

# Drill / SQL / Optiq

Julian Hyde

Apache Drill User Group  
2013-03-13

O'REILLY®

# Strata

CONFERENCE  
Making Data Work

Feb. 26 – 28, 2013  
SANTA CLARA, CA



SQL

# SQL: Pros & cons

Fact:

- SQL is older than Macaulay C

Less interesting but more relevant

- Can be written by (lots of) humans
- Can be written by machines
- Requires query optimization
- Allows query optimization
- Based on “flat” relations and basic relational operations



# Quick intro to Optiq

# Introducing Optiq

Framework

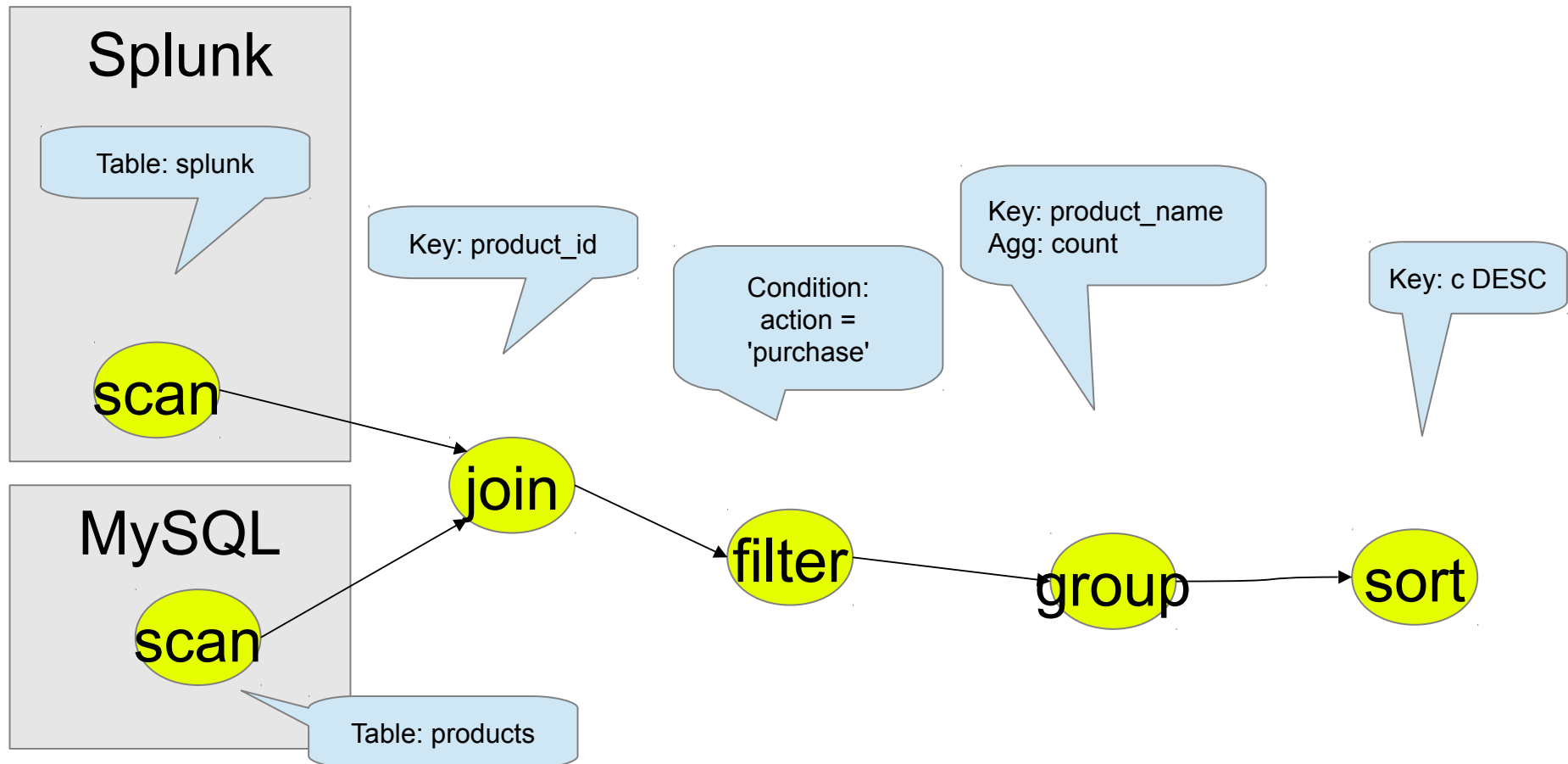
Derived from LucidDB

Minimal query mediator:

- No storage
- No runtime
- No metadata
- Query planning engine
- Core operators & rewrite rules
- Optional SQL parser/validator

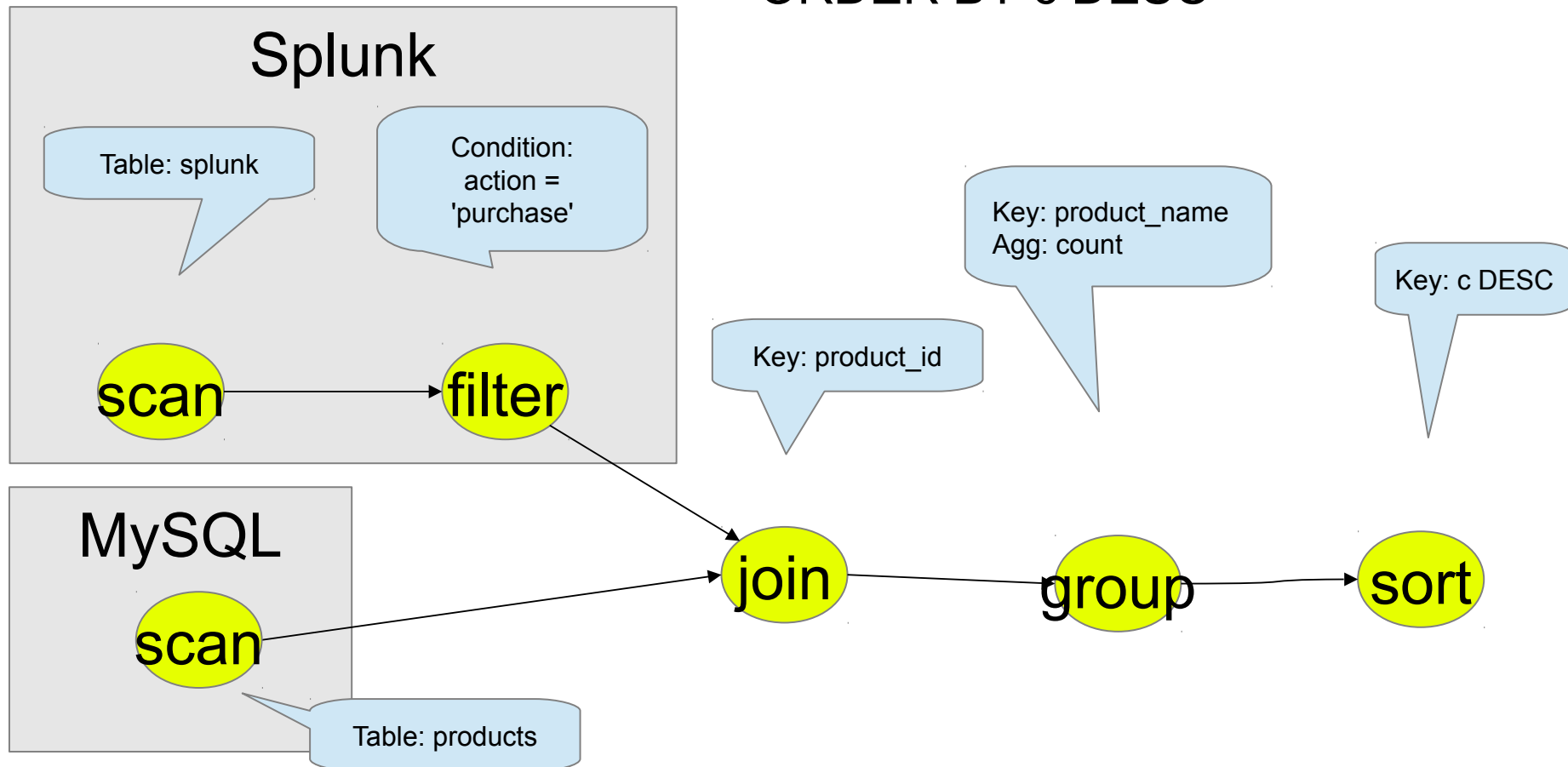
# Expression tree

```
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
```



# Metadata SPI

- interface Table
  - RelDataType getRowType()
- interface TableFunction
  - List<Parameter> getParameters()
  - Table apply(List arguments)
  - e.g. ViewTableFunction
- interface Schema
  - Map<String, List<TableFunction>>  
getTableFunctions()

# Operators and rules

- Rule: interface RelOptRule
- Operator: interface RelNode
- Core operators: TableAccess, Project, Filter, Join, Aggregate, Order, Union, Intersect, Minus, Values
- Some rules: MergeFilterRule, PushAggregateThroughUnionRule, RemoveCorrelationForScalarProjectRule + 100 more

# Planning algorithm

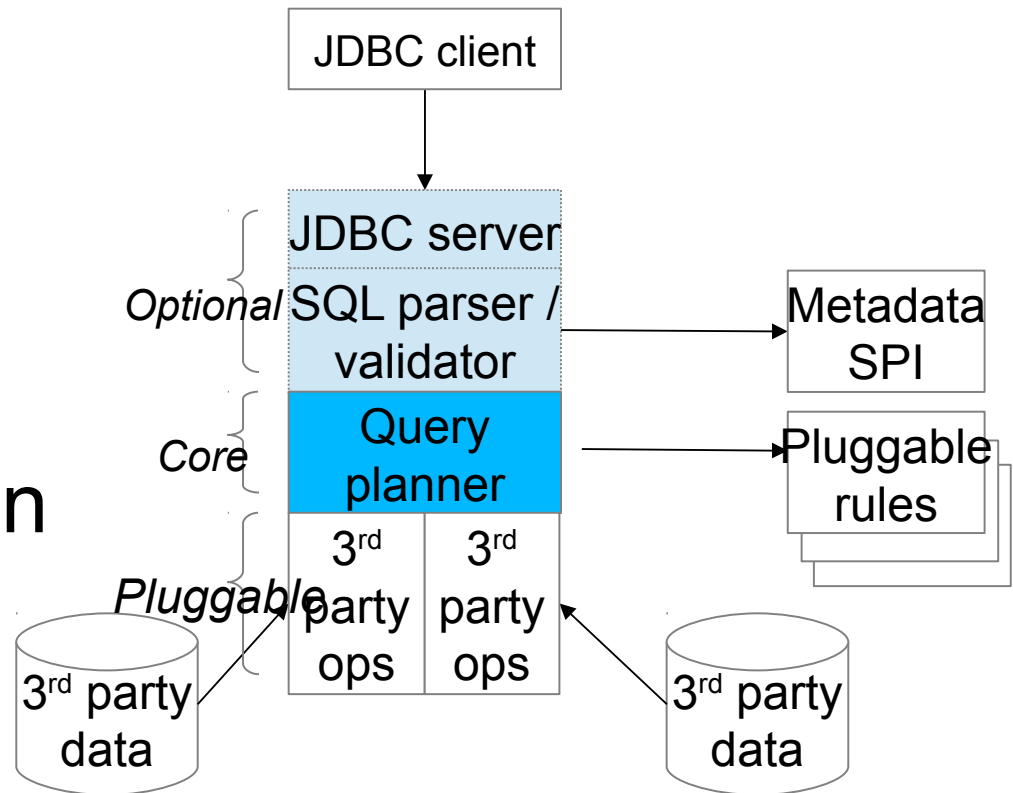
- Start with a logical plan and a set of rewrite rules
- Form a list of rewrite rules that are applicable (based on pattern-matching)
- Fire the rule that is likely to do the most good
- Rule generates an expression that is equivalent (and hopefully cheaper)
- Queue up new rule matches
- Repeat until cheap enough

# Concepts

- Cost
- Equivalence sets
- Calling convention
- Logical vs Physical
- Traits
- Implementation

# Outside the kernel

- SQL parser/validator
- JDBC driver
- SQL function library (validation + code-generation)
- Lingual (Cascading adapter)
- Splunk adapter
- Drill adapter



# Optiq roadmap

- Building blocks for analytic DB:
  - In-memory tables in a distributed cache
  - Materialized views
  - Partitioned tables
- Faster planning
- Easier rule development
- ODBC driver
- Adapters for XXX, YYY

# Applying Optiq to Drill

- 1. Enhance SQL**
2. Query translation

# Drill vs Traditional SQL

- SQL:
  - Flat data
  - Schema up front
- Drill:
  - Nested data (list & map)
  - No schema
- We'd like to write:
  - `SELECT name, toppings[2] FROM donuts  
WHERE ppu > 0.6`
- Solution: ARRAY, MAP, ANY types

# ARRAY & MAP SQL types

- ARRAY is like `java.util.List`
- MAP is like `java.util.LinkedHashMap`

Examples:

- `VALUES ARRAY ['a', 'b', 'c']`
- `VALUES MAP ['Washington', 1, 'Obama', 44]`
- `SELECT name, address[1], address[2], state  
FROM Employee`
- `SELECT * FROM Donuts WHERE  
CAST(donuts['ppu'] AS DOUBLE) > 0.6`

# ANY SQL type

- ANY means “type to be determined at runtime”
- Validator narrows down possible type based on operators used
- Similar to converting Java's type system into JavaScript's. (Not easy.)

# Sugaring the donut

Query:

- `SELECT c['ppu'], c['toppings'][1] FROM Donuts`

Additional syntactic sugar:

- `c.x` means `c [ ' x ' ]`

So:

- `CREATE TABLE Donuts(c ANY)`
- `SELECT c.ppu, c.toppings[1] FROM Donuts`

Better:

- `CREATE TABLE Donuts( MAP`

# UNNEST

Employees nested inside departments:

- `CREATE TYPE employee (empno INT, name VARCHAR(30));`
- `CREATE TABLE dept (deptno INT, name VARCHAR(30), employees EMPLOYEE ARRAY);`

Unnest:

- `SELECT d.deptno, d.name, e.empno, e.name  
FROM department AS d  
CROSS JOIN UNNEST(d.employees) AS e`

SQL standard provides other operations on collections:

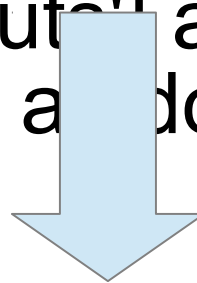
# Applying Optiq to Drill

1. Enhance SQL
- 2. Query translation**

# Query translation

SQL:

- `select d['name'] as name, d['xx'] as xx  
from (  
 select _MAP['donuts'] as d from donuts)  
where cast(d['ppu'] as double) > 0.6`



Drill:

- `{ head: { ... },  
 storage: { ... },  
 query: [ {  
 op: "sequence", do: [  
 { op: "scan", ... selection: { path:  
 "/donuts.json" ... } } ]  
 } ]`

# Planner log

Original rel:

AbstractConverter(subset=[rel#14:Subset#3.ARRAY], convention=[ARRAY])

ProjectRel(subset=[rel#10:Subset#3.NONE], NAME=[ITEM(\$0, 'name')], XX=[ITEM(\$0, 'xx')])

FilterRel(subset=[rel#8:Subset#2.NONE], condition=[>(CAST(ITEM(\$0, 'ppu')):DOUBLE NOT NULL, 0.6)])

ProjectRel(subset=[rel#6:Subset#1.NONE], D=[ITEM(\$0, 'donuts')])

DrillScan(subset=[rel#4:Subset#0.DRILL],

# Next

- Translate join, aggregate, sort, set ops
- Operator overloading with ANY
- Mondrian on Drill

# Thank you!

<https://github.com/julianhyde/share/tree/master/slides>

<https://github.com/julianhyde/incubator-drill>

<https://github.com/julianhyde/optiq>

<http://incubator.apache.org/drill>

@julianhyde