

# DataFrames and SparkSQL

Prof. Dr. Stephan Trahasch  
Hochschule Offenburg

# Outline

---

- DataFrames, Datasets and Spark SQL
- Operations on DataFrames

# History of Spark APIs



- Distribute collection of JVM objects
- Functional Operators (map, filter, etc.)

- DataFrames are dynamically typed
- Distribute collection of Row objects
- Expression-based operations and UDFs
- Logical plans and optimizer
- Fast/efficient internal representations

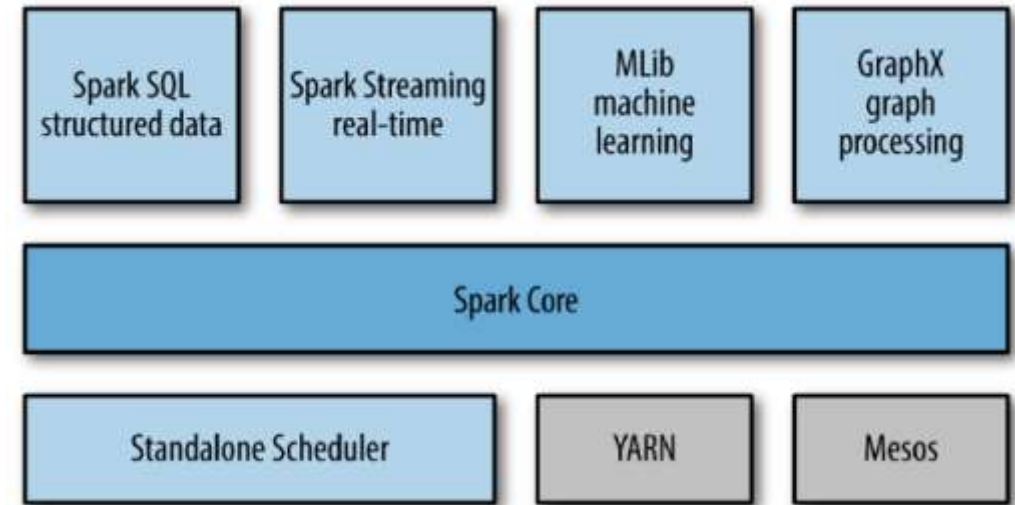
- Datasets: statically typed
- Internally rows, externally JVM objects
- “Best of both worlds” type safe + fast
- No Python support

# Spark SQL

Part of the core distribution since 1.0 (April 2014)

What does Spark SQL provide?

- A SQL Engine and command line interface
- You can interact with the SQL interface using the command-line or over JDBC/ODBC.
- Spark SQL can also be used to read data from an existing Hive installation
- Runs SQL / HiveQL queries, optionally alongside or replacing existing Hive deployments
- CatalystOptimizer – an extensible optimization framework
- Usually the Data source for spark-core is a text file, Avro file, etc.  
Data Sources for Spark SQL are Parquet, JSON, HIVE tables, Cassandra database.



# DataFrames and Spark SQL

- DataFrames are fundamentally tied to Spark SQL
- The DataFrames API provides a programmatic interface, a domain-specific language (DSL) - for interacting with your data.
- Spark SQL provides a SQL-like interface.
- Anything you can do in Spark SQL, you can do in DataFrames ... and vice versa

To use SQL, you must either:

- query a persisted Hive table, or
- make a table alias for a DataFrame, using `registerTempTable()` (later more)

# DataFrames

- Enable wider audiences beyond “Big Data” engineers to leverage the power of distributed processing
- Inspired by data frames in R and Python (Pandas)
- Designed from the ground-up to support modern big data and data science applications
- Extension to the existing RDD API, APIs for Python, Java, Scala, and R
- State-of-the-art optimization and code generation through the Spark SQL Catalyst optimizer

See <https://spark.apache.org/docs/latest/sql-programming-guide.html>

[databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html](https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html)

# Creating a DataFrame

## Entry point to all functionality in Spark is the SparkSession

- You create a DataFrame with a SQLContext object (or one of its descendants)
- In the Spark Scala shell (spark-shell) or pyspark, you have a SQLContext available automatically, as sqlContext. From 2.x you use only SparkSession.
- In an application you can create one yourself from a SparkContext.
- The DataFrame data source API is consistent, across data formats.
- “Opening” a data source works pretty much the same way, no matter what.

```
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._
```

# How to create a DataFrame

DataFrames can be created

- From an existing structured data source (Parquet file, JSON file, etc.)  
<https://docs.databricks.com/spark/latest/data-sources/index.html>
- From a Hive table
- From an existing RDD
- By performing an operation or query on another DataFrame
- By programmatically defining a schema



# Creating a DataFrame in Scala

Program: including setup; the DataFrame reads are 1 line each

```
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 spark = SparkContext.getOrCreate(conf)

// Ditto
val sqlContext = SQLContext.getOrCreate(sc) // not necessary any more.

val df = spark.read.parquet("/path/to/data.parquet")
val df2 = spark.read.json("/path/to/data.json")
```

# Creating a DataFrame in Scala - json

File: people.json

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

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

# Load from JDBC

```
// Note: JDBC loading and saving can be achieved via either the load/save or jdbc methods
// Loading data from a JDBC source
val jdbcDF = spark.read
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "schema.tablename") // SQL Queries possible!
  .option("user", "username")
  .option("password", "password")
  .load()

val connectionProperties = new Properties()
connectionProperties.put("user", "username")
connectionProperties.put("password", "password")
val jdbcDF2 = spark.read
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties)
```

# Unified interface to reading/writing data in a variety of formats.

```
val df = spark
    .read
    .format("json")
    .option("samplingRatio", "0.1")
    .load("/Users/spark/data/stuff.json")

df.write
    .format("parquet")
    .mode("append")
    .partitionBy("year")
    .saveAsTable("faster-stuff")
```

# Creating Datasets

```
// Note: Case classes in Scala 2.10 can support only up to 22 fields. To work around this limit,
// you can use custom classes that implement the Product interface
case class Person(name: String, age: Long)

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

// 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()
// +-----+-----+
//      ...
```

## DataFrames have Schemas

- In the previous example, we created DataFrames from Parquet and JSON data.
- A Parquet table has a schema (column names and types) that Spark can use.
- Parquet also allows Spark to be efficient about how it pares down data.
- Spark can infer a Schema from a JSON file.
- Some data sources can expose a formal schema; others (e.g., plain text files) don't.

How do we fix that?

- You can create an RDD of a particular type and let Spark infer the schema from that type. We'll see how to do that in a moment.
- 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  
...
```

<b>First name:</b>	<b>string</b>
<b>Last name:</b>	<b>string</b>
<b>Gender:</b>	<b>string</b>
<b>Age:</b>	<b>integer</b>

The file has no schema, but it's obvious there is one.

# Schema Inference

```
import spark.implicits._

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

val rdd = spark.read.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]
```



# Schema Inference

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

```
val rdd = spark.read.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.

# Inferring the Schema Using Reflection

```
// 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.read
    .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")

// The columns of a row in the result can be accessed by field index or by name
// 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")
```

# Programmatically Specifying the Schema

```
import org.apache.spark.sql.types._
val peopleRDD = spark.sparkContext.textFile("examples/src/main/resources/people.txt") // Create RDD

// 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)

// 2. Convert records of the RDD (people) to Rows
val rowRDD = peopleRDD.map(_._split(",")).map(attributes => Row(attributes(0), attributes(1).trim))

// 3. 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")
```

# Outline

---

- DataFrames and Datasets
- Operations on DataFrames

# DataFrame: Columns

- 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 are important because much of the DataFrame API consists of functions that take or return 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	<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>	first	last	age	Joe	Smith	42	Jill	Jones	33
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
<code>df["first"]</code> <code>df.first<sup>†</sup></code>	<code>df.col("first")</code>	<code>df("first")</code> <code>\$"first"</code>	<code>df\$first</code>

# 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

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

## Basic Operations deal with DataFrame metadata

- `schema` – returns a Schema object describing the data
- `printSchema` – displays the schema as a visual tree
- `cache / persist` – persists the DataFrame to disk or memory
- `columns` – returns an array containing the names of the columns
- `dtypes` – returns an array of (column-name, type) pairs
- `explain` – prints debug information about the DataFrame to the console



## Actions (similar to RDDs)

- collect – return all rows as an array of Row objects
- take(n) – return the first n rows as an array of Row objects
- count – return the number of rows
- show(n) – display the first n rows (default=20)

# Queries

DataFrame query methods return new DataFrames

Queries can be chained like transformations

Some query methods

- `distinct` – returns a new DataFrame with distinct elements of this DF
- `join` – joins this DataFrame with a second DataFrame
- `limit` – a new DF with the first `n` rows of this DataFrame
- `select` – a new DataFrame with data from one or more columns of the base DataFrame
- `filter` – a new DataFrame with rows meeting a specified condition
- `where ...`

# Aggregations

- DataFrames functions provide common aggregations such as `count()`, `countDistinct()`, `avg()`, `max()`, `min()`, etc.
- Documentation at [https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions\\$](https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$)
- While those functions are designed for DataFrames, Spark SQL also has type-safe versions for some of them in Scala and Java to work with strongly typed Datasets.
- You can also create your own function
  - Untyped User-Defined Aggregate Functions
  - Type-Safe User-Defined Aggregate Functions

# Temporary and global views for SQL

Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates.

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

If you want to have a view that is shared among all sessions and keep alive until the Spark application terminates, create global temporary view.

```
// Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

// Global temporary view is tied to a system
// preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
// +-----+-----+
// | age|    name|
// +-----+-----+
// |null|Michael|
// | 30|    Andy|
// ...
// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM
global_temp.people").show()
```

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

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

# filter()

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

```
scala> df.filter($"age" > 49).
        select($"first_name", $"age").show()
```

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

```
scala> df.filter($"age"> 49).select($"firstName", $"age").show()
```

```
+-----+-----+
|firstName|age|
+-----+-----+
|   Norman| 81|
|   Miguel| 64|
|  Abigail| v75|
+-----+-----+
```

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

```
scala> df.filter($"age" > 49).
      select($"firstName", $"age").
      orderBy($"age",
              $"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|
+-----+----+
```

## as() or alias()

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

```
scala> df.select($"firstName", $"age", ($"age" < 30).as("young")).show()
```

```
+-----+---+-----+
|firstName|age|young|
+-----+---+-----+
|    Erin| 42|false|
|  Claire| 23| true|
|   Norman| 81|false|
|  Miguel| 64|false|
|Rosalita| 14| true|
+-----+---+-----+
```

```
spark.sql("SELECT firstName, age, age < 30 AS young FROM names")
```

```
+-----+---+-----+
|firstName|age|young|
+-----+---+-----+
|    Erin| 42|false|
|  Claire| 23| true|
|   Norman| 81|false|
|  Miguel| 64|false|
|Rosalita| 14| true|
+-----+---+-----+
```



# groupBy()

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

```
> df.groupBy("age").count().show()
```

```
+---+-----+
|age|count|
```

```
+---+-----+
```

```
| 39|    1|
```

```
| 42|    2|
```

```
| 64|    1|
```

```
| 75|    1|
```

```
| 81|    1|
```

```
| 14|    1|
```

```
| 23|    2|
```

```
+---+-----+
```

```
scala> spark.sql("SELECT age, count(age) FROM names  
                GROUP BY age")
```

```
+---+-----+
|age|count|
```

```
+---+-----+
```

```
| 39|    1|
```

```
| 42|    2|
```

```
| 64|    1|
```

```
| 75|    1|
```

```
| 81|    1|
```

```
| 14|    1|
```

```
| 23|    2|
```

```
+---+-----+
```

# Joins

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

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

```
scala> val df2 = sqlContext.read.json("artists.json")
# Schema inferred as DataFrame[firstName: string, LastName: string, medium: string]
scala> df.join(df2, (df("first_name") === df2("firstName")) & (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

We'll create a function to extract the month field from a timestamp-typed column and return the month name.

```
scala> df.select(monthName($"time"))
console>:23: error: not found: value monthName
df.select(monthName($"time")).show()
```

```
scala> val monthName = spark.udf.register("monthName", (t: Timestamp) => {
  import java.text.SimpleDateFormat
  import java.util.Date
  val fmt = new SimpleDateFormat("MMM")
  fmt.format(new Date(t.getTime))
})
monthName: org.apache.spark.sql.UserDefinedFunction =
UserDefinedFunction(<function1>,StringType,List())

scala> df.select(monthName($"time")).show(5)
+-----+
| UDF(time) |
+-----+
|      Oct |
|      Oct |
+-----+
```

# Summary

---

- SparkSQL is another way to manipulate DataFrames

## Documentation

<https://docs.databricks.com/spark/latest/spark-sql/index.html#spark-sql-language-manual>

<https://docs.databricks.com/spark/latest/dataframes-datasets/index.html>

<https://docs.databricks.com/spark/latest/data-sources/index.html>