Smarter Together

Bringing Relational Algebra, Powered by Apache Calcite, into Looker’s Query Engine

[Julian Hyde | JOIN 2019 | 5-7 November 2019 | San Francisco]

looker

An algebra problem

Ice cream costs $1.50

Topping costs $0.50

How much do 3 ice creams with toppings cost?

An algebra problem

How did you solve it?

Ice cream costs $1.50
Topping costs $0.50
How much do 3 ice creams with toppings cost?

Method 1 (3 * $1.50) + (3 * $0.50) = $4.50 + $1.50 = $6.00
Method 2 3 * ($1.50 + $0.50) = 3 * $2.00 = $6.00

An algebra problem

It’s best to apply toppings first

Ice cream costs $1.50
Topping costs $0.50
How much do 3 ice creams with toppings cost?

Method 1 (3 * $1.50) + (3 * $0.50) = $4.50 + $1.50 = $6.00
Method 2 3 * ($1.50 + $0.50) = 3 * $2.00 = $6.00

Algebraic identity: (a * b) + (a * c) = a * (b + c)

Algebra

Algebra is a mathematical language for combining values using operators.

Algebraic identities tell us how the operators and values interact.

We can prove things by using the identities as **rewrite rules**.

Values numbers

Operators + - * /

Algebraic identities

- a * (b + c) = (a * b) + (a * c)
- a + 0 = a
- a * 0 = 0
- a + b = b + a

Relational algebra

Relational algebra is an algebra where the values are relations (aka tables, multisets of records).

Values relations

Operators join ($\bowtie$) union($\cup$) project (Π) filter (σ)

Algebraic identities

- $A \bowtie (B \cup C) = A \bowtie B \cup A \bowtie C$
- $A \cup \varnothing = A$
- $A \bowtie \varnothing = A$
- $A \cup B = B \cup A$
- $\sigma_P (\sigma_Q (A)) = \sigma_{P \wedge Q}(A)$

Algebraic identities in SQL

$A \bowtie (B \cup C) = A \bowtie B \cup A \bowtie C$

```
SELECT * FROM products
JOIN (SELECT FROM orders_2018
         UNION ALL
          SELECT * FROM orders_2019)
```

=

```
SELECT * FROM products JOIN orders_2018
UNION ALL
SELECT * FROM products JOIN orders_2019
```

$\sigma_P (\sigma_Q (A)) = \sigma_{P \wedge Q} (A)$

```
SELECT *
FROM (SELECT * FROM products
          WHERE color = ‘red’)
WHERE type = ‘bicycle’
```

=

```
SELECT * FROM products
WHERE color = ‘red’
AND type = ‘bicycle’
```

Agenda

Relational algebra and query optimization

How Looker executes a query (and how Calcite can help)

Aggregate awareness & materialized views

Generate LookML model from SQL

Agenda

Relational algebra and query optimization

How Looker executes a query (and how Calcite can help)

Aggregate awareness & materialized views

Generate LookML model from SQL

Apache Calcite

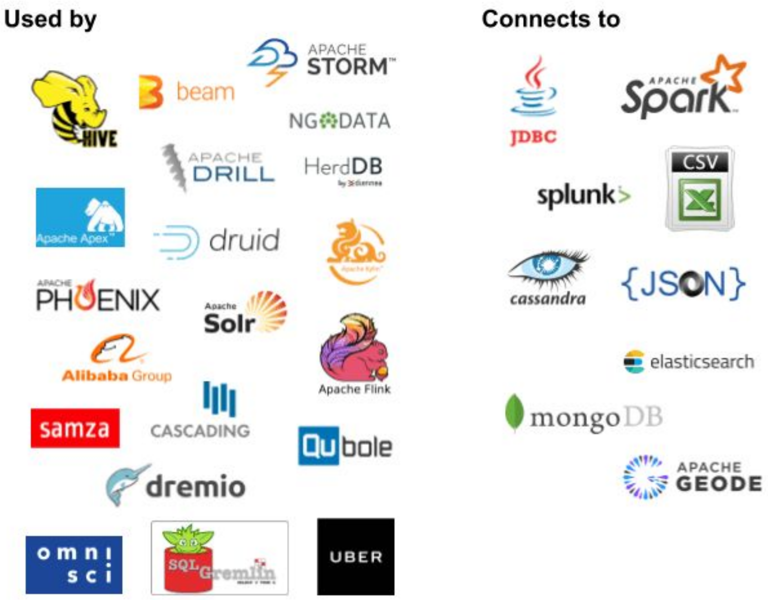
Apache top-level project

Query planning framework used in many projects and products

Also works standalone: embedded federated query engine with SQL / JDBC front end

Apache community development model

https://calcite.apache.org
https://github.com/apache/calcite



Apache Calcite – goals

Make it easier to write a simple DBMS

Advance the state of the art for complex DBMS by pooling resources

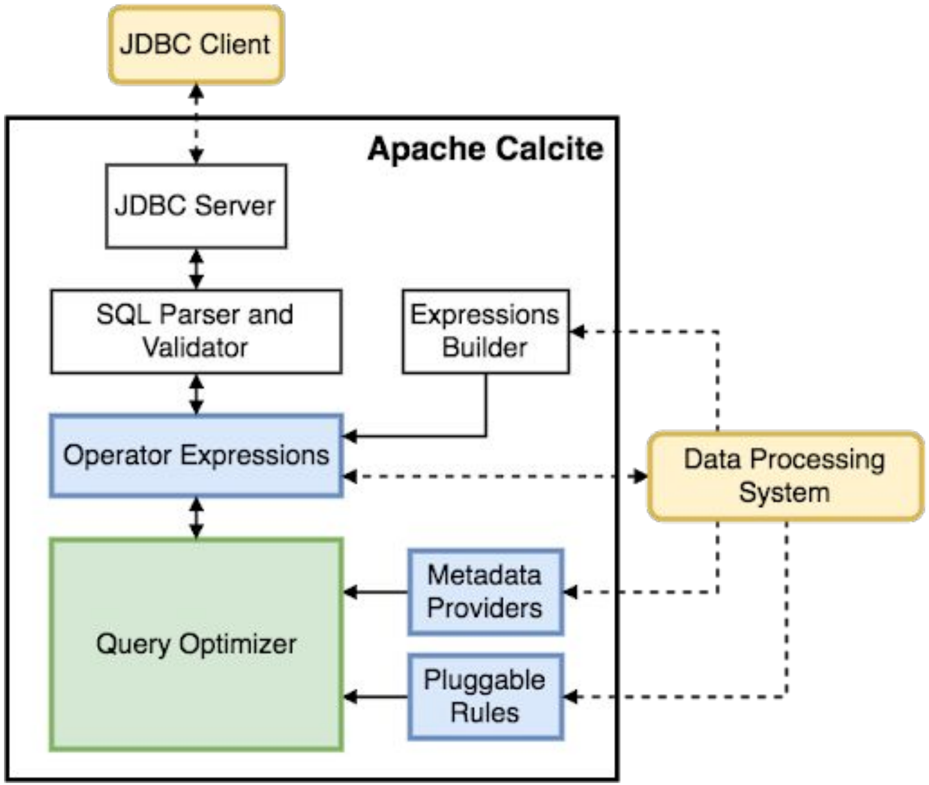
Bring database approaches to new areas (e.g. streaming)

Allow create a DBMS by composing pieces (federation, etc.)

Customize by plugging into framework, evolving framework when necessary

Apache license & governance

Apache Calcite - architecture



Core – Operator expressions (relational algebra) and planner (based on Volcano/Cascades)

External – Data storage, algorithms and catalog

Optional – SQL parser, JDBC & ODBC drivers

Extensible – Planner rewrite rules, statistics, cost model, algebra, UDFs

Relational algebra

Translation to SQL

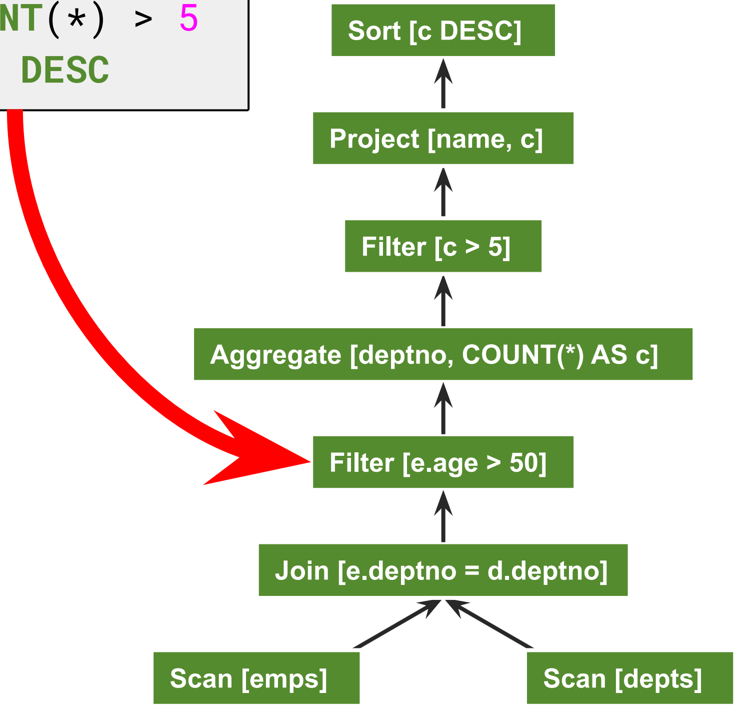
Based on set theory, plus operators: Project, Filter, Aggregate, Union, Join, Sort

Requires: declarative language (SQL), query planner

Original goal: data independence

Enables: query optimization, new algorithms and data structures

```
SELECT d.name, COUNT(*) AS c
FROM emps AS e
JOIN depts AS d USING (deptno)
WHERE e.age > 50
GROUP BY d.deptno
HAVING COUNT(*) > 5
ORDER BY c DESC
```



Sort [c DESC]

Project [name, c]

Filter [c > 5]

Aggregate [deptno, COUNT(*) AS c]

Filter [e.age > 50]

Join [e.deptno = d.deptno]

Scan [emps]

Scan [depts]

Relational algebra

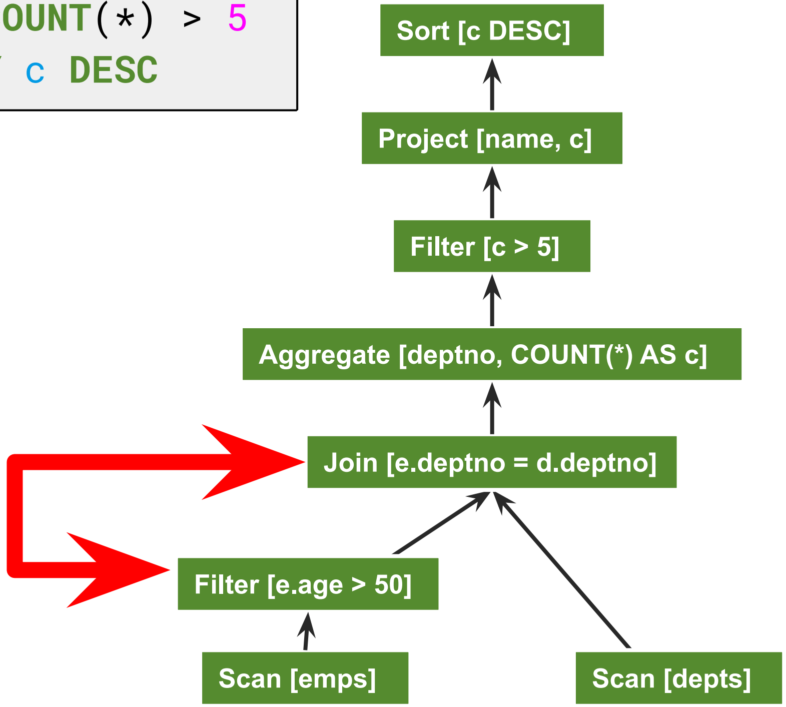
Algebraic rewrite

Optimize by applying rewrite rules that preserve semantics

Hopefully the result is less expensive; but it’s OK if it’s not (planner keeps “before” and “after”)

Planner uses dynamic programming, seeking the lowest total cost

```
SELECT d.name, COUNT(*) AS c
FROM emps AS e
JOIN depts AS d USING (deptno)
WHERE e.age > 50
GROUP BY d.deptno
HAVING COUNT(*) > 5
ORDER BY c DESC
```



Calcite framework

Relational algebra

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

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

RelBuilder

SQL parser

SqlNode
SqlParser
SqlValidator

Metadata

Schema
Table
Function
- TableFunction
- TableMacro

Lattice

JDBC driver

Remote driver
Local driver

Transformation rules

RelOptRule
- FilterMergeRule
- AggregateUnionTransposeRule
- 100+ more

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

Cost, statistics

RelOptCost
RelOptCostFactory
RelMetadataProvider
- RelMdColumnUniqueness
- RelMdDistinctRowCount
- RelMdSelectivity

Agenda

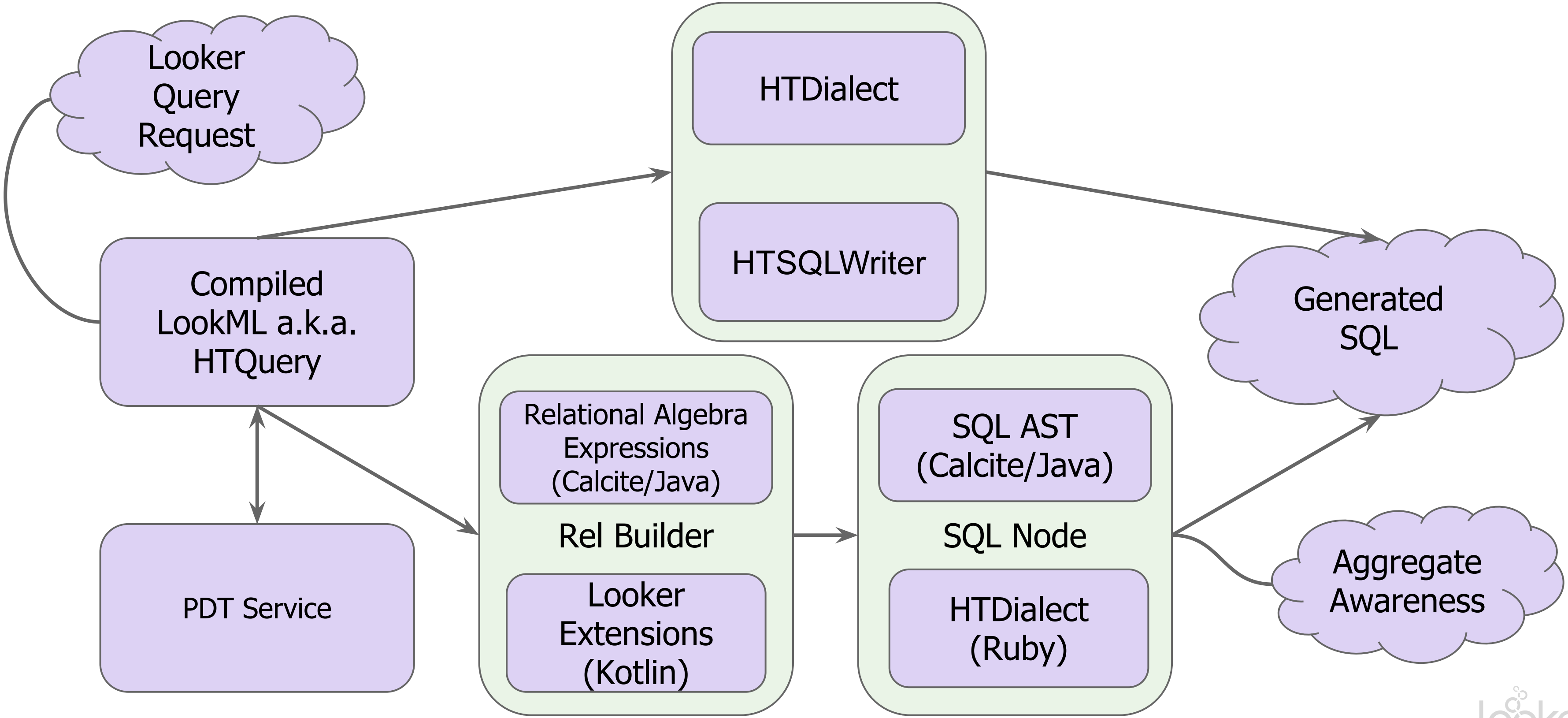
Relational algebra and query optimization

How Looker executes a query (and how Calcite can help)

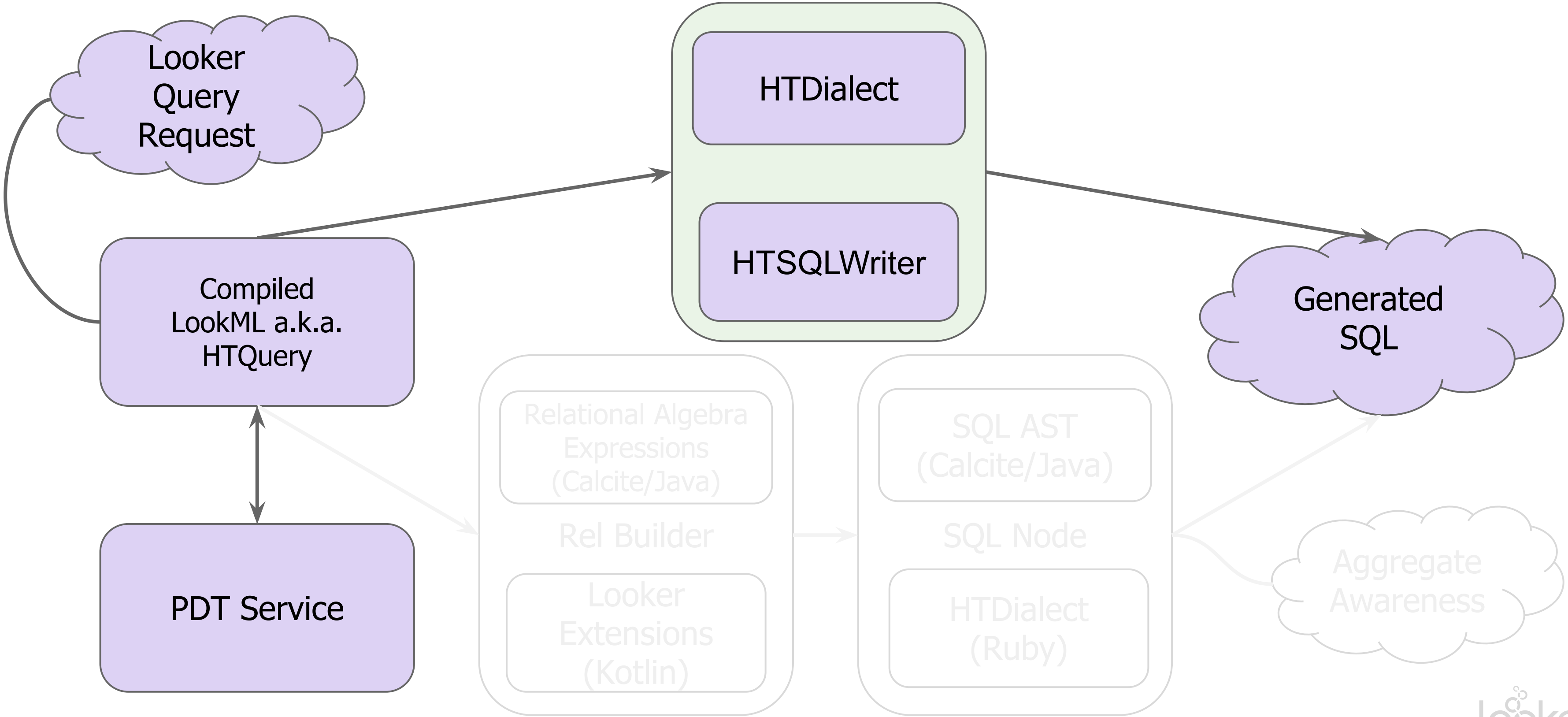
Aggregate awareness & materialized views

Generate LookML model from SQL

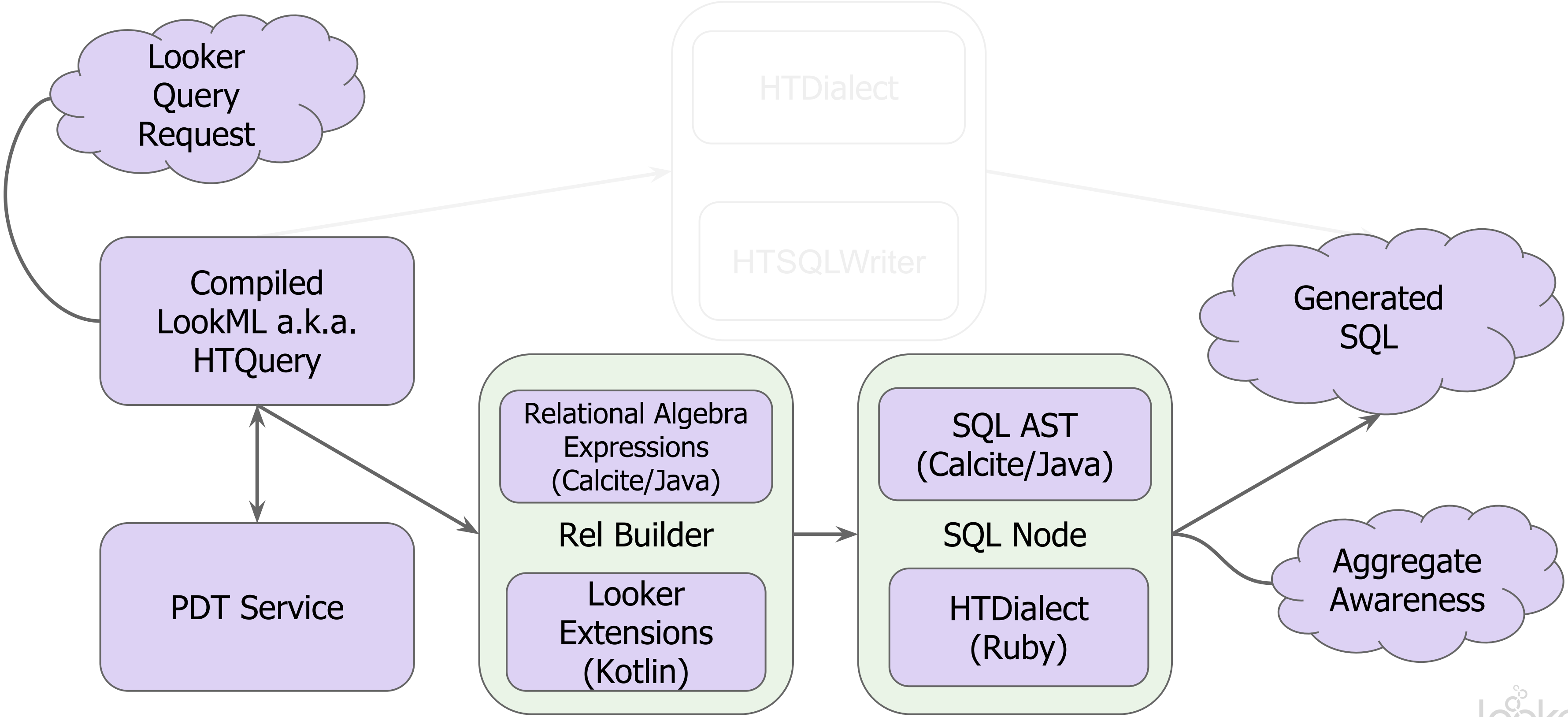
Looker SQL Generation Overview



Looker SQL Generation (old)



Looker SQL Generation (new)



Agenda

Relational algebra and query optimization

How Looker executes a query (and how Calcite can help)

Aggregate awareness & materialized views

Generate LookML model from SQL

Aggregates to the rescue

Business question:

Which state had the most revenue in 2018?

customers (12 million rows)

customer_id	state
12345	CA
23456	TX

orders (100 million rows)

date	product_id	customer_id	revenue
2018-01-05	100	12345	$17
2018-01-07	105	23456	$120

Aggregates to the rescue

Business question:

Which state had the most revenue in 2018?

Raw data – 120 million rows

Summary data – 2,000 rows

customers (12 million rows)

customer_id	state
12345	CA
23456	TX

orders (100 million rows)

date	product_id	customer_id	revenue
2018-01-05	100	12345	$17
2018-01-07	105	23456	$120

orders_by_state_and_quarter (2,000 rows)

quarter	state	sum_revenue	sales_count
2018-Q1	CA	$15,567	46
2018-Q1	TX	$23,006	88

Aggregates in LookML

LookML - orders

```
view: orders {
  dimension_group: created_at {
    type: time
    sql: ${TABLE}.created_at ;;
  }
  dimension: product_id {}
  dimension: customer_id {}
  measure: revenue {}
}
view: customers {
  dimension: customer_id {}
  dimension: state {}
}
explore: orders {
  join: customers {}
}
```

LookML - orders_by_state_quarter

```
view: orders_by_state_quarter {
  derived_table: {
    explore_source: orders {
      column: created_at_quarter {
        field: orders.created_at_quarter
      }
      column: state {
        field: customers.state
      }
      column: sum_revenue {
        sql: sum(${orders.revenue});;
      }
}
explore: orders_by_state_quarter {}
```

NOTE: This feature is not production, and syntax may change

Views, materialized views, aggregate awareness

A **view** is a virtual table that expands to a relational expression

- No data stored on disk (except its definition)
- Purpose: abstraction

A **materialized view** is a table whose contents are defined by a relational expression

- Purpose: improve query performance

A **aggregate table** or **summary table** is a materialized view that uses GROUP BY (and perhaps JOIN) and has many fewer rows than its base table

A system with **aggregate awareness** automatically builds aggregate tables and uses them to make queries faster

View

```
SELECT deptno, MIN(salary)
FROM managers
WHERE age > 50
GROUP BY deptno
```

Query that uses the `managers` view

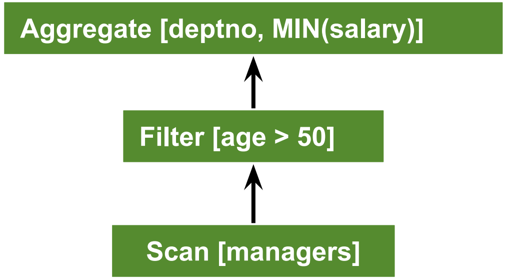
```
CREATE VIEW managers AS
SELECT *
FROM emps AS e
WHERE EXISTS (
  SELECT *
  FROM emps AS underling
  WHERE underling.manager = e.id)
```

Declaration of the `managers` view

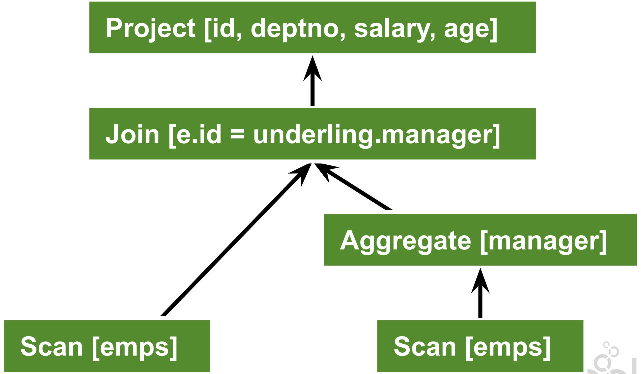
View

With relational algebra

```
SELECT deptno, MIN(salary)
FROM managers
WHERE age > 50
GROUP BY deptno
```



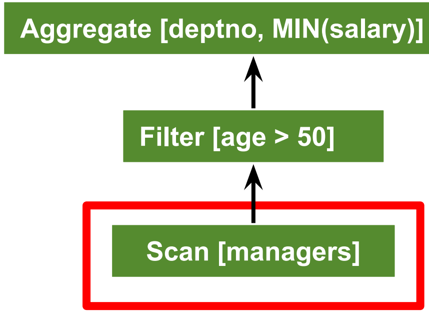
```
CREATE VIEW managers AS
SELECT *
FROM emps AS e
WHERE EXISTS (
  SELECT *
  FROM emps AS underling
  WHERE underling.manager = e.id)
```


Project [id, deptno, salary, age]
Join [e.id = underling.manager]
Aggregate [manager]
Scan [emps]
Scan [emps]

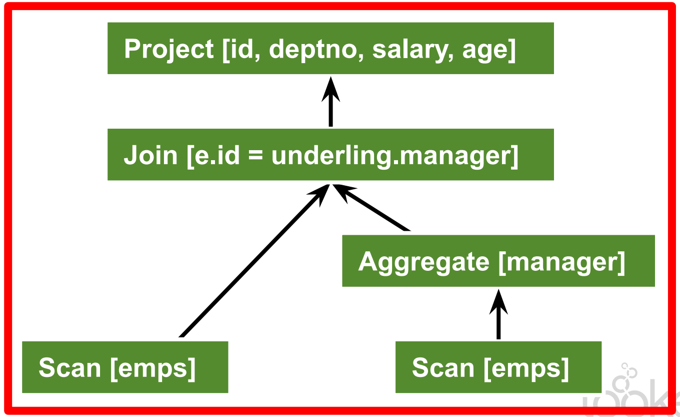
View

Showing the view and the relational expression that will replace it

```
SELECT deptno, MIN(salary)
FROM managers
WHERE age > 50
GROUP BY deptno
```



```
CREATE VIEW managers AS
SELECT *
FROM emps AS e
WHERE EXISTS (
  SELECT *
  FROM emps AS underling
  WHERE underling.manager = e.id)
```

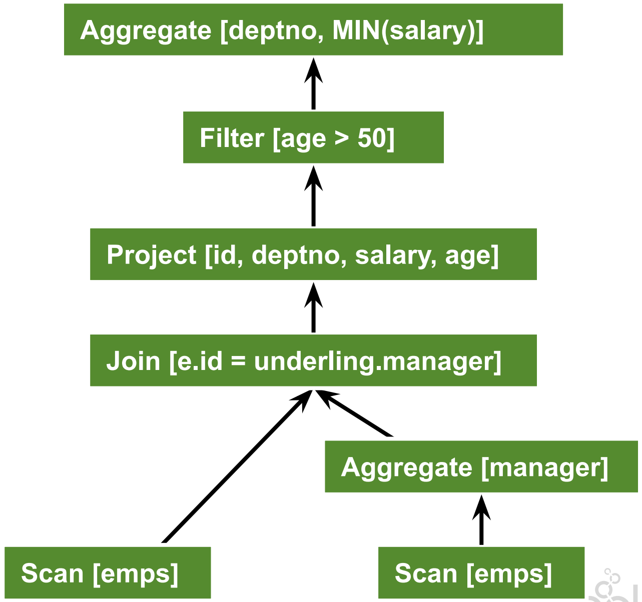

Project [id, deptno, salary, age]
Join [e.id = underling.manager]
Aggregate [manager]
Scan [emps]
Scan [emps]

View

After view has been expanded

```
SELECT deptno, MIN(salary)
FROM managers
WHERE age > 50
GROUP BY deptno
```

```
CREATE VIEW managers AS
SELECT *
FROM emps AS e
WHERE EXISTS (
  SELECT *
  FROM emps AS underling
  WHERE underling.manager = e.id)
```



Aggregate [deptno, MIN(salary)]

Filter [age > 50]

Project [id, deptno, salary, age]

Join [e.id = underling.manager]

Aggregate [manager]

Scan [emps]

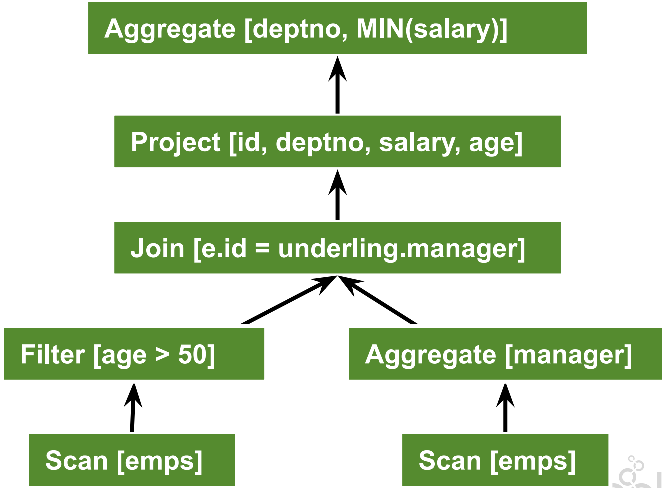
Scan [emps]

View

After pushing down “age > 50” filter

```
SELECT deptno, MIN(salary)
FROM managers
WHERE age > 50
GROUP BY deptno
```

```
CREATE VIEW managers AS
SELECT *
FROM emps AS e
WHERE EXISTS (
  SELECT *
  FROM emps AS underling
  WHERE underling.manager = e.id)
```


Aggregate [deptno, MIN(salary)]
Project [id, deptno, salary, age]
Join [e.id = underling.manager]
Filter [age > 50]
Aggregate [manager]
Scan [emps]
Scan [emps]

Materialized view

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

Declaration of the emps_by_deptno_gender materialized view

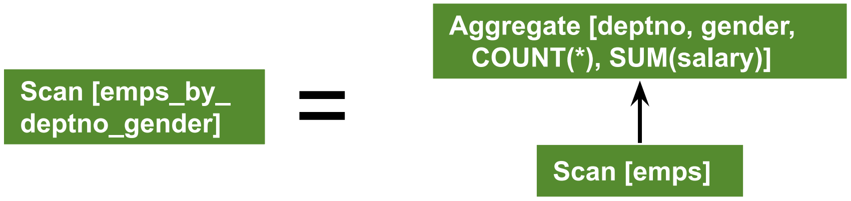
```
SELECT COUNT(*) AS c
FROM emps
WHERE deptno = 10
AND gender = ‘M’
```

Query that uses the emps table but could potentially use the emps_by_deptno_gender materialized view

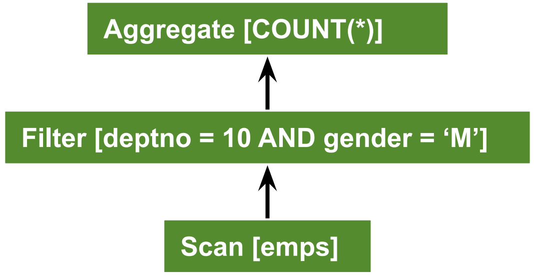
Materialized view

With relational algebra

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



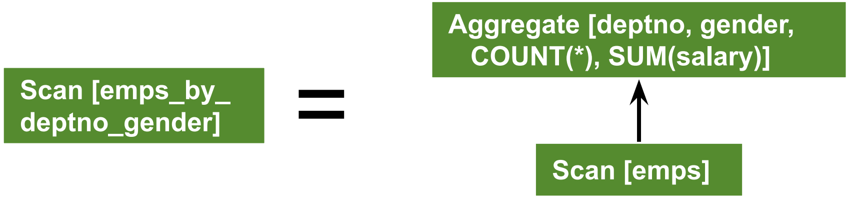
```
SELECT COUNT(*) AS c
FROM emps
WHERE deptno = 10
AND gender = 'M'
```



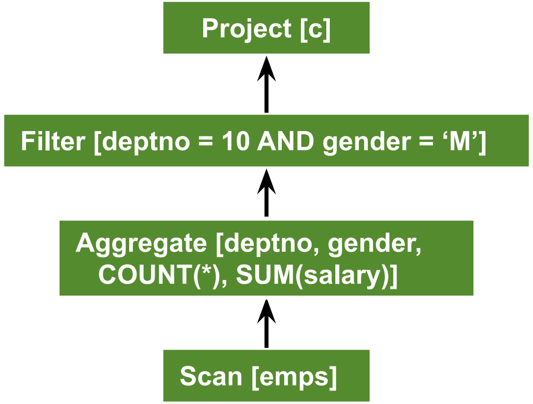
Materialized view

Rewrite query to match

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


Scan [emps_by_ deptno_gender]
=
Aggregate [deptno, gender, COUNT(*), SUM(salary)]
Scan [emps]

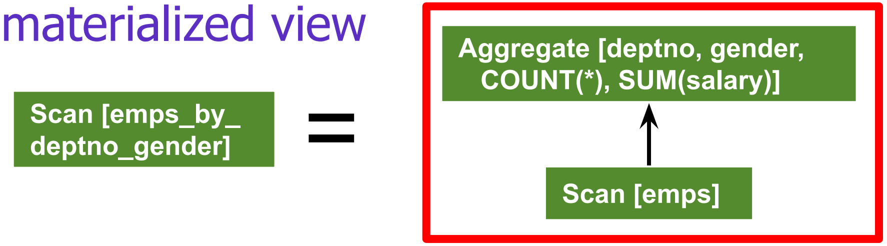
```
SELECT COUNT(*) AS c
FROM emps
WHERE deptno = 10
AND gender = 'M'
```


Project [c]
Filter [deptno = 10 AND gender = 'M']
Aggregate [deptno, gender, COUNT(*), SUM(salary)]
Scan [emps]

Materialized view

Fragment of query is now identical to materialized view

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



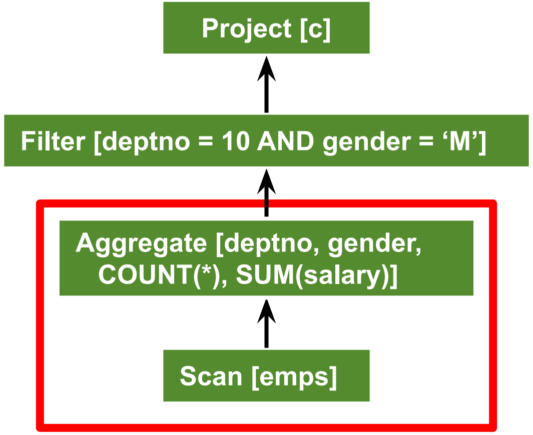
Scan [emps_by_ deptno_gender]

=

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

Scan [emps]

```
SELECT COUNT(*) AS c
FROM emps
WHERE deptno = 10
AND gender = ‘M’
```



Project [c]

Filter [deptno = 10 AND gender = ‘M’]

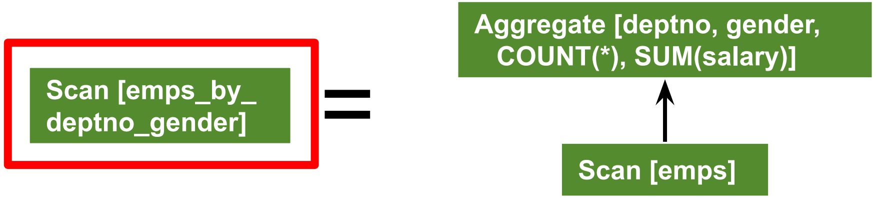
Aggregate [deptno, gender, COUNT(*), SUM(salary)]

Scan [emps]

Materialized view

Substitute table scan

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



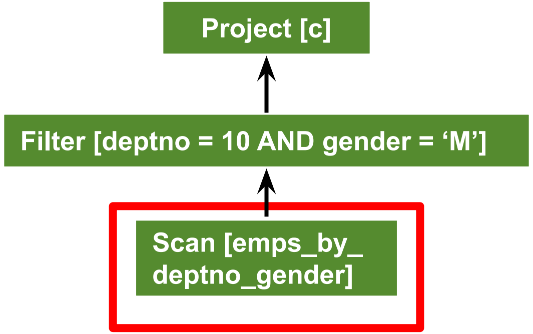
Scan [emps_by_ deptno_gender]

=

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

Scan [emps]

```
SELECT COUNT(*) AS c
FROM emps
WHERE deptno = 10
AND gender = ‘M’
```



Project [c]

Filter [deptno = 10 AND gender = ‘M’]

Scan [emps_by_ deptno_gender]

Aggregates in LookML

Aggregate table is used even by queries on the `orders` explore

LookML - orders

```
view: orders {
  dimension_group: created_at {
    type: time
    sql: ${TABLE}.created_at ;;
  }
  dimension: product_id {}
  dimension: customer_id {}
  measure: revenue {}
}
view: customers {
  dimension: customer_id {}
  dimension: state {}
}
explore: orders {
  join: customers {}
}
```

LookML - orders_by_state_quarter

```
view: orders_by_state_quarter {
  derived_table: {
    explore_source: orders {
      column: created_at_quarter {
        field: orders.created_at_quarter
      }
      column: state {
        field: customers.state
      }
      column: sum_revenue {
        sql: sum(${orders.revenue});;
      }
}
explore: orders_by_state_quarter {}
```

Advanced features & crazy ideas

- Join elimination
    - If my query uses one table but my aggregate table uses two, is it safe to rewrite?
- Time zones
    - If the aggregate table is rolled up by day, and what if my day ends at a different time than yours?
- Recommend/build aggregate tables based on...
    - Queries over the last month
    - Queries over the last ten minutes
    - Queries needed to satisfy a given dashboard

Agenda

Relational algebra and query optimization

How Looker executes a query (and how Calcite can help)

Aggregate awareness & materialized views

Generate LookML model from SQL

A model out of thin air

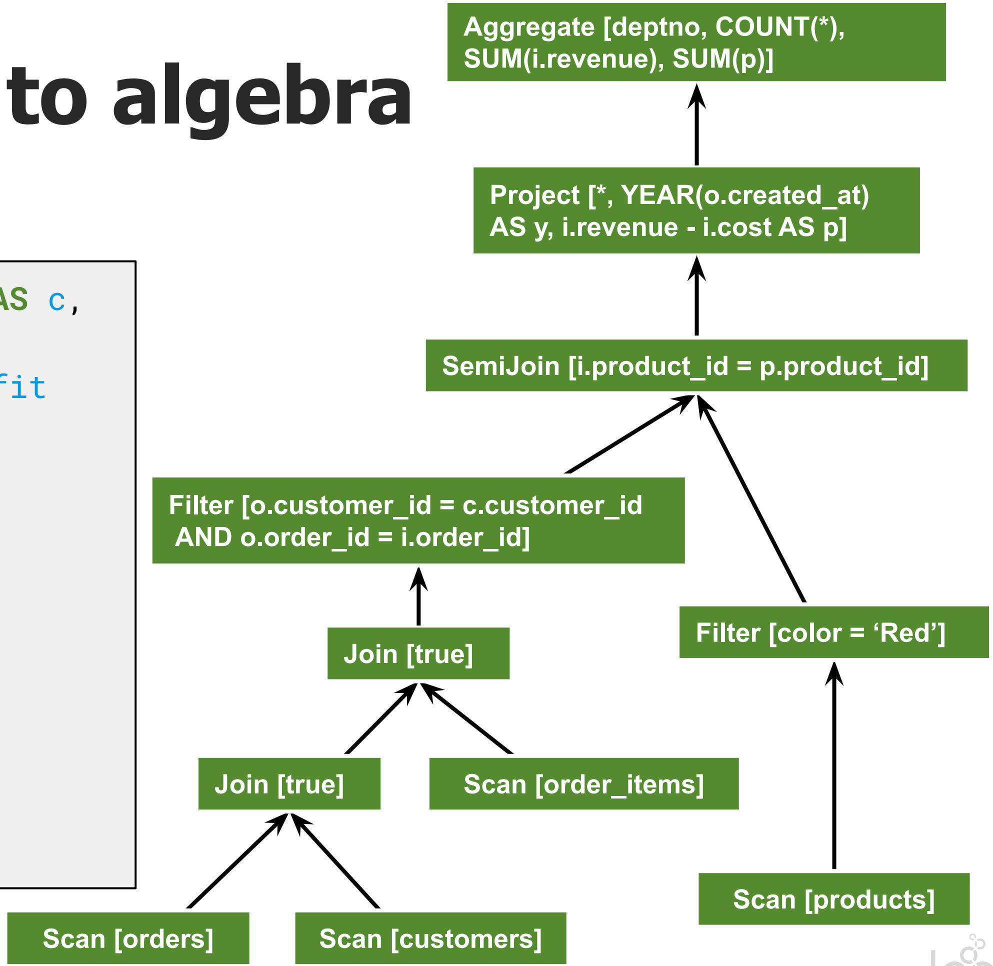
Companies that are just starting to use Looker often have a database and log of queries run against that database.

Parsing the SQL queries in the log, we can recover the dimensions, measures, relationships, and put them into a LookML model.

We call this process “model induction”.

Convert a query to algebra

```
SELECT YEAR(o.created_at), COUNT(*) AS c,
  SUM(i.revenue) AS sum_revenue,
  SUM(i.revenue - i.cost) AS sum_profit
FROM orders AS o,
  customers AS c,
  order_items AS i
WHERE o.customer_id = c.customer_id
AND o.order_id = i.order_id
AND i.product_id IN (
    SELECT product_id
    FROM products AS p
    WHERE color = ‘Red’)
GROUP BY YEAR(o.created_at)
```



Aggregate [deptno, COUNT(*), SUM(i.revenue), SUM(p)]

Project [*, YEAR(o.created_at) AS y, i.revenue - i.cost AS p]

SemiJoin [i.product_id = p.product_id]

Filter [o.customer_id = c.customer_id AND o.order_id = i.order_id]

Filter [color = ‘Red’]

Join [true]

Join [true]

Scan [order_items]

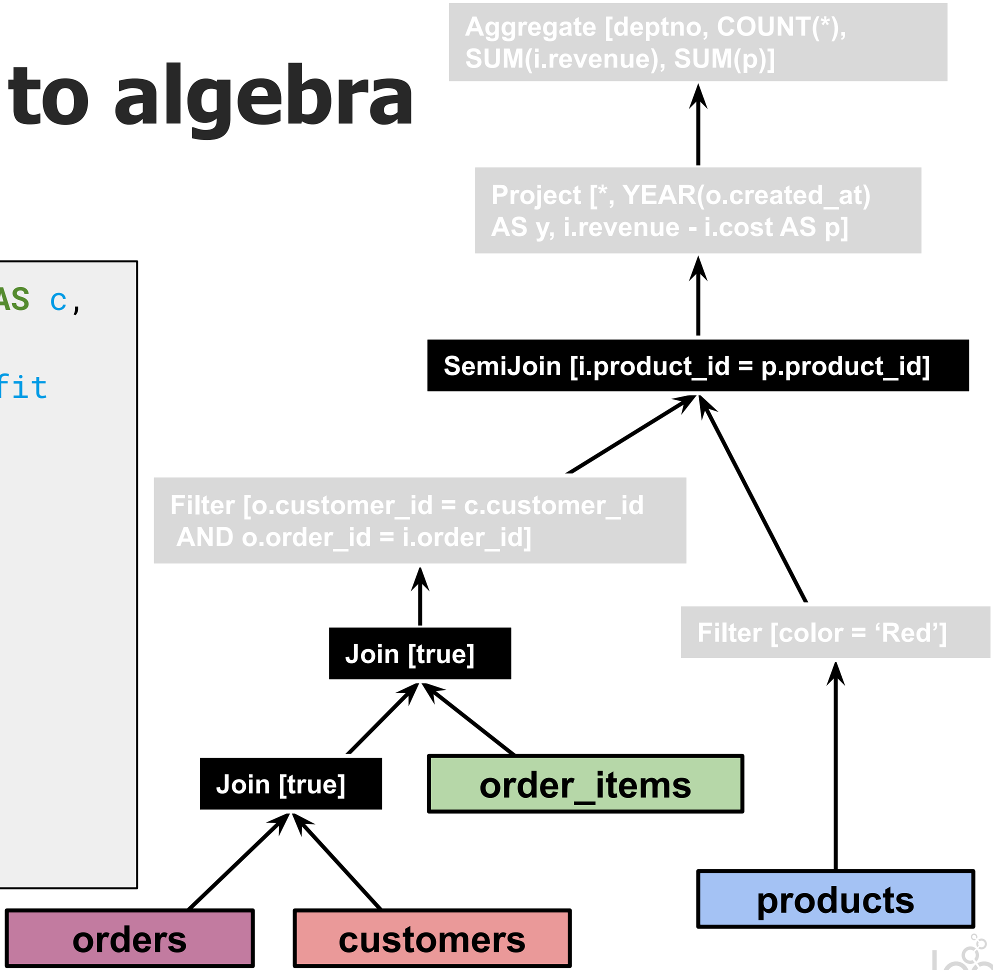
Scan [orders]

Scan [customers]

Scan [products]

Convert a query to algebra

```
SELECT YEAR(o.created_at), COUNT(*) AS c,
  SUM(i.revenue) AS sum_revenue,
  SUM(i.revenue - i.cost) AS sum_profit
FROM orders AS o,
  customers AS c,
  order_items AS i
WHERE o.customer_id = c.customer_id
AND o.order_id = i.order_id
AND i.product_id IN (
    SELECT product_id
    FROM products AS p
    WHERE color = ‘Red’)
GROUP BY YEAR(o.created_at)
```



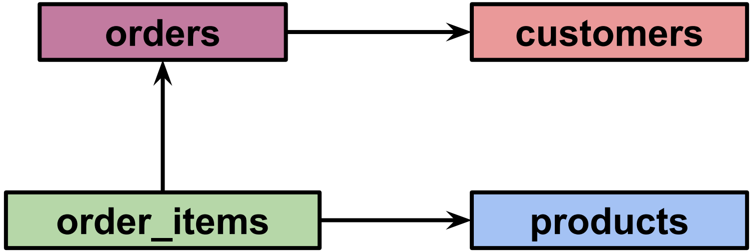
Aggregate [deptno, COUNT(*), SUM(i.revenue), SUM(p)]
Project [*, YEAR(o.created_at) AS y, i.revenue - i.cost AS p]
SemiJoin [i.product_id = p.product_id]
Filter [o.customer_id = c.customer_id AND o.order_id = i.order_id]
Filter [color = ‘Red’]
Join [true]
Join [true]
order_items
products
orders
customers

Convert query to join graph, measures

```sql
SELECT YEAR(o.created_at), COUNT(*) AS c,
  SUM(i.revenue) AS sum_revenue,
  SUM(i.revenue - i.cost) AS sum_profit
FROM orders AS o,
  customers AS c,
  order_items AS i
WHERE o.customer_id = c.customer_id
AND o.order_id = i.order_id
AND i.product_id IN (
    SELECT product_id
    FROM products AS p
    WHERE color = ‘Red’)
GROUP BY YEAR(o.created_at)
```

measures

COUNT(*)
SUM(i.revenue)
SUM(i.revenue - i.cost)



<show LookML generated from previous query>

Generate LookML model from SQL queries

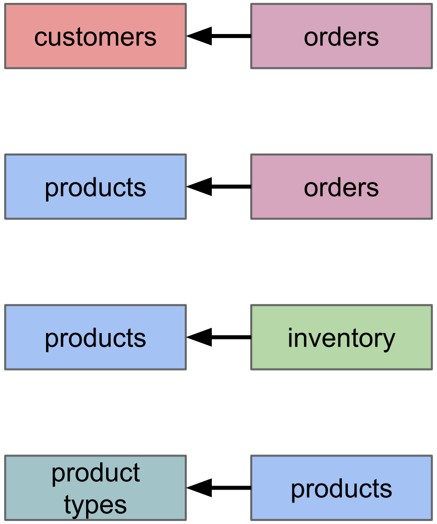
Query file

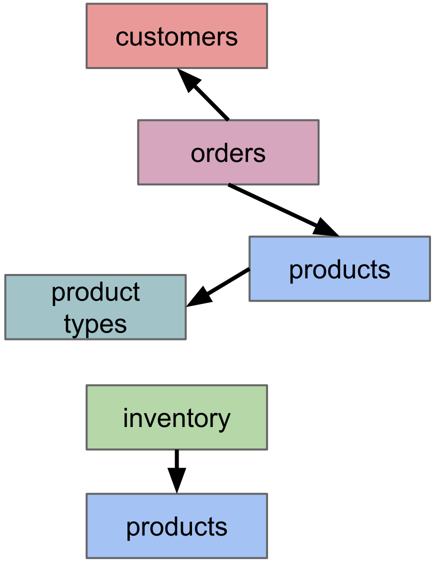
```
SELECT d1, d2, m1, m2
FROM customers
JOIN orders

SELECT d1, d3, m2, m3
FROM products
JOIN orders

SELECT d2, m4
FROM products
JOIN inventory

SELECT m5
FROM products
JOIN product_types
```





LookML

```
view: customers {}
view: orders {}
view: products {}
view: product_types {}
explore: orders {
  join: customers {}
  join: products {}
  join: product_types {}
}

view: products {}
view: inventory {}
explore: inventory {
  join: products {}
}
```

1. Parse & validate SQL query log; convert to relational algebra

2. Construct join graphs; deduce PK-FK relationships by running cardinality queries

3. Cluster join graphs into stars, each with a central “fact table”

4. Generate LookML; one explore per star

< show screenshots or demo of inducer reading query set and generating model >

< show screenshots or demo of inducer driven from a single query in SQL Runner >

< a few more possible futures: querying explores via SQL; a JDBC driver as an alternative interface; an internal query language (perhaps SQL, perhaps something else) that allows defining views, explores and queries on top of existing explores >

Thank you! Any questions?

@julianhyde
@ApacheCalcite

https://calcite.apache.org
https://github.com/apache/calcite