An Overview on Optimization in Apache Hive: Past, Present, Future

Jesús Camacho Rodríguez
Apache Calcite committer, PMC Chair
Apache Hive committer, PMC Member
jcamacho@apache.org

Apache Big Data Europe
November 14, 2016
Acknowledgments

- **Apache Hive** and **Apache Calcite** communities
  - Ashutosh Chauhan, Julian Hyde, Pengcheng Xiong, Vineet Garg, John Pullokkaran, Harish Butani, Hari Sankar Sivarama Subramaniyan, Michael Mior, and many others

- Julian Hyde for sharing slides of previous optimizer talks
Agenda

An overview on optimization in Apache Hive

Introduction
Apache Hive

- Development started at Facebook in 2007
  - Open-sourced in 2008
  - First as a subproject of Apache Hadoop
  - Later became a top-level Apache project

- Initial use case: batch processing
  - Read-only data
  - HiveQL (SQL-like query language)
  - MapReduce

- Effort to take Hive beyond its batch processing roots
  - Started in Apache Hive 0.10.0 (January 2013)
  - Latest released version: Apache Hive 2.1.0 (June 2016)

- Extensive renovation to improve three different axes
  - Latency: allow interactive and sub-second queries
  - Scalability: from TB to PB of data
  - SQL support: move from HiveQL to SQL standard
Apache Hive

Important execution internals improvements

- Multiple execution engines: Apache Tez and Apache Spark
- Partition pruning
- More efficient join execution algorithms
- Vectorized query execution
  - Integration with columnar storage formats: Apache ORC, Apache Parquet
- LLAP (*Live Long and Process*)
  - Persistent daemons for low-latency queries
Apache Hive

Motivation for query optimizer

- Determine the most efficient way to execute a given query
  - Huge benefits for declarative query languages like SQL
  - Especially if queries are generated by BI tools

- **Challenging**: trade-off between plan generation latency and optimality

- Helps exploiting Apache Hive capabilities
  - Join reordering and most efficient algorithms selection, static and dynamic partition pruning, column pruning, etc.
An overview on optimization in Apache Hive

Introduction

Query optimizer overview
Query preparation

Before Apache Hive 0.14.0

✗ Optimization difficult to perform over AST
   – Missing opportunities
Query preparation

After Apache Hive 0.14.0

- Logical optimizer powered by Apache Calcite
  - Each new version of Hive tightens this integration
    - Improvements in the optimizer to be more effective and more efficient
    - Shifting logic from Apache Hive to Apache Calcite: predicate pushdown, constant folding, etc.

SQL -> SQL Parser -> AST -> Semantic Analyzer -> Logical Optimizer -> Optimized logical plan -> Physical Optimizer -> Physical plan -> Execution Engine
# Query optimization

## Optimizer improvements since Apache Hive 0.14.0

<table>
<thead>
<tr>
<th>Release</th>
<th>Date</th>
<th>Features (not exhaustive)</th>
</tr>
</thead>
</table>
| Apache Hive 0.14.0 | Nov 2014 | • Cost-based ordering of joins  
                      • Cost model improvements (expressions selectivity)  
                      • Extended coverage (bail out from Calcite optimizer if feature not supported)  
                      • Improvements to partition pruner |
| Apache Hive 1.1.0 | Mar 2015 | • Enable Apache Calcite optimization by default  
                      • More rules (identity project removal, union merge) |
| Apache Hive 1.2.0 | May 2015 | • Improvements in predicate inference (leads to more pruning opportunities)  
                      • More rules (predicate folding, transitive inference)  
                      • Chained predicate pushdown, inference, propagation, and folding |
| Apache Hive 2.0.0 (*) | Feb 2016 | • Apache Calcite rewriting rules run even without stats (subset)  
                      • Disable rules depending on input query profile  
                      • Improvements in metadata caching  
                      • More rules (sort and limit pushdown, aggregate pushdown) |
| Apache Hive 2.1.0 | Jun 2016 | • Enhancement to predicate pushdown, folding and inference existing rules  
                      • Disabling equivalent rules written in Apache Hive |

(*) LLAP brings new challenges: queries might take *longer to plan than to execute*
Other improvements impacting query optimization

**Statistics**

- **Collection**
  - Faster computation
  - All statistics stored per partition vs per table
  - Automatic gathering while executing *insert overwrite* statement

- **Estimation**
  - Improved selectivity on estimates, based on NDVs, *min*, *max*

- **Retrieval**
  - Merge partition statistics
  - Multiple ongoing efforts to improve metastore scalability by getting rid of ORM layer
    - Use Apache Hbase to store metadata
    - Redesign underlying schema for metastore
      - Yahoo recorded improvements of up to 90% for some queries
Agenda

An overview on optimization in Apache Hive

Introduction

Query optimizer overview

Logical optimization
Logical optimization

Apache Calcite

- Top-level project since October 2015
- Query planning framework
  - Relational algebra, transformation rules, cost model
  - Extensible
  - Usable standalone (JDBC) or embedded
- Wide adoption by the Apache community
  - Apache Apex, Apache Drill, Apache Flink, Apache Hive, Apache Kylin, Apache Samza, etc.
Logical optimization

Apache Calcite – Architecture
Logical optimization

Apache Calcite – Architecture
Logical optimization

Apache Calcite – Query planner

- Rewriting rules
  - Define a pattern to match in the operator plan
  - Describe a transformation to generate an equivalent plan

- Planner graph
  - Node for each operator in plan
  - Each node is a set of alternate (sub)plans
  - Set further divided into subsets (based on traits like sortedness)

- Planner logic
  - Add matched rule to priority queue
  - Apply matched rule transformation to plan graph
  - Iterate for fixed iterations or until cost does not change
    - Apache Hive: *cardinality-based cost estimation*
Logical optimization

- Multi-phase optimization using **Apache Calcite**
  - Both rule-based and cost-based phases

- More than 40 different rewriting rules
  - Pushdown Filter predicates
  - Pushdown Project expressions
  - Infer new Filter predicates
  - Infer new constant expressions
  - Prune unnecessary operator columns
  - Simplify expressions
  - Pushdown Sort and Limit operators
  - Merge operators
  - Pull up constants
  - ...
Logical optimization – Example 1 (cost-based)
Logical optimization – Example 1 (cost-based)

Join reordering

- Capable of generating **bushy join operator trees**
  - Both inputs of a join operator can receive results from other join operators
  - Well suited for parallel execution engines: increases parallelism degree, performance and cluster utilization

![Diagram of Left-deep join tree and Bushy join tree]

- Combines **exhaustive search with greedy algorithm**
  - Exhaustive search finds every possible plan using rewriting rules
    - Not practical for large number of joins
  - Greedy algorithm builds the plan iteratively
    - Uses heuristic to choose best join to add next
Logical optimization – Example 1 (cost-based)

Join reordering

- Combining two star schemas
  - Fact tables:
    - *sales*, *inventory*
  - Dimension tables:
    - *customer*, *time*, *product*, *warehouse*
Logical optimization – Example 2 (rule-based)
Logical optimization – Example 2 (rule-based)

Interaction of different rules allows to generate more efficient plans

- **Predicate pushdown**: push predicates as close to Scan operators as possible
- **Predicate inference**: infer new predicates from existant ones (maybe redundant) that help to optimize the plan further
- **Predicate propagation**: propagate inferred values for expressions through the plan
- **Predicate folding**: fold constant expressions in predicates
Logical optimization – Example 2 (rule-based)

```sql
SELECT quantity FROM sales s
JOIN addr a ON s.c_id=a.c_id
WHERE (a.country='US' AND a.c_id=2452276)
    OR (a.country='US' AND a.c_id=2452640)
```

Query logical plan:
- **Project**
- **Filter**
- **Join**
- **Scan**
- **Scan**

- `s.c_id=a.c_id`
- `(a.country='US' AND a.c_id=2452276) OR (a.country='US' AND a.c_id=2452276)`
Logical optimization – Example 2 (rule-based)

- Predicate pushdown
- Predicate inference
- Predicate propagation
- Predicate folding

```
Provider

Scan sales
Join s.c_id=a.c_id
Project

quantity

Scan addr
Filter

(a.country='US' AND a.c_id=2452276) OR
(a.country='US' AND a.c_id=2452276)
```
Logical optimization – Example 2 (rule-based)

- Predicate pushdown
- Predicate inference
- Predicate propagation
- Predicate folding

```
Predicate pushdown
Predicate inference
Predicate propagation
Predicate folding
```

```
(sales addr s.c_id=a.c_id)

Project

Join

Scan

Filter

Scan

quantity

(a.country='US' AND a.c_id=2452276) OR
(a.country='US' AND a.c_id=2452276)
AND a.country='US'
AND (a.c_id=2452276 OR a.c_id=2452276)
```
Logical optimization – Example 2 (rule-based)

- Predicate pushdown
- Predicate inference
- Predicate propagation
- Predicate folding

```
quantity

Project

Join

s.c_id=a.c_id

Filter

s.c_id=2452276 OR s.c_id=2452276

Scan

(a.country='US' AND a.c_id=2452276) OR (a.country='US' AND a.c_id=2452276) AND a.country='US'

Filter

(a.c_id=2452276 OR a.c_id=2452276)

Scan

addr

sales
```
Logical optimization – Example 2 (rule-based)

- Predicate pushdown
- Predicate inference
- Predicate propagation
- Predicate folding

```
Project

Join

Filter
  s.c_id=a.c_id
  s.c_id=2452276 OR s.c_id=2452276

Scan
  sales

Filter
  s.c_id=2452276 OR s.c_id=2452276

Scan
  addr

(quantity)

Filter
  ('US'='US' AND a.c_id=2452276) OR ('US'='US' AND a.c_id=2452276)
  AND a.country='US'
  AND (a.c_id=2452276 OR a.c_id=2452276)
```
Logical optimization – Example 2 (rule-based)

- Predicate pushdown
- Predicate inference
- Predicate propagation
- Predicate folding

```
quantity

(\text{true \ AND \ a.c_id=}2452276) \ OR \\
(\text{true \ AND \ a.c_id=}2452276) \ AND \ a.country='US' \\
AND (a.c_id=}2452276 \ OR \ a.c_id=}2452276)
```

```
sales

s.c_id=}a.c_id

s.c_id=}2452276 \ OR \\
s.c_id=}2452276

addr

Filter

Scan

Filter

Scan

Join

Project
Logical optimization – Example 2 (rule-based)

- **Predicate pushdown**
- **Predicate inference**
- **Predicate propagation**
- **Predicate folding**

```
sales addr
s.c_id=a.c_id

Filter

Scan

quantity

Project

Join

Filter

Filter

Scan

Scan

s.c_id=2452276 OR s.c_id=2452276

(a.c_id=2452276 OR a.c_id=2452276)
AND a.country='US'
AND (a.c_id=2452276 OR a.c_id=2452276)
```
Logical optimization – Example 2 (rule-based)

- Predicate pushdown
- Predicate inference
- Predicate propagation
- Predicate folding

```
Join

s.c_id=a.c_id

Filter

s.c_id=2452276 OR s.c_id=2452276

Scan

sales

Project

quantity

Filter

a.country='US' AND (a.c_id=2452276 OR a.c_id=2452276)

Scan

addr
```
Agenda

An overview on optimization in Apache Hive

Introduction
Query optimizer overview
Logical optimization
Physical optimization
Physical optimization

- Determine how the plan will be executed
  - Algorithms to use for each of the operators
  - Partitioning and sorting data between operators

- Finds additional optimization opportunities
  - Removal of unnecessary stages

- Completely done in Apache Hive
  - As we tighten the integration between Apache Hive and Apache Calcite, more decisions should be taken in Apache Calcite
    - Example: join algorithm selection
Physical optimization – Example
Physical optimization - Example

- Dynamic partition pruning
  - Runtime decision to cancel scan of partitions depending on subquery results
  - Special case of the more general semi-join reduction optimization

```
SELECT ...
FROM sales JOIN time ON sales.time_id = time.time_id
WHERE time.year = 2014 AND time.quarter IN ('Q1', 'Q2')
```

`sales` table partitioned by `time_id`
Physical optimization - Example

- Dynamic partition pruning
  - Runtime decision to cancel scan of partitions depending on subquery results
  - Special case of the more general semi-join reduction optimization

```
SELECT ...
FROM sales JOIN time ON sales.time_id = time.time_id
WHERE time.year = 2014 AND time.quarter IN ('Q1', 'Q2')
```

Physical query plan

```
sales.time_id = time.time_id
```

Join

```
sales
Scan
```

```
ReduceSink
key: time_id
```

```
Filter
year = 2014 AND quarter IN ('Q1', 'Q2')
```

```
Scan
time
```
Physical optimization - Example

- Dynamic partition pruning
  - Runtime decision to cancel scan of partitions depending on subquery results
  - Special case of the more general semi-join reduction optimization

```
SELECT ...
FROM sales JOIN time ON sales.time_id = time.time_id
WHERE time.year = 2014 AND time.quarter IN ('Q1', 'Q2')
```

**Physical query plan**

- **Scan** on `sales` with condition `sales.time_id = time.time_id`
- **ReduceSink** with key `time_id`
- **Filter** with condition `year = 2014 AND quarter IN ('Q1', 'Q2')`
- **Scan** on `time`

*Scan only subset of partitions with given `time_id`*
Agenda

An overview on optimization in Apache Hive

Introduction
Query optimizer overview
Logical optimization
Physical optimization
Optimizer performance
Optimizer performance

Apache Hive 2.1.0 - TPCDS (10TB) queries

* Max execution time: 1000 s
Agenda

An overview on optimization in Apache Hive

Introduction
Query optimizer overview
Logical optimization
Physical optimization
Optimizer performance

Road ahead
Work in progress and future work

Extended cost model

- Cost model should reflect actual execution cost
- Takes into account CPU and IO cost for operators
- Dependent on underlying execution engine
  - In contrast to cardinality-based model (default cost model)
  - Only available for Apache Tez
- Requires ‘tuning’ parameters
  - CPU atomic operation cost
  - Local disk byte read/write cost
  - HDFS byte read/write cost
  - Network byte transfer cost
- Allows join algorithm selection in Apache Calcite
  - Sort-merge join, mapjoin, bucket mapjoin, sorted bucket mapjoin
Materialized views support

- A materialized view contains the results of a query
- Accelerate query execution by using automatic rewriting
  - Automatically benefits from LLAP, since materialized view is represented as a table
  - Possibility to use other systems to store views (federation)
- Powerful side-effects: accurate statistics for query plans
Work in progress and future work

Materialized views support

- Table cmv_table

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>unit_price</th>
<th>discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>'p1'</td>
<td>10.30</td>
<td>0.15</td>
</tr>
<tr>
<td>2</td>
<td>'p2'</td>
<td>3.14</td>
<td>0.35</td>
</tr>
<tr>
<td>3</td>
<td>'p3'</td>
<td>172.2</td>
<td>0.35</td>
</tr>
<tr>
<td>4</td>
<td>'p4'</td>
<td>978.76</td>
<td>0.35</td>
</tr>
</tbody>
</table>

- Syntax

```sql
CREATE MATERIALIZED VIEW cmv_mat_view ENABLE REWRITE
AS
    SELECT id, discount
    FROM cmv_table
    WHERE discount > 0.2;
```

Enable automatic rewriting using this materialized view
Materialized views support – Automatic rewriting

- Implemented as a set of rewriting rules
- Early pre-filtering of irrelevant views
- Currently supports Project, Filter, Scan operators

```
SELECT id, discount
FROM cmv_table
WHERE discount > 0.3;
```
Work in progress and future work

Materialized views support – Automatic rewriting

- Implemented as a set of rewriting rules
- Early pre-filtering of irrelevant views
- Currently supports Project, Filter, Scan operators

```
SELECT id, discount
FROM cmv_table
WHERE discount > 0.3;
```

[Diagram showing query plans for cmv_table and cmv_mat_view]
Work in progress and future work

Materialized views support – Automatic rewriting

- Implemented as a set of rewriting rules
- Early pre-filtering of irrelevant views
- Currently supports Project, Filter, Scan operators

```
SELECT id, discount
FROM cmv_table
WHERE discount > 0.3;
```
Work in progress and future work

Materialized views support – Upcoming features

- Allow partitioned materialized views
- Extend automatic rewriting to cover more cases
  - Support at least Project, Filter, Join, Aggregate operators
- Explore materialized view maintenance
  - Incremental view update (ACID should help)
  - Time window for valid automatic rewriting
Conclusion

Progress of Apache Hive optimizer

- Improvements on **effectiveness** and **efficiency** over the last 3 years
  - Contributions to **Apache Hive** and **Apache Calcite**

- Interesting road ahead for both communities
  - Discussions around query decorrelation, materialized views, query factoring, plan caching, parameterized plans, and many more
Conclusion

Importance of Apache Calcite

- Explosion in the number of data processing systems
  - Organizations continue investing heavily on creating new data processing systems tailored towards their specific needs

- Extremely valuable to use a common layer of abstraction or framework that integrate easily with these systems
  - Reduce end-to-end time to create data processing applications with an optimizer at the core, while hiding the complexity of the optimization process
  - Ensure correctness and consistent semantics
  - Integrate query languages with different expressive power within the same formal model
  - Enable cross-platform optimization by exposing a common framework for all the systems
Thank You

http://calcite.apache.org | @ApacheCalcite
http://hive.apache.org | @ApacheHive