Apache Calcite

Julian Hyde
Bloomberg
December 9, 2016

“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

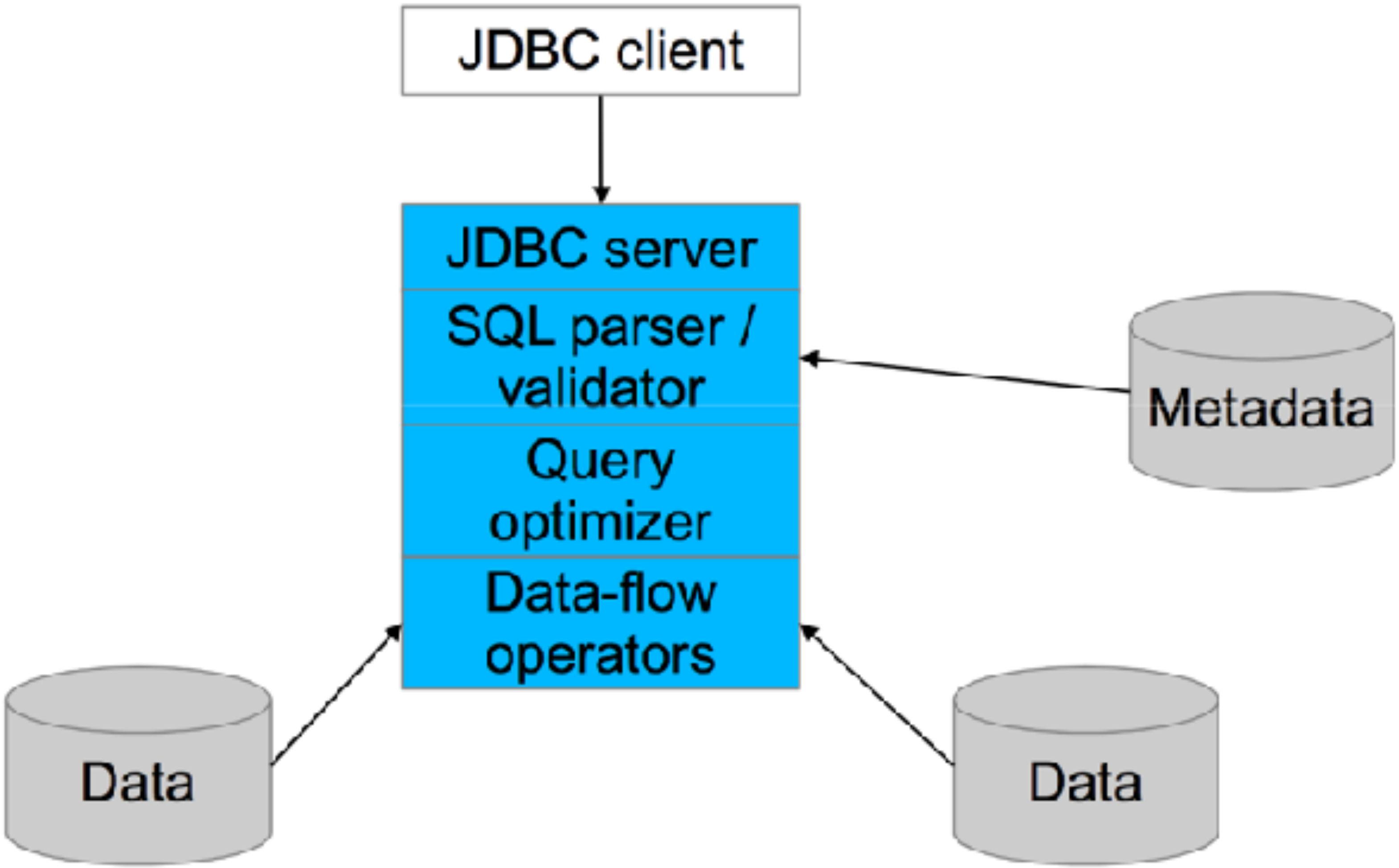
Packaging

- Library (JDBC server optional)
- Open source
- Community-authored rules, adapters

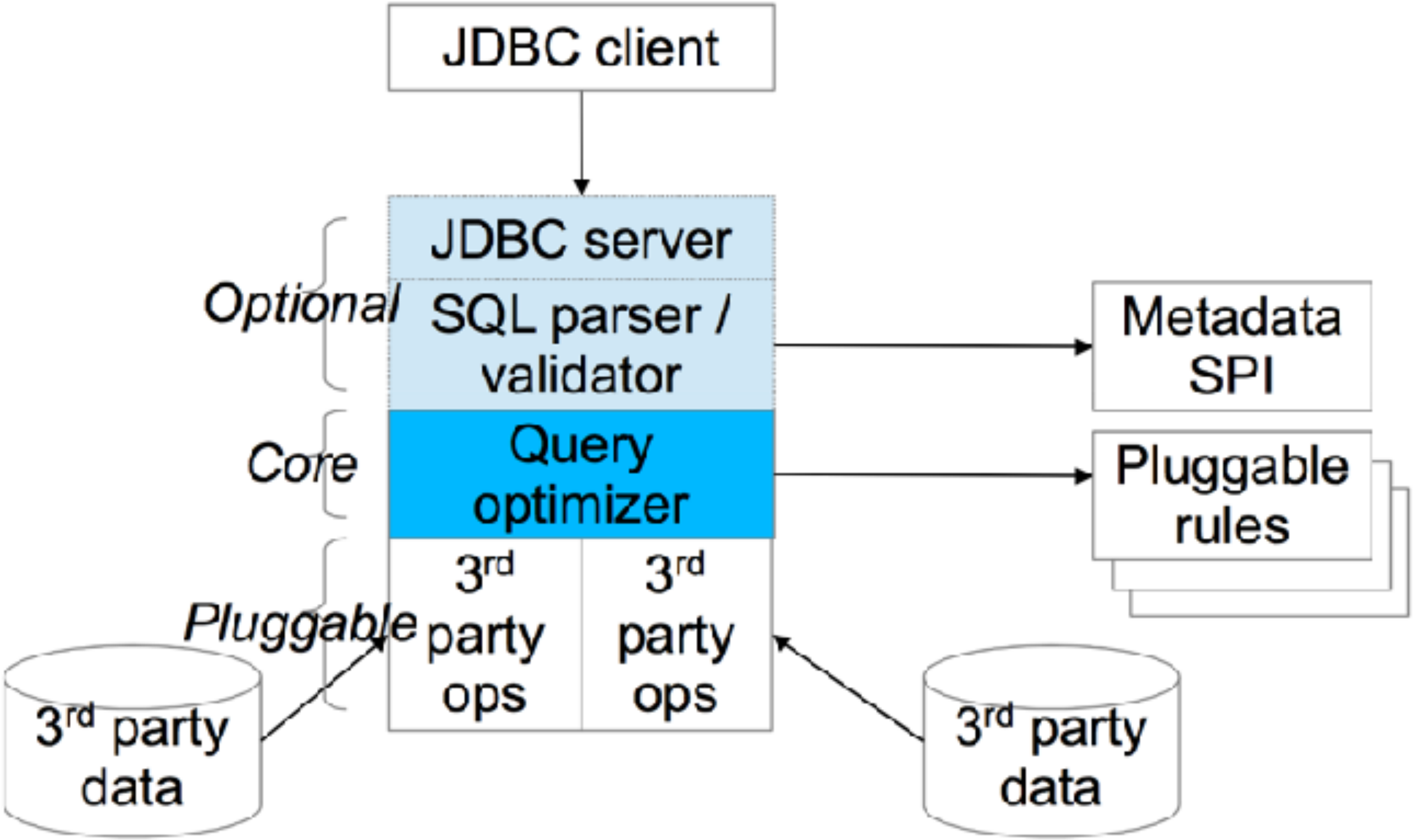
Adoption

- **Embedded**: Lingual (SQL interface to Cascading), Apache Drill, Hive, Kylin, Phoenix, Druid
- **Adapters**: Splunk, Apache Spark, Cassandra, Storm, Samza, Flink, MongoDB, JDBC, CSV, JSON, Web tables, In-memory data

Conventional DB architecture

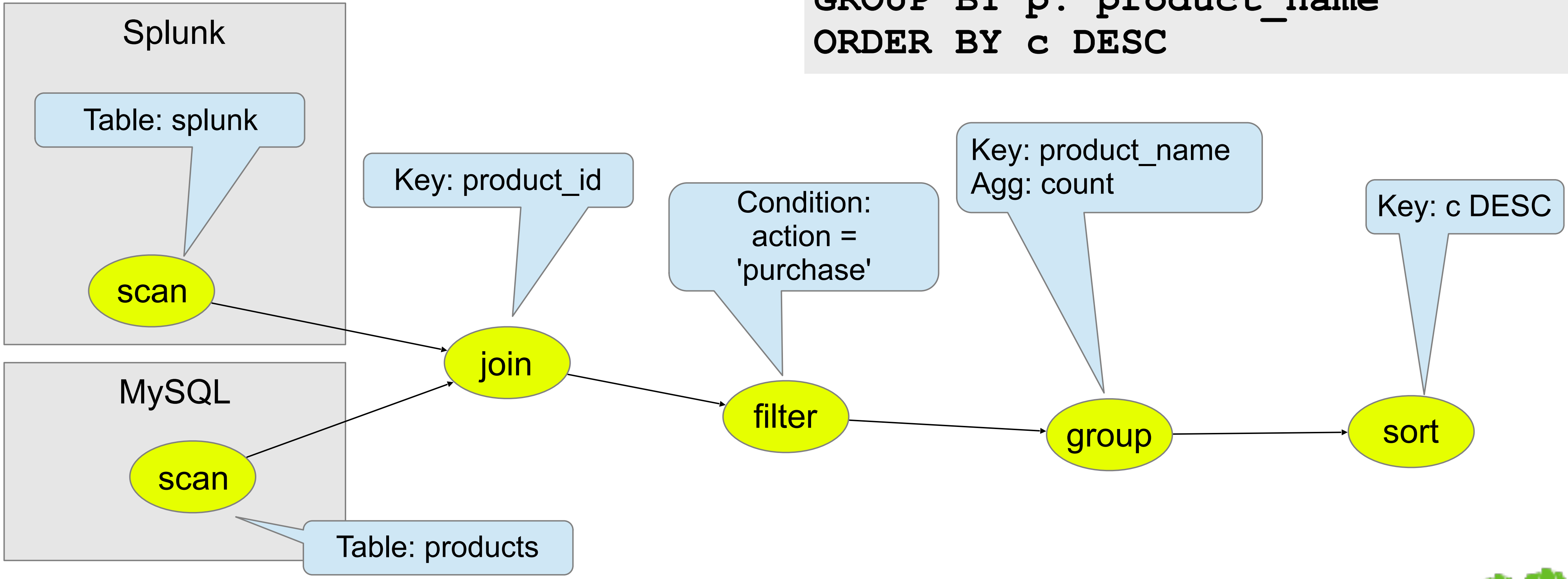


Calcite architecture



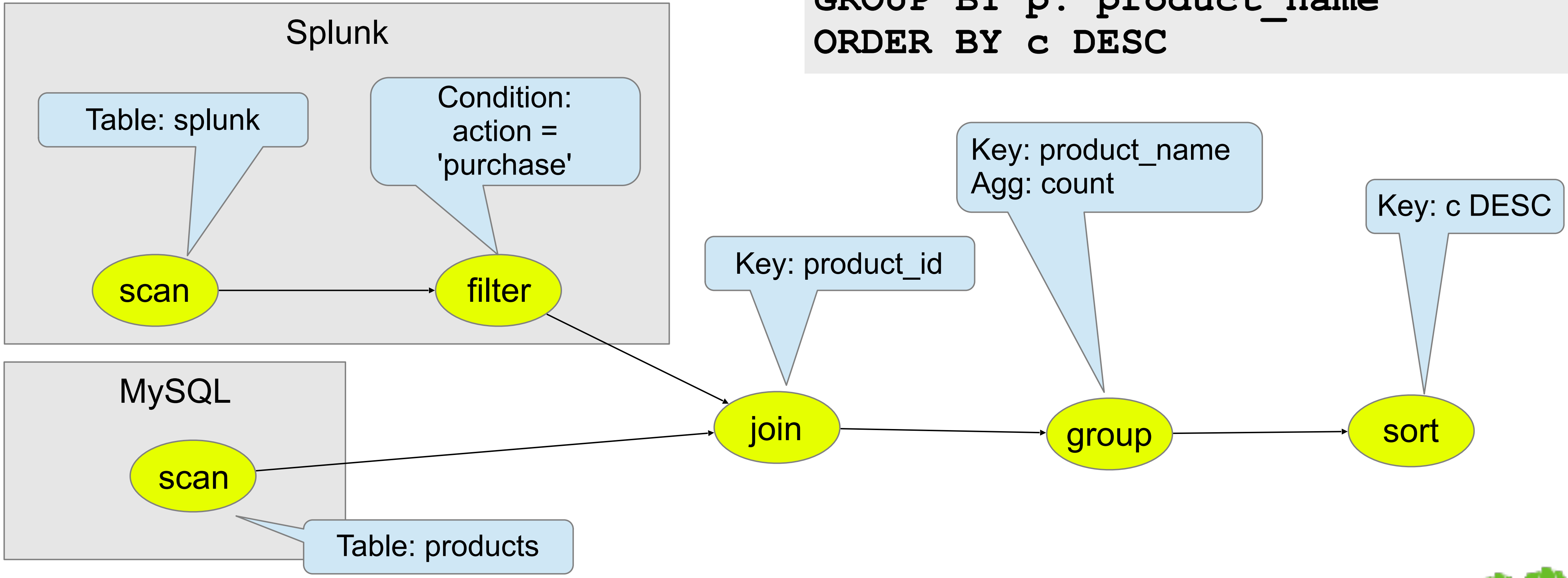
Expression tree

```
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)
- TBD (bucketedness/distribution)

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

SqlNode → RelNode + RexNode

SQL parse tree → Sql-to-Rel Converter → RelNode Graph

Logical Plan

Planner

Rule Match Queue

- Add Rule matches to Queue
- Apply Rule match transformations to plan graph
- Iterate for fixed iterations or until cost doesn't change
- Match importance based on cost of RelNode and height

1. Plan Graph

- Node for each node in Input Plan
- Each node is a Set of alternate Sub Plans
- Set further divided into Subsets: based on traits like sortedness

2. Rules

- Rule: specifies an Operator sub-graph to match and logic to generate equivalent 'better' sub-graph
- New and original sub-graph both remain in contention

3. Cost Model

- RelNodes have Cost & Cumulative Cost

4. Metadata Providers

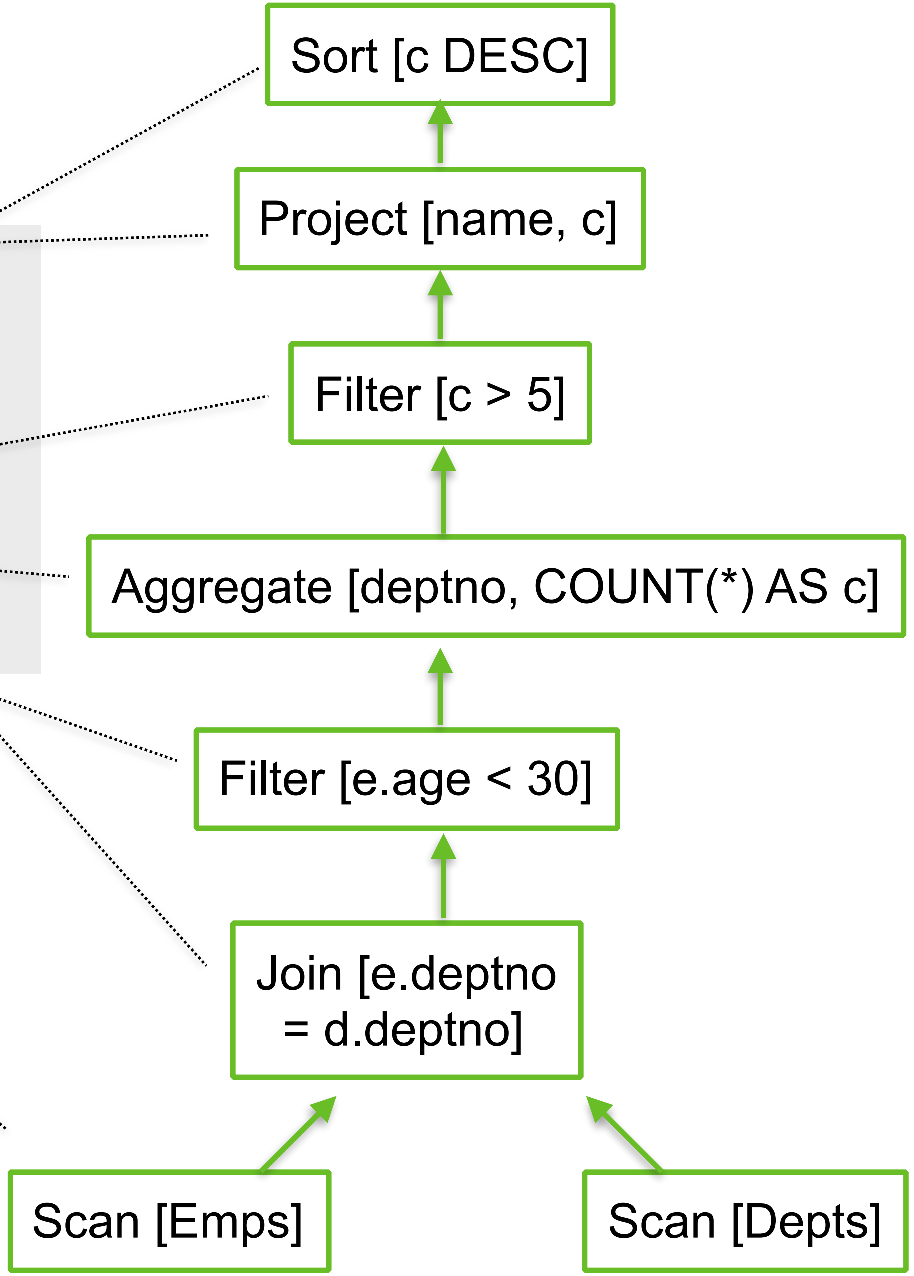
- Used to plug in Schema, cost formulas
- Filter selectivity
- Join selectivity
- NDV calculations

Best RelNode Graph → Translate to runtime

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

Relational algebra

```
SELECT d.name, COUNT(*) AS c
FROM Emps AS e
  JOIN Depts AS d ON e.deptno = d.deptno
WHERE e.age < 30
GROUP BY d.deptno
HAVING COUNT(*) > 5
ORDER BY c DESC
```



(Column names are simplified. They would usually be ordinals, e.g. $0 is the first column of the left input.)

Algebraic transformations

(R filter c1) filter c2 → R filter (c1 and c2)

(R1 union R2) join R3 on c → (R1 join R3 on C) union (R2 join R3 on c)

- Compare distributive law of arithmetic: (x + y) * z → (x * z) + (y * z)

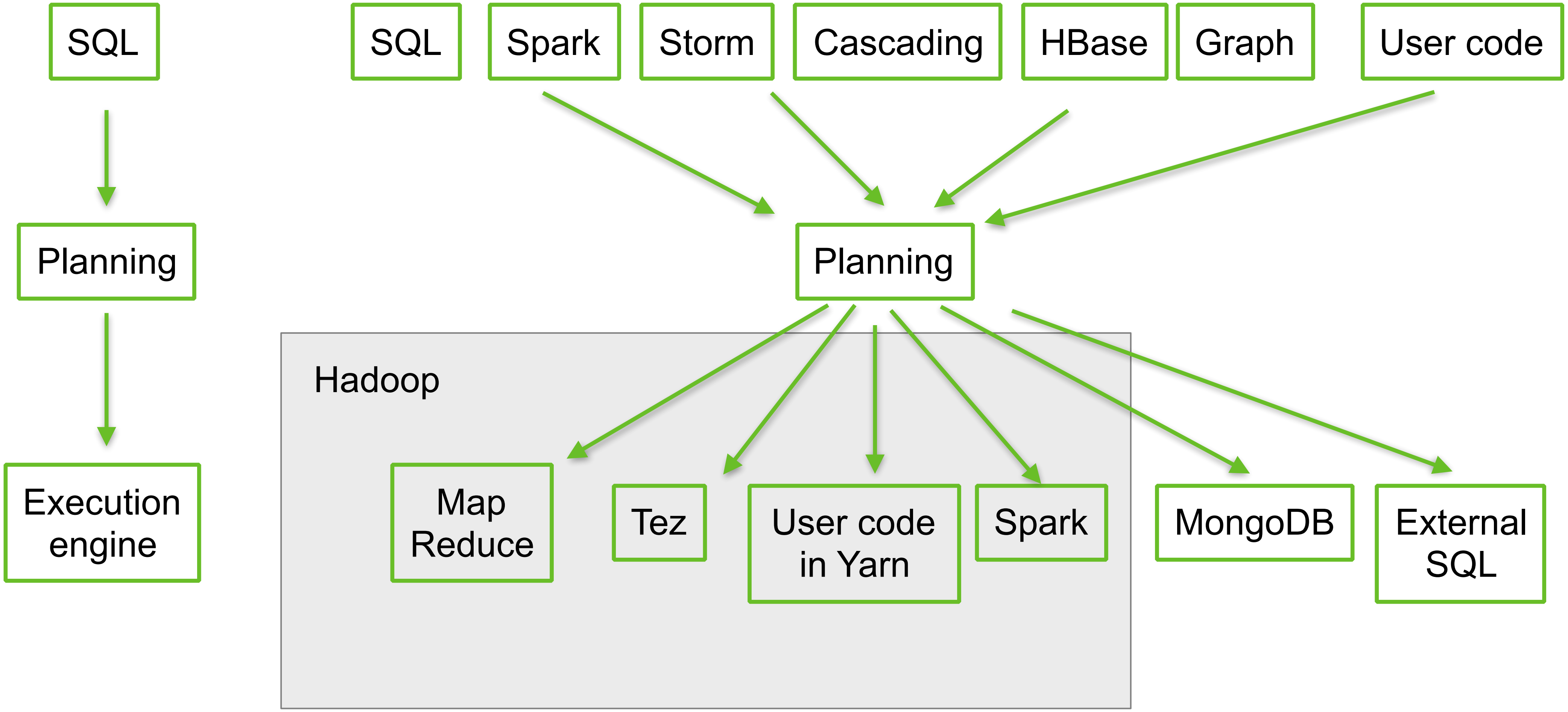
(R1 join R2 on c) filter c2 → (R1 filter c2) join R2 on c (provided C2 only depends on columns in E, and join is inner)

(R1 join R2 on c) → (R2 join R2 on c) project [R1.*, R2.*]

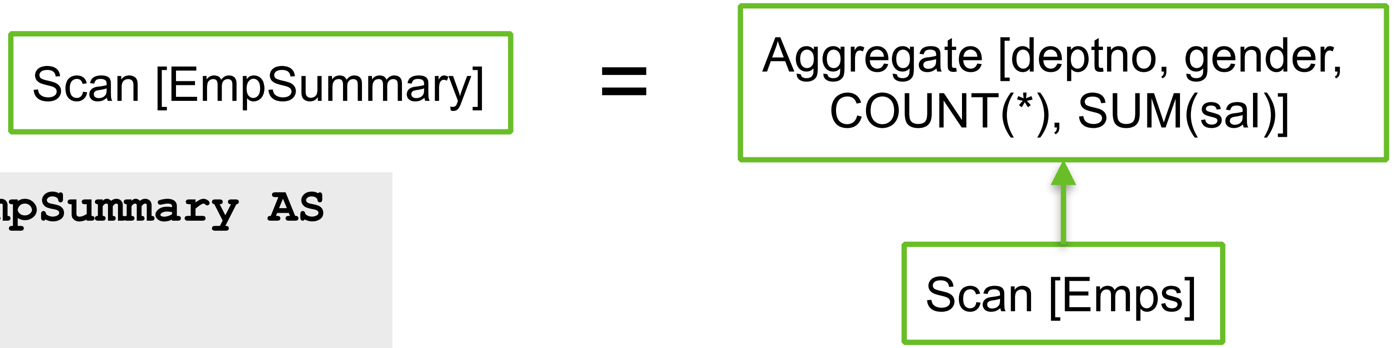
(R1 join R2 on c) join R3 on c2 → R1 join (R2 join R3 on c2) on c (provided c, c2 have the necessary columns)

Many, many others…

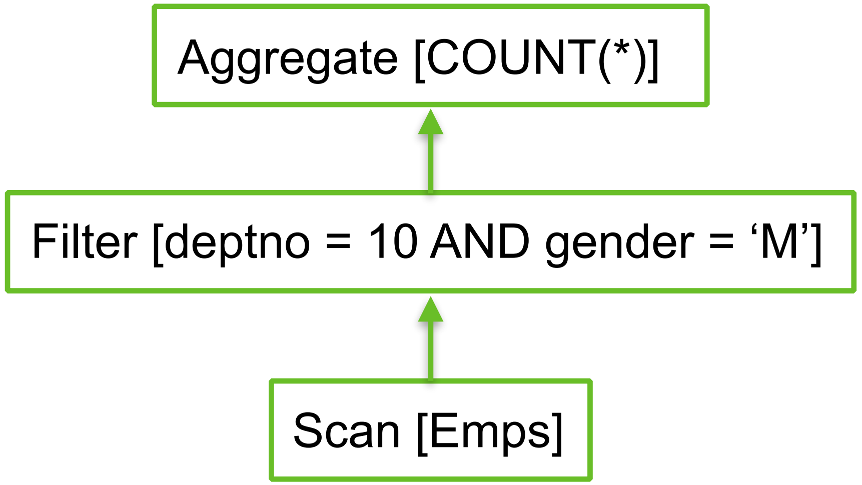
Many front ends, many engines



Materialized view

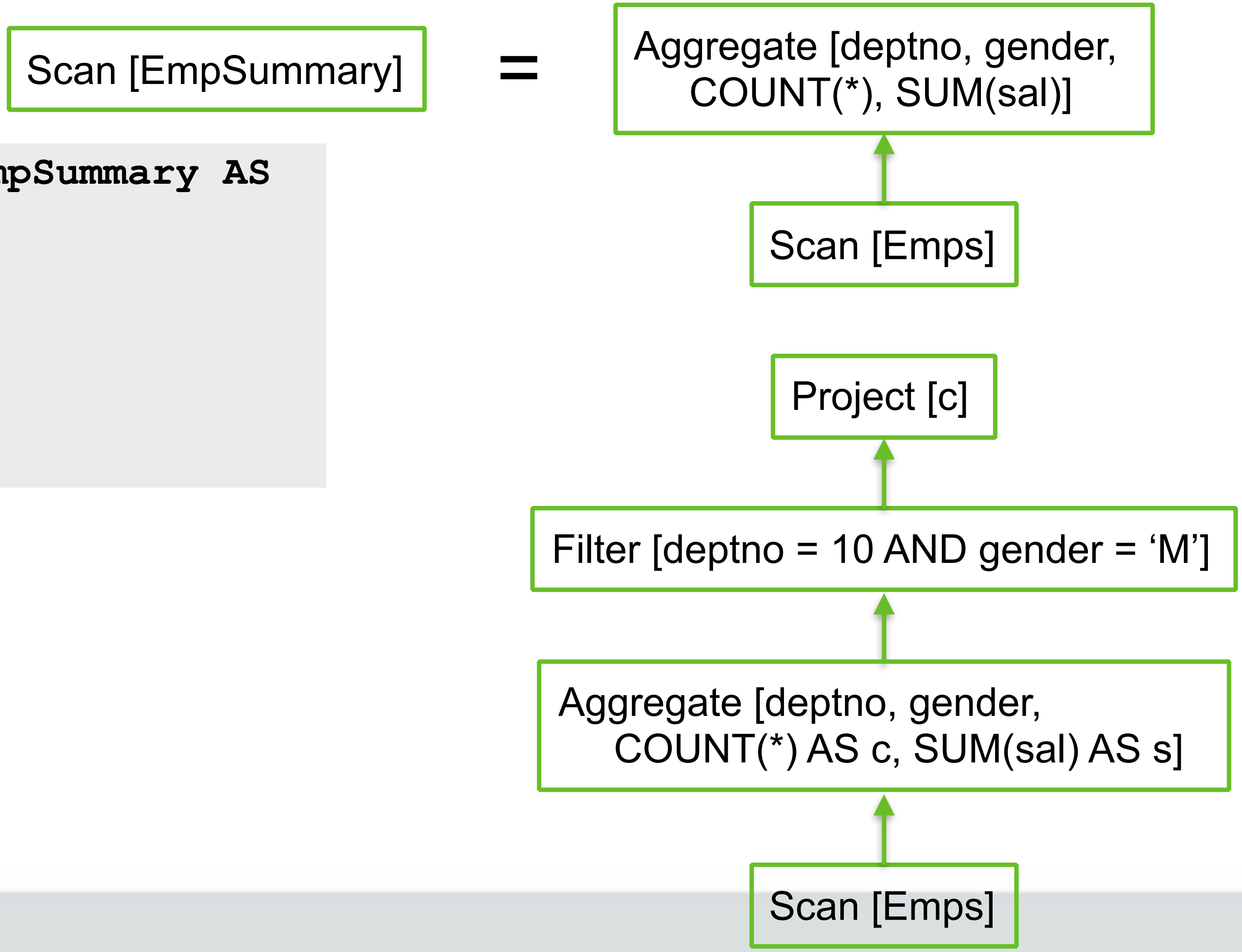



```
CREATE MATERIALIZED VIEW EmpSummary AS
SELECT deptno,
  gender,
  COUNT(*) AS c,
  SUM(sal) AS s
FROM Emps
GROUP BY deptno, gender
```




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

Materialized view, step 2: Rewrite query to match




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

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

Materialized view, step 3: Substitute table

Scan [EmpSummary] =

Aggregate [deptno, gender, COUNT(*), SUM(sal)]

Scan [Emps]

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

Project [c]

Filter [deptno = 10 AND gender = ‘M’]

Scan [EmpSummary]

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

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

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
```

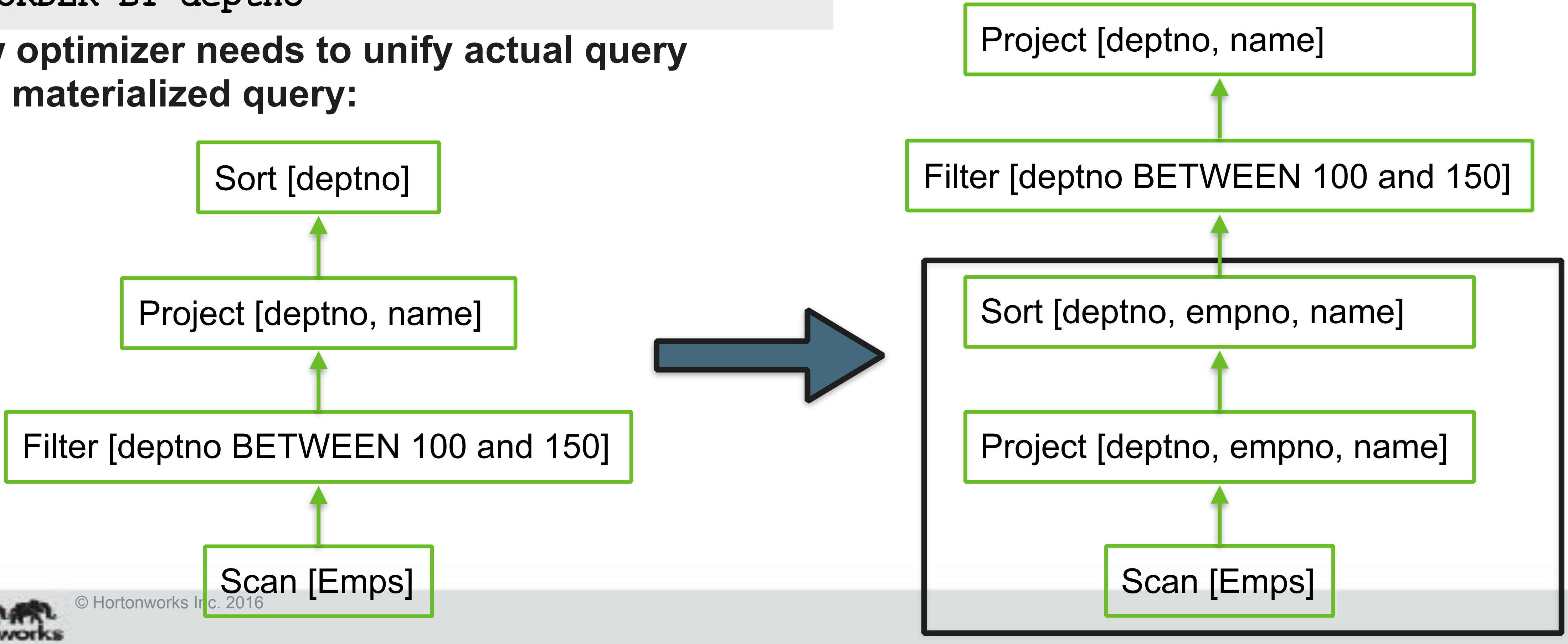
- Skip scan on leading edge of index
- No sort necessary

Modeling a 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:

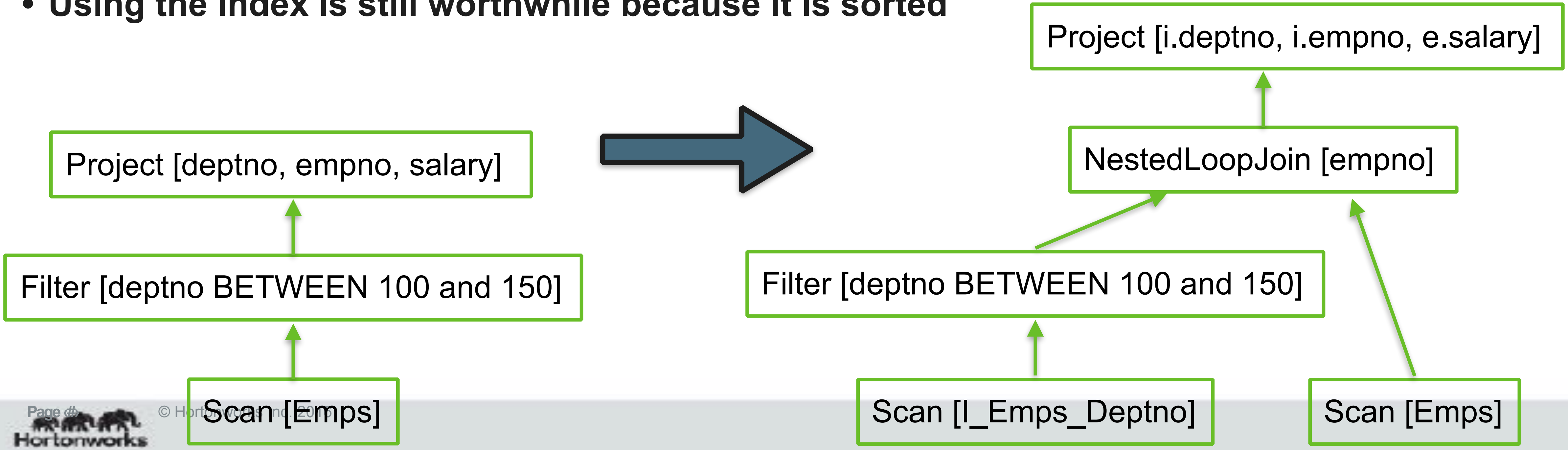


Non-covering index

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**



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
- 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

@julianhyde

@ApacheCalcite

Extra material

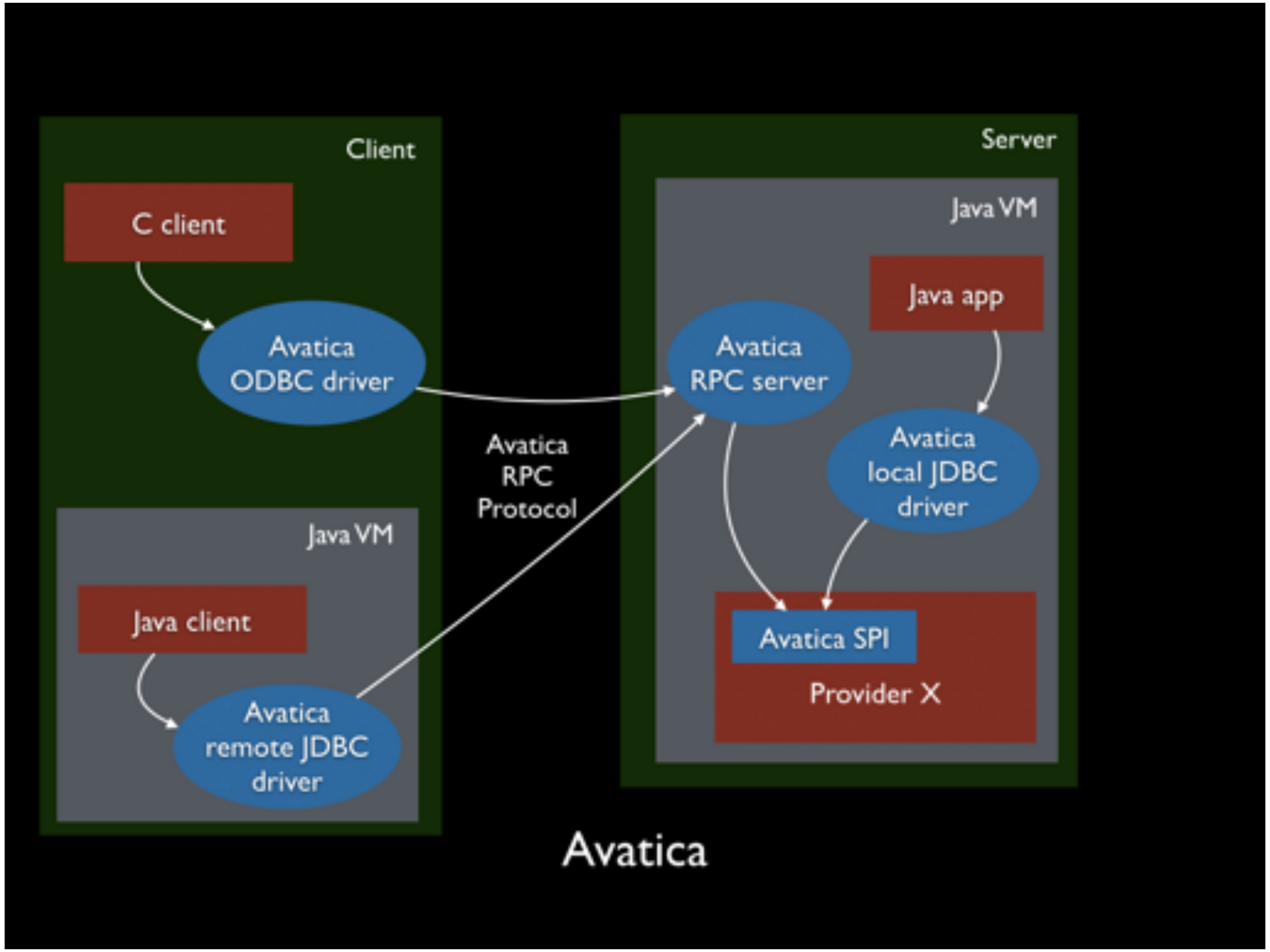
Calcite Avatica & Phoenix Query Server

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



Streaming

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

Delta

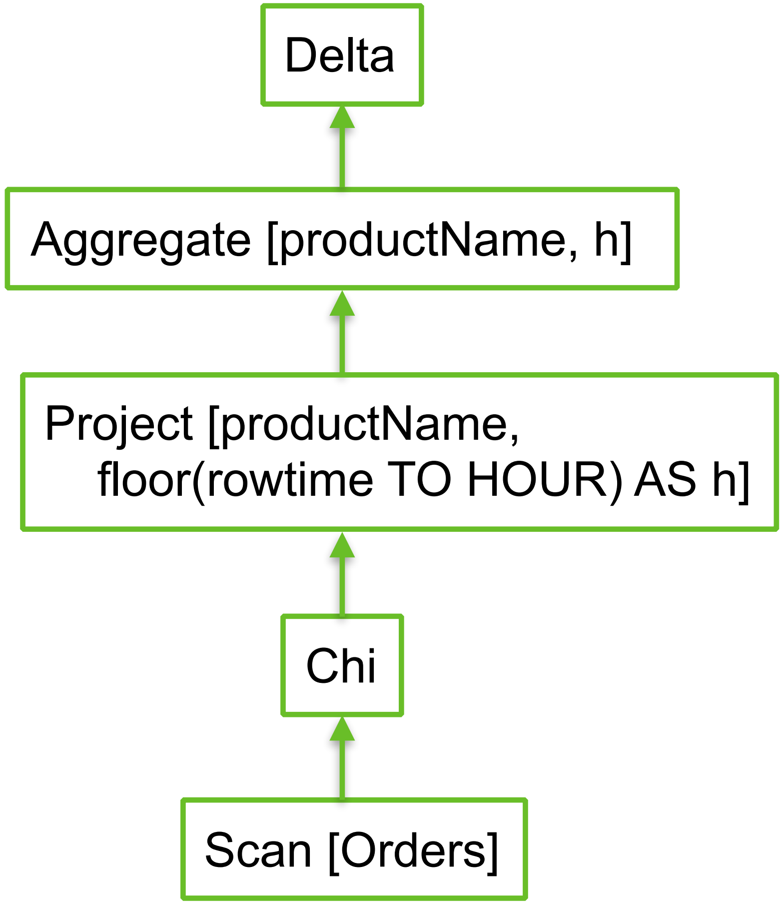
Converts a table to a stream

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

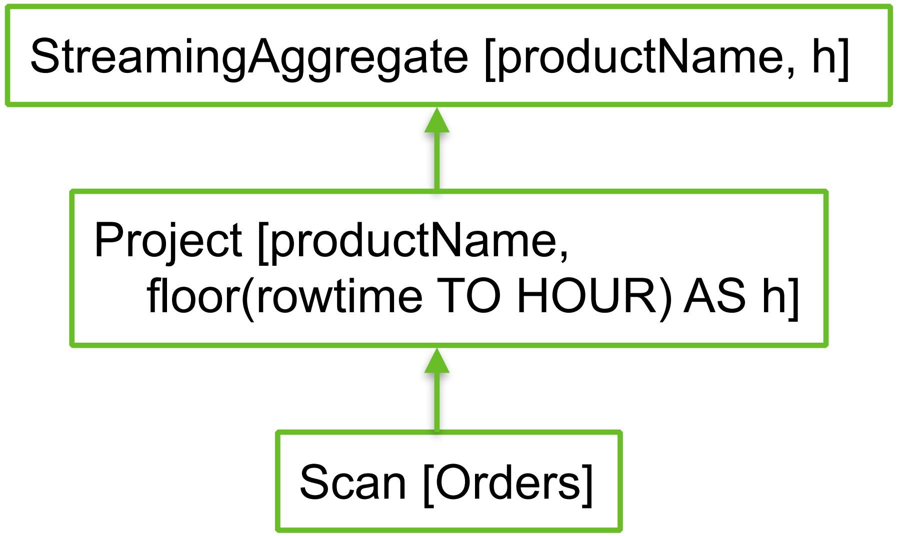


Streaming - efficient implementation

```
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))

(f + g)’ = f’ + g’

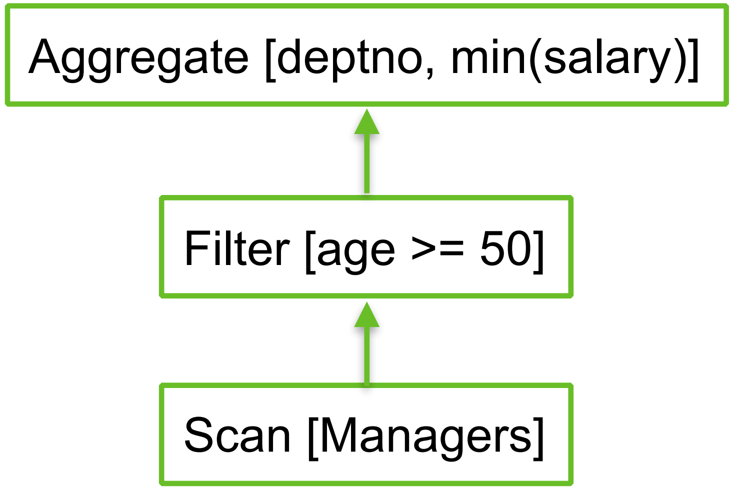
delta(join(R1, R2, c)) → union(join(R1, delta(R2), c),
join(delta(R1), R2), c)

(f . g)’ = f.g’ + f’.g

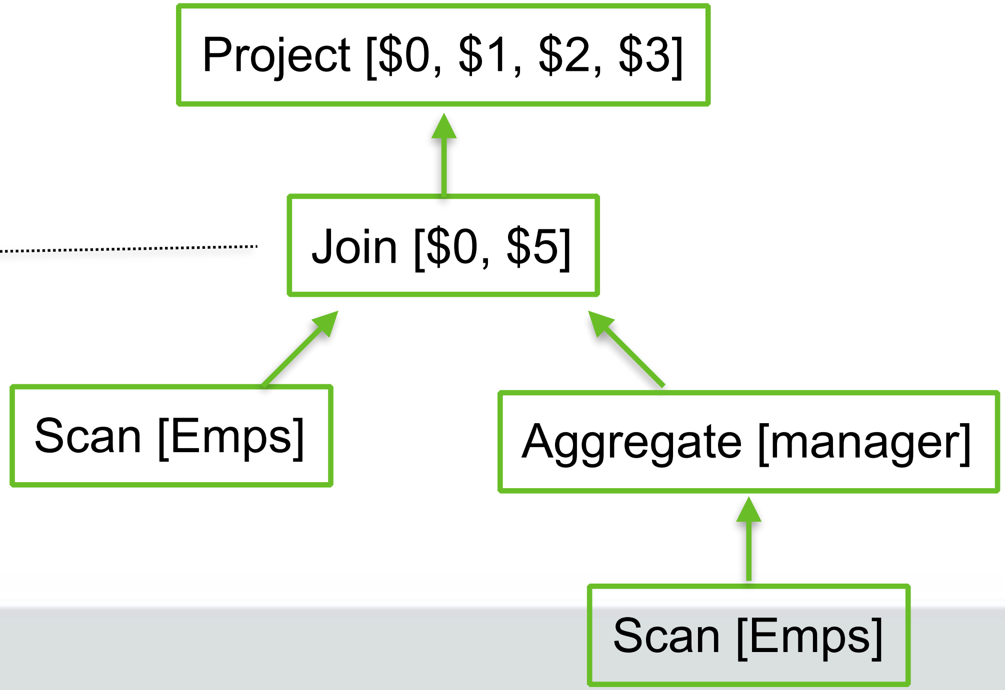
Delta behaves like “differentiate” in differential calculus,
Chi like “integrate”.

Query using a view

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

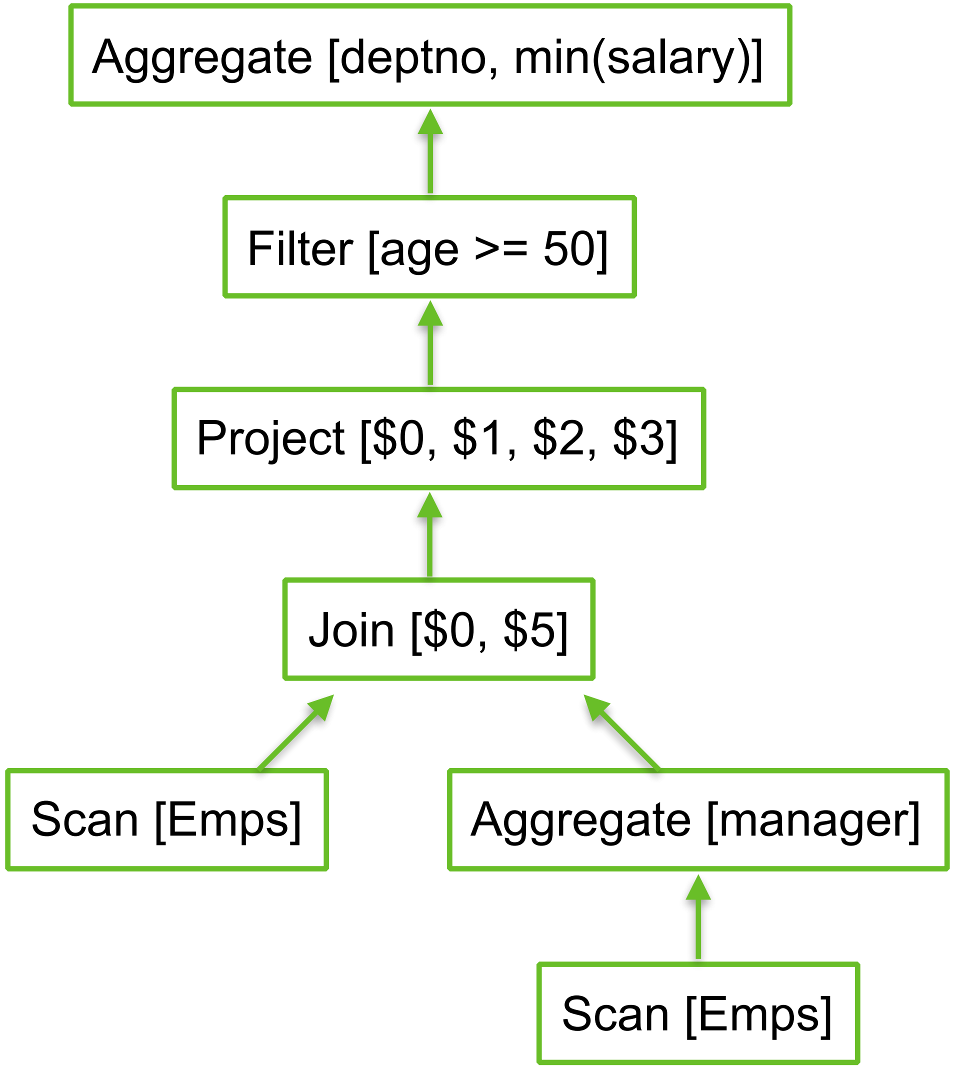



```
CREATE VIEW Managers AS
SELECT *
FROM Emps
WHERE EXISTS (
  SELECT *
  FROM Emps AS underling
  WHERE underling.manager = emp.id)
```



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)
```