Apache Calcite

Julian Hyde
Credit Suisse
February 23, 2016

Julian Hyde

Creator & PMC member of Apache Calcite

PMC member of Apache Arrow, Drill, Eagle, Kylin

Creator of Pentaho Mondrian (OLAP engine)

Architect at Hortonworks

Several years building SQL, OLAP, streaming data engines

Member of Apache Software Foundation

@julianhyde

“SQL inside”

Implementing SQL well is hard

- System cannot just “run the query” as written
- Require relational algebra, query planner (optimizer) & metadata

…but it’s worth the effort

Algebra-based systems are more flexible

- Add new algorithms (e.g. a better join)
- Re-organize data
- Choose access path based on statistics
- Dumb queries (e.g. machine-generated)
- Relational, schema-less, late-schema, non-relational (e.g. key-value, document)

Apache Calcite

Apache top-level project

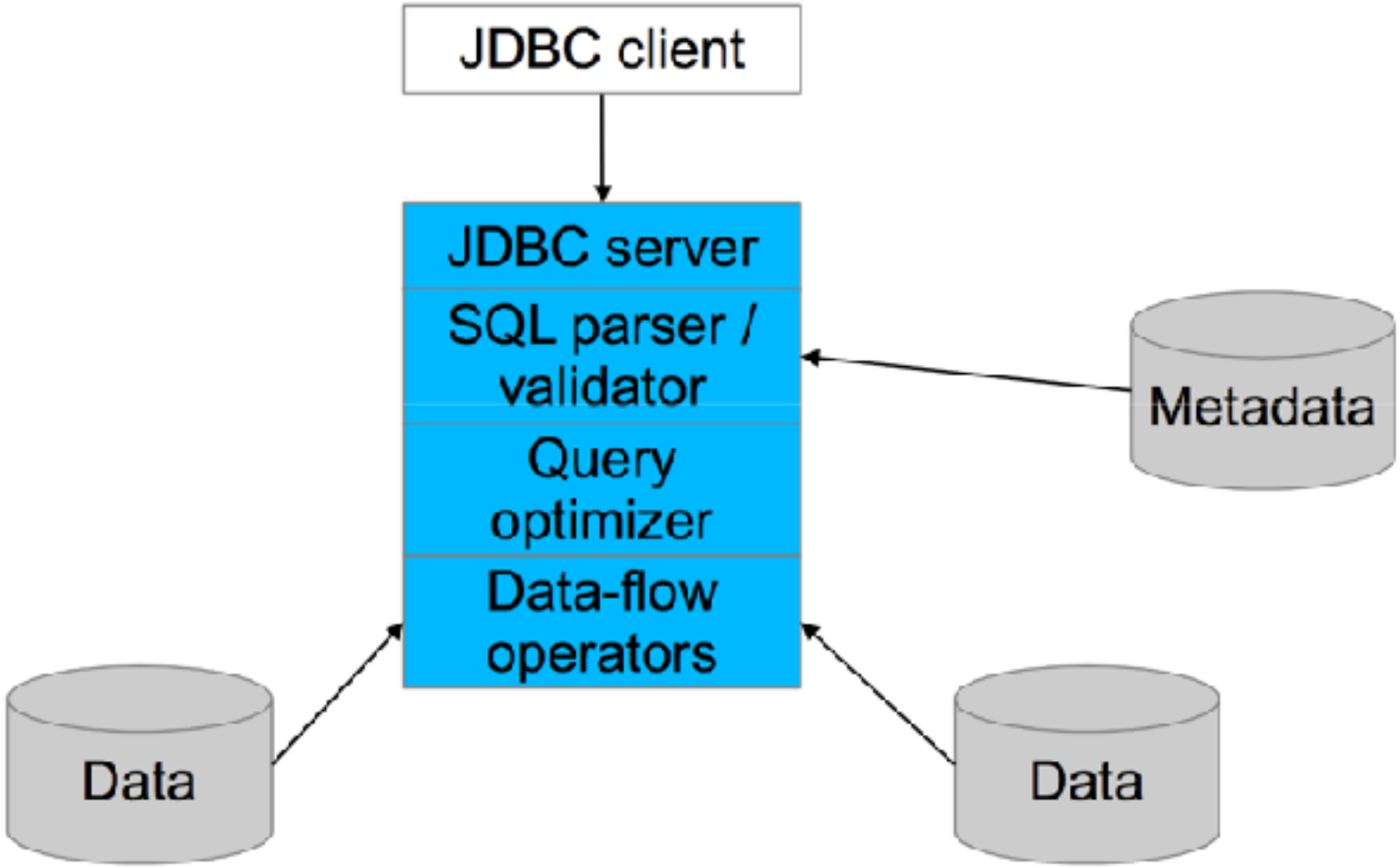
Query planning framework

- Relational algebra, rewrite rules, cost model
- Extensible
- Streaming extensions
- SQL parser

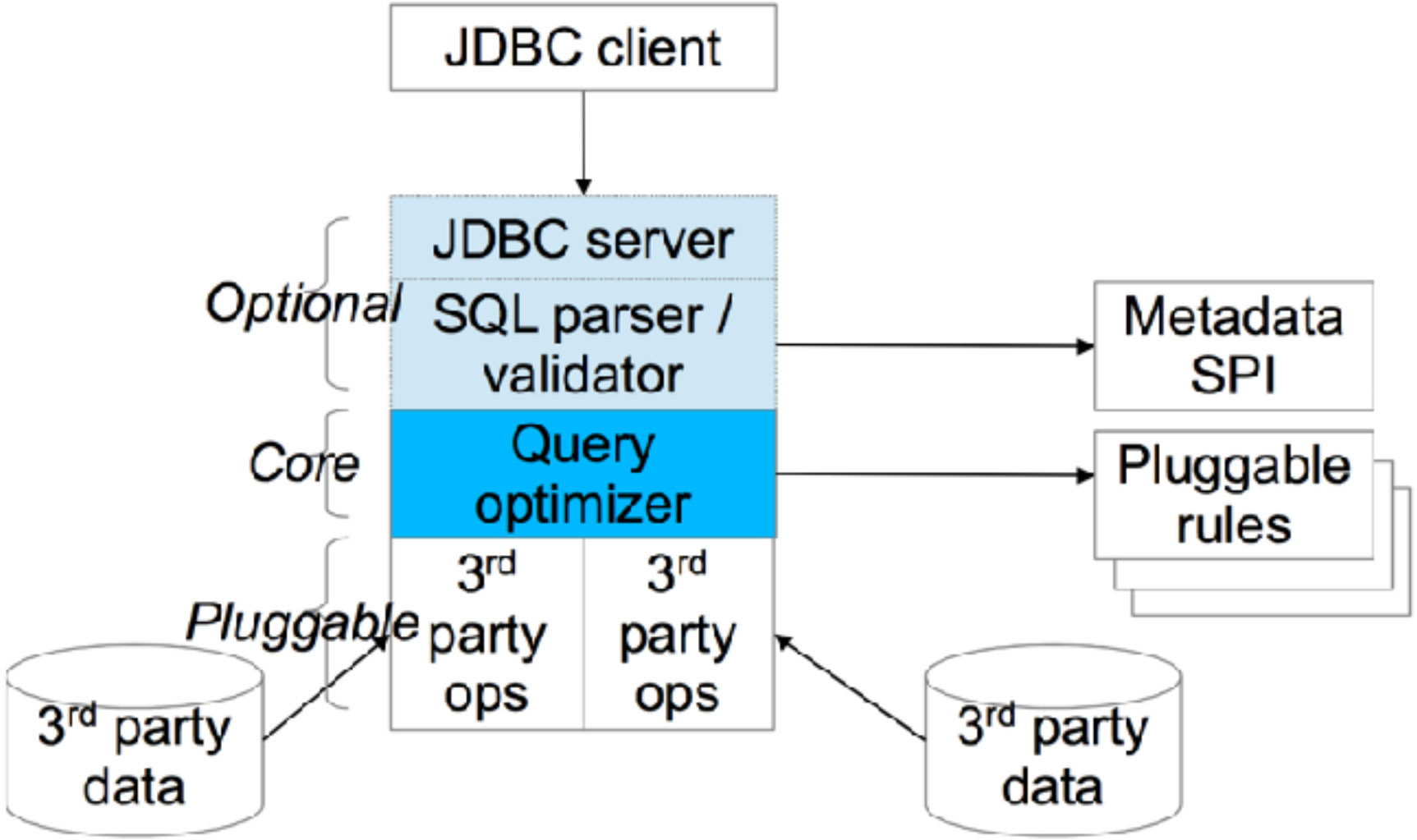
Packaging

- Library (JDBC server optional)
- Open source
- Community-authored rules, adapters
- Can be embedded in another engine, or be a container for adapters

Conventional DB architecture



Calcite architecture



Calcite & Phoenix

Apache Phoenix is a SQL layer on Apache HBase

Phoenix originally had its own SQL parser, validator, rule-based optimizer

Drivers to adopt Calcite:

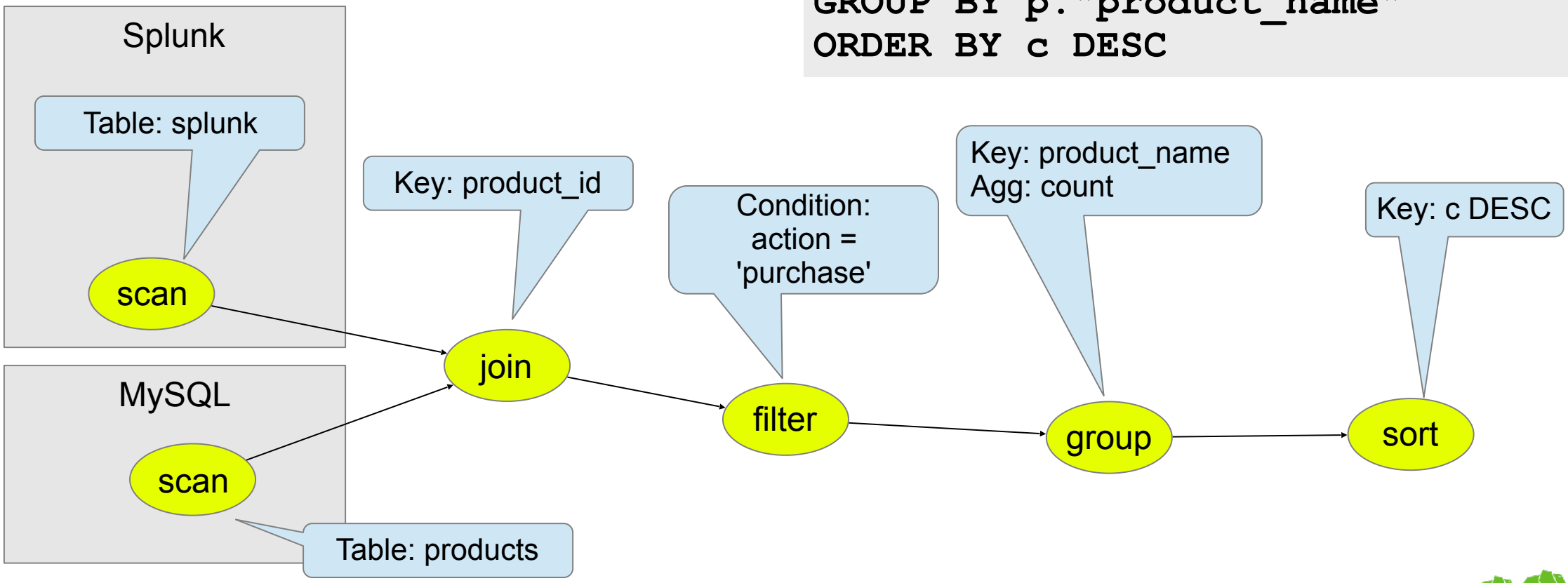
- Maintenance overhead
- SQL standards compliance
- Cost-based optimization
- Integration with other engines

Status:

- End-to-end query execution complete
- Remaining tasks are to ensure compatibility with current Phoenix

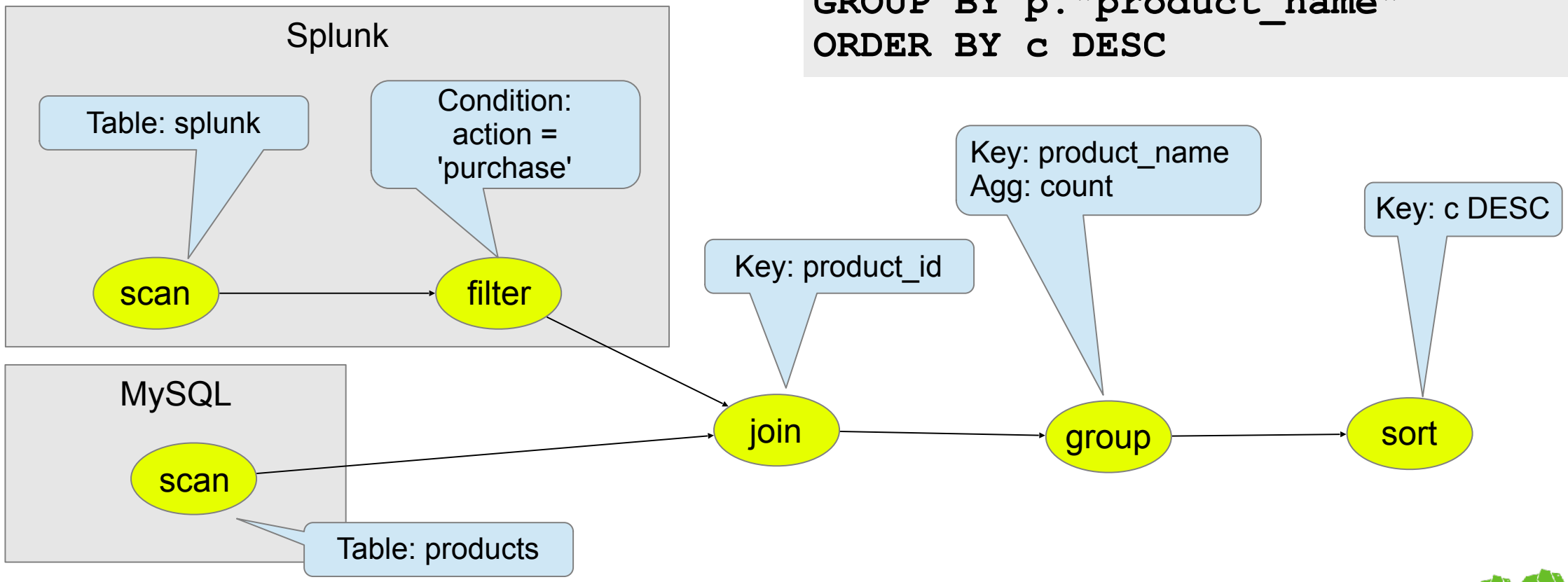
Expression tree

```
SELECT p."product_name", COUNT(*) AS c
FROM "splunk"."splunk" AS s
    JOIN "mysql"."products" AS p
    ON s."product_id" = p."product_id"
WHERE s."action" = 'purchase'
GROUP BY p."product_name"
ORDER BY c DESC
```



Expression tree (optimized)

```
SELECT p."product_name", COUNT(*) AS c
FROM "splunk"."splunk" AS s
    JOIN "mysql"."products" AS p
    ON s."product_id" = p."product_id"
WHERE s."action" = 'purchase'
GROUP BY p."product_name"
ORDER BY c DESC
```



Calcite – APIs and SPIs

Relational algebra

RelNode (operator)
- TableScan
- Filter
- Project
- Union
- Aggregate
- …

RelDataType (type)
RexNode (expression)
RelTrait (physical property)
- RelConvention (calling-convention)
- RelCollation (sortedness)
- RelDistribution (bucketedness)

SQL parser

SqlNode
SqlParser
SqlValidator

Metadata

Schema
Table
Function
- TableFunction
- TableMacro

Lattice

JDBC driver

Transformation rules

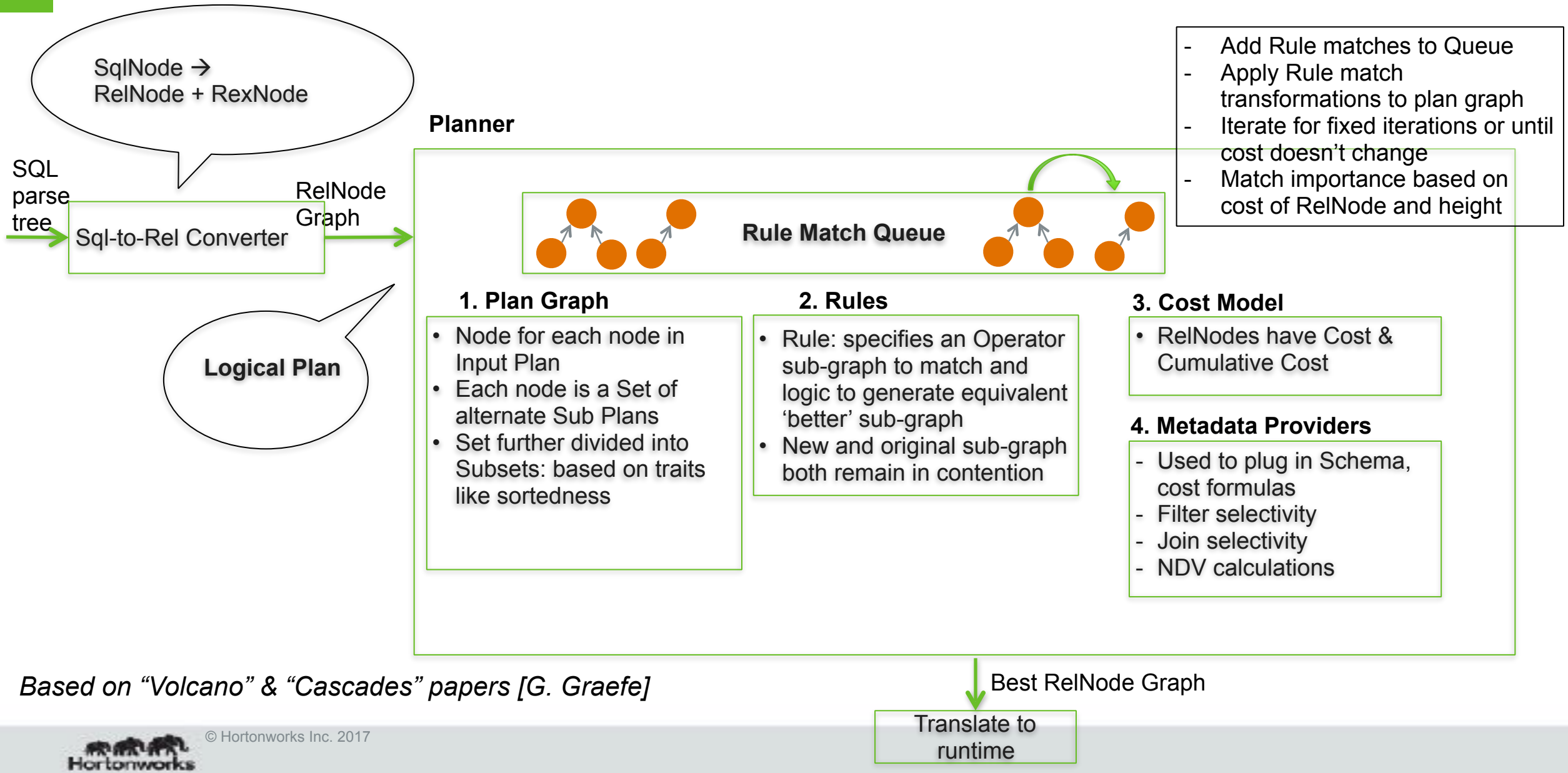
RelOptRule
- MergeFilterRule
- PushAggregateThroughUnionRule
- 100+ more

Global transformations
- Unification (materialized view)
- Column trimming
- De-correlation

Cost, statistics

RelOptCost
RelOptCostFactory
RelMetadataProvider
- RelMdColumnUniquensss
- RelMdDistinctRowCount
- RelMdSelectivity

Calcite Planning Process



Based on "Volcano" & "Cascades" papers [G. Graefe]

Comparison to Apache Spark / Spark SQL / Catalyst

Calcite and Spark SQL / Catalyst similarities

- SQL parser
- Represents queries as relational algebra
- Optimize by transforming queries using “rules”
- Adding support for streaming queries

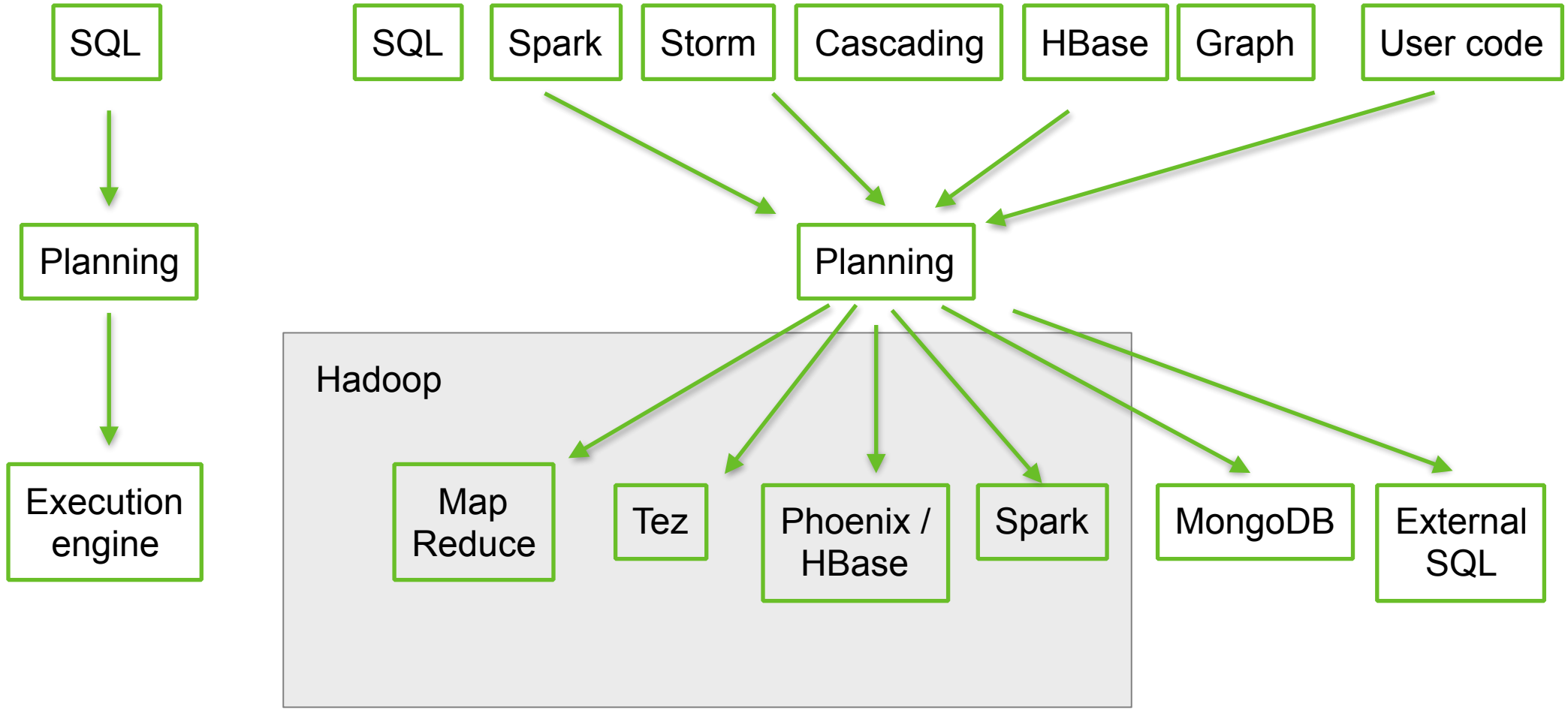
Catalyst differences

- Works exclusively with Spark engine
- Leverages Scala to write rules very concisely
- Heuristic, not cost-based - doesn’t use equivalence sets / Volcano
- Good at pushing down filters and projections to sources

Calcite differences

- Works with multiple engines (Hive, Drill, Phoenix, Pig, Spark); can generate hybrid execution plans
- Proven for optimizing complex queries with large search-spaces (e.g. TPC-DS queries in Hive)
- Community-authored Spark adapter reports 20x performance improvement over Catalyst

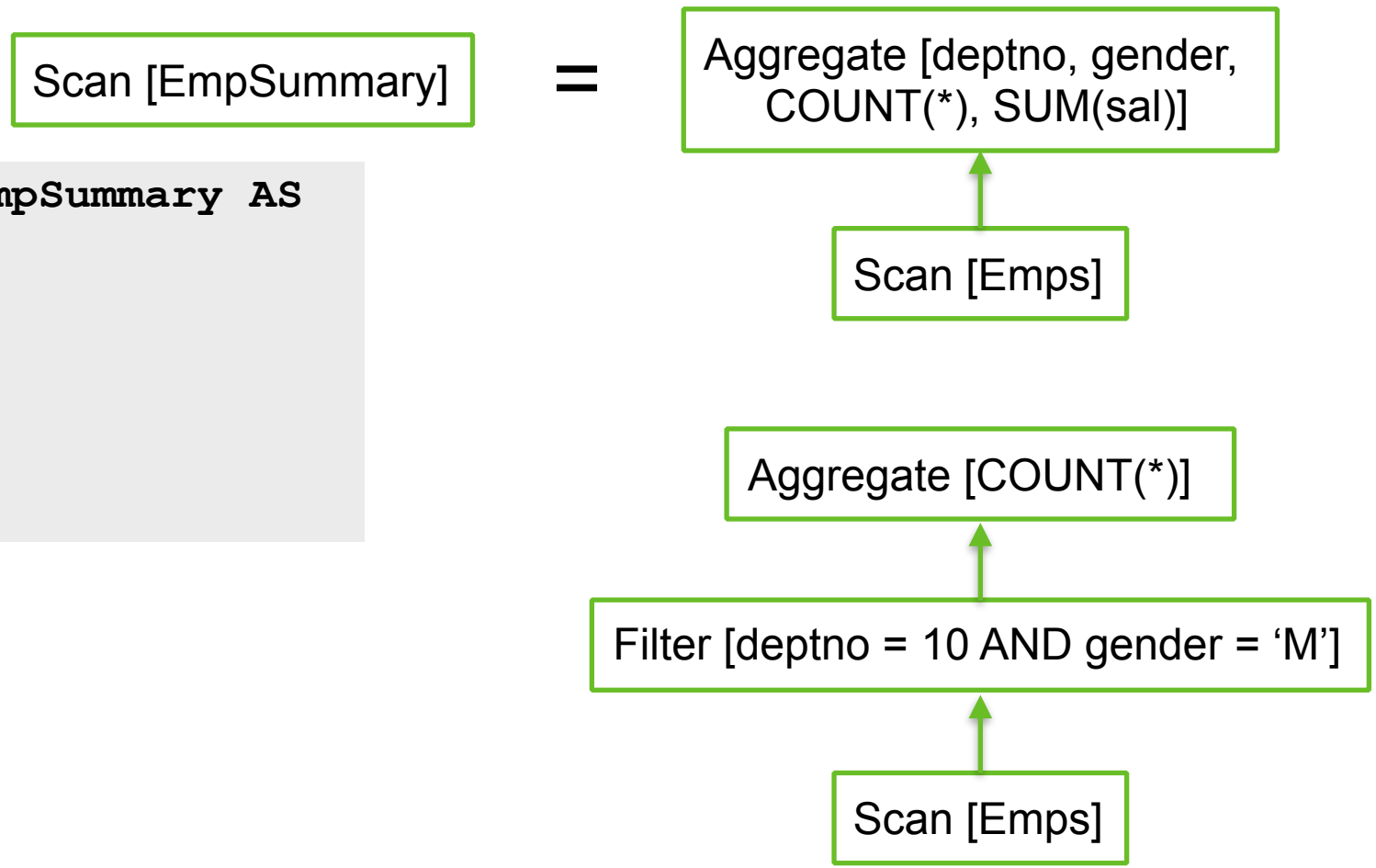
Many front ends, many engines



Materialized view

```
CREATE MATERIALIZED VIEW EmpSummary AS
SELECT deptno,
  gender,
  COUNT(*) AS c,
  SUM(sal) AS s
FROM Emps
GROUP BY deptno, gender
```

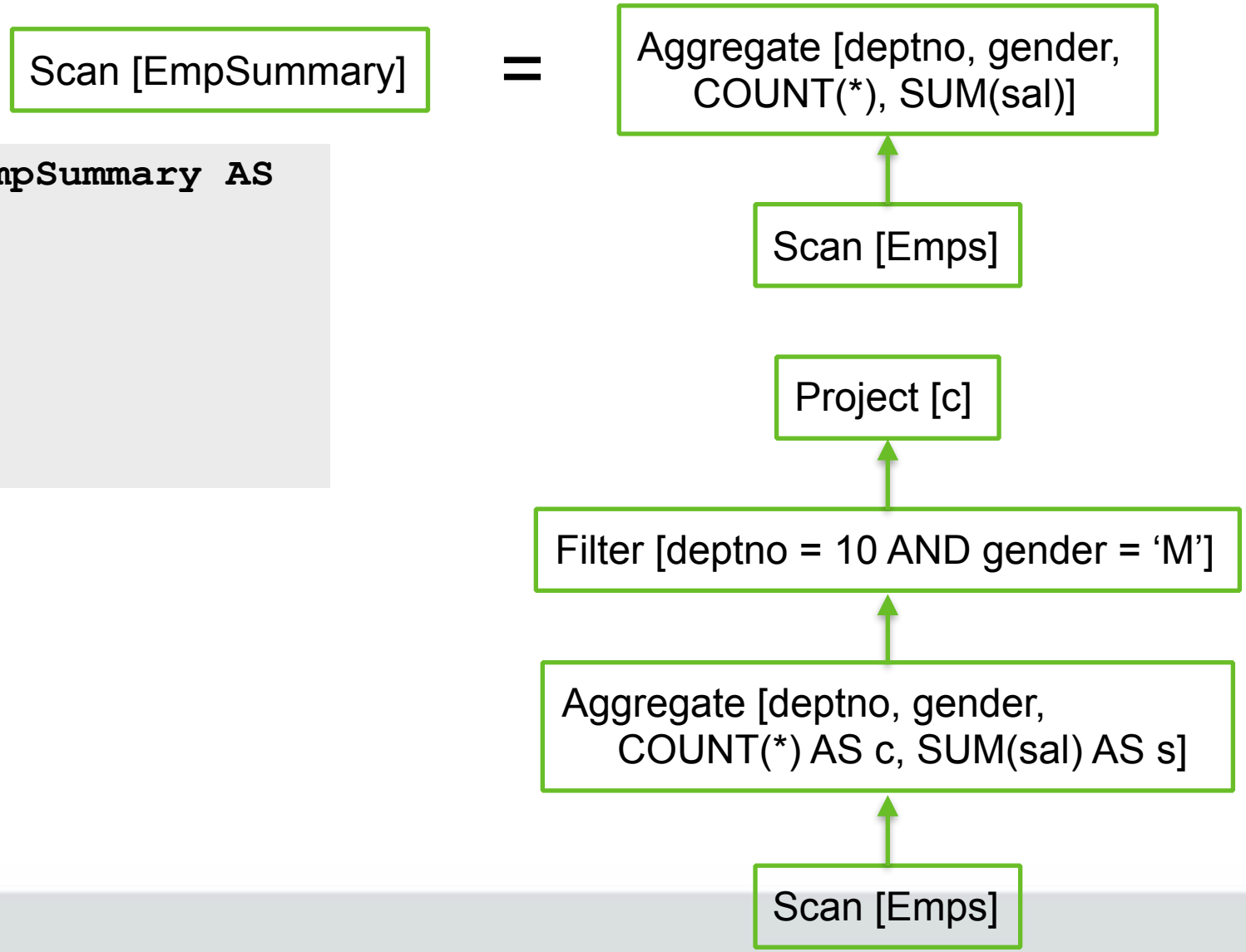
```
SELECT COUNT(*)
FROM Emps
WHERE deptno = 10
AND gender = 'M'
```



Materialized view, step 2: Rewrite query to match

```
CREATE MATERIALIZED VIEW EmpSummary AS
SELECT deptno,
  gender,
  COUNT(*) AS c,
  SUM(sal) AS s
FROM Emps
GROUP BY deptno, gender
```

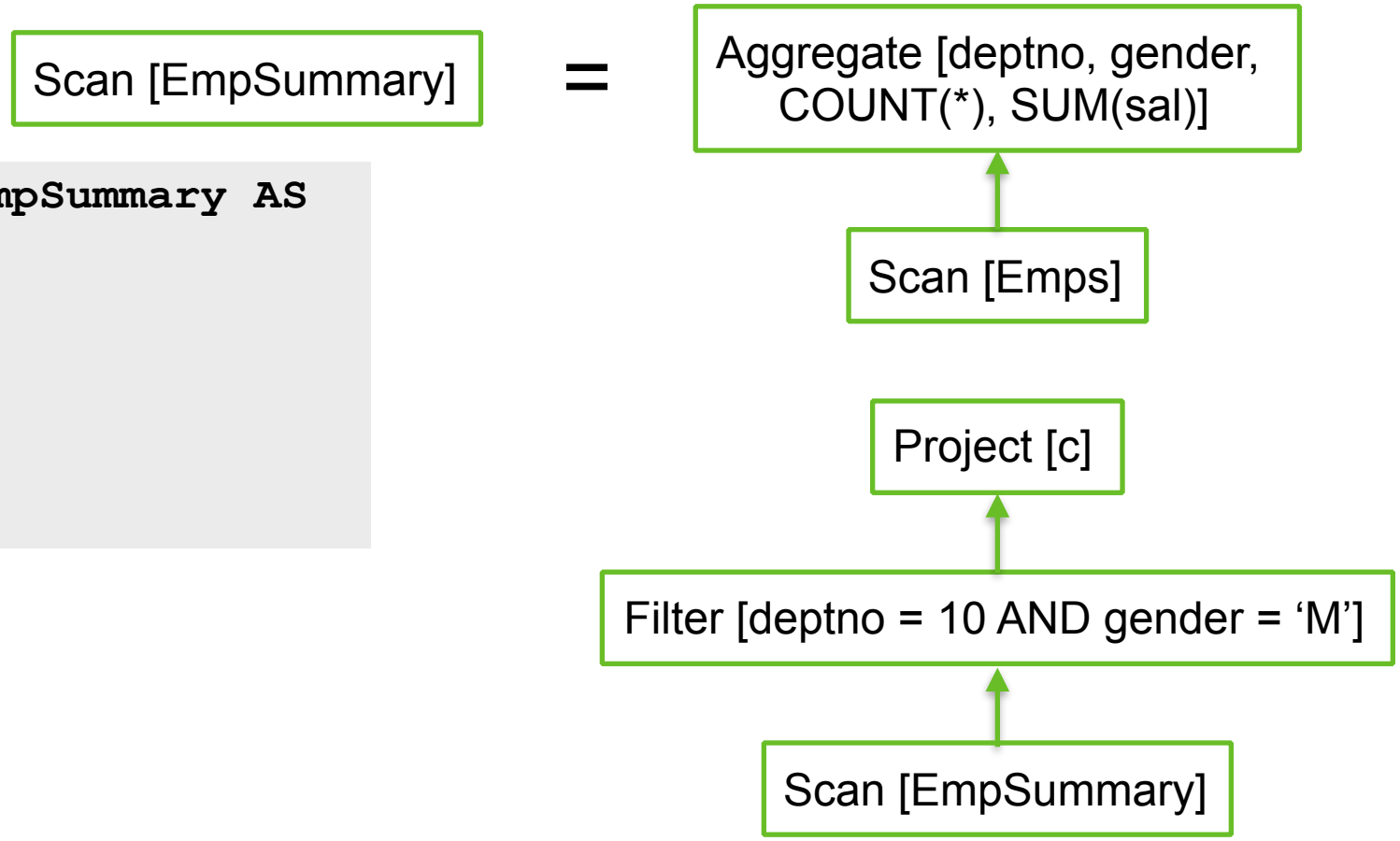
```
SELECT COUNT(*)
FROM Emps
WHERE deptno = 10
AND gender = 'M'
```



Materialized view, step 3: Substitute table

```
CREATE MATERIALIZED VIEW EmpSummary AS
SELECT deptno,
  gender,
  COUNT(*) AS c,
  SUM(sal) AS s
FROM Emps
GROUP BY deptno, gender
```

```
SELECT COUNT(*)
FROM Emps
WHERE deptno = 10
AND gender = 'M'
```



Optimizing for secondary indexes

Schema:

- **Table: Emps (empno, deptno, name, gender, salary); key: (empno)**
- **Index: I_Emps_Deptno (deptno, empno, name); key: (deptno, empno)**

Query:

```
SELECT deptno, name
FROM Emps
WHERE deptno BETWEEN 100 AND 150
ORDER BY deptno
```

Optimal equivalent query:

```
SELECT deptno, name
FROM I_Emps_Deptno
WHERE deptno BETWEEN 100 AND 150
```

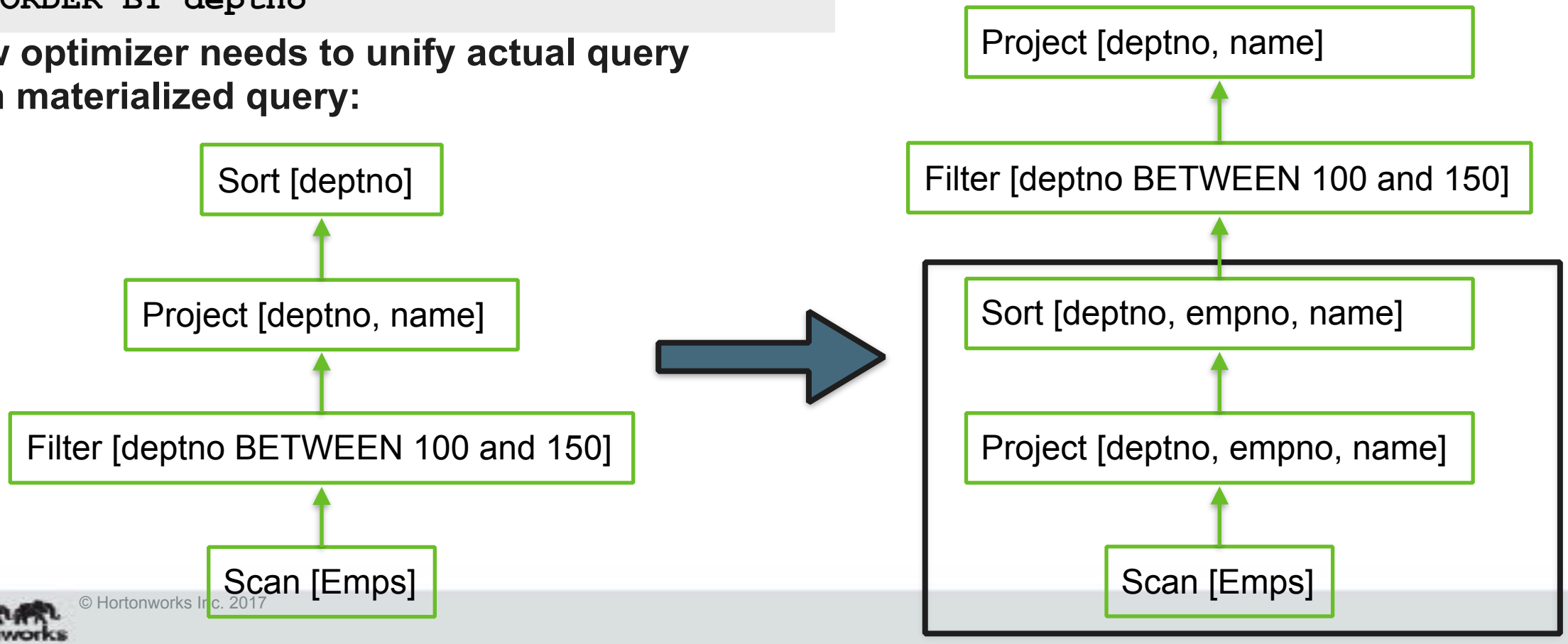
- **Skip scan on leading edge of index**
- **No sort necessary**

Modeling an index as a materialized view

Optimizer internally creates a mapping (query, table) equivalent to:

```
CREATE MATERIALIZED VIEW I_Emp_Deptno AS
SELECT deptno, empno, name
FROM Emps
ORDER BY deptno
```

Now optimizer needs to unify actual query with materialized query:

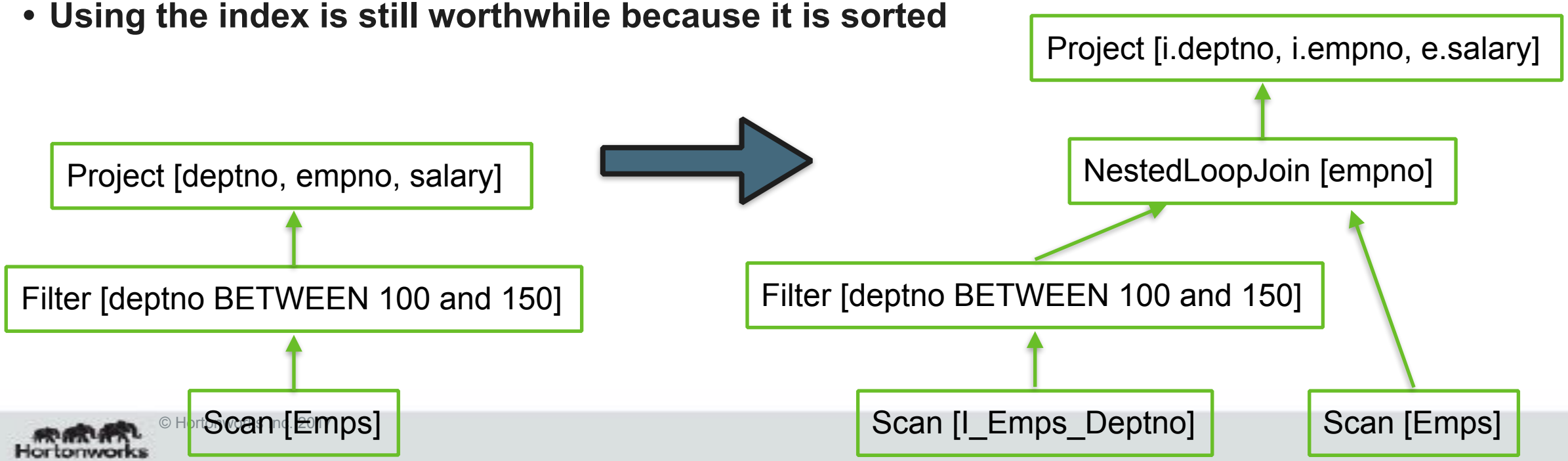


Non-covering index

Query:

```
SELECT deptno, empno, salary
FROM Emps
WHERE deptno BETWEEN 100 AND 150
```

- **Salary is not in the index - we have to join the Emps table to get it**
- **Using the index is still worthwhile because it is sorted**



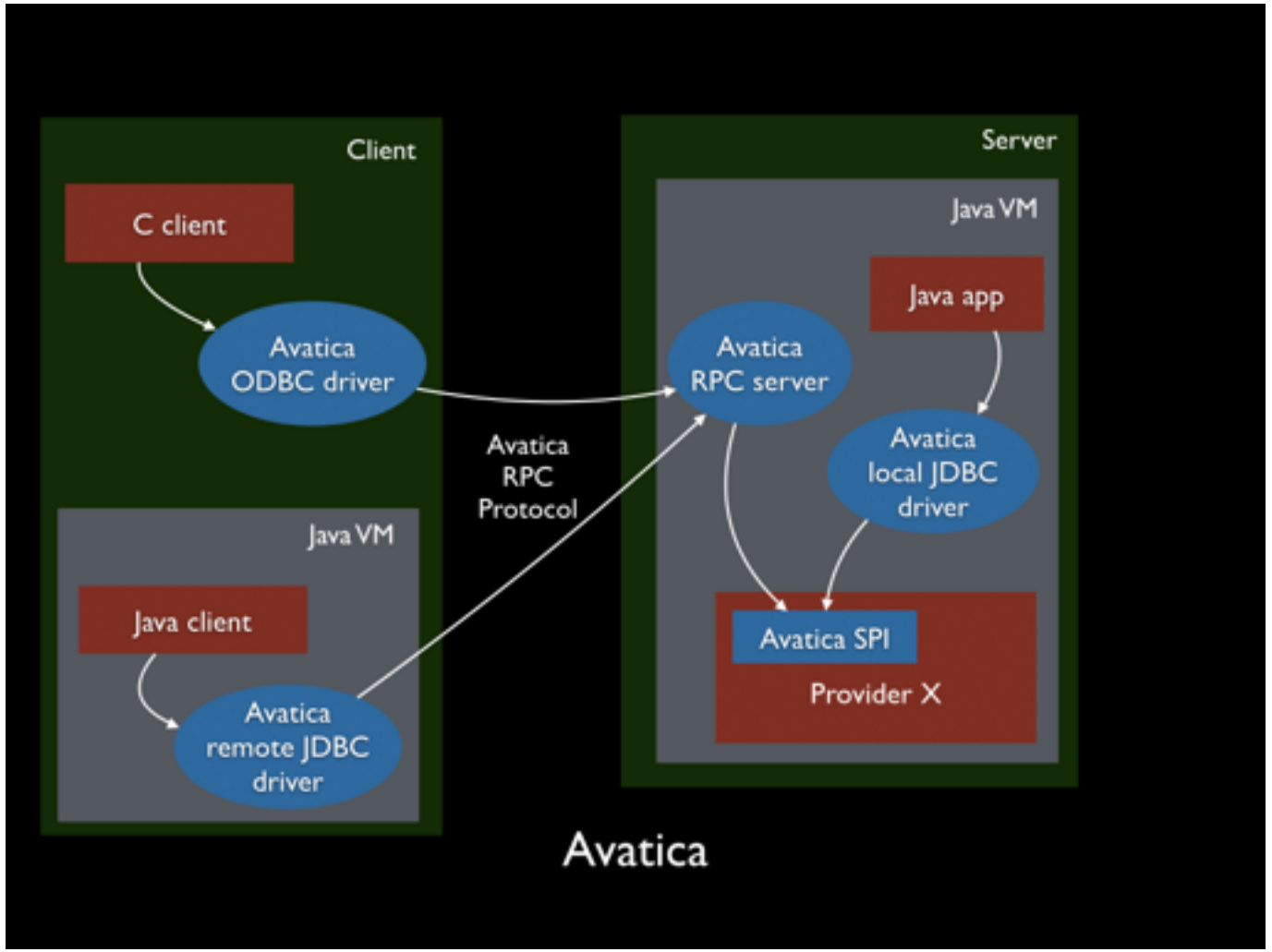
Calcite Avatica & Phoenix Query Server

Avatica is a framework for building portable, distributed ODBC and JDBC drivers

Module within Calcite

RPC: Protobuf over HTTP

Phoenix “thin” remote JDBC driver talks to Phoenix query server



Summary

Calcite is a toolkit to build a database

It’s not just about SQL: the real foundation is relational algebra

Algebra allows:

- Cost-based optimization
- Multiple copies of the data (indexes, materialized views)
- Any front-end (query language) on any back-end (engine and storage)
- Queries that span streaming / hot / cold data

Thank you!

http://calcite.apache.org

http://calcite.apache.org/community/#talks

@julianhyde

@ApacheCalcite

Extra material

Streaming

```
SELECT STREAM DISTINCT productName,
  floor(rowtime TO HOUR) AS h
FROM Orders
```

Delta

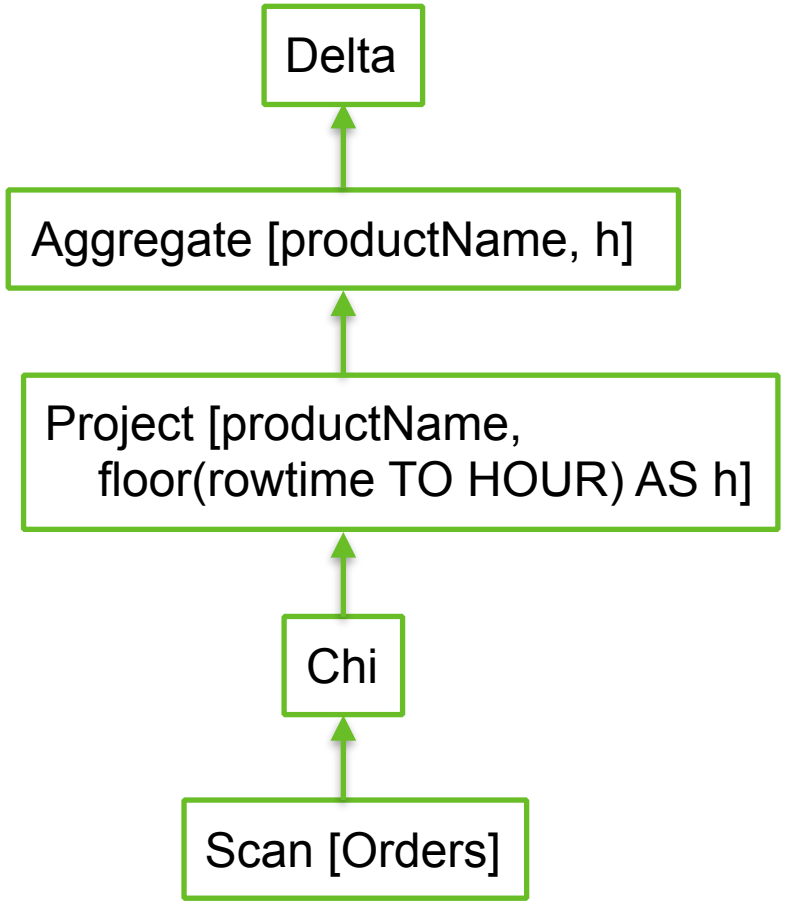
Converts a table to a stream

Each time a row is inserted into the table, a record appears in the stream

Chi

Converts a stream into a table

Often we can safely narrow the table down to a small time window

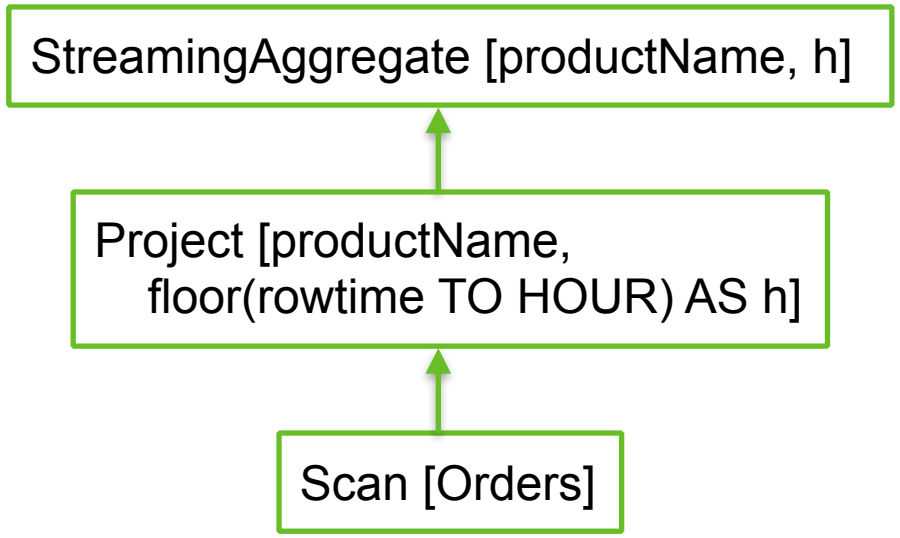


Streaming - efficient implementation

```
SELECT STREAM DISTINCT productName,
  floor(rowtime TO HOUR) AS h
FROM Orders
```

Can create efficient implementation:

- **Input is sorted by timestamp**
- **Only need to aggregate an hour at a time**
- **Output timestamp tracks input timestamp**
- **Therefore it is safe to cancel out the Chi and Delta operators**



Algebraic transformations - streaming

delta(filter(c, R)) → filter(delta(c, R))

delta(project(e1, …, en, R) → project(delta(e1, …, en, R))

delta(union(R1, R2)) → union(delta(R1), delta(R2))

delta(join(R1, R2, c)) → union(join(R1, delta(R2), c),
join(delta(R1), R2), c)

(f + g)’ = f’ + g’

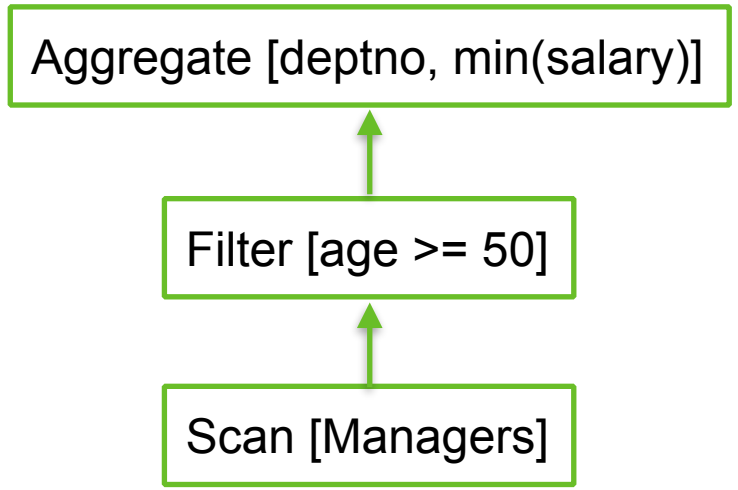
(f . g)’ = f.g’ + f’.g

Delta behaves like “differentiate” in differential calculus,

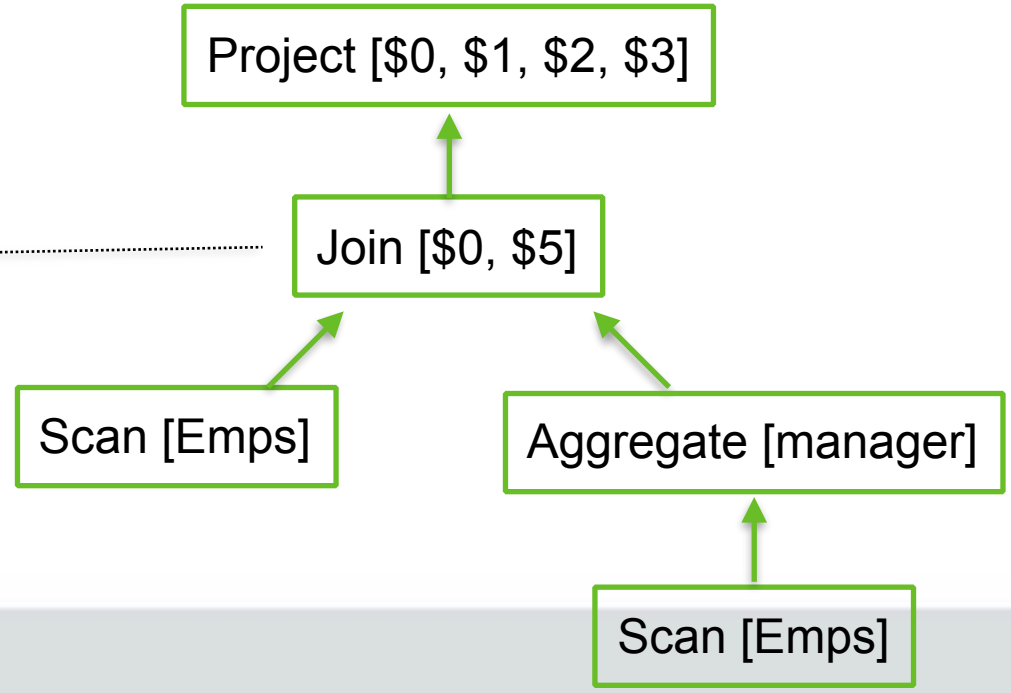
Chi like “integrate”.

Query using a view

```
SELECT deptno, min(salary)
FROM Managers
WHERE age >= 50
GROUP BY deptno
```



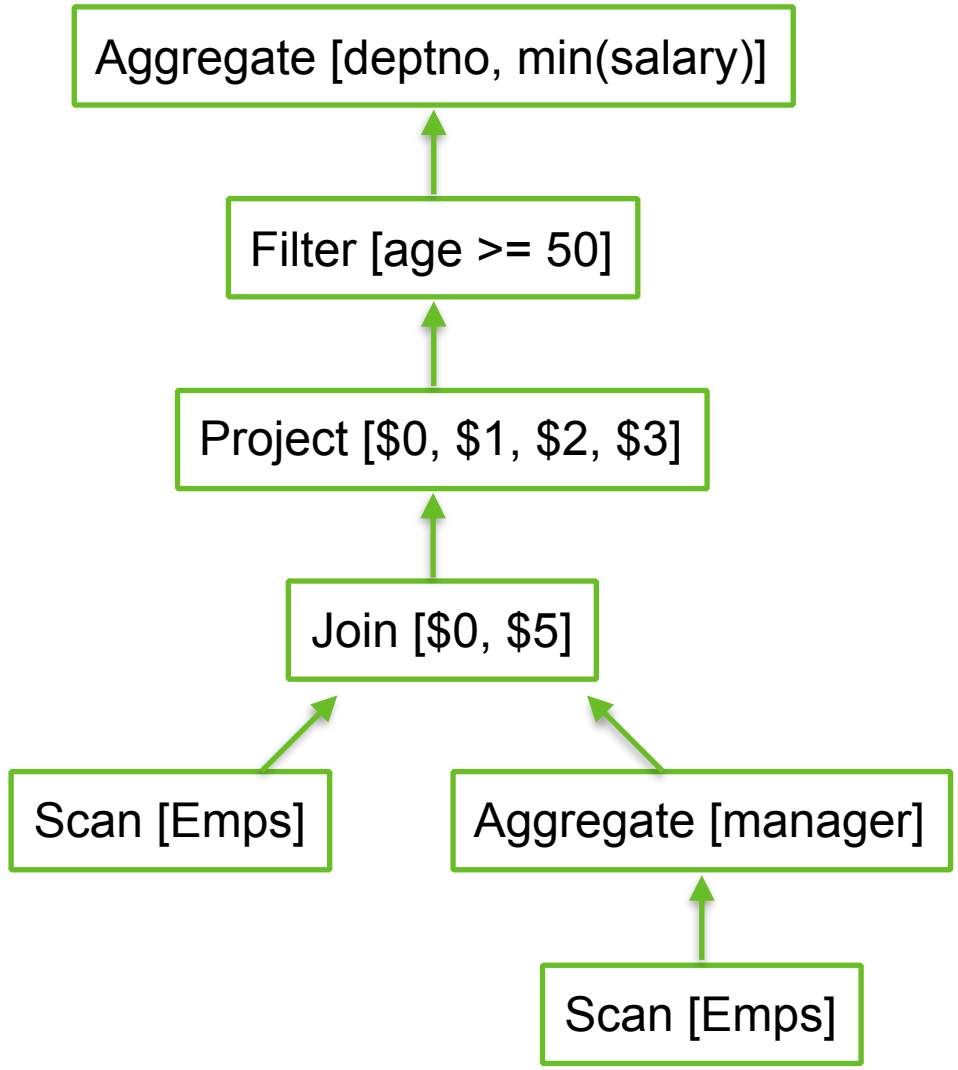

```
CREATE VIEW Managers AS
SELECT *
FROM Emps
WHERE EXISTS (
  SELECT *
  FROM Emps AS underling
  WHERE underling.manager = emp.id)
```



After view expansion

```
SELECT deptno, min(salary)
FROM Managers
WHERE age >= 50
GROUP BY deptno
```

```
CREATE VIEW Managers AS
SELECT *
FROM Emps
WHERE EXISTS (
  SELECT *
  FROM Emps AS underling
  WHERE underling.manager = emp.id)
```



After pushing down filter

```
SELECT deptno, min(salary)
FROM Managers
WHERE age >= 50
GROUP BY deptno
```

```
CREATE VIEW Managers AS
SELECT *
FROM Emps
WHERE EXISTS (
  SELECT *
  FROM Emps AS underling
  WHERE underling.manager = emp.id)
```