UE Large Scale Processing @ ESIGELEC
2019/2020

12 - Spark SQL - Catalyst Optimzer

Abdel Dadouche
DJZ Consulting

adadouche@hotmail.com
@adadouche

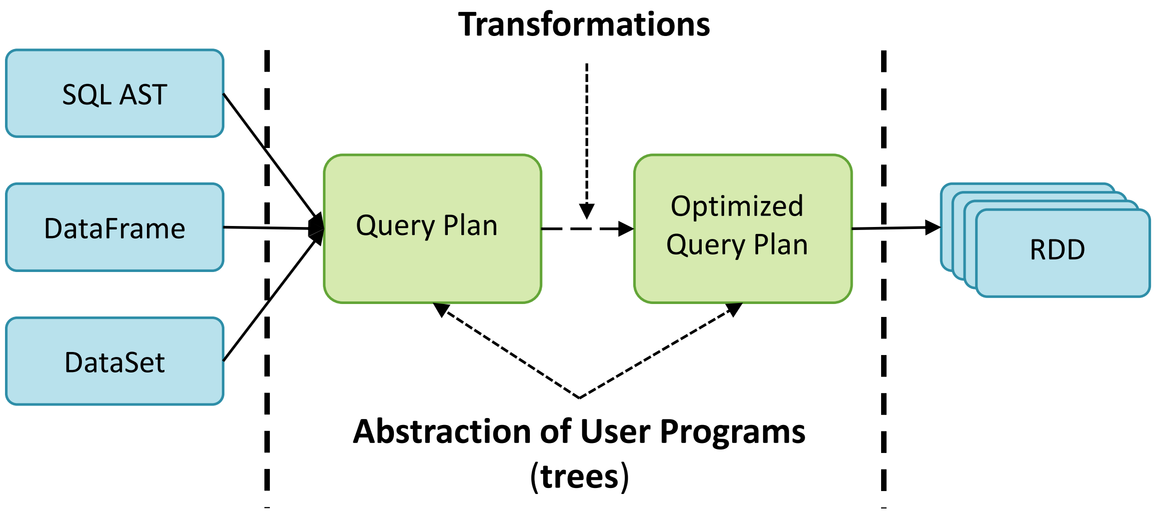
The Catalyst Optimizer

- Initially based on the functional programming constructs in Scala
- Designed around these two key purposes:
  - Easily add new optimization techniques and features to Spark SQL
  - Enable external developers to extend the optimizer
- Offers several public extension points, including external data sources and user-defined types.
- Supports both rule-based and cost-based optimization

https://databricks.com/glossary/catalyst-optimizer

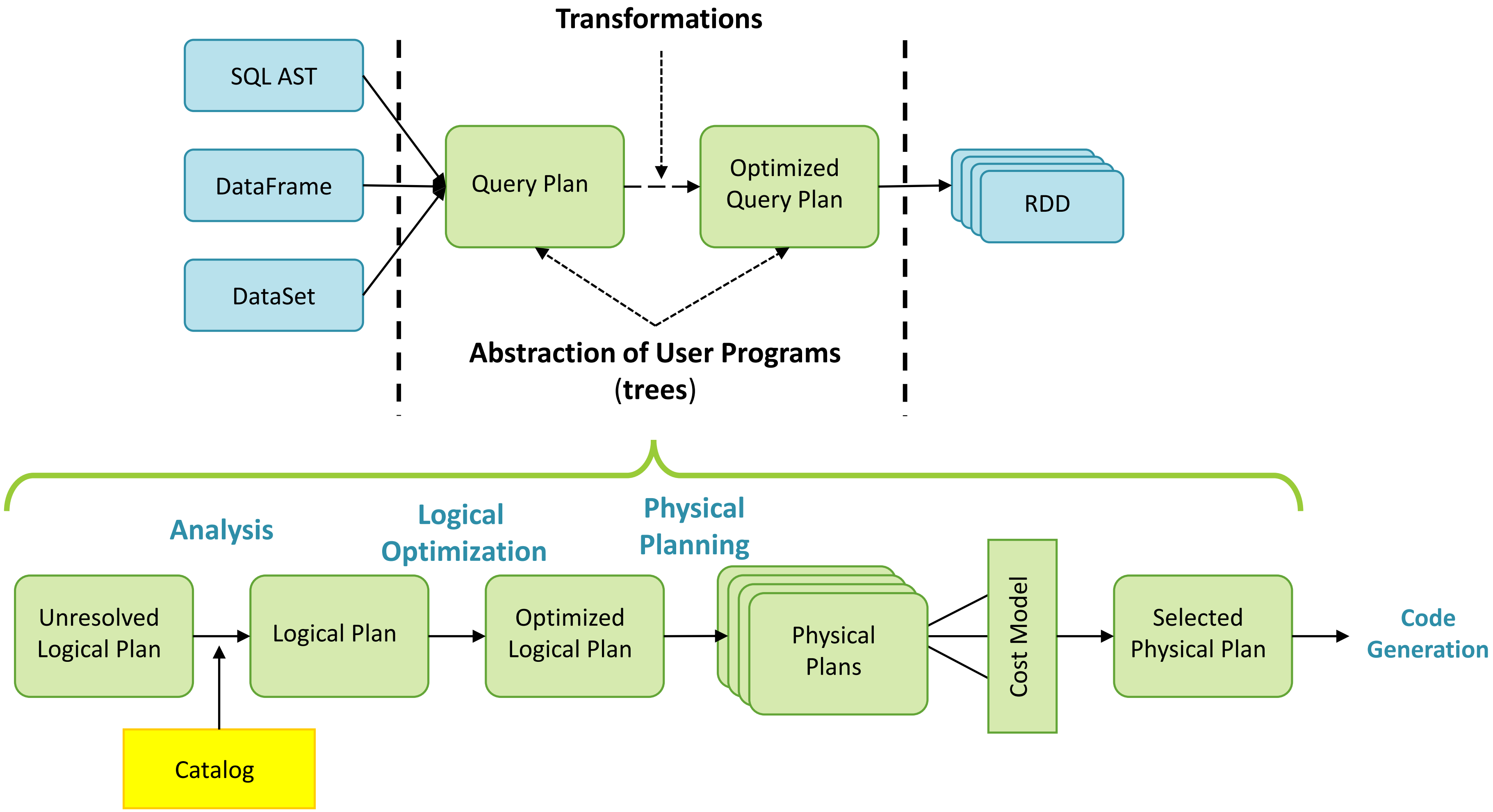
Overview of the Catalyst Optimizer

- Can process SQL AST, DataFrame or DataSet
- Apply a series of **transformations** on an initial "Query Plan" to find the "Optimized Query Plan"
- Uses a transformation framework based on a **tree** representation and **rules** for the manipulations
- The output is "code" that will build the RDD stack representing the optimized query plan

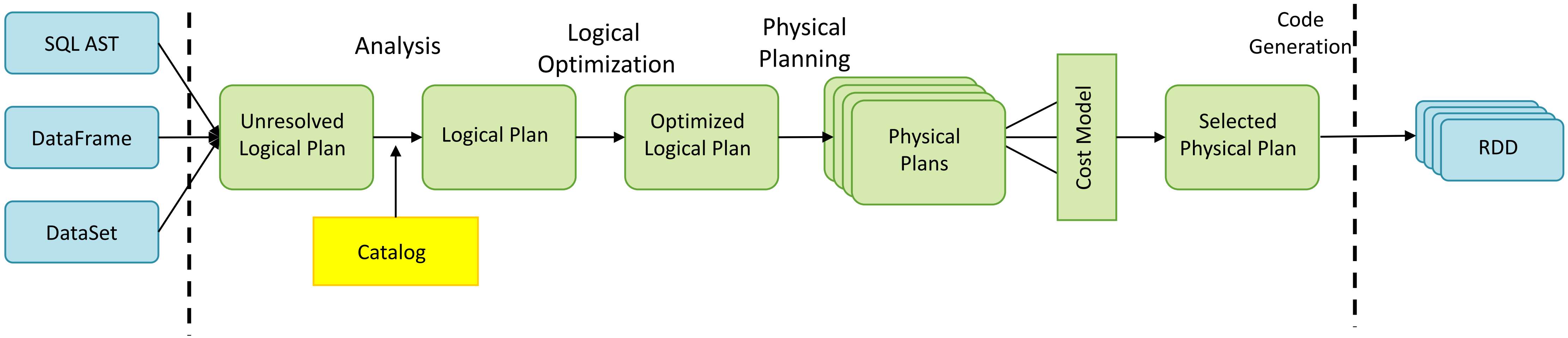


AST : abstract syntax tree
https://databricks.com/glossary/catalyst-optimizer

Catalyst Optimizer in more details



Catalyst Optimizer in more details



- The Catalyst Optimizer process can be decomposed in 4 **transformations** :
  - analyzing an unresolved logical plan to resolve references
  - logical plan optimization
  - physical planning
  - code generation

https://databricks.com/glossary/catalyst-optimizer

Transformations

A Query Example – Attributes & Expressions

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

Attributes

Represents a column of a dataset (t1.id) or a column generated by a specific data operation (the v column)

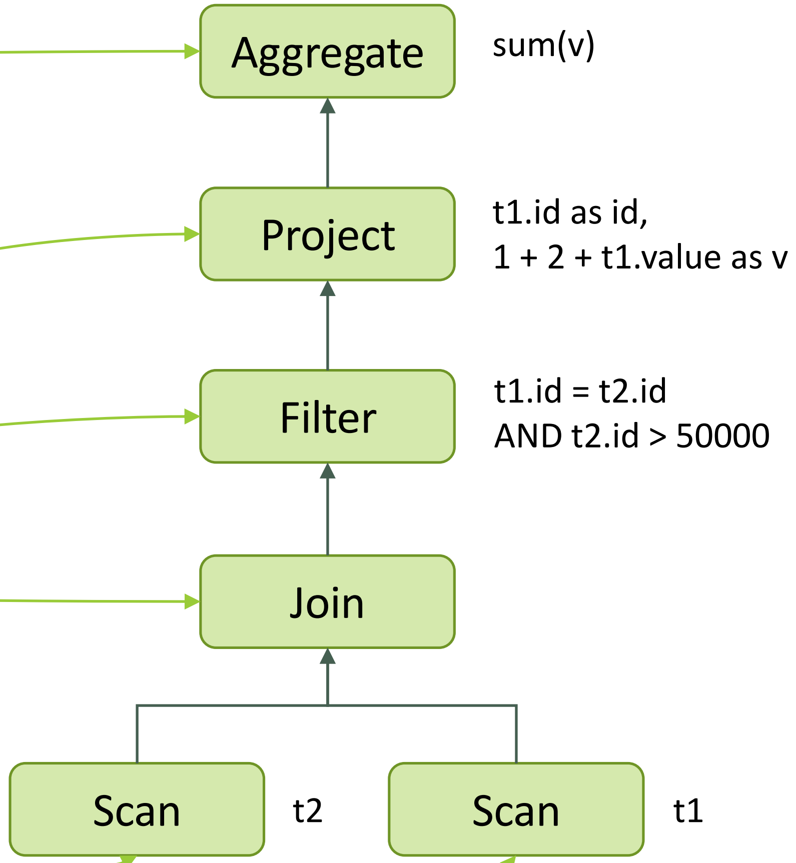
Expressions :

Represents a new value computed out of an input value (1 + 2 + t1.value)

Attributes & Expressions are represented as trees/nodes

A Query Example – Logical Plan

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



Transformations

- Use abstraction of user programs represented as trees to apply rules
- 2 types of trees transformations
  - Doesn’t change the tree type
    - Expression → Expression
    - Logical Plan → Logical Plan
    - Physical Plan → Physical Plan
  - Change the tree type
    - Logical Plan → Physical Plan

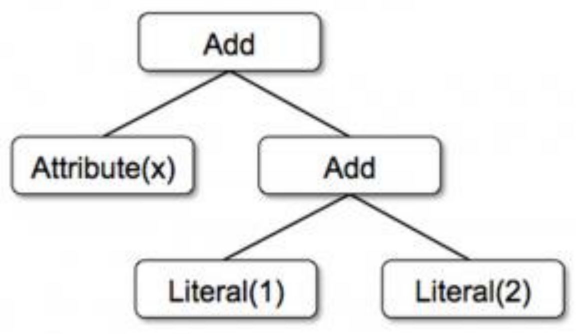
Transformations – Trees

- Trees are composed of “node” objects
- Each node has a node type/class
- Each node has zero or more “child” nodes
- New node types are defined in Scala as subclasses of the TreeNode class
- These objects are immutable and can only be manipulated using functional transformations

Example : x + (1 + 2)

- 3 node classes:
  - Literal(value: Int): a constant value
  - Attribute(name: String): an attribute from an input row, e.g.,"x"
  - Add(left: TreeNode, right: TreeNode): sum of two expressions
- In Scala code:

```
Add(Attribute(x), Add(Literal(1), Literal(2)))
```



Transformations – Rules / Transform

- Rules are used to manipulate Trees (a functions from a Tree to another Tree)
- While a rule can run any arbitrary code on its input tree, the most common approach is to use a set of pattern matching functions that find and replace subtrees with a specific structure
- Pattern matching allows extracting values from potentially nested structures.
- In Catalyst, trees offer a **transform** method that applies a pattern matching function recursively on all nodes of the tree, transforming the ones that match each pattern to a result.

Example : x+(1+2)

- implement a rule that folds Add operations between constants :

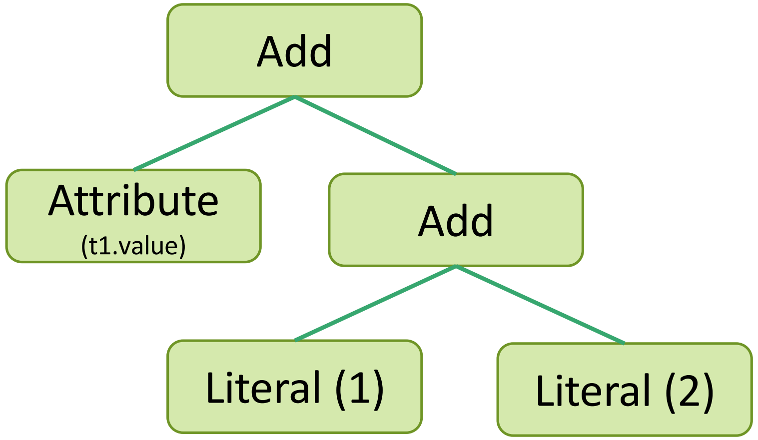
```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
```

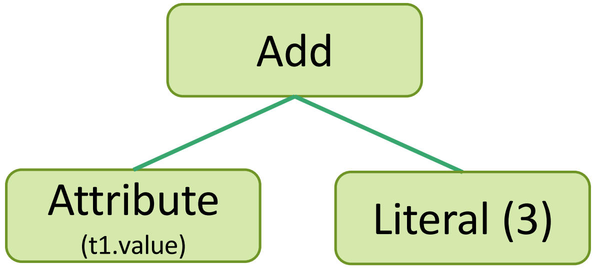
- Rules can match multiple patterns in the same transform call, making it very concise to implement multiple transformations at once:

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
  case Add(left, Literal(0)) => left
  case Add(Literal(0), right) => right
}
```

Transform – Single Rule Example

$x + (1 + 2)$ ➔ $x + 3$

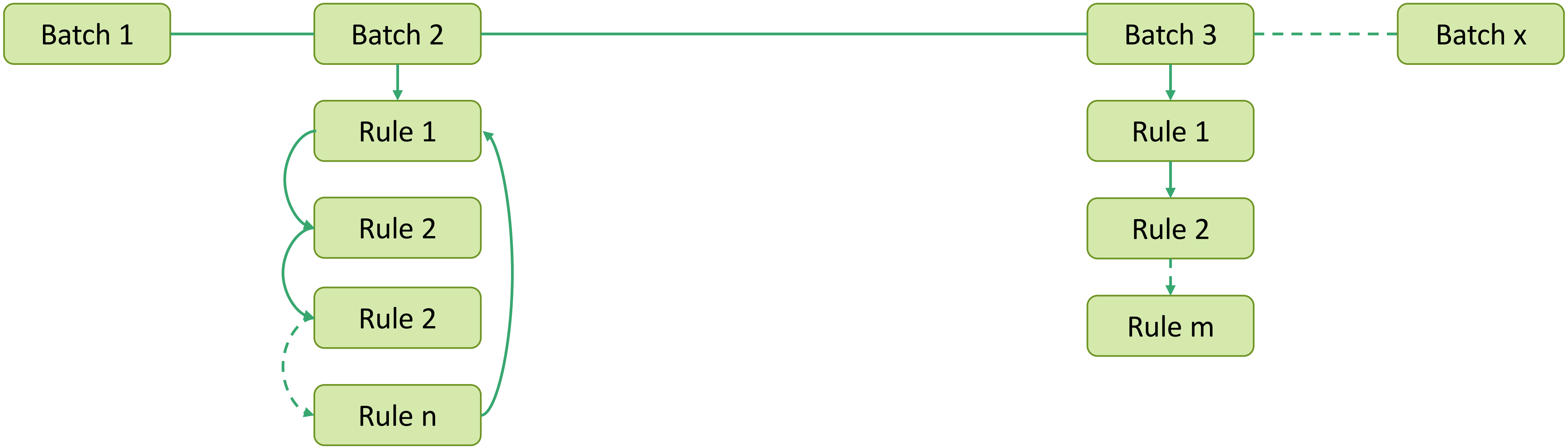




Will evaluate 1 + 2 for every row ➔ waste of calculation resource

Will evaluate 1 + 2 once ➔ more efficient

Transform – Multiple Rule Example – Batch



Fixed point batch

Rules are applied until the plan doesn't change anymore (converge)

Executed once

Rules are applied only once.

Like doing some validation or simple manipulation like removing nodes

Catalyst Optimizer Phase

Analysis

Catalyst Optimizer Phase – Analysis

- **Transform** an "**unresolved logical plan**" (tree build on an input DataFrame, DataSet or SQL AST) with unresolved attributes references and data types into a "**resolved logical plan**"

- Applies rules for:
  - Looking up relations by name from the catalog
  - Mapping named attributes to the input provided given operator's children.
  - Determining which attributes refer to the same value to give them a unique ID
  - Propagating and coercing types through expressions:
    - for example, we cannot know the return type of 1 + col until we have resolved col and possibly casted its subexpressions to a compatible types.

- An attribute is called unresolved if we do not know its type or have not matched it to an input table (or an alias) ➔ will fail into an error and exit

- Catalyst rules / transforms and Catalog object are used to track the tables in all data sources and resolve attributes

A Query Example – Logical Plan

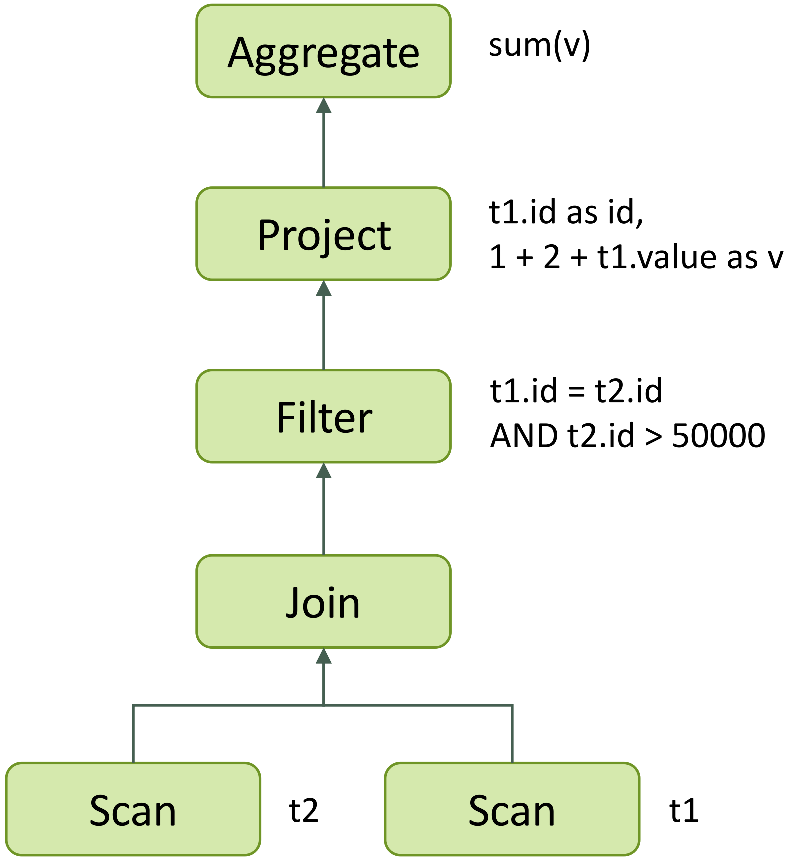
- A Logical Plan describes a series of computation on datasets without defining the execution details

- It includes / provides:

  - outputs : a list of attributes generated by the logical plan (e.g. [sum] or [id, v])

  - constraints: a set of invariants about the result generated by the plan (e.g. t2.id > 50000) not to be confused with conditions like t1.id = t2.id

  - statistics: size of the plan in row/bytes + per columns stats (min/max/nulls/number of distinct values)

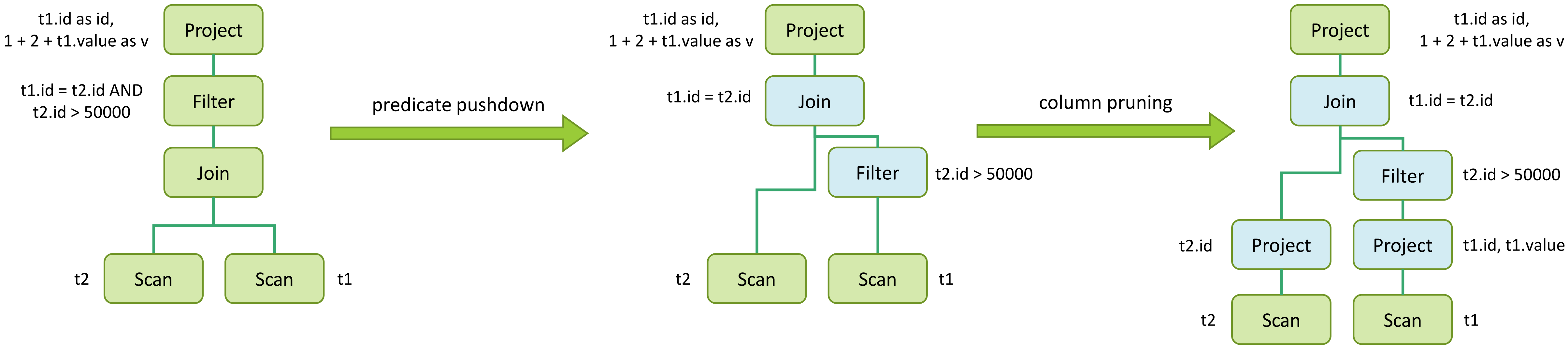


Catalyst Optimizer Phase

Logical Optimization

Catalyst Optimizer Phase (2/4) – Logical Optimization

- Applies standard rule-based optimizations to a logical plan
- Cost-based optimization is performed by generating multiple plans using rules, and then computing their costs
- These optimizations include:
  - constant folding
  - predicate pushdown
  - projection pruning
  - null propagation
  - boolean expression simplification
  - and many other rules…



Catalyst Optimizer Phase

Physical Planning

Catalyst Optimizer Phase (3/4) – Physical Planning

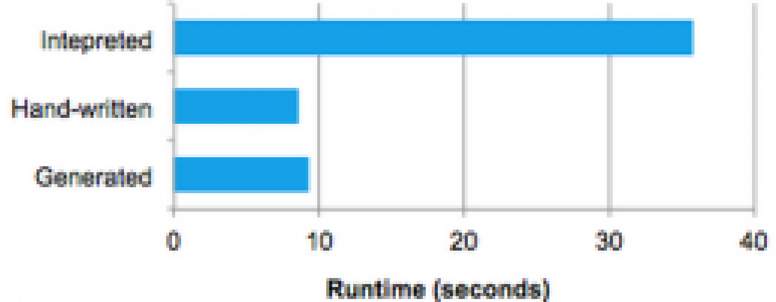
- Use a physical planner in phases:
  - Uses Strategies with pattern matching to generate one or more Physical Plans from an Optimized Logical Plan
  - Then, apply Rules to make the Physical Plans ready for execution:
    - Prepare Scalar sub-queries
    - Ensure requirements on input rows (partitioning for example)
    - Apply physical optimizations (rule-based, such as pipelining projections or filters)
- Then, selects a plan using a cost model

Catalyst Optimizer Phase

Code Generation

Catalyst Optimizer Phase (4/4) – Code Generation

- The goal is to generates Java bytecode to run on each Spark node
- Initially, relied on a Scala feature: "quasiquotes"
  - allow the construction of abstract syntax trees (ASTs) in the Scala language
  - can then be fed to the Scala compiler at runtime to generate bytecode
- Now uses other framework named Janino
- Code with performance comparison:



- Converts an expression like (x+y) + 1 to:

```
def compile (node: Node) : AST = node match {
  case Litteral (value) => q"$value"
  case Attribute (name) => q"row.get($name)"
  case Add(left, right) => q"{compile($left)} + {compile($right)}"
}
```

- 100+ native functions with optimized code generation implementations:
  - String : concat, format_string, lower, lpad
  - Data/Time : current_timestamp, date_format, date_add
  - Math : sqrt, randn