




# Spark SQL

Jorge Acosta Hernández

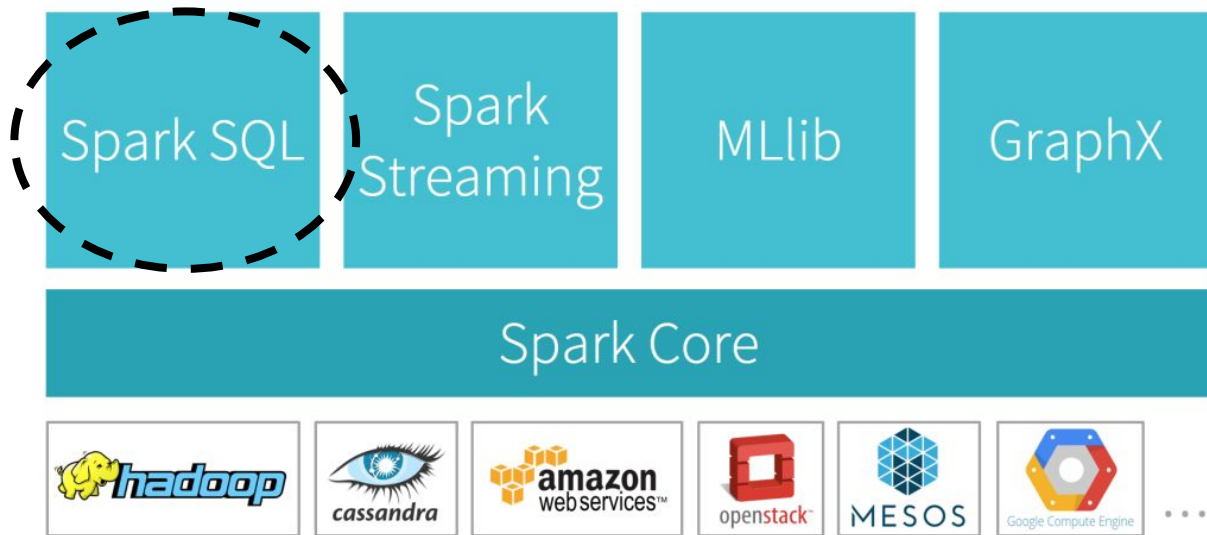
[jorge.acosta@upm.es](mailto:jorge.acosta@upm.es)

With some slides from Jesús Montes

Nov. 2024



# Spark SQL in the Spark Stack



# Spark SQL

- Spark SQL is a Spark module for structured data processing
- Compared to the Spark Core (RDDs), Spark SQL tools provide Spark with more information about structure of data and computation
- Internally, Spark SQL uses this extra information to perform optimizations
- Spark SQL is based on the Spark Core, and implements many optimized operations over RDDs

# The SQL interface

- The Spark distribution includes a SQL console (`bin/spark-sql`)
- It can be used to perform Spark SQL operations and also run Hive queries (even when you do not have a Hive server running!)
- SQL queries in Spark SQL can also be executed from inside Spark applications (written in Scala, Java, ...)
- When running SQL from within another programming language, the results will be returned as a Dataset/DataFrame
- Spark SQL also provides support for JDBC/ODBC

# The Dataset API

- Spark SQL main interface is the Dataset API
- Provides a high-level solution for programming data processing/analysis in Spark
  - Opposed to the RDD API, that can be considered low-level in “Spark terms”
- The Dataset API was introduced in Spark 1.6
  - In earlier versions, there was the DataFrame API
  - Since Spark 2.0, DataFrames are considered a particular type of Datasets
- Fully supported for Scala and Java
- Partially supported (DataFrame operations) in Python and R
- **We will be using DataFrames as the available option on PySpark**

# Project Tungsten

In 2015, an initiative called “Project Tungsten” started, to incorporate advanced optimizations in the Spark Core and DataFrame APIs (at the time, the Dataset API did not exist yet).

- Memory Management and Binary Processing: leveraging application semantics to manage memory explicitly and eliminate the overhead of JVM object model and garbage collection.
- Cache-aware computation: algorithms and data structures to exploit memory hierarchy
- Code generation: using code generation to exploit modern compilers and CPUs

Check out this blog post:

<https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html>

# Catalyst Optimizer

At the core of Spark SQL is the Catalyst optimizer, which leverages advanced programming language features in a novel way to build an extensible query optimizer.

- Includes a query planner, optimizer, and code generation, enabling efficient execution of queries and better resource management.
- Uses rule-based and cost-based optimizations to improve performance.
- Provides improved performance, cost savings, and better resource management

# Dataset

*“Datasets is the guys at Spark realizing that types matter”*

Holden Karau  
(Big Data Spain, 2018)

- A Dataset is a distributed collection of data.
- Constructed from JVM objects.
- Manipulated using functional transformations (map, flatMap, filter, ...).
- Datasets provide the benefits of RDDs...
  - Fault tolerance
  - Strong typing
  - Ability to use powerful lambda functions
  - ...
- ... and also benefit from SparkSQL's optimized execution engine.
- Datasets can be seen as a high-level, highly optimized version of RDDs.



# Dataset

- Instead of using Java serialization or Kryo, Datasets use a specialized Encoder.
  - Data processing
  - Network transmission
- Encoders are code generated dynamically and use a format that allows Spark to perform many operations like filtering, sorting and hashing without deserializing the bytes back into an object.
- Spark includes a set of implicit encoders to simplify Dataset creation (`import spark.implicits._`). For Scala and Java.

# DataFrame

- A DataFrame is a Dataset organized into named columns
  - In Scala DataFrame is an alias for Dataset[Row]
  - In Java the term DataFrame does not exist any more. We use Dataset<Row>
- Conceptually equivalent to a table in a relational database.
- DataFrames can be constructed from many sources:
  - Structured files
  - Hive tables
  - RDDs
  - ...
- DataFrames are used in many situations where organizing data as tables is useful or natural (batch processing, data analysis,...).

# SparkSession

To start with Spark SQL, we need a SparkSession.

- The SparkSession is the entry point into all functionality in Spark SQL.
  - High-level equivalent of SparkContext.
- To create a basic SparkSession:  
`SparkSession.builder()`
- SparkSession includes Hive support.

```
from pyspark.sql import SparkSession

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

# Creating and Storing a DataFrame

Datasets can be created in many ways:

- Reading local/hdfs files
  - Json
  - Parquet
  - text/csv
  - ...
- Constructed from RDDs
- Loaded from Hive tables

They can also be stored in many formats:

- Files (text, parquet,...)
- Hive tables
- ...

```
read_path = '../data/csv_example/one_file'
```

```
print('+ Read csv from file: \n')  
df_from_csv = spark.read.csv(read_path)  
df_from_csv.printSchema()  
df_from_csv.show()
```

```
print('+ Read csv with header from file: \n')  
df_from_csv = spark.read.csv(read_path,header=True)  
df_from_csv.printSchema()  
df_from_csv.show()
```

```
print('+ Read csv with header and inferring data types  
from file: \n')  
df_from_csv = spark.read.csv(read_path,header=True,  
inferSchema=True)  
df_from_csv.printSchema()  
df_from_csv.show()
```



# Operating with DataFrames

Datasets provide transformations and actions similar to RDDs:

map	sort	join
flatMap	union	reduce
filter	intersect	count
distinct	groupBy	foreach
sample	collect	collectAsList
	...	

They also have specific operations:

- printSchema

```
>>> personDF.printSchema()
root
 |-- name: string (nullable = true)
 |-- age: long (nullable = false)
```

- show

```
>>> personDF.show()
+----+----+
|name|age|
+----+----+
|Matt| 49|
|Gill| 62|
+----+----+
```

# Running SQL queries over DataFrames

- The `sql` method of the `SparkSession` object allows to interact with Spark SQL using SQL queries
  - `sql(sqlText: String): DataFrame`
- This method interprets Hive SQL.
- DataFrames can be registered as “TempViews” in the SQL context, and then accessed as tables in the SQL queries.
- Hive tables can also be used in the SQL queries.

```
personDF.createOrReplaceTempView("person")
```

```
query = spark.sql("SELECT * FROM person WHERE age > 50")  
query.show()
```

```
query = spark.sql("SELECT * FROM some_hive_table")  
query.show()
```

```
query = spark.sql("SELECT COUNT(*) FROM person")  
query.show()
```



# DataFrame operations

## Typical DataFrame operations:

- `withColumn(colName: String,col: Column): DataFrame`
- `drop(col: Column): DataFrame`
- `select(cols: Column*): DataFrame`  
`select(col: String,cols: String*): DataFrame`
- `filter(condition: Column): DataFrame`  
`filter(conditionExpr: String): DataFrame`
- `groupBy(cols: Column*): RelationalGroupedDataset`  
`groupBy(col1: String,cols: String*): RelationalGroupedDataset`



# Grouping and Aggregation

- The `groupBy` operation takes a `Dataset/DataFrame` and outputs a `RelationalGroupedDataset`.
  - An object that represents the groups in the Dataset
- The most important method of a `RelationalGroupedDataset` is `agg(expr: Column, exprs: Column*): DataFrame`
- This method performs aggregation over columns in each group. The result is a `DataFrame` with one row per group, containing the results of the aggregations.
- Spark provides many typical aggregation functions  
`(import org.apache.spark.sql.functions).`
  - `count`, `first`, `last`, `max`, `min`, `avg`, `sum`..



# Example of DataFrame operations

```
// Add a new column to the DataFrame
personDF = personDF.withColumn("young", personDF("age") < 40)

val groupAge = personDF.groupBy("young")
                        .agg(avg(personDF("age")).as("average_age"))

groupAge.show()

// SQL equivalent

spark.sql("select young, avg(age) as average_age from (select *, (age < 40) as
young from person) group by young").show()
```

# DataFrame functions

- The Spark SQL library provide an extensive set of Column operations, that can be combined with transformations like, select, filter, groupBy, etc.
  - Aggregation
  - Collection
  - Date/time
  - Math
  - Non-aggregation
  - Sorting
  - String transformation
  - Window functions
- Available in `org.apache.spark.sql.functions`

# Examples of DataFrame functions

- **Aggregation**

```
avg(columnName: String): Column  
collect_set(e: Column): Column  
mean(columnName: String): Column
```

- **Collection**

```
explode(e: Column): Column
```

- **Date/time**

```
to_timestamp(s: Column): Column  
datediff(end: Column, start:  
Column): Column
```

- **Math**

```
exp(e: Column): Column  
sqrt(e: Column): Column
```

- **Non-aggregation**

```
array(cols: Column*): Column  
coalesce(e: Column*): Column
```

- **Sorting**

```
asc_nulls_first(columnName: String):  
Column  
desc(columnName: String): Column
```

- **String transformation**

```
regex_extract(e: Column, exp:  
String, groupIdx: Int): Column  
upper(e: Column): Column
```

# User Defined Functions (UDFs)

- Sometimes, there is no easy way to achieve the desired result using the provided functions.
- Spark SQL includes a specific API for defining custom functions.
- Once defined, these UDFs can be used normally.
- To create one UDF, we use the `udf` method included in `org.apache.spark.sql.functions`
- You can specify the return type

```
def is_even_udf(x):  
    return x % 2 == 0  
  
is_even = udf(is_even_udf, BooleanType())  
  
df = spark.range(1, 100).toDF("x")  
  
res = df.select(col("x"),  
                is_even(col("x")).alias("is_even"))  
  
res.show()
```



# User Defined Aggregation Functions (UDAFs)

- Aggregation functions are used inside the `agg` method of `RelationalGroupedDataset`.
- UDAFs are a specific type of UDFs that can be used for aggregation.
- Implementing an UDAF is more complex than implementing a regular UDF.
- Aggregation functions are executed following a MapReduce-like pattern, and UDAFs need to be implemented accordingly.
- UDAFs are classes that implement the `UserDefinedAggregateFunction` abstract class.
  - `initialize(buffer: MutableAggregationBuffer): Unit`
  - `update(buffer: MutableAggregationBuffer, input: Row): Unit`
  - `merge(buffer1: MutableAggregationBuffer, buffer2: Row)`
  - `evaluate(buffer: Row): Any`

See <https://docs.databricks.com/spark/latest/spark-sql/udaf-scala.html>



# DataFrame joins

- Spark SQL provides basic join capabilities
- The main restriction is that joins can only be performed by column equality.
- All standard types of joins are available (full, left outer,...)
- Be careful, join big tables can result in performance issues.
- Many join types:
  - Inner
  - Outer
  - Right, Left
  - ...