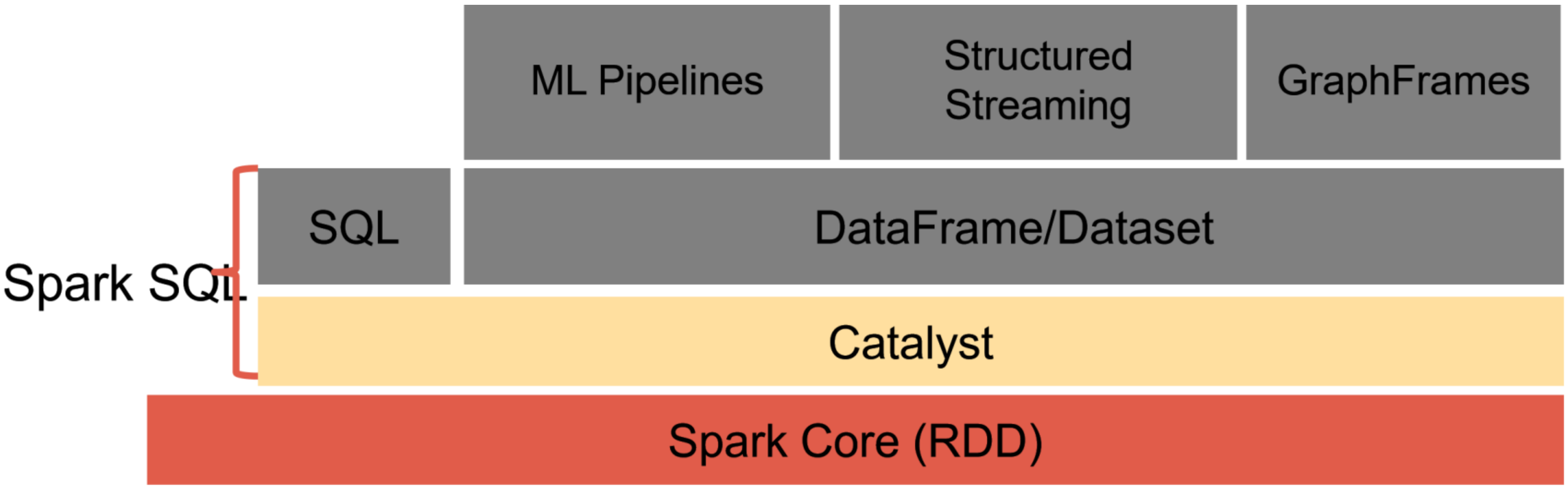
Apache Spark

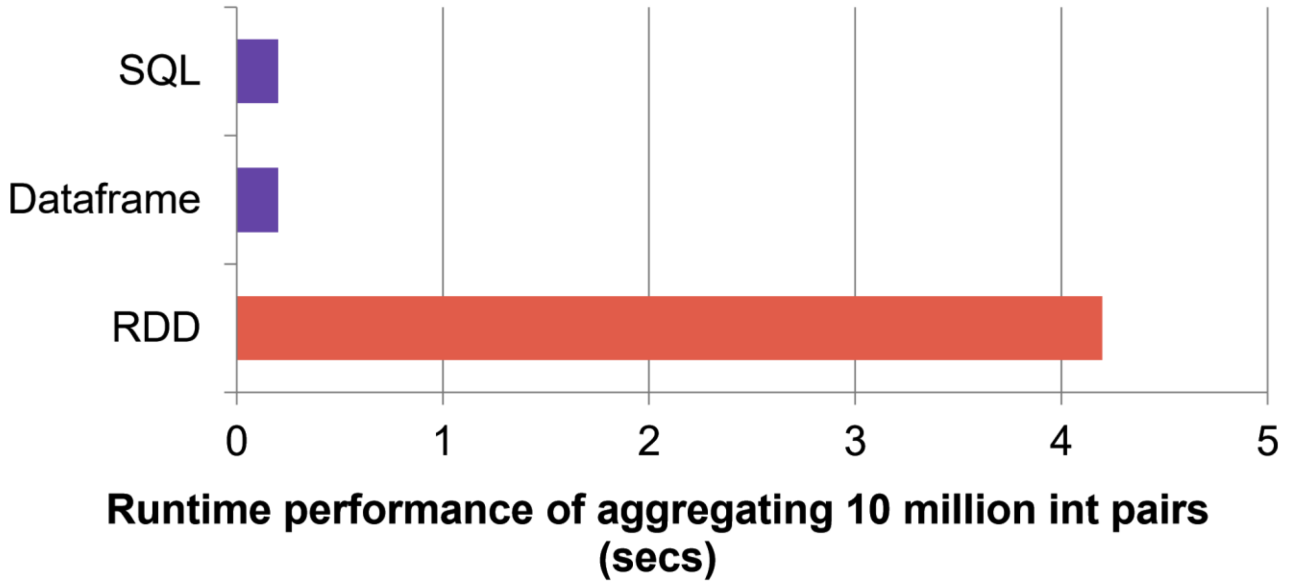
Dmitry Lakhvich (frostball@gmail.com) Data Engineer

Новое API

SparkSQL


ML Pipelines
Structured Streaming
GraphFrames
SQL
DataFrame/Dataset
Spark SQL
Catalyst
Spark Core (RDD)

Преимущества нового API



Новое API - зачем ?

Dataframe

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

SQL

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

RDD

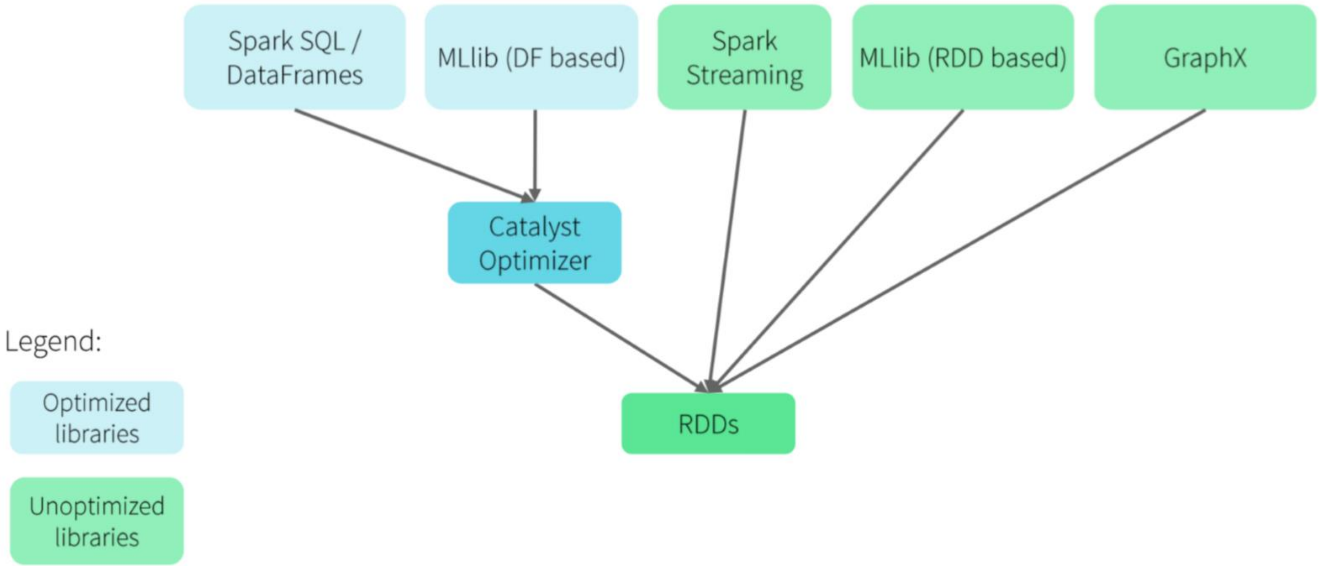
```
data.map { case (dept, age) => dept -> (age, 1) }
     .reduceByKey { case ((a1, c1), (a2, c2)) => (a1 + a2, c1 + c2)}
     .map { case (dept, (age, c)) => dept -> age / c }
```

Structured API

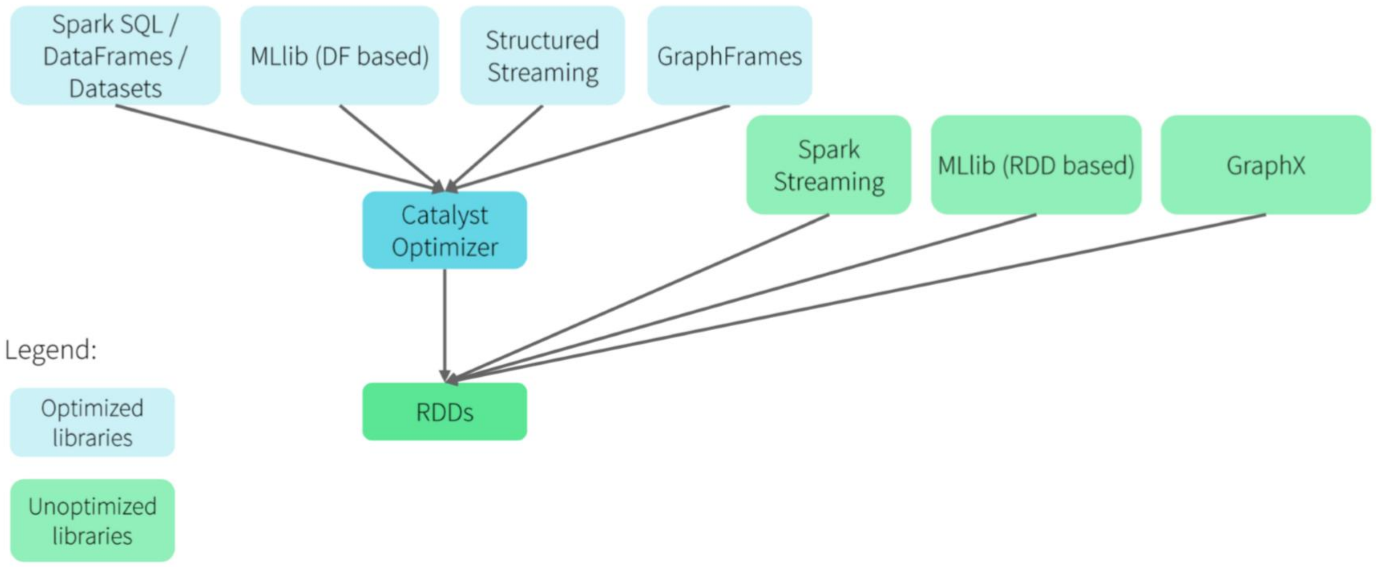
- Структура - уменьшает пространство того, что может быть выражено
- Большая часть вычислений может быть выражена

Уменьшение гибкости API приводит к тому, что вычисления можно сильно оптимизировать исходя из тех ограничений, что мы наложили.

Spark 1.6.x

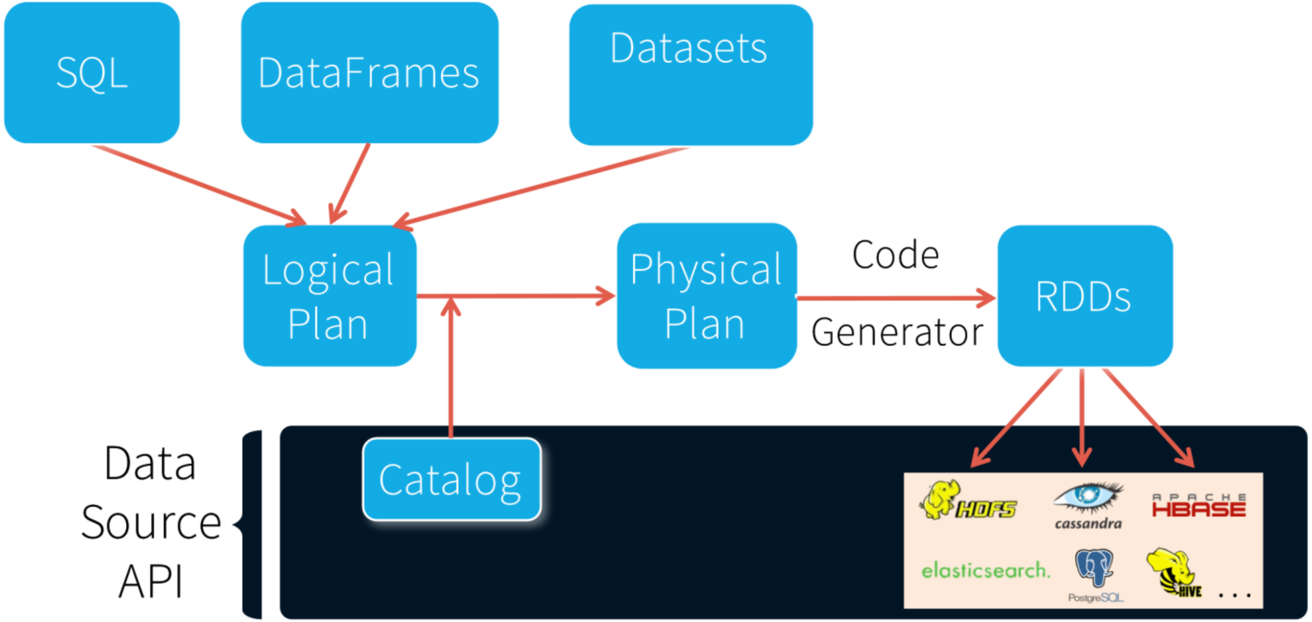

Spark SQL / DataFrames
MLlib (DF based)
Spark Streaming
MLlib (RDD based)
GraphX
Catalyst Optimizer
RDDs
Legend:
Optimized libraries
Unoptimized libraries

Spark 2.x.x

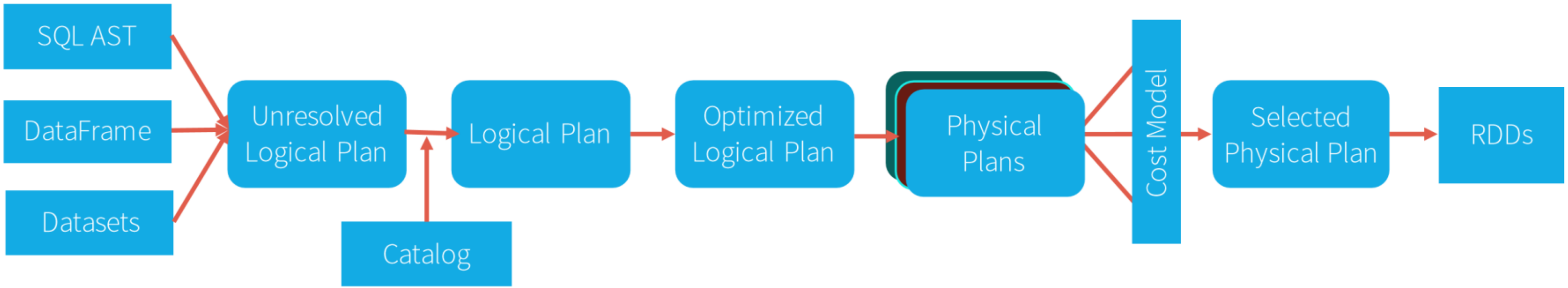


Spark SQL / DataFrames / Datasets
MLlib (DF based)
Structured Streaming
GraphFrames
Spark Streaming
MLlib (RDD based)
GraphX
Catalyst Optimizer
RDDs
Legend:
Optimized libraries
Unoptimized libraries

Новое API - архитектрура



Новое API - архитектрура



Откуда прирост?

Оптимизатор Spark'a пытается построить наиболее оптимальный план вычисления конкретной пользовательсткой программы, понимая с какими данными он работает, какие операции он производит, а также какие данные ожидаются на выходе

SparkSession - Scala

```scala
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

SparkSession - Scala

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

Создание DataFrame – people.json

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

Создание DataFrame - Scala

```
val df = spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Создание DataFrame - Python

```
# spark is an existing SparkSession
df = spark.read.json("examples/src/main/resources/people.json")
# Displays the content of the DataFrame to stdout
df.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

Создание DataFrame - Python

```
# spark is an existing SparkSession
df = spark.read.json("examples/src/main/resources/people.json")
# Displays the content of the DataFrame to stdout
df.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

Операции над DataFrame - Scala

```scala
// This import is needed to use the $-notation
import spark.implicits._
// Print the schema in a tree format
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+
```

```scala
// Select everybody, but increment the age by 1
df.select($"name", $"age" + 1).show()
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+

// Select people older than 21
df.filter($"age" > 21).show()
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+
```

Операции над DataFrame - Scala

```
// Select people older than 21
df.filter($"age" > 21).show()
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+

// Count people by age
df.groupBy("age").count().show()
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

Операции над DataFrame - Python

```
# spark, df are from the previous example
# Print the schema in a tree format
df.printSchema()
# root
# |-- age: long (nullable = true)
# |-- name: string (nullable = true)

# Select only the "name" column
df.select("name").show()
# +-------+
# |   name|
# +-------+
# |Michael|
# |   Andy|
# | Justin|
# +-------+
```

```
# Select everybody, but increment the age by 1
df.select(df['name'], df['age'] + 1).show()
# +-------+---------+
# |   name|(age + 1)|
# +-------+---------+
# |Michael|     null|
# |   Andy|       31|
# | Justin|       20|
# +-------+---------+

# Select people older than 21
df.filter(df['age'] > 21).show()
# +---+----+
# |age|name|
# +---+----+
# | 30|Andy|
# +---+----+
```

Операции над DataFrame - Python

```
# Count people by age
df.groupBy("age").count().show()
# +----+-----+
# | age|count|
# +----+-----+
# |  19|    1|
# |null|    1|
# |  30|    1|
# +----+-----+
```

Больше операций

https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$

SparkSQL - простой select

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Dataset – Scala

```scala
case class Person(name: String, age: Long)

// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS.show()
// +----+---+
// |name|age|
// +----+---+
// |Andy| 32|
// +----+---+

// Encoders for most common types are automatically provided by importing spark.implicits._
val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)
```

Dataset – Scala

```scala
case class Person(name: String, age: Long)

// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS.show()
// +----+---+
// |name|age|
// +----+---+
// |Andy| 32|
// +----+---+

// Encoders for most common types are automatically provided by importing spark.implicits._
val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)
```

Dataset – Scala

```
// DataFrames can be converted to a Dataset by providing a class. Mapping will be done by name
val path = "examples/src/main/resources/people.json"
val peopleDS = spark.read.json(path).as[Person]
peopleDS.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Dataset - Больше операций

https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset

Взаимодействие с RDD– Scala

```scala
// For implicit conversions from RDDs to DataFrames
import spark.implicits._

// Create an RDD of Person objects from a text file, convert it to a Dataframe
val peopleDF = spark.sparkContext
  .textFile("examples/src/main/resources/people.txt")
  .map(_.split(","))
  .map(attributes => Person(attributes(0), attributes(1).trim.toInt))
  .toDF()
// Register the DataFrame as a temporary view
peopleDF.createOrReplaceTempView("people")
```

Взаимодействие с RDD– Scala

```
// SQL statements can be run by using the sql methods provided by Spark
val teenagersDF = spark.sql("SELECT name, age FROM people WHERE age BETWEEN 13 AND 19")

// The columns of a row in the result can be accessed by field index
teenagersDF.map(teenager => "Name: " + teenager(0)).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+
```

Взаимодействие с RDD– Scala

```scala
// SQL statements can be run by using the sql methods provided by Spark
val teenagersDF = spark.sql("SELECT name, age FROM people WHERE age BETWEEN 13 AND 19")

// The columns of a row in the result can be accessed by field index
teenagersDF.map(teenager => "Name: " + teenager(0)).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+
```

Взаимодействие с RDD– Scala

```
// or by field name
teenagersDF.map(teenager => "Name: " + teenager.getAs[String]("name")).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+

// No pre-defined encoders for Dataset[Map[K,V]], define explicitly
implicit val mapEncoder = org.apache.spark.sql.Encoders.kryo[Map[String, Any]]
// Primitive types and case classes can be also defined as
// implicit val stringIntMapEncoder: Encoder[Map[String, Any]] = ExpressionEncoder()

// row.getValuesMap[T] retrieves multiple columns at once into a Map[String, T]
teenagersDF.map(teenager => teenager.getValuesMap[Any](List("name", "age"))).collect()
// Array(Map("name" -> "Justin", "age" -> 19))
```

Взаимодействие с RDD– Scala

```scala
// or by field name
teenagersDF.map(teenager => "Name: " + teenager.getAs[String]("name")).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+

// No pre-defined encoders for Dataset[Map[K,V]], define explicitly
implicit val mapEncoder = org.apache.spark.sql.Encoders.kryo[Map[String, Any]]
// Primitive types and case classes can be also defined as
// implicit val stringIntMapEncoder: Encoder[Map[String, Any]] = ExpressionEncoder()

// row.getValuesMap[T] retrieves multiple columns at once into a Map[String, T]
teenagersDF.map(teenager => teenager.getValuesMap[Any](List("name", "age"))).collect()
// Array(Map("name" -> "Justin", "age" -> 19))
```

Взаимодействие с RDD– people.txt

Michael, 29
Andy, 30
Justin, 19

Взаимодействие с RDD– Python

```python
from pyspark.sql import Row

sc = spark.sparkContext

# Load a text file and convert each line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

# Infer the schema, and register the DataFrame as a table.
schemaPeople = spark.createDataFrame(people)
schemaPeople.createOrReplaceTempView("people")
```

Взаимодействие с RDD– Python

```
# SQL can be run over DataFrames that have been registered as a table.
teenagers = spark.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

# The results of SQL queries are Dataframe objects.
# rdd returns the content as an :class:`pyspark.RDD` of :class:`Row`.
teenNames = teenagers.rdd.map(lambda p: "Name: " + p.name).collect()
for name in teenNames:
    print(name)
# Name: Justin
```

Явное указание схемы данных - Scala

```scala
import org.apache.spark.sql.types._

// Create an RDD
val peopleRDD = spark.sparkContext.textFile("examples/src/main/resources/people.txt")
// The schema is encoded in a string
val schemaString = "name age"
// Generate the schema based on the string of schema
val fields = schemaString.split(" ")
  .map(fieldName => StructField(fieldName, StringType, nullable = true))
val schema = StructType(fields)
// Convert records of the RDD (people) to Rows
val rowRDD = peopleRDD
  .map(_.split(","))
  .map(attributes => Row(attributes(0), attributes(1).trim))
```

Явное указание схемы данных - Scala

```scala
// Apply the schema to the RDD
val peopleDF = spark.createDataFrame(rowRDD, schema)
// Creates a temporary view using the DataFrame
peopleDF.createOrReplaceTempView("people")
// SQL can be run over a temporary view created using DataFrames
val results = spark.sql("SELECT name FROM people")
// The results of SQL queries are DataFrames and support all the normal RDD operations
// The columns of a row in the result can be accessed by field index or by field name
results.map(attributes => "Name: " + attributes(0)).show()
// +-------------+
// |        value|
// +-------------+
// |Name: Michael|
// |   Name: Andy|
// | Name: Justin|
// +-------------+
```

Явное указание схемы данных - Python

```
# Import data types
from pyspark.sql.types import *

sc = spark.sparkContext

# Load a text file and convert each line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
# Each line is converted to a tuple.
people = parts.map(lambda p: (p[0], p[1].strip()))
# The schema is encoded in a string.
schemaString = "name age"
fields = [StructField(field_name, StringType(), True) for field_name in schemaString.split()]
schema = StructType(fields)
```

Явное указание схемы данных - Python

```
# Apply the schema to the RDD.
schemaPeople = spark.createDataFrame(people, schema)
# Creates a temporary view using the DataFrame
schemaPeople.createOrReplaceTempView("people")
# SQL can be run over DataFrames that have been registered as a table.
results = spark.sql("SELECT name FROM people")
results.show()
# +-------+
# |   name|
# +-------+
# |Michael|
# |   Andy|
# | Justin|
# +-------+
```

Пользовательские emploees.json

```
{"name":"Michael","salary":3000}
{"name":"Andy","salary":4500}
{"name":"Justin","salary":3500}
{"name":"Berta","salary":4000}
```

Пользовательские агрегаторы(UDAF)

```
import org.apache.spark.sql.{Row, SparkSession}
import org.apache.spark.sql.expressions.MutableAggregationBuffer
import org.apache.spark.sql.expressions.UserDefinedAggregateFunction
import org.apache.spark.sql.types._
```

Пользовательские агрегаторы(UDAF)

```
object MyAverage extends UserDefinedAggregateFunction {
```

Пользовательские агрегаторы(UDAF)

```scala
// Data types of input arguments of this aggregate function
def inputSchema: StructType = StructType(StructField("inputColumn", LongType) :: Nil)
// Data types of values in the aggregation buffer
def bufferSchema: StructType = {
  StructType(StructField("sum", LongType) :: StructField("count", LongType) :: Nil)
}
```

Пользовательские агрегаторы(UDAF)

```
def dataType: DataType = DoubleType
// Whether this function always returns the same output on the identical input
def deterministic: Boolean = true
// Initializes the given aggregation buffer. The buffer itself is a `Row` that in addition to
// standard methods like retrieving a value at an index (e.g., get(), getBoolean()), provides
// the opportunity to update its values. Note that arrays and maps inside the buffer are still
// immutable.
def initialize(buffer: MutableAggregationBuffer): Unit = {
  buffer(0) = 0L
  buffer(1) = 0L
}
```

Пользовательские агрегаторы(UDAF)

```
  // Updates the given aggregation buffer `buffer` with new input data from `input`
  def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
      buffer(0) = buffer.getLong(0) + input.getLong(0)
      buffer(1) = buffer.getLong(1) + 1
    }
  }
  // Merges two aggregation buffers and stores the updated buffer values back to `buffer1`
  def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
    buffer1(0) = buffer1.getLong(0) + buffer2.getLong(0)
    buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1)
  }
  // Calculates the final result
  def evaluate(buffer: Row): Double = buffer.getLong(0).toDouble / buffer.getLong(1)
}
```

Пользовательские агрегаторы(UDAF)

```
// Register the function to access it
spark.udf.register("myAverage", MyAverage)

val df = spark.read.json("examples/src/main/resources/employees.json")
df.createOrReplaceTempView("employees")
df.show()
// +-------+------+
// |   name|salary|
// +-------+------+
// |Michael|  3000|
// |   Andy|  4500|
// | Justin|  3500|
// |  Berta|  4000|
// +-------+------+
```

Пользовательские агрегаторы(UDAF)

```
val result = spark.sql("SELECT myAverage(salary) as average_salary FROM employees")
result.show()
// +--------------+
// |average_salary|
// +--------------+
// |        3750.0|
// +--------------+
```

Типизированные пользовательские агрегаторы(Typed UDAF)

```
import org.apache.spark.sql.{Encoder, Encoders, SparkSession}
import org.apache.spark.sql.expressions.Aggregator

case class Employee(name: String, salary: Long)
case class Average(var sum: Long, var count: Long)
```

Типизированные пользовательские агрегаторы(Typed UDAF)

```
import org.apache.spark.sql.{Encoder, Encoders, SparkSession}
import org.apache.spark.sql.expressions.Aggregator

case class Employee(name: String, salary: Long)
case class Average(var sum: Long, var count: Long)
```

Типизированные пользовательские агрегаторы(Typed UDAF)

```
object MyAverage extends Aggregator[Employee, Average, Double] {
```

Типизированные пользовательские агрегаторы(Typed UDAF)

```
// A zero value for this aggregation. Should satisfy the property that any b + zero = b
def zero: Average = Average(0L, 0L)
// Combine two values to produce a new value. For performance, the function may modify `buffer`
// and return it instead of constructing a new object
def reduce(buffer: Average, employee: Employee): Average = {
  buffer.sum += employee.salary
  buffer.count += 1
  buffer
}
// Merge two intermediate values
def merge(b1: Average, b2: Average): Average = {
  b1.sum += b2.sum
  b1.count += b2.count
  b1
}
```

Типизированные пользовательские агрегаторы(Typed UDAF)

```
// Transform the output of the reduction
def finish(reduction: Average): Double = reduction.sum.toDouble / reduction.count
// Specifies the Encoder for the intermediate value type
def bufferEncoder: Encoder[Average] = Encoders.product
// Specifies the Encoder for the final output value type
def outputEncoder: Encoder[Double] = Encoders.scalaDouble
```

Типизированные пользовательские агрегаторы(Typed UDAF)

```
val ds = spark.read.json("examples/src/main/resources/employees.json").as[Employee]
ds.show()
// +-------+------+
// |   name|salary|
// +-------+------+
// |Michael|  3000|
// |   Andy|  4500|
// | Justin|  3500|
// |  Berta|  4000|
// +-------+------+

// Convert the function to a `TypedColumn` and give it a name
val averageSalary = MyAverage.toColumn.name("average_salary")
val result = ds.select(averageSalary)
result.show()
// +--------------+
// |average_salary|
// +--------------+
// |        3750.0|
// +--------------+
```

Источники данных - Python

```python
df = spark.read.load("examples/src/main/resources/users.parquet")
df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

Источники данных - Scala

```
val peopleDF = spark.read.format("json").load("examples/src/main/resources/people.json")
peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet")

val peopleDFCsv = spark.read.format("csv")
  .option("sep", ";")
  .option("inferSchema", "true")
  .option("header", "true")
  .load("examples/src/main/resources/people.csv")

usersDF.write.format("orc")
  .option("orc.bloom.filter.columns", "favorite_color")
  .option("orc.dictionary.key.threshold", "1.0")
  .save("users_with_options.orc")
```

Sql над файлами - Python

```
val df = spark.sql("SELECT * FROM parquet.`examples/src/main/resources/users.parquet`")
```

Режимы сохранения

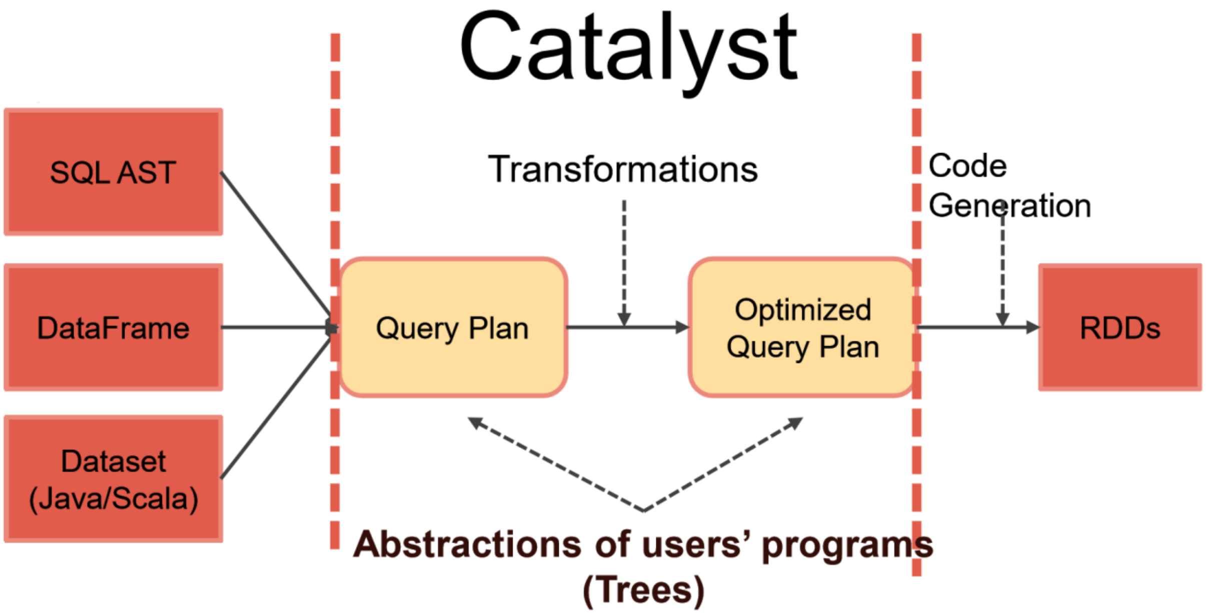
Scala/Java		Определение
SaveMode.ErrorIfExists (default)	"error" or "errorifexists" (default)	
SaveMode.Append	"append"	
SaveMode.Overwrite	"overwrite"	
SaveMode.Ignore	"ignore"	

Бакетирование, сортировка, партицирование

```
peopleDF.write.bucketBy(42, "name").sortBy("age").saveAsTable("people_bucketed")
usersDF.write.partitionBy("favorite_color").format("parquet").save("namesPartByColor.parquet")
usersDF
  .write
  .partitionBy("favorite_color")
  .bucketBy(42, "name")
  .saveAsTable("users_partitioned_bucketed")
```

Catalyst optimizator

Catalyst



Catalyst
SQL AST
DataFrame
Dataset (Java/Scala)
Transformations
Code Generation
Query Plan
Optimized Query Plan
RDDs
Abstractions of users' programs (Trees)

Catalyst - Expression

```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 > 50000) tmp
```

Catalyst - Exprexssion

```
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
```

- Выражение представляет собой новое значение, которое получается вычислением входных данных(1 + 2 + t1.value)
- Аттрибут - колонка датасета или результирующая колонка какой либо функции

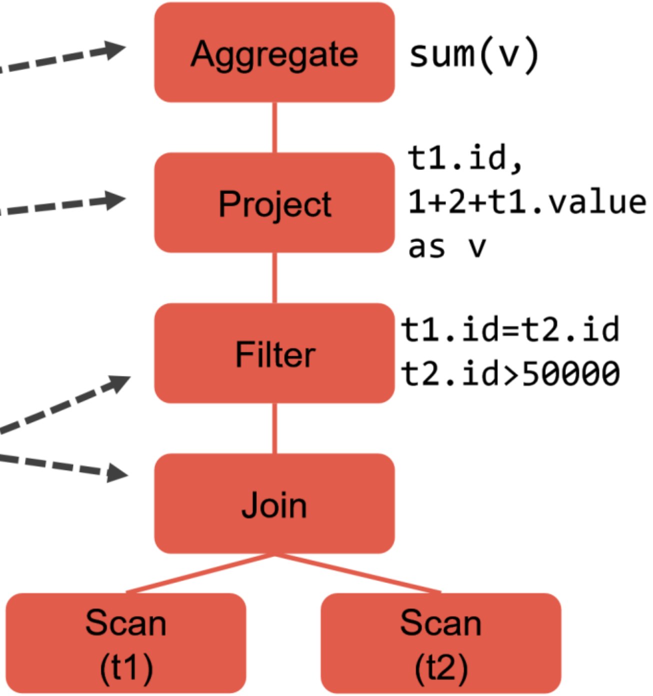
Catalyst - Exprexssion

```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 > 50000) tmp
```

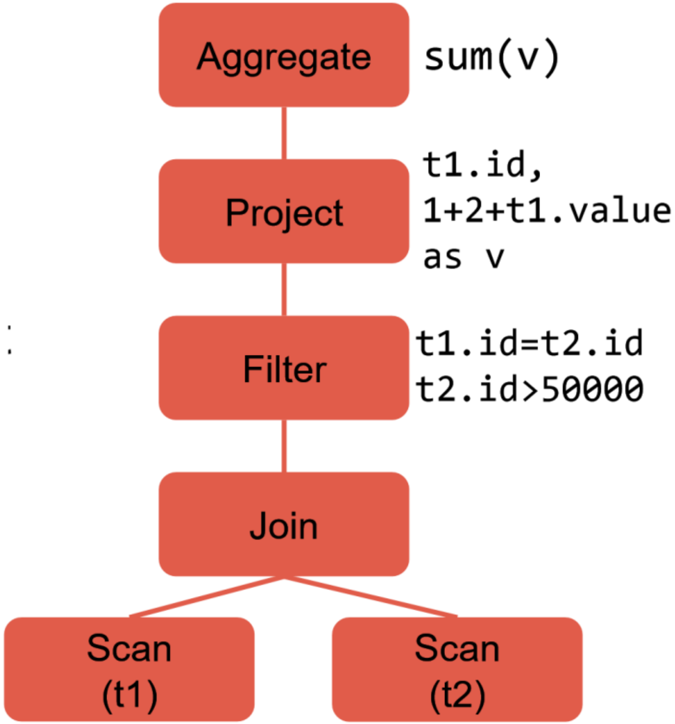
- Выражение представляет собой новое значение, которое получается вычислением входных данных(1 + 2 + t1.value)
- Аттрибут - колонка датасета или результирующая колонка какой либо функции

Catalyst - Exprexssion

```
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
```

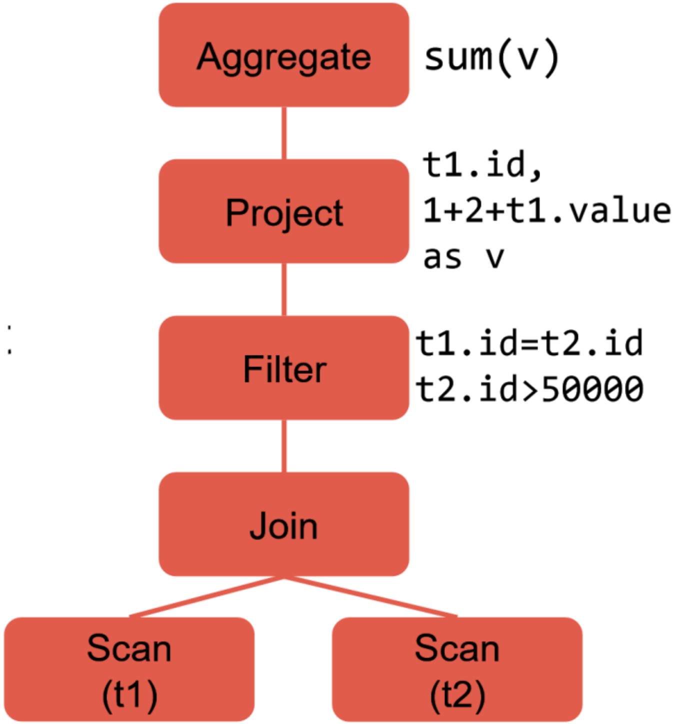


Catalyst – Logical Plan



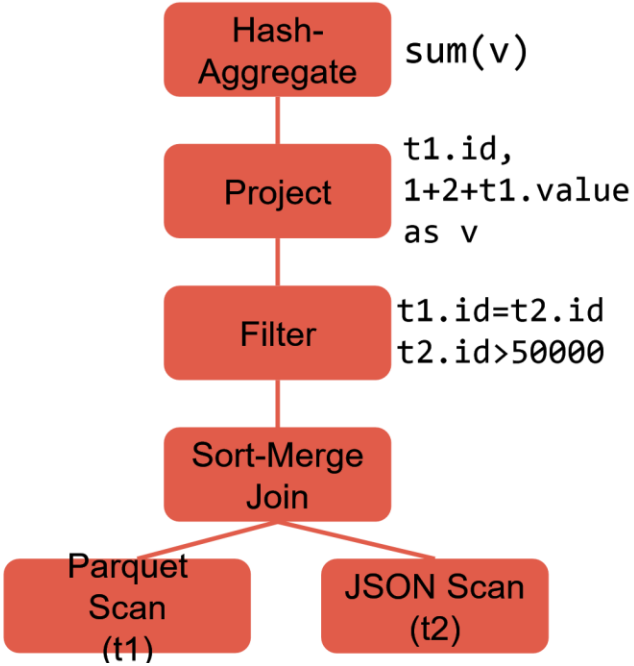
- Логический план описывает вычислениение датасета, без описания того, как именно будут произведены вычисления
- Output: список аттрибутов которые генерирует логический план [v,id]
- Ограничения(constrains): множество инвариантов для строк полученные из логического плана [t2.id>5000]
- Статистика: различные статсистики(размер плана, max/min/null колонок и т.д.)

Catalyst – Логический план



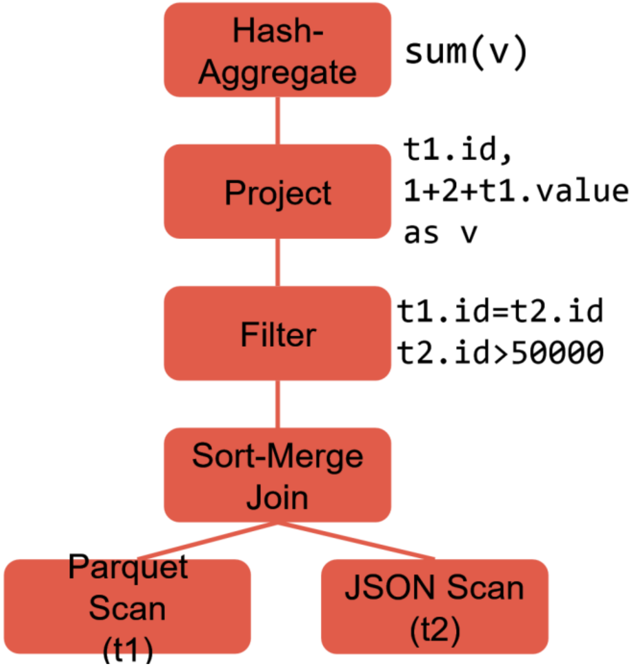
- Логический план
  описывает вычислениение датасета, без описания того, как именно будут произведены вычисления
- Output: список аттрибутов которые генерирует логический план [v,id]
- Ограничения(constrains): множество инвариантов для строк полученные из логического плана [t2.id>5000]
- Статистика: различные статсистики(размер плана, max/min/null колонок и т.д.)

Catalyst – Физический план



- Физический план описывает вычислениение датасета, с указанием того, как именно будут получены и вычислены те или иные данные
- Можно запустить

Catalyst – Физический план



- Физический план описывает вычислениение датасета, с указанием того, как именно будут получены и вычислены те или иные данные
- Можно запустить

Catalyst – Transformation

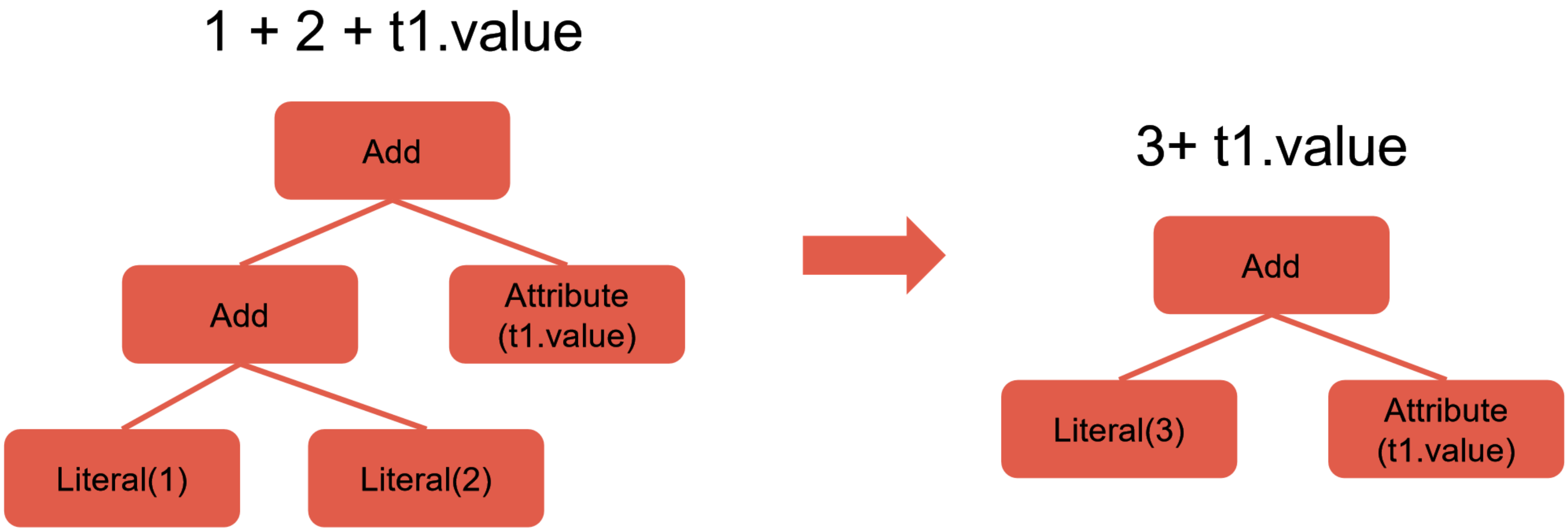
Трасформации без изменения типа(Transform и Rule Executor)

- Expression => Expression
- Logical Plan => Logical Plan
- Physical Plan => Physical Plan

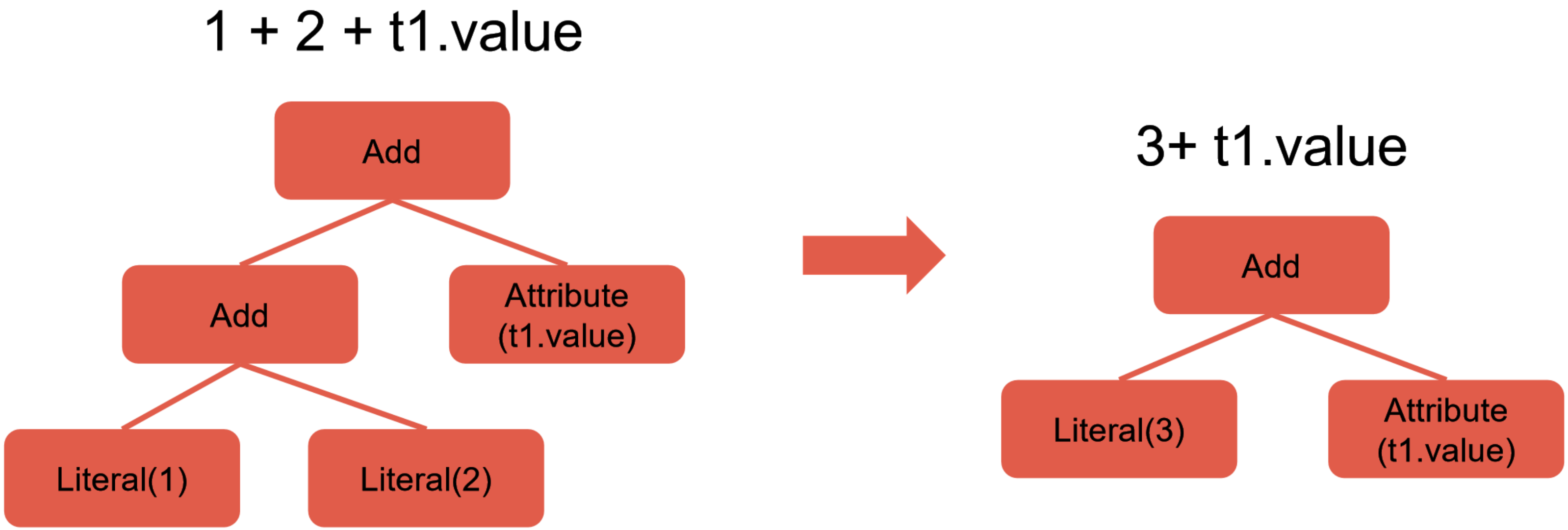
Трансформации с изменением типа дерева

Logical Plan => Physical Plan

Catalyst – Transformation



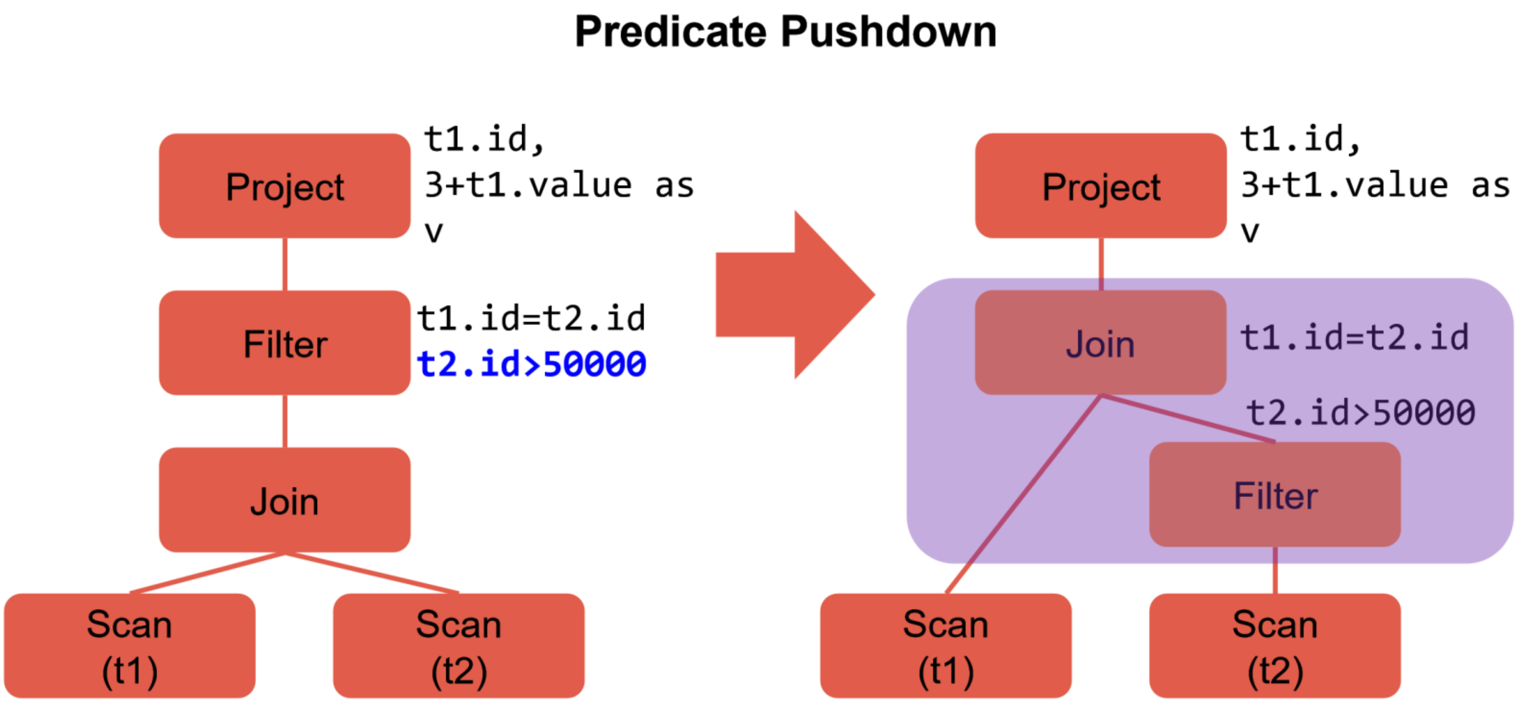
Catalyst – Transformation



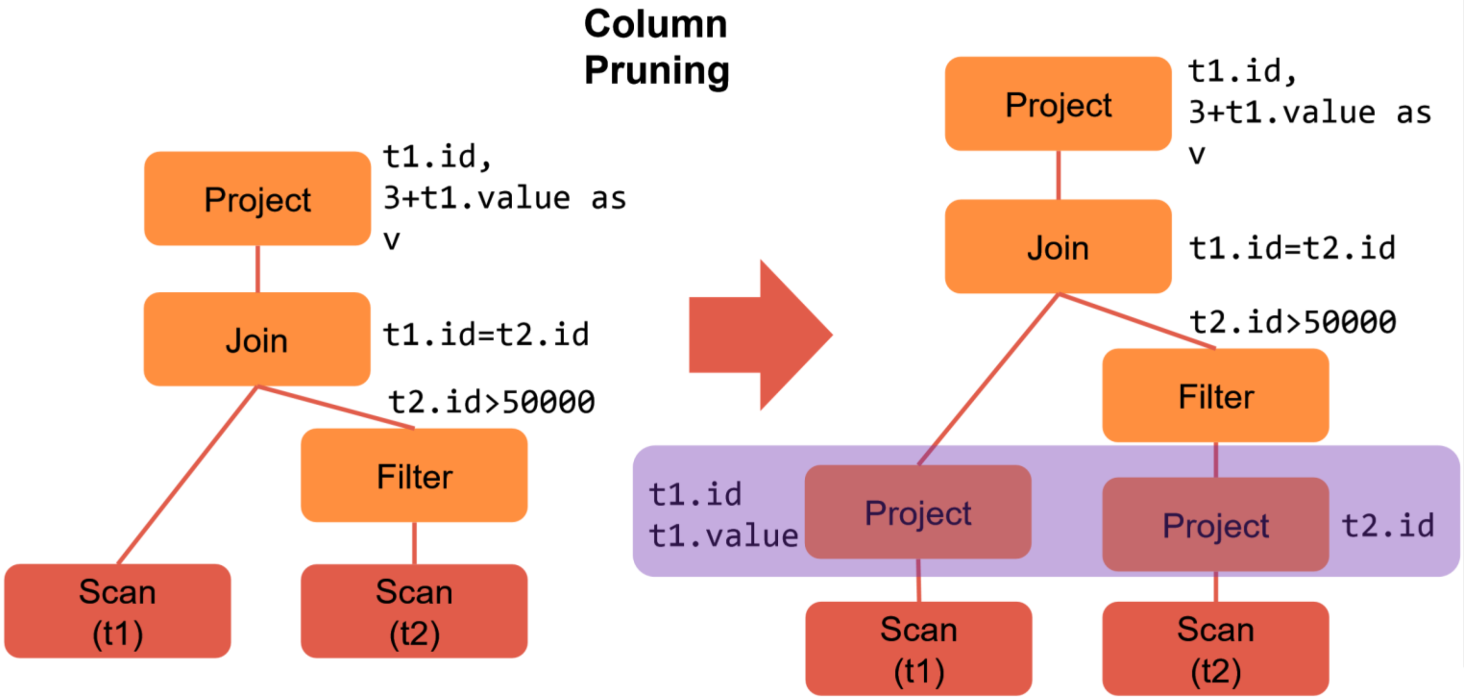
Catalyst – Transformation

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

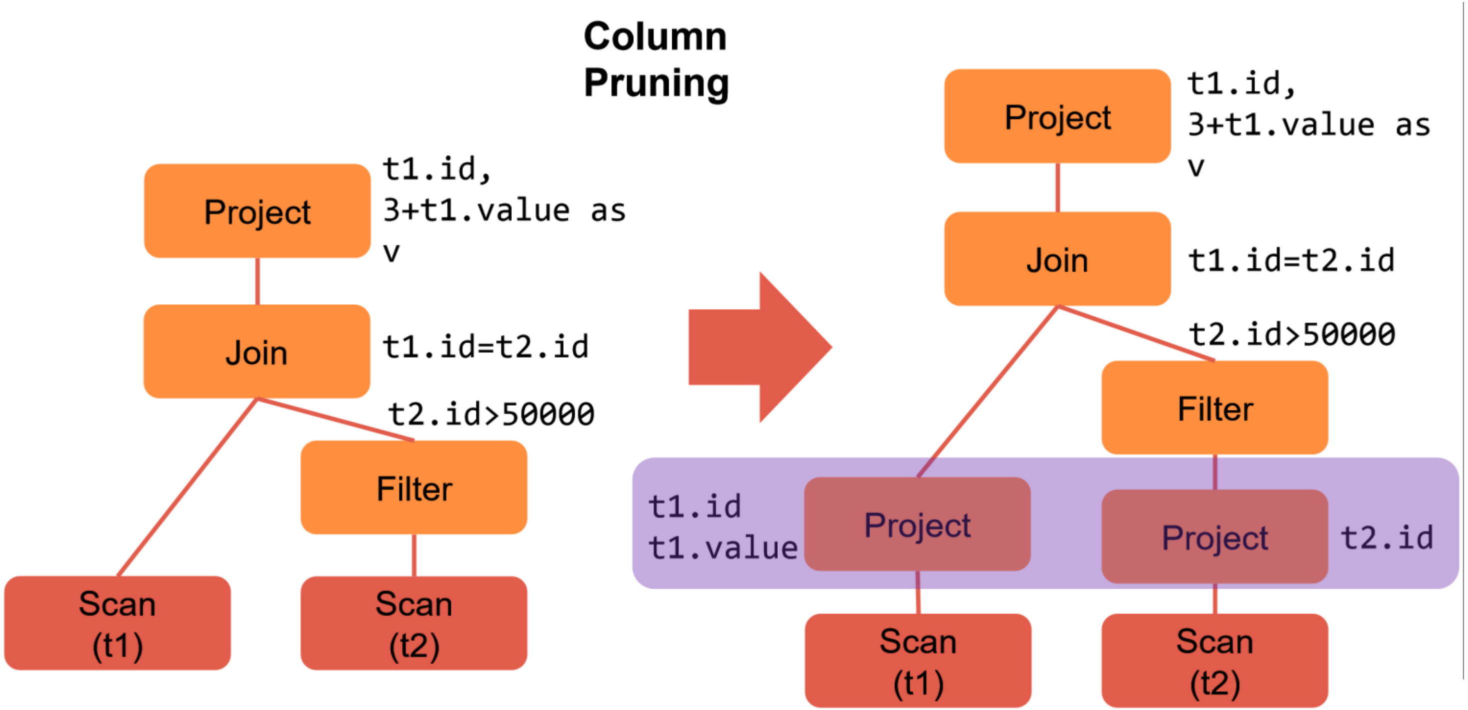
Catalyst – Комбинирование нескольких правил

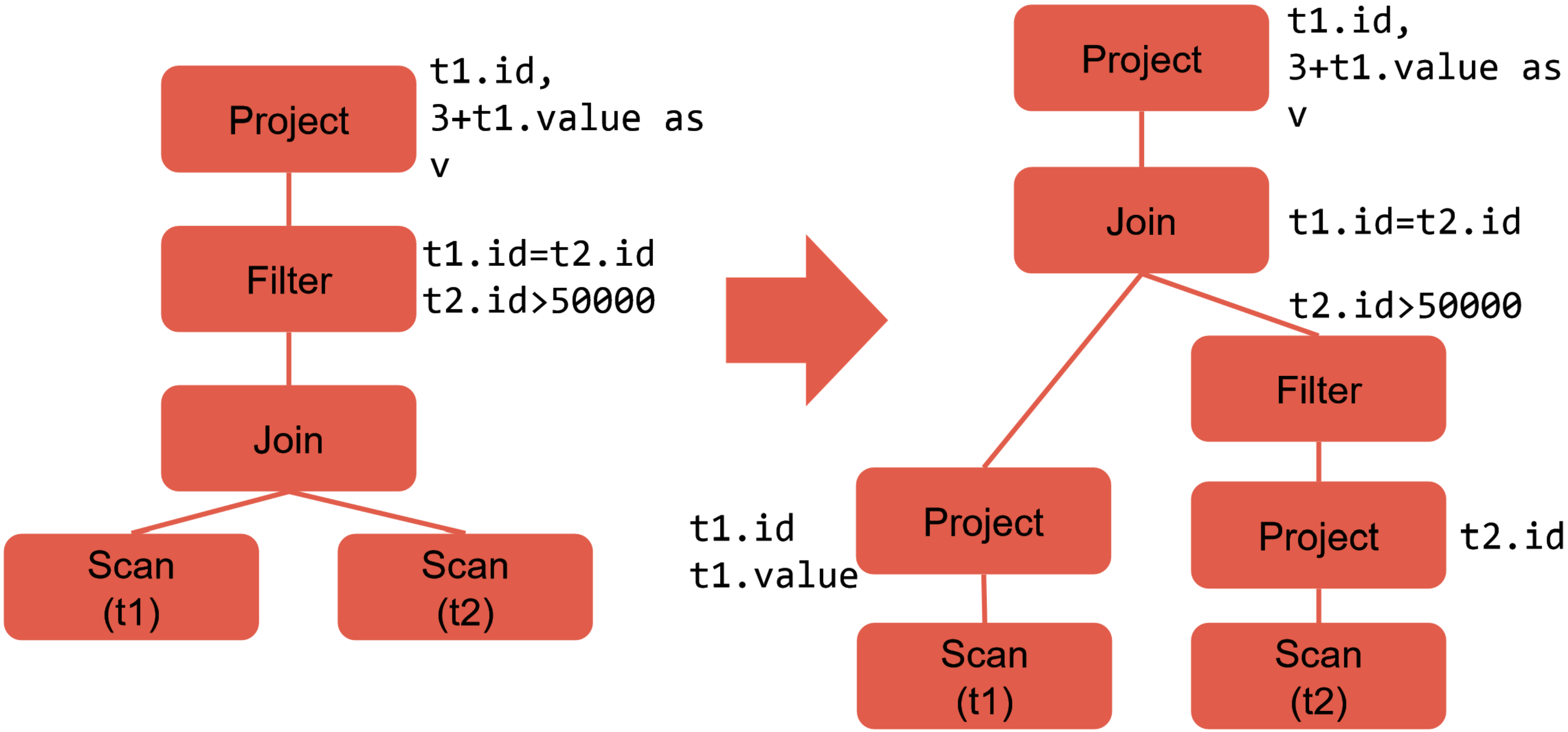


Catalyst – Комбинирование нескольких правил



Catalyst – Комбинирование нескольких правил




Project
t1.id,
3+t1.value as
v
Filter
t1.id=t2.id
t2.id>50000
Join
Scan
(t1)
Scan
(t2)
Project
t1.id,
3+t1.value as
v
Join
t1.id=t2.id
t2.id>50000
Filter
t1.id
t1.value
Project
Project
t2.id
Scan
(t1)
Scan
(t2)

Catalyst: Logical Plan => 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
    …
  }
}
```

Catalyst: Logical Plan => 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
    …
  }
}
```

Catalyst: Logical Plan => Physical Plan

- Анализ(Rule Executor): трансформирует неразрешенный логический план в разрешенный логичечский план
- Логическая оптимизация(Rule Executor): Трасформирует разрешенный логический план в оптимизированный логический план
- Физическое планирование(Стратегии + Rule Executor):

1. Трансформирмация оптимизированного логический план в физический план
2. Применение rule executor и подготовка плана к исполнению

Спасибо

Dmitry Lakhvich

LinkedIn: https://www.linkedin.com/in/dmitry-lakhvich-a610706b/
Twitter: @KrivdaTheTriewe
Telegram: @KrivdaTheTriewe
Email: frostball@gmail.com