Apache Calcite

Julian Hyde
Salesforce.com
November 11, 2015

"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

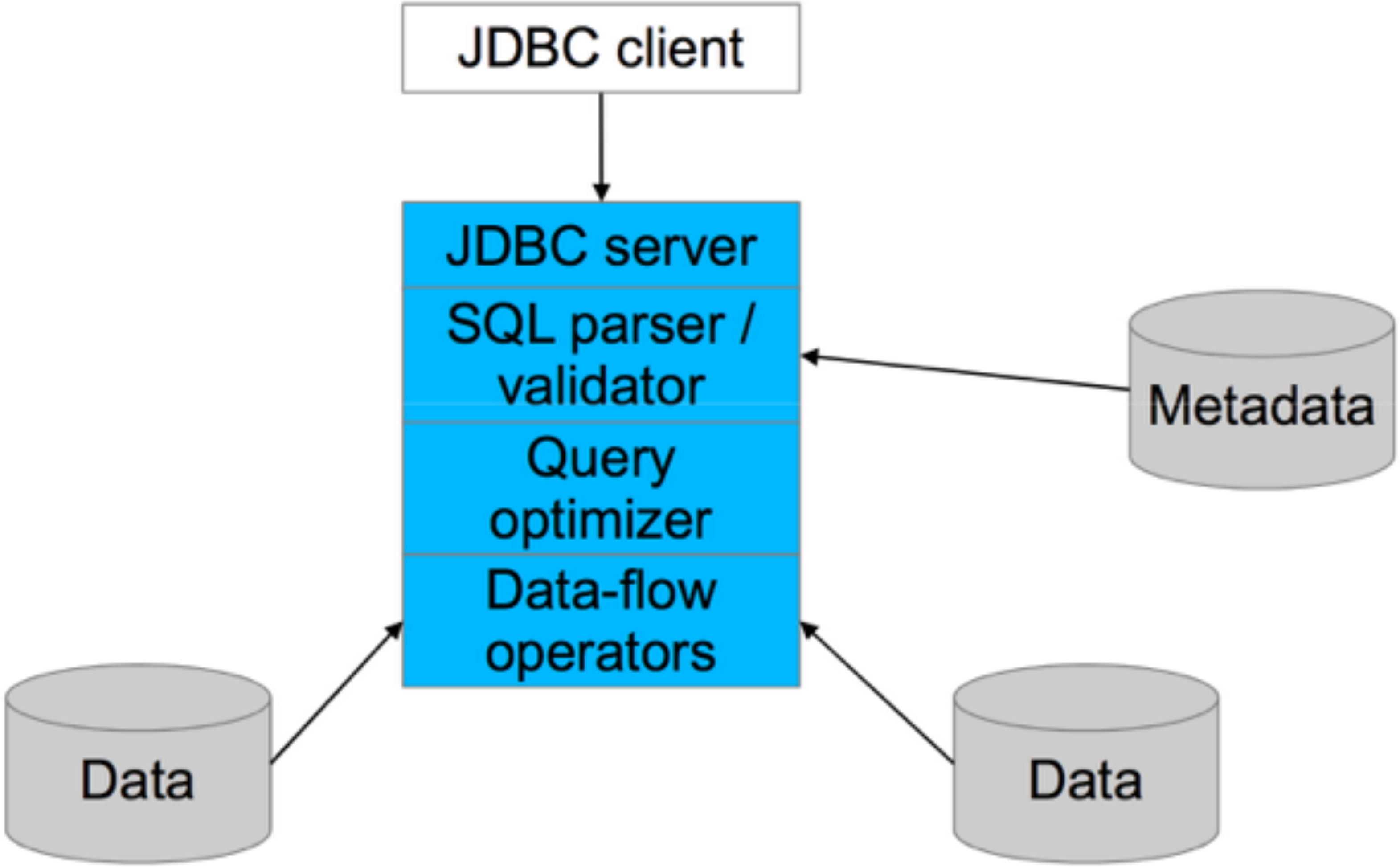
Packaging

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

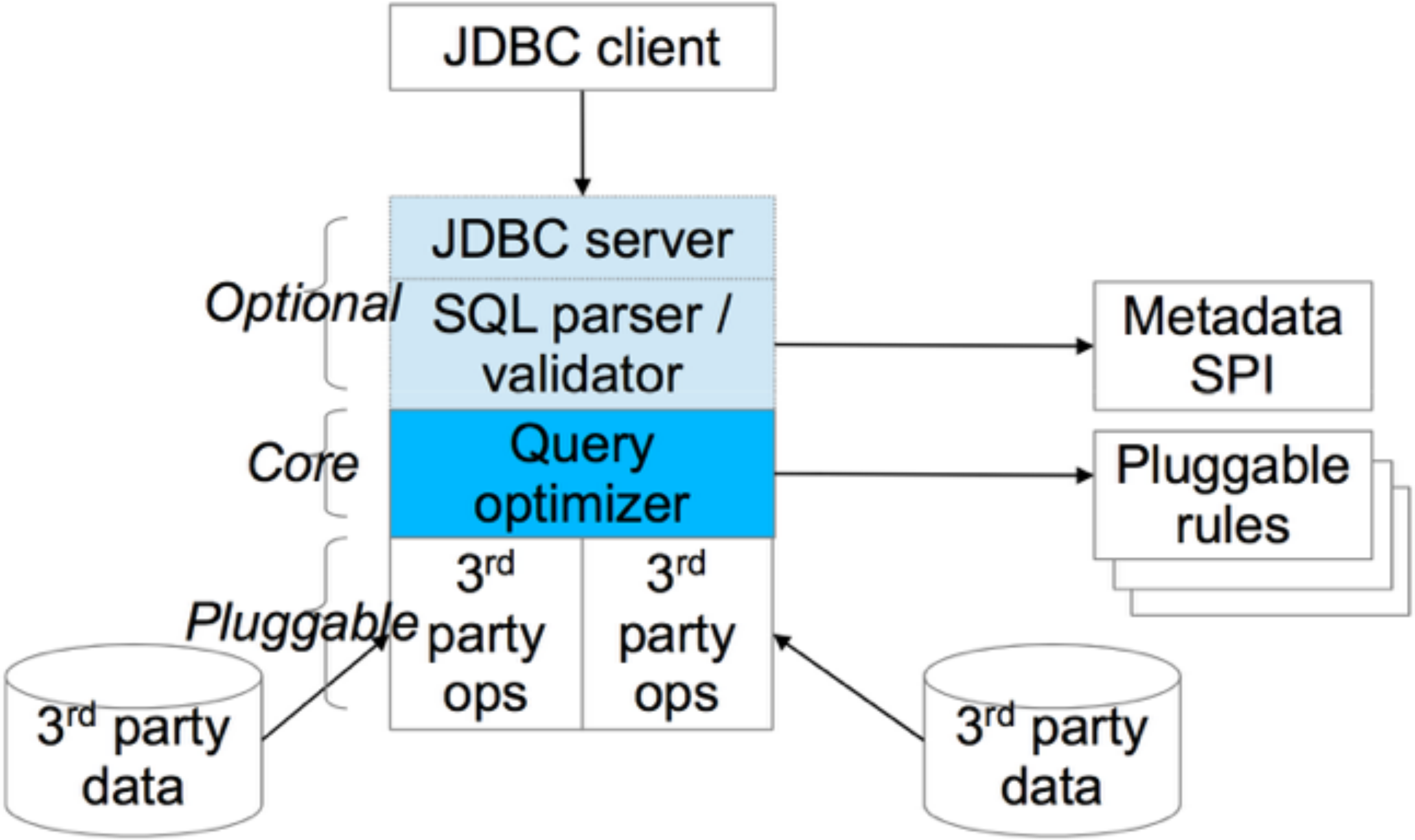
Adoption

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

Conventional DB architecture



Calcite architecture

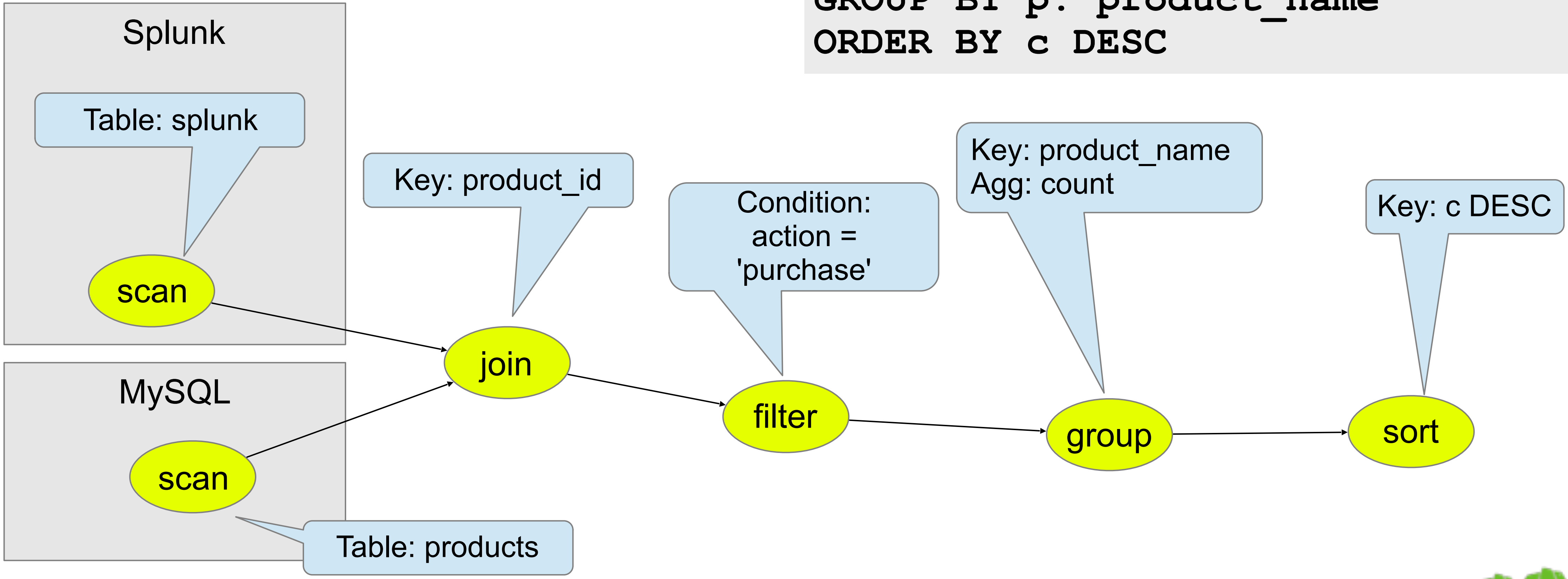


Demo

{sqlline, apache-calcite-1.5, .csv}

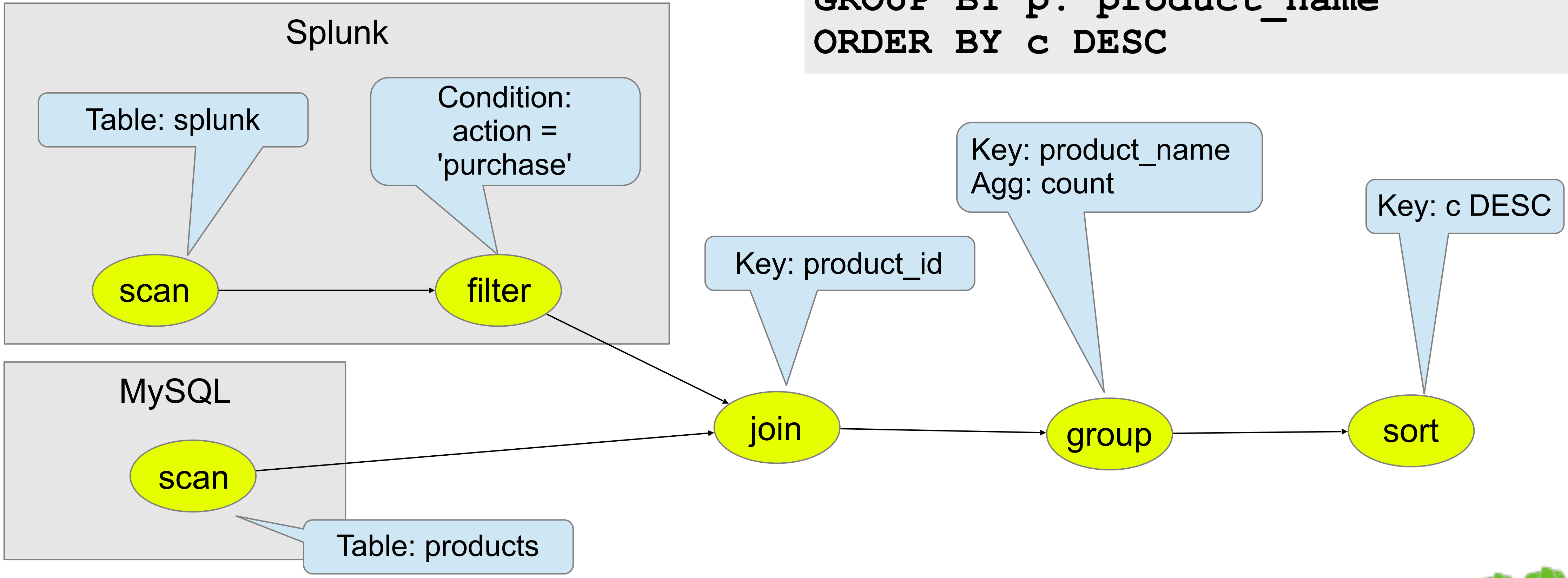
Expression tree

```
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

Hortonworks

Calcite Planning Process

SqlNode → RelNode + RexNode

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

Logical Plan

Planner

- 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

Rule Match Queue

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

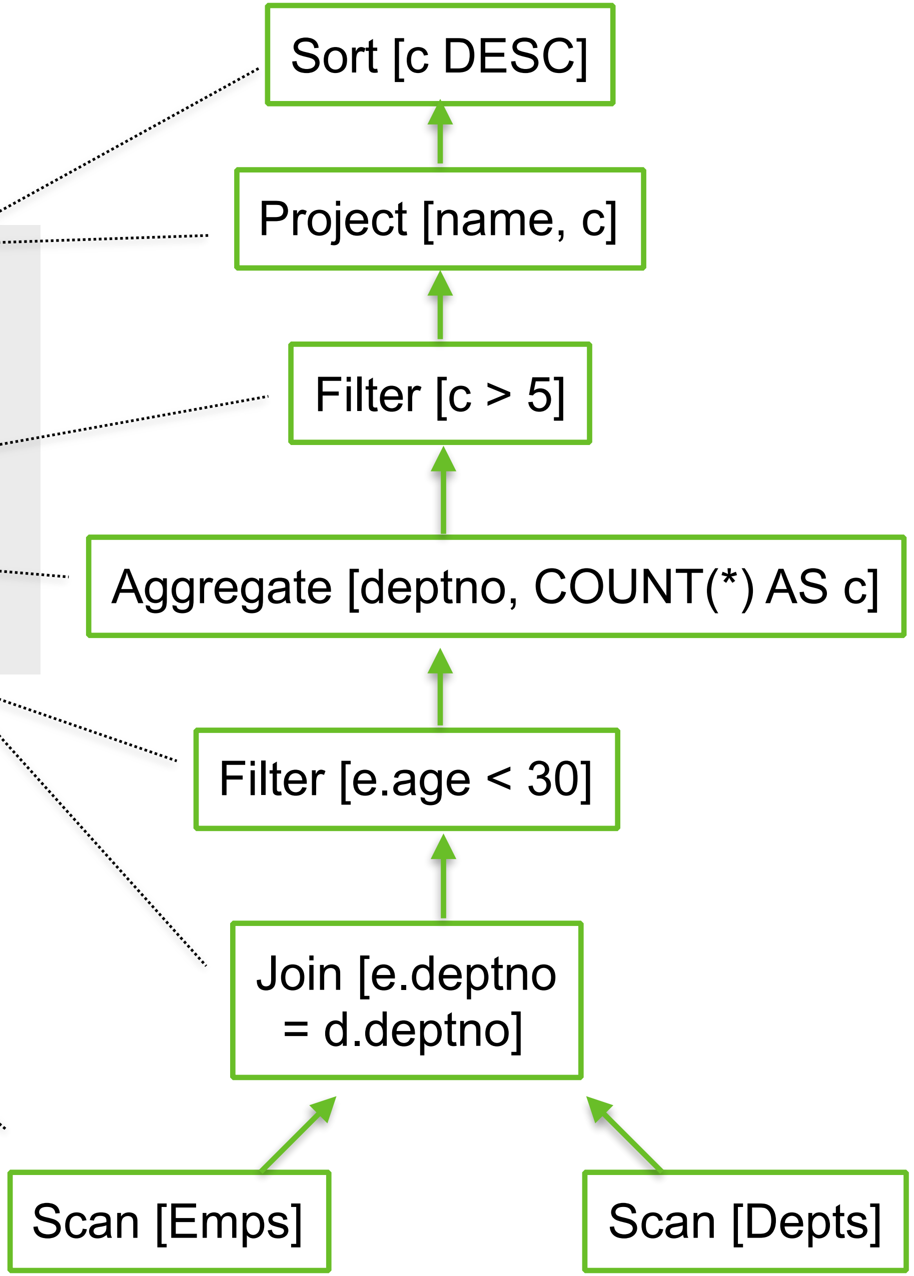
Based on "Volcano" & "Cascades" papers [G. Graefe]

Hortonworks

Core concept #1: Relational algebra

Relational algebra

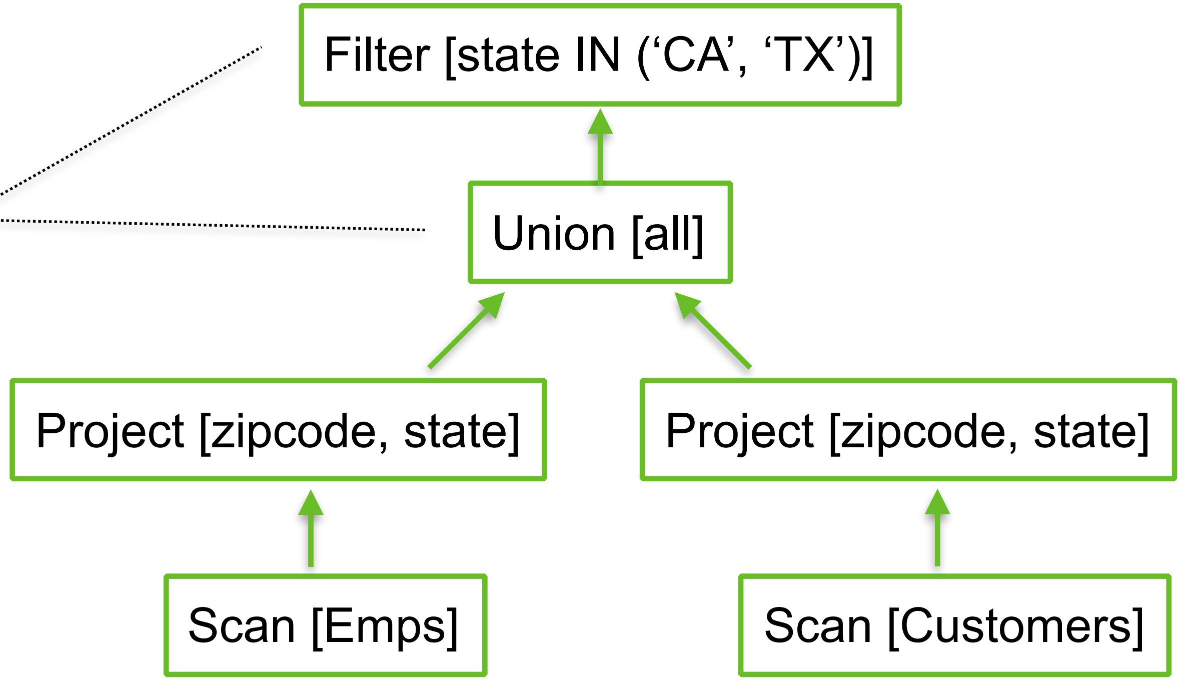
```
SELECT d.name, COUNT(*) AS c
FROM Emps AS e
  JOIN Depts AS d ON e.deptno = d.deptno
WHERE e.age < 30
GROUP BY d.deptno
HAVING COUNT(*) > 5
ORDER BY c DESC
```



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

Relational algebra - Union and sub-query

```
SELECT * FROM (
  SELECT zipcode, state
  FROM Emps
  UNION ALL
  SELECT zipcode, state
  FROM Customers)
WHERE state IN ('CA', 'TX')
```



Relational algebra - Insert and Values

```
INSERT INTO Facts
VALUES ('Meaning of life', 42),
  ('Clever as clever', 6)
```

Insert [Facts]

Values [['Meaning of life', 42],
['Clever as clever', 6]]

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)

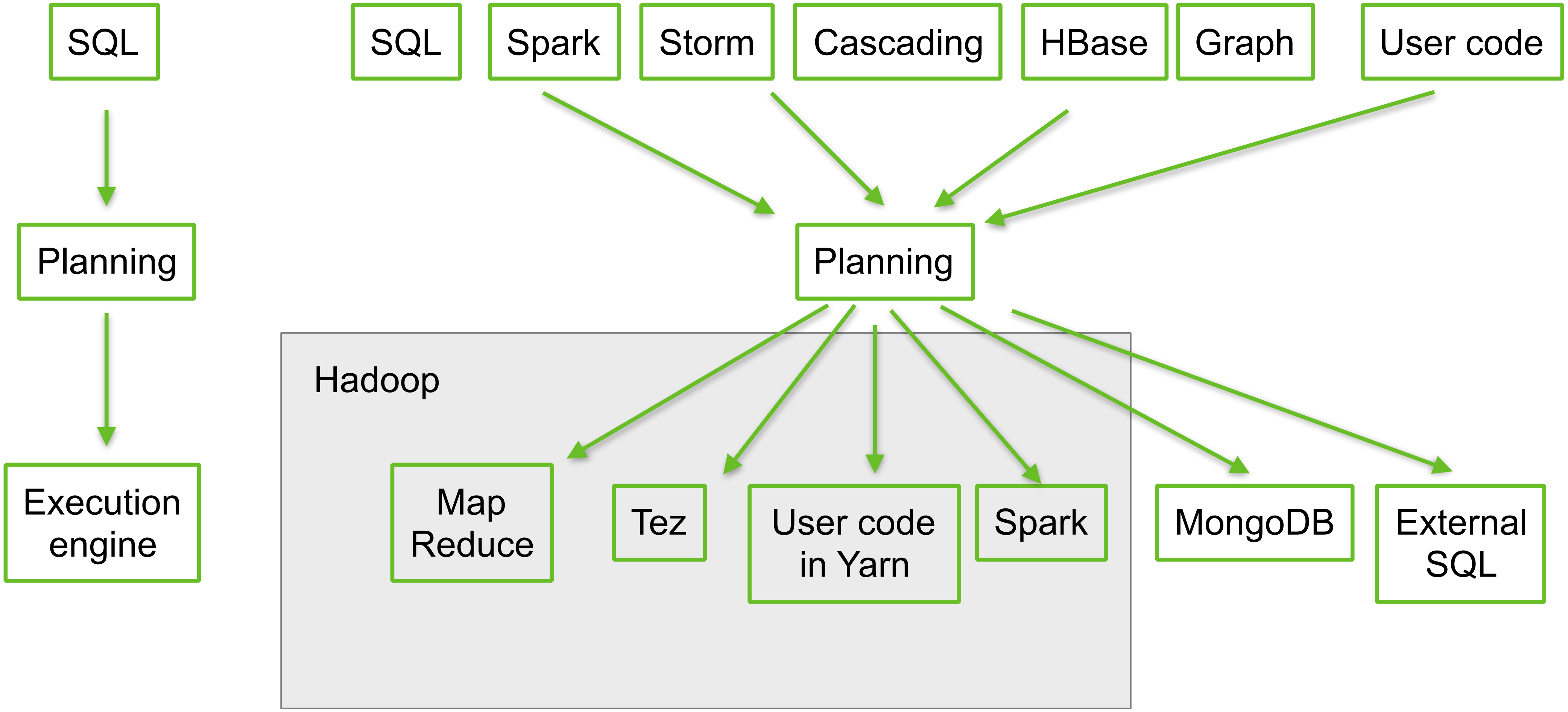
(R1 join R2 on c) filter c2 → (R1 filter c2) join R2 on c (provided C2 only depends on columns in E, and join is inner)

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

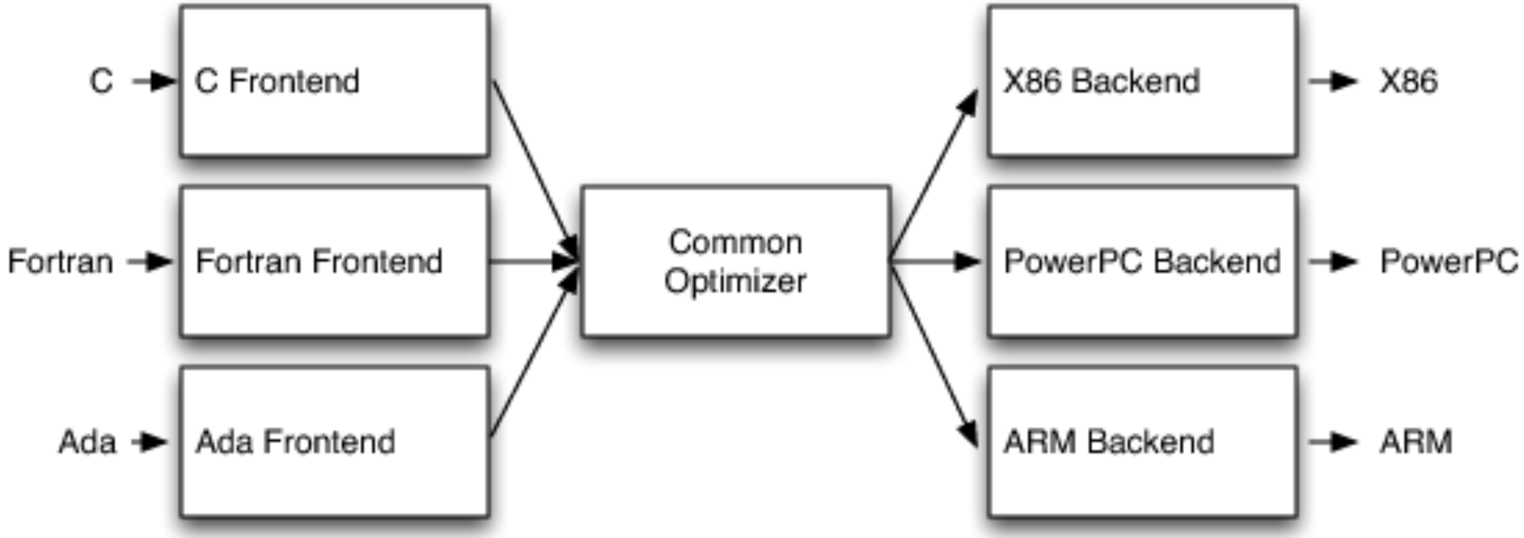
(R1 join R2 on c) join R3 on c2 → R1 join (R2 join R3 on c2) on c (provided c, c2 have the necessary columns)

Many, many others…

Many front ends, many engines



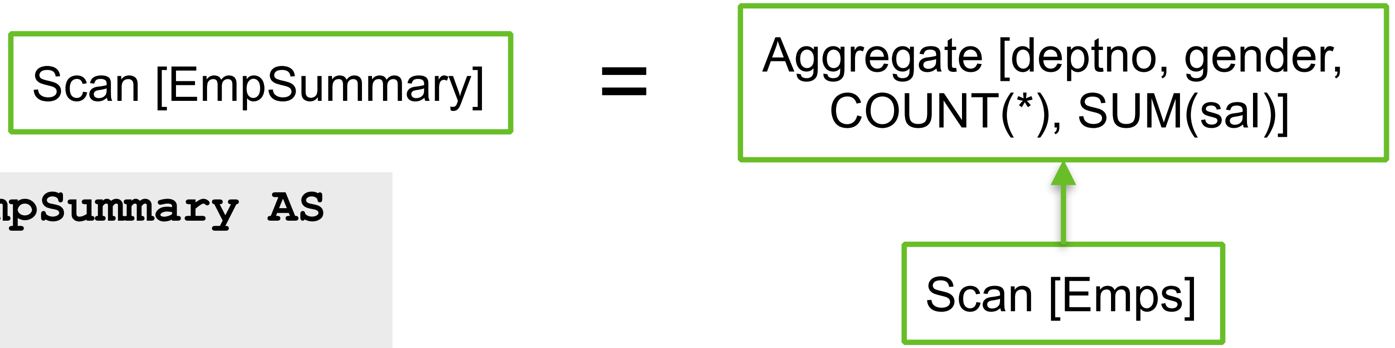
Analogy: LLVM



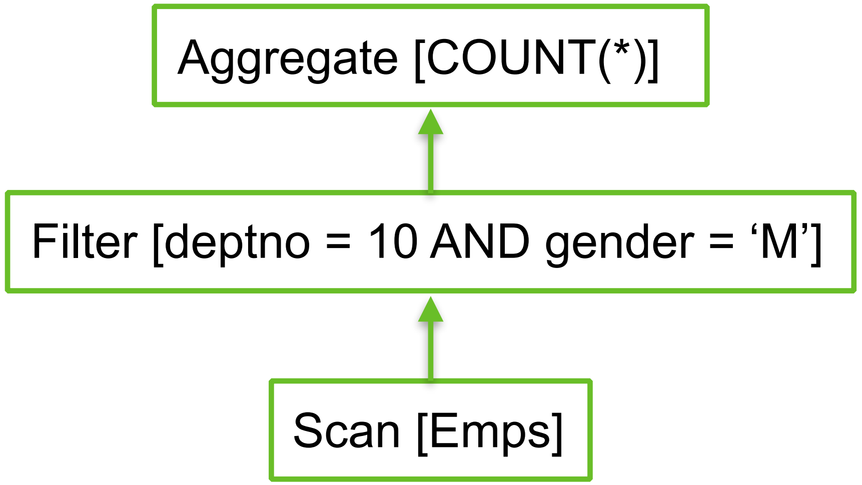
Lessons from the compiler community:

- **Writing a front end is hard**
- **Writing a back end is hard**
- **Writing an optimizer is *really* hard**
- **Most of the logic in the optimizer is independent of front end and back end**
  - **E.g. register assignment**
- **The optimizer is a collection of separate algorithms**
- **Common language between algorithms**

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

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

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
```

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

Project [c]

Filter [deptno = 10 AND gender = ‘M’]

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

Scan [Emps]

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'
```

Core concept #2: Data independence

Hortonworks

Data independence

A core principle of data management

Data independence is a contract:

- Applications do not make assumptions about the location or organization of data
- The DBMS chooses the most efficient access path

Requires:

- Declarative query language
- Query planner

Allows:

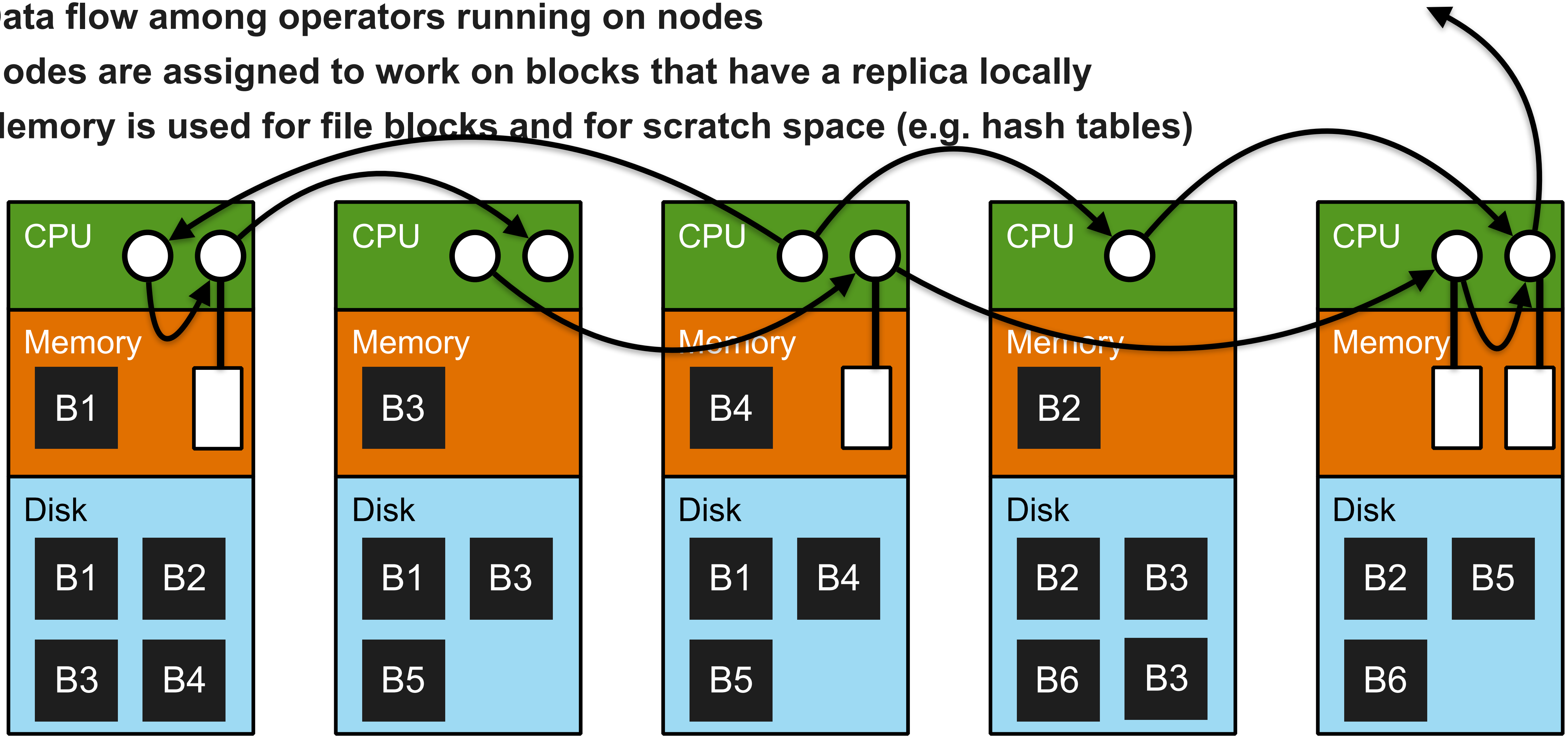
- The DBMS (or administrator) can re-organize the data without breaking the application
- Redundant copies of the data (indexes, materialized views, replicas)
- Novel algorithms
- Novel data formats and organizations (e.g. b-tree, r-tree, column store)

Hadoop query execution

Data flow among operators running on nodes

Nodes are assigned to work on blocks that have a replica locally

Memory is used for file blocks and for scratch space (e.g. hash tables)



Hortonworks

Data independence and Hadoop

Hadoop is very flexible when data is loaded

That flexibility has made it hard for the system to optimize access

Data independence saves the day:

- Make the *system* aware of the data layout without bothering the *application*
- A common trick is to “crack” the data, and create copies in other formats
- Materialized views tell the system about the various copies

Theory into practice

Theory into practice

Bringing data independence to Hadoop

Automatic summary tables to speed up OLAP queries

Optimizing Phoenix queries to use secondary indexes

Piglet: “Pig on anything”

Streaming

Simple analytics problem?

System

100M US census records

1KB each record, 100GB total

4 SATA 3 disks, total read throughput 1.2GB/s

Requirement

Count all records in < 5s

Solution #1

It's not possible! It takes 80s just to read the data

Solution #2

Cheat!

How to cheat

Multiple tricks

Compress data

Column-oriented storage

Store data in sorted order

Put data in memory

Cache previous results

Pre-compute (materialize) aggregates

Common factors

Make a copy of the data

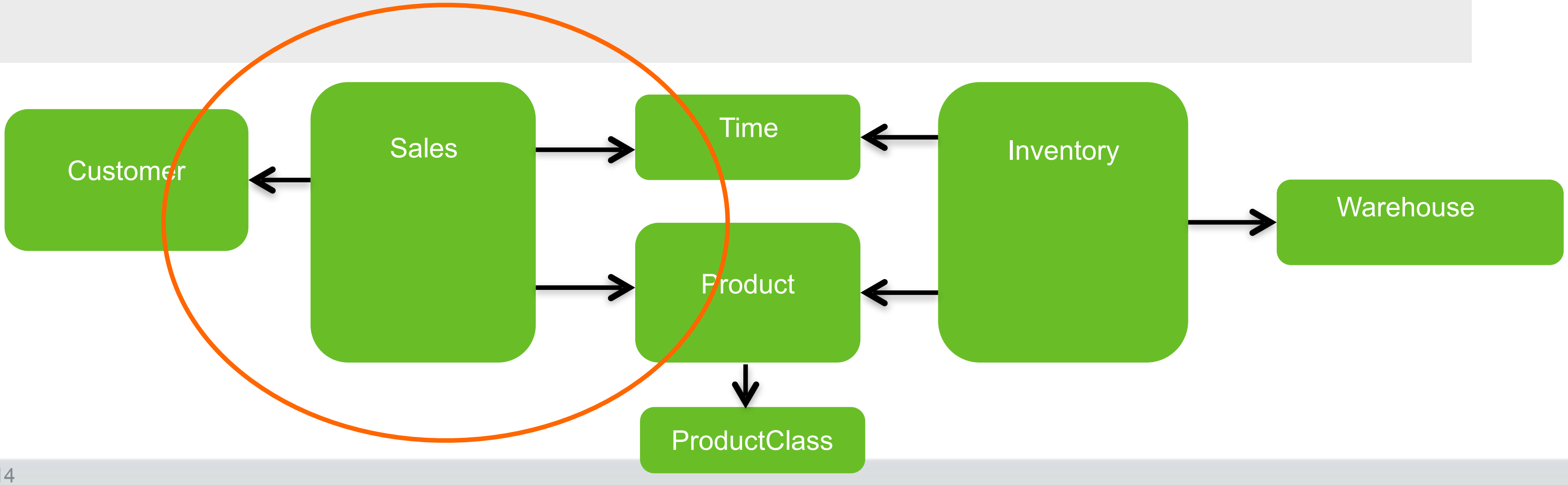
Organize it in a different way

Optimizer chooses the most suitable data organization

SQL query is unchanged

Filter-join-aggregate query

```
SELECT product.id,  sum(sales.units), sum(sales.price), count(*)
FROM sales …
JOIN customer ON …
JOIN time ON …
JOIN product ON …
JOIN product_class ON …
WHERE time.year = 2014
AND time.quarter = 'Q1'
AND product.color = 'Red'
GROUP BY …
```



Materialized view, lattice, tile

Materialized view

A table whose contents are guaranteed to be the same as executing a given query.

Lattice

Recommends, builds, and recognizes summary materialized views (tiles) based on a star schema.

A query defines the tables and many:1 relationships in the star schema.

Tile

A summary materialized view that belongs to a lattice.

A tile may or may not be materialized.

Materialization methods:

- Declare in lattice
- Generate via recommender algorithm
- Created in response to query

(FAKE SYNTAX)

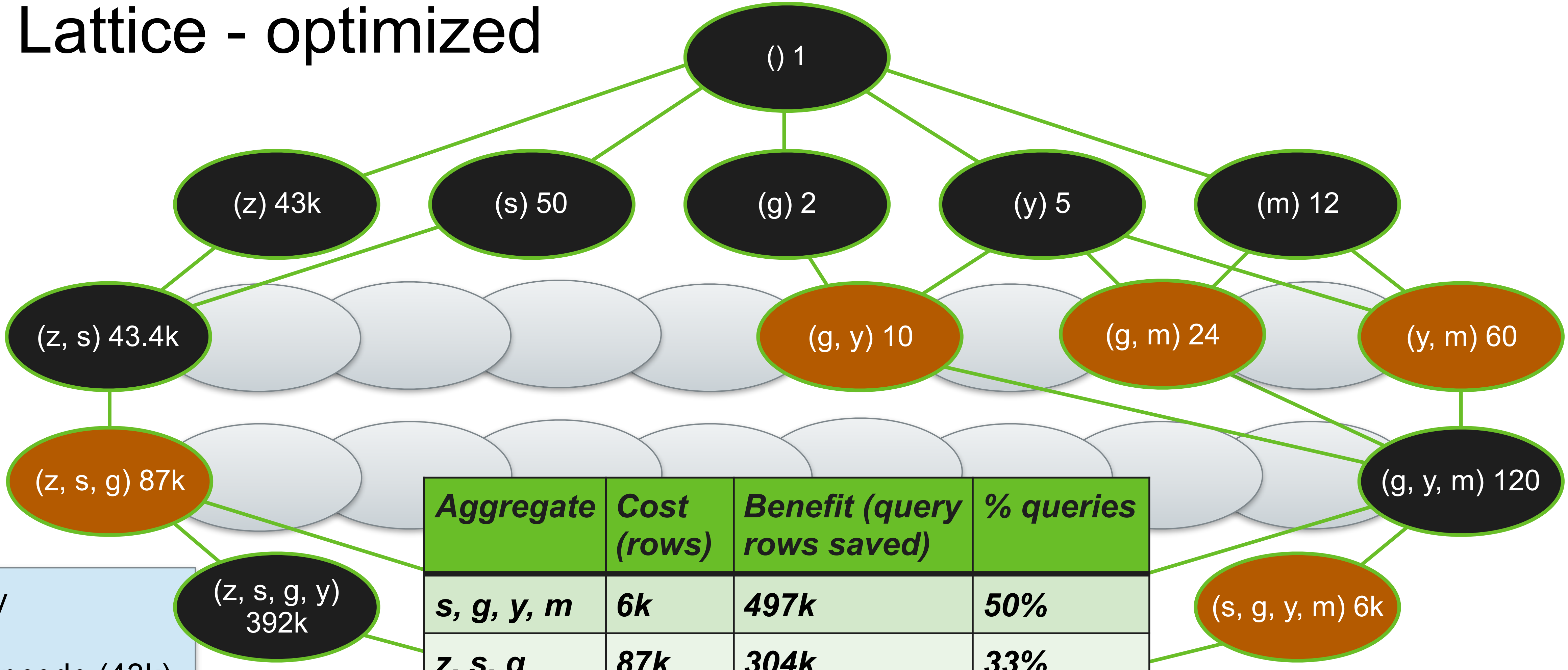
```
CREATE MATERIALIZED VIEW t AS
SELECT * FROM emps
WHERE deptno = 10;
```

```
CREATE LATTICE star AS
SELECT *
FROM sales_fact_1997 AS s
JOIN product AS p ON …
JOIN product_class AS pc ON …
JOIN customer AS c ON …
JOIN time_by_day AS t ON …;
```

```
CREATE MATERIALIZED VIEW zg IN star
SELECT gender, zipcode,
  COUNT(*), SUM(unit_sales)
FROM star
GROUP BY gender, zipcode;
```


Hortonworks

Lattice - optimized



Aggregate	Cost (rows)	Benefit (query rows saved)	% queries
s, g, y, m	6k	497k	50%
z, s, g	87k	304k	33%
g, y	10	1.5k	25%
g, m	24	1.5k	25%
s, g	100	1.5k	25%
y, m	60	1.5k	25%

Key

z zipcode (43k)
s state (50)
g gender (2)
y year (5)
m month (12)

Demo

{mysql-foodmart-lattice-model.json}

Phoenix

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

Hortonworks

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
```

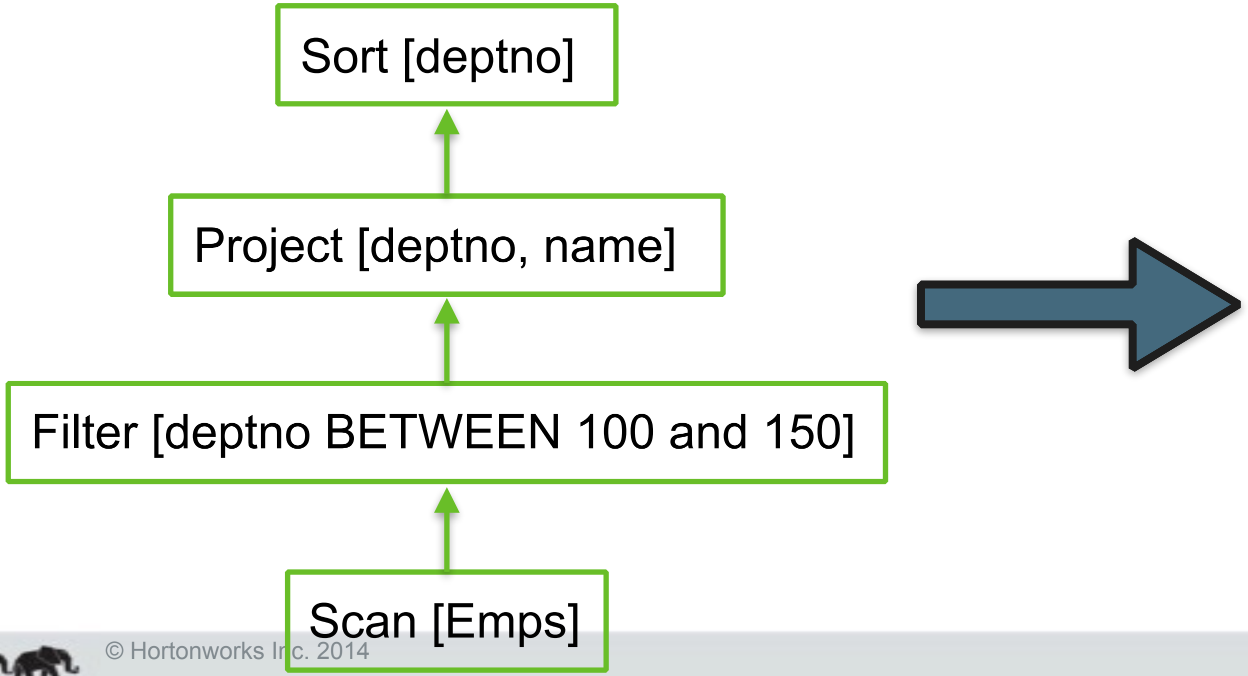
- **Skip scan on leading edge of index**
- **No sort necessary**

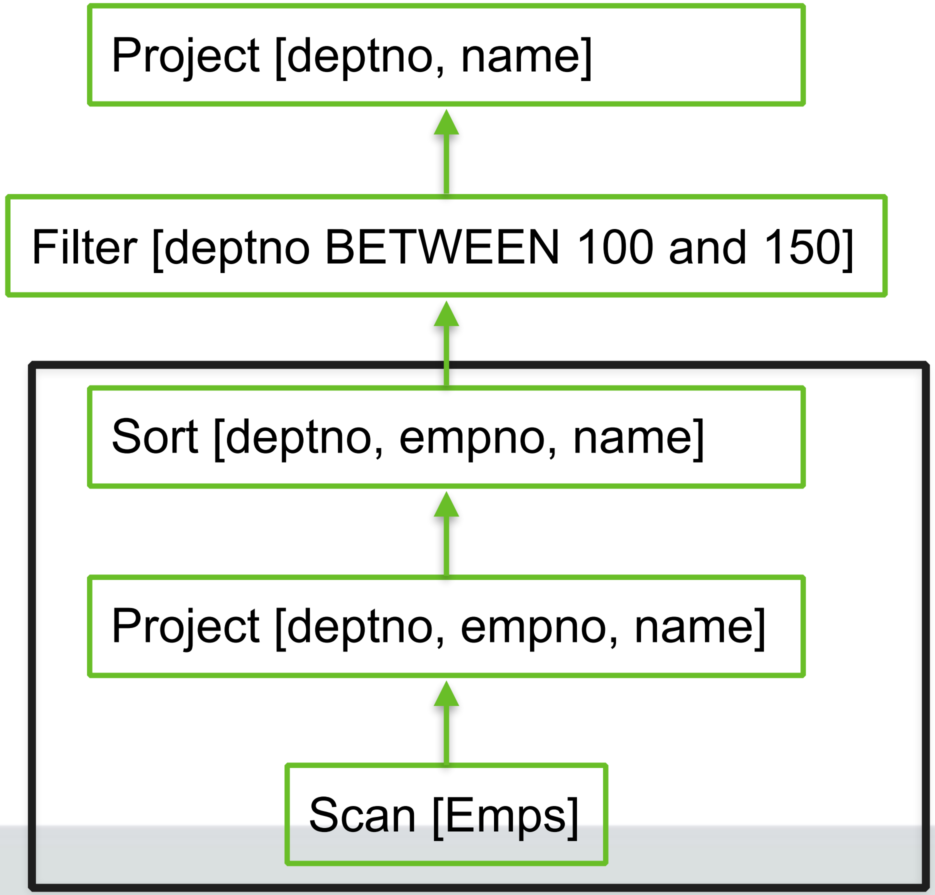
Modeling a index as a materialized view

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:



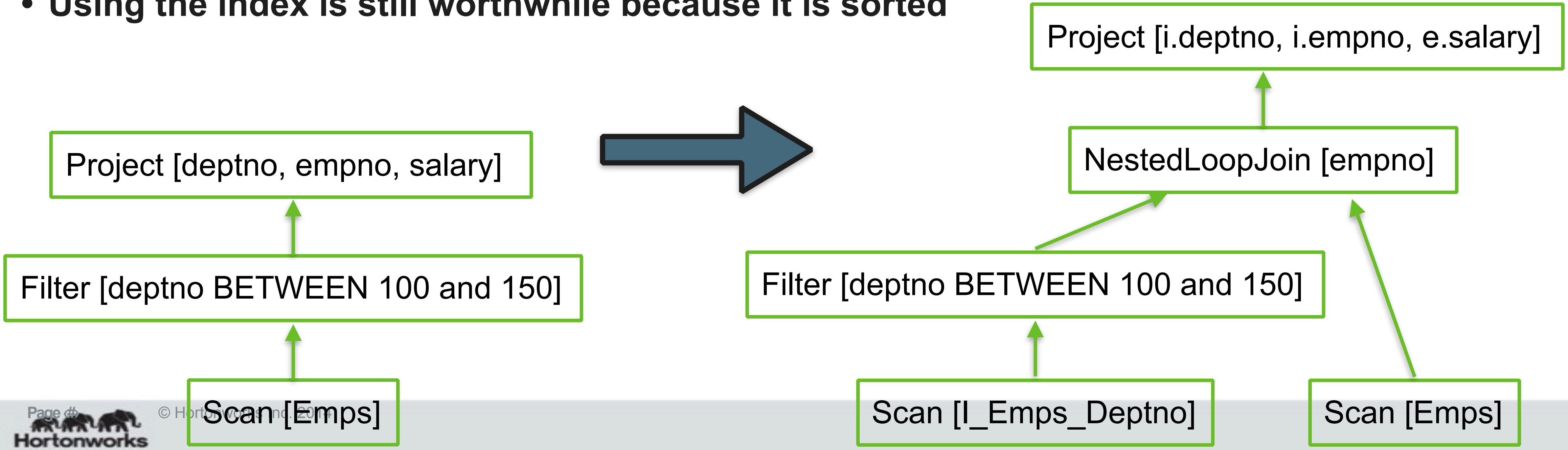


Non-covering index

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**



Pig

Pig

Apache Pig: data-flow language, high-level language on MapReduce

Operators: LOAD, FILTER, DISTINCT, GROUP BY, FOREACH … GENERATE, ORDER , LIMIT, SPLIT, UNION, DUMP

Most operators correspond to relational algebra

Interesting ones: FOREACH (nested), SPLIT, GROUP BY

```
A = LOAD 'DEPT';
B = FOREACH A GENERATE DNAME, $2;
DUMP B;
```

Piglet

Re-implementation of core Pig

Goals:

- Extend algebra for nested collections
- Run Pig on any back-end

Technology:

- PigletParser
- Extensions to RelBuilder
- One new operator: VALUES
- Existing algebra + physical algebras

```
// Piglet
A = VALUES ('John',18,4.0F),
    ('Mary',19,3.8F),
    (‘Bill',20,3.9F),
    (‘Joe',18,3.8F))
  AS (name:chararray,age:int,gpa:float);
B = GROUP A BY age;
DUMP B;

(18,{(John,18,4.0F),(Joe,18,3.8F)})
(19,{(Mary,19,3.8F)})
(20,{(Bill,20,3.9F)})


// SQL
SELECT age,
  COLLECT(ROW(name, age, gpa)) AS b
FROM Students
GROUP BY age;
```

RelBuilder

```
interface RelBuilder {
  RelBuilder push(RelNode r);
  RelNode build();

  RelBuilder scan(String tableName);
  RelBuilder filter(RexNode… conditions);
  RelBuilder aggregate(GroupKey key, AggCall…);
  RelBuilder join(RexNode… conditions);

  // extensions for Pig
  RelBuilder push(RelNode r, Map<String, RelDataType> aliases);
  RelBuilder distinct();
  RelBuilder load(String path);
  RelBuilder group(GroupOption o, Partitioner p, int  par, GroupKey… keys);
}
```

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

Streaming

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

Delta

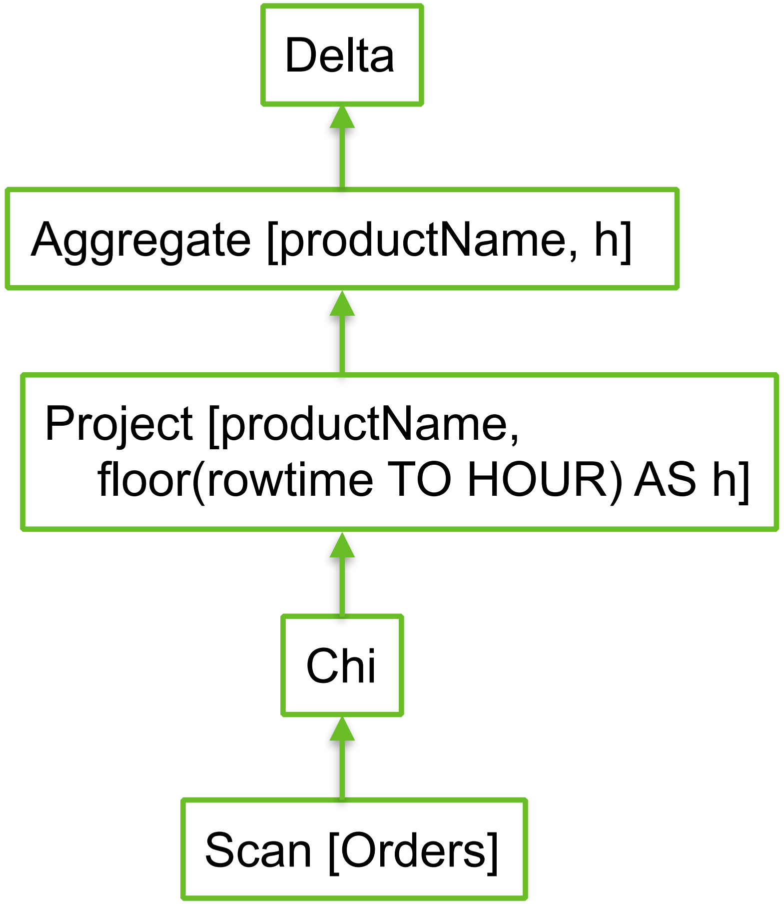
Converts a table to a stream

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

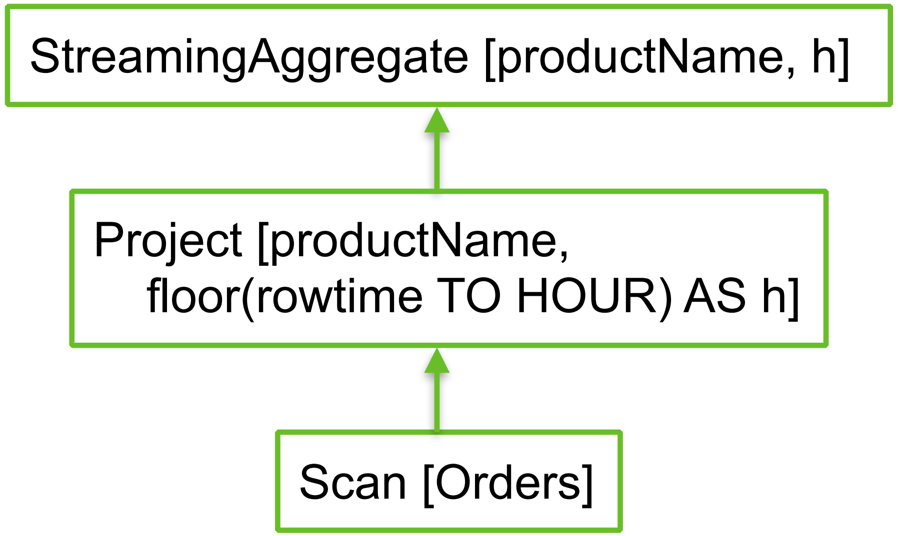


Streaming - efficient implementation

```
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”.

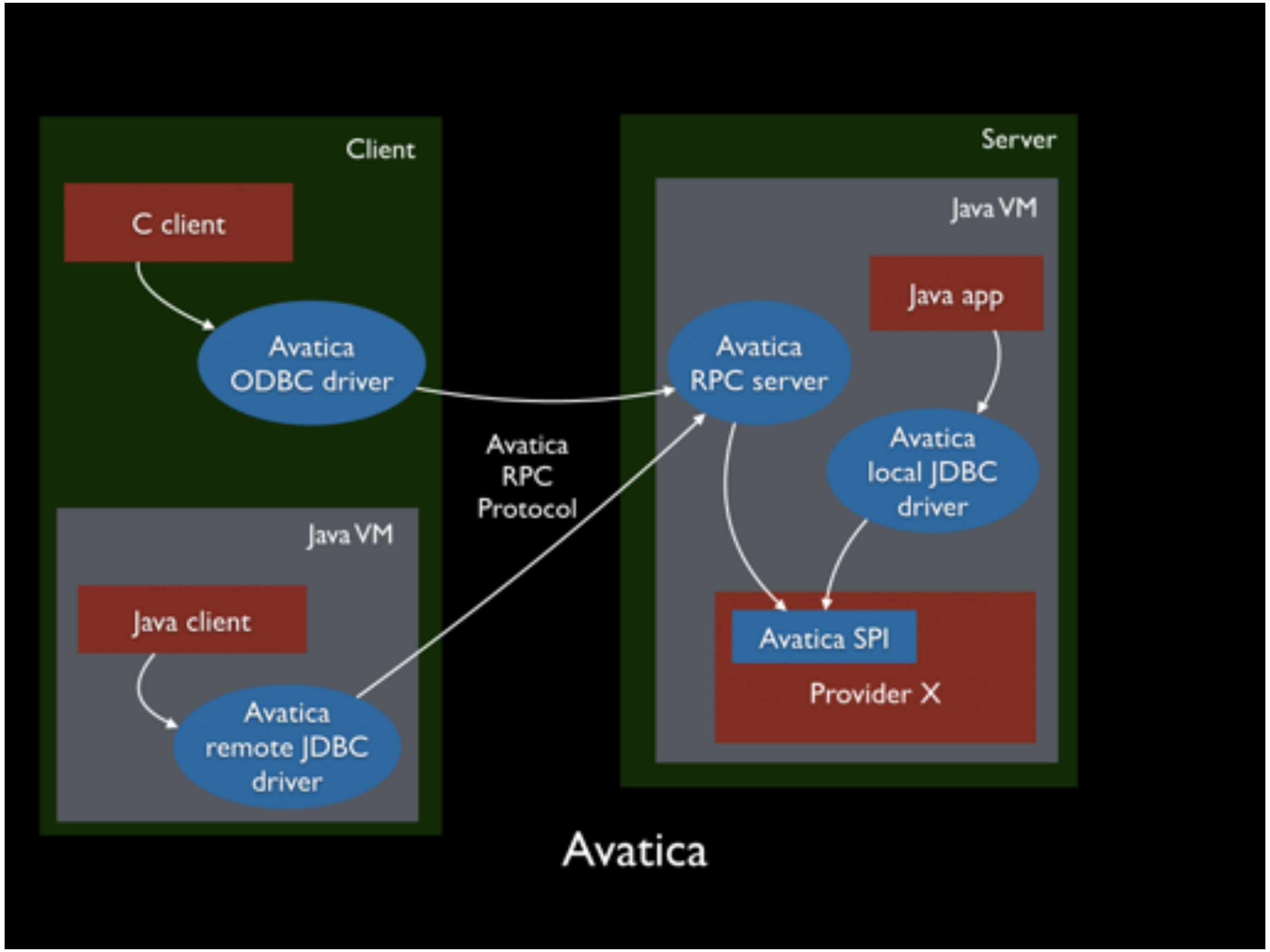
Calcite Avatica & Phoenix Query Server

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

Module within Calcite

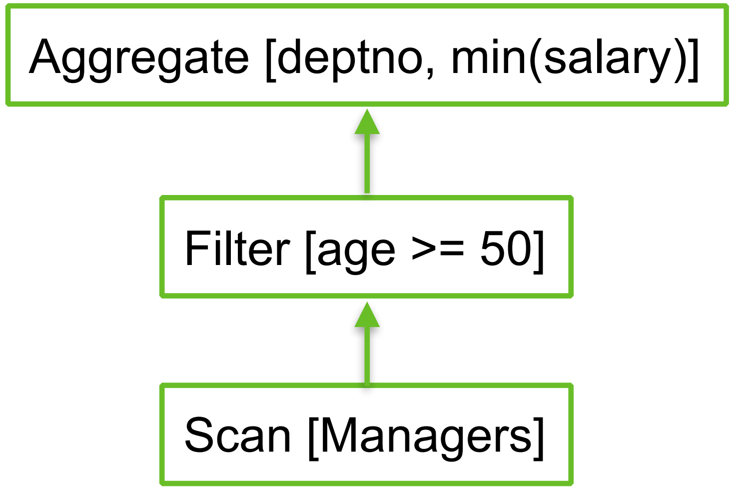
RPC: Protobuf over HTTP

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

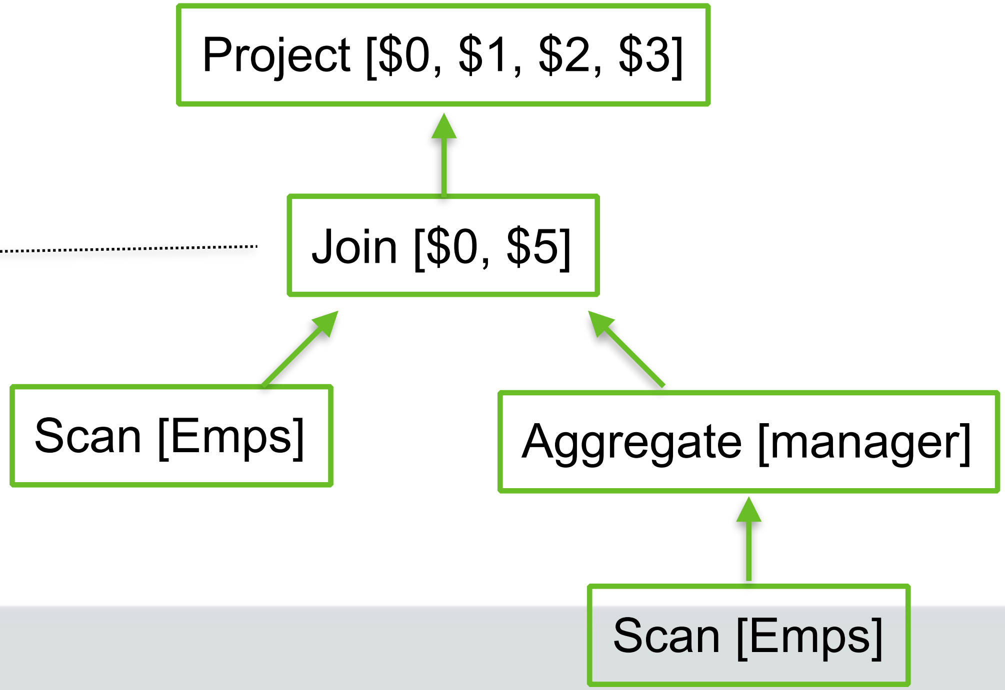


Query using a view

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



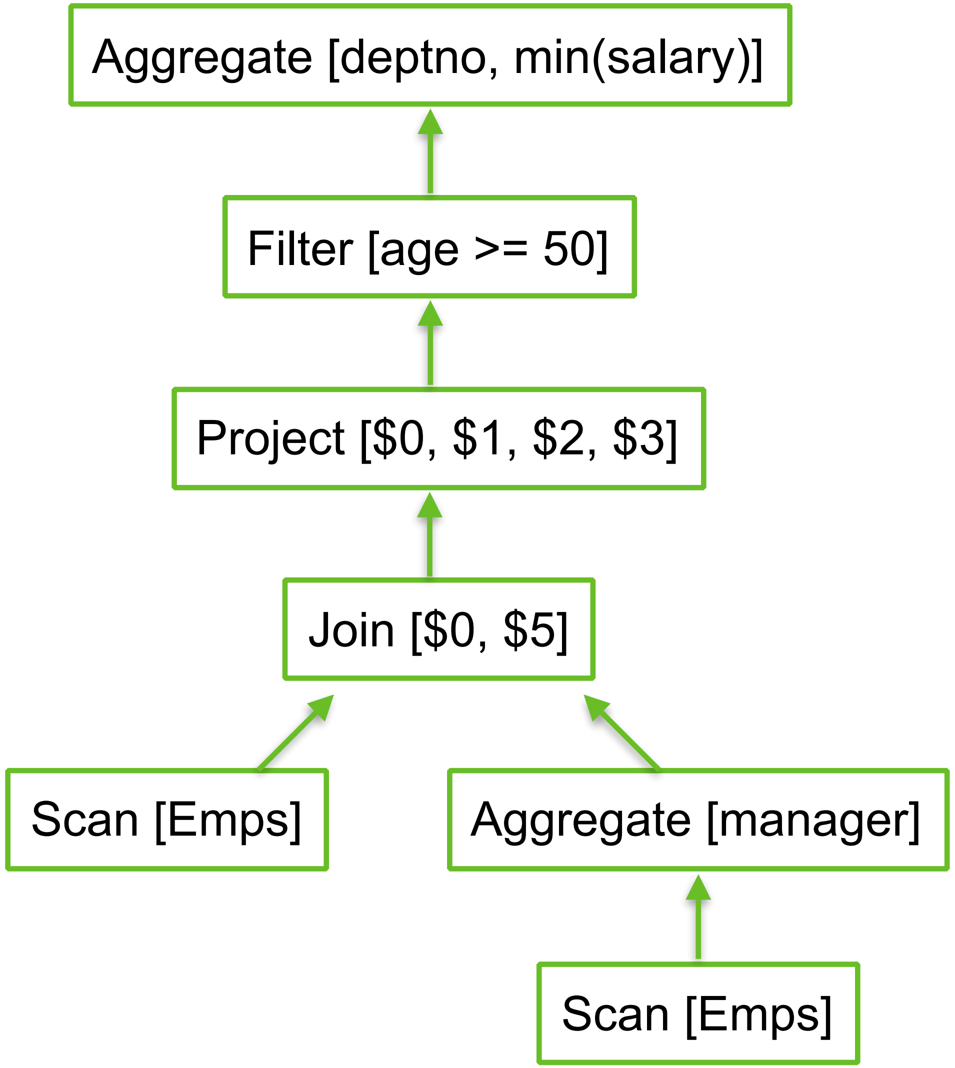

```
CREATE VIEW Managers AS
SELECT *
FROM Emps
WHERE EXISTS (
  SELECT *
  FROM Emps AS underling
  WHERE underling.manager = emp.id)
```



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)
```