Apache Spark SQL: Relational data processing in Spark

CS562 - Lab 4

Michail Giannoulis

What we will be discussing...

- Apache Spark SQL
- DataFrame
- Catalyst Optimizer
- Examples in DSL and SQL
- Example of adding a new rule on Catalyst Optimizer

Nowadays Challenges and Solutions

Challenges	Solutions
Perform ETL to and from various (semi or unstructured) data sources	A *DataFrame* API that can perform relational operations on both external data sources and Spark's built-in RDDs
Perform advanced analytics (e.g. machine learning, graph processing) that are hard to express in relational systems	A highly extensible optimizer, *Catalyst*, that uses features of Scala to add composable rule, control code gen., and define extensions.

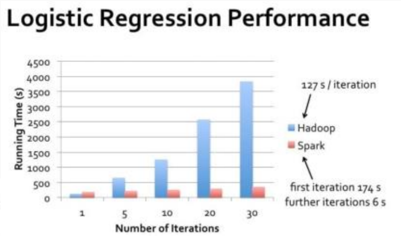
Why Apache Spark ?

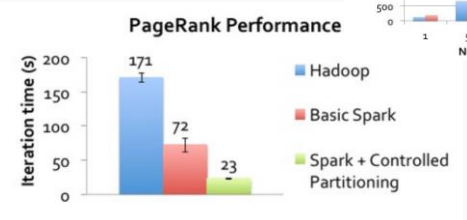
Fast and general cluster computing system, interoperable with Hadoop

Improves efficiency through:

- In-memory computing primitives
- General computation graphs

Up to 100× faster (2-10× on disk)

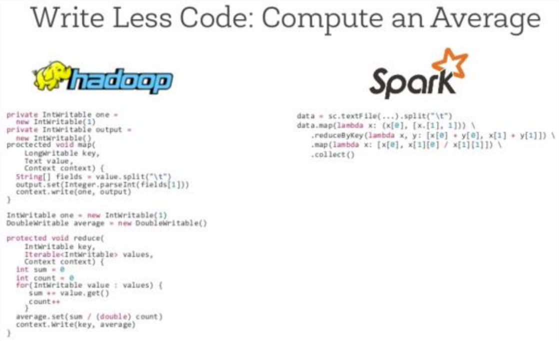




Improves usability through:

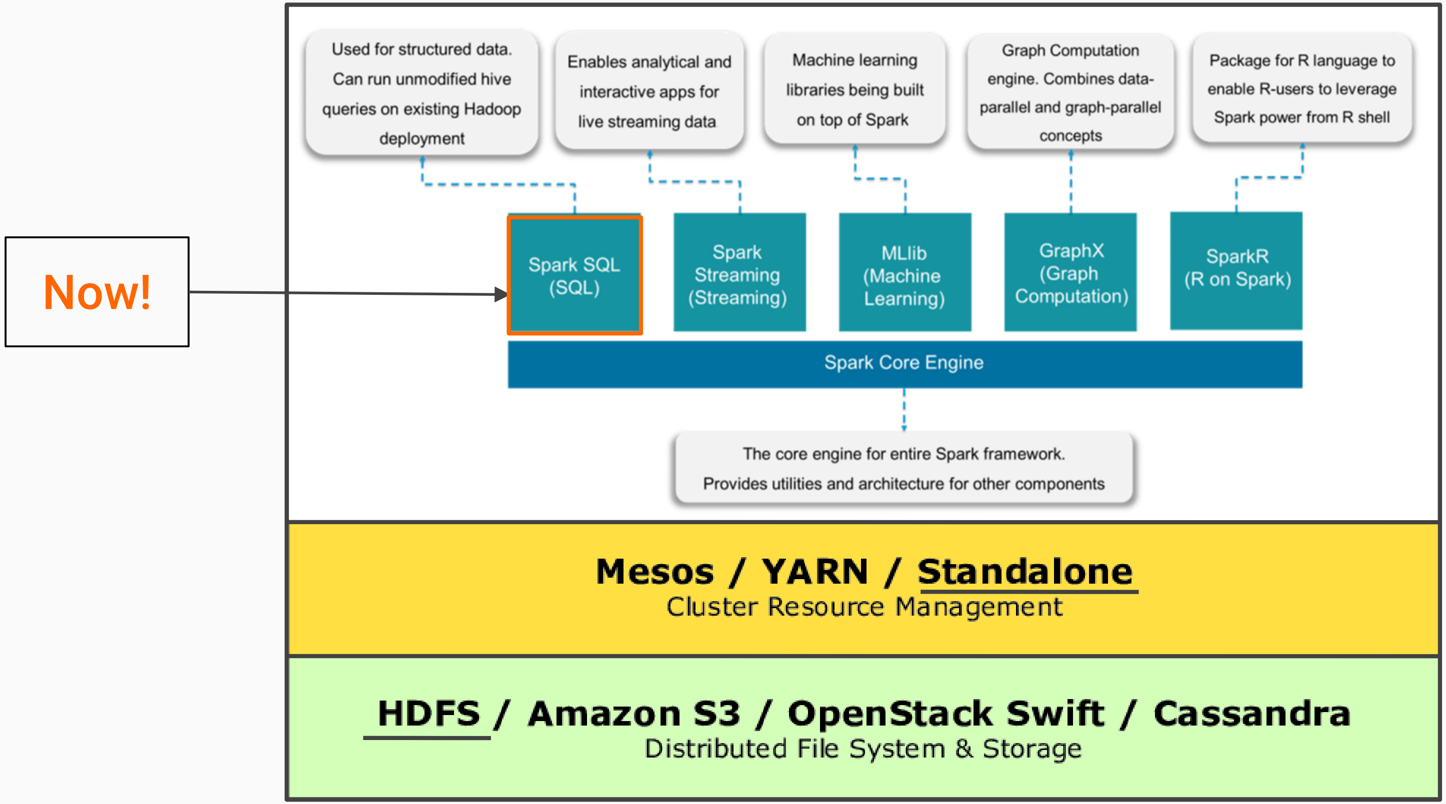
- Rich APIs in Scala, Java, Python
- Interactive shell

2-5× less code



Note: More about Hadoop versus Spark [here](#).

Apache Spark Software Stack



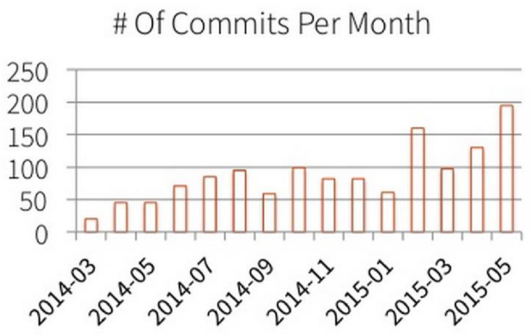
Spark SQL

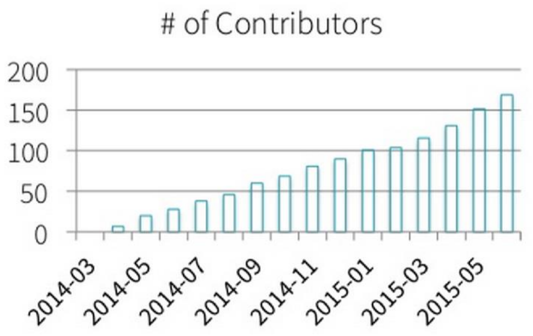
Is a Spark module which Integrates relational processing with Spark's functional programming API

Module Characteristics:

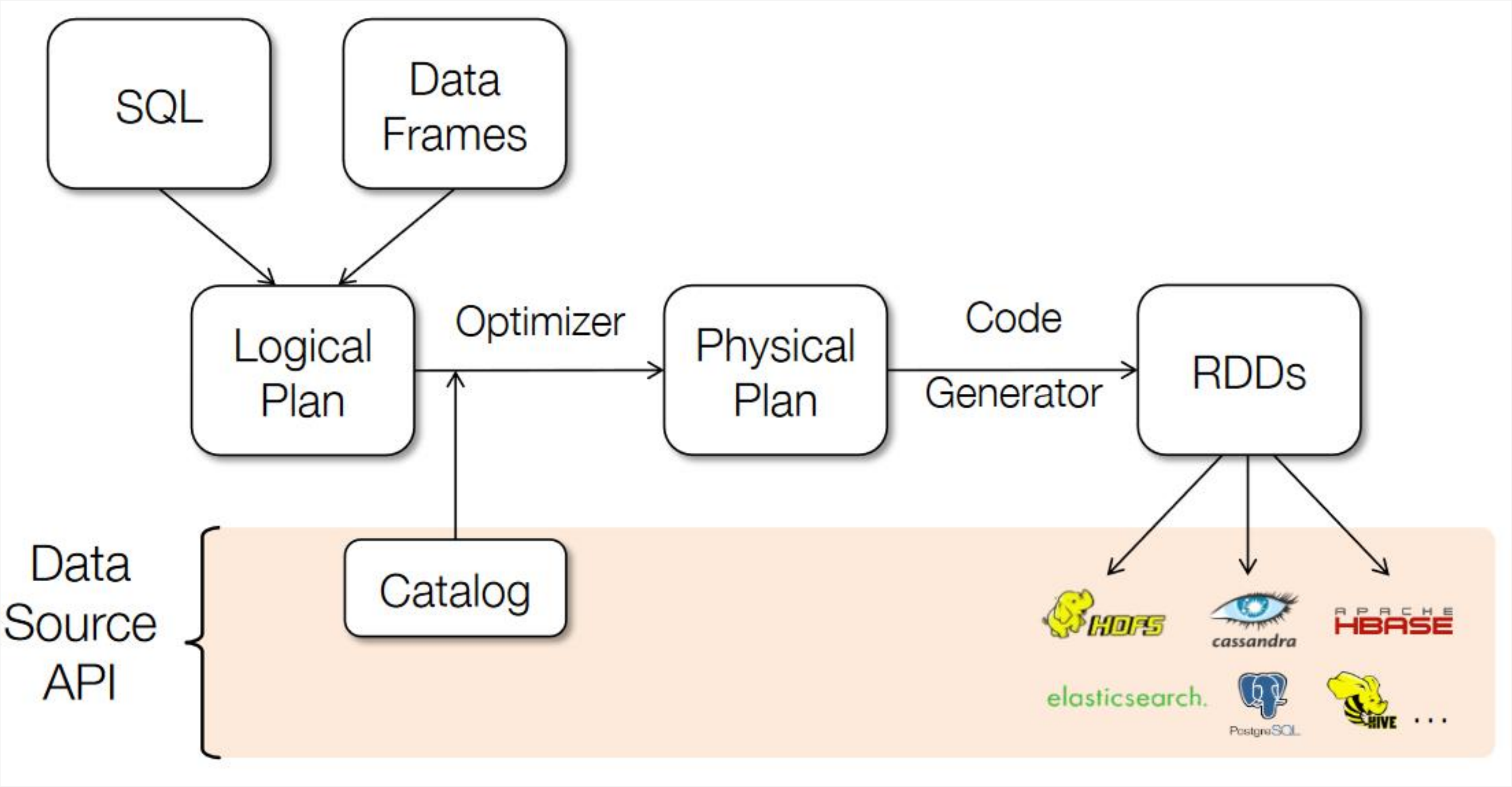
- Supports querying data either via SQL or via Hive Query Language
- Extends the traditional relational data processing

Part of the core distribution since Spark 1.0 (April 2014):





Spark SQL Architecture



How to use Spark SQL ?

You issue SQL queries through a SQLContext or HiveContext, using the sql() method.

- The *sql()* method returns a DataFrame
- You can mix DataFrame methods and SQL queries in the same code

To use SQL you must either:

- Query a persisted Hive table
- Make a table alias for a DataFrame, using the *registerTempTable()* method

Note: a complete guide how to use, can be find here

DataFrame API

Provides a higher level abstraction (built on RDD API), allowing us to use a query language to manipulate data

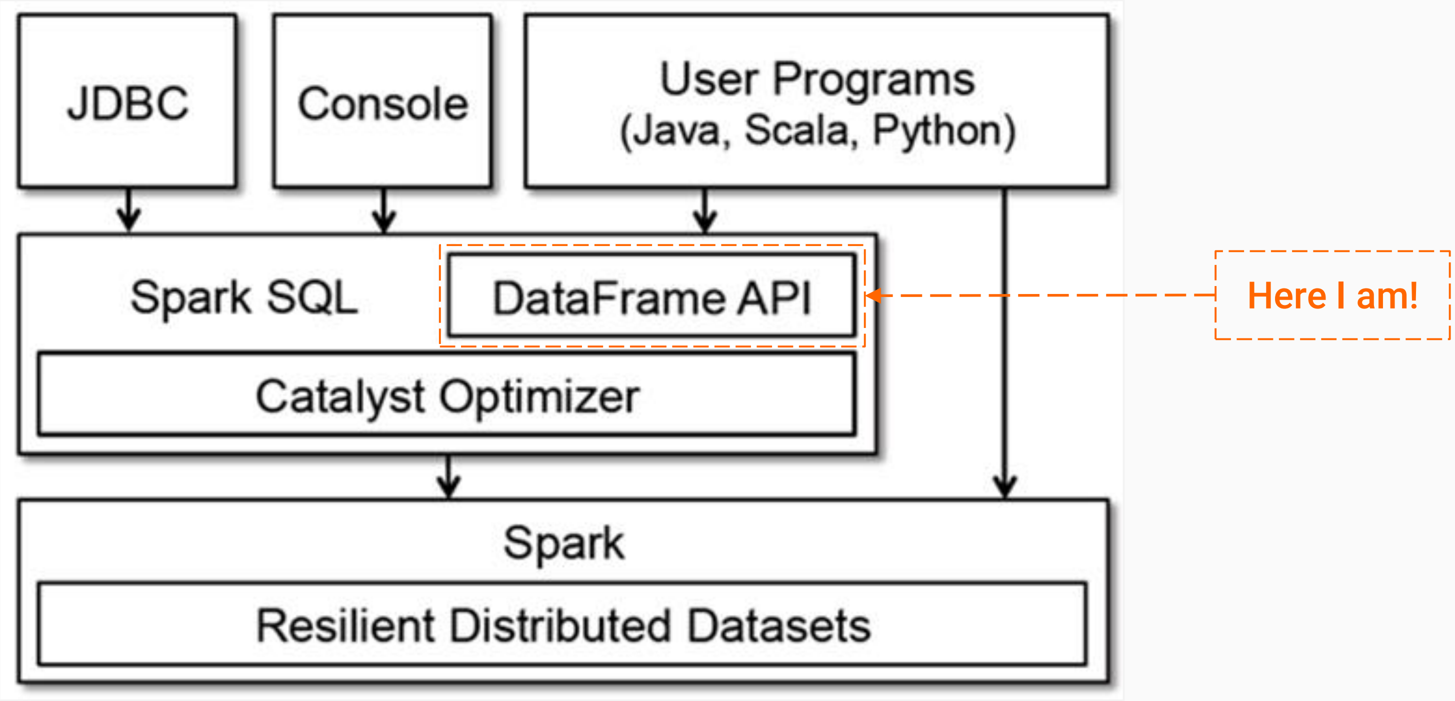
Formal Definition:

- A ***DataFrame (DF)*** is a size-mutable, potentially heterogeneous tabular data structure with labeled axes (i.e., rows and columns)

Characteristics:

- Supports all the RDD operations → but may return back an RDD not a DF
- Ability to scale from kB of data in a single laptop to petabytes on a large cluster
- Support for a wide array of data formats and storage systems
- State-of-the-art optimization and code generation through the Spark SQL Catalyst optimizer
- ...

Spark SQL Interfaces Interaction with SPARK



- Seamless integration with all big data tooling and infrastructure via Spark.
- APIs for Python, Java and R

Why DataFrame ?

What are the advantages over Resilient Distributed Datasets ?

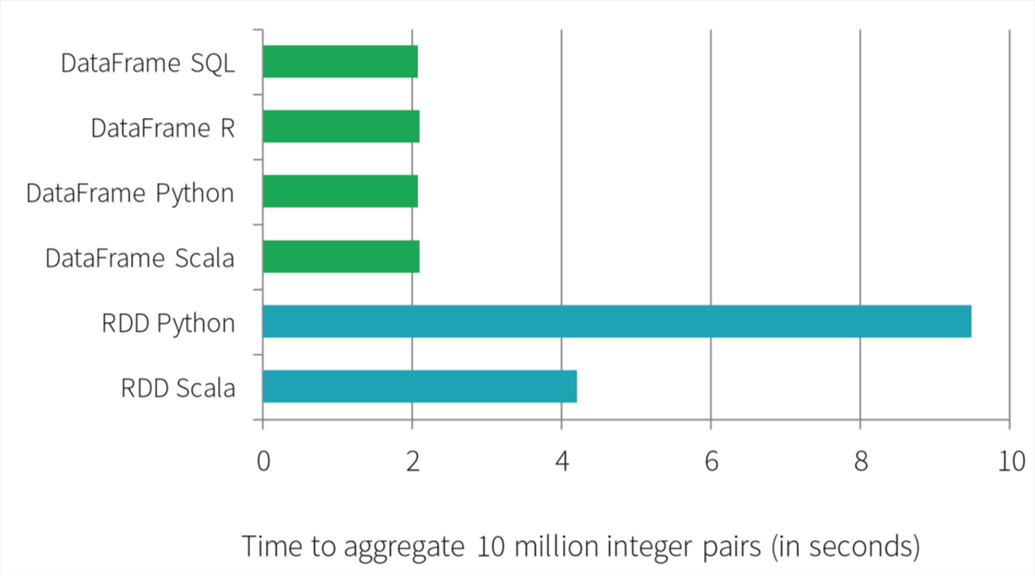
1. Compact binary representation
   - Columnar, compressed cache; rows for processing
2. Optimization across operations (join, reordering, predicate pushdown, etc)
3. Runtime code generation

What are the advantages over Relational Query Languages ?

- Holistic optimization across functions composed in different languages
- Control structures (e..g if, for)
- Logical plan analyzed eagerly → identify code errors associated with data schema issues on the fly

Why DataFrame ?

A DF can be **significantly faster** than RDDs and they perform the **same regardless** the **language**:



But, we have **lost type safety** → ***Array[org.apache.spark.sql.Row]***, *because Row* extends serializable. **Mapping** it back to something **useful** e.g. ***row(0).asInstanceOf[String]***, its **ugly** and **error-prone**.

Querying Native Datasets

Infer column names and types directly from data objects:

```
case class User(name: String, age: Int)
```

- Native objects accessed in-place to avoid expensive data format transformation

Benefits:

- Run relational operations on existing Spark Programs
- Combine RDDs with external structured data

RDD[String] → (User Defined Function) → RDD[User] → (toDF method) → DataFrame

User-Defined Functions (UDFs)

Easy extension of limited operations supported

Allows inline registration of UDFs

- Compare with Pig, which requires the UDF to be written in java package that’s loaded into the Pig script

Can be defined on simple data types or entire tables

UDFs available to other interfaces after registration

```
val model: LogisticRegressionModel = ...

ctx.udf.register("predict",
  (x: Float, y: Float) => model.predict(Vector(x, y)))

ctx.sql("SELECT predict(age, weight) FROM users")
```

DataFrame API: Transformations, Actions, Laziness

- Transformations contribute to the query plan, but they don't execute anything.

Actions cause the execution of the query

DataFrames are lazy!

Transformation examples	Action examples
• filter	• count
• select	• collect
• drop	• show
• intersect	• head
• join	• take

What exactly does "execution of the query" means?

- Spark initiates a distributed read of the data source
- The data flows through the transformations (the RDDs resulting from the catalyst query plan)
- The result of the action is pulled back into the driver JVM

DataFrame API: Actions

Actions

- `def collect(): Array[Row]`
  Returns an array that contains all of Rows in this DataFrame.
- `def collectAsList(): List[Row]`
  Returns a Java list that contains all of Rows in this DataFrame.
- `def count(): Long`
  Returns the number of rows in the DataFrame.
- `def describe(cols: String*): DataFrame`
  Computes statistics for numeric columns, including count, mean, stddev, min, and max.
- `def first(): Row`
  Returns the first row.
- `def head(): Row`
  Returns the first row.
- `def head(n: Int): Array[Row]`
  Returns the first n rows.
- `def show(): Unit`
  Displays the top 20 rows of DataFrame in a tabular form.
- `def show(numRows: Int): Unit`
  Displays the DataFrame in a tabular form.
- `def take(n: Int): Array[Row]`
  Returns the first n rows in the DataFrame.

DataFrame API: Basic Functions

Basic DataFrame functions

```
def cache(): DataFrame.this.type
```

```
def columns: Array[String]
```

Returns all column names as an array.

```
def dtypes: Array[(String, String)]
```

Returns all column names and their data types as an array.

```
def explain(): Unit
```

Only prints the physical plan to the console for debugging purposes.

```
def explain(extended: Boolean): Unit
```

Prints the plans (logical and physical) to the console for debugging purposes.

```
def isLocal: Boolean
```

Returns true if the `collect` and `take` methods can be run locally (without any Spark executors).

```
def persist(newLevel: StorageLevel): DataFrame.this.type
```

```
def persist(): DataFrame.this.type
```

```
def printSchema(): Unit
```

Prints the schema to the console in a nice tree format.

```
def registerTempTable(tableName: String): Unit
```

Registers this DataFrame as a temporary table using the given name.

DataFrame API: Basic Functions

Basic DataFrame functions

```
def schema: StructType
```

Returns the schema of this DataFrame.

```
def toDF(colNames: String*): DataFrame
```

Returns a new DataFrame with columns renamed.

```
def toDF(): DataFrame
```

Returns the object itself.

```
def unpersist(): DataFrame.this.type
```

```
def unpersist(blocking: Boolean): DataFrame.this.type
```

DataFrame API: Language Integrated Queries

Language Integrated Queries

```
def agg(expr: Column, exprs: Column*): DataFrame
```
Aggregates on the entire DataFrame without groups.

```
def agg(exprs: Map[String, String]): DataFrame
```
(Java-specific) Aggregates on the entire DataFrame without groups.

```
def agg(exprs: Map[String, String]): DataFrame
```
(Scala-specific) Aggregates on the entire DataFrame without groups.

```
def agg(aggExpr: (String, String), aggExprs: (String, String)*): DataFrame
```
(Scala-specific) Aggregates on the entire DataFrame without groups.

```
def apply(colName: String): Column
```
Selects column based on the column name and return it as a Column.

```
def as(alias: Symbol): DataFrame
```
(Scala-specific) Returns a new DataFrame with an alias set.

Note: More details about these functions here.

DataFrame API: Relational Operations

Relational operations, select, where, join, groupBy via a domain-specific language:

- Operators take ***expression*** objects
- Operators build up an ***Abstract Syntax Tree (AST)***, which is then optimized by *Catalyst*

```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

Alternatively, register as temp SQL table and perform traditional SQL query strings:

```
users.where(users("age") < 21)
     .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```

SOS

DataFrame API: Output Operations

Output Operations

```
def write: DataFrameWriter
```

Interface for saving the content of the DataFrame out into external storage.

DataFrame API: RDD Operations

RDD Operations

`def coalesce(numPartitions: Int): DataFrame`
Returns a new DataFrame that has exactly numPartitions partitions.

`def flatMap[R](f: (Row) ⇒ TraversableOnce[R])(implicit arg0: ClassTag[R]): RDD[R]`
Returns a new RDD by first applying a function to all rows of this DataFrame, and then flattening the results.

`def foreach(f: (Row) ⇒ Unit): Unit`
Applies a function f to all rows.

`def foreachPartition(f: (Iterator[Row]) ⇒ Unit): Unit`
Applies a function f to each partition of this DataFrame.

`def javaRDD: JavaRDD[Row]`
Returns the content of the DataFrame as a JavaRDD of Rows.

`def map[R](f: (Row) ⇒ R)(implicit arg0: ClassTag[R]): RDD[R]`
Returns a new RDD by applying a function to all rows of this DataFrame.

`def mapPartitions[R](f: (Iterator[Row]) ⇒ Iterator[R])(implicit arg0: ClassTag[R]): RDD[R]`
Returns a new RDD by applying a function to each partition of this DataFrame.

`lazy val rdd: RDD[Row]`
Represents the content of the DataFrame as an RDD of Rows.

`def repartition(numPartitions: Int): DataFrame`
Returns a new DataFrame that has exactly numPartitions partitions.

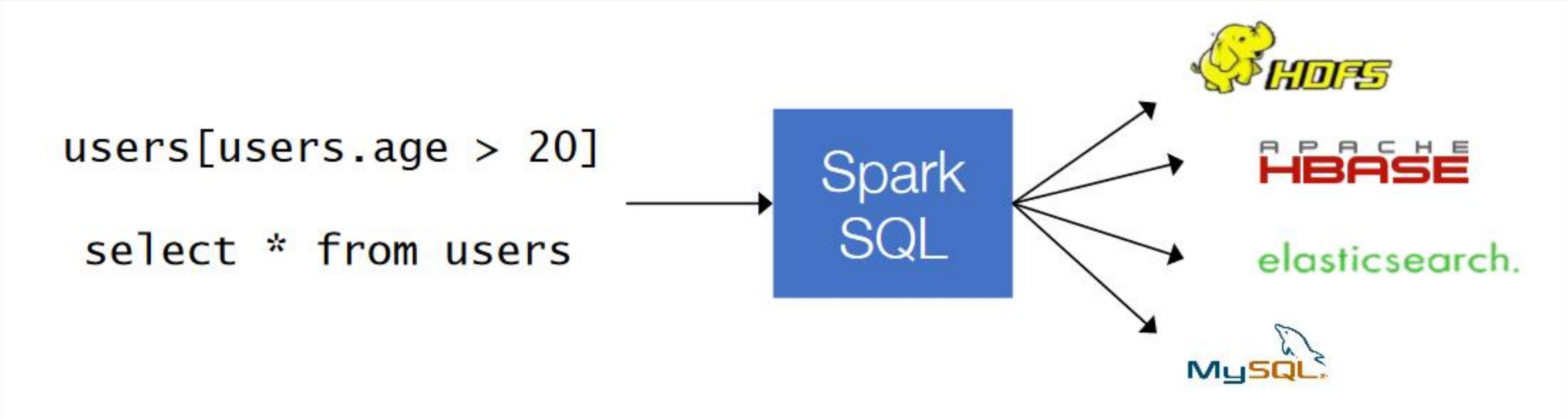
`def toJSON: RDD[String]`
Returns the content of the DataFrame as a RDD of JSON strings.

`def toJavaRDD: JavaRDD[Row]`
Returns the content of the DataFrame as a JavaRDD of Rows.

Data Sources

Uniform way to access structured data:

- Apps can migrate across Hive, Cassandra, JSON, Parquet, etc..
- Rich semantics allows query pushdown into data sources



Apache Spark Catalyst Internals

Deep Dive into Spark SQL's Catalyst Optimizer

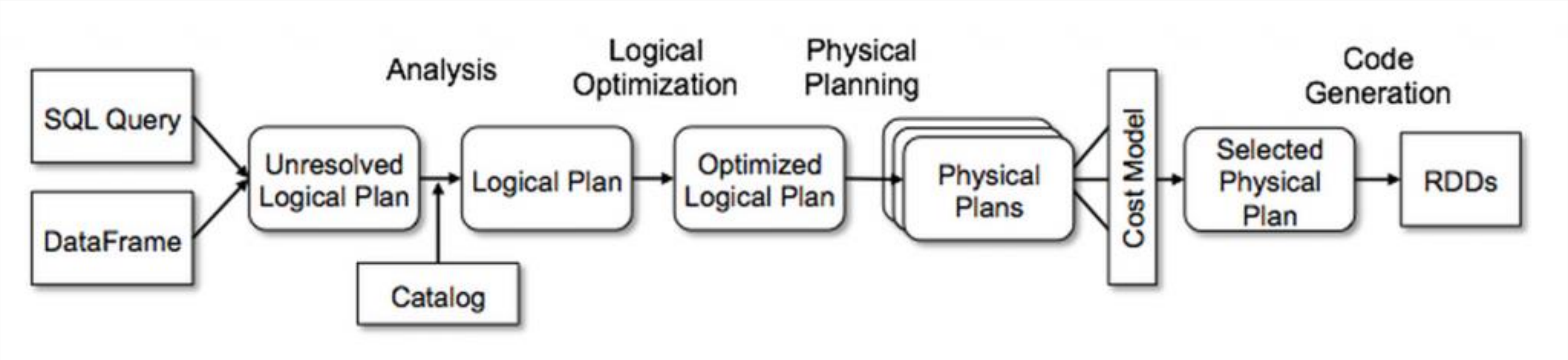
April 13, 2015 | by Michael Armbrust, Yin Huai, Cheng Liang, Reynold Xin and Matei Zaharia

Spark SQL is one of the newest and most technically involved components of Spark. It powers both SQL queries and the new DataFrame API. At the core of Spark SQL is the Catalyst optimizer, which leverages advanced programming language features (e.g. Scala's pattern matching and quasiquotes) in a novel way to build an extensible query optimizer.

We recently published a paper on Spark SQL that will appear in SIGMOD 2015 (co-authored with Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, and Ali Ghodsi). In this blog post we are republishing a section in the paper that explains the internals of the Catalyst optimizer for broader consumption.

More info about this article here.

Apache Spark Execution Plan



- From the above diagram, you can already predict the amount of work that is being done by Spark Catalyst to execute your Spark SQL queries 😨
- The SQL queries of Spark application will be converted to Dataframe APIs
- *Logical Plan* is converted to an ***Optimized Logic plan*** and then to one or more ***Physical Plans***

Note: Find more about what happening under the hood of Spark SQL [here]() and [here]().

The Analyzer

Spark Catalyst's analyzer is responsible for resolving types and names of attributes in SQL queries

- The analyzer looks at the table statistics to know the types of the referred column For example:

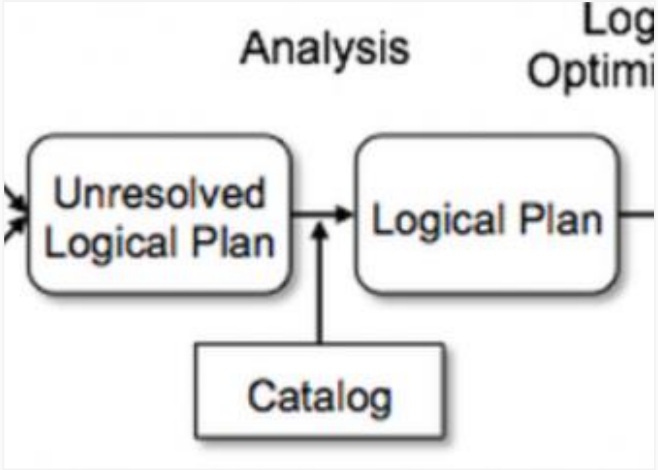
```
SELECT (col1 + 1) FROM mytable;
```

- Now, Spark needs to know:
  1. If `col1` is actually a valid column in `mytable`
  2. If the type of the referred column needs to be known so that `(col1 + 1)` can be validated and necessary type casts cam e added

How analyzer resolve attributes ?

To resolve attributes:

- Look up relations by name from the catalog
- Map named attributes to the input provided given operator's children
- UID for references to the same value
- Propagate the coerce types through expressions (e.g. 1 + col1)



The Optimizer

Spark Catalyst's optimizer is responsible for generating an optimized logical plan from the analyzed logical plan

- Optimization is done by applying rules in batches. Each operation is represented as a *TreeNode* in Spark SQL
- When an analyzed plan goes through the optimizer, the tree is transformed to a new tree repeatedly by applying a set of optimization rules

For instance, a simple Rule:

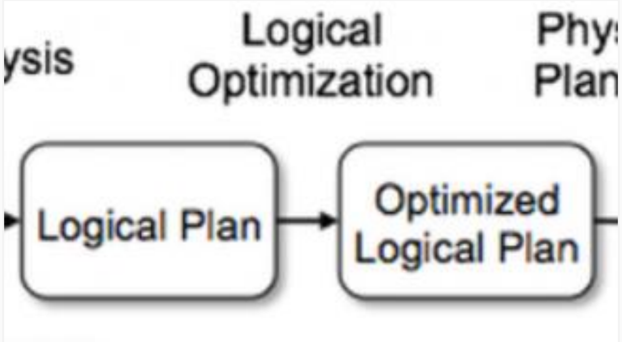
```
Replace the addition of Literal values with new Literal
```

Then, expressions of the form *(1+5)* will be replaced by *6*. Spark will be repeatedly apply such rules to the expression tree until the tree becomes constant

What are the Optimization Rules ?

The optimizer applies standard rule-based optimization rules:

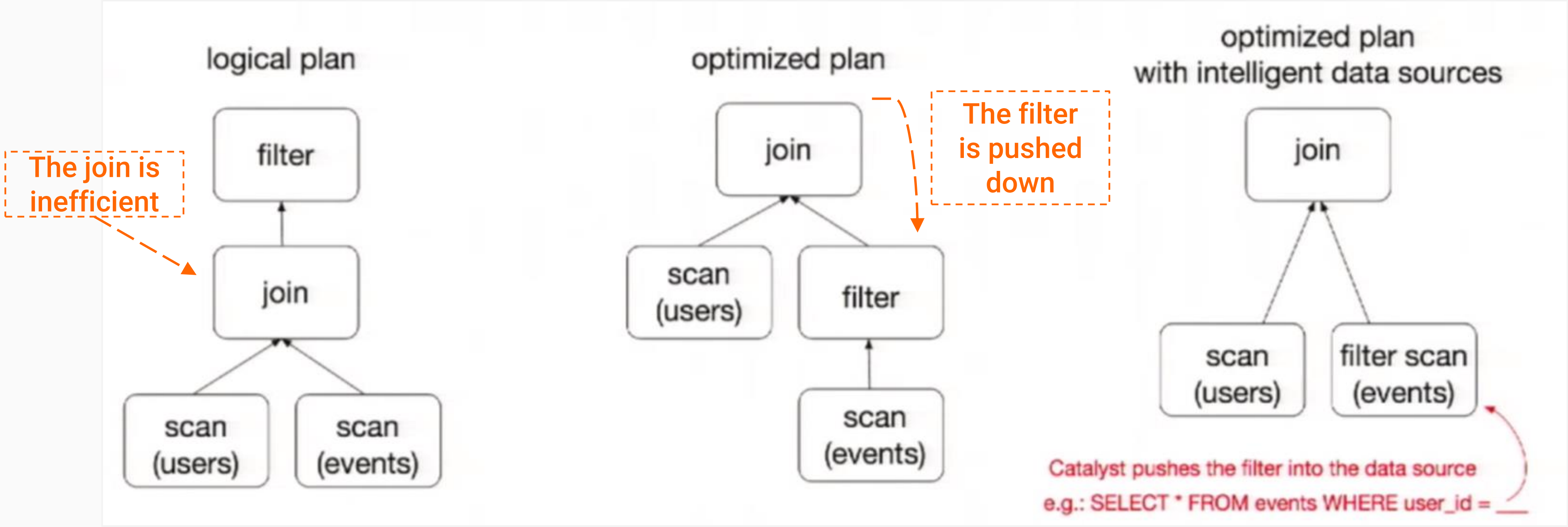
- Constant folding
- Predicate-pushdown
- Projection
- Null propagation
- Boolean expression simplification
- ...



Note: Find more optimization rules [here]

Optimizer: Example

- An inefficient query where ***filter*** is used before *join* operation → Costly shuffle operation (Find more about this example here)

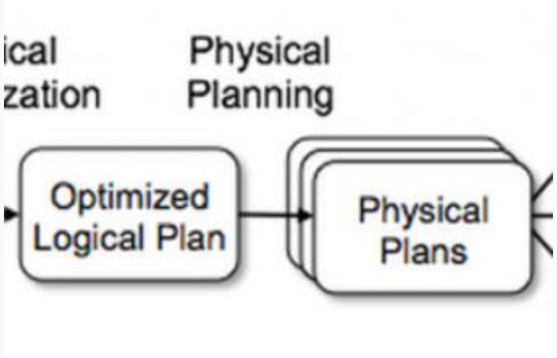



```
users.join(events, users("id") === events("uid"))
     .filter(events("date") > "2015-01-01")
```

Physical Planner

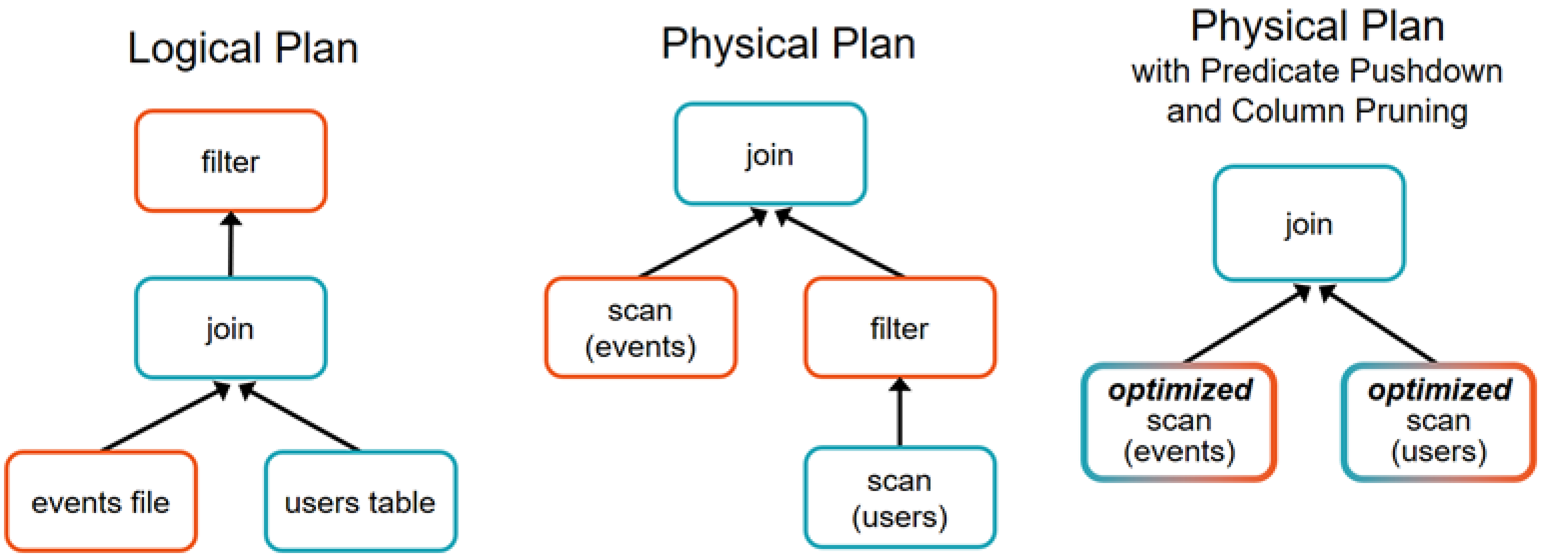
Physical plans are the ones that can actually be executed on a cluster. They actually translate optimized logical plans into RDD operations to be executed on the data source

- A generated ***Optimized Logical Plan*** is passed through a series of Spark strategies that produce one or more ***Physical plans*** (More about these strategies here)
- Spark uses cost based optimization (CBO) to select the best physical plan based on the data source (i.e. table sizes)



Physical Planner: Example

```
def add_demographics(events):
    u = sqlCtx.table("users")                     # Load partitioned Hive table
    events \
      .join(u, events.user_id == u.user_id) \     # Join on user_id
      .withColumn("city", zipToCity(df.zip))      # Run udf to add city column
events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "Melbourne").select(events.timestamp).collect()
```

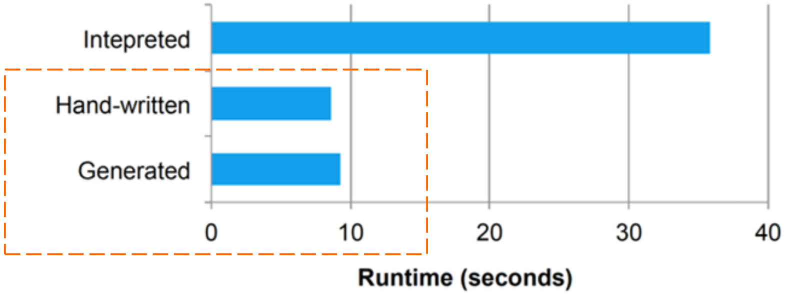


Code Generation

This phase involves generating java bytecode to run on each machine

A comparison of the performance evaluating the expression “x + x + x”, where x is an integer, 1 billion times:

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



- Catalyst transforms a SQL tree into an abstract syntax tree (AST) for scala code to evaluate expressions and generate code

Apache Spark SQL Example

Scala

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

// Create a Spark Session
val spark = SparkSession.builder().appName("test").master("local").getOrCreate()

// read some text source file
val srcDF = spark.read.format("csv").option("header", "true").option("inferSchema", "true").load("/home/jovyan/sales_src.csv")

// self explainatory i guess ? multiply Units Sold column by 2
val unitsBy2 = srcDF.withColumn("Units Sold", $"Units Sold" * 2) // transformation

// Filter rows by order id
val filterOrderId = unitsBy2.filter($"Order Id" > 100) // transformation

// select only
val select = filterOrderId.select($"Region") // transformation

select.take(10) // action

select.explain(extended=true) // spark, please tell me what you did under the hood
```

Save it as ***spark_sql_example.scala*** (Find the source code here)

How to run Apache Spark correctly ?

Run your first .scala script, in three simple steps:

1. Open a command line → `win + R` and type `CMD`
2. Run the spark shell using user-defined memory → `spark-shell --driver-memory 5g`
3. Load the script → `:load <path to>\spark_sql_example.scala`

Schema Inference Example

Scala

Suppose you have a **text file** that looks like this:

```
Erin,Shannon,F,42
Norman,Lockwood,M,81
Miguel,Ruiz,M,64
Rosalita,Ramirez,F,14
Ally,Garcia,F,39
Claire,McBride,F,23
Abigail,Cottrell,F,75
José,Rivera,M,59
Ravi,Dasgupta,M,25
…
```

The file has **no schema**, but looks like:

- **First name**: string
- **Last name**: string
- **Gender**: string
- **Age**: integer

```scala
case class Person(firstName: String,
                  lastName:  String,
                  gender:     String,
                  age:        Int)

val rdd = sc.textFile("people.csv")
val peopleRDD = rdd.map { line =>
  val cols = line.split(",")
  Person(cols(0), cols(1), cols(2), cols(3).toInt)
}
val df = peopleRDD.toDF
// df: DataFrame = [firstName: string, lastName: string,
gender: string, age: int]
```

How to see the Content of a DataFrame?

Scala

You can have Spark tell you what it thinks the data schema is, by calling the *printSchema()* method (This is mostly useful in the shell)

```
scala> df.printSchema()
root
 |-- firstName: string (nullable = true)
 |-- lastName: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- age: integer (nullable = false)
```

You can look at the first *n* elements in a DataFrame with the ***show()*** method

If not specified, *n* defaults to 20

```
scala> df.show()
+---------+--------+------+---+
|firstName|lastName|gender|age|
+---------+--------+------+---+
|     Erin| Shannon|     F| 42|
|   Claire| McBride|     F| 23|
|   Norman|Lockwood|     M| 81|
|   Miguel|    Ruiz|     M| 64|
| Rosalita| Ramirez|     F| 14|
|     Ally|  Garcia|     F| 39|
|  Abigail|Cottrell|     F| 75|
|     José|  Rivera|     M| 59|
+---------+--------+------+---+
```

How to Persist a DataFrame in Memory ?

Spark can cache a DataFrame, using an in-memory columnar format, by calling:

```
scala> df.cache()
```

Which just calls *df.persist(MEMORY_ONLY)*

- Spark will scan only those columns used by the DataFrame and will automatically tune compression to minimize memory usage and GC pressure.

You can remove the cached data from memory, by calling:

```
scala> df.unpersist()
```

How to Select Cols from a DataFrame ? SQL Scala

The ***select()*** is like a SQL SELECT, allowing you to limit the results to specific columns

- The DSL also allows you create on-the-fly derived columns
- The SQL version is also available

```
scala> df.select($"firstName", $"age").show(5)
+---------+---+
|firstName|age|
|     Erin| 42|
|   Claire| 23|
|   Norman| 81|
|   Miguel| 64|
| Rosalita| 14|
+---------+---+
```

```
In[1]: df.registerTempTable("names")
In[2]: sqlContext.sql("SELECT first_name, age, age > 49 FROM names").\
                show(5)
+----------+---+-----+
|first_name|age|  _c2|
+----------+---+-----+
|      Erin| 42|false|
|    Claire| 23|false|
|    Norman| 81| true|
|    Miguel| 64| true|
|  Rosalita| 14|false|
+----------+---+-----+
```

```
scala> df.select($"firstName",
                 $"age",
                 $"age" > 49,
                 $"age" + 10).show(5)
+---------+---+----------+----------+
|firstName|age|(age > 49)|(age + 10)|
+---------+---+----------+----------+
|     Erin| 42|     false|        52|
|   Claire| 23|     false|        33|
|   Norman| 81|      true|        91|
|   Miguel| 64|      true|        74|
| Rosalita| 14|     false|        24|
+---------+---+----------+----------+
```

How to Filter the Rows of a DataFrame?

The ***filter()*** method allows you to filter rows out of your results

- The DSL as well as SQL version are available

```
scala> df.filter($"age" > 49).select($"firstName", $"age").show()
+---------+---+
|firstName|age|
+---------+---+
|   Norman| 81|
|   Miguel| 64|
|  Abigail| 75|
+---------+---+
```

```
In[1]: SQLContext.sql("SELECT first_name, age FROM names " + \
                      "WHERE age > 49").show()
+---------+---+
|firstName|age|
+---------+---+
|   Norman| 81|
|   Miguel| 64|
|  Abigail| 75|
+---------+---+
```

How to Sort the Rows of a DataFrame

The ***orderBy()*** method allows you to sort the results

- The DSL as well as SQL version are available
- It's easy to reverse the sort order

```
scala> df.filter(df("age") > 49).
         select(df("firstName"), df("age")).
         orderBy(df("age"), df("firstName")).
         show()
+---------+---+
|firstName|age|
+---------+---+
|   Miguel| 64|
|  Abigail| 75|
|   Norman| 81|
+---------+---+
```

```
scala> df.filter($"age" > 49).
         select($"firstName", $"age").
         orderBy($"age".desc, $"firstName").
         show()
+---------+---+
|firstName|age|
+---------+---+
|   Norman| 81|
|  Abigail| 75|
|   Miguel| 64|
+---------+---+
```

```
scala> sqlContext.SQL("SELECT first_name, age FROM names " +
     |   "WHERE age > 49 ORDER BY age DESC, first_name").show()
+----------+---+
|first_name|age|
+----------+---+
|    Norman| 81|
|   Abigail| 75|
|    Miguel| 64|
+----------+---+
```

Change the Col Name of a Table in DF?

The *as()* or *alias()* allows you to rename a column. It's especially useful with generated columns

- The DSL as well as SQL version are available

```
scala> df.select($"firstName", $"age", ($"age" < 30).as("young")).
         show()
+----------+---+-----+
|first_name|age|young|
+----------+---+-----+
|      Erin| 42|false|
|    Claire| 23| true|
|    Norman| 81|false|
|    Miguel| 64|false|
|  Rosalita| 14| true|
+----------+---+-----+
```

```
scala> sqlContext.sql("SELECT firstName, age, age < 30 AS young " +
     |                 "FROM names")
+----------+---+-----+
|first_name|age|young|
+----------+---+-----+
|      Erin| 42|false|
|    Claire| 23| true|
|    Norman| 81|false|
|    Miguel| 64|false|
|  Rosalita| 14| true|
+----------+---+-----+
```

Add a New Optimization Rule to Spark Catalyst

Implement the Collapse sorts optimizer rule

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql._
import org.apache.spark.sql.{SparkSession, SparkSessionExtensions}


import org.apache.spark.sql.catalyst.rules.Rule
import org.apache.spark.sql.catalyst.plans.logical._
import org.apache.spark.sql.catalyst.analysis._
import org.apache.spark.sql.catalyst.catalog._
import org.apache.spark.sql.catalyst.expressions.{Expression,
InputFileBlockLength,
InputFileBlockStart,
InputFileName,
RowOrdering}
```

The Optimized logical Plan **with** our new Rule

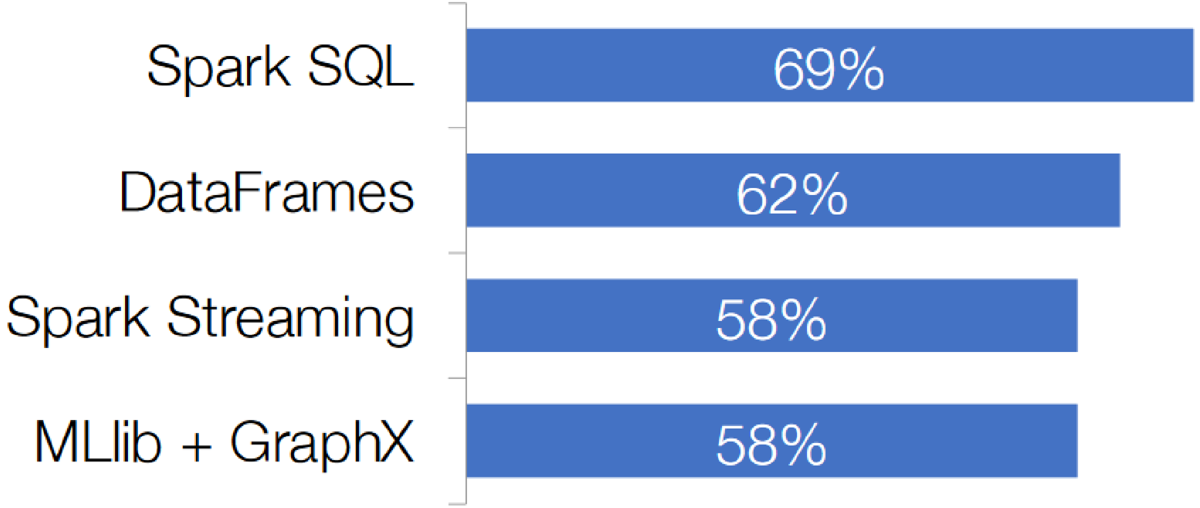
The Optimized logical Plan **without** our new Rule

Query:

- val data = Seq((‘a’, 1), (‘b’,2), (‘c’, 3)).toDF(‘a’, ‘b’)
- val query = data.select(a, b).orderBy(b.asc).filter(‘b ==2’).orderBy(a.asc)

Note: Find more information of this example [here]

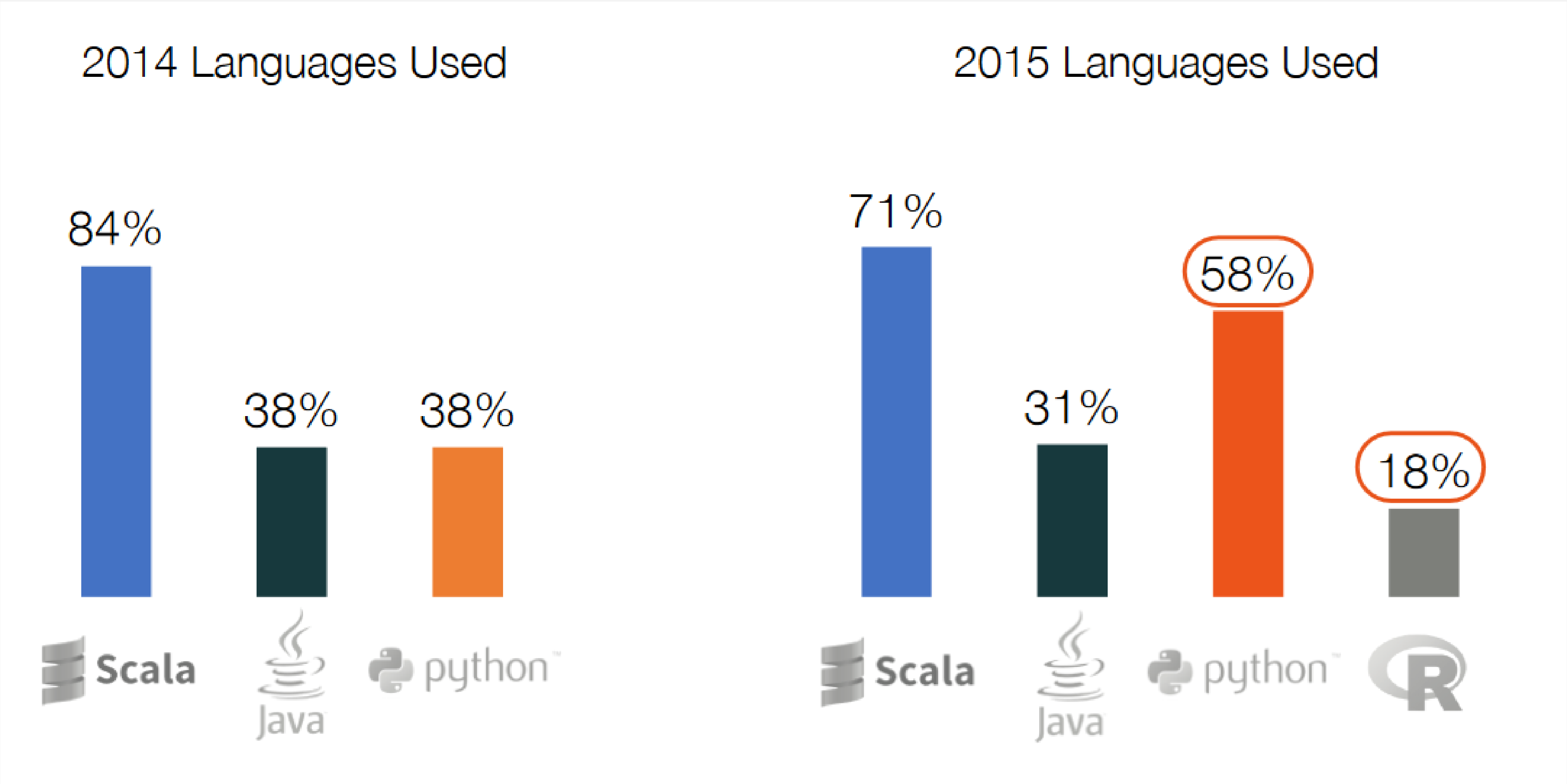
Which Spark Components do People Use?



75% of users use 2 or more components

(Survey 2015)

Which Languages are Used ?



Special Thanks!

Intro to DataFrames and Spark SQL	2015	Databricks
RDDs, DataFrames and Datasets in Apache Spark	2016	Akmal B. Chaudhri
Spark SQL: Relational Data Processing in Spark	2015	Databricks, MIT and Amplab