Generalized Sub-Query Fusion for Eliminating Redundant I/O from Big-Data Queries

Kaushik Rajan,
Partho Sarthi, Akash Lal (Microsoft Research)
Abhishek Modi, Ashit Gosalia, Prakhar Jain, Mo Liu, (Microsoft)
Saurabh Kalikar (Intel, interned at Microsoft Research)
Big data query compilation

Query compilation and execution in big-data systems (Spark, Hadoop, Snowflake, Amazon Redshift, Google BigQuery, Azure Synapse)
Big data query compilation

Exchanges expensive as they induce disk and network I/O

Plans with fewer stages preferable
Redundant stages of processing

- TPCDS, 40% of queries have redundant I/O
- 16% of all queries, High-impact spend at least 50% time on stages with redundant I/O
- 9% medium impact, spend 10-50% time on stages with redundant I/O

Redundancy analysis in SPARK on TPCDS
RESIN: MapReduce reasoning during optimization

SQL → SQL rewrites
Logical plan
SQL PhyOps, exchanges
Physical plan

State-of-the-art
RESIN

SQL → SQL+MR
Logical plan
MR ops
Physical plan

SQL → SQL rewrites
Logical plan
SQL PhyOps, exchanges
Physical plan

Produce plans with fewer stages
RESIN: MapReduce reasoning during optimization

SQL → SQL rewrites
Logical plan
SQL PhyOps, exchanges
Physical plan

State-of-the-art
RESIN

SQL→SQL+MR
Logical plan
MR ops
Physical plan

1. Subquery Fusion
2. Binary elimination

RESINMap
ResinReduce

Produce plans with fewer stages
Impact of RESIN on I/O and Memory

![Graphs showing cumulative disk I/O and CPU task time for queries in increasing order of impact. The graphs compare baseline I/O and resin I/O, as well as baseline task cost and resin task cost.]
Rest of the talk

1. **ResinMap** and **ResinReduce**
2. Generalized sub-query fusion
3. Implementation on Spark
4. Experimental evaluation
T1 = SELECT id, hr ← hr_1, signal ← signal_1 FROM rawLogs WHERE hr_1 ≥ 0 ∧ hr_1 < 24 ∧ signal_1 ≥ 0

T2 = SELECT id, hr ← hr_2, signal ← signal_2 FROM rawLogs WHERE hr_2 ≥ 0 ∧ hr_2 < 24 ∧ signal_2 ≥ 0

signals = SELECT * FROM T1 UNION ALL SELECT * FROM T2

ResinMap:
/*1*/ Filter(hr_1 ≥ 0 ∧ hr_1 < 24 ∧ signal_1 ≥ 0), Cols(id, hr ← hr_1, signal ← signal_1)
/*2*/ Filter(hr_2 ≥ 0 ∧ hr_2 < 24 ∧ signal_2 ≥ 0), Cols(id, hr ← hr_2, signal ← signal_2)

signals

(a) SQL Query

(b) Standard execution plan

(c) Optimized execution plan

A row-wise operator, can produce multiple output rows per input row.
A row-wise operator, can produce multiple output rows per input row.
A row-wise operator, can produce multiple output rows per input row

Method ResinMap(m) {
    foreach(id, hr1, signal1, hr2, signal2) ∈ iotLogs[m] {
        if(hr1 ≥ 0 ∧ hr1 < 24 ∧ signal1 ≥ 0) {
            hr = hr1; signal = signal1; output(id, hr, signal)
        }
        if(hr2 ≥ 0 ∧ hr2 < 24 ∧ signal2 ≥ 0) {
            hr = hr2; signal = signal2; output(id, hr, signal)
        }
    }
}

// Each mapper m processes a partition rawlogs[m]

ResinMap[/*1*/{Filter(hr1 ≥ 0 ∧ hr1 < 24 ∧ signal1 ≥ 0)), Cols(id, hr ← hr1, signal ← signal1)}],
          /*2*/{Filter(hr2 ≥ 0 ∧ hr2 < 24 ∧ signal2 ≥ 0), Cols(id, hr ← hr2, signal ← signal2)}]

Single table select, project, union queries in one stage
ResinReduce

Key based operator, process rows sharing key, produce one row

Join \((c_1 = c_2)\)

- Project\((c_1 \leftarrow id, s_1)\)
- Project\((c_2 \leftarrow id, s_2)\)
- GroupBy\(id, s_1 \leftarrow \max(signal_1)\)
- GroupBy\(id, s_2 \leftarrow \min(signal_2)\)
- Project\((id, signal_1)\)
- Project\((id, signal_2)\)
- Filter \((hr_1 \leq 12)\)
- Filter \((hr_2 \leq 18)\)

\[\text{ResinReduce}\left[\{key = id\}, \begin{array}{l} /*1*/\{\text{filter}(hr_1 \leq 12)), \text{aggregate}(s_1 \leftarrow \max(signal_1), rc_1 \leftarrow \text{count}())\} \\ /*2*/\{\text{filter}(hr_2 \leq 18)\text{aggregate }(s_2 \leftarrow \min(signal_2)) rc_2 \leftarrow \text{count}())\}\end{array}\right] \]

Filter \((hr_1 \leq 12 \lor hr_2 \leq 18)\)

Eliminate multiple shuffles from single table join queries
Sub-query fusion

Eliminate scans/shuffles from multi-table queries
Sub-query fusion

Eliminate scans/shuffles from multi-stage queries
Sub-query fusion

Eliminate scans/shuffles from multi-stage queries
Sub-query fusion

(S7) Project(c₁ ← city, s₁)
   GroupBy(city, s₁←max(signal))
   Filter(φ₁ ∧ φ₃)
   Project(city, signal, hr, ht)
   Filter(φ₁ V φ₂(hr))
   Scan signals

(S8) Project(c₂ ← city, s₂)
   GroupBy(city, s₂←max(signal))
   Filter(φ₂ ∧ φ₄)
   Project(city, d_id, ht)
   Filter(φ₃ V φ₄(ht))
   Scan dInfo

(S9) Join (c₁ = c₂)
   Filter(φ₁ ∧ φ₃)
   Project(city, s₁)
   Filter(φ₂ ∧ φ₄)
   Project(city, s₂)

(S13) Filter(rc₁ > 0 ∧ rc₂ > 0)

ResinReduce[(key = city),
/*1*/{filter(φ₁ ∧ φ₃), aggregate(s₁ ← max(signal), rc₁ ← count(*))}
/*2*/{ filter(φ₂ ∧ φ₄) aggregate(s₂ ← max(signal) rc₂ ← count(*))}]

Eliminate scans/shuffles from multi-stage queries
Sub-query fusion

Eliminate scans/shuffles from multi-table queries
In the paper

• Parameters for ResinMap and ResinReduce operators, semantics and implementation
• Fusing of operators without increasing the number of rows shuffled
• Fusion rules for all sparkSQL operators, conditions under which fusion is possible
Rest of the talk

1. ResinMap and ResinReduce
2. Generalized sub-query fusion
3. Implementation on Spark
4. Experimental evaluation
Implementation

Implemented RESIN on catalyst optimizer in SPARK 2.4

1. Added logical and physical operators for ResinMap and ResinReduce

2. Added a new batch of optimization rules
   • Perform fusion in a single traversal of the tree
   • Perform Union and Join elimination by checking fused parent
   • Introduce exchanges if parent after fusion cannot be eliminated

3. Added implementations for our operators with codegen support
Evaluation

• Evaluated with TPCDS at 1TB and 10TB scale, data stored in parquet
• Two different clusters <120 cores, 480 GB memory> and <480 cores, 1.6TB memory>
• Detailed evaluation of 40 (out of 104) queries with redundant I/O
• Note: baseline that already has basic I/O optimizations (predicate and project pushdown to store, exchange reuse)
Speedup at 1TB
Impact of RESIN on I/O and Memory

**DISK**

- Cumulative Disk I/O (bytes)
- Queries in increasing order of impact
- Baseline I/O (blue line)
- RESIN I/O (red line)

**MEMORY**

- Cumulative Memory Footprint (bytes)
- Queries in increasing order of impact
- Baseline memory used (blue line)
- RESIN memory used (red line)
Conclusions

• Big-data optimizers produce plans with redundant I/O and compute
• Proposed optimizer extensions to perform first class map-reduce reasoning
• Added generic map and reduce operators, rewrites that fuse stages and eliminate redundant I/O
• Demonstrated savings in terms of latency, disk and network I/O
Thank You

Email krajan@microsoft.com or any of the other authors to contact us