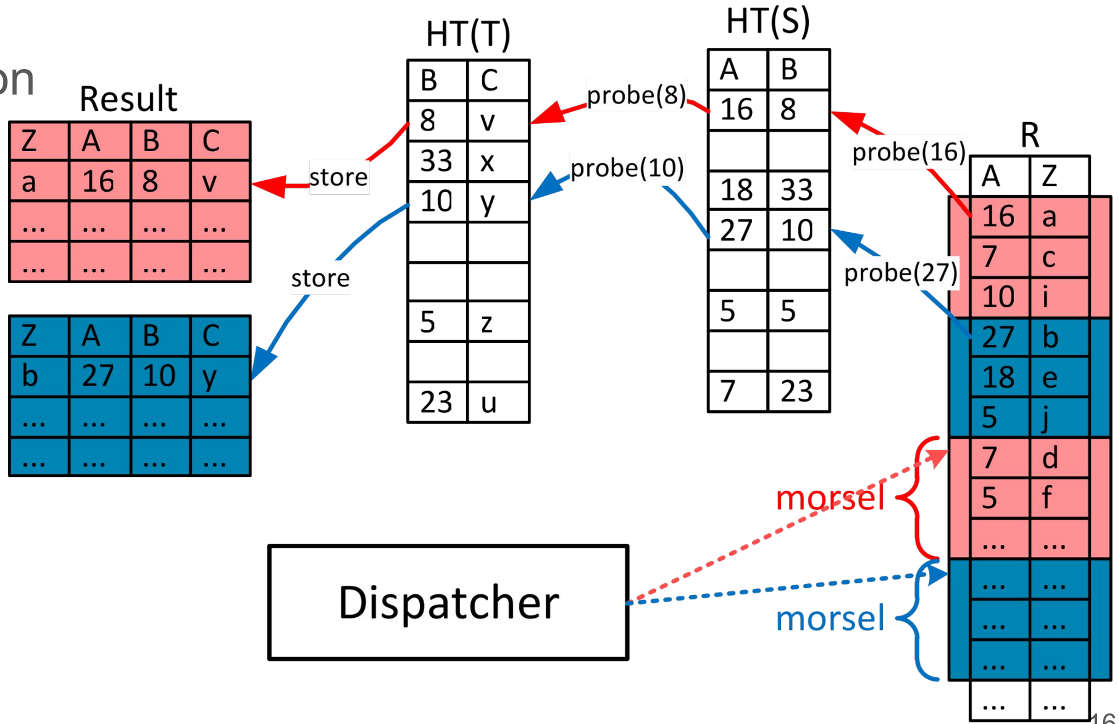
- Pipelined execution
- Data-centric code generation
  - Efficient code for complex expressions

```
%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
```

CedarDB

Physical Execution in CedarDB

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

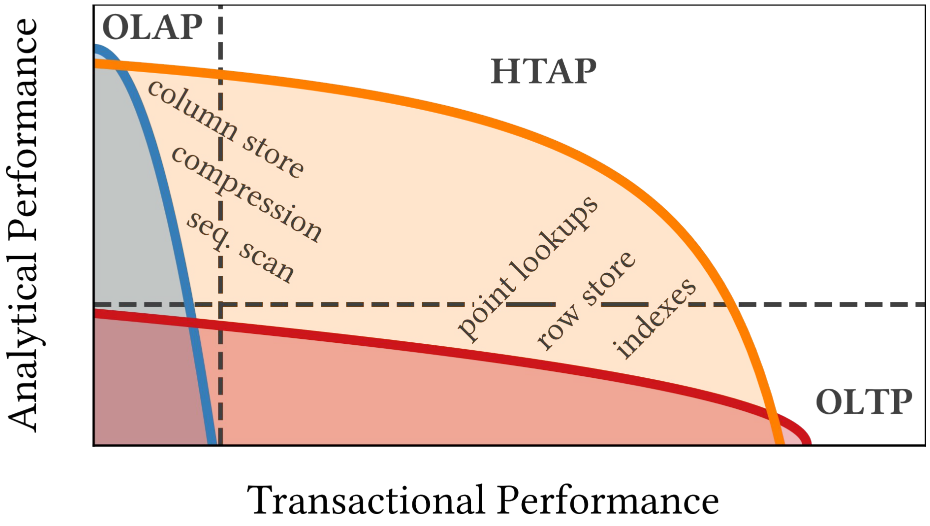


CedarDB

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

Query Optimization

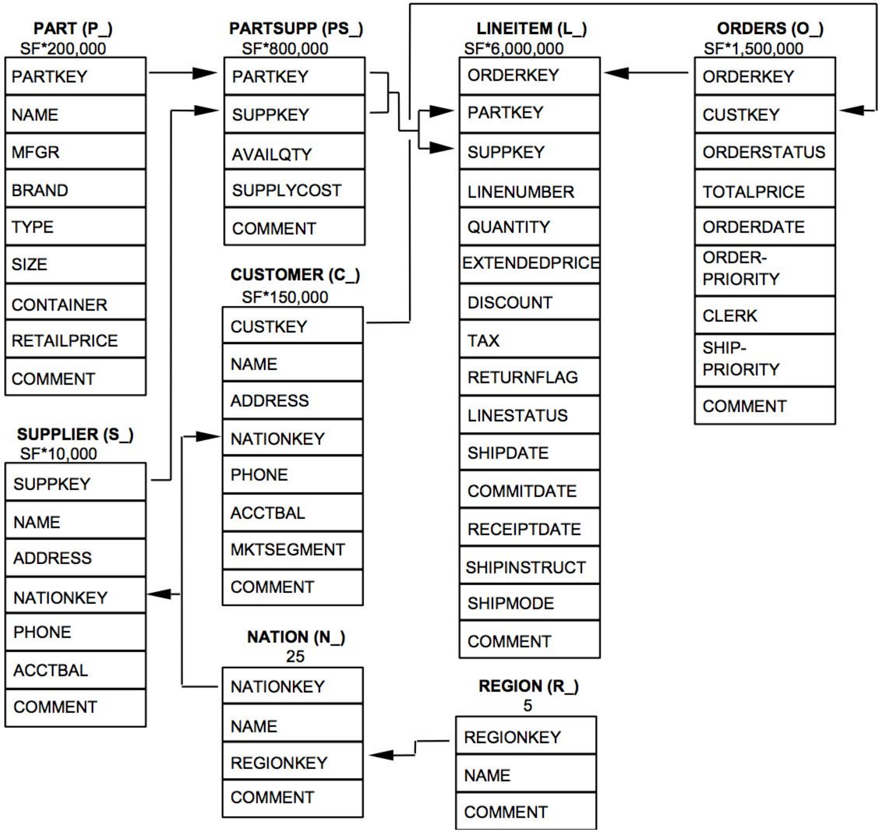
- PostgreSQL grammar
- Parsed into relational algebra
  - Example: TPC-H Q17
  - https://umbra-db.com/interface/

CedarDB

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.

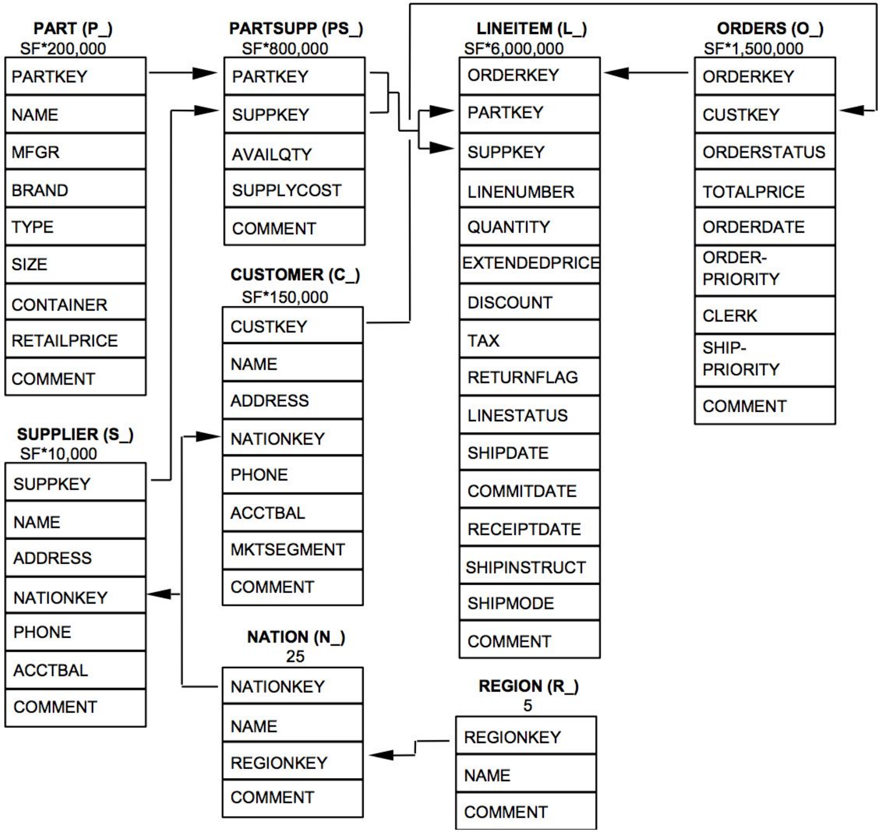
Figure 2: The TPC-H Schema



Running Example: TPC-H Q17

```sql
-- 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
)
```

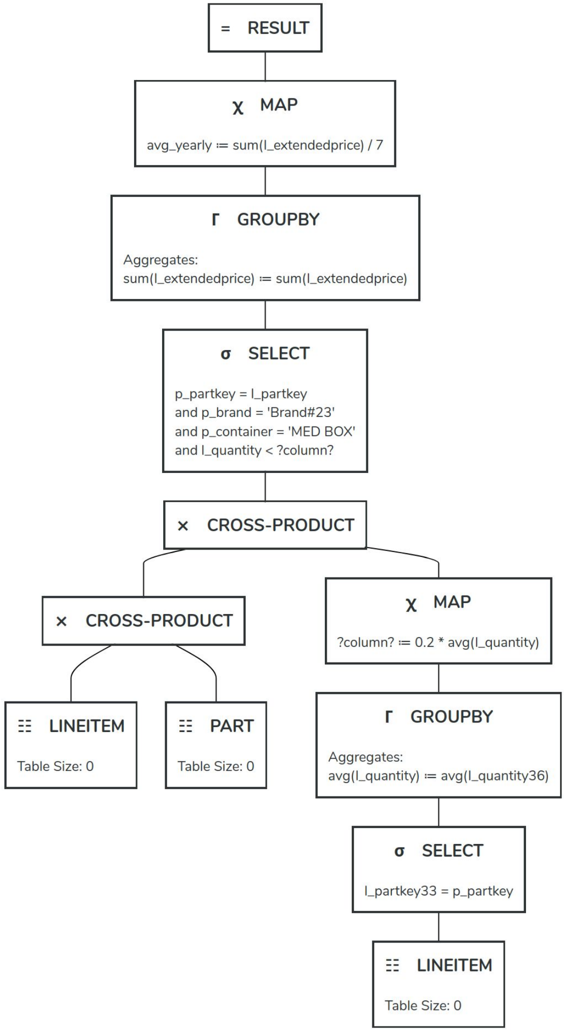
Figure 2: The TPC-H Schema



CedarDB

Query Optimization

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



CedarDB

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

CedarDB

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

CedarDB

Expression Simplification

- Fold constants
- Canonicalize expressions

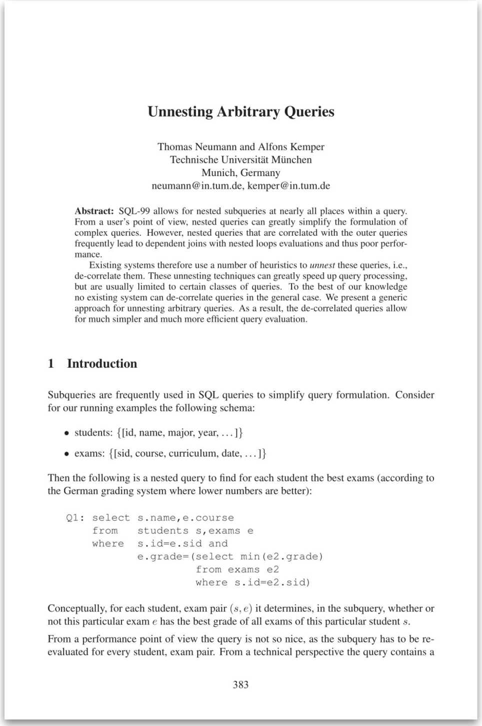
```
    o_orderdate >= date '1994-01-01'
and o_orderdate <  date '1994-01-01' + interval '1' year

                      ==

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

- Execute in evaluation engine

CedarDB

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, ...]}

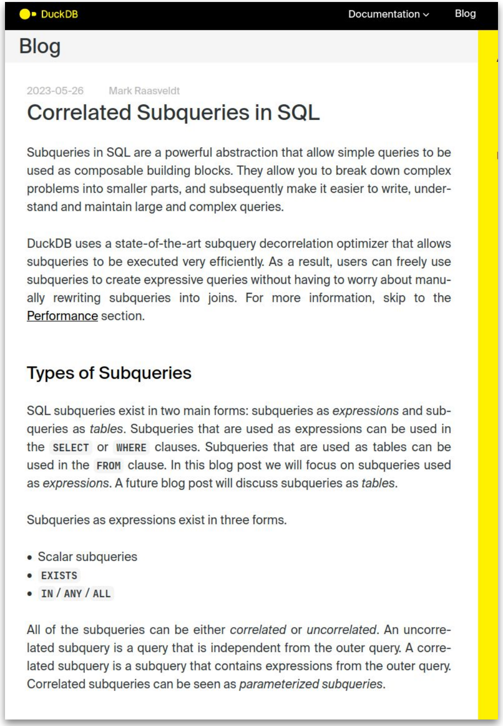
Then the following is a nested query to find for each student the best exams (according to the German grading system where lower numbers are better):

```
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.

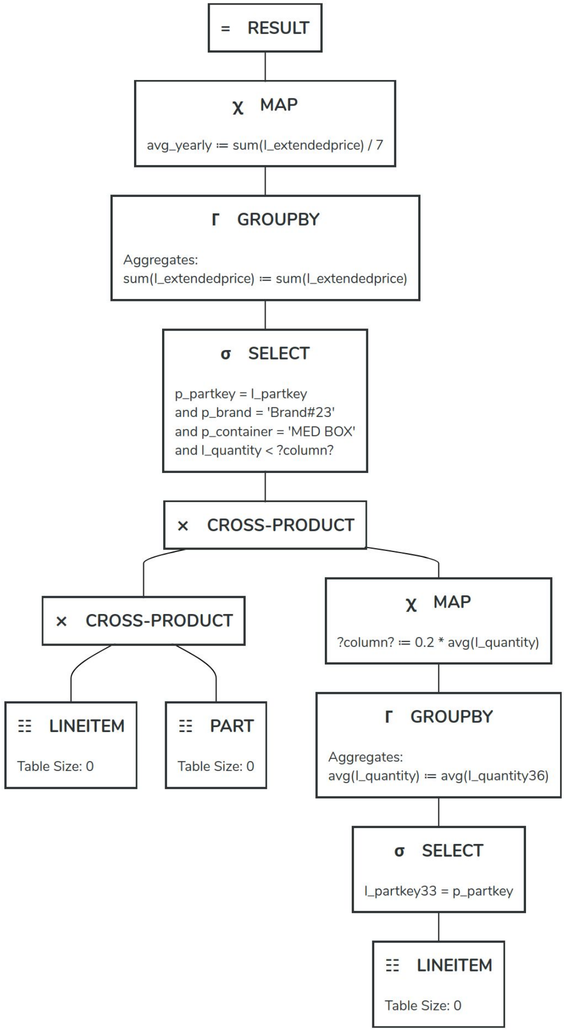
- Scalar subqueries
- `EXISTS`
- `IN` / `ANY` / `ALL`

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*.

CedarDB

Query Unnesting

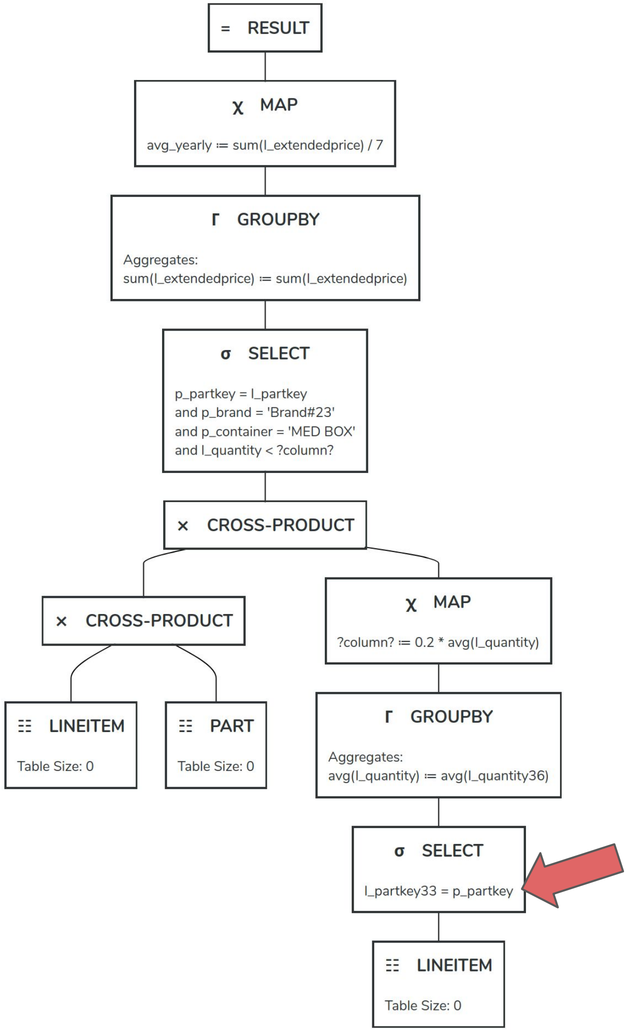
- Unnesting Arbitrary Queries
  - $O(n^2)$



CedarDB

Query Unnesting

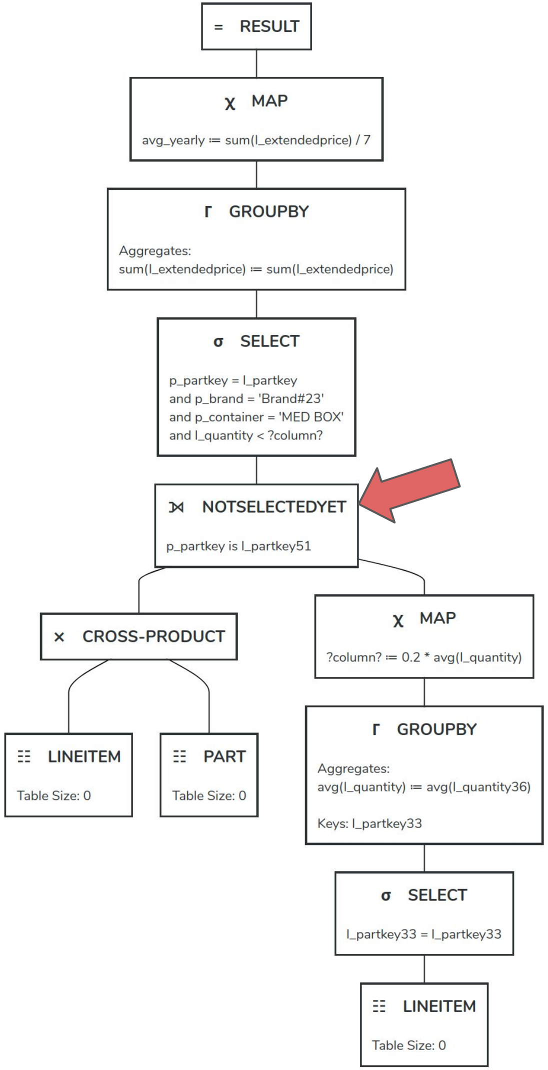
- Unnesting Arbitrary Queries
  - O(n²)



CedarDB

Query Unnesting

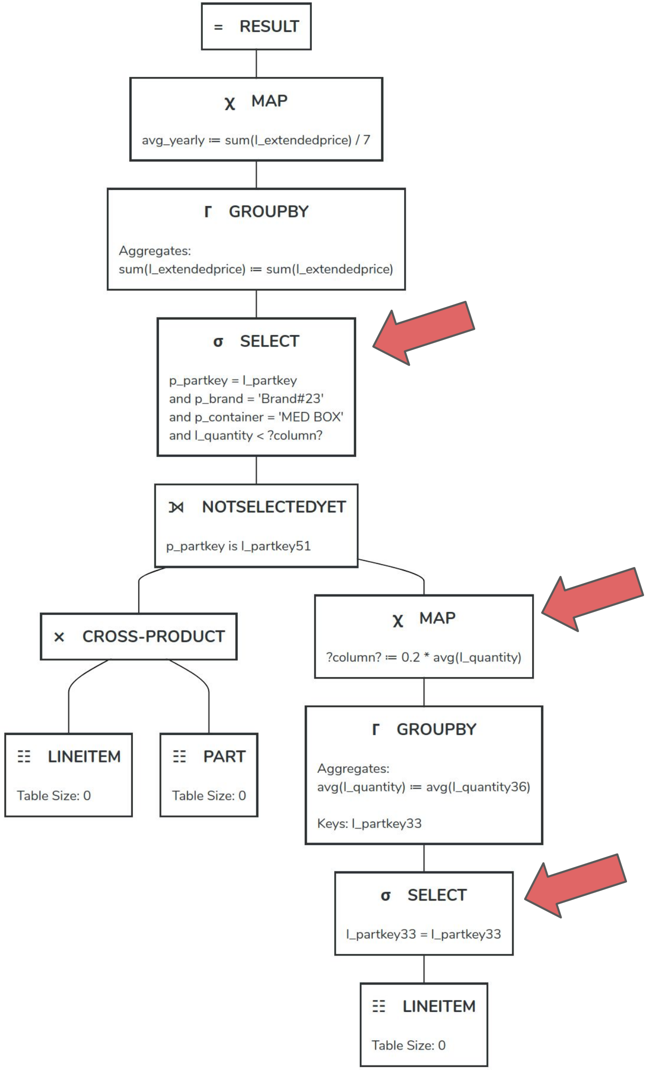
- Unnesting Arbitrary Queries
  - $O(n^2)$ -> $O(n)$
  - Huge improvement



CedarDB

Predicate Pushdown

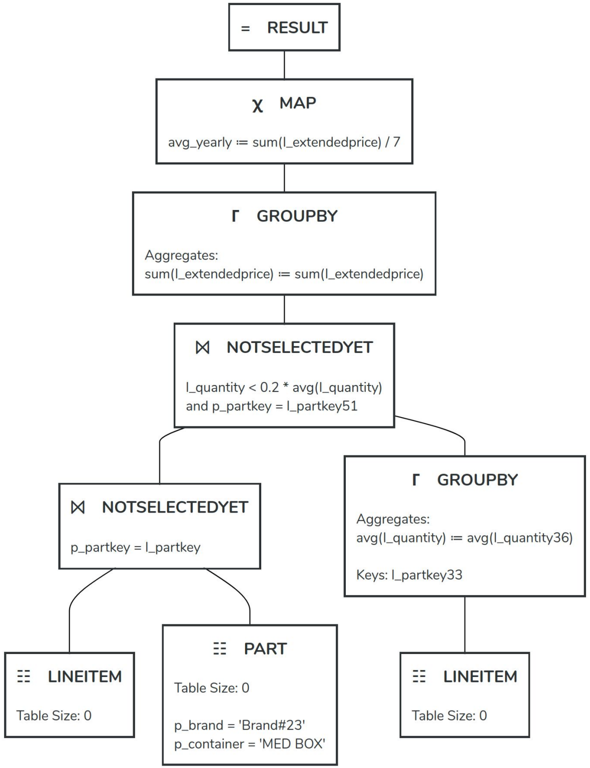
- Place predicates at scan
- Propagate & fold constants



CedarDB

Predicate Pushdown

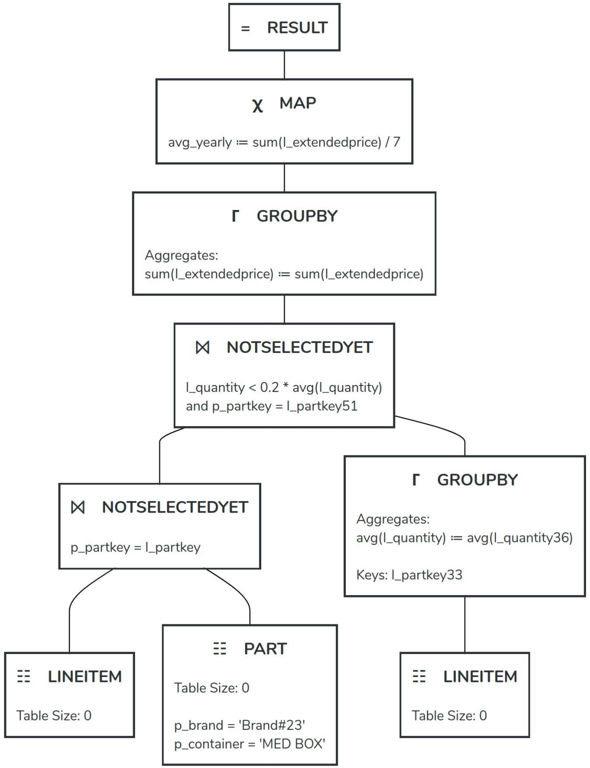
- Place predicates at scan
- Propagate & fold constants



CedarDB

Predicate Pushdown

- Place predicates at scan
- Propagate & fold constants

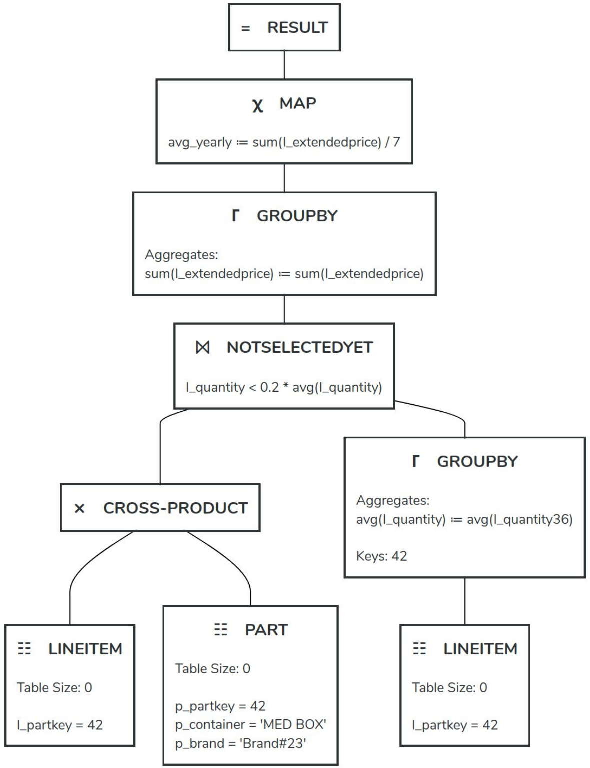



```
+   where p_partkey = 42
```

CedarDB

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
```

CedarDB

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
```

CedarDB

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

CedarDB

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

- 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
```

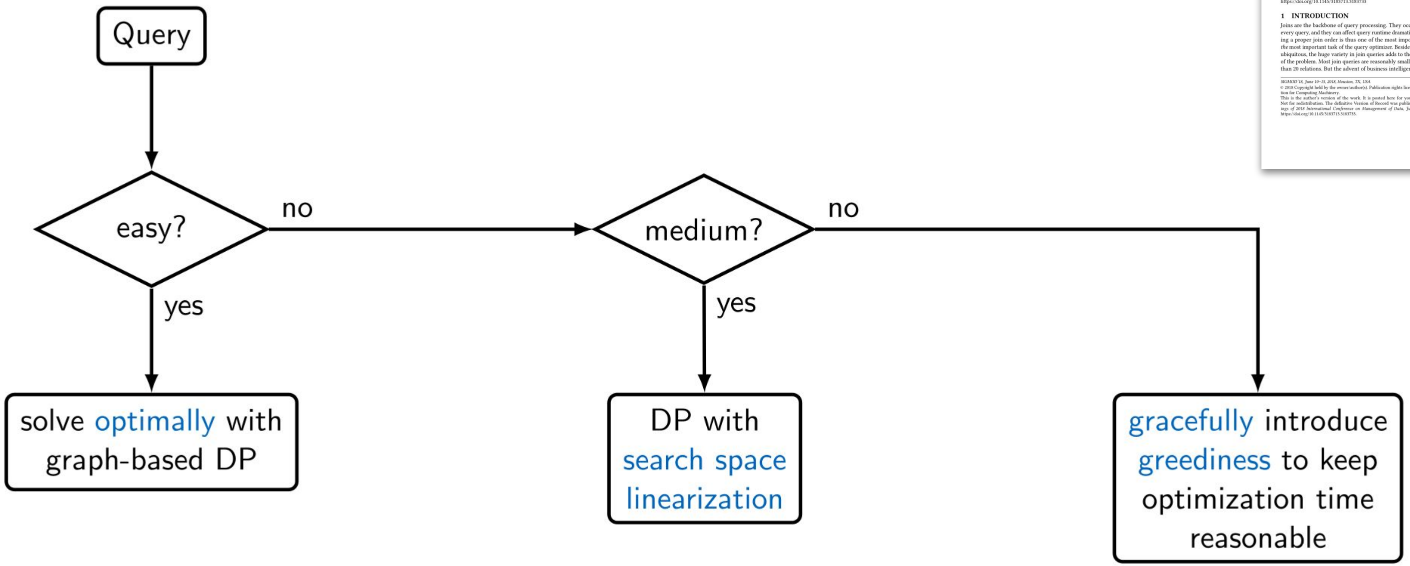
CedarDB

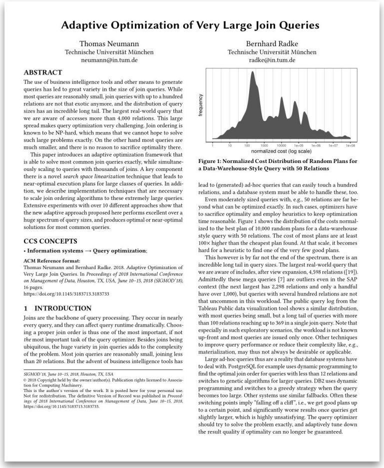
Join Ordering

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

Join Ordering

- Mostly Hash Joins





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.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of 2018 International Conference on Management of Data, June 10–15, 2018*, https://doi.org/10.1145/3183713.3183733.

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

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

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.

CedarDB

Join Ordering

- Mostly Hash Joins
- Distinct count estimates with Pat Selinger's equations:

  column1 = column2

  F = 1 / MAX(ICARD(column1), ICARD(column2))
- HyperLogLog intersections
- Mean & stddev approximations for `l_quantity < 0.2 * avg(l_quantity)`

CedarDB

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.

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

CedarDB

Physical Optimization

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

CedarDB

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

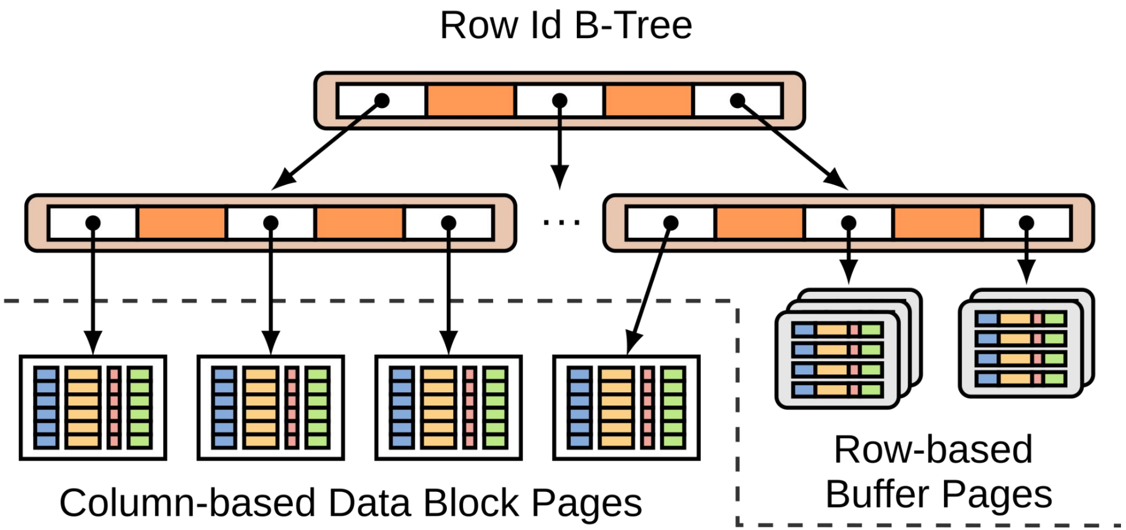
- Recap: DBMS Components
- Relational Algebra Optimization
- **Storage: B-Tree deep dive**

CedarDB

B-Trees

Hybrid storage engine:

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

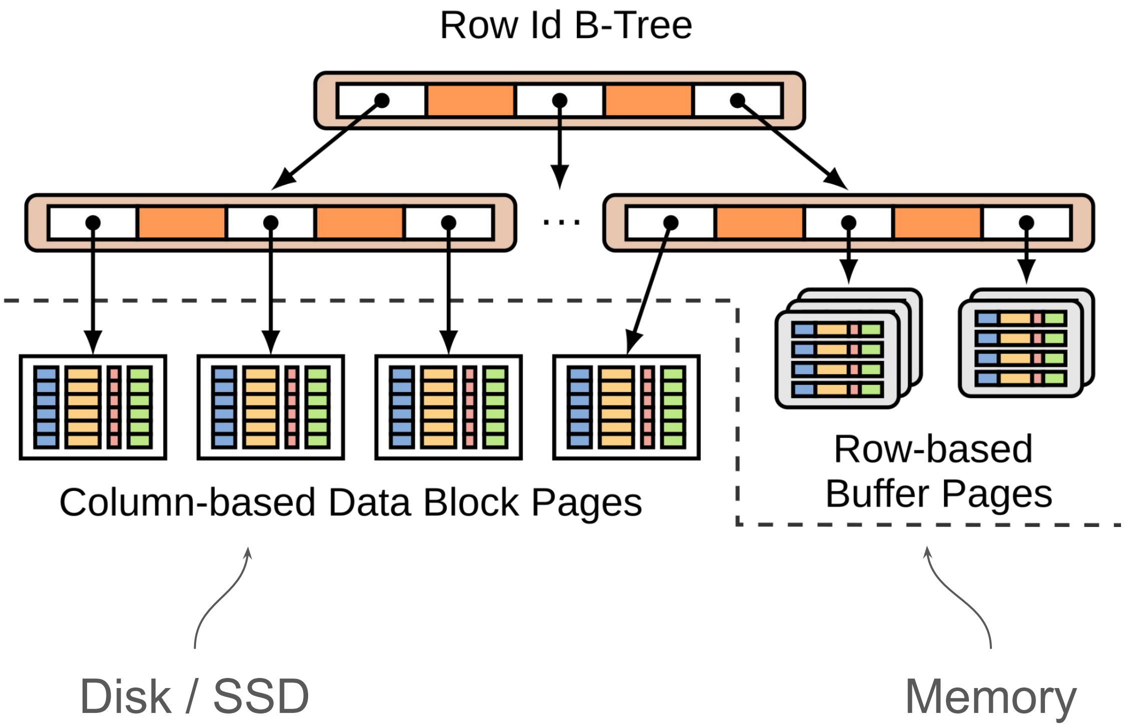


CedarDB

B-Trees

Hybrid storage engine:

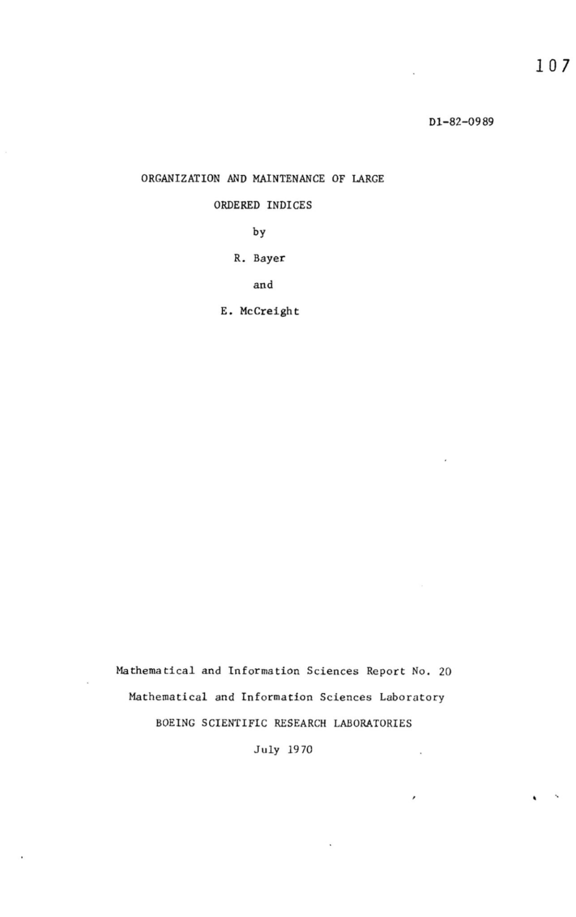
- Row oriented
- Columnar storage
- Hybrid structure
- Best of both worlds
- Hot writes at end



CedarDB

B-Trees

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



107

D1-82-0989

ORGANIZATION AND MAINTENANCE OF LARGE

ORDERED INDICES

by

R. Bayer

and

E. McCreight

Mathematical and Information Sciences Report No. 20

Mathematical and Information Sciences Laboratory

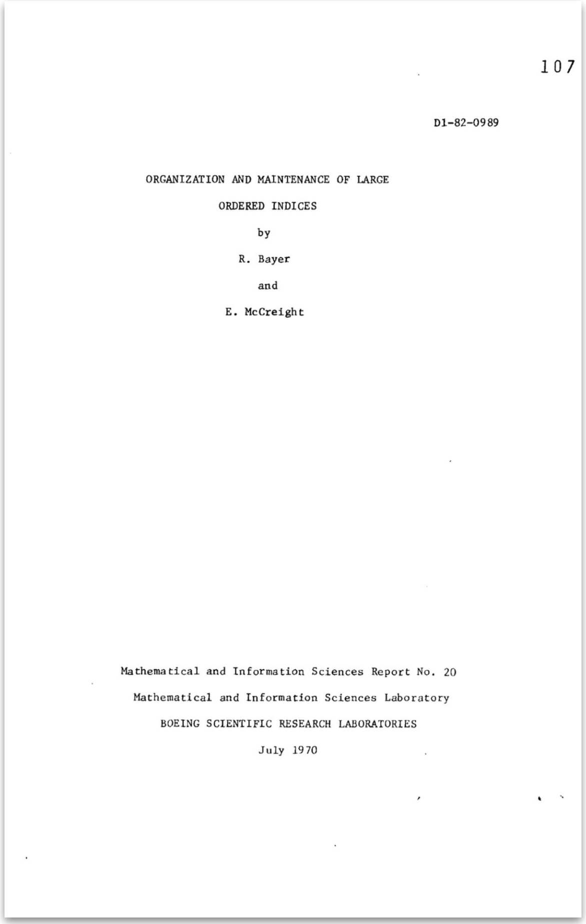
BOEING SCIENTIFIC RESEARCH LABORATORIES

July 1970

CedarDB

B-Trees

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



107

D1-82-0989

ORGANIZATION AND MAINTENANCE OF LARGE

ORDERED INDICES

by

R. Bayer

and

E. McCreight

Mathematical and Information Sciences Report No. 20

Mathematical and Information Sciences Laboratory

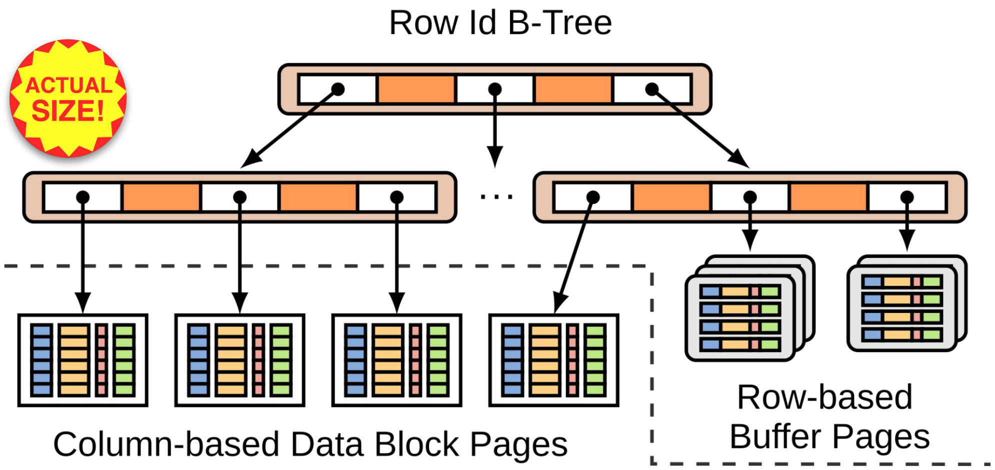
BOEING SCIENTIFIC RESEARCH LABORATORIES

July 1970

CedarDB

B-Trees

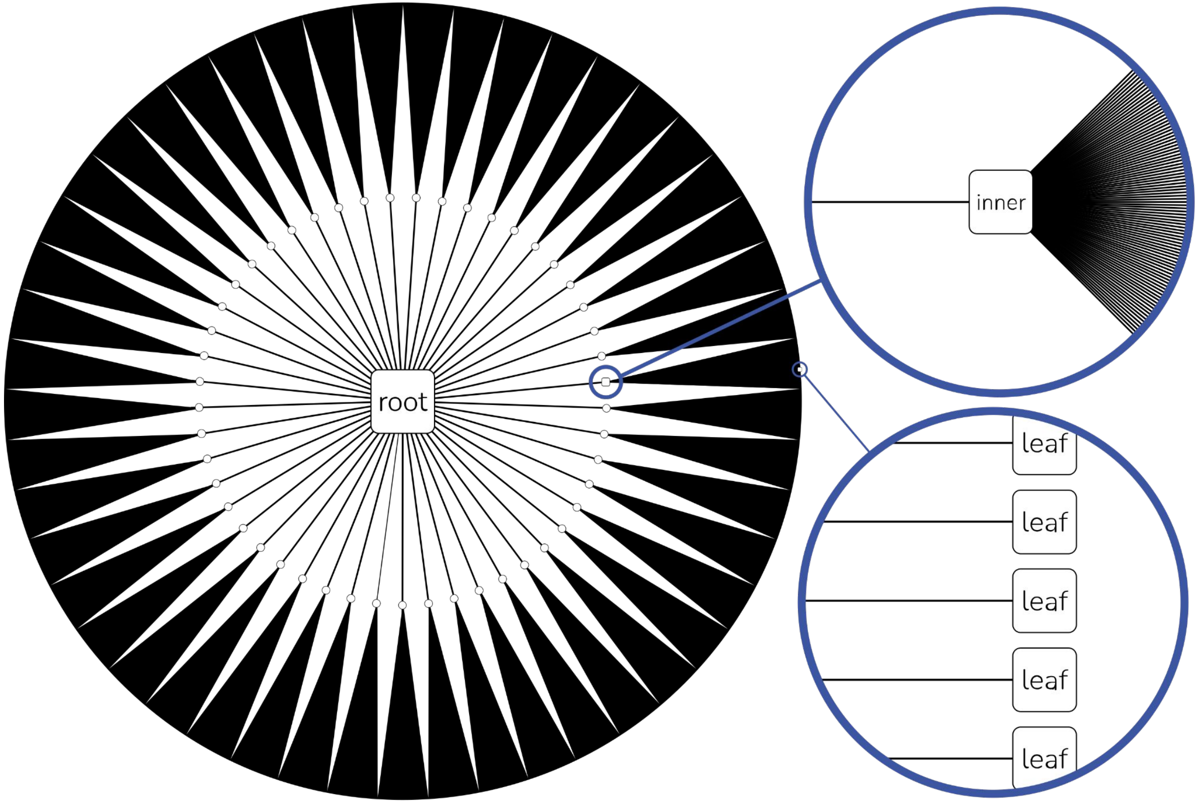
- Example dataset: ClickBench hits
- 70GB, 100M rows
- 3 levels
  - Fanout: 50 * 1500 * 1500



CedarDB

B-Trees

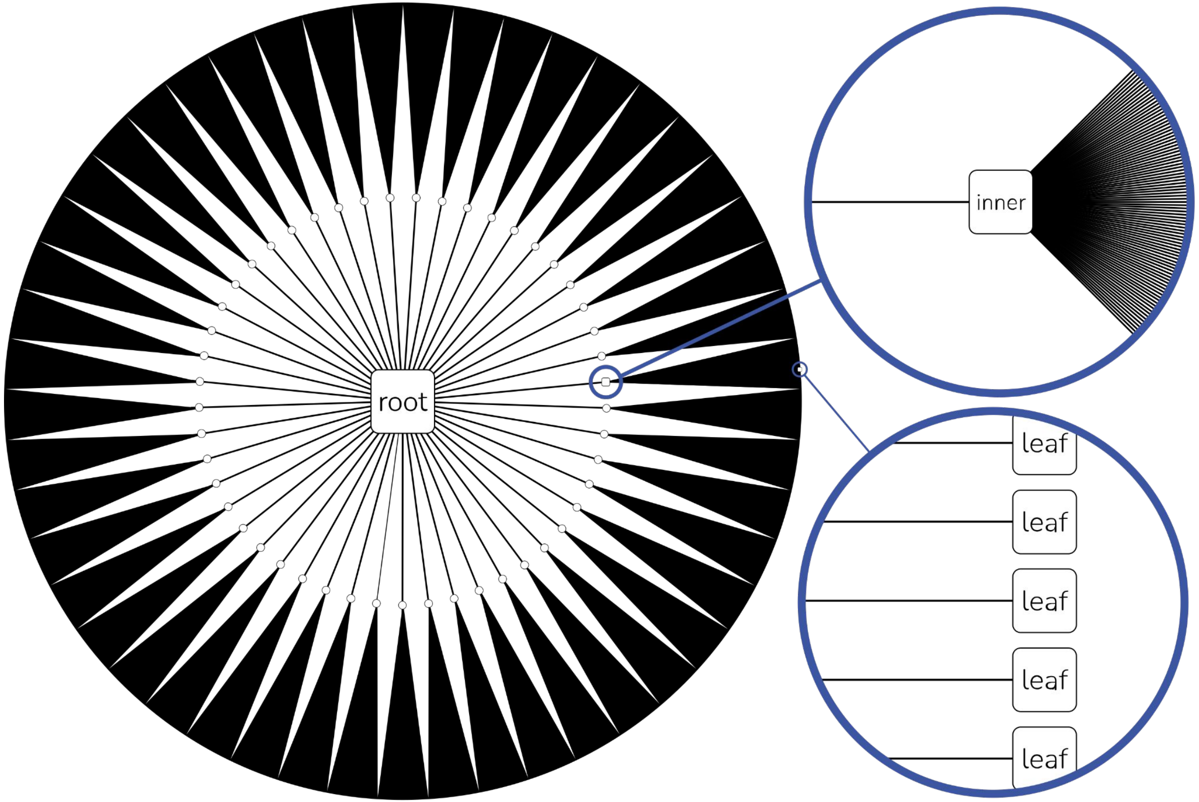
- Example dataset: ClickBench hits
- 70GB, 100M rows
- 3 levels
  - Fanout: 50 * 1500 * 1500



CedarDB

B-Trees

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

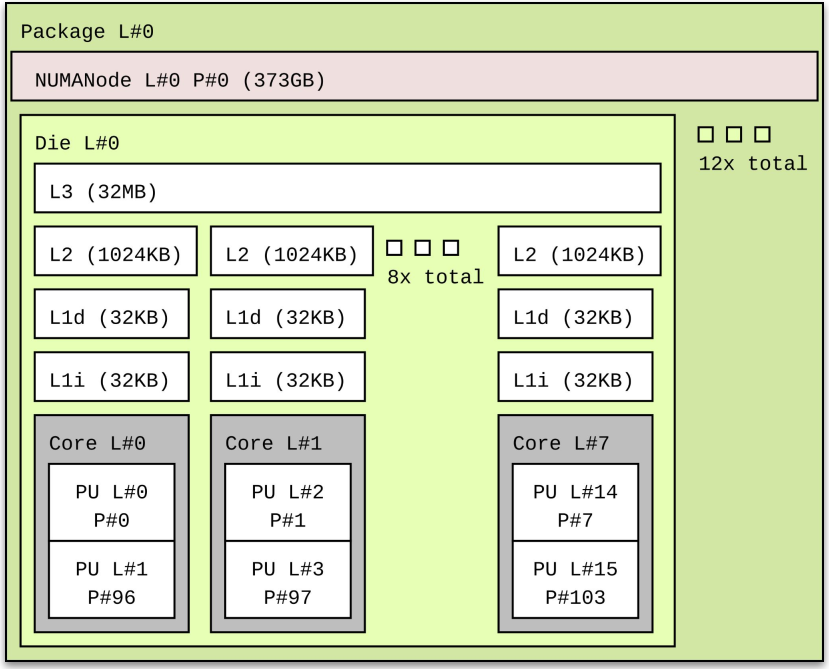


CedarDB

On Modern Hardware

Cache efficiency

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

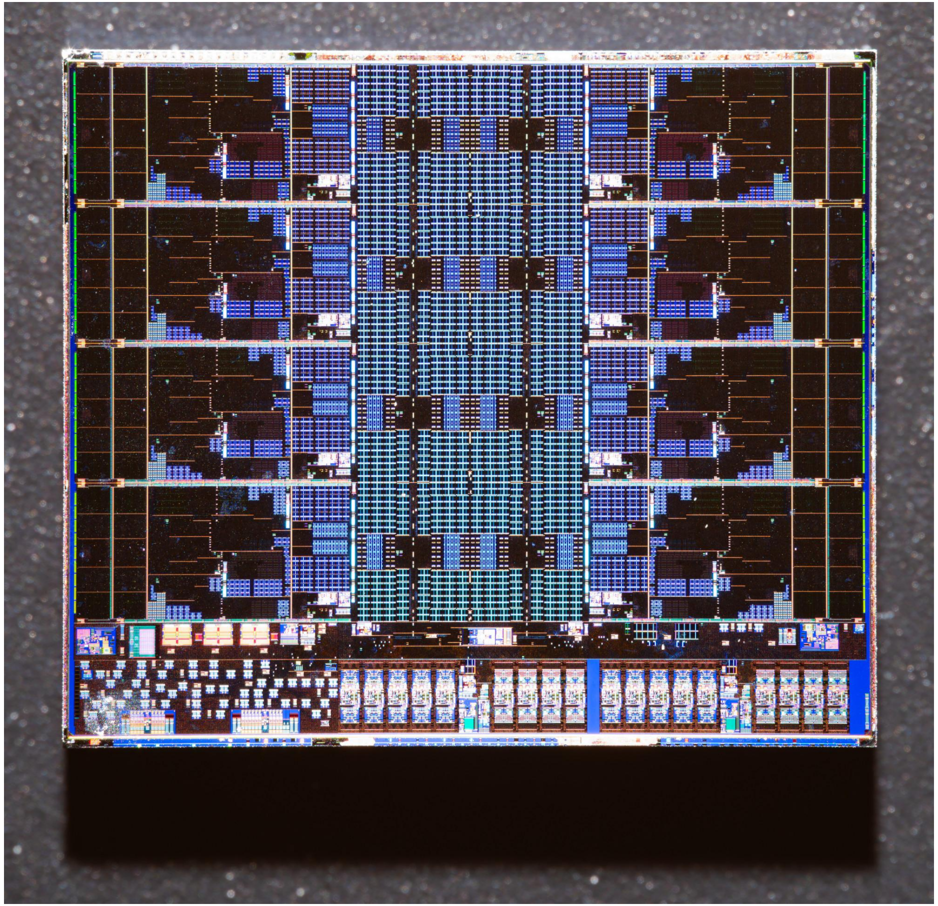


CedarDB

On Modern Hardware

Cache efficiency

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



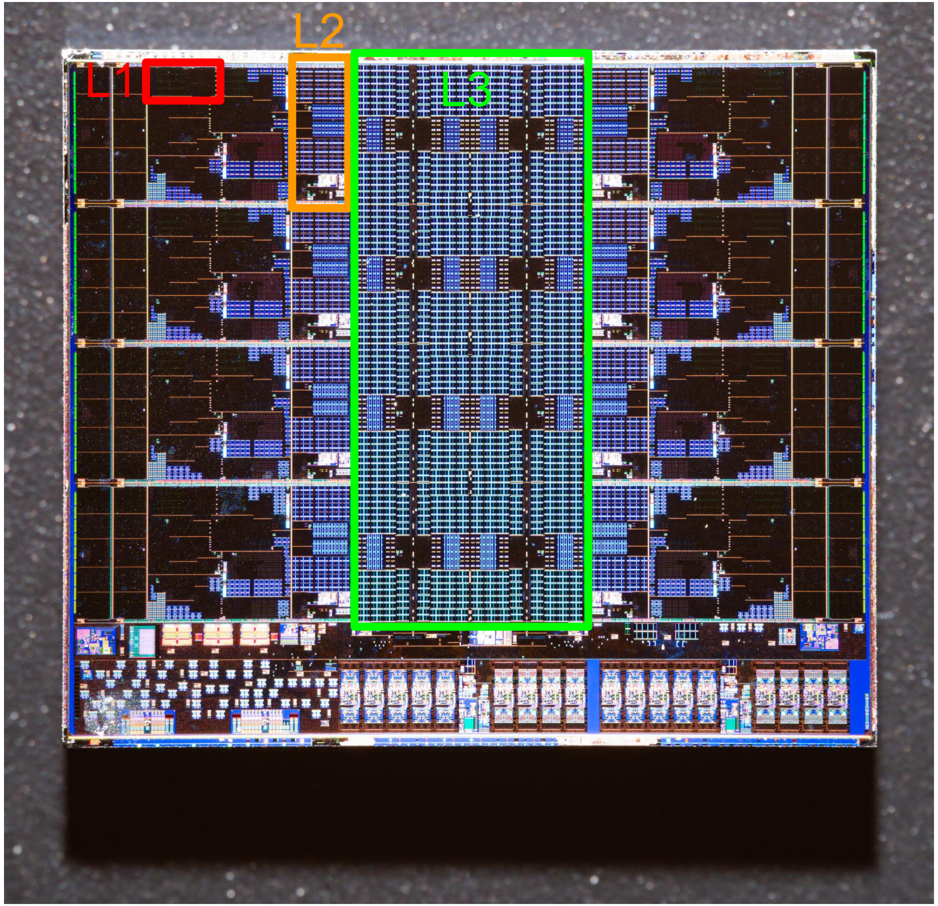
CedarDB

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

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

```
1. lock   A
2. access A
```

CedarDB

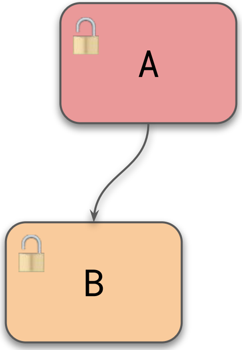
Lock Coupling on Modern Hardware

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
```



CedarDB

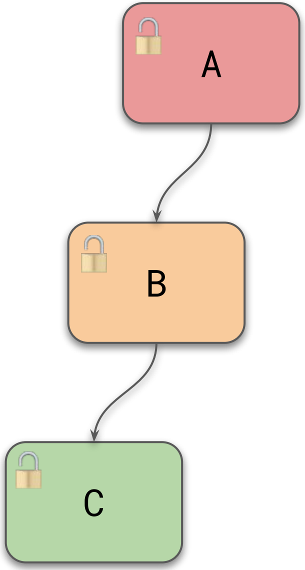
Lock Coupling on Modern Hardware

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
```



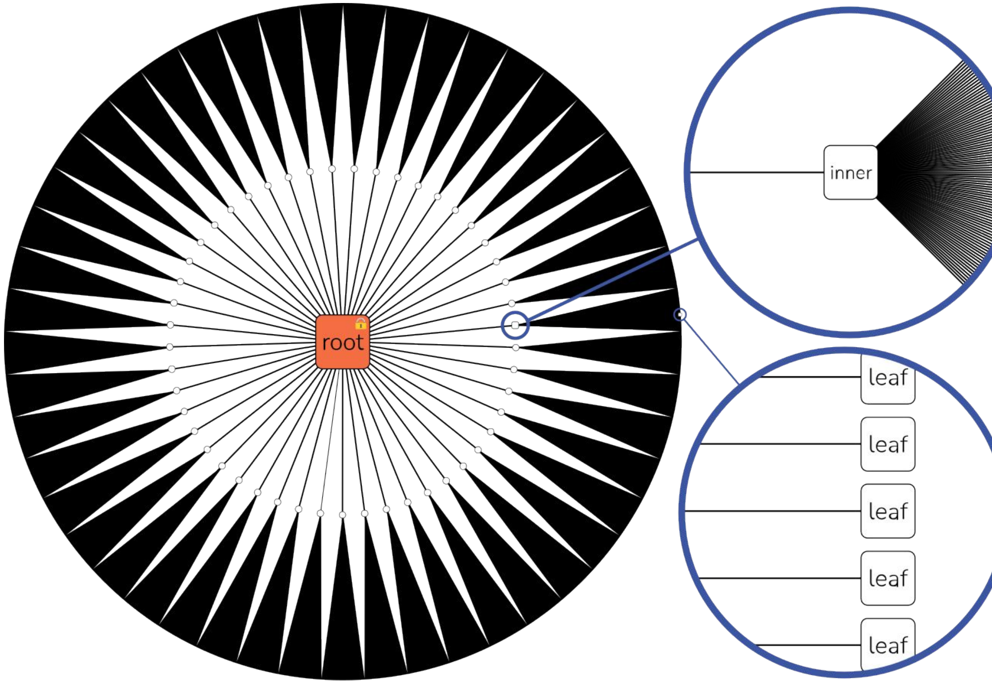
CedarDB

Lock Coupling on Modern Hardware

Problem:

Synchronization over 100s of cores

- Lock coupling
- All accesses through root node

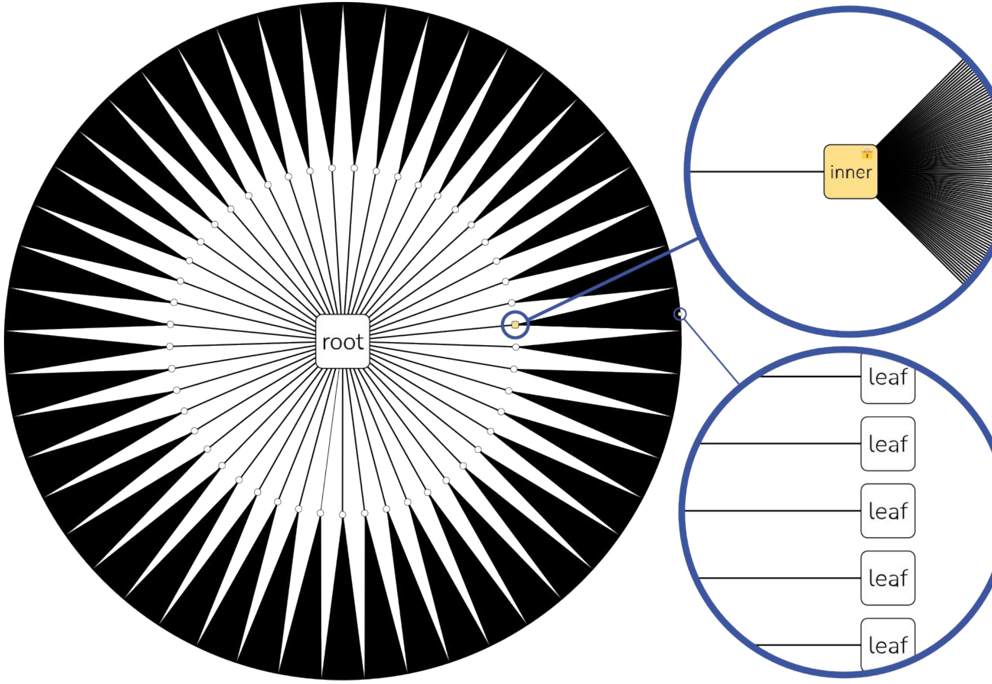


Lock Coupling on Modern Hardware

Problem:

Synchronization over 100s of cores

- Lock coupling
- All accesses through root node



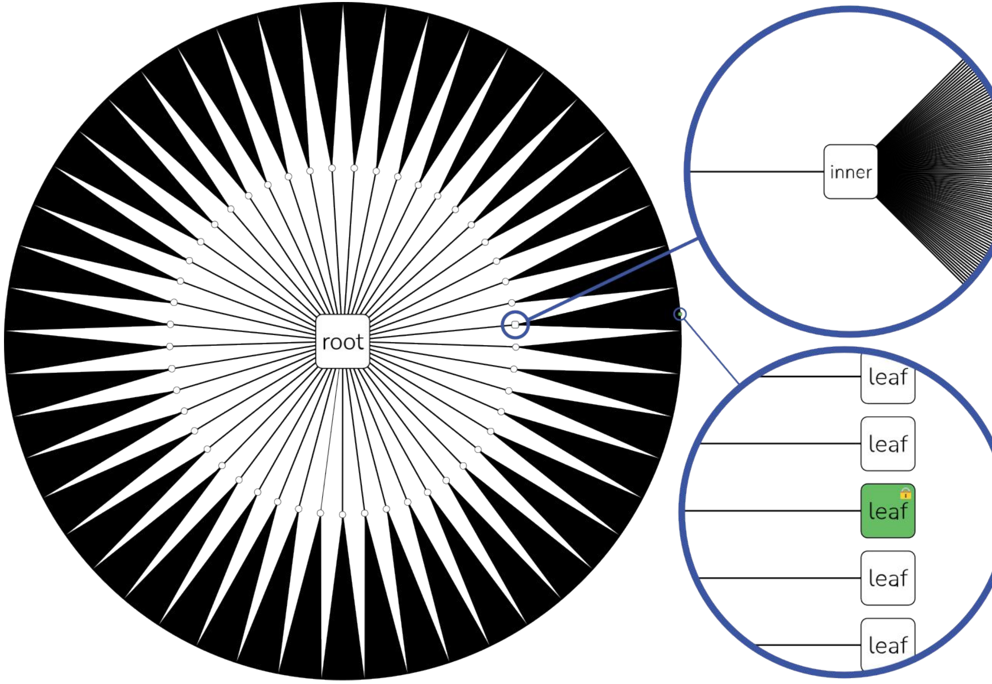
CedarDB

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



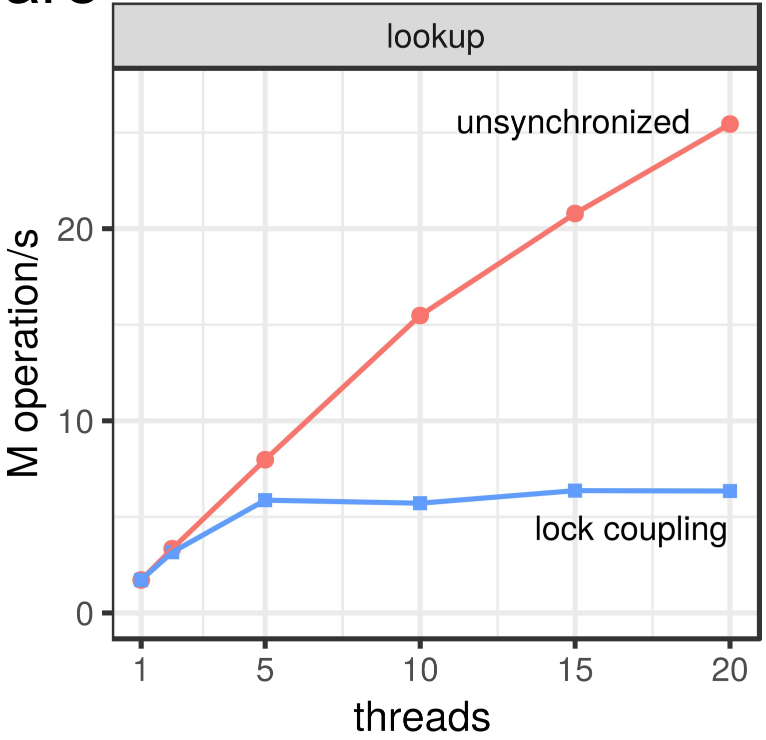
CedarDB

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**



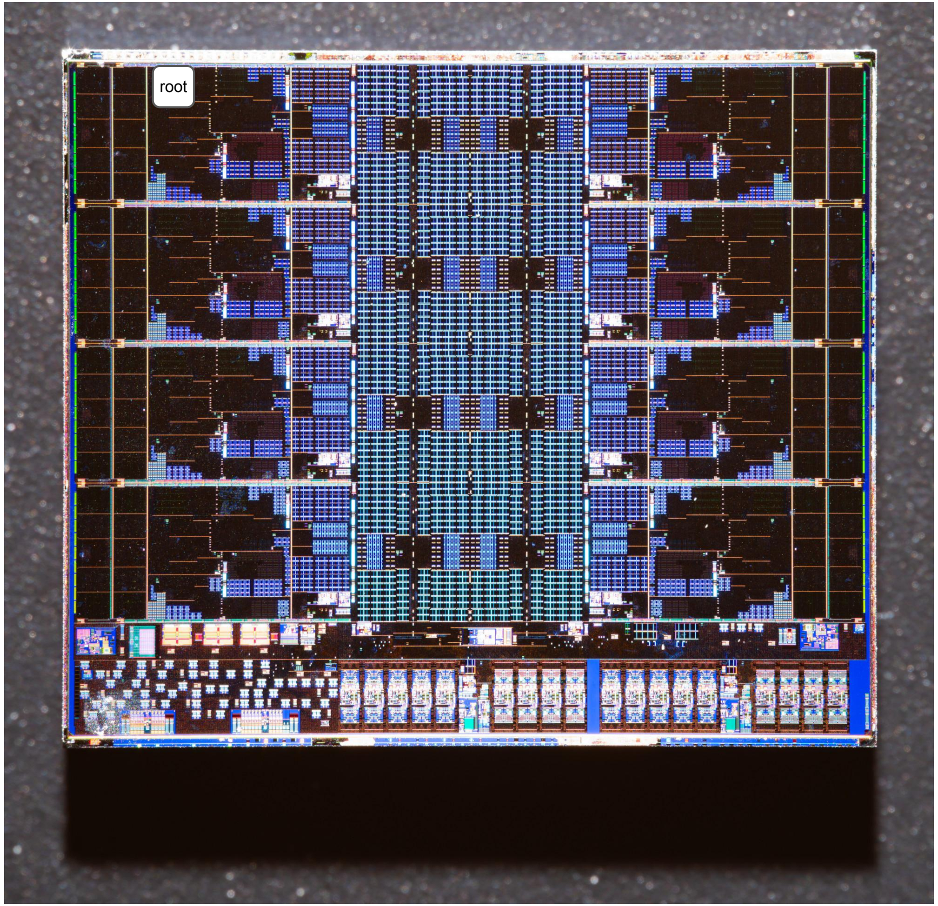
CedarDB

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**



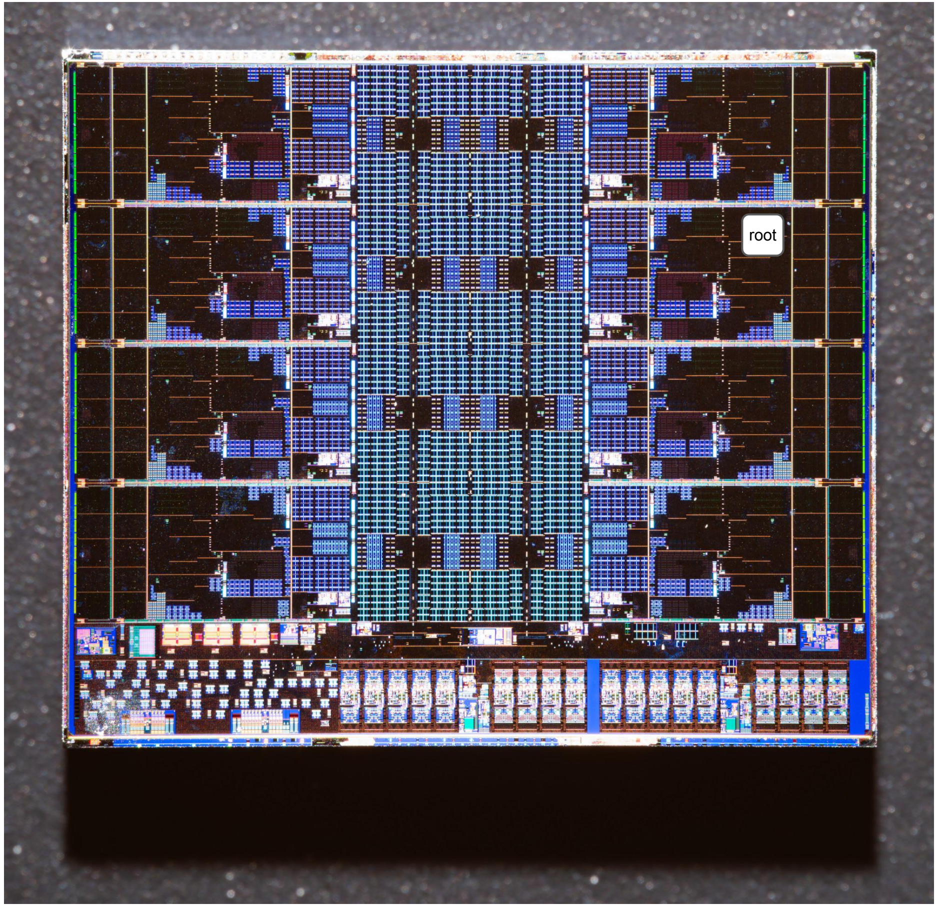
CedarDB

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**



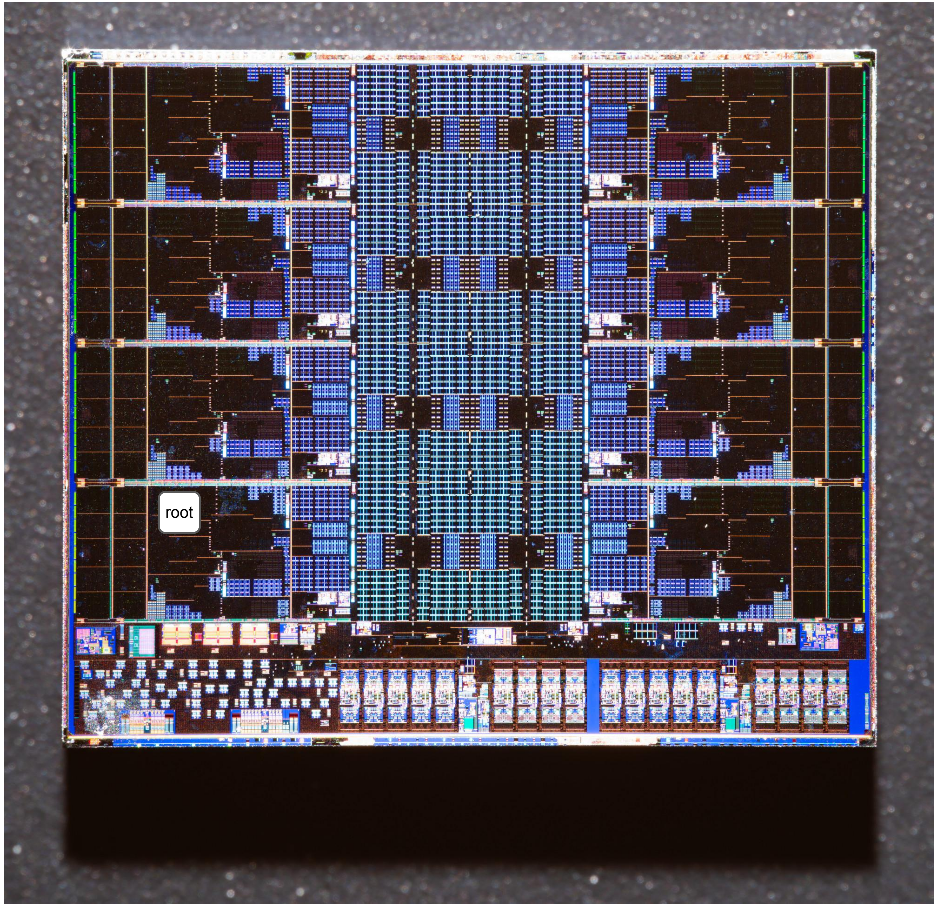
CedarDB

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

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

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

CedarDB

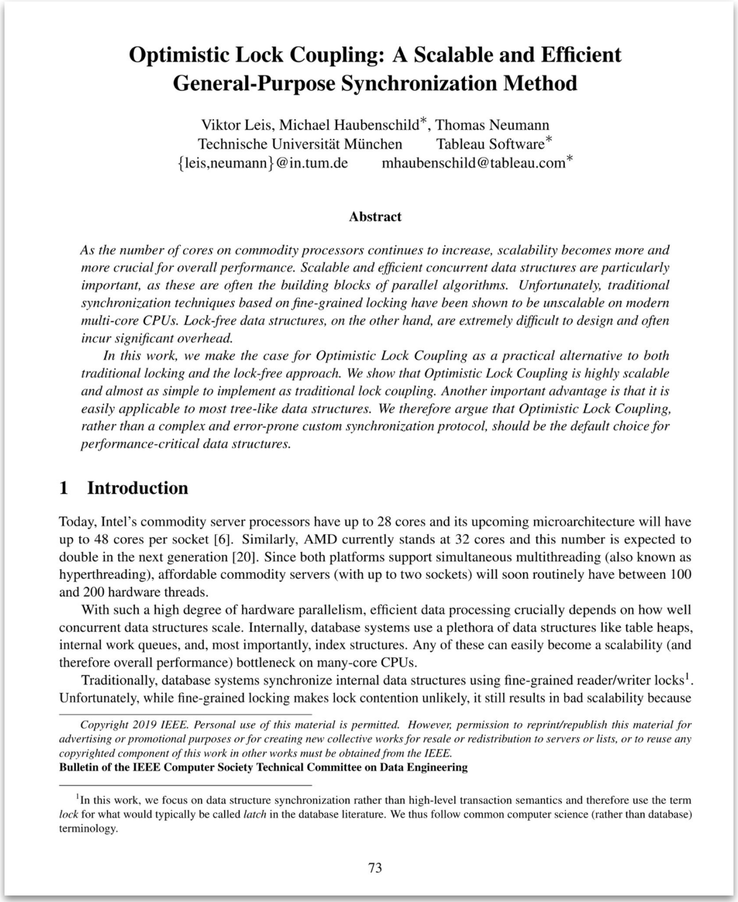
Optimistic Lock Coupling

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: 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[1]. Unfortunately, while fine-grained locking makes lock contention unlikely, it still results in bad scalability because

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] 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.

73

CedarDB

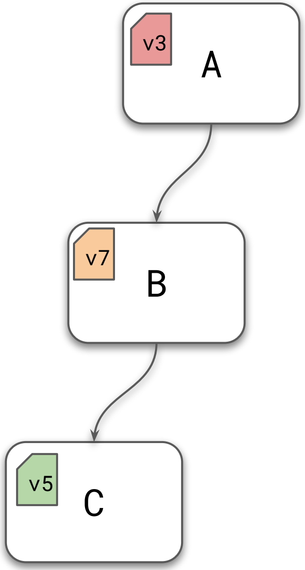
Optimistic Lock Coupling

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

CedarDB

Optimistic Lock Coupling

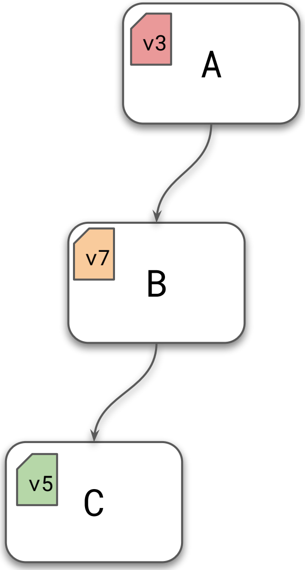
Lock Coupling:

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

6 atomic writes



Optimistic:

1. read v3
2. access A

3. read v7
4. validate v3
5. access B

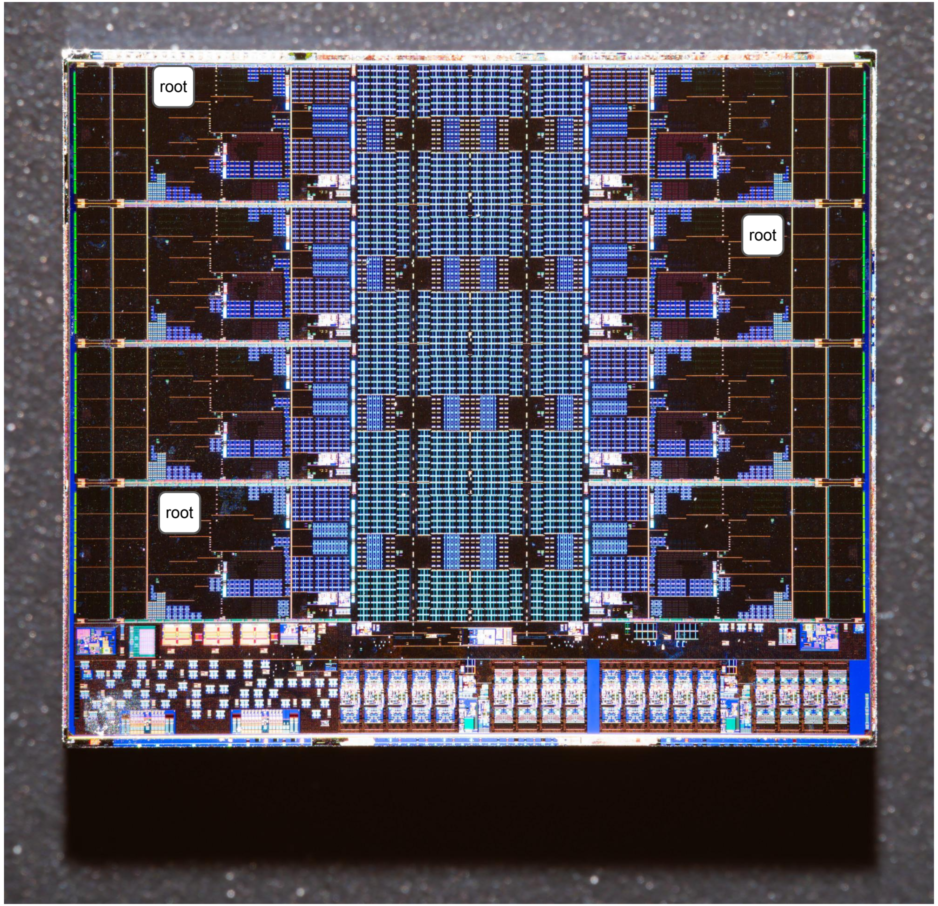
6. read v5
7. validate v7
8. access C
9. validate v5

read only

CedarDB

Optimistic Lock Coupling

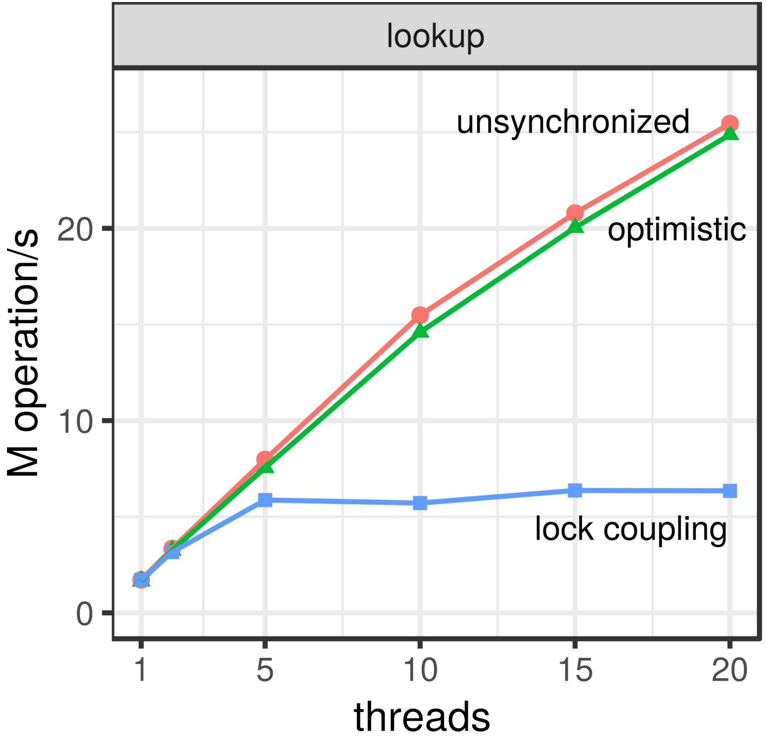
- Shared data
- No contention
- Less memory traffic



CedarDB

Optimistic Lock Coupling

Much better scalability!

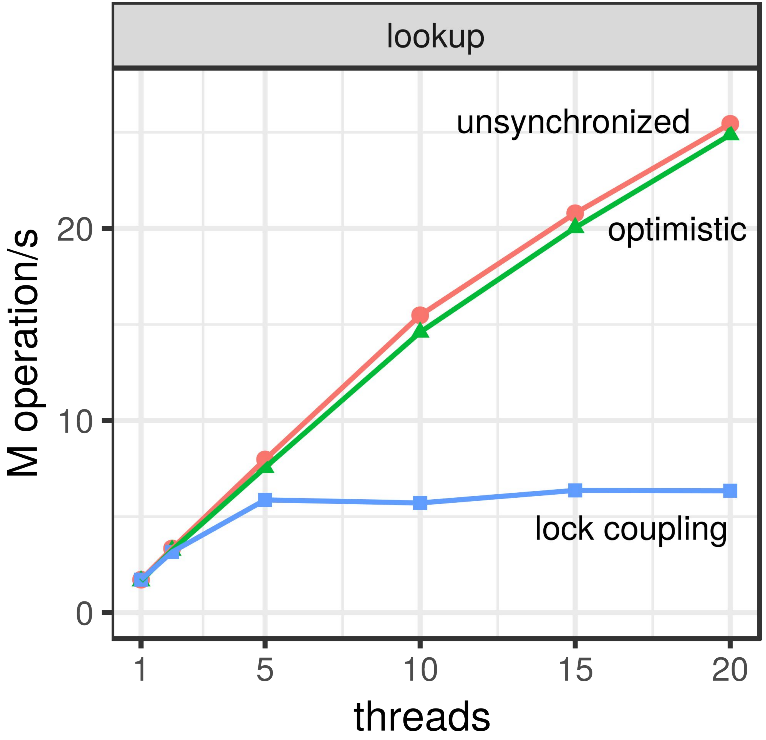


Optimistic Lock Coupling

Much better scalability!

- Practically lock free
- Practically cache oblivious

But: Still rarely used



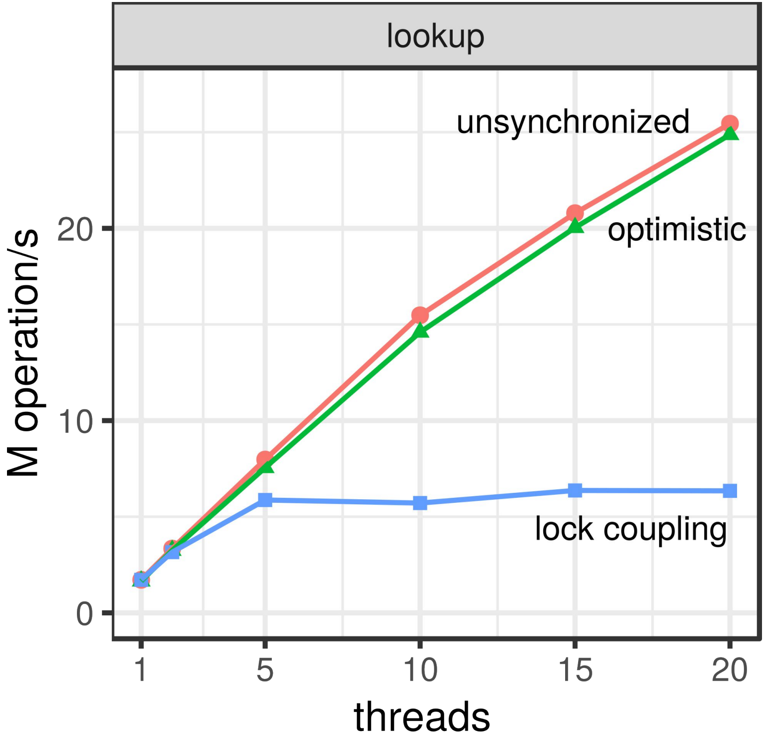
Optimistic Lock Coupling

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



CedarDB

Try it now:

Free Community Edition for Linux / Docker:

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

philipp@cedardb.com