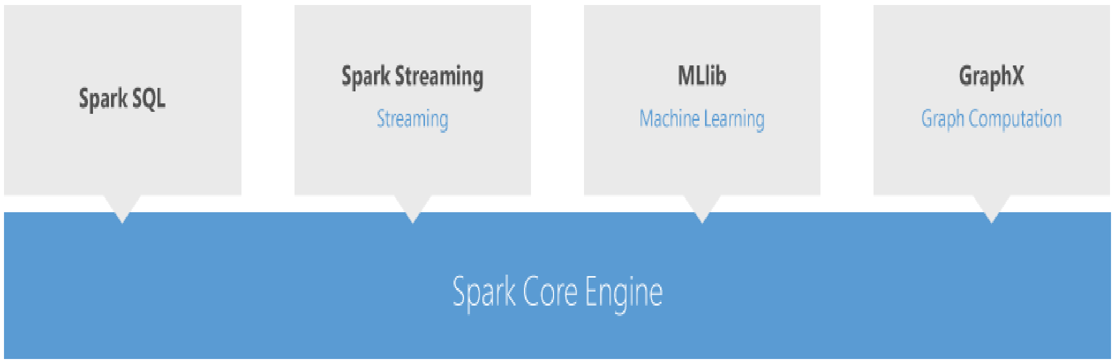
Spark Dataframes

Spark Short Introduction 1/2

- Apache Spark, is a very fast optimized engine that offers APIS in Java, Scala, Python,R and .NET.
- It can run standalone or over Hadoop or Mesos and access data sources like HDFS, Cassandra, and HBase.

Spark Short Introduction 2/2

The core system of Spark consists of different libraries and components that provide a rich set of higher tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for processing graphs and Spark Streaming



Apache Spark ecosystem

Apache Spark Core 1/2

The main block of Spark Core Engine is the Resilient Distributed Dataset (RDD).

- The term resilient means that if the dataset is entirely missing or partially damaged, Spark can recover the computation of data by retrieving them the memory and recompute them.

- The term distributed means that the dataset doesn’t have to be set on specific node of the cluster, but it can reside on any node.

- The term dataset means a collection (set) of data.

Apache Spark Core 2/2

The creation of RDDs can be achieved in three ways

- by reading from a storage source
- by using an in-memory collection
- by transforming an existing RDD

Two types of operations are supported in RDDs, the transformations, and actions.

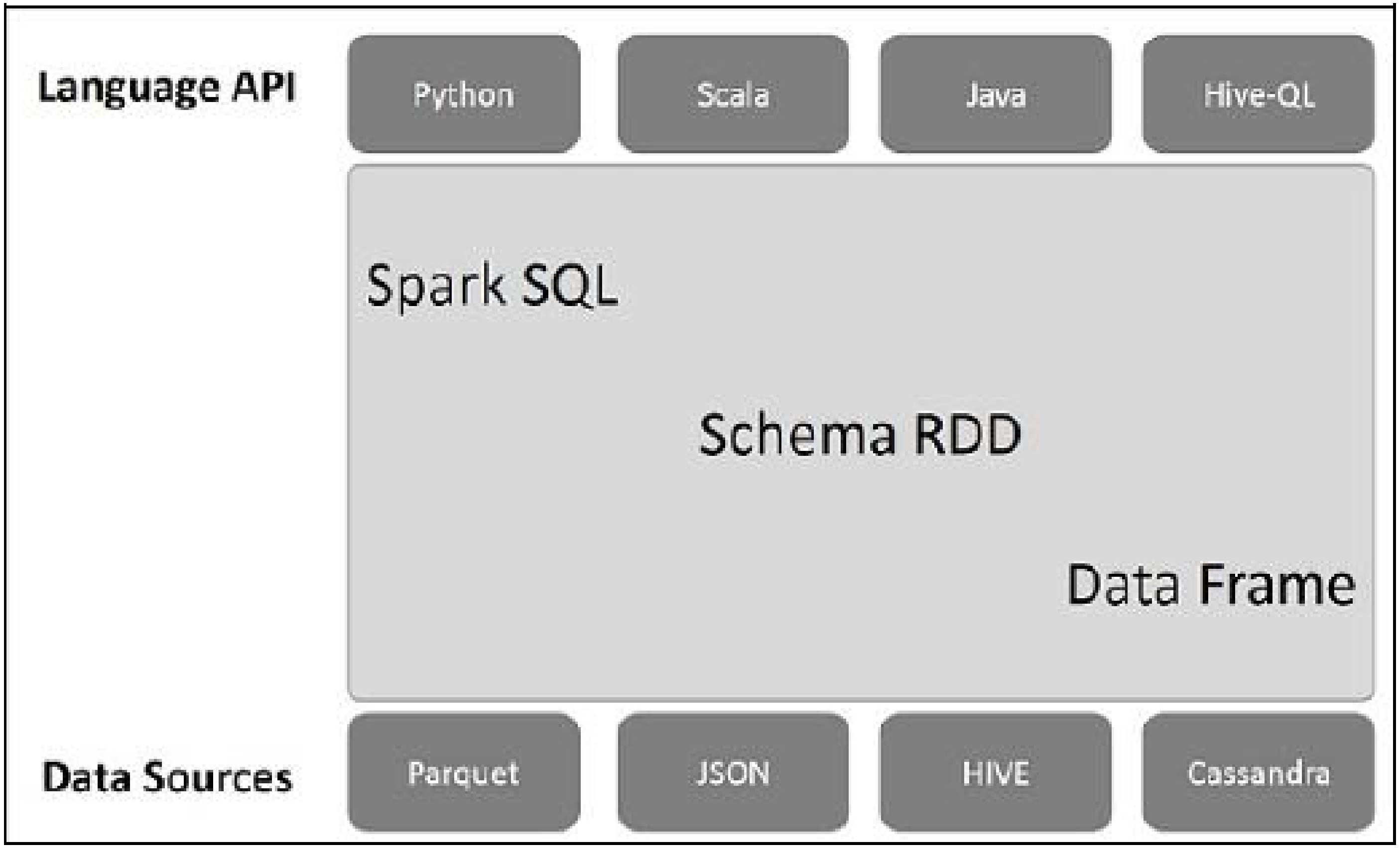
1. In transformations an existing RDD is changed to a new one, transformed RDD. In case a failure occurs, the RDD is rebuilded by the data lineage of transformations.

2. In actions, an RDD triggers a Spark job and returns a value. The actions result in a Directed Acyclic Graph (DAG) operation

Spark Dataframes Introduction 1/4

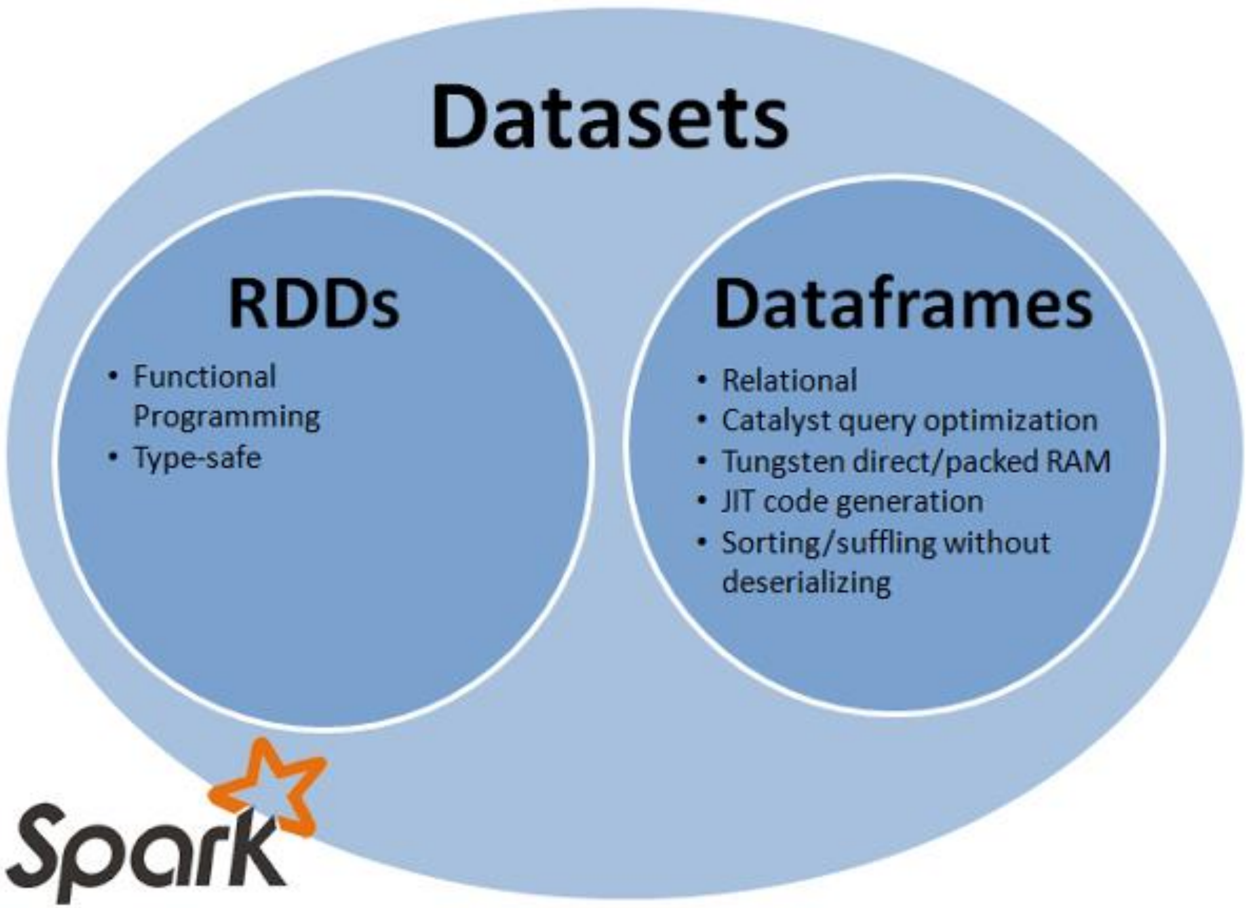
- Spark DataFrames are the standard way of dealing with data for Scala and Spark
- Spark is moving away from the RDD syntax in favor of a simpler to understand DataFrame syntax
- Spark DataFrames are also now the standard way of using Spark's Machine Learning Capabilities
- An extension to RDD API

Spark Dataframes Introduction 2/4



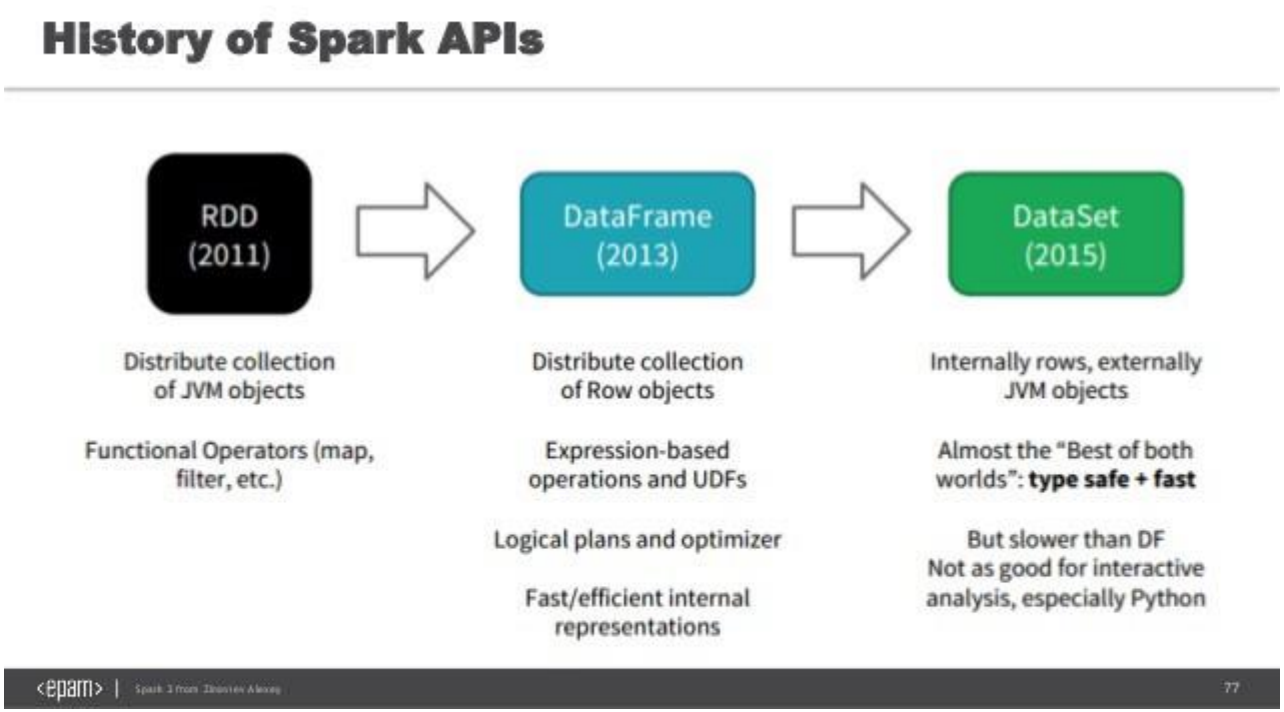
Spark SQL architecture ecosystem

Spark Dataframes Introduction 3/4



The goal of Project Tungsten is to improve Spark execution by optimizing Spark jobs for CPU and memory efficiency (as opposed to network and disk I/O which are considered fast enough).

Spark Dataframes Introduction 4/4



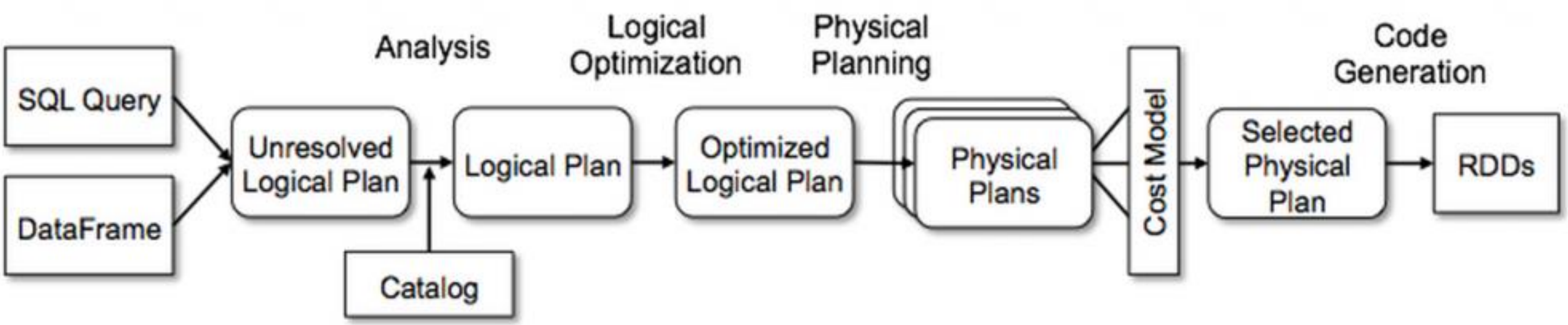
https://image.slidesharecdn.com/spark2zinovievforit-subbotnik-161028213227/95/joker16-spark-2-api-changes-structured-streaming-encoders-77-638.jpg?cb=1483965803

Features of DataFrames 1/2

- Ability to **scale** from kilobytes of data on 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**
- Seamless integration with all big data tooling and infrastructure via Spark
- APIs for **Python, Java, Scala, R and .NET**

Features of DataFrames 2/2

Spark SQL Catalyst optimizer



DataFrames and Spark SQL

- DataFrames are fundamentally tied to Spark SQL.
- The DataFrames API provides a *programmatistic* interface for interacting with your data.
- Spark SQL provides **a *SQL-like* interface**.
- What you can do in **Spark SQL**, you can do in DataFrames

What, exactly, is Spark SQL?

- Spark SQL allows you to manipulate distributed data with SQL queries.

  Currently, two SQL dialects are supported:

- If you're using a Spark `SQLContext`, the only supported dialect is "sql", a rich subset of SQL 92

- If you're using a `HiveContext`, the default dialect is "hiveql", corresponding to Hive's SQL dialect. "sql" is also available, but "hiveql" is a richer dialect

Spark SQL

- You issue SQL queries through a `SQLContext` method.
- The `sql()` method returns a `DataFrame`
- You can mix DataFrame methods and SQL queries in the same code
- To use SQL, you *must* :

  - make a table alias for a DataFrame, using `registerTempTable()`

  - or to create a temporary view using `createOrReplaceTempView()`

DataFrames

- Like Spark SQL, the DataFrames API assumes that the data has a **table-like structure**.

- Formally, a DataFrame is a size-mutable, potentially heterogeneous tabular data structure with rows and columns.

- Just think of it as a table in a distributed database: a distributed collection of data organized into named, typed columns.

Transformations, Actions, Laziness

DataFrames are *lazy*.

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

***Actions* cause the execution of the query.**

Transformation examples

- filter
- select
- drop
- intersect
- join

Action examples

- count
- collect
- show
- head
- take

Transformations, Actions, Laziness

Actions cause the execution of the query.

What, exactly does "**execution of the query"** mean?

It 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.

All Actions on a DataFrame 1/3

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.

All Actions on a DataFrame 2/3

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.

All Actions on a DataFrame 3/3

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
```

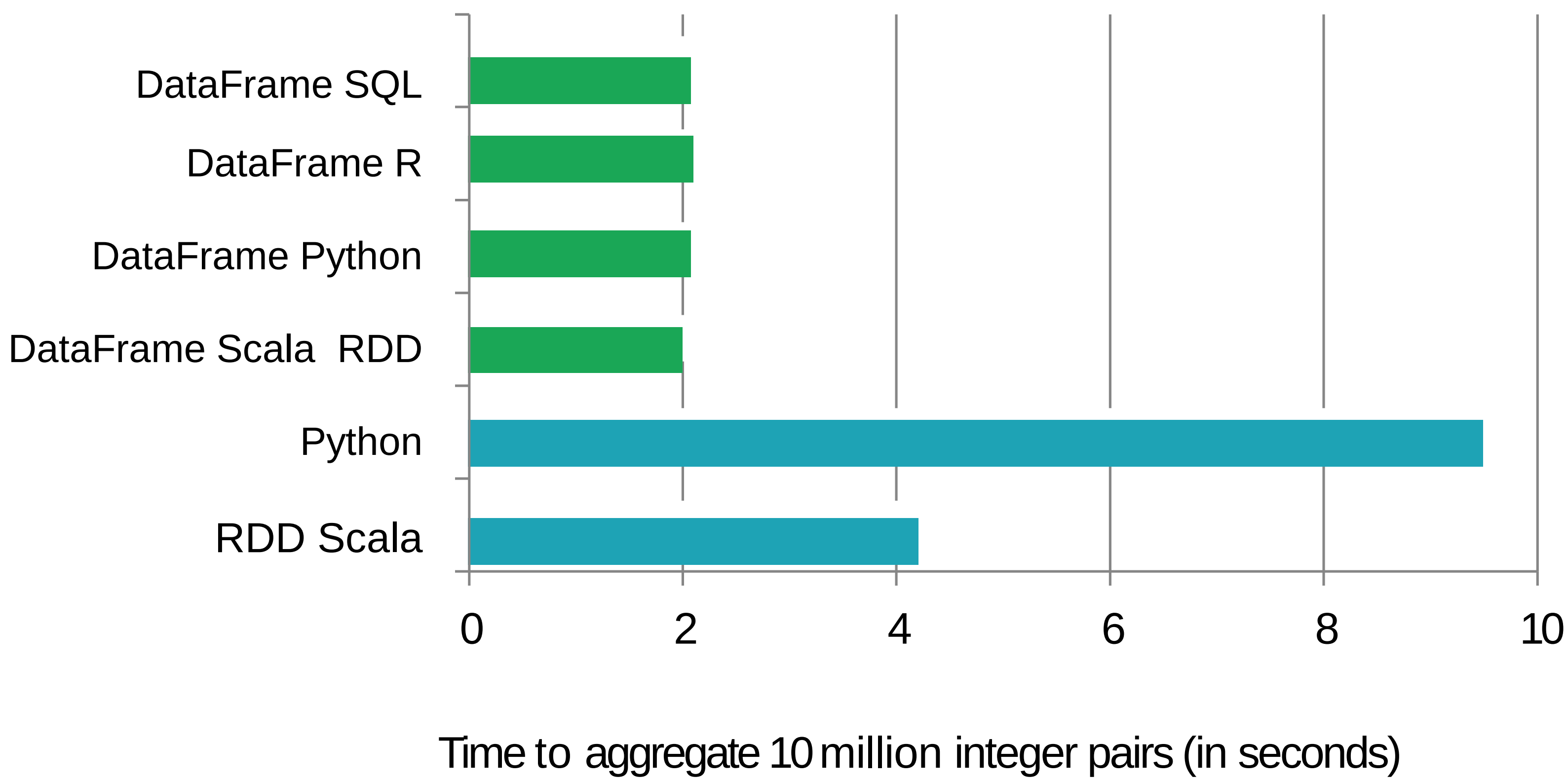
DataFrames & Resilient Distributed Datasets (RDDs) 1/2

- DataFrames are built on top of the Spark RDD API.
  - This means you can use **normal RDD operations** on DataFrames.

- However, **stick** with the DataFrame API, wherever possible.
  - Using RDD operations will often give you back an RDD, not a DataFrame.
  - The DataFrame API is likely to be **more efficient**, because it can **optimize** the underlying operations with Catalyst.

DataFrames & Resilient Distributed Datasets (RDDs) 2/2

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



Creating a DataFrame

- You create a DataFrame with a `SQLContext` object (or one of its descendants)
- In the Spark Scala shell (*spark-shell*) you have a `SQLContext` available automatically, as `sqlContext`.
- In an application, you can easily create one yourself, from a `SparkContext`.
- The DataFrame *data source API* is the same, across data formats.

Creating a DataFrame in Scala

```scala
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.SQLContext

val conf = new SparkConf().setAppName(appName).
                          setMaster(master)
// Returns existing SparkContext, if there is one
// otherwise, creates a new one from the config.
val sc = SparkContext.getOrCreate(conf)
//
val sqlContext = SQLContext.getOrCreate(sc)

val df =  sqlContext.read.json("/path/to/data.json")
```

Data Sources supported by DataFrames

built-in

Parquet

JDBC

{ JSON }

PostgreSQL

HIVE

MySQL

HDFS

amazon web services S3

H2

external

AVRO

CSV

dBase

APACHE HBASE

elasticsearch.

cassandra

amazon web services Amazon Redshift

and more …

Schema Inference

What if your data file doesn't have a schema? (e.g., You're reading a CSV file or a plain text file.)

- You can create an RDD of a particular type and let Spark infer the schema from that type.

- You can use the API to specify the schema programmatically.

Schema Inference Example

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 it's obvious there *is* one:

First name: *string*
Last name: *string*
Gende *string*
r: Age: *integer*

Schema Inference ::Scala

```
import sqlContext.implicits._

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
```

Schema Inference ::Scala

We can also force schema inference without creating our own People type, by using a fixed length data structure and supplying the column names to the toDF() method.

```
val rdd = sc.textFile("people.csv")
val peopleRDD = rdd.map { line =>
val cols = line.split(",")
(cols(0), cols(1), cols(2), cols(3).toInt)
}

val df = peopleRDD.toDF("firstName", "lastName","gender",
"age")
```

If you don’t supply the column names, the API defaults to “_1”, “_2”, etc.

Additional Input Formats

The DataFrames API can be extended to understand additional input formats (or, input *sources*).

For instance, if you're dealing with CSV files, a *very* common data file format, you can use the *spark-csv* package (spark-packages.org/package/databricks/spark- csv)

This package augments the DataFrames API so that it understands CSV files.

Spark installation 1/3

- Apache Spark runs on the Java Virtual Machine (JVM). The Software Development Kit (SDK) is required for building application with Spark and not the Java Runtime Environment (JRE).
- The recommended version of Java is 7 or higher. The most suitable version of Java for working with Scala and Python is 8, because of the functional programming methods are included.

```
# install oracle java 8
$ sudo apt-get install software-properties-common
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

```
# $ echo JAVA_HOME
```

Spark installation 2/3

The Spark download page is http://spark.apache.org/downloads.html. The webpage also archives earlier versions of Spark in different packages. In this project we have selected the release, pre-built for Hadoop 2.7 and later. An easy way to install Spark is to use a prebuilt package, and not building it from source. The downloaded has to be moved to the directory ~/spark under the root directory.

So the first step is to download the latest release of Spark

1. We select the latest stable Spark release
2. We choose the package type Prebuilt for Hadoop 2.7 and later,
3. We choose the download type Direct Download
4. We download the .tgz file
5. We verify this release using the appropriate signatures and checksums

Spark installation 3/3

- It is essential also to install Eclipse IDE which is a development environment commonly used for creating Java applications.

- The installation of the Eclipse IDE is a straightforward procedure. The program is available to download via the following link:

http://www.eclipse.org/downloads/eclipse-packages/?osType=linux&release=undefined

We choose the Linux 64 Bit version and save the tarball file to a local folder named eclipse.

A brief look at *spark-csv 1/3*

Let’s assume our data file has a header:

```
first_name,last_name,gender,age
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
…
```

A brief look at *spark-csv 2/3*

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

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

spark-csv uses the header to infer the schema, but the column types will always be *string*.

```
df: org.apache.spark.sql.DataFrame = [first_name: string,
last_name: string, gender: string, age: string]
```

A brief look at *spark-csv* 3/3

You can also declare the schema programmatically, which allows you to specify the column types.

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

// A schema is a StructType, built from a List of StructField  objects.
val schema =  StructType(
  StructField("firstName", StringType, false) ::  StructField("gender",
                     StringType, false) ::  StructField("age",
                     IntegerType, false) ::
  )

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

Columns 1/3

A DataFrame *column* is an abstraction.

It provides a common column-oriented view of the underlying data, *regardless* of how the data is really organized.

Columns 2/3

<table>
<tr><th>Input Source Format</th><th>Data Frame Variable Name</th><th>Data</th></tr>
<tr><td>JSON</td><td><code>dataFrame1</code></td><td><pre>[
{"first": "Amy",
"last":   "Bello",
"age":   29 },
{"first": "Ravi",
"last":   "Agarwal",
"age":   33 },
…
]</pre></td></tr>
<tr><td>CSV</td><td><code>dataFrame2</code></td><td><pre>first,last,age
Fred,Hoover,91
Joaquin,Hernandez,24
…</pre></td></tr>
<tr><td>SQL Table</td><td><code>dataFrame3</code></td><td><table>
<tr><th>first</th><th>last</th><th>age</th></tr>
<tr><td>Joe</td><td>Smith</td><td>42</td></tr>
<tr><td>Jill</td><td>Jones</td><td>33</td></tr>
</table></td></tr>
</table>

how DataFrame columns map onto some common data sources.

Columns 3/3

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	<table><tr><th>first</th><th>last</th><th>age</th></tr><tr><td>Joe</td><td>Smith</td><td>42</td></tr><tr><td>Jill</td><td>Jones</td><td>33</td></tr></table>

dataFrame1
column: "first"

dataFrame2
column: "first"

dataFrame3
column: "first"

printSchema()

You can have Spark respond 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)
```

show()

You can look at the first *n* elements in a DataFrame with the `show()` method.
If not specified, *n* defaults to 20.

This method is an *action* - It:

- reads (or re-reads) the input source
- executes the RDD DAG across the cluster
- pulls the *n* elements back to the driver JVM
- displays those elements in a tabular form

show()

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

select()

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

```
scala> df.select($"firstName", $"age").show(5)
+--------------------+------+
|           firstName|   age|

|                Erin|    42|
|              Claire|    23|
|              Norman|    81|
|              Miguel|    64|
|            Rosalita|    14|
+--------------------+------+
```

select()

The select() also allows you create on-the-fly *derived columns*.

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

select()

And, of course, you can also use SQL.

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

filter()

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

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

filter()

The SQL version.

```
In[1]: SQLContext.sql("SELECT  first_name, age FROM names " + \
                         "WHERE age > 49").show()

+------------------+------+
|      firstName   |age|
+------------------+------+
|      Norman      | 81|
|      Miguel      | 64|
|      Abigail     | 75|
+------------------+------+
```

orderBy()

The `orderBy()` method allows you to sort the results.

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

orderBy()

It's easy to reverse the sort order.

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

orderBy()

In SQL:

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

as() or alias()

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

Other Useful Transformations

Method	Description
`limit(`*n*`)`	Limit the results to *n* rows. `limit()` is not an action, like `show()` or the RDD `take()` method. It returns another DataFrame.
`distinct()`	Returns a new DataFrame containing only the unique rows from the current DataFrame
`drop`*(column)*	Returns a new DataFrame with a column dropped. *column* is a name or a Column object.
`intersect(`*dataframe*`)`	Intersect one DataFrame with another.
`join(`*dataframe*`)`	Join one DataFrame with another, like a SQL join.

Writing DataFrames

- You can write DataFrames out, as well.
- In most cases, if you can read a data format, you can write that data format, as well.
- If you're writing to a text file format (e.g., JSON), you'll typically get multiple output files.

Writing DataFrames

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