Deep Dive Into Catalyst: Apache Spark's Optimizer

Yin Huai, yhuai@databricks.com

2017-06-06, Spark Summit

databricks

About me

- Software engineer at Databricks
- Apache Spark committer and PMC member
- One of the original developers of Spark SQL
- Before joining Databricks: Ohio State University

About Databricks

TEAM

Started Spark project (now Apache Spark) at UC Berkeley in 2009

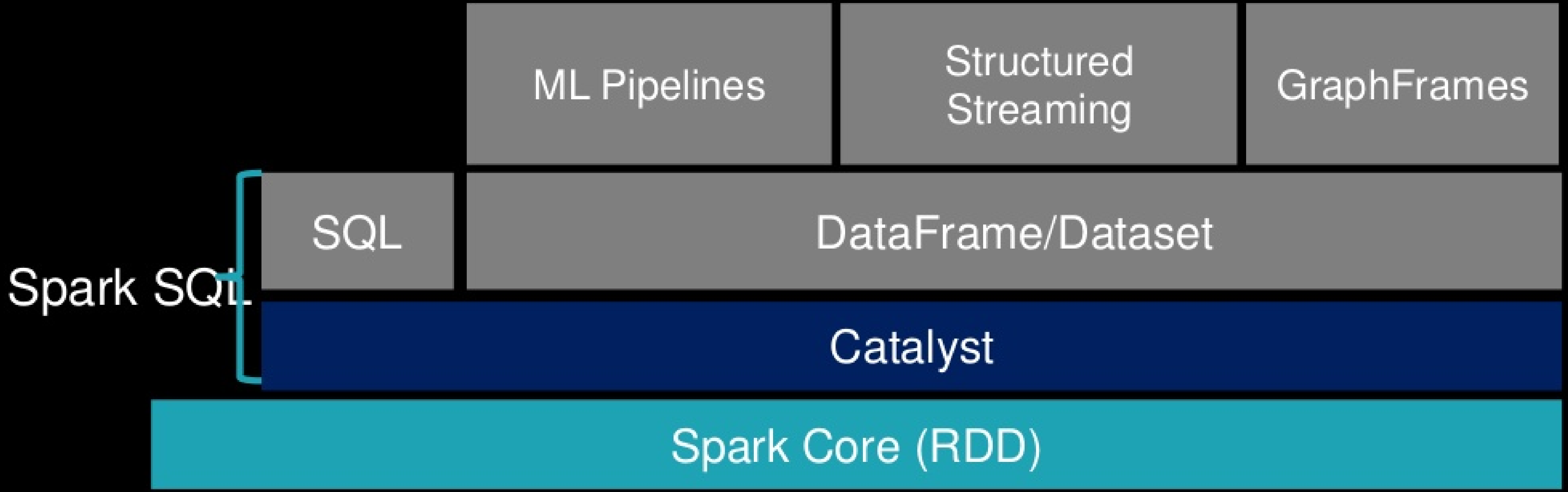
MISSION

Making Big Data Simple

PRODUCT

Unified Analytics Platform

Overview



Spark SQL applies structured views to data from different systems stored in different kinds of formats.

Why structure APIs?

Dataframe

```
data.groupBy("dept").avg("age")
```

SQL

```
select dept, avg(age) from data group by 1
```

RDD

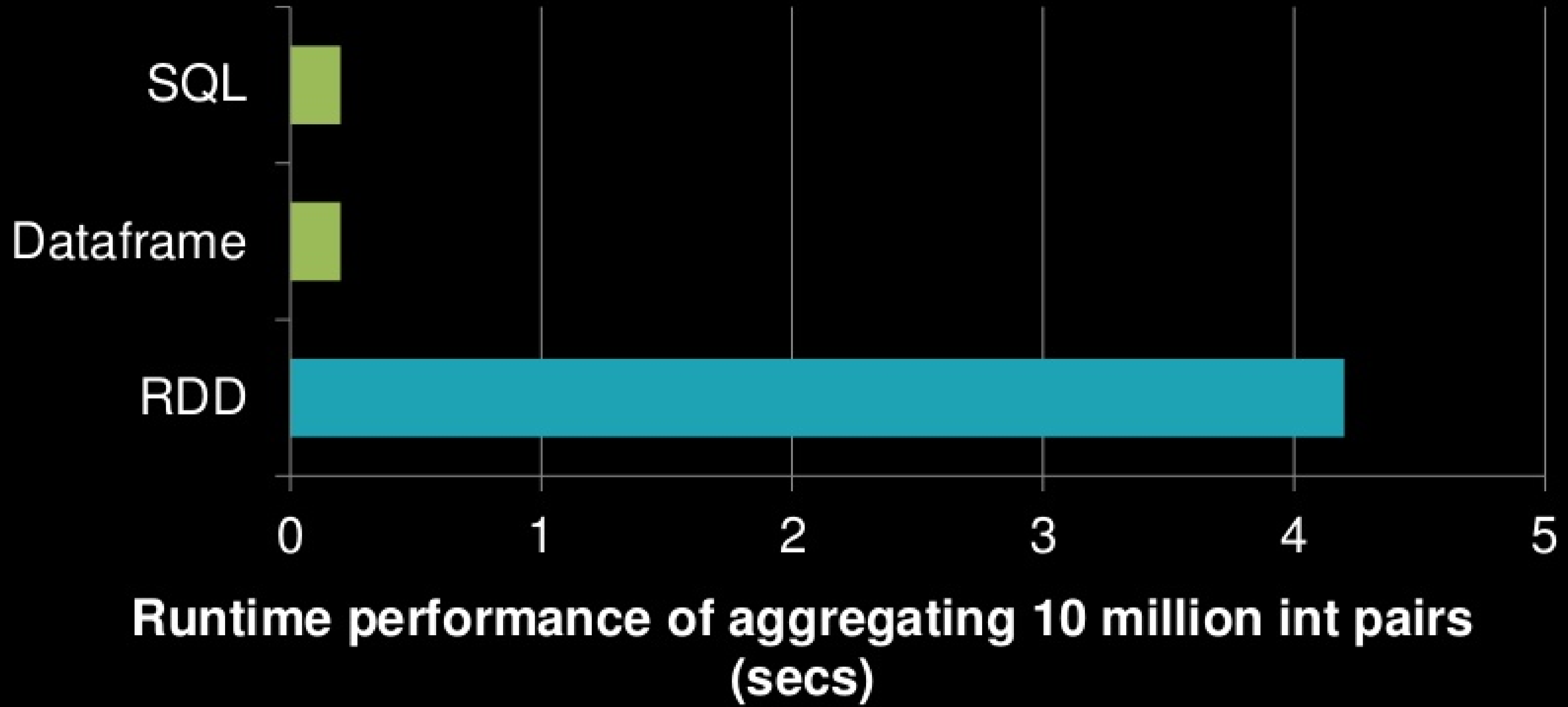
```
data.map { case (dept, age) => dept -> (age, 1) }
    .reduceByKey { case ((a1, c1), (a2, c2)) => (a1 + a2, c1 + c2)}
    .map { case (dept, (age, c)) => dept -> age / c }
```

Why structure APIs?

- Structure will *limit* what can be expressed.
- In practice, we can accommodate the vast majority of computations.

Limiting the space of what can be expressed enables optimizations.

Why structure APIs?



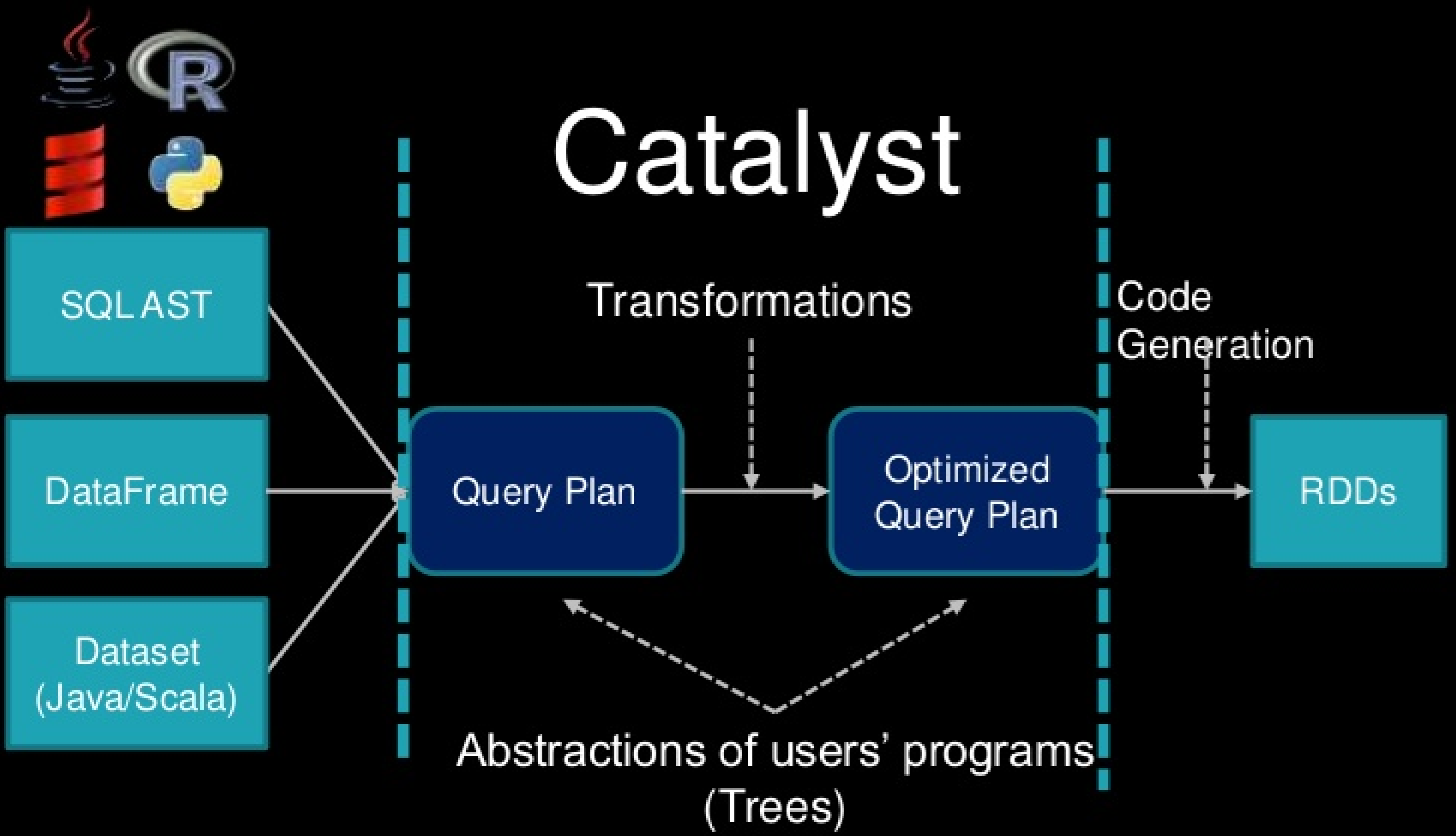
How to take advantage of optimization opportunities?

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

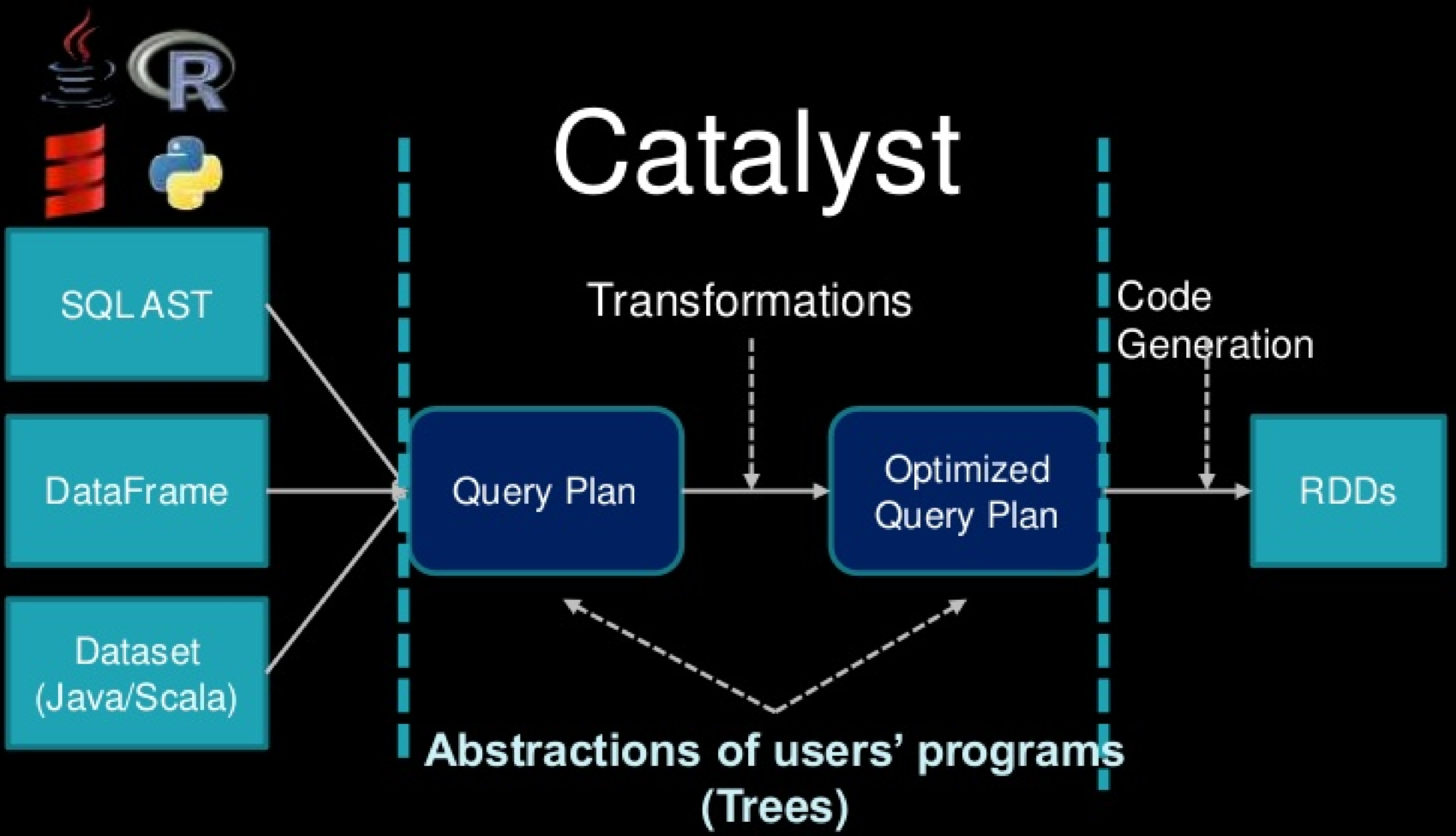
Catalyst:
Apache Spark’s Optimizer

databricks

How Catalyst Works: An Overview



How Catalyst Works: An Overview



Trees: Abstractions of Users' Programs

```
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 > 50000) tmp
```

Trees: Abstractions of Users' Programs

Expression

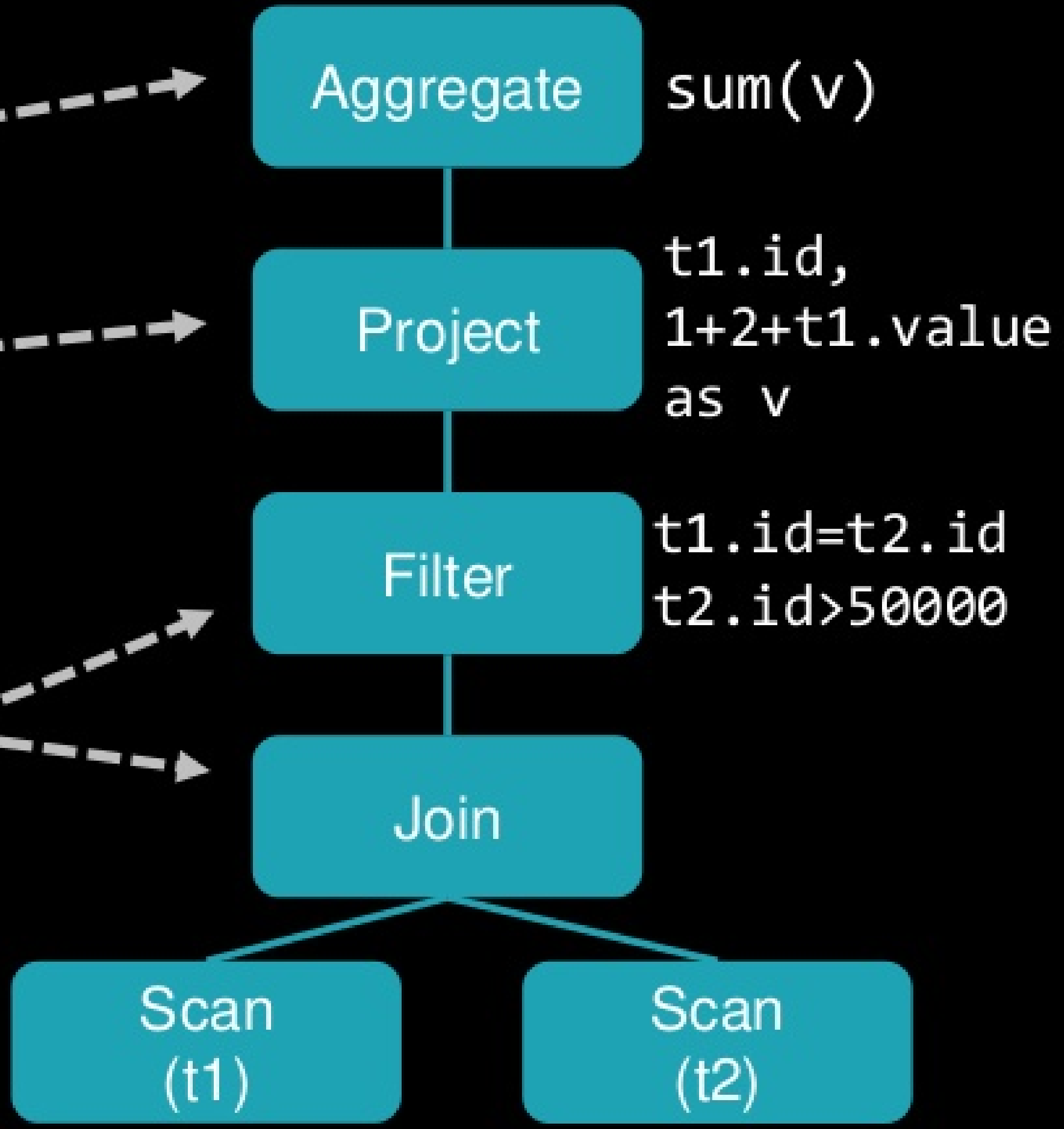
```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50000) tmp
```

- 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)

Trees: Abstractions of Users' Programs

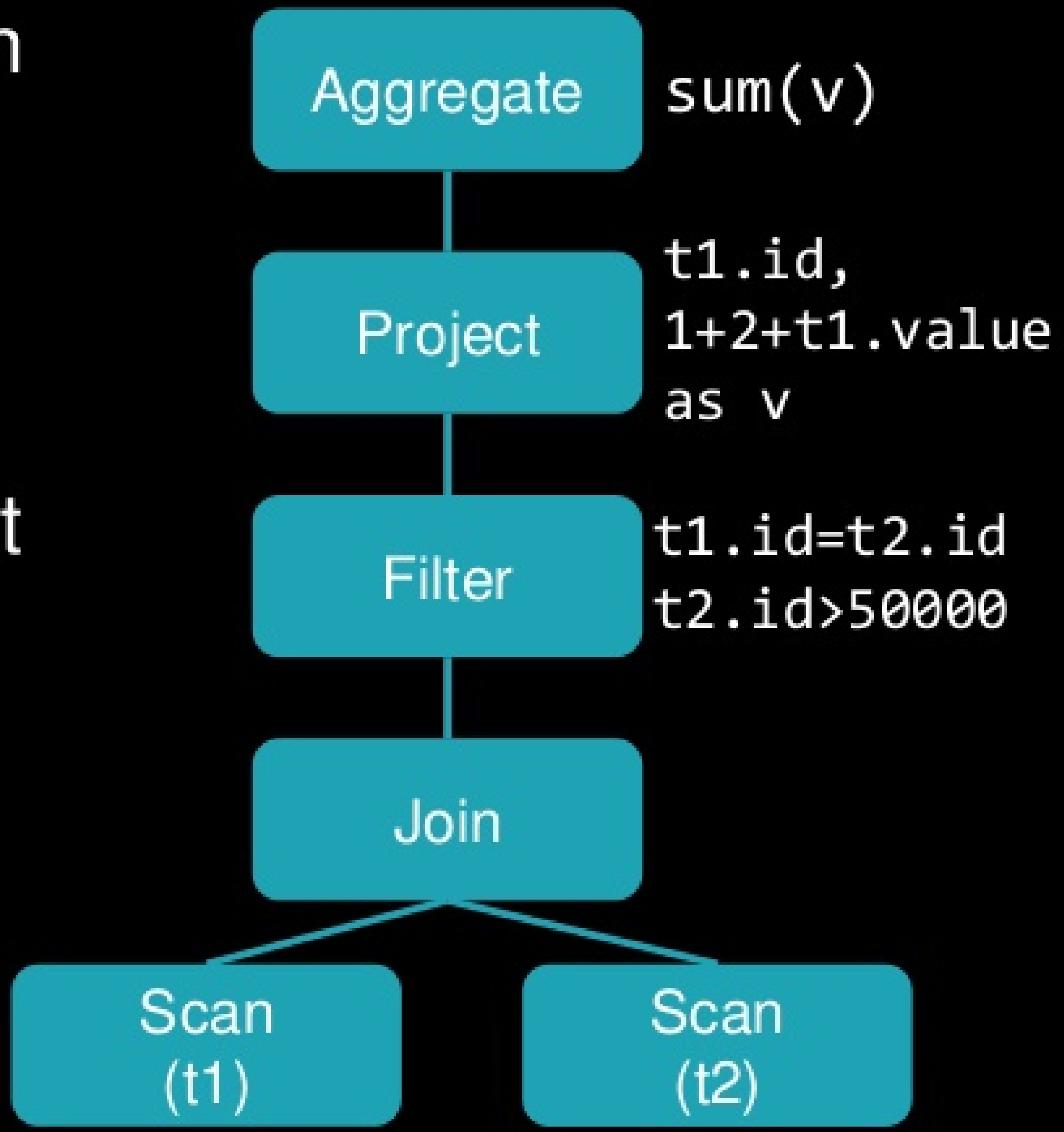
Query Plan

```
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 > 50000) tmp
```



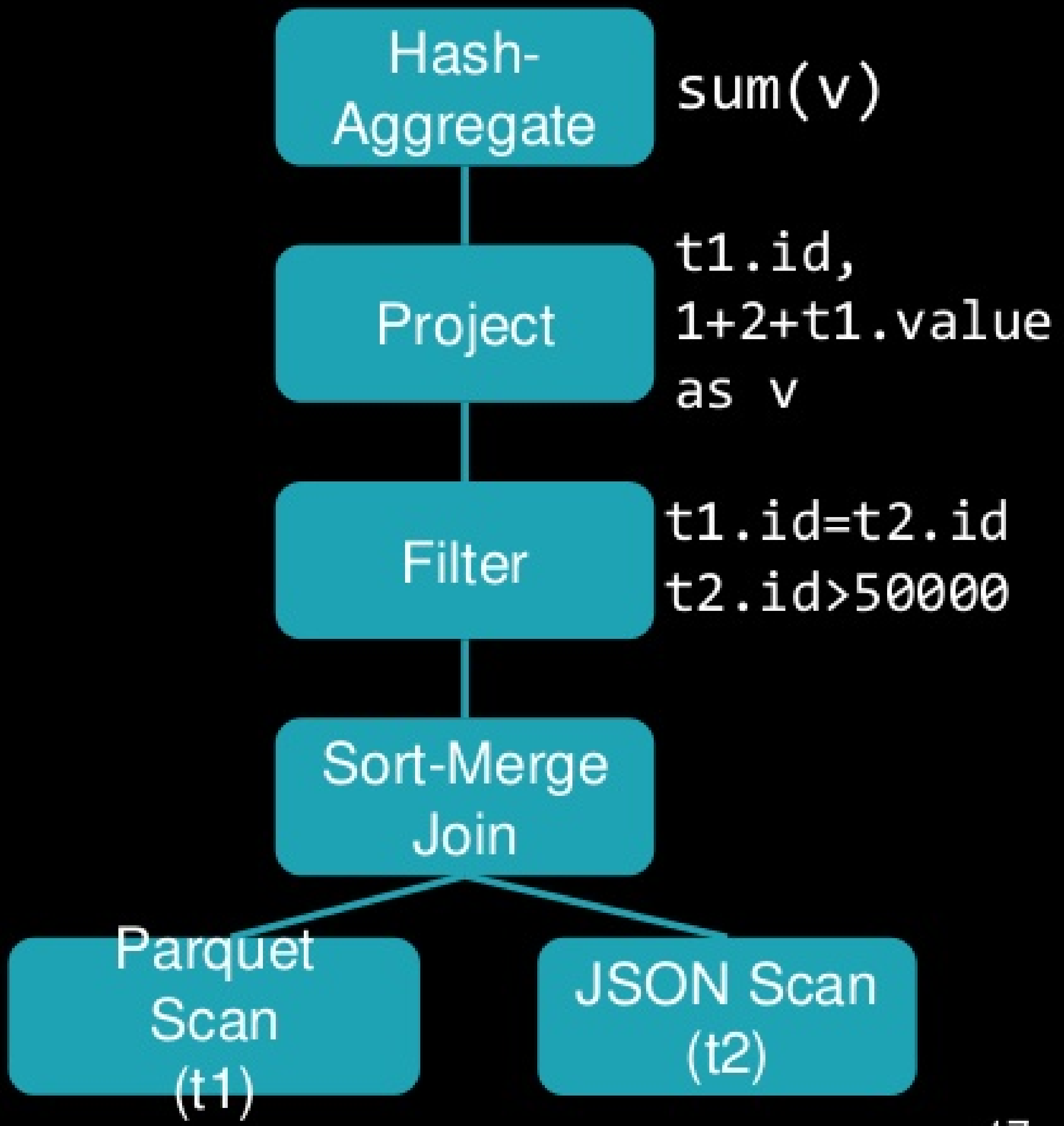
Logical Plan

- 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 > 50000`
- **statistics**: size of the plan in rows/bytes. Per column stats (min/max/ndv/nulls).

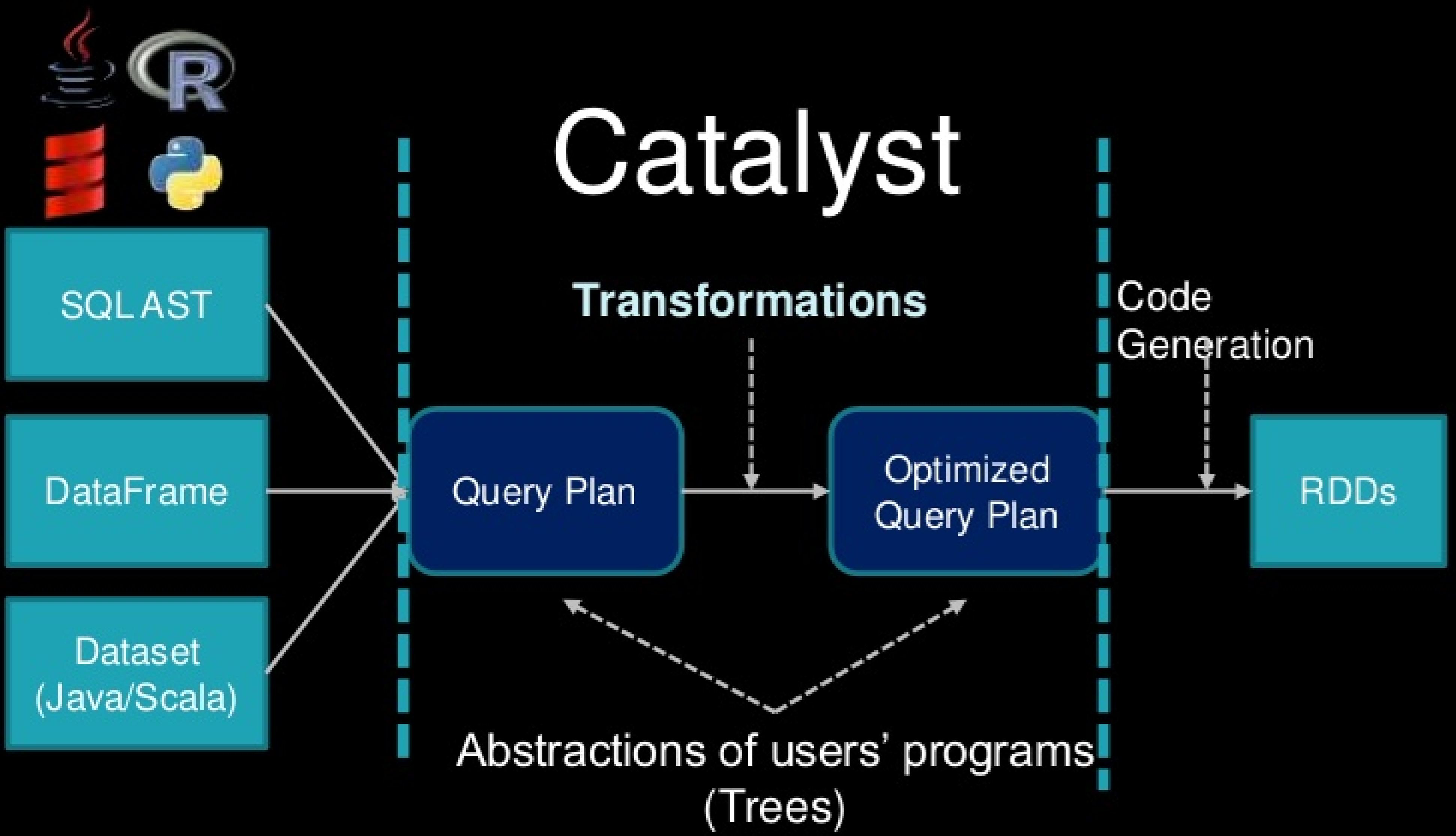


Physical Plan

- 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

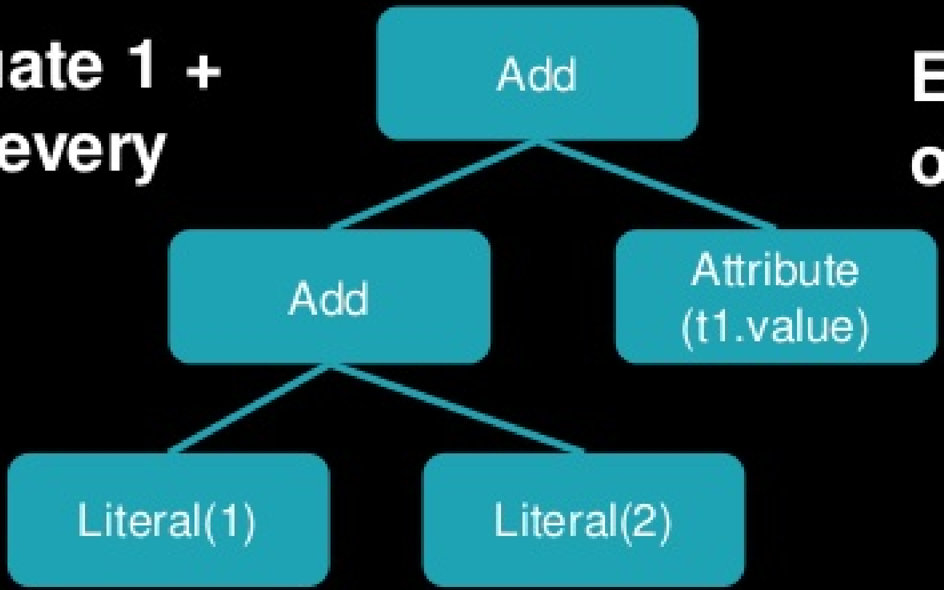
- Transforming a tree to another kind of tree
  - Logical Plan => Physical Plan

Transform

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

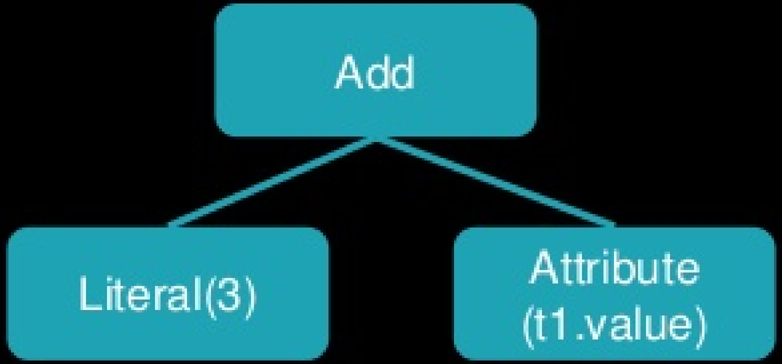
1 + 2 + t1.value

Evaluate 1 + 2 for every row



Evaluate 1 + 2 once

3+ t1.value



Transform

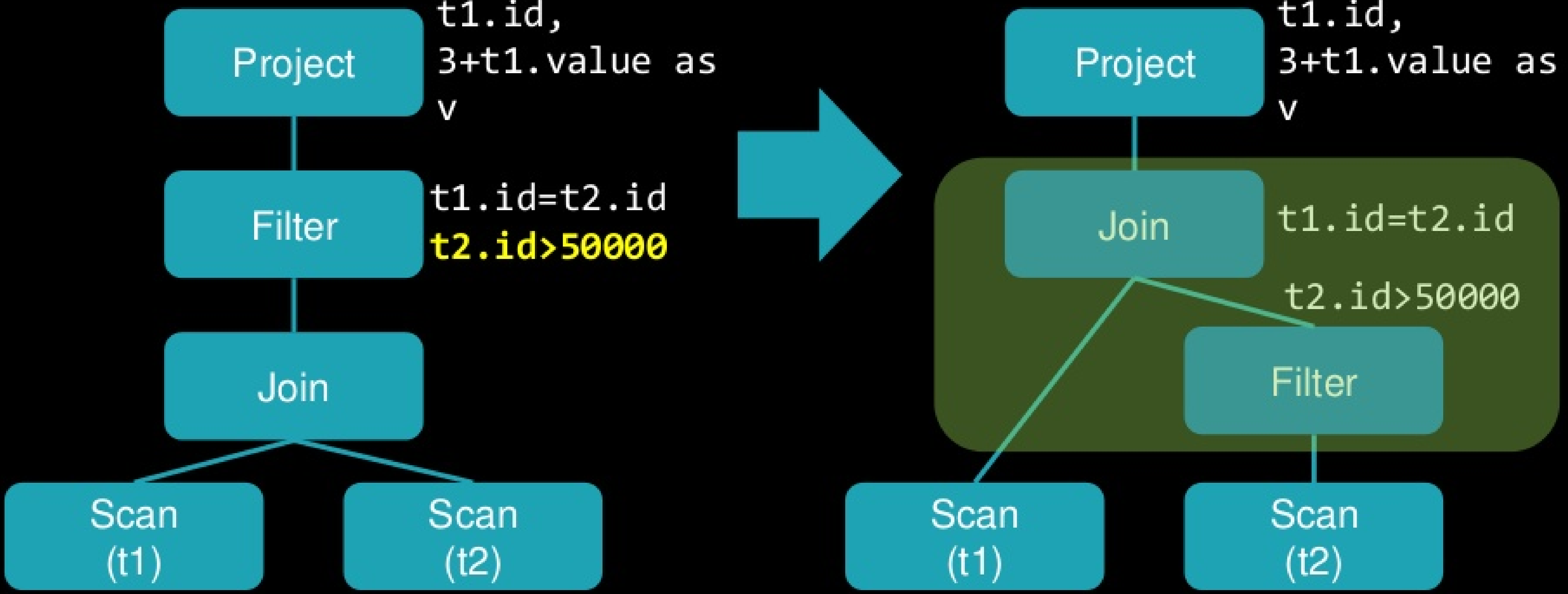
- A transformation is defined as a Partial Function
- Partial Function: A function that is defined for a subset of its possible arguments

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

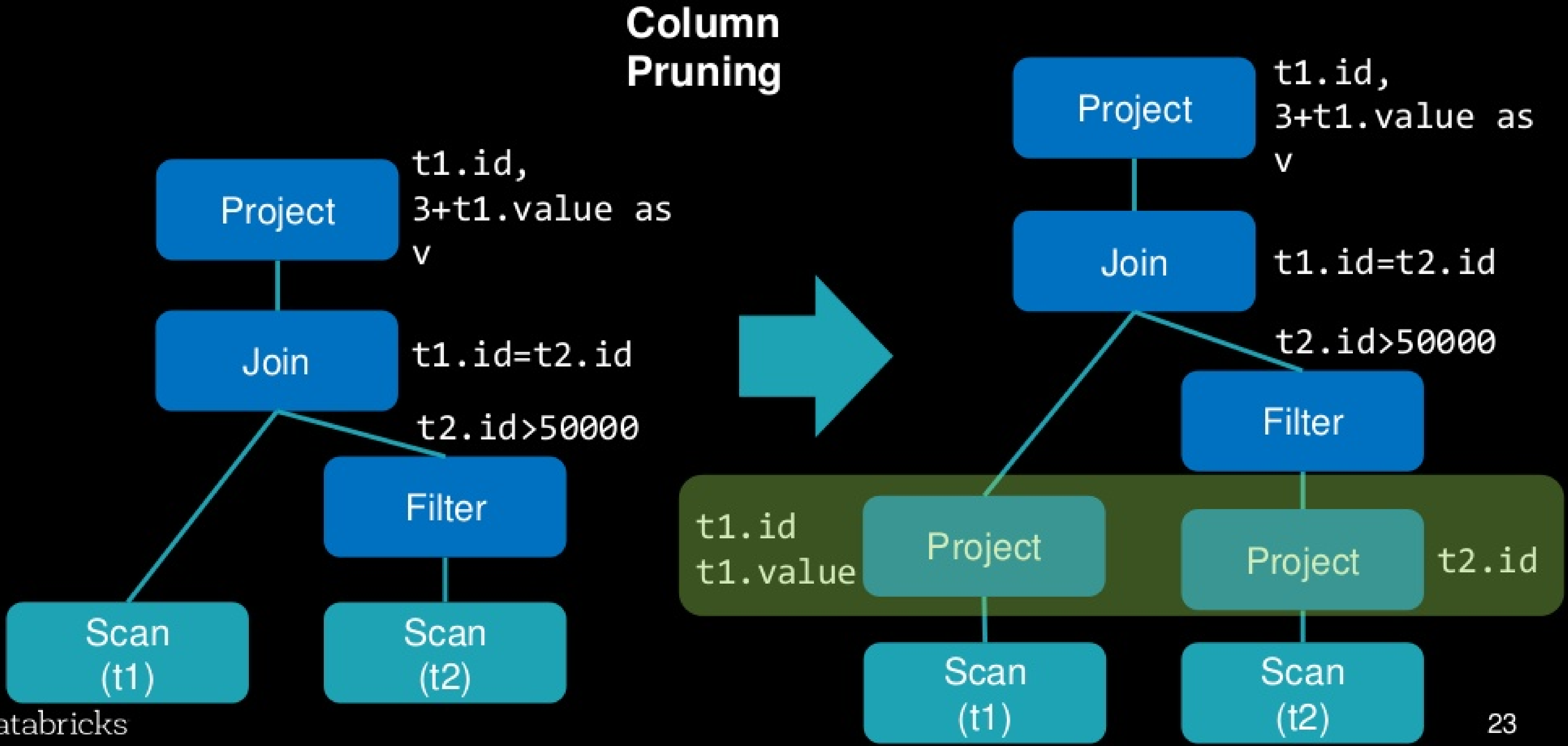
Case statement determines if the partial function is defined for a given input

Combining Multiple Rules

Predicate Pushdown



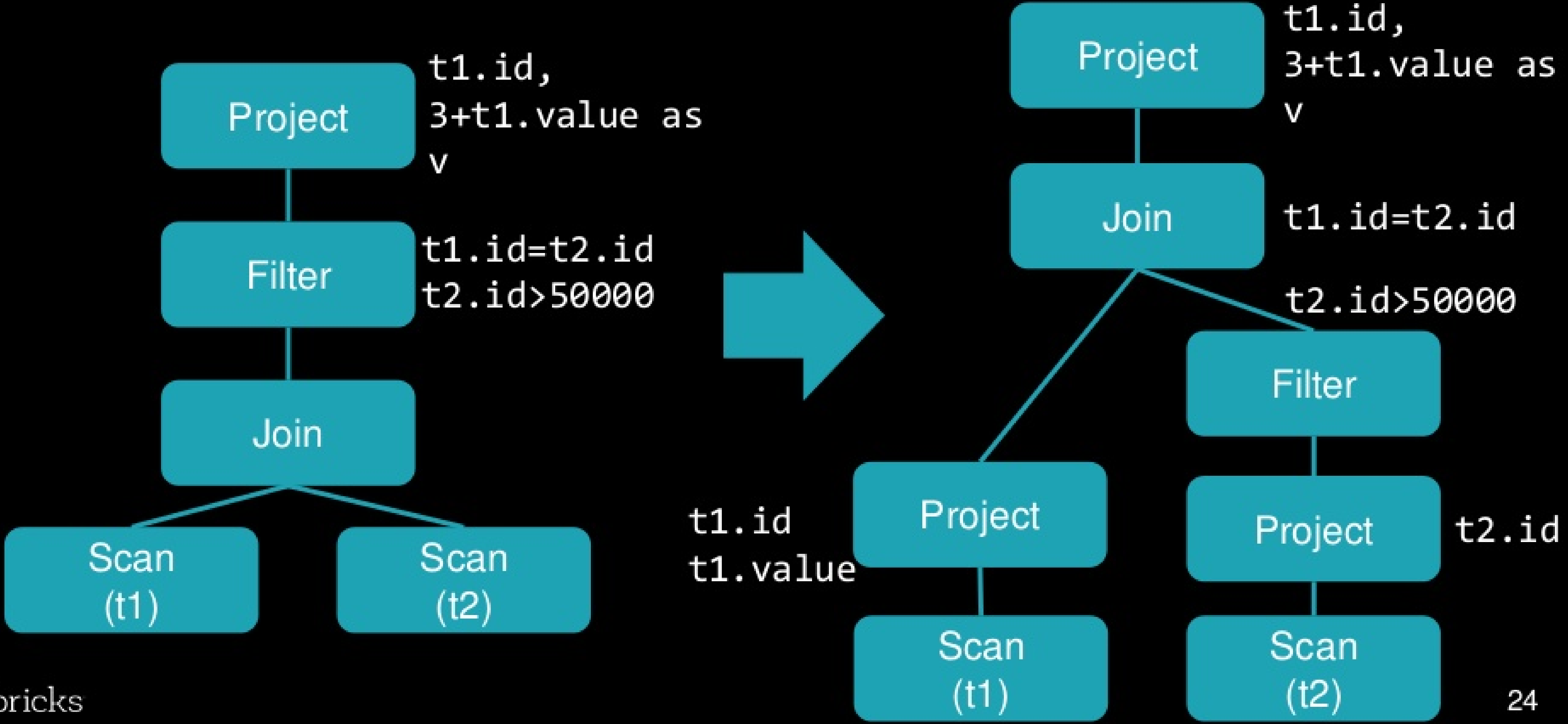
Combining Multiple Rules



Combining Multiple Rules

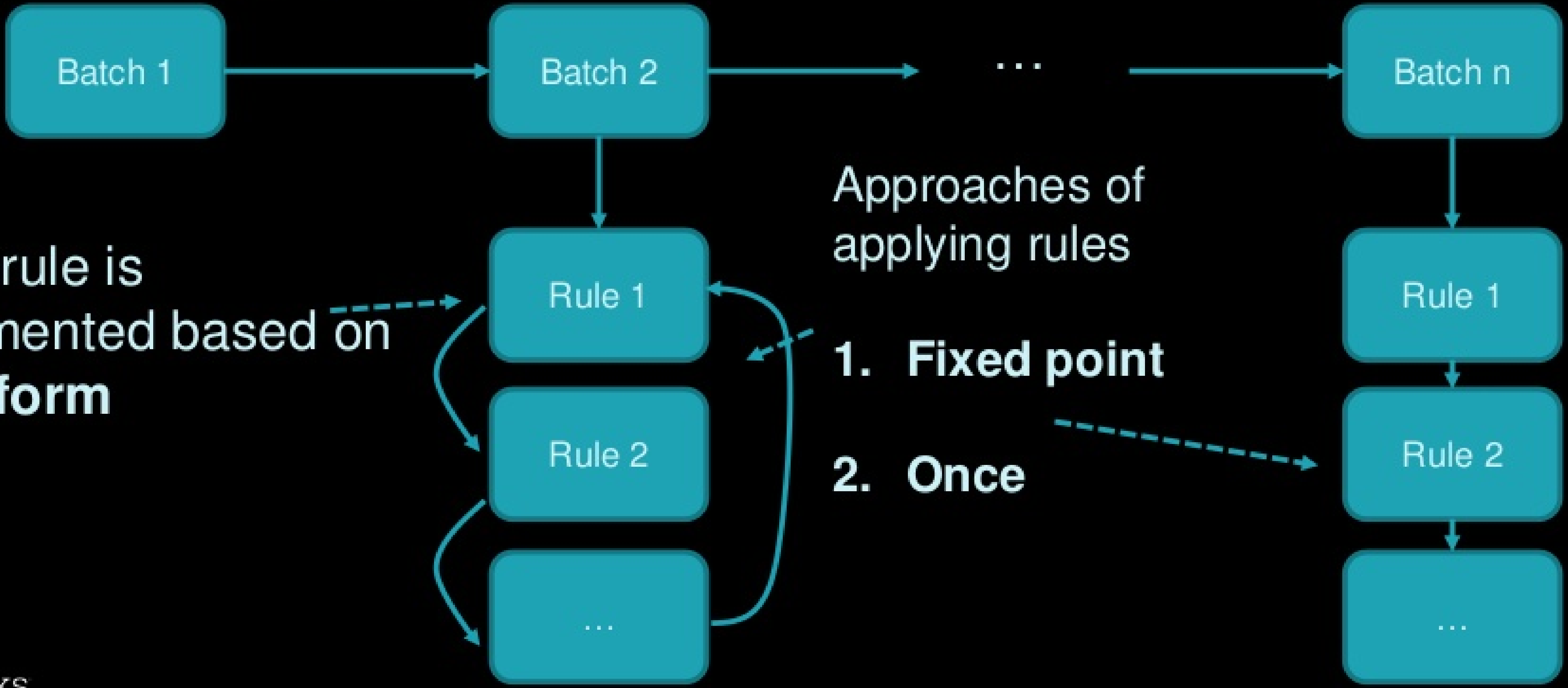
Before transformations

After transformations



Combining Multiple Rules: Rule Executor

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



Every rule is implemented based on **Transform**

Approaches of applying rules

1. **Fixed point**
2. **Once**

Transformations

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

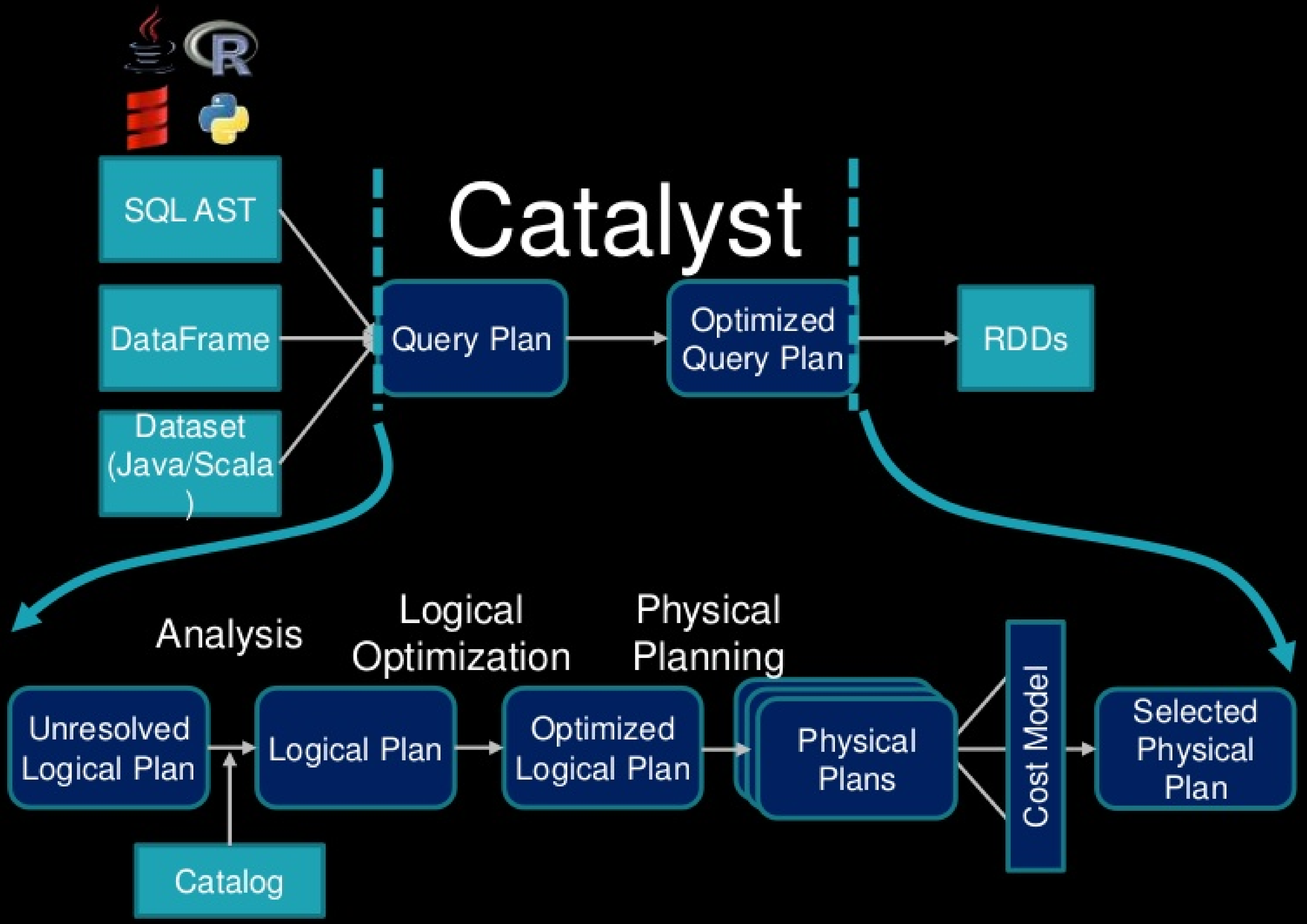
- Transforming a tree to another kind of tree
  - Logical Plan => Physical Plan

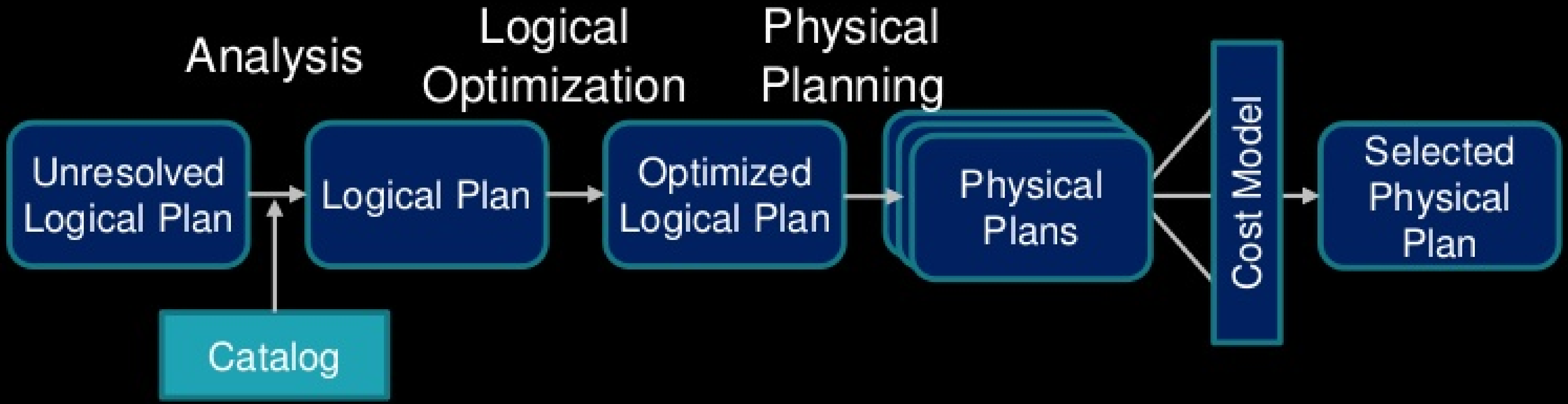
From Logical Plan to Physical Plan

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

```
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


Catalyst
SQL AST
DataFrame
Dataset (Java/Scala)
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
databricks



- **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)**:
  - Phase 1: Transforms an Optimized Logical Plan to a Physical Plan
  - Phase 2: Rule executor is used to adjust the physical plan to make it ready for execution

Put what we have learned in action

Use Catalyst's APIs to customize Spark

Roll your own planner rule

Roll your own Planner Rule

```
import org.apache.spark.sql.functions._

 // tableA is a dataset of integers in the ragne of [0, 19999999]
val tableA = spark.range(20000000).as('a)
// tableB is a dataset of integers in the ragne of [0, 9999999]
val tableB = spark.range(10000000).as('b)
// result shows the number of records after joining tableA and tableB
val result = tableA
  .join(tableB, $"a.id" === $"b.id")
  .groupBy()
  .count()
result.show()
```

This takes 4-8s on Databricks Community edition

Roll your own Planner Rule

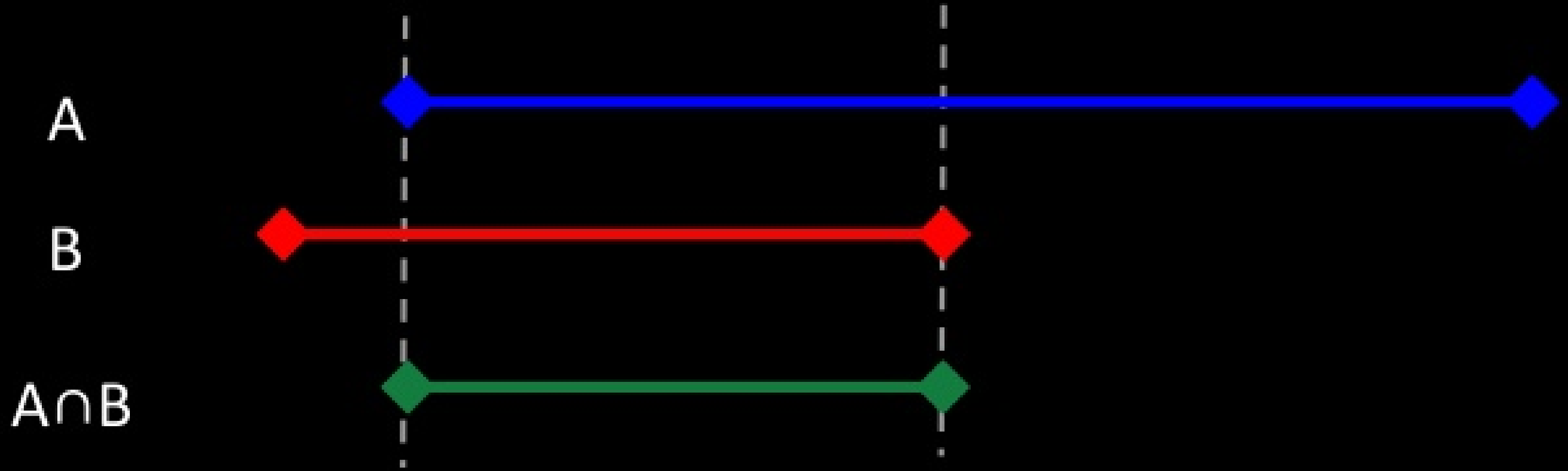
```
result.explain()
```

```
== Physical Plan ==
*HashAggregate(keys=[], functions=[count(1)])
+- Exchange SinglePartition
   +- *HashAggregate(keys=[], functions=[partial_count(1)])
      +- *Project
         +- *SortMergeJoin [id#642L], [id#646L], Inner
            :- *Sort [id#642L ASC NULLS FIRST], false, 0
            :  +- Exchange hashpartitioning(id#642L, 200)
            :     +- *Range (0, 20000000, step=1, splits=8)
            +- *Sort [id#646L ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(id#646L, 200)
                  +- *Range (0, 10000000, step=1, splits=8)
```

Roll your own Planner Rule

Exploit the structure of the problem

We are joining two intervals; the result will be the intersection of these intervals



Roll your own Planner Rule

```
// Import internal APIs of Catalyst
import org.apache.spark.sql.Strategy
import org.apache.spark.sql.catalyst.expressions.{Alias, EqualTo}
import org.apache.spark.sql.catalyst.plans.logical.{LogicalPlan, Join, Range}
import org.apache.spark.sql.catalyst.plans.Inner
import org.apache.spark.sql.execution.{ProjectExec, RangeExec, SparkPlan}

case object IntervalJoin extends Strategy with Serializable {
  def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
    case Join(
      Range(start1, end1, 1, part1, Seq(o1)), // mathces tableA
      Range(start2, end2, 1, part2, Seq(o2)), // matches tableB
      Inner, Some(EqualTo(e1, e2)))           // matches the Join
        if ((o1 semanticEquals e1) && (o2 semanticEquals e2)) ||
           ((o1 semanticEquals e2) && (o2 semanticEquals e1)) =>
        // See next page for rule body
    case _ => Nil
  }
}
```

Roll your own Planner Rule

```
// matches cases like:
// tableA: start1----------------------------end1
// tableB: ...------------------end2
if ((end2 >= start1) && (end2 <= end2)) {
  // start of the intersection
  val start = math.max(start1, start2)
  // end of the intersection
  val end = math.min(end1, end2)
  val part = math.max(part1.getOrElse(200), part2.getOrElse(200))
  // Create a new Range to represent the intersection
  val result = RangeExec(Range(start, end, 1, Some(part), o1 :: Nil))
  val twoColumns = ProjectExec(
    Alias(o1, o1.name)(exprId = o1.exprId) :: Nil,
    result)
  twoColumns :: Nil
} else {
  Nil
}
```

Roll your own Planner Rule

Hook it up with Spark

```
spark.experimental.extraStrategies = IntervalJoin :: Nil
```

Use it

```
result.show()
```

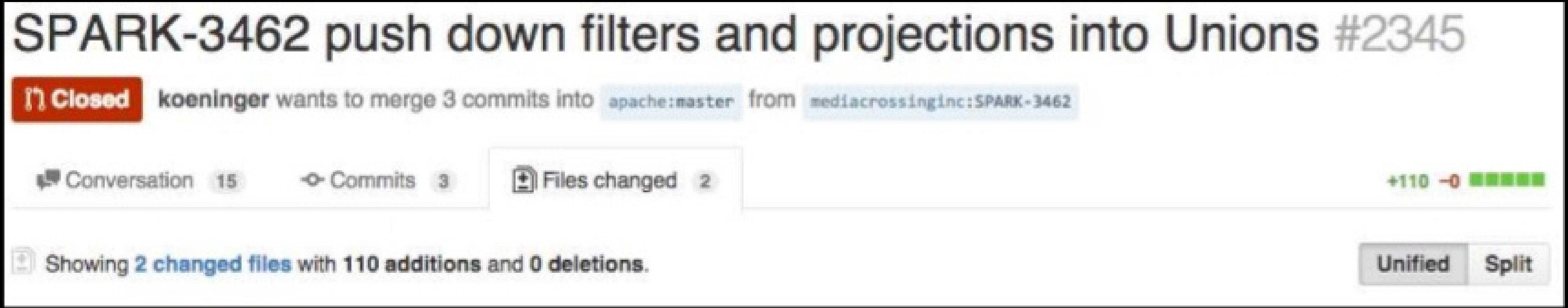
This now takes ~0.5s to complete

Roll your own Planner Rule

```
result.explain()


== Physical Plan ==
*HashAggregate(keys=[], functions=[count(1)])
+- Exchange SinglePartition
   +- *HashAggregate(keys=[], functions=[partial_count(1)])
      +- *Project
         +- *Project [id#642L AS id#642L]
            +- *Range (0, 10000000, step=1, splits=8)
```

Contribute your ideas to Spark



110 line patch took a user's query from "never finishing" to 200s.

Overall 200+ people have contributed to the analyzer/optimizer/planner in the last 2 years.

Try Apache Spark in Databricks!

UNIFIED ANALYTICS PLATFORM

- Collaborative cloud environment
- Free version (community edition)

DATABRICKS RUNTIME 3.0

- Apache Spark - optimized for the cloud
- Caching and optimization layer - DBIO
- Enterprise security - DBES

Try for free today.
databricks.com

Thank you!

What to chat?
Find me after this talk or at Databricks booth 3-3:40pm