Umbra

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

TPC-H SF10

- Blue: Postgres
- Orange: DuckDB
- Green: Umbra
What makes Umbra fast?
What makes Umbra fast?

● Pipelined execution
  ○ Keeps values in registers
  ○ Minimizes materialization
What makes Umbra fast?

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

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

- Pipelined execution
- Data-centric code generation
- Fully parallel algorithms
  - Allows scaling
  - Benefits from new hardware
What makes Umbra fast?

- Pipelined execution
- Data-centric code generation
- Fully parallel algorithms
- State-of-the-art query optimizer
What makes Umbra fast?

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

Research system with all custom advanced parts

We’re commercializing soon!
Query Optimization

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

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

Cost-based Optimization
Expression Simplification

- Fold constants
- Canonicalize expressions

\[
\begin{align*}
o_{\text{orderdate}} & \geq \text{date '1994-01-01'} \\
\text{and } o_{\text{orderdate}} & < \text{date '1994-01-01'} + \text{interval '1'} \text{ year} \\
\end{align*}
\]

\[
\begin{align*}
\text{between date '1994-01-01' and date '1994-12-31'}
\end{align*}
\]

- Execute in evaluation engine
Query Unnesting & Decorrelation

- Unnesting Arbitrary Queries

Unnesting Arbitrary Queries

Thomae Neumann and Aliaks Krueger
Technische Universität München

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 concreted with the same queries frequently lead to dependent plans with nested loop evaluations and thus poor performance. Existing systems therefore use a number of heuristics to unnest these queries, i.e., unnesting is difficult. Those unnesting heuristics are partly based on queryprocessing but are usually limited to certain classes of queries. To the best of our knowledge, no existing system can unnest queries in the general case. We present a generic approach for unnesting arbitrary queries. As a result, the de-unnest 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: (id, course, credits, date, ...)  

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

```sql
Q1: select s.name, e.course  
    from students s, exam e  
    where s.id = e.student  
    and e.grade = (select max(e2.grade)  
                   from exams e2  
                   where s.id = e2.student)
```

Consequently, for each student, exam pair is $s.e$ a determinant 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...

Blog

2023-05-26 Mark Rosendahl

Correlated Subqueries in SQL

Subqueries in SQL are a powerful abstraction that allow simple queries to be used as compositional 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 decoration 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. Unrelated 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(n²) -> 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

\[
c_{\text{nationkey}} = s_{\text{nationkey}} \\
\text{and } s_{\text{nationkey}} = n_{\text{nationkey}}
\]

\[
==
\]

\[
c_{\text{nationkey}} = s_{\text{nationkey}} \\
\text{and } s_{\text{nationkey}} = n_{\text{nationkey}} \\
\text{and } c_{\text{nationkey}} = n_{\text{nationkey}}
\]
Initial Join Tree

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

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

==

```sql
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 $\sigma$ 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

```python
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

```python
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
```

```python
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
```

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

Variant C: Combined branch
```

<table>
<thead>
<tr>
<th>Variant</th>
<th>branch-misses</th>
<th>instructions</th>
<th>loads</th>
<th>exec. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.63 / tpl</td>
<td>7.62 / tpl</td>
<td>2.85 / tpl</td>
<td>18.4 ms</td>
</tr>
<tr>
<td>B</td>
<td>0.58 / tpl</td>
<td>7.91 / tpl</td>
<td>3.00 / tpl</td>
<td>17.7 ms</td>
</tr>
<tr>
<td>C</td>
<td>0.13 / tpl</td>
<td>11.67 / tpl</td>
<td>3.37 / tpl</td>
<td><strong>12.7 ms</strong></td>
</tr>
</tbody>
</table>
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**

**Abstract**

Every year more and more advanced approaches to cardinality estimation are published, using learned models or in other data and worlded specific synopsis. To contrast, the majority of commercial in-memory database systems still rely on sampling. It is arguably the most general and robust estimate to implement. While most methods do not seem to improve much over sampling based estimates 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 triple-qualities optimizes 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 low selectivities. It is ready to use in any DBMS capable of sampling, showing a negligible impact on optimization time. One experiment on real-world and synthetic data demonstrated up to two orders of magnitude reduced estimate errors. Formulating single filter predicates according to our estimate results 1 to 10 times better query responses for complex filters.

**ACM Reference Format**


1 INTRODUCTION

Cardinality estimates guide query optimizers towards correct execution plans and lower the risk of incorrect plans [25, 26]. Although many approaches were published on cardinality estimation, e.g., using histograms [39], sampling [11], or machine learning [13], it is still considered a great challenge [29]. Especially statistical models remain challenging especially when we analyze a multitude of correlated filter predicates. The comprehensive analysis of the real-world RDB data repositories by Yingpeng Gu et al. [38] underline the importance of filter operations and tuples. Hard data is stored in string format, which enables arbitrary string expressions.
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'

```
Early Execution

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

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

==

select 3
```
Join Ordering

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

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

- Query
  - easy?
    - yes: solve optimally with graph-based DP
    - no: medium?
  - medium?
    - yes: DP with search space linearization
    - no: gracefully introduce greediness to keep optimization time reasonable
Join Ordering

- Hash Joins rule
  - Indexes don’t allow bushy plans -> less useful
- Distinct count estimates with Pat Sellinger’s equations
- HyperLogLog intersections
- Mean & stddev approximations for $l_{quantity} < 0.2 \times \text{avg}(l_{quantity})$
Early Probing

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

- Indexes
- Worst-case optimal join
Physical Optimization

- Indexes
- Worst-case optimal join
- Groupjoin
Physical Optimization

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

● Query compilation & optimization
  ○ Optimizer passes
  ○ Rule-based canonicalization
  ○ Cost-based optimization

● Cutting-edge research
  ○ Join ordering
  ○ Cardinality estimation
  ○ Integrated in a running system
Conclusion

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

We are commercializing

Reach out:

fent@in.tum.de