Deep Dive Into Catalyst: Apache Spark 2.0's Optimizer

Yin Huai

Spark Summit 2016

databricks™

Write Programs Using RDD API

```
SELECT count(*)
FROM (
  SELECT t1.id
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```

databricks

Solution 1

t1 join t2

t1.id = t2.id

```
val count = t1.cartesian(t2).filter {
  case (id1FromT1, id2FromT2) => id1FromT1 == id2FromT2
}.filter {
  case (id1FromT1, id2FromT2) => id1FromT2 > 50 * 1000
}.map {
  case (id1FromT1, id2FromT2) => id1FromT1
}.count
println("Count: " + count)
```

t2.id > 50 * 1000

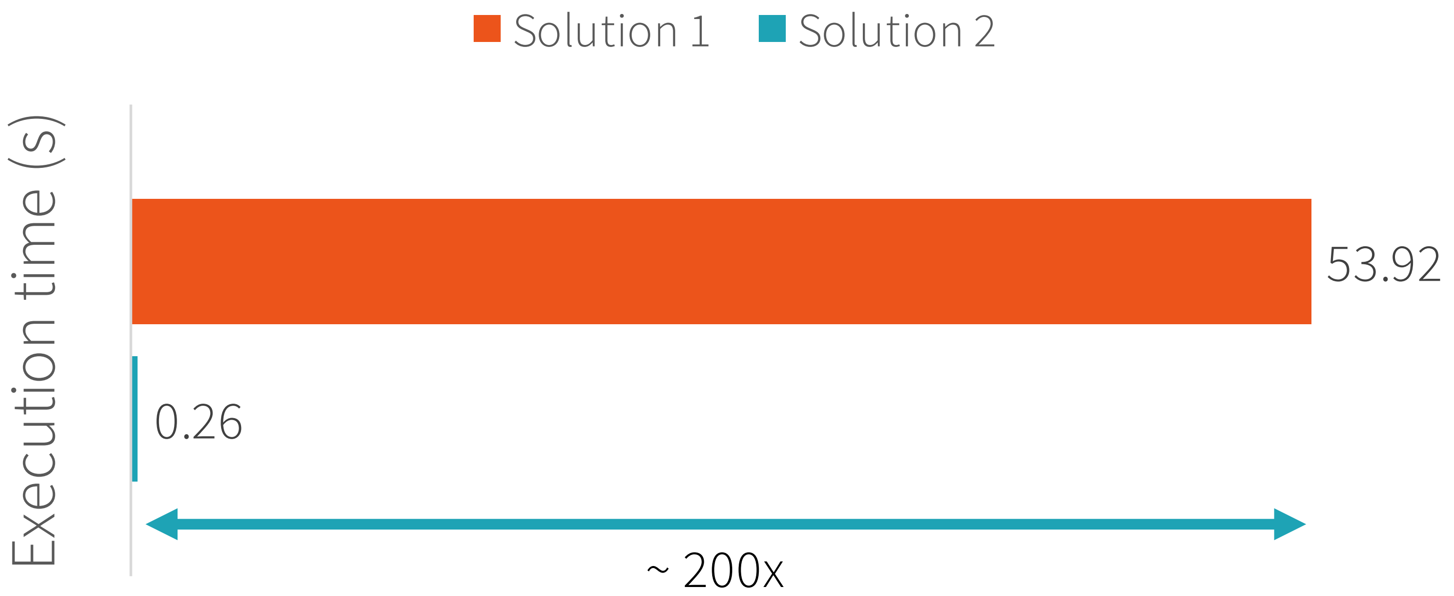
Solution 2

t2.id > 50 * 1000

```
val filteredT2 =
  t2.filter(id1FromT2 => id1FromT2 > 50 * 1000)
val preparedT1 =
  t1.map(id1FromT1 => (id1FromT1, id1FromT1))
val preparedT2 =
  filteredT2.map(id1FromT2 => (id1FromT2, id1FromT2))
val count = preparedT1.join(preparedT2).map {
  case (id1FromT1, _) => id1FromT1
}.count
println("Count: " + count)
```

t1 join t2
WHERE t1.id = t2.id

Solution 1 vs. Solution 2



Solution 1

t1 join t2

t1.id = t2.id

```
val count = t1.cartesian(t2).filter {
  case (id1FromT1, id2FromT2) => id1FromT1 == id2FromT2
}.filter {
  case (id1FromT1, id2FromT2) => id1FromT2 > 50 * 1000
}.map {
  case (id1FromT1, id2FromT2) => id1FromT1
}.count
println("Count: " + count)
```

t2.id > 50 * 1000

Solution 2

t2.id > 50 * 1000

```
val filteredT2 =
  t2.filter(id1FromT2 => id1FromT2 > 50 * 1000)
val preparedT1 =
  t1.map(id1FromT1 => (id1FromT1, id1FromT1))
val preparedT2 =
  filteredT2.map(id1FromT2 => (id1FromT2, id1FromT2))
val count = preparedT1.join(preparedT2).map {
  case (id1FromT1, _) => id1FromT1
}.count
println("Count: " + count)
```

t1 join t2
WHERE t1.id = t2.id

Write Programs Using RDD API

- Users' functions are black boxes
  - Opaque computation
  - Opaque data type
- Programs built using RDD API have total control on how to execute every data operation
- Developers have to write efficient programs for different kinds of workloads

Is there an easy way to write efficient programs?

The easiest way to write efficient programs is to not worry about it and get your programs **automatically optimized**

How?

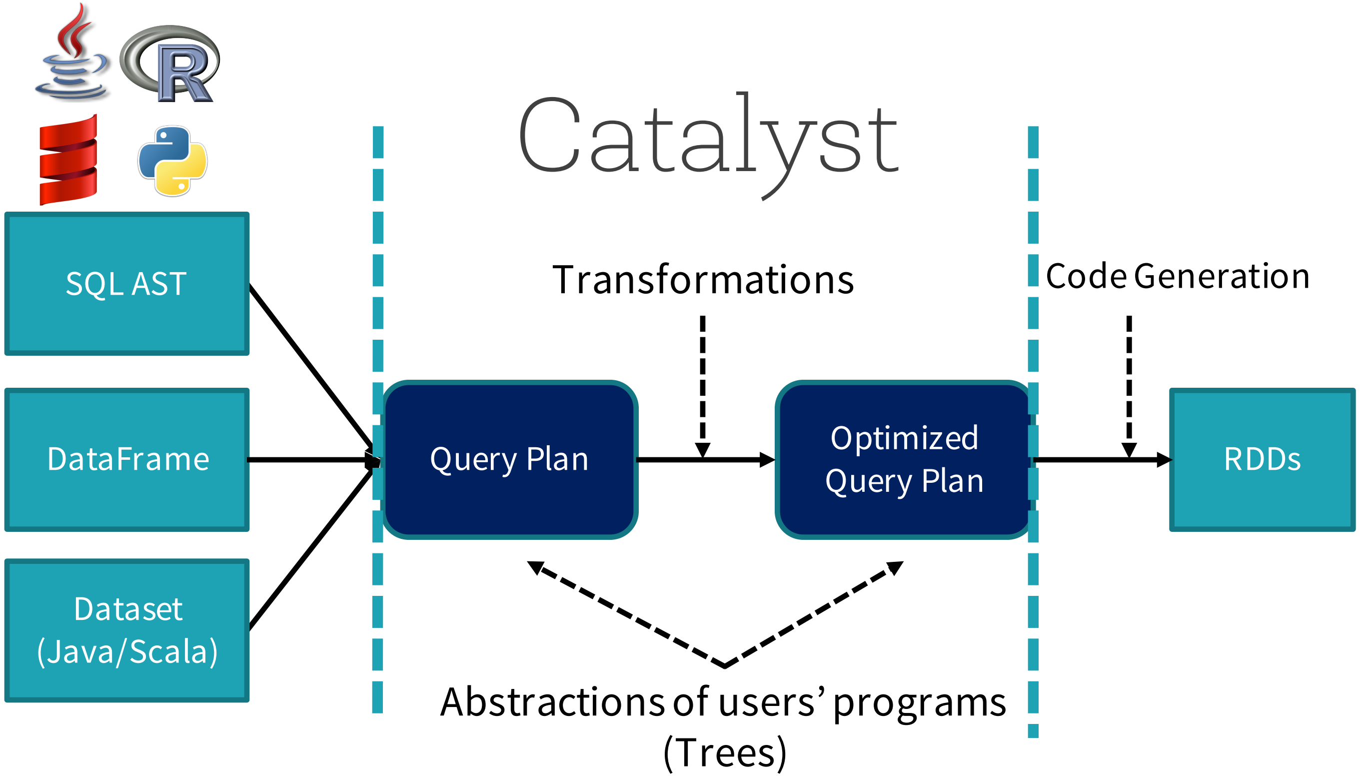
- Write programs using high level programming interfaces
  - Programs are used to describe what data operations are needed without specifying how to execute those operations
  - High level programming interfaces: SQL, DataFrame, and Dataset

- Get an optimizer that **automatically** finds out the most efficient plan to execute data operations specified in the user's program

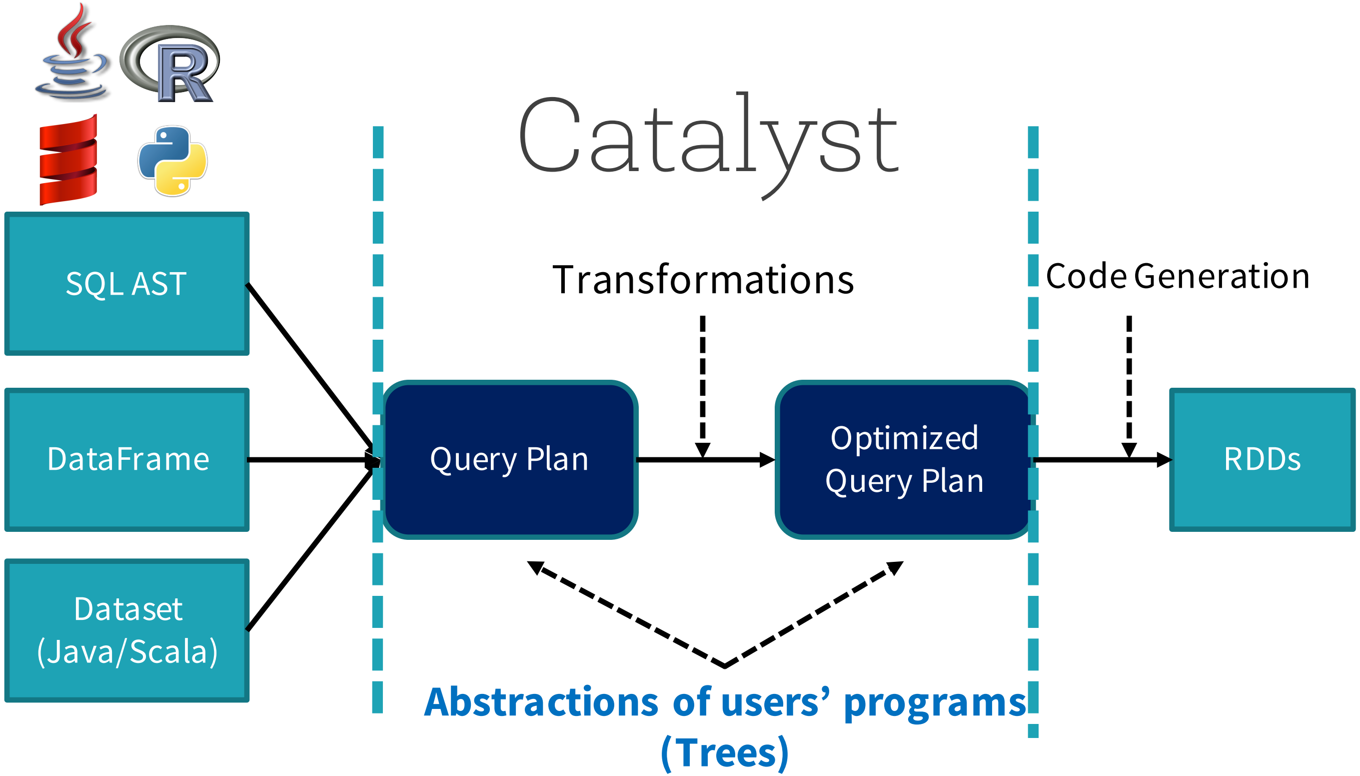
databricks

Catalyst:
Apache Spark's Optimizer

How Catalyst Works: An Overview



How Catalyst Works: An Overview



Trees: Abstractions of Users' Programs

```sql
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```

Trees: Abstractions of Users' Programs

Expression

```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```

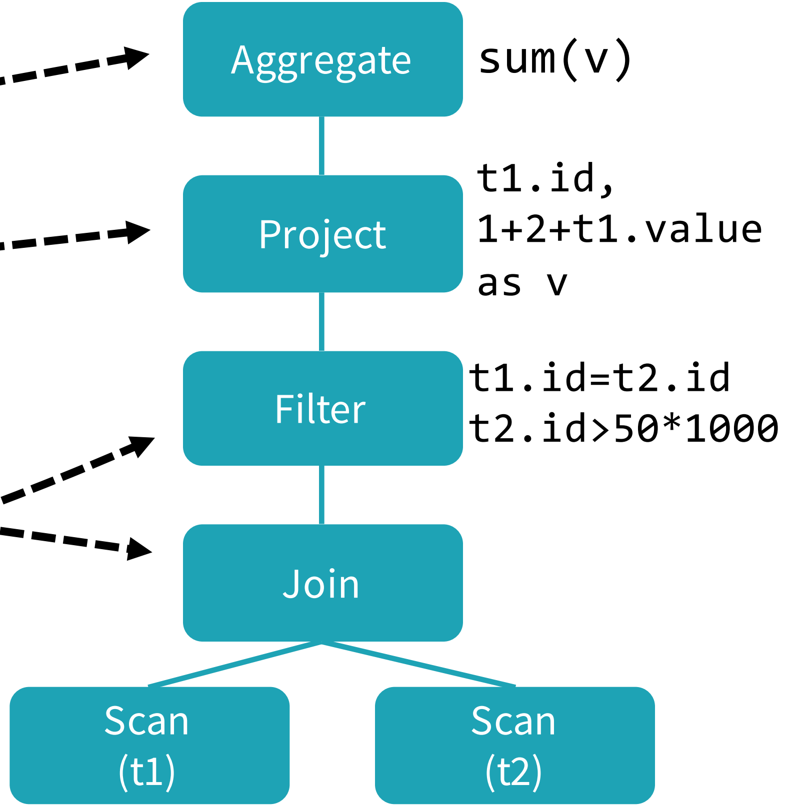
- An expression represents a new value, computed based on input values
  - e.g. `1 + 2 + t1.value`
- Attribute: A column of a dataset (e.g. `t1.id`) or a column generated by a specific data operation (e.g. `v`)

databricks

Trees: Abstractions of Users' Programs

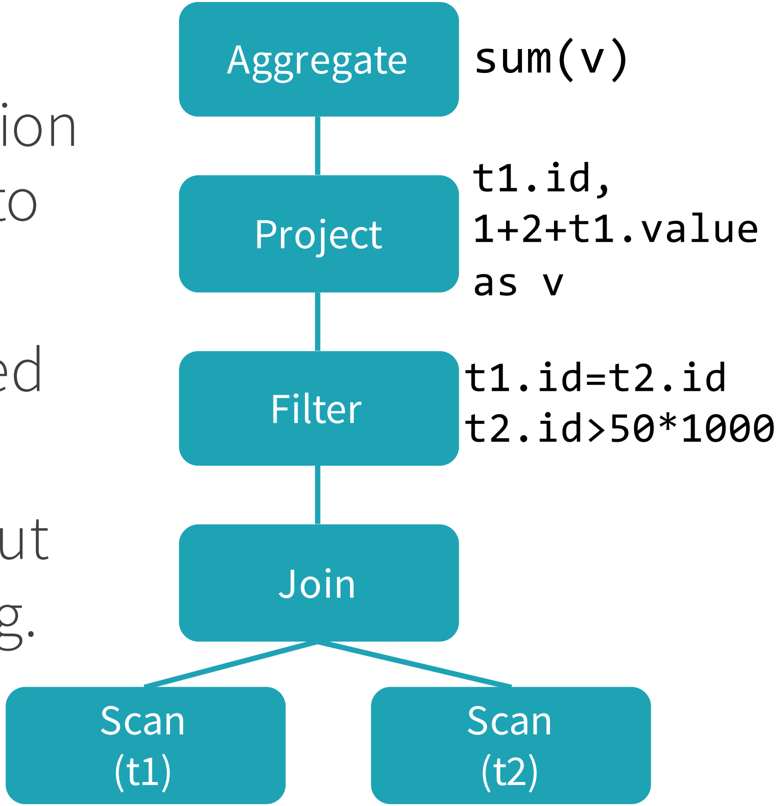
Query Plan

```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```



Logical Plan

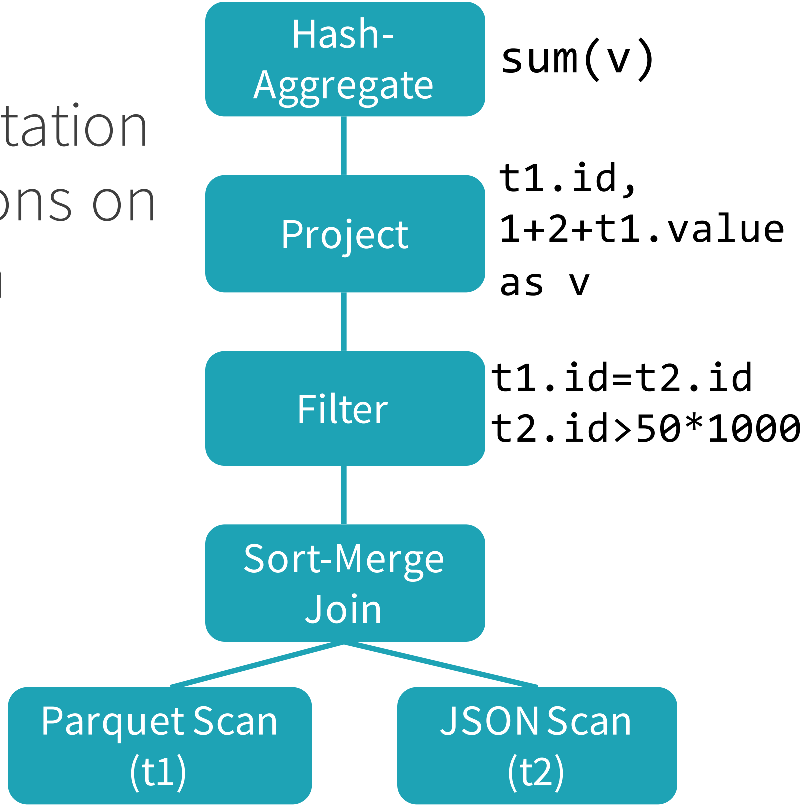
- A Logical Plan describes computation on datasets **without** defining how to conduct the computation
- **output**: a list of attributes generated by this Logical Plan, e.g. [`id, v`]
- **constraints**: a set of invariants about the rows generated by this plan, e.g. `t2.id > 50 * 1000`



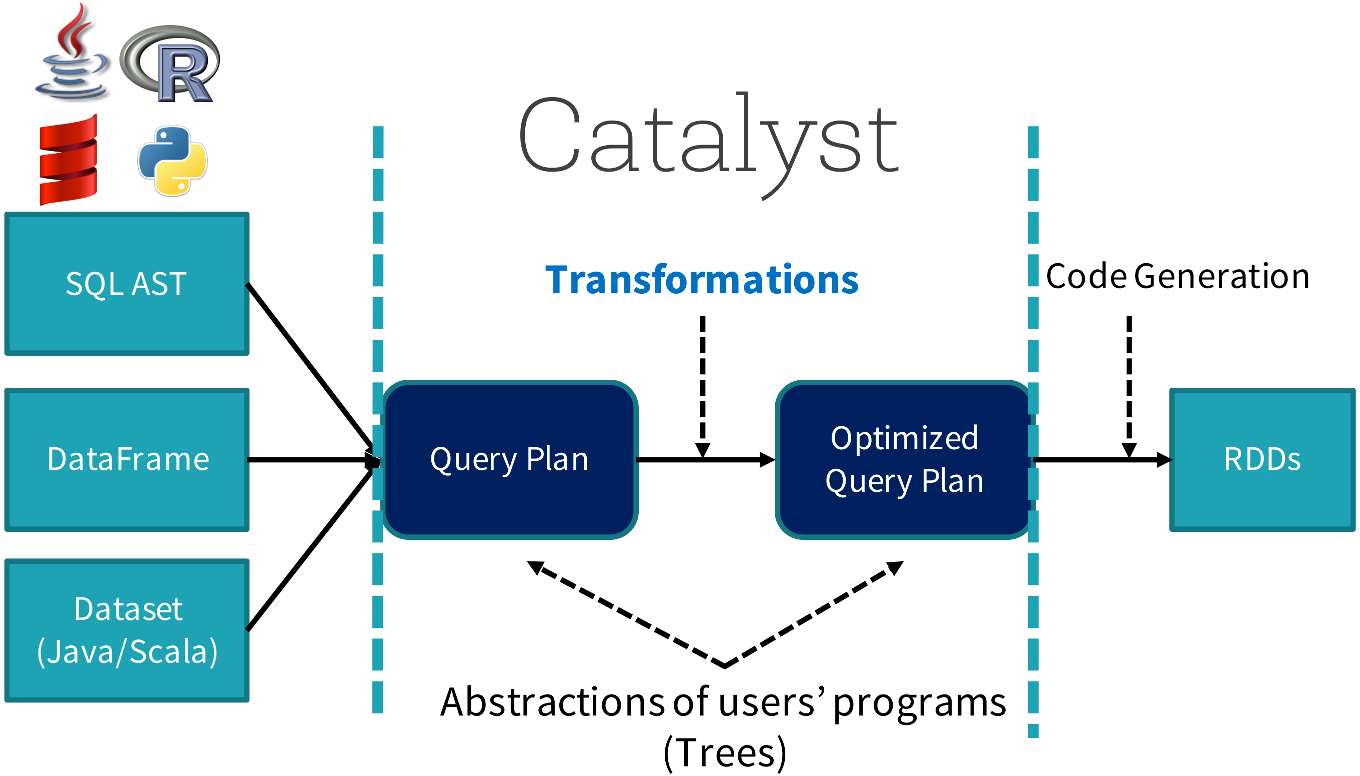
Physical Plan

- A Physical Plan describes computation on datasets with specific definitions on how to conduct the computation

- A Physical Plan is executable



How Catalyst Works: An Overview



Transformations

- Transformations without changing the tree type (Transform and Rule Executor)
  - Expression => Expression
  - Logical Plan => Logical Plan
  - Physical Plan => Physical Plan

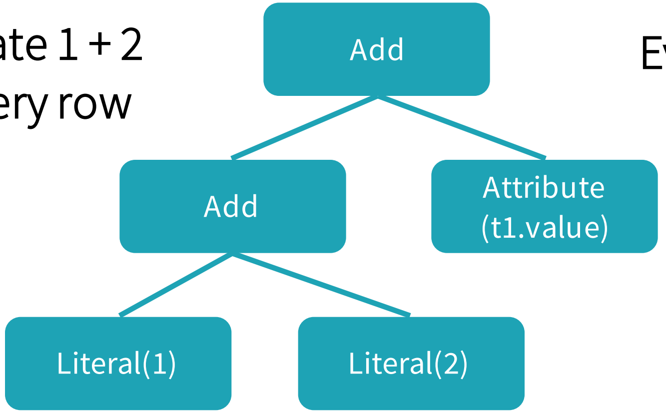
- Transforming a tree to another kind of tree
  - Logical Plan => Physical Plan

Transform

- A function associated with every tree used to implement a single rule

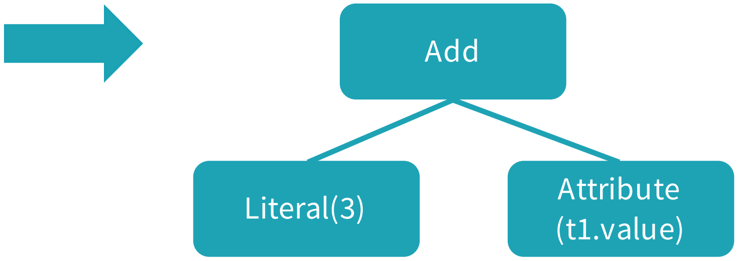
1 + 2 + t1.value

Evaluate 1 + 2 for every row



Evaluate 1 + 2 once

3+ t1.value



Transform

- A transformation is defined as a Partial Function
- Partial Function: A function that is defined for a subset of its possible arguments

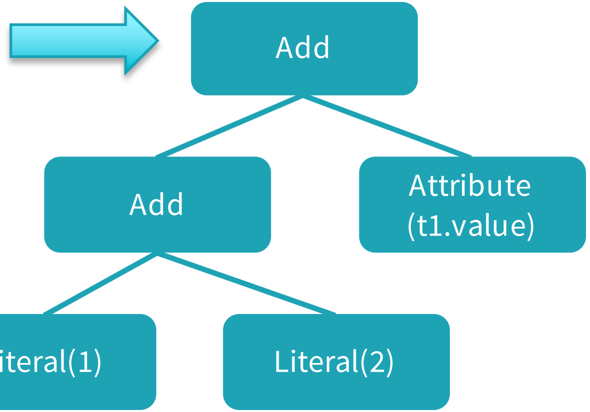
```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

Case statement determine if the partial function is defined for a given input

Transform

```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

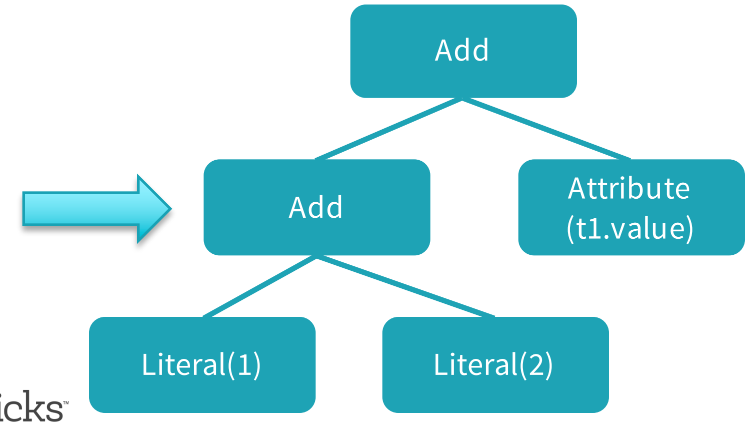
1 + 2 + t1.value



Transform

```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

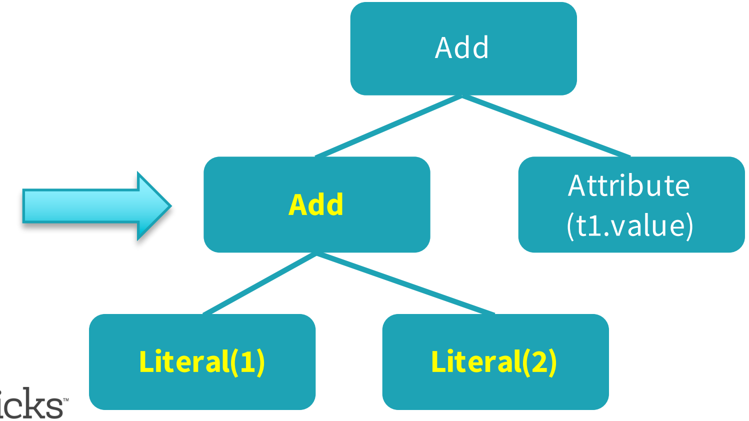
1 + 2 + t1.value



Transform

```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

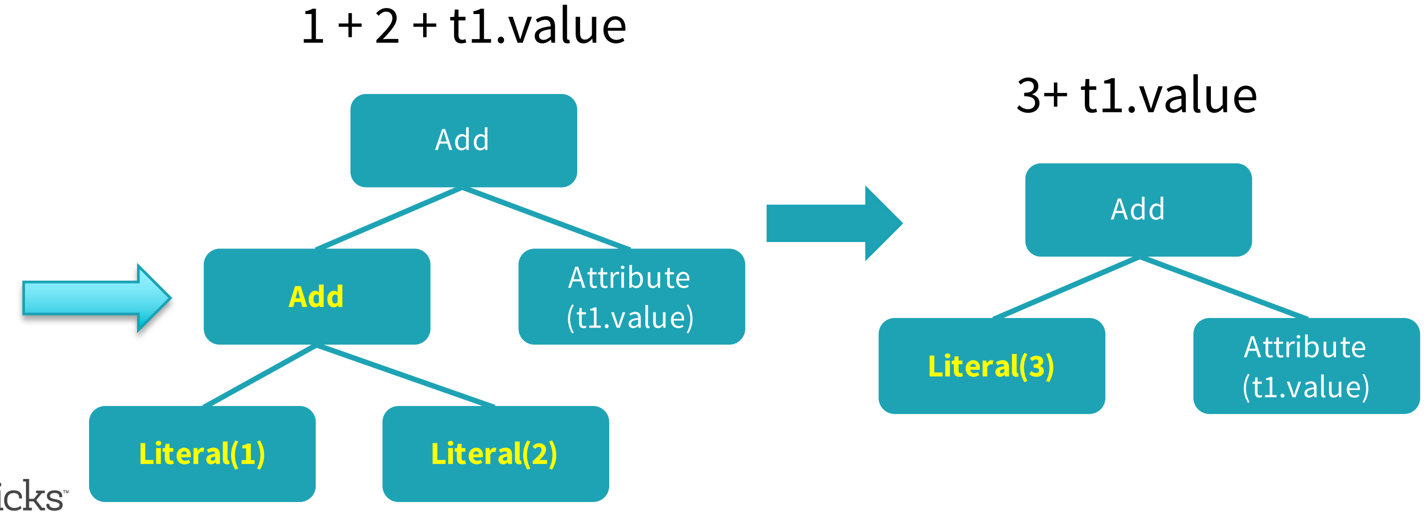
1 + 2 + t1.value



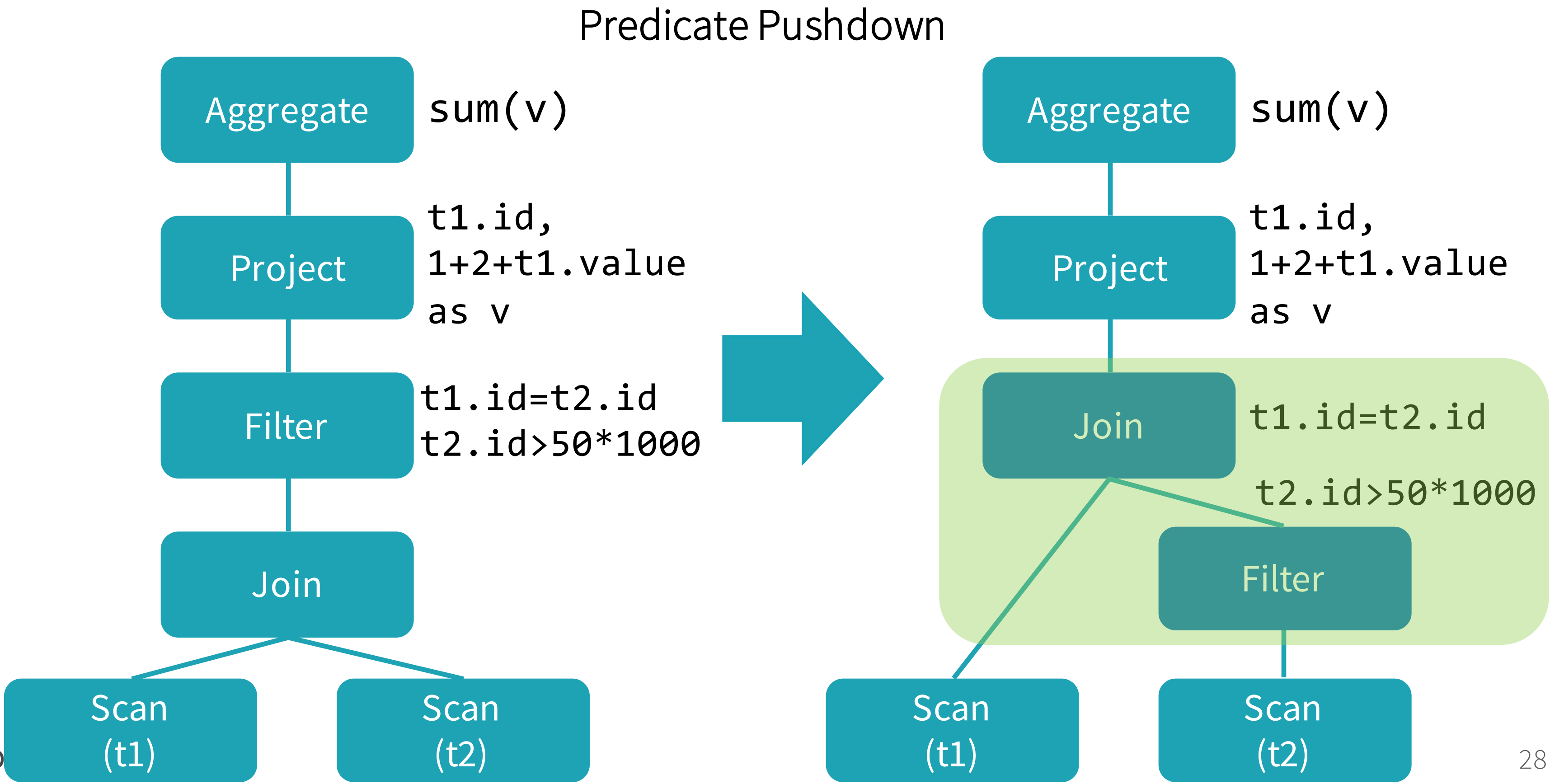
databricks

Transform

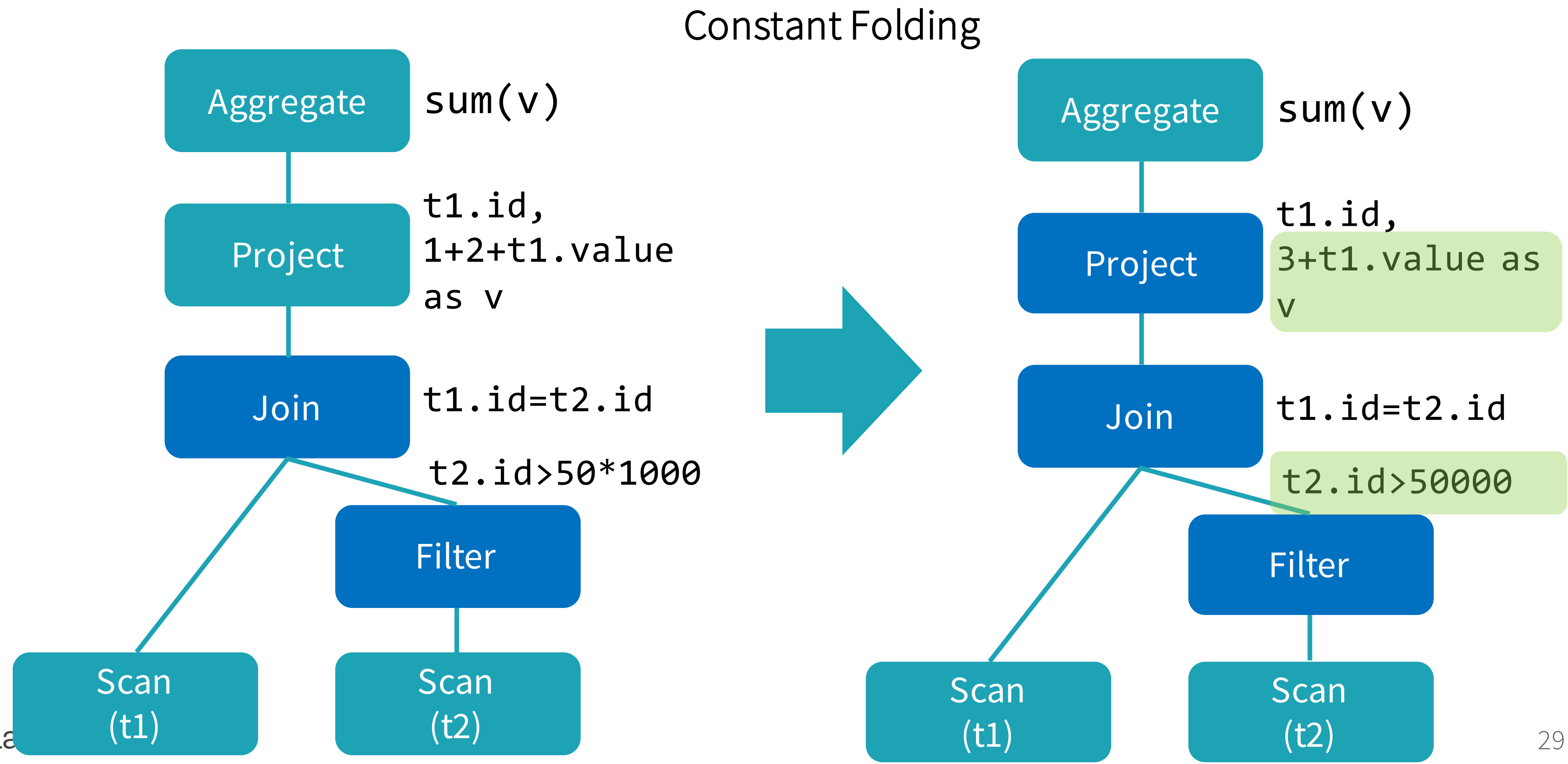
```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```



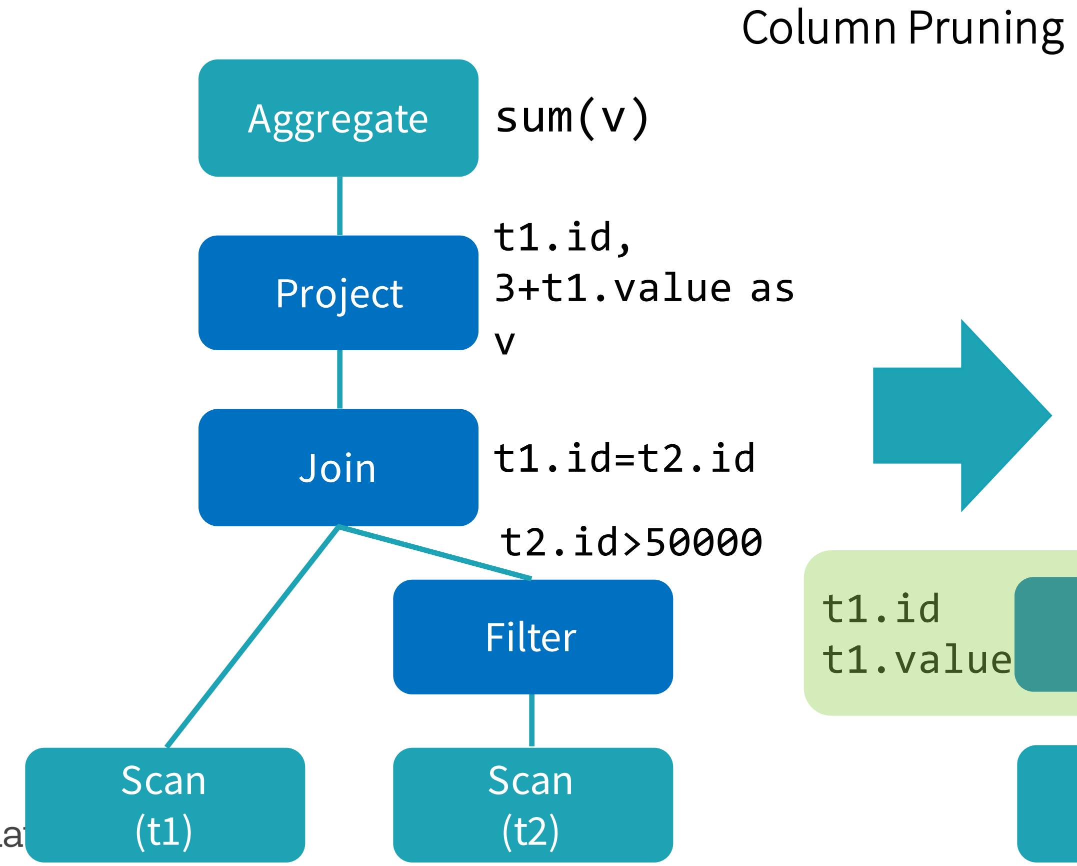
Combining Multiple Rules

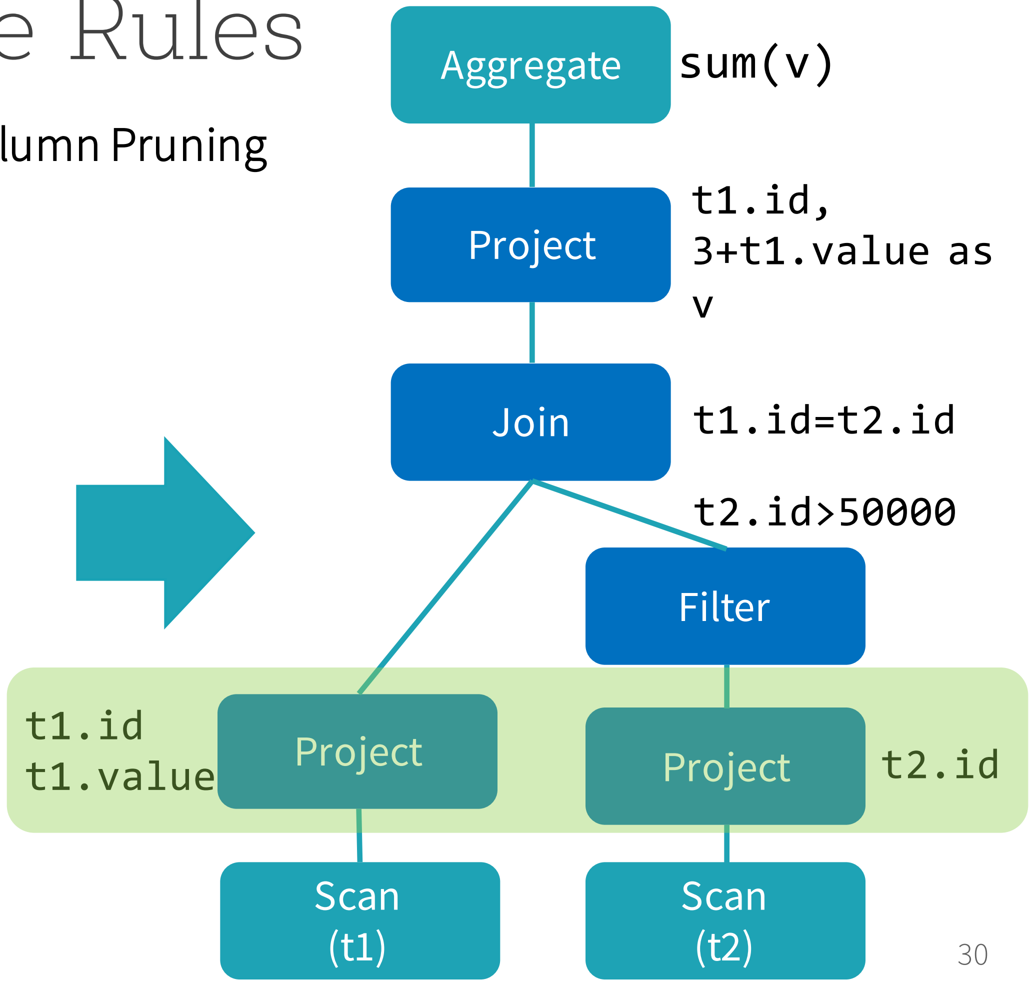


Combining Multiple Rules

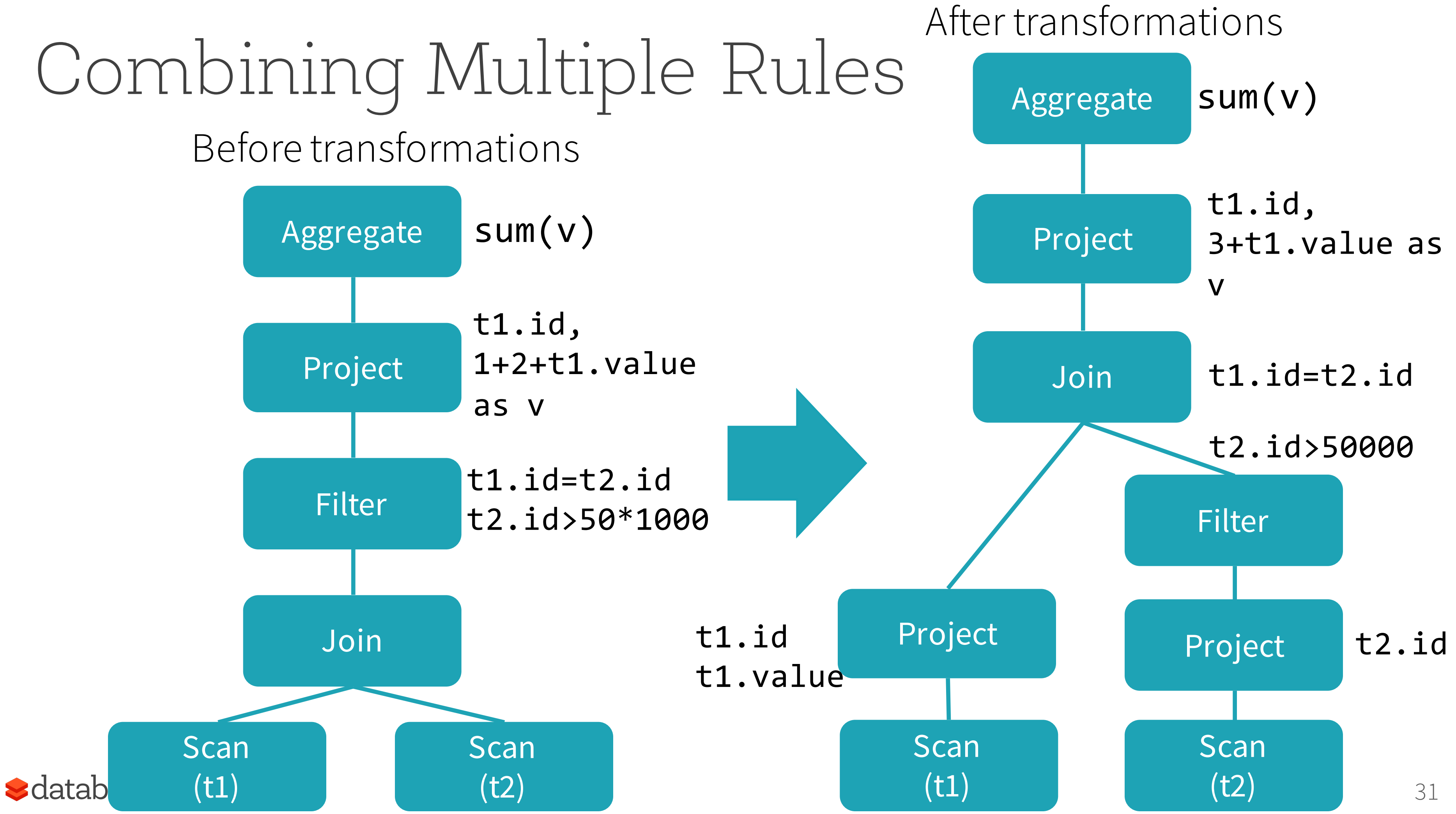


Combining Multiple Rules



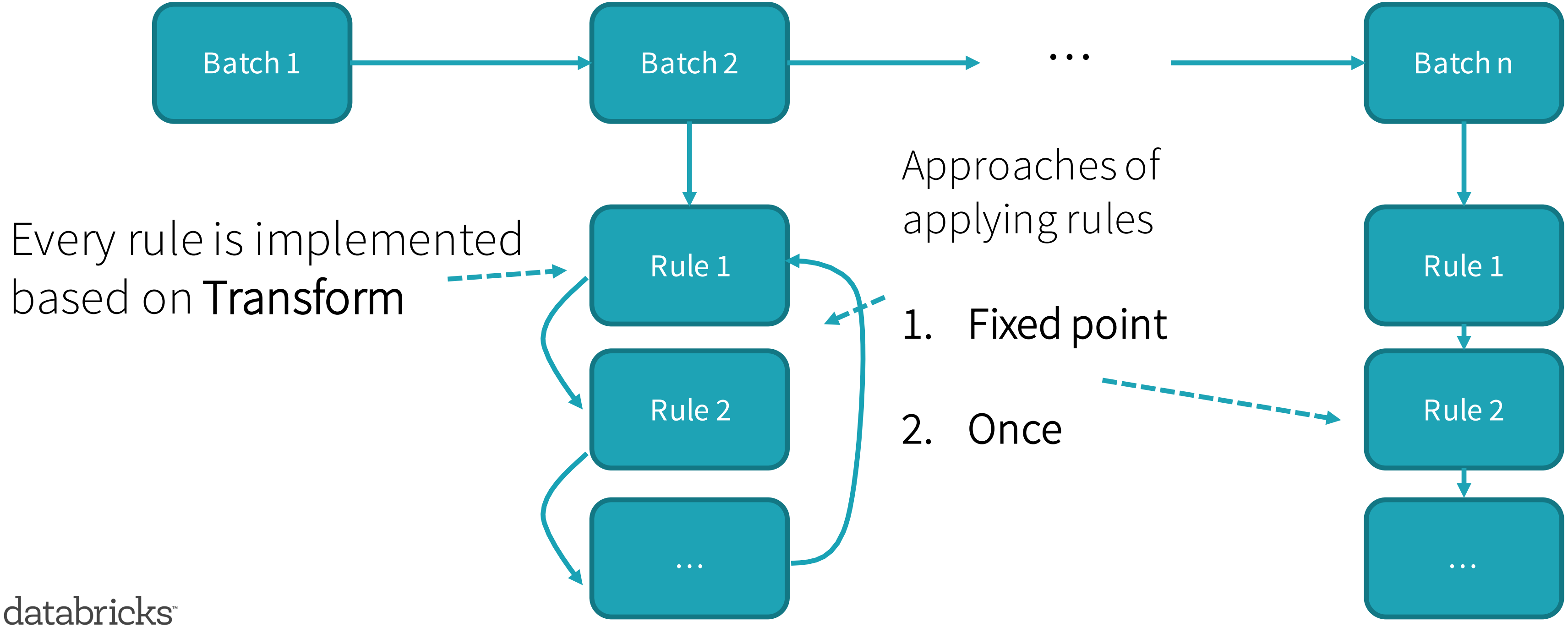


Combining Multiple Rules



Combining Multiple Rules: Rule Executor

A Rule Executor transforms a Tree to another same type Tree by applying many rules defined in batches



Transformations

- Transformations without changing the tree type (Transform and Rule Executor)
  - Expression => Expression
  - Logical Plan => Logical Plan
  - Physical Plan => Physical Plan

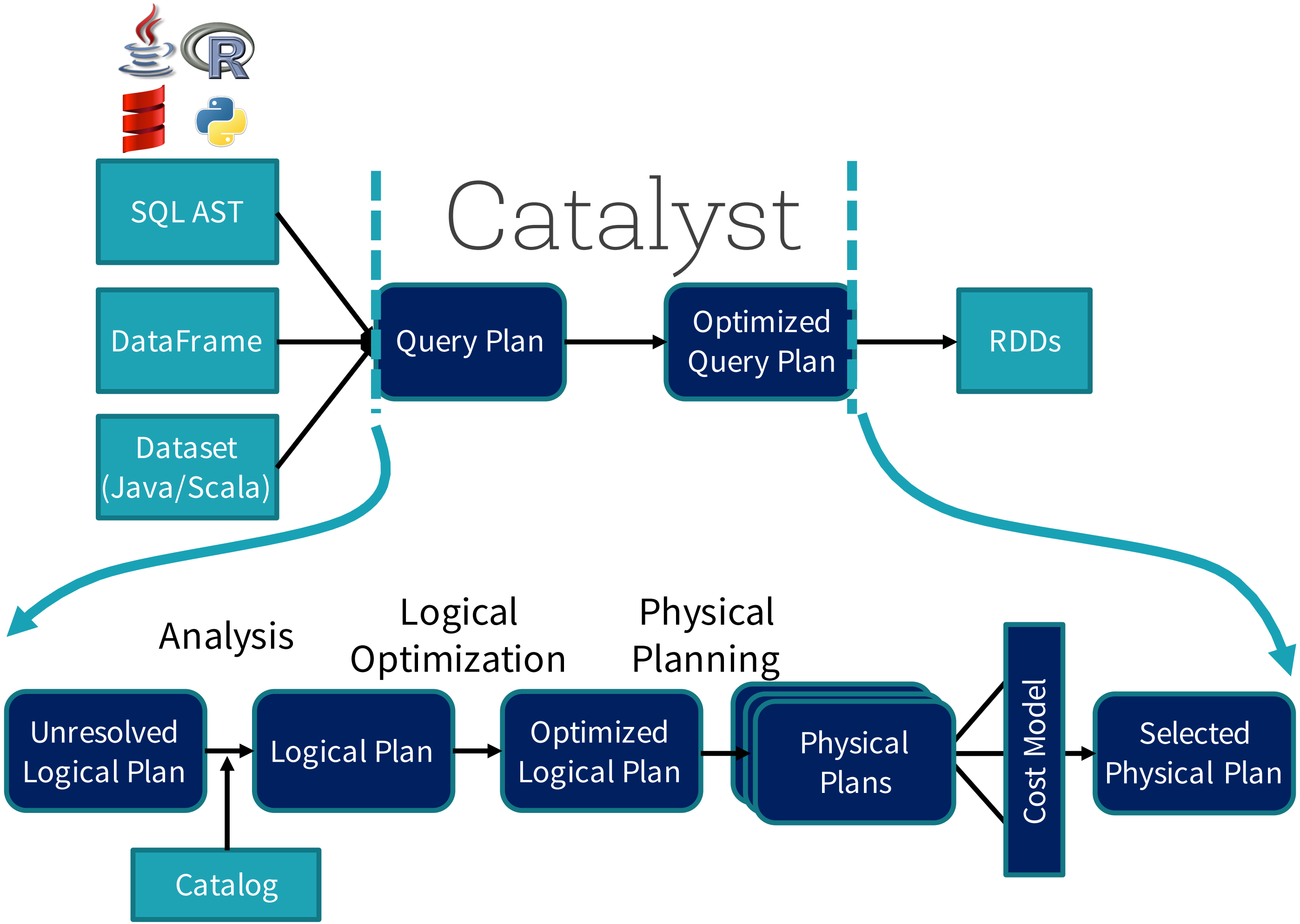
- Transforming a tree to another kind of tree
  - Logical Plan => Physical Plan

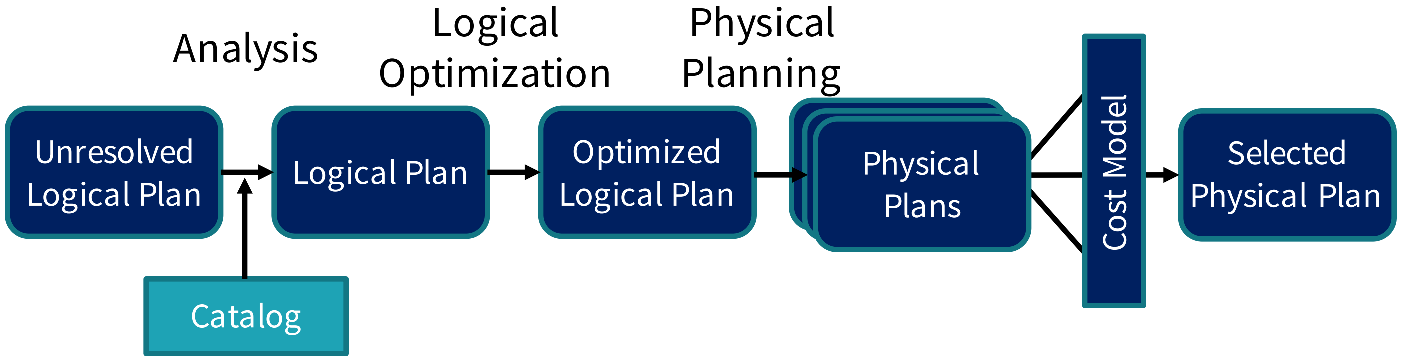
From Logical Plan to Physical Plan

- A Logical Plan is transformed to a Physical Plan by applying a set of Strategies
- Every Strategy uses pattern matching to convert a Tree to another kind of Tree

```
object BasicOperators extends Strategy {
  def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
    …
    case logical.Project(projectList, child) =>
      execution.ProjectExec(projectList, planLater(child)) :: Nil
    case logical.Filter(condition, child) =>
      execution.FilterExec(condition, planLater(child)) :: Nil
    …
  }
}
```

Triggers other Strategies


SQL AST
DataFrame
Dataset (Java/Scala)
Catalyst
Query Plan
Optimized Query Plan
RDDs
Analysis
Logical Optimization
Physical Planning
Unresolved Logical Plan
Logical Plan
Optimized Logical Plan
Physical Plans
Cost Model
Selected Physical Plan
Catalog



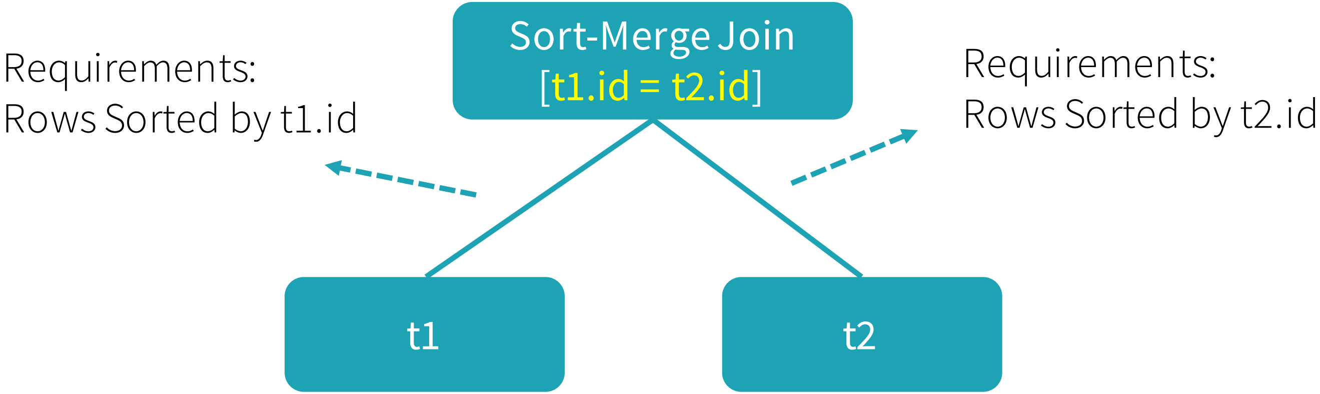
- **Analysis (Rule Executor):** Transforms an Unresolved Logical Plan to a Resolved Logical Plan
  - Unresolved => Resolved: Use Catalog to find where datasets and columns are coming from and types of columns
- **Logical Optimization (Rule Executor):** Transforms a Resolved Logical Plan to an Optimized Logical Plan
- **Physical Planning (Strategies + Rule Executor):** Transforms a Optimized Logical Plan to a Physical Plan

Spark's Planner

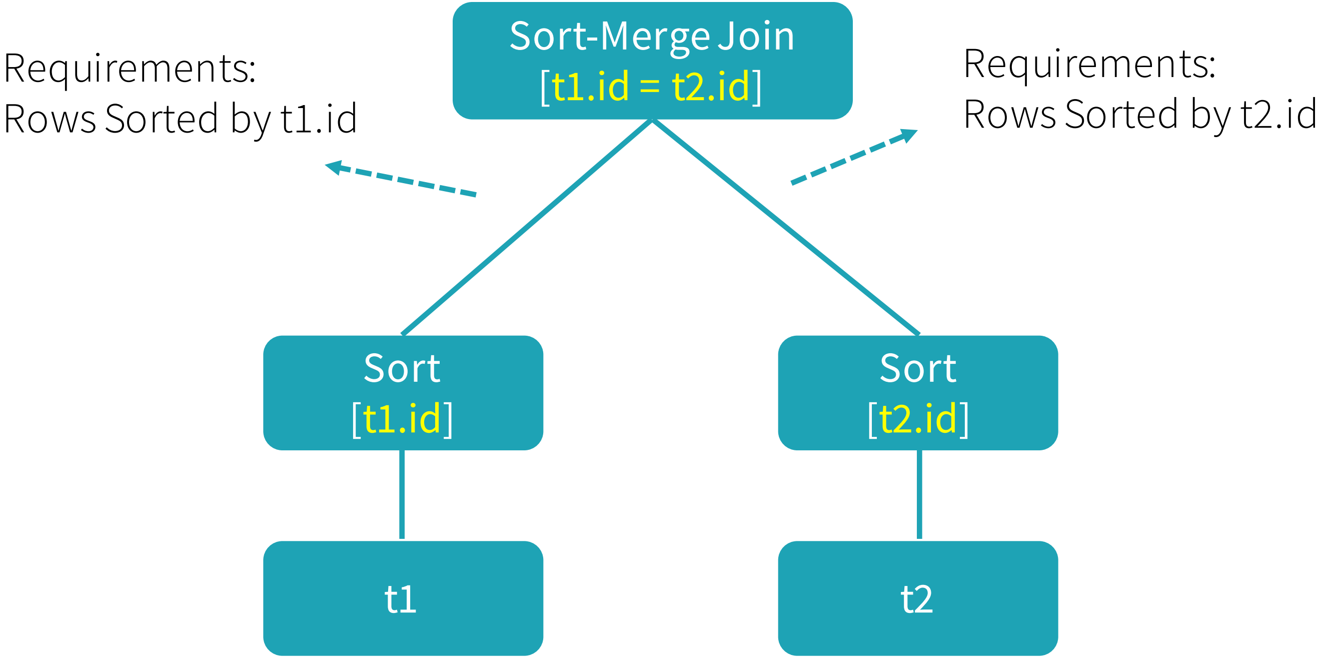
- 1st Phase: Transforms the Logical Plan to the Physical Plan using Strategies

- 2nd Phase: Use a Rule Executor to make the Physical Plan ready for execution
  - Prepare Scalar sub-queries
  - Ensure requirements on input rows
  - Apply physical optimizations

Ensure Requirements on Input Rows



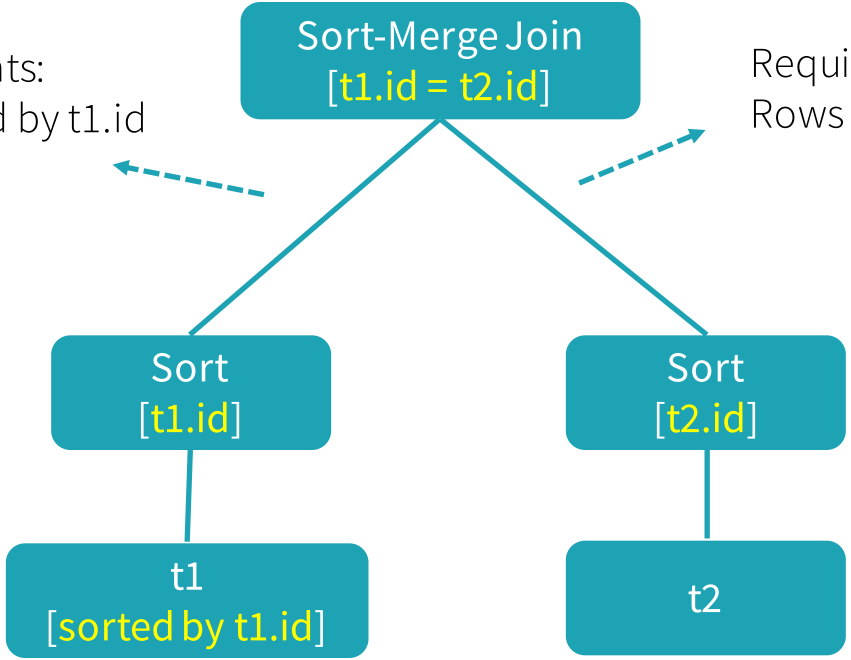
Ensure Requirements on Input Rows



Ensure Requirements on Input Rows

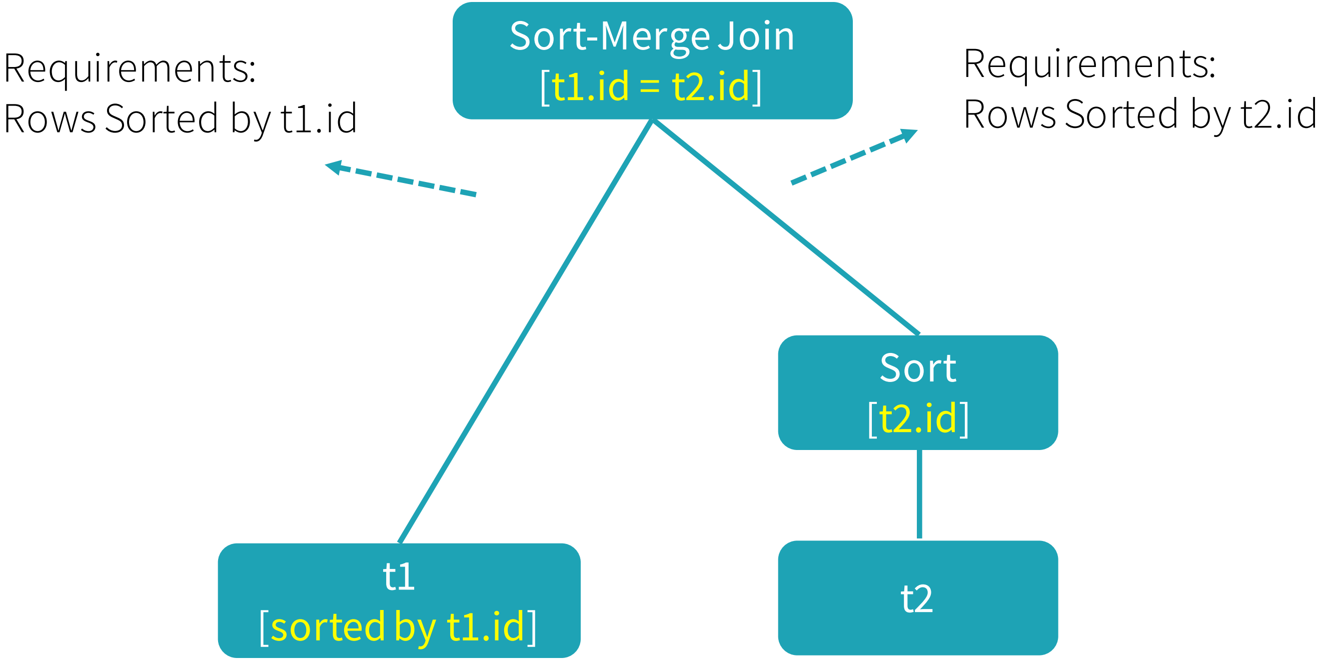
Requirements:
Rows Sorted by t1.id

Requirements:
Rows Sorted by t2.id



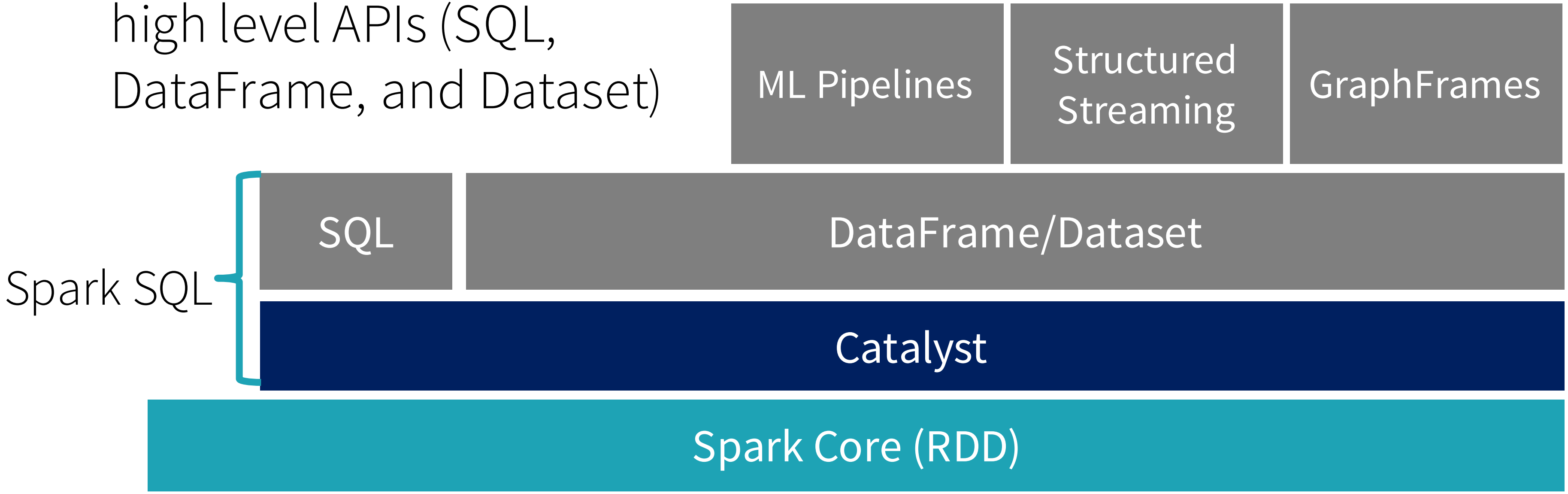
What if t1 has been sorted by t1.id?

Ensure Requirements on Input Rows



Catalyst in Apache Spark

With Spark 2.0, we expect most users to migrate to high level APIs (SQL, DataFrame, and Dataset)



{ JSON } HIVE HDFS amazon web services S3 APACHE HBASE cassandra

JDBC Parquet MySQL elasticsearch.

and more…

databricks

Where to Start

- Source Code:
  - Trees: TreeNode, Expression, Logical Plan, and Physical Plan
  - Transformations: Analyzer, Optimizer, and Planner
- Check out previous pull requests
- Start to write code using Catalyst

SPARK-12032

[SPARK-12032] [SQL] Re-order inner joins to do join with conditions f…

Browse files

…irst

Currently, the order of joins is exactly the same as SQL query, some conditions may not pushed down to the correct join, then those join will become cross product and is extremely slow.

This patch try to re-order the inner joins (which are common in SQL query), pick the joins that have self-contain conditions first, delay those that does not have conditions.

After this patch, the TPCDS query Q64/65 can run hundreds times faster.

cc marmbrus nongli

Author: Davies Liu <davies@databricks.com>

Closes #10073 from davies/reorder_joins.

master (#1) 2.0.0-preview

davies committed with **davies** on Dec 7, 2015

1 parent 6fd9e70 commit 9cde7d5fa87e7ddfff0b9c1212920a1d9000539b

Showing **3 changed files** with **185 additions** and **6 deletions**.

Unified Split

Certain workloads can run hundreds times faster

~ 200 lines of changes (95 lines are for tests)

databricks

SPARK-8992

Pivot table support

[SPARK-8992][SQL] Add pivot to dataframe api

Browse files

```
This adds a pivot method to the dataframe api.

Following the lead of cube and rollup this adds a Pivot operator that is translated into an Aggregate by the analyzer.

Currently the syntax is like:
~~courseSales.pivot(Seq($"year"), $"course", Seq("dotNET", "Java"), sum($"earnings"))~~

~~Would we be interested in the following syntax also/alternatively? and~~

    courseSales.groupBy($"year").pivot($"course", "dotNET", "Java").agg(sum($"earnings"))
    //or
    courseSales.groupBy($"year").pivot($"course").agg(sum($"earnings"))

Later we can add it to `SQLParser`, but as Hive doesn't support it we cant add it there, right?

~~Also what would be the suggested Java friendly method signature for this?~~

Author: Andrew Ray <ray.andrew@gmail.com>

Closes #7841 from aray/sql-pivot.
```

master (#3) **2.0.0-preview**

aray committed with **yhuai** on Nov 11, 2015 — 1 parent 1a21be1 commit b8ff6888e76d437287d[illegible]2d4b9c759710a195

~ 250 lines of changes

Showing **6 changed files** with **255 additions** and **10 deletions**. Unified Split

(99 lines are for tests)

databricks

Try Apache Spark with Databricks

- Try latest version of Apache Spark and preview of Spark 2.0

  http://databricks.com/try

FULL-PLATFORM TRIAL

Put Apache® Spark™ to work

- Unlimited clusters
- Notebooks, dashboards, production jobs, RESTful APIs
- Interactive guide to Spark and Databricks
- Deployed to your AWS VPC
- BI tools integration

14-Day Free Trial

START YOUR TRIAL TODAY

COMMUNITY EDITION

Learn Apache® Spark™ for free

- Mini 6GB cluster for learning Spark
- Interactive notebooks and dashboards
- Online learning resources
- Public environment to share your work

Free

SIGN UP

Thank you.

Office hour: 2:45pm – 3:30pm @ Expo Hall

databricks™