# Apache Spark SQL Optimization

Surendra Panpaliya


# Agenda

- Catalyst Optimizer
- What optimizations does catalyst do?
- "Explaining" Catalyst optimizations
- Overview of Explain output
- Optimization: Predicate Pushdown

# Agenda

- Tungsten Optimizer
- Tungsten Overview
- Tungsten's Binary Format

# What is Catalyst?

- Spark SQL was designed
- with an optimizer
- called Catalyst based on
- the functional programming of Scala.

# What is Catalyst?

Main purposes are:

to add new optimization techniques

to solve some problems

with “big data”

# What is Catalyst?

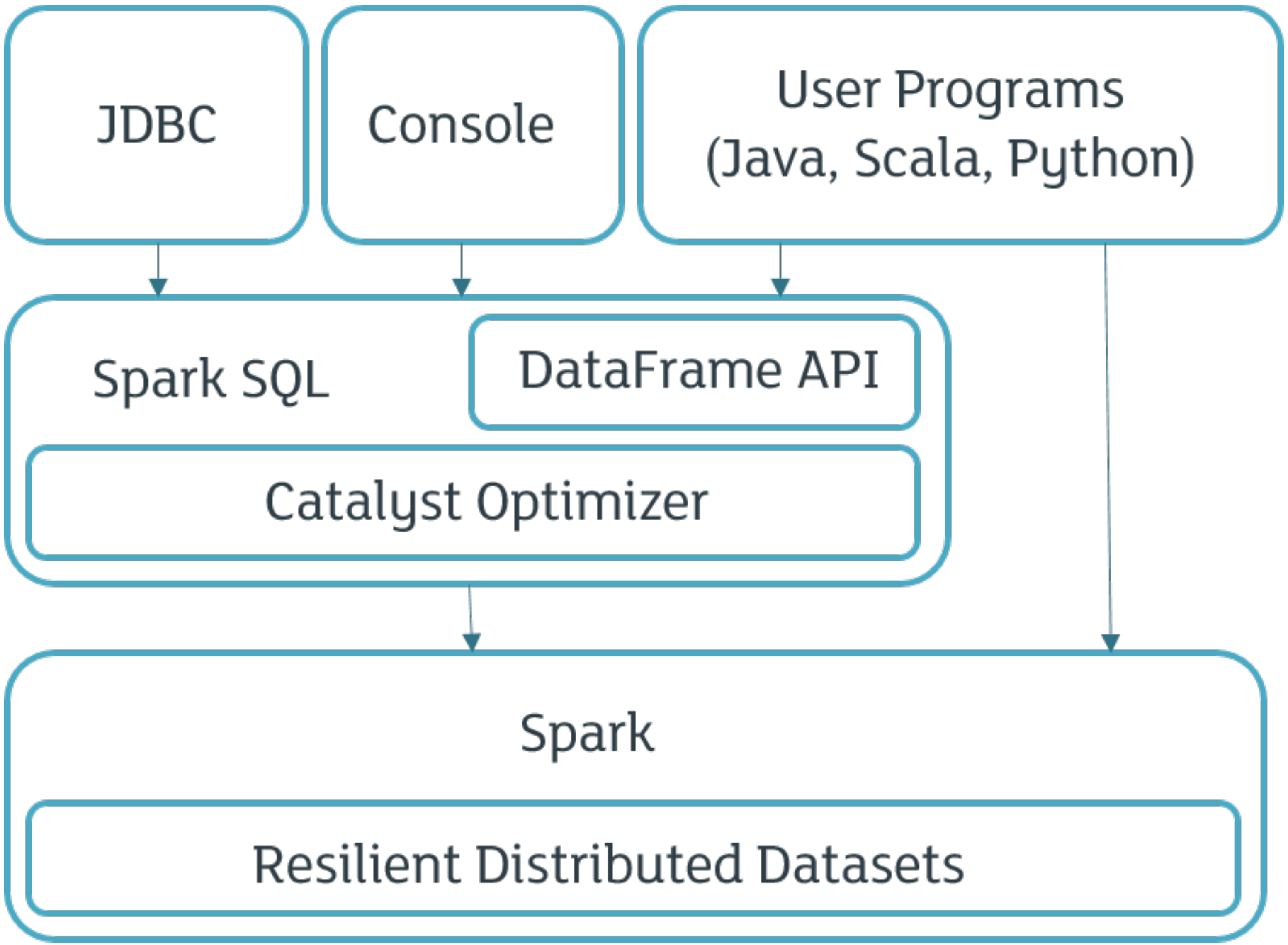
Main purposes are:

2. to allow developers

to expand and

customize the functions

of the optimizer.




Catalyst Spark SQL architecture and Catalyst optimizer integration

# Catalyst components

Trees

The main data type in Catalyst is the tree.

Each tree is composed of nodes, and

each node has a node type and

zero or more children.

# Catalyst components

- Trees
- Objects are immutable and
- can be manipulated
- with functional language.

# Catalyst components

Trees

As an example

Merge(Attribute(x), Merge(Literal(1), Literal(2))

# Catalyst components

Literal(value: Int): a constant value

Attribute(name: String): an attribute as input row

Merge(left: TreeNode, right: TreeNode):

mix of two expressions

# Rules

Trees can be manipulated

using rules,

which are functions of

a tree to another tree.

# Rules

The transformation method applies

the pattern matching function

recursively

on all nodes of

the tree transforming

each pattern to the result.

# Rules

Example of a rule applied to a tree.

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

# Using Catalyst in Spark SQL

The Catalyst Optimizer in Spark offers

rule-based and cost-based optimization.

Rule-based optimization indicates

how to execute the query

from a set of defined rules.

# Using Catalyst in Spark SQL

Cost-based optimization

generates multiple execution plans and

compares them to choose the lowest cost one.

# The Catalyst optimizer Example

- case class EmployeeClass(empid: String, empname: String, salary: Int)

- val employeeDataset  = Seq(
- EmployeeClass("001", "Surendra", 4000000),
- EmployeeClass("002", "Satish", 5000000)).toDS

- employeeDataset.show()

# The Catalyst optimizer Example

- val equery = employeeDataset.groupBy("salary").count().as("total")
- equery.show()
- equery.explain(extended = true)

# Phases

1. Analysis
2. Logic Optimization Plan
3. Physical plan
4. Code generation

# 1. Analysis

The first phase of Spark SQL

optimization is the analysis.

Spark SQL starts

with a relationship

to be processed

that can be in two ways.

# 1. Analysis

A serious form from an

AST (abstract syntax tree)

returned by an SQL parser

DataFrame object

of the Spark SQL API.

# 2. Logic Optimization Plan

- The second phase is the logical optimization plan.
- In this phase, rule-based optimization is applied
- to the logical plan.
- It is possible to easily add new rules.

# 3. Physical plan

Spark SQL takes the logical plan and

generates one or more physical plans

using the physical operators

that match the Spark execution engine.

# 3. Physical plan

The plan to be executed is

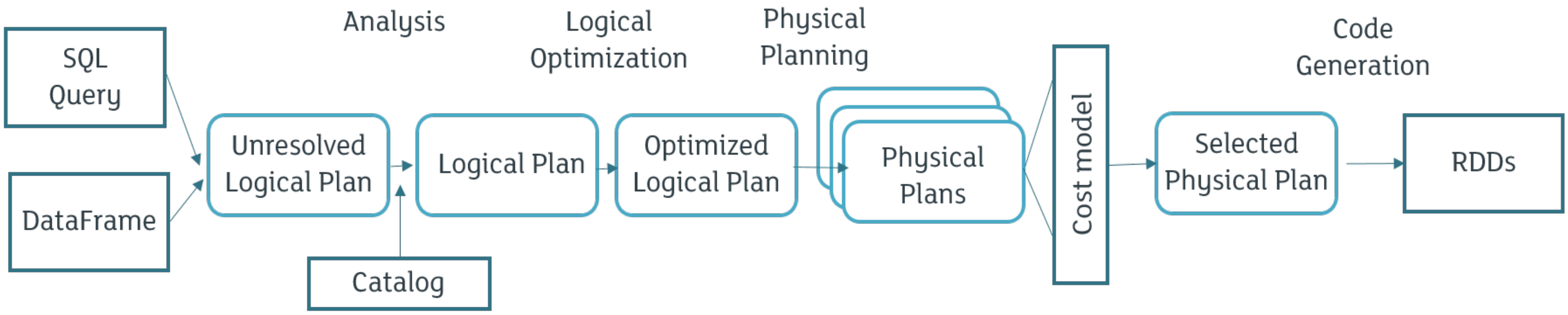
selected using the cost-based model

(comparison between model costs).

# 4. Code generation

- Code generation is the final phase of
- optimizing Spark SQL.
- To run on each machine,
- it is necessary to generate
- Java code bytecode.

# *Phases of the query plan in Spark SQL. Rounded squares represent the Catalyst trees*



# What is Optimization in Apache Spark?

- Optimization is
- the method by which the long running
- application or system is fine tuned and
- made some change to make application
- to manage resources effectively and
- reduce the processing time efficiently.

# What is Optimization in Apache Spark?

- Apache Spark 2.0 is a major release
- that brought drastic changes
- to many  framework,
- API's and libraries in that framework.
- In earlier version of Apache Spark,
- for optimization, developers need
- to change the source code of spark framework.

# What is Optimization in Apache Spark?

- This solved the problem
- for the particular user or
- to the particular account and
- also it is not advisable
- to change the code base of framework.

# What is Optimization in Apache Spark?

- There comes an idea
- to enhance the optimization features
- in later version that will act as
- an catalyst
- to run the application efficiently.

# What is Optimization in Apache Spark?

- In advanced Apache spark framework,
- we have a pluggable method
- which helps one
- to define a set of optimization rules and
- add it to the Catalyst.

# What is Optimization in Apache Spark?

- “The term **optimization** refers
- to a process in which
- a system is modified
- in such a way that
- it work more efficiently or
- it uses fewer resources.”

# What is Optimization in Apache Spark?

- **Spark SQL** is the most
- technically involved component of Apache Spark.
- Spark SQL deals with both
- SQL queries and DataFrame API.
- In the depth of Spark SQL
- there lies a **catalyst optimizer**.

# What is Optimization in Apache Spark?

- Catalyst optimization allows
- some advanced programming language
- features that allow
- you to build an extensible query optimizer.

# What is Optimization in Apache Spark?

- A new extensible optimizer called Catalyst
- emerged to implement Spark SQL.
- This optimizer is based on
- functional programming
- construct in **Scala**.

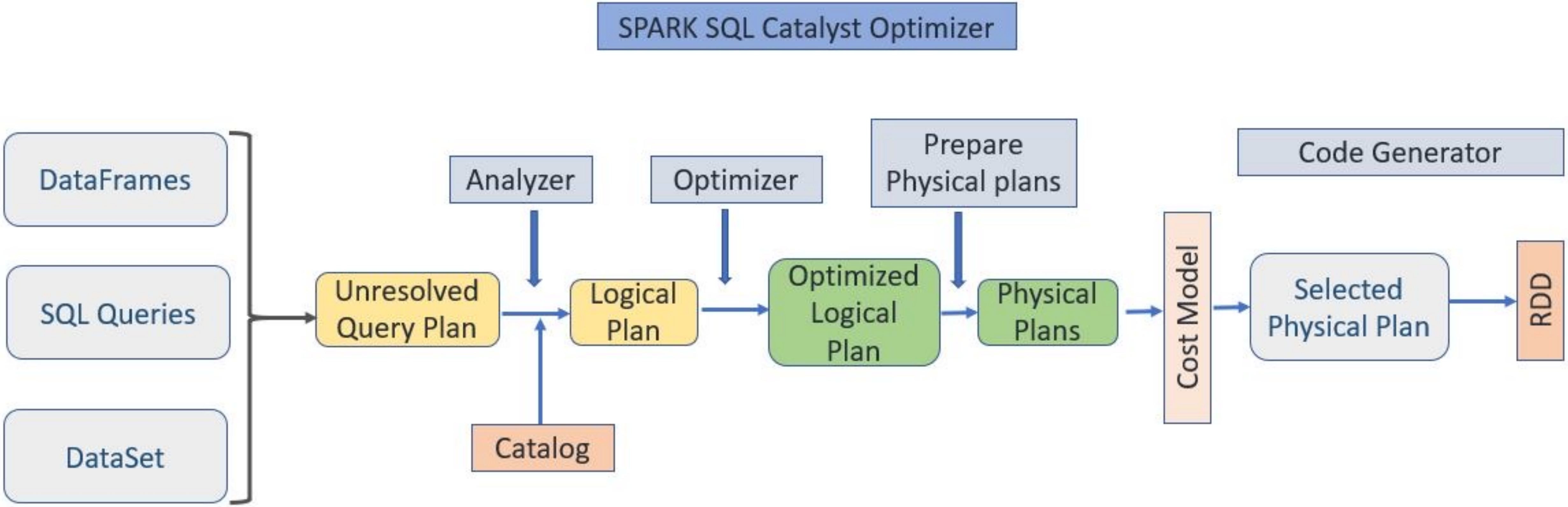
# What is Optimization in Apache Spark?

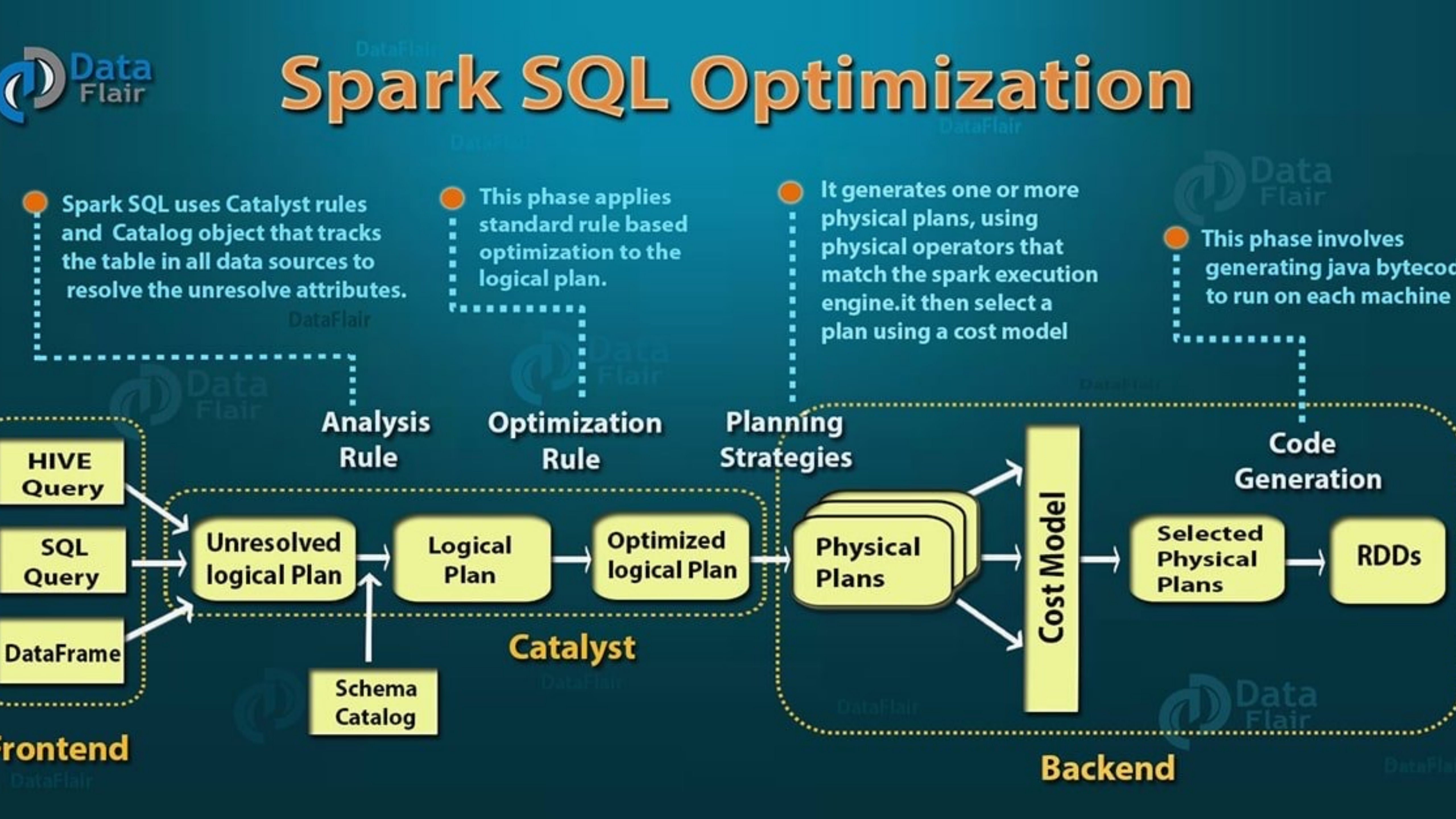
- Catalyst Optimizer supports both
- **rule-based** and **cost-based** optimization.
- In *rule-based optimization*
- the rule based optimizer
- use set of rule to determine
- how to execute the query.

# What is Optimization in Apache Spark?

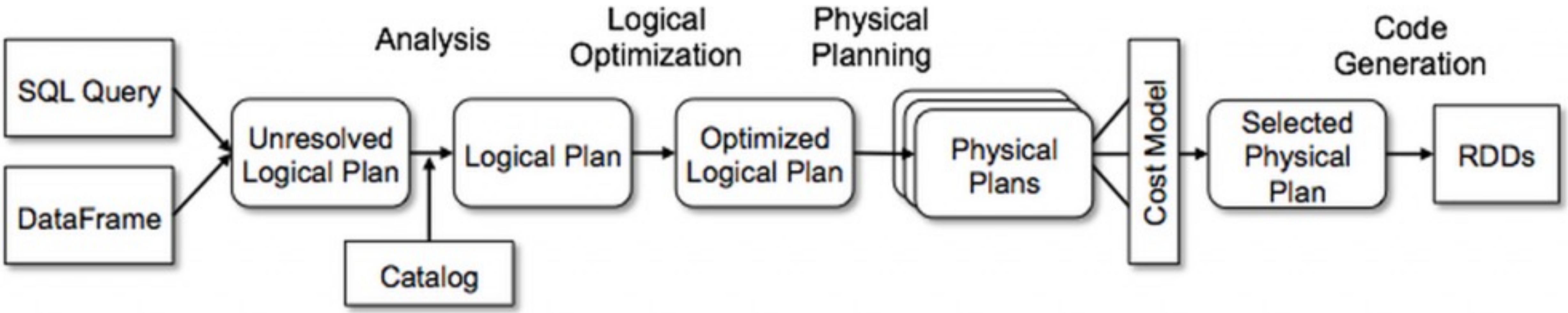
- While the *cost based optimization*
- finds the most suitable way
- to carry out SQL statement.
- In cost-based optimization,
- multiple plans are generated using
- rules and then their cost is computed.

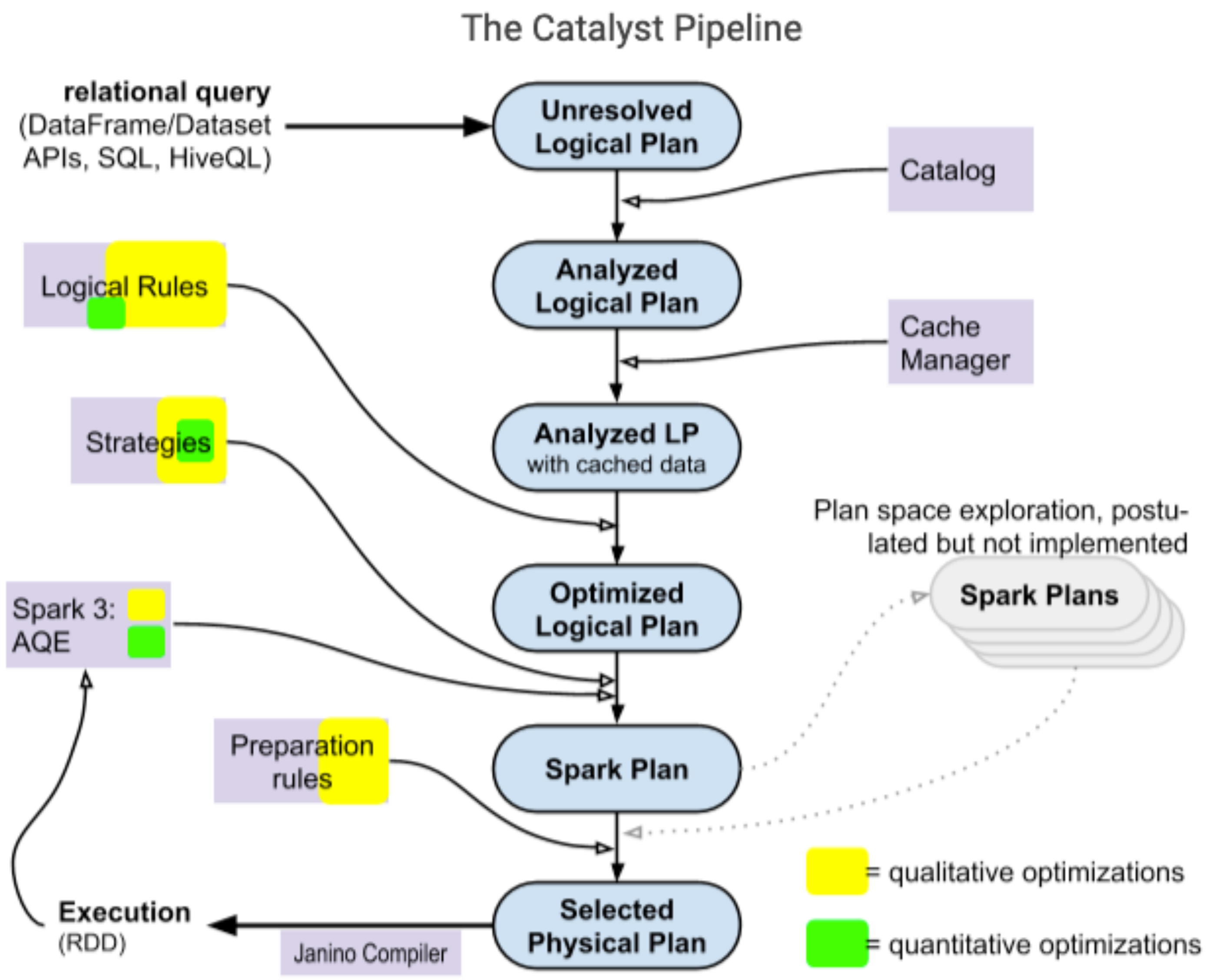
# Execution Model





Spark SQL Optimization
Spark SQL uses Catalyst rules and Catalog object that tracks the table in all data sources to resolve the unresolve attributes.
This phase applies standard rule based optimization to the logical plan.
It generates one or more physical plans, using physical operators that match the spark execution engine.it then select a plan using a cost model
This phase involves generating java bytecod to run on each machine
Analysis Rule
Optimization Rule
Planning Strategies
Code Generation
HIVE Query
SQL Query
DataFrame
Unresolved logical Plan
Logical Plan
Optimized logical Plan
Physical Plans
Cost Model
Selected Physical Plans
RDDs
Schema Catalog
Catalyst
rontend
Backend


SQL Query
DataFrame
Analysis
Unresolved Logical Plan
Catalog
Logical Plan
Logical Optimization
Optimized Logical Plan
Physical Planning
Physical Plans
Cost Model
Selected Physical Plan
Code Generation
RDDs


The Catalyst Pipeline
relational query
(DataFrame/Dataset APIs, SQL, HiveQL)
Unresolved Logical Plan
Catalog
Analyzed Logical Plan
Cache Manager
Logical Rules
Analyzed LP
with cached data
Strategies
Plan space exploration, postu-lated but not implemented
Optimized Logical Plan
Spark Plans
Spark 3: AQE
Spark Plan
Preparation rules
Selected Physical Plan
Execution
(RDD)
Janino Compiler
= qualitative optimizations
= quantitative optimizations

# What is the need of Catalyst Optimizer?

- There are two purposes behind Catalyst's extensible design:
- We want to add the easy solution to tackle various problems with **Bigdata** like a problem with semi-structured data and advanced data analytics.
- We want an easy way such that external developers can extend the optimizer.

# Fundamentals of Catalyst Optimizer

- Catalyst optimizer
- makes use of standard
- features of Scala programming like
- pattern matching.
- Catalyst contains the **tree** and
- the set of **rules**
- to manipulate the tree.

# Fundamentals of Catalyst Optimizer

- There are specific libraries
- to process relational queries.
- There are various rule sets
- which handle different
- phases of query execution

# Fundamentals of Catalyst Optimizer

- Like *analysis*, *query optimization,*
- *physical planning,* and
- *code generation*
- to compile parts of queries
- to Java bytecode.

# 4.1. Trees

- A tree is the main data type in the catalyst.
- A tree contains node object.
- For each node, there is a node.
- A node can have one or more children.

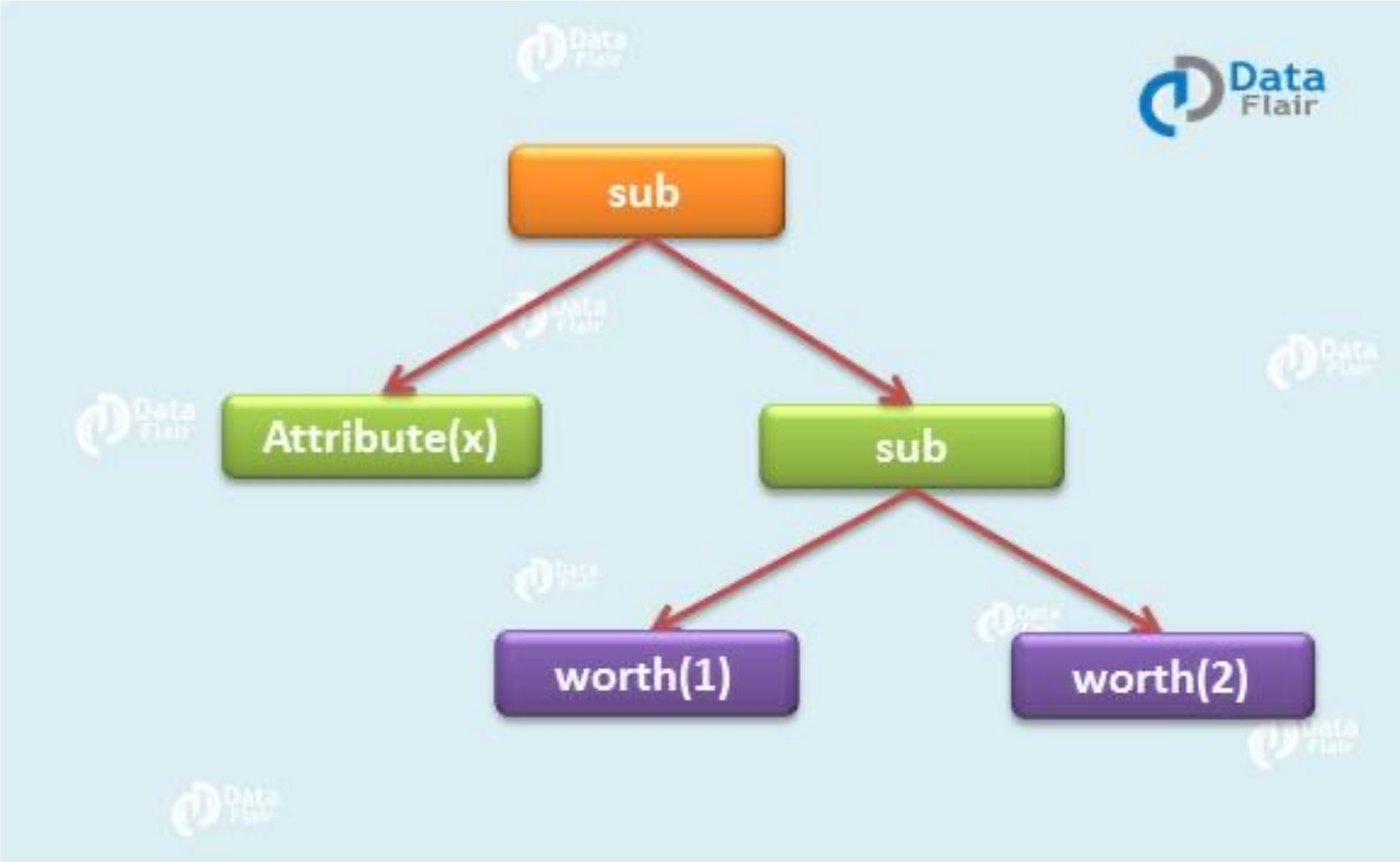
# 4.1. Trees

- New nodes are defined as
- subclasses of TreeNode class.
- These objects are immutable in nature.
- The objects can be manipulated
- using functional transformation.

# Trees

- For example, if we have three node classes:
- **worth**, **attribute**, and **sub** in which-
- worth(value: Int): a constant value
- attribute(name: String)
- sub (left: TreeNode, right: TreeNode):
- subtraction of two expressions.

# Trees



# 4.2. Rules

- We can manipulate tree using **rules**.
- We can define rules as a function from one tree to another tree.
- With rule we can run arbitrary code on input tree,
- the common approach
- to use a pattern matching function and
- replace subtree with a specific structure.

# 4.2. Rules

- In a tree with the help of
- **transform function**,
- we can recursively apply
- pattern matching on
- all the node of a tree.

# 4.2. Rules

- We get the pattern that
- matches each pattern
- to a result.

# 4.2. Rules

- For example:

*tree.transform {case Sub(worth(c1),worth(c2)) => worth(c1+c2) }*

# 4.2. Rules

- The expression that
- is passed during
- pattern matching
- to transform is a partial function.

# 4.2. Rules

- By partial function,
- it means it only needs to match
- to a subset of all possible input trees.

# 4.2. Rules

- Catalyst will see, to which part of a tree
- the given rule applies, and
- will automatically skip over
- the tree that does not match.
- With the same transform call,
- the rule can match multiple patterns.

# 4.2. Rules

- For example-
- *tree.transform {*
  *case Sub(worth(c1), worth(c2)) =>worth(c1-c2)*
  *case Sub(left , worth(0)) => left*
  *case Sub(worth(0), right) => right*
  *}*

# 4.2. Rules

- To fully transform a tree,
- rule may be needed
- to execute multiple time.

# 4.2. Rules

- Catalyst work by grouping rules
- into batches and
- these batches are executed
- until a fixed point is achieved.

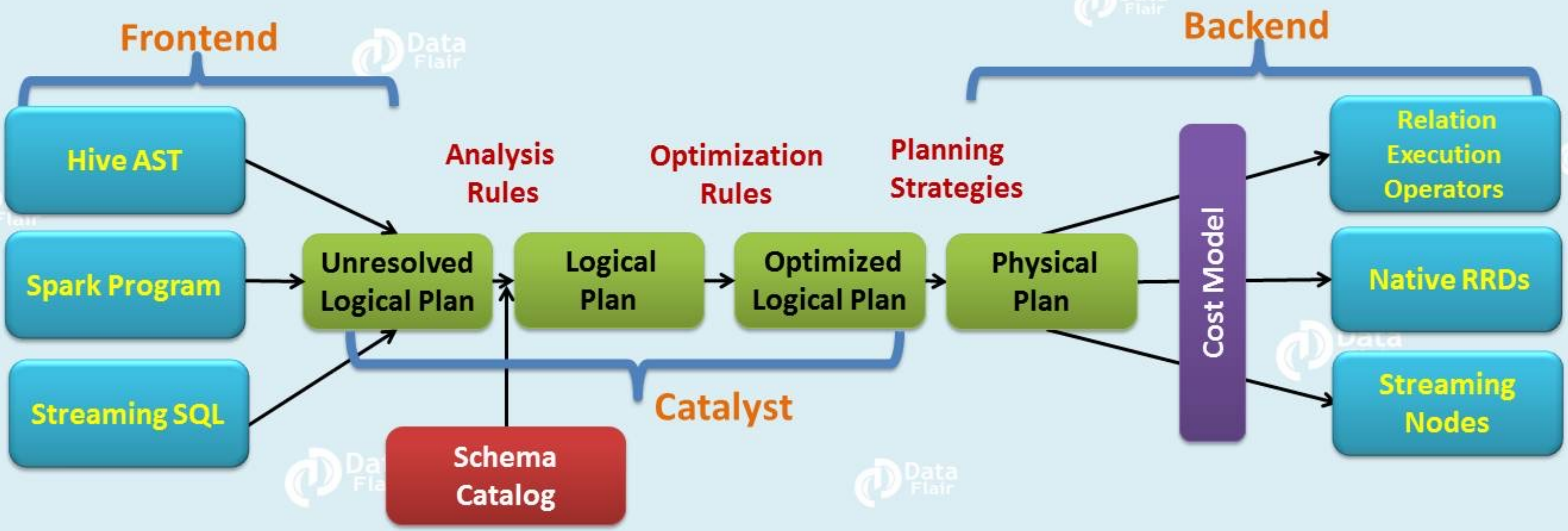
# 4.2. Rules

- Fixed point is a point
- after which
- tree stops changing
- even after applying rules.

# 5. Spark SQL Execution Plan

- In four phases we use Catalyst’s general tree transformation framework:
- Analysis
- Logical Optimization
- Physical planning
- Code generation

# Spark SQL Execution Plan



# 5.1. Analysis

- Spark SQL Optimization starts from relation to be computed. It is computed either from **abstract syntax tree (AST)** returned by **SQL parser** or dataframe object created using API.
- Both may contain unresolved attribute references or relations.

# 5.1. Analysis

- By unresolved attribute,
- it means we don’t know its type or
- have not matched it to an input table.

# 5.1. Analysis

- Spark SQL make use of Catalyst rules and
- a Catalog object that track data
- in all data sources
- to resolve these attributes.

# 5.1. Analysis

- It starts by creating
- an unresolved logical plan, and
- then apply the following steps:
- Search relation BY NAME FROM CATALOG.
- Map the name attribute

# 5.1. Analysis

- For example,
- col, to the input provided
- given operator’s children.

# 5.1. Analysis

- Determine which attributes
- match to the same value
- to give them unique ID.
- Propagate and push type
- through expressions

# Logical Optimization

- In this phase of Spark SQL optimization,
- the standard rule-based optimization is applied to the logical plan.
- It includes **constant folding**, **predicate pushdown**, **projection pruning** and other rules.
- It became extremely easy
- to add a rule for various situations.

# Physical Planning

- There are about 500 lines of code in the physical planning rules.
- In this phase, one or more physical plan is formed
- from the logical plan,
- using physical operator matches
- the Spark execution engine.
- And it selects the plan using the cost model.

# Physical Planning

- It uses Cost-based optimization only to select join algorithms.
- For small relation SQL uses broadcast join,
- the framework supports broader
- use of cost-based optimization.
- It can estimate the cost recursively
- for the whole tree using the rule.

# Physical Planning

- Rule-based physical optimization,
- such as pipelining projections or filters
- into one Spark *map* Operation is also carried out
- by the physical planner.
- Apart from this, it can also push operations from the logical plan into data sources that support predicate or projection pushdown.

# Code Generation

- The final phase of Spark SQL optimization is code generation.
- It involves generating Java bytecode to run on each machine.
- Catalyst uses the special feature of Scala language,
- "**Quasiquotes**" to make code generation easier
- because it is very tough to build code generation engines.

# Code Generation

- Quasiquotes lets the programmatic construction of abstract syntax trees (ASTs) in the Scala language,
- which can then be fed to the Scala compiler at runtime to generate bytecode.
- With the help of a catalyst,

# Code Generation

- we can transform
- a tree representing an expression in SQL
- to an AST for Scala code
- to evaluate that expression, and
- then compile and
- run the generated code.

# Conclusion

- Spark SQL optimization
- enhances the productivity of developers and
- the performance of the queries that they write.

# Conclusion

- A good query optimizer
- automatically rewrites relational queries
- to execute more efficiently

# Conclusion

- Uses techniques such as
- filtering data early,
- utilizing available indexes, and
- even ensuring different data sources are
- joined in the most efficient order

# Conclusion

- By performing these transformations,
- the optimizer improves the execution times
- of relational queries and
- frees the developer from
- focusing on the semantics of their application
- instead of its performance.