Spark DataFrame

Programming with distributed tabular data

Spark DataFrame Summary

- Similar to Pandas DataFrame
- Note that operations do not modify the Spark DataFrame
  - They only change copies of the original data frame
  - To update a variable, df, you'll need this pattern
    - df = df.map(…)
- SQL operations are also applied to data frames
- The API for the underlying data structure, RDD, is used less often
  - The resilient distributed dataset (RDD) provides the lower-level parallelism

DataFrame

- Like a
  - R or Python Pandas DataFrame
  - An SQL table
- Better than RDD
  - Higher performance through Catalyst optimization
  - Simpler to use
- Typical usage
  1. Read CVS, JSON data into DataFrame
  2. Process (clean) using Python DataFrame functions
  3. Save a SQL table
  4. Mine with SQL query, then processed with Python ML functions

Using a DataFrame

Example of using common simple operations on DataFrame

Read data into DataFrame

- CSV

```
sdata = spark.read.csv('file:/databricks/driver/USA_Housing.csv', inferSchema=True, header=True)
```

- Others
  - jdbc, json, orc, parquet
- InferSchema
  - Mostly works
  - Often need to
    - specify schema & read
    - After inferSchema=True, then convert columns to appropriate type

```
from pyspark.sql.types import StructType, IntegerType, DateType

schema = StructType([
    StructField("col_01", IntegerType()),
    StructField("col_02", DateType()),
    StructField("col_03", IntegerType())
])

df = spark.read.csv(filename, header=True, nullValue='NA', schema=schema)
```

```
df = df.withColumn('Avg_Area_Income',df['Avg_Area_Income'].cast("double"))
```

https://spark.apache.org/docs/latest/api/python/reference/index.html

Information about DataFrame

- summary, dtypes, printSchema, describe

```
1  display(sdata.summary())
```

▸ (2) Spark Jobs

	summary	Avg_Area_Income	Avg_Area_House_Age	Avg_Area_Number_of_Rooms
1	count	5000	5000	5000
2	mean	68583.10898395971	5.97722203528029	6.987791850907942
3	stddev	10657.991213830363	0.9914561798281722	1.0058332312773866
4	min	17796.631189543397	2.644304186036705	3.2361940234262048
5	25%	61478.633929567324	5.322269839263871	6.298966338516728
6	50%	68803.55207659505	5.969905376273397	7.002864274301249
7	75%	75782.33514026614	6.650746733224263	7.665643100697559

```
1  sdata.describe()
```

▸ (2) Spark Jobs

```
Out[16]: DataFrame[summary: string,
_Population: string, Price: string,
```

```
1  sdata.dtypes
```

```
Out[14]: [('Avg_Area_Income', 'double'),
 ('Avg_Area_House_Age', 'double'),
 ('Avg_Area_Number_of_Rooms', 'double'),
 ('Avg_Area_Number_of_Bedrooms', 'double'),
 ('Area_Population', 'double'),
 ('Price', 'double'),
 ('Address', 'string')]
```

```
1  sdata.printSchema()
```

```
root
 |-- Avg_Area_Income: double (nullable = true)
 |-- Avg_Area_House_Age: double (nullable = true)
 |-- Avg_Area_Number_of_Rooms: double (nullable = true)
 |-- Avg_Area_Number_of_Bedrooms: double (nullable = true)
 |-- Area_Population: double (nullable = true)
 |-- Price: double (nullable = true)
 |-- Address: string (nullable = true)
```

Rename columns

- withColumnRenamed
  - Most common

```
for col in sdata.columns:  # Get rid of dot (.) in column name (causes trouble)
  sdata = sdata.withColumnRenamed(col,col.replace(".", "").replace(" ", "_")) # Get rid of space in column name (causes trouble)
sdata = sdata.withColumn('Avg_Area_Income',sdata['Avg_Area_Income'].cast("double"))
sdata = sdata.withColumn('Avg_Area_House_Age',sdata['Avg_Area_House_Age'].cast("double"))
```

- Using toDF()

```
newColumns = ["newCol1","newCol2","newCol3","newCol4"]
df.toDF(*newColumns).printSchema()
```

https://sparkbyexamples.com/pyspark/pyspark-rename-dataframe-column/

Filter DataFrame

- filter, where
  - Query string

```
1  sdata = sdata.dropna() # Drop rows with any column missing data
2  sdata.where('Avg_Area_Income > 89000 AND Avg_Area_House_Age>8').show()
```

  - Column notation

```
dataframe.filter((dataframe.college == "DU") &
                 (dataframe.student_ID == "1")).show()
```

Expressions must be within parentheses ()

  - SQL functions

```
dataframe.filter((col("college") == "DU") &
                 (col("student_NAME") == "Amit")).show()
```

from pyspark.sql.functions import *

3 ways to refer to DataFrame columns

1. **df.col**: e.g. `F.count(df.col)`
2. **df['col']**: e.g. `df['col'] == 0`
3. **F.col('col')**: e.g. `df.filter(F.col('col').isNull())`

https://stackoverflow.com/questions/55105363/pyspark-dataframe-column-reference-df-col-vs-dfcol-vs-f-colcol

DataFrame to SQL Table

```
3  df = spark.read.csv('/databricks-datasets/definitive-guide/data/retail-data/by-day/2010-12-01.csv', inferSchema=True, header=True)
4  df.show(5)
```

▸ (3) Spark Jobs

▸ df: pyspark.sql.dataframe.DataFrame = [InvoiceNo: string, StockCode: string ... 6 more fields]

```
+---------+---------+--------------------+--------+-------------------+---------+----------+--------------+
|InvoiceNo|StockCode|         Description|Quantity|        InvoiceDate|UnitPrice|CustomerID|       Country|
+---------+---------+--------------------+--------+-------------------+---------+----------+--------------+
|   536365|   85123A|WHITE HANGING HEA...|       6|2010-12-01 08:26:00|     2.55|   17850.0|United Kingdom|
|   536365|    71053| WHITE METAL LANTERN|       6|2010-12-01 08:26:00|     3.39|   17850.0|United Kingdom|
```

```
1  # To SQL view
2  df.createOrReplaceTempView("retail")
```

```
1  %sql
2  select * from retail
3  limit 5
```

▸ (1) Spark Jobs

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
1	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850	United Kingdom
2	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850	United Kingdom
3	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850	United Kingdom

SQL Table to DataFrame

```
1  # from SQL to DataFrame
2  df2 = spark.sql("select * from retail")
3  display(df2)
```

▸ (1) Spark Jobs

▸ df2: pyspark.sql.dataframe.DataFrame = [InvoiceNo: string, StockCode: string ... 6 more fields]

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
1	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850	United Kingdom
2	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850	United Kingdom
3	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850	United Kingdom

RDD, Pandas, and DataFrame

- DataFrame to Pandas
  - pandasDF = pysparkDF.toPandas()
- Pandas to DataFrame
  - df_spark = spark.createDataFrame(panda_data)
- RDD to DataFrame
  - df = rdd.toDF(schema)
- DataFrame to RDD
  - df.rdd

DataFrame API

Spark history of APIs

- RDD
  - In Spark 1.0 release
  - Low-level distributed data handling
  - Immutable data
    - Common programming idiom to update variable: df = df.filter(...)
- DataFrames
  - Introduced in Spark 1.3 release
  - Optimized data processing
- Dataset (Scala, Java, but not PySpark)
  - Introduced in Spark 1.6 release
  - Typed data (checked at compile time)

DataFrame & Dataset

- DataFrame
  - Data organized into named columns, e.g. a table in a relational database.
  - Imposes a structure onto a distributed collection of data, allowing higher-level abstraction
- Dataset (only in typed languages: Java, Scala)
  - Extension of DataFrame API which provides type-safe, object-oriented programming interface (compile-time error detection)
- Built on Spark SQL engine
- Use Catalyst to optimize logical and physical query plan
- Rely on an underlying RDD

https://data-flair.training/blogs/apache-spark-rdd-vs-dataframe-vs-dataset/, https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

DataFrame vs DataSet distinction: typing

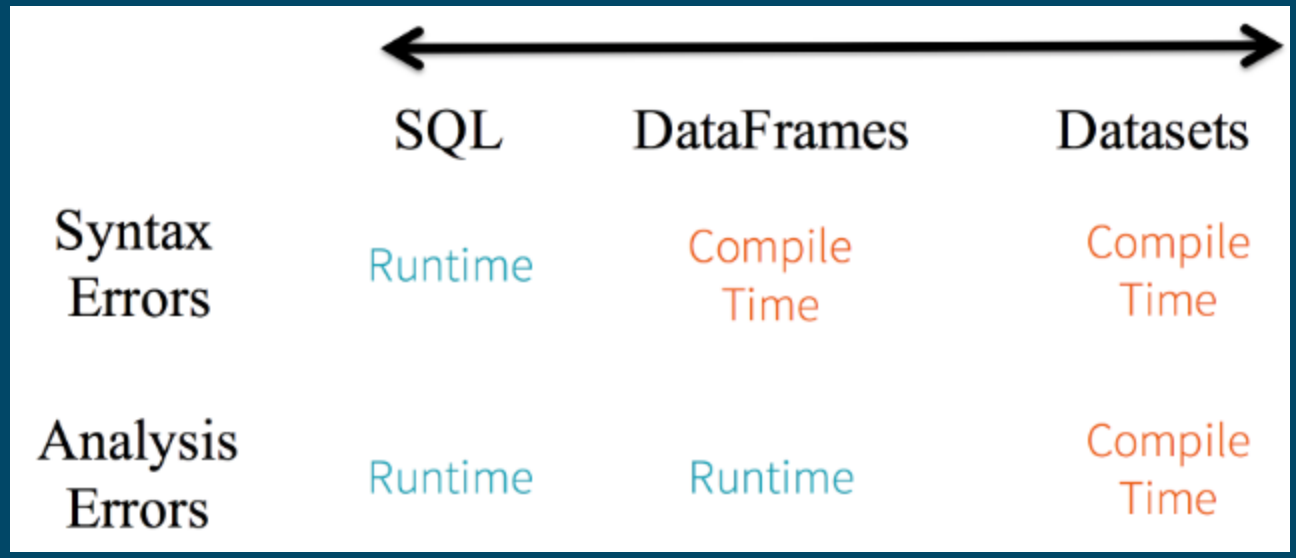
- Python & R don’t have compile-time type safety checks, so only support DataFrame
  - Error detection only at runtime
- Java & Scala support compile-time type safety checks (variables have specified types), so support both DataSet and DataFrame
  - Any mismatch of typed-parameters will be detected at compile time.

https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

DataFrame vs Dataset

- DataFrame in all languages
- Dataset only typed languages
  - Scala, Java

Language	Main Abstraction
Scala	Dataset[T] & DataFrame (alias for Dataset[Row])
Java	Dataset[T]
Python*	DataFrame
R*	DataFrame

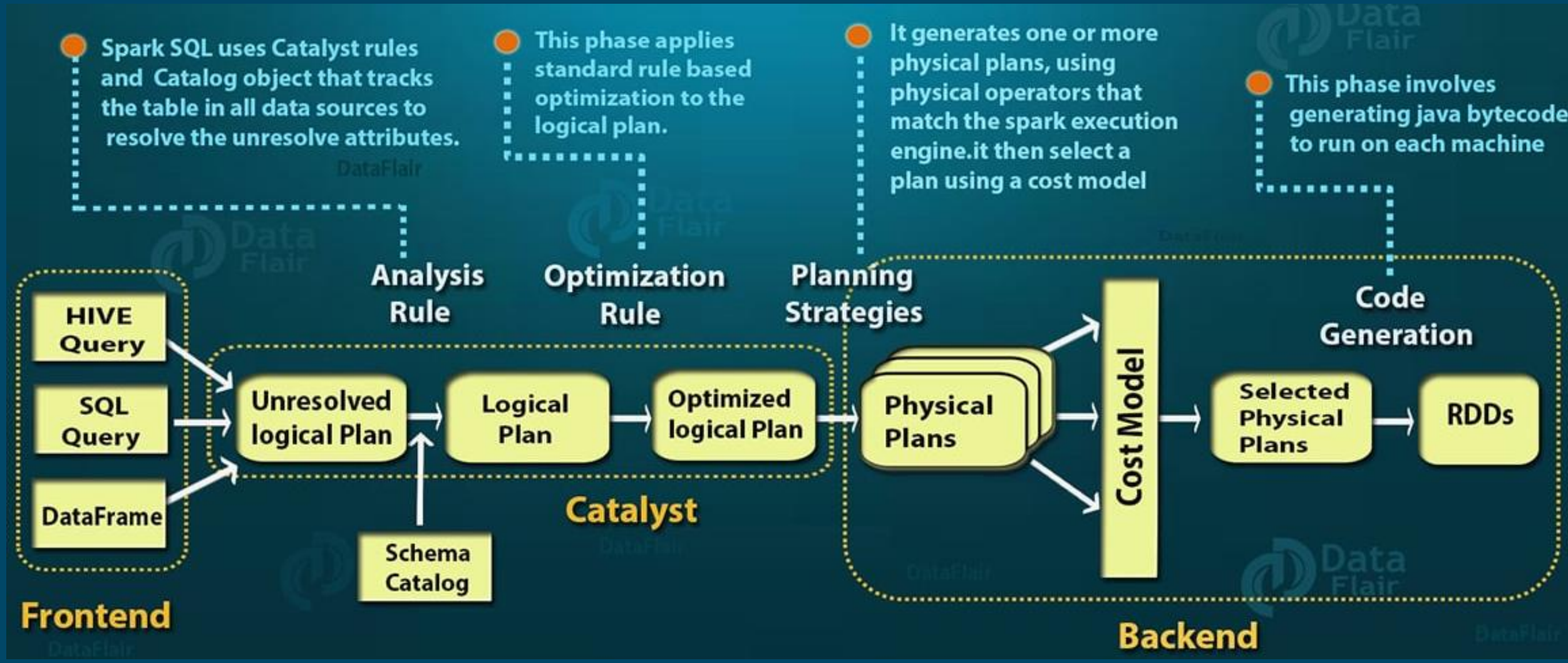


DataFrame optimization

Optimization makes the program run faster.

Another reason to prefer DataFrame to RDD

Spark Catalyst Optimizer

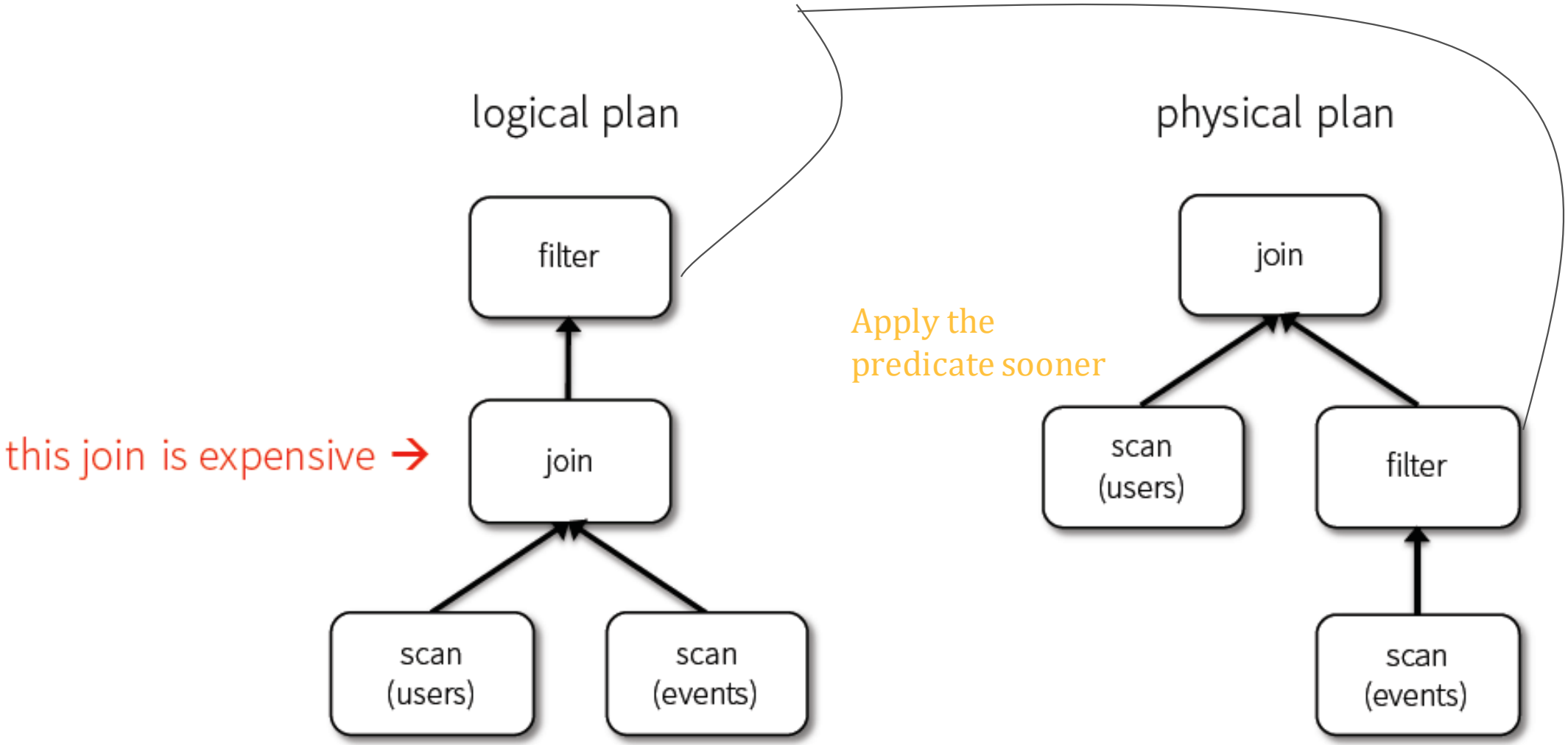


Plan Optimization steps

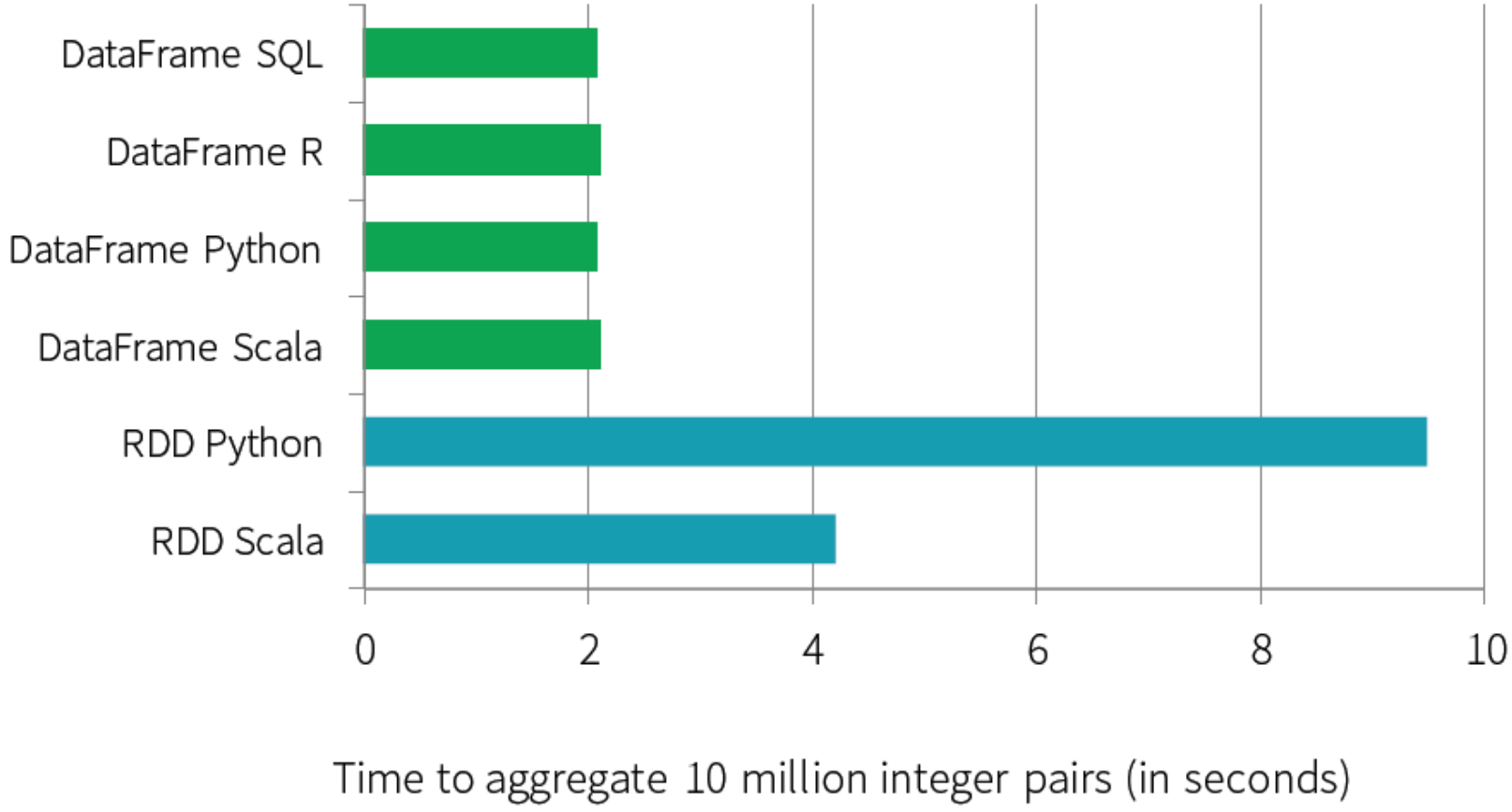
- The logical plan only represents a set of abstract transformations that do not refer to executors or drivers
- Spark uses the catalog, a repository of all table and DataFrame information, to resolve columns and tables in the analyzer
  - An unresolved logical plan may be rejected if the required table or column name does not exist in the catalog
- Catalyst Optimizer relies on a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections
  - E.g., Filter tables before join (rather than the reverse)
- A physical plan is selected that is optimal based on the cluster, data distribution, etc.

Plan Optimization & Execution

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



DataFrames can be *significantly* faster than RDDs.
And they perform the same, regardless of language.



- The DataFrame API is more efficient, because
- it can optimize the underlying operations with Catalyst.

Important to remember

- DataFrame is similar to Pandas DataFrame
- Functions are similar to SQL
  - Apply equally to DataFrame and SQL table
- DataFrame operations are optimized over the underlying RDD
  - Always prefer DataFrame over RDD

Common DataFrame API References

FYI reference

https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.html

DataFrame Functions

Some slides from
Intro to DataFrames and Spark SQL
databricks July, 2015

Construct a DataFrame

```python
# Construct a DataFrame from a "users" table in Hive.
df = sqlContext.table("users")

# Construct a DataFrame from a log file in S3.
df = sqlContext.load("s3n://someBucket/path/to/data.json", "json")
```

```scala
val people = sqlContext.read.parquet("...")
```

```java
DataFrame people = sqlContext.read().parquet("...")
```

Use DataFrames

```python
# Create a new DataFrame that contains only "young" users
young = users.filter(users["age"] < 21)

# Alternatively, using a Pandas-like syntax
young = users[users.age < 21]

# Increment everybody's age by 1
young.select(young["name"], young["age"] + 1)

# Count the number of young users by gender
young.groupBy("gender").count()

# Join young users with another DataFrame, logs
young.join(log, logs["userId"] == users["userId"], "left_outer")
```

DataFrames and Spark SQL

```
young.registerTempTable("young")
sqlContext.sql("SELECT  count(*)  FROM young")
```

- Spark SQLContext is a rich subset of SQL 92

All Actions on a DataFrame

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.

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.

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
```

https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame

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.

CSV file read illustrated

A brief look at *spark-csv*

With *spark-csv*, we can simply create a DataFrame directly from our CSV file.

```scala
// Scala
val df = sqlContext.read.format("com.databricks.spark.csv").
                         option("header", "true").
                         load("people.csv")
```

```python
# Python
df = sqlContext.read.format("com.databricks.spark.csv").\
                    load("people.csv", header="true")
```

Reading file formats

Columns

Input Source Format	Data Frame Variable Name	Data
JSON	`dataFrame1`	`[ {"first": "Amy", "last": "Bello", "age": 29 }, {"first": "Ravi", "last": "Agarwal", "age": 33 }, … ]`

CSV	`dataFrame2`	`first,last,age`
		`Fred,Hoover,91`
		`Joaquin,Hernandez,24`
		…

SQL Table	`dataFrame3`	first last age Joe Smith 42 Jill Jones 33

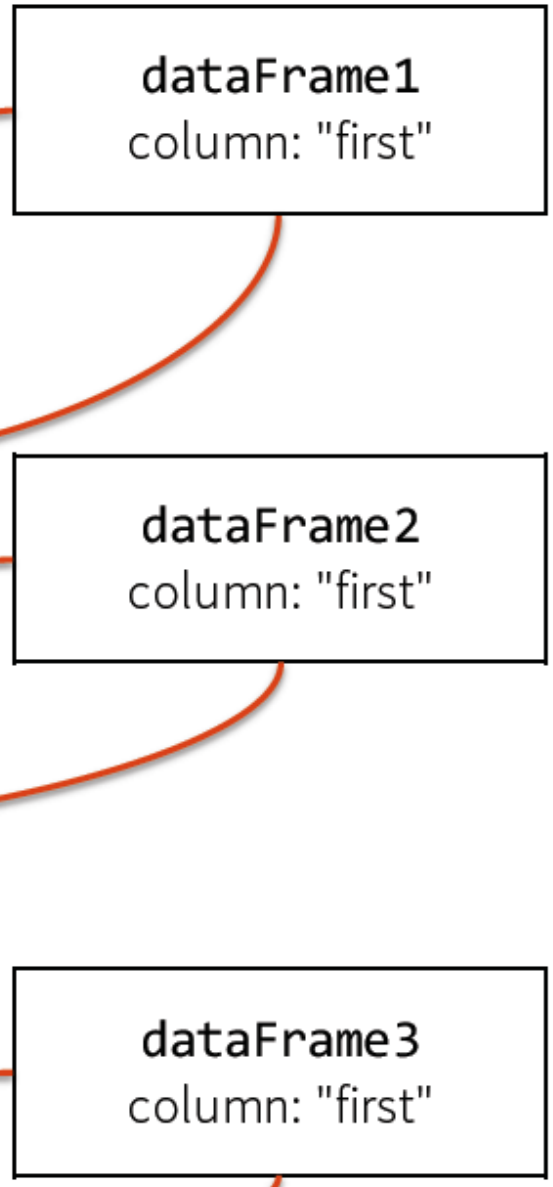
Let's see how DataFrame columns map onto some common data sources.

Columns

Input Source Format	Data Frame Variable Name	Data
JSON	dataFrame1	`[ {"first": "Amy", "last": "Bello", "age": 29 }, {"first": "Ravi", "last": "Agarwal", "age": 33 }, … ]`

CSV	dataFrame2	`first,last,age`
		`Fred,Hoover,91`
		`Joaquin,Hernandez,24`
		…

SQL Table	dataFrame3	first | last | age Joe | Smith | 42 Jill | Jones | 33



Columns

Assume we have a DataFrame, df, that reads a data source that has "first", "last", and "age" columns.

Python	Java	Scala	R
`df["first"]` `df.first`†	`df.col("first")`	`df("first")` `$"first"`‡	`df$first`

df.first: dot notation is **not** recommended because future API may change

Writing a DataFrame

Writing DataFrames

```
scala> df.write.format("json").save("/path/to/directory")
scala> df.write.format("parquet").save("/path/to/directory")
```

```
In [20]: df.write.format("json").save("/path/to/directory")
In [21]: df.write.format("parquet").save("/path/to/directory")
```

DataFrame SQL examples

select()

The Python DSL is slightly different.

```
In[1]: df.select(df['first_name'], df['age'], df['age'] > 49).show(5)
+----------+---+----------+
|first_name|age|(age > 49)|
+----------+---+----------+
|      Erin| 42|     false|
|    Claire| 23|     false|
|    Norman| 81|      true|
|    Miguel| 64|      true|
|  Rosalita| 14|     false|
+----------+---+----------+
```

select()

And, of course, you can also use SQL. (This is the Python API, but you issue SQL the same way in Scala and Java.)

```
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|
+----------+---+-----+
```

This is very common because you can first run your queries in %sql command block, and then copy them into Python code, as df.sql(query)

In a Databricks cell, you can replace the second line with:

```
%sql SELECT first_name,  age, age > 49 FROM names
```

filter()

Here's the Python version.

```
In[1]: df.filter(df['age'] > 49).\
           select(df['first_name'], df['age']).\
           show()
+---------+---+
|firstName|age|
+---------+---+
|   Norman| 81|
|   Miguel| 64|
|  Abigail| 75|
+---------+---+
```

orderBy()

And, in Python:

```
In [1]: df.filter(df['age'] > 49).\
            select(df['first_name'], df['age']).\
            orderBy(df['age'].desc(), df['first_name']).show()
+----------+---+
|first_name|age|
+----------+---+
|    Norman| 81|
|   Abigail| 75|
|    Miguel| 64|
+----------+---+
```

groupBy()

Often used with `count()`, `groupBy()` groups data items by a specific column value.

```
In [5]: df.groupBy("age").count().show()
+---+-----+
|age|count|
+---+-----+
| 39|    1|
| 42|    2|
| 64|    1|
| 75|    1|
| 81|    1|
| 14|    1|
| 23|    2|
+---+-----+
```

as() or alias()

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

```
In [7]: df.select(df['first_name'],\
                  df['age'],\
                  (df['age'] < 30).alias('young')).show(5)
+----------+---+-----+
|first_name|age|young|
+----------+---+-----+
|      Erin| 42|false|
|    Claire| 23| true|
|    Norman| 81|false|
|    Miguel| 64|false|
|  Rosalita| 14| true|
+----------+---+-----+
```

Note: In Python, you *must* use `alias`, because `as` is a keyword.

Joins

Let's assume we have a second file, a JSON file that contains records like this:

```
[
  {
    "firstName":  "Erin",
    "lastName":  "Shannon",
    "medium":  "oil on canvas"
  },
  {
    "firstName":  "Norman",
    "lastName":  "Lockwood",
    "medium":  "metal (sculpture)"
  },
  …
]
```

Joins

We can load that into a second DataFrame and join it with our first one.

```
In [1]: df2 = sqlContext.read.json("artists.json")
# Schema inferred as DataFrame[firstName: string, lastName: string, medium:
string]
In [2]: df.join(
            df2,
            df.first_name == df2.firstName and df.last_name == df2.lastName
        ).show()
+---------+---------+------+---+---------+--------+----------------+
|first_name|last_name|gender|age|firstName|lastName|          medium|
+---------+---------+------+---+---------+--------+----------------+
|    Norman| Lockwood|     M| 81|   Norman|Lockwood|metal (sculpture)|
|      Erin|  Shannon|     F| 42|     Erin| Shannon|    oil on canvas|
|  Rosalita|  Ramirez|     F| 14| Rosalita| Ramirez|         charcoal|
|    Miguel|     Ruiz|     M| 64|   Miguel|    Ruiz|    oil on canvas|
+---------+---------+------+---+---------+--------+----------------+
```

User defined functions for DataFrame/SQL

User Defined Functions

Suppose our JSON data file capitalizes the names differently than our first data file. The obvious solution is to force all names to lower case before joining.

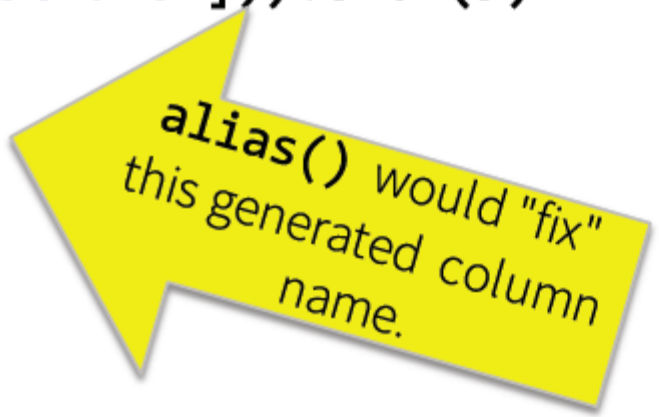
Alas, there is no `lower()` function…

```
In[6]: df3 = df.join(df2, lower(df.first_name) == lower(df2.firstName) and \
                          lower(df.last_name) == lower(df2.lastName))
NameError: name 'lower' is not defined
```

User Defined Functions

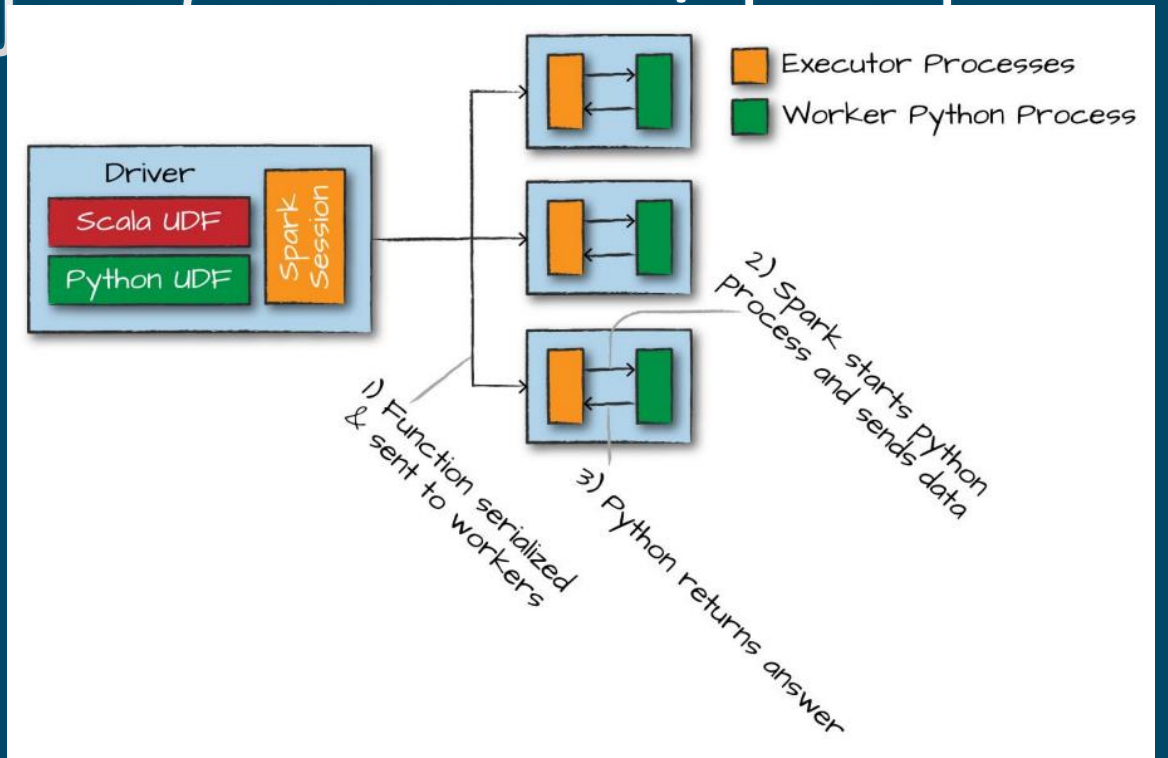
However, this deficiency is easily remedied with a *user defined function*.

```
In [8]: from pyspark.sql.functions import udf
In [9]: lower = udf(lambda s: s.lower())
In [10]: df.select(lower(df['firstName'])).show(5)
+------------------------------+
|PythonUDF#<lambda>(first_name)|
+------------------------------+
|                          erin|
|                        claire|
|                        norman|
|                        miguel|
|                      rosalita|
+------------------------------+
```



PySpark UDF's communicate with Python

- Serializing and deserializing data with JVM is slow
  - This occurs with **each row**
- Write UDFs in Scala or J



Aggregation function example

Write Less Code: Compute an Average

Using RDDs

```
var data = sc.textFile(...).split("\t")
data.map { x => (x(0), (x(1), 1)) }
    .reduceByKey { case (x, y) =>
      (x._1 + y._1, x._2 + y._2) }
    .map { x => (x._1, x._2(0) / x._2(1)) }
    .collect()
```

Full API Docs

- Scala
- Java
- Python
- R

Using DataFrames

```
sqlContext.table("people")
          .groupBy("name")
          .agg("name", avg("age"))
          .collect()
```

Using SQL

```
SELECT name, avg(age)
FROM people
GROUP BY name
```

Using Pig

```
P = load '/people' as (name, name);
G = group P by name;
R = foreach G generate … AVG(G.age);
```