

## SPARK SQL

### Entering Spark SQL - SparkSession (2.x) or SQLContext (1.x)

#### SparkSession

In earlier versions of spark, spark context was entry point for Spark. Essentially, SparkContext allows your application to access the cluster through a resource manager. For every other API, we needed to use different contexts. For streaming, we needed StreamingContext, for SQL sqlContext and for hive HiveContext. HiveContext is a super set of SQLContext that you would need if you want to access Hive tables, or to use richer functionalities and the trade-off is that HiveContext requires many dependencies to run.

So in Spark 2.x, we have a new entry point for DataSet and Dataframe API's called as Spark Session.

SparkSession is essentially combination of SQLContext, HiveContext and future StreamingContext. All the API's available on those contexts are available on spark session also. Spark session internally has a spark context for actual computation.

```
val sqlc=new org.apache.spark.sql.SQLContext(sc)
val hqlc=new org.apache.spark.sql.hive.HiveContext(sc)
```

#### Creating a SparkSession

```
// The builder automatically reuse an existing SparkContext if one exists and creates a
SparkContext if it does not exist.
```

A SparkSession can be created using a builder pattern.

Configuration options set in the builder are automatically propagated to Spark and Hadoop during I/O.

```
import org.apache.spark.sql.SparkSession
val sparkSession = SparkSession.builder().getOrCreate()
```

```
//Check the version of the Spark
spark.version

//Get a spark context out of SparkSession
spark.sparkContext

//Get a hiveContext context
val sparkSession =
SparkSession.builder.enableHiveSupport.getOrCreate()
```

## Creating a Dataframe

### Working with native SQL

Load data from csv, create temp table and write sql queries

### Setting log level to Error

```
sparkSession.sparkContext.setLogLevel("ERROR")
```

Loading data from structured files, RDBMS, NOSQLs, Cloud storage, RDDs, Hive etc.,

### Working with Dataframes

Spark SQL introduces a tabular functional data abstraction called **DataFrame**. It is designed to ease developing Spark applications for processing large amount of structured tabular data on Spark infrastructure.

A DataFrame is a distributed collection of data, which is organized into named columns. Conceptually, it is equivalent to relational tables with good optimization techniques. DataFrame provides a domain-specific language API for structured data manipulation, with **structured** and **semi-structured data**.

DataFrame is a collection of rows with a schema that is the result of executing a structured query (once it will have been executed).

DataFrame uses the immutable, in-memory, resilient, distributed and parallel capabilities of RDD, and applies a structure called schema to the data.

## Dataframes can be created using multiple methods

### a. Create DF by Inferring the Schema using Reflection

1. Create case class as per the structure of data
2. Iterate on every line of rdd, split on the delimiter and apply the structure calling the case class (SchemaedRDD)
3. Convert the schemaedRDD to DF.
4. Ready to write DSL
5. Register the DF to temp view.
6. Write ISO sql on top of the temp view created.

Case classes are just regular classes that are: Immutable by default. Decomposable through pattern matching. Compared by structural equality instead of by reference.

```
case class Auction(auctionid: String, bid: Float, bidtime: Float, bidder: String, bidderrate: Integer, openbid: Float, price: Float, item: String, daystolive: Integer)
```

```
// load the data into an RDD
```

```
val auctionRDD = spark.sparkContext.textFile("file:///home/hduser/sparkdata/auctiondata")
```

```
// create an RDD of Auction objects
```

```
val ebay = auctionRDD.map(_._split("~")).map(p => Auction(p(0), p(1).toFloat, p(2).toFloat, p(3), p(4).toInt, p(5).toFloat, p(6).toFloat, p(7), p(8).toInt))
```

```
// change ebay RDD of Auction objects to a DataFrame
```

```
val auction = spark.createDataFrame(ebay)
```

```
//implicits object gives implicit conversions for converting scala objects (incl. RDDs) into a Dataset, DataFrame, Columns using toDF function
```

```
import spark.sqlContext.implicits._
```

```
val auctiontodf=ebay.toDF
```

### DF transformation operations simple example:

// How many auctions were held

```
val count = auction.select("auctionid").distinct.count
```

// Identify maximum bids and amount grouped by items

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

```
val grpbid = auction.groupBy("item").agg(max("bid").alias("max_bid"),sum("price").alias("sp")).filter($"max_bid">100).sort($"sp".desc)
```

```
auction.printSchema
```

```
grpbid.printSchema
```

```
grpbid.show(10,false)
```

```
auction.show(10,false)
```

//Write the same functionality in native SQL

```
auction.createOrReplaceTempView("view1")
```

```
sql("""select item,max(bid) as max_bid,sum(price) as sp
from view1
group by item
having max(bid)>100
order by sp desc""").show(10,false)
```

//Stores the above dataframes in JSON format

```
auction.write.mode("overwrite").json("file:/home/hduser/sparkdata/auctiondata.json");
```

```
grpbid.write.mode("overwrite").json("file:/home/hduser/sparkdata/grpbid.json");
```

### *Creating DataFrame from CSV file*

//Reading CSV files in local or distributed filesystem as Spark DataFrames.

## *Creating DataFrame from CSV files using spark-csv module*

Use [spark-csv](#) module to load data from a CSV data source that handles proper parsing and loading.

```
//Get a sql context  
var sqlctx =spark.sqlContext
```

### **b. Create Dataframe programatically or using csv module**

```
val uscsvdf =  
sqlctx.read.option("header","true").option("inferschema","true").option("delimiter","," ).csv(  
"file:///home/hduser/sparkdata/usdata.csv")  
  
uscsvdf.printSchema  
  
uscsvdf.show(10,false)  
  
// Creates a temporary view using the DataFrame.  
  
uscsvdf.createOrReplaceTempView("custinfo")  
spark.catalog.listDatabases.show(10,false);  
spark.catalog.listTables.show(10,false);  
  
// Perform SQL queries directly on the registered temp view  
sqlctx.sql("describe custinfo").show(10,false)  
  
sql("select concat(first_name,' ',last_name),company_name,address,city,phone1 from  
custinfo where upper(company_name) like 'CH%'").show(10,false)  
  
sql("select count(1),city from custinfo group by city having count(1) >1 order by city").show()  
  
sql("select count(1) as cnt,city from custinfo group by city having count(1)> 5 order by  
city").show()  
  
val agecat = sqlctx.sql("select distinct age,case when age <=10 then 'childrens' when age >10  
and age < 20 then 'teen' else 'others' end as agecat  from custinfo order by agecat")
```

```
agecat.show()
```

//Stores the output in Parquet and orc format, Parquet files are self-describing so the schema is preserved

```
agecat.write.mode("overwrite").parquet("file:/home/hduser/sparkdata/agecategory.parquet");
```

```
agecat.write.mode("append").orc("file:/home/hduser/sparkdata/agecategory1.orc");
```

// Override the column names and type using collection type such as structtype and structfield

// StructType objects define the schema of Spark DataFrames. StructType objects contain a list of StructField objects that define the name, type, and nullable flag for each column in a DataFrame.

```
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType}
```

```
val custschema = StructType(Array(StructField("first_name", StringType, true),StructField("last_name", StringType, true),StructField("company_name", StringType, true),StructField("address", StringType, true),StructField("city", StringType, true),StructField("country", StringType, true),StructField("state", StringType, true),StructField("zip", StringType, true),StructField("age", IntegerType, true),StructField("phone1", StringType, true),StructField("phone2", StringType, true),StructField("email", StringType, true),StructField("website", StringType, true)));
```

```
val uscsvdf1 =  
sqlctx.read.option("delimiter",",").schema(custschema).csv("file:///home/hduser/sparkdata/usdata.csv")
```

```
uscsvdf1.printSchema
```

// Read in the parquet file created above

// Parquet files are self-describing so the schema is preserved

// The result of loading a Parquet file is also a DataFrame

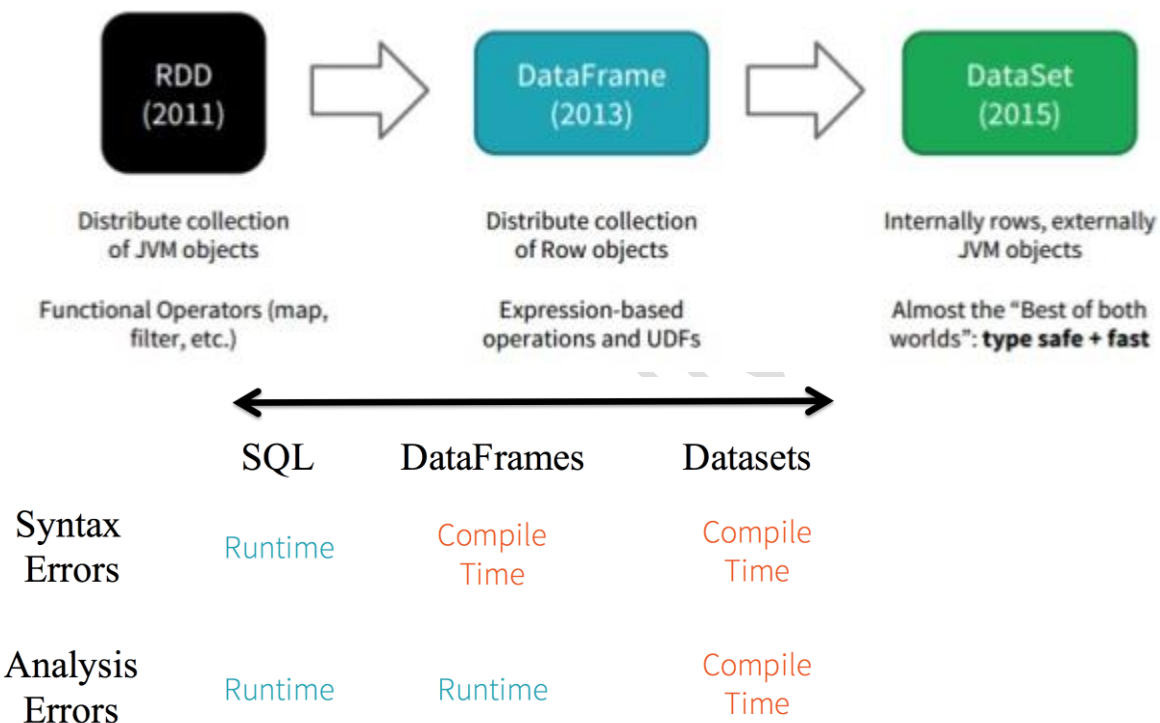
```
val parquetagecatdf =  
spark.read.parquet("file:/home/hduser/sparkdata/agecategory.parquet")
```

```
// Parquet files can also be used to create a temporary view and then used in SQL statements
```

```
parquetagecatdf.createOrReplaceTempView("parquetagecat")
```

```
spark.sql("SELECT max(age),min(age),count(distinct agecat) FROM parquetagecat ").show()
```

## Dataset (1.6) Vs Dataframe (1.3)



A Dataset is a strongly typed collection of domain-specific objects (like object of type auction declared below) that can be transformed in parallel using functional or relational operations. Each Dataset also has an untyped view called a DataFrame, which is a Dataset of Row.

```
Dataframe=dataset[Row]
```

```
DataSet=dataset[Schema]
```

Dataframe is merged with Dataset API. So we can use any method available for dataframe in datasets.

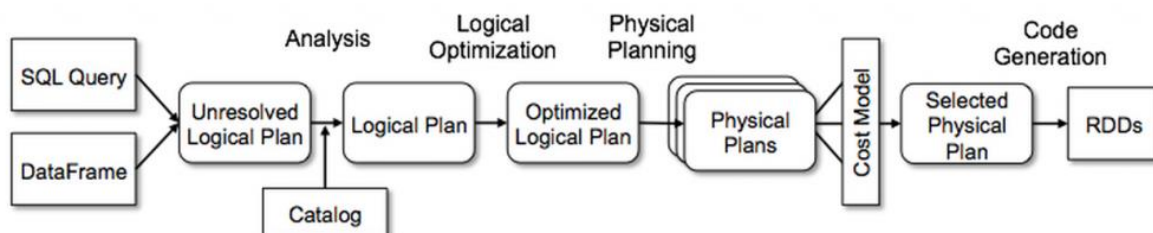
In summation, the choice of when to use RDD or DataFrame and/or Dataset seems obvious. While the former offers you low-level functionality and control, the latter allows custom view and structure, offers high-level and domain specific operations, saves space, and executes at superior speeds.

To simplify Spark for developers, how to optimize and make it performant it was decided to elevate the low-level RDD APIs to a high-level abstraction as DataFrame and Dataset and to build this unified data abstraction across libraries a top Catalyst Optimizer and Tungsten Serializer.

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).

- Tungsten includes specialized in-memory data structures tuned for the type of operations required by Spark with specialized efficient serialization called encoders rather than using Kryo java serialization.
- As Tungsten does not depend on Java objects, both on-heap and off-heap allocations are supported. Since Tungsten no longer depends on working with Java objects, you can use either on-heap (in the JVM) or off-heap storage.
- Tungsten will not do deserialization when processing data, for example of this is with sorting, a common and expensive operation using tungsten this can be done without having to deserialize the data again.
- By avoiding the memory and GC overhead of regular Java objects, Tungsten is able to process larger data sets than the same hand-written aggregations.

#### Catalyst Optimizer:



Catalyst supports both rule-based and cost-based optimization.

At its core, Catalyst contains a general library for representing trees and applying rules to manipulate them. On top of this framework, we have built libraries specific to relational query processing (e.g., expressions, logical query plans), and several sets of rules that handle different phases of query execution: analysis, logical optimization, physical planning, and code generation to compile parts of queries to Java bytecode. For the latter, we use another Scala feature, that makes it easy to generate code at runtime from composable expressions. Finally, Catalyst offers several public extension points, including external data sources and user-defined types.

Pick one—DataFrames and/or Dataset or RDDs APIs—that meets your needs and use-case



// A JSON dataset is pointed to a path. The path can be either a single text file or a directory storing text files

//weakly typed Dataframes

```
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType}
val custschema = StructType(Array(StructField("first_name", IntegerType,
true),StructField("last_name", StringType, true),StructField("company_name", StringType,
true),StructField("address", StringType, true),StructField("city", StringType,
true),StructField("country", StringType, true),StructField("state", StringType,
true),StructField("zip", StringType, true),StructField("age", IntegerType,
true),StructField("phone1", StringType, true),StructField("phone2", StringType,
true),StructField("email", StringType, true),StructField("website", StringType, true)));
```

// No Compile Time Safety

```
val uscsvdf1 =
sqlctx.read.option("header","true").option("delimiter",";").schema(custschema).csv("file:///
/home/hduser/sparkdata/usdata.csv")
uscsvdf1.printSchema
```

// Run time Safety

```
uscsvdf1.show(10);
```

//Strongly typed DataSets

```
case class
custschemaoops(first_name:Int,last_name:String,company_name:String,address:String,city:
String,county:String,state:String,zip:String,age:Int,phone1:String,phone2:String,email:String,
web:String)
```

// Compile Time Safety

```
val uscsvds1 =
sqlctx.read.option("inferschema","true").option("header","true").csv("file:///home/hduser/s
parkdata/usdata.csv").as[custschemaoops]
```

```
uscsvds1.printSchema
```

```
uscsvds1.show(10);
```

```
//Strongly typed DataSets
```

```
case class
```

```
custschemaoops1(first_name:String,last_name:String,company_name:String,address:String,  
city:String,county:String,state:String,  
zip:String,age:Int,phone1:String,phone2:String,email:String,web:String)
```

```
// Compile Time Safety
```

```
val uscsvds1 =
```

```
sqlctx.read.option("inferschema","true").option("header","true").csv("file:///home/hduser/s  
parkdata/usdataupdated.csv").as[custschemaoops1]
```

```
uscsvds1.printSchema
```

```
uscsvds1.show(10);
```

|                        | <b>RDD</b>                                                                                         | <b>DataFrame</b>                                                                                                                     | <b>DataSet</b>                                                                                                             | <b>TempView</b>                                        |
|------------------------|----------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------|
| Definition             | An RDD stands for Resilient Distributed Datasets. It is Read-only partition collection of objects. | DataFrame in Spark allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction. | Datasets in Apache Spark are an extension of DataFrame API which provides type-safe, object-oriented programming interface | creates an in memory reference to the Dataframe in use |
| Data Type Supported    | process data which is structured as well as unstructured                                           | It works only on structured and semi-structured data                                                                                 | It works only on structured and semi-structured data                                                                       | same as DF or DS                                       |
| Data Sources Supported | Text file                                                                                          | Different formats (AVRO, CSV, JSON, and storage system HDFS, Cloud, HIVE tables, RDBMS)                                              | Same as DF                                                                                                                 | same as DF or DS                                       |
| Schema Bounding        | SchemaRDD with Caseclass                                                                           | After transforming into DataFrame one cannot regenerate the schema RDD                                                               | After transforming into DataSet can regenerate the schema RDD                                                              | same as DF or DS                                       |
| Datatype Safety        | Compile-time type safety                                                                           | No compile-time type safety                                                                                                          | Compile-time type safety                                                                                                   | No Compile time Type safety                            |
| Optimization           | No inbuilt optimization engine is available in RDD                                                 | Optimization takes place using catalyst optimizer                                                                                    | Same as Dataframe, It stores tabular representation using spark internal Tungsten binary format for serialization          | same as DF or DS                                       |

## Hive operations:

```
import spark.implicits._
import spark.sql

//Initialize hive context wrapping spark context

val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)

//Get a hiveContext context

val sparkSession =
  SparkSession.builder.enableHiveSupport.getOrCreate()

//Create a hive table

sql("create database if not exists sparkdb")
sql("use sparkdb")

sql("DROP TABLE IF EXISTS employee ")

sql("CREATE TABLE IF NOT EXISTS employee(id INT, name STRING, age INT) ROW FORMAT
DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'")

spark.catalog.listDatabases.show(10,false);
spark.catalog.listTables.show(10,false);

//Load data

sql("LOAD DATA LOCAL INPATH '/home/hduser/sparkdata/employee.txt' INTO TABLE
employee")

//View data

val results = sql("FROM employee SELECT id, name, age")
```

```
results.show()
```

### **Additional Hive Usecases (For self practices):**

```
//Drop table
```

```
sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'")
```

```
sql("LOAD DATA LOCAL INPATH '/home/hduser/sparkdata/sampleddata' OVERWRITE INTO  
TABLE src")
```

```
sql("SELECT * FROM src").show()
```

```
sql("drop table src")
```

```
//Write the table content into the given hdfs location in orc format
```

```
results.write.format("orc").save("hdfs:/user/hduser/emp_orc")
```

```
//Read the table from the given hdfs location in orc format and create a dataset out of it.
```

```
case class empclass (id :Int,name :String,age:Int);
```

```
val empdata_orc =  
spark.read.format("orc").load("hdfs:/user/hduser/emp_orc").as[empclass];
```

```
empdata_orc.createOrReplaceTempView("orcddata")
```

```
sql("SELECT * from orcddata").collect.foreach(println)
```

### **Hive Partitioning:**

```
sql("""create table txnrecords(txnno INT, txndate STRING, custno INT, amount DOUBLE,  
category STRING, product STRING, city STRING, state STRING, spendby STRING) row format  
delimited fields terminated by ','
```

lines terminated by '\n'  
stored as textfile""")

```
sql("LOAD DATA LOCAL INPATH '/home/hduser/hive/data/txns' OVERWRITE INTO TABLE  
txnrecords")
```

```
sql("select * from txnrecords order by 1 limit 10").show
```

```
sql (""create external table exttxnrecsByCat(txnno INT, txndate STRING, custno INT, amount  
DOUBLE, product STRING, city STRING, state STRING, spendby STRING)  
partitioned by (category STRING) row format delimited  
fields terminated by ','  
stored as textfile  
location '/user/hive/warehouse/exttxnrecsbycat'""")
```

```
sql("""Insert into table exttxnrecsbycat partition (category='Games')  
select txnno,txndate,custno,amount,product,city,state,spendby  
from txnrecords  
where category='Games'""")
```

```
sql("select count(1) from exttxnrecsbycat").show
```