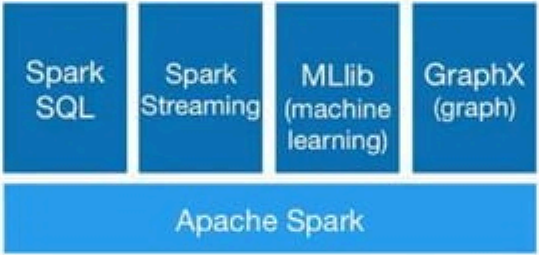
Spark SQL

- a programming module for **structured data** processing
- provides a programming abstraction called DataFrame (SchemaRDD)
- act as distributed SQL query engine
- Transform RDDs using SQL
- Data source integration: Hive, Parquet, JSON, and more

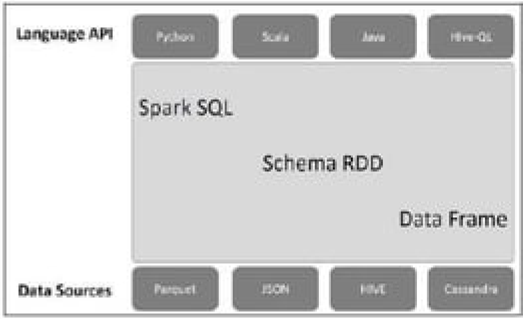


Spark SQL features

- **Integrated**
  - Seamlessly mix SQL queries with Spark programs
  - Spark SQL lets you query structured data as a distributed dataset (RDD) in Spark
  - integrated APIs in Python, Scala and Java and R
- **Unified Data Access**
  - Load and query data from a variety of sources
  - Schema-RDDs provide a single interface for efficiently working with structured data, including Apache Hive tables, parquet files and JSON files
- **Hive Compatibility**
  - Run unmodified Hive queries on existing warehouses.
  - Spark SQL reuses the Hive frontend and MetaStore, giving you full compatibility with existing Hive data, queries, and UDFs.
  - Simply install it alongside Hive
- **Standard Connectivity**
  - Connect through JDBC or ODBC
  - Spark SQL includes a server mode with industry standard JDBC and ODBC connectivity.
- **Scalability**
  - Use the same engine for both interactive and long queries. Spark SQL takes advantage of the RDD model to support mid-query fault tolerance, letting it scale to large jobs too.

Spark SQL Architecture

- 3 Layers
  - **Language API**
    - supported by these (python, scala, java, HiveQL).
  - **Schema RDD**
    - Generally, Spark SQL works on schemas, tables, and records. Therefore, we can use the Schema RDD as temporary table. We can call this Schema RDD as Data Frame.
  - **Data Sources**
    - Usually the Data source for spark-core is a text file, Avro file, etc. However, the Data Sources for Spark SQL is different. Those are Parquet file, JSON document, HIVE tables, and Cassandra database



Language API

Python

Scala

Java

Hive-QL

Spark SQL

Schema RDD

Data Frame

Data Sources

Parquet

JSON

HIVE

Cassandra

Spark DataFrame

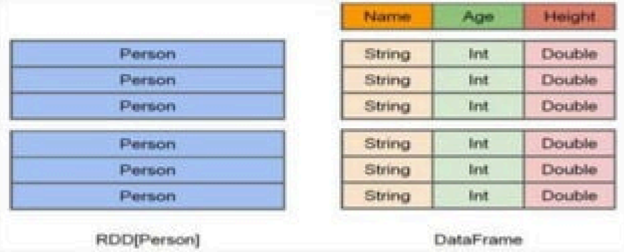
Adding Schema to RDDs

Spark + RDDs

- Functional transformations on partitioned collections of opaque objects.

SQL + SchemaRDDs

- Declarative transformations on partitioned collections of tuples.



Person
Person
Person

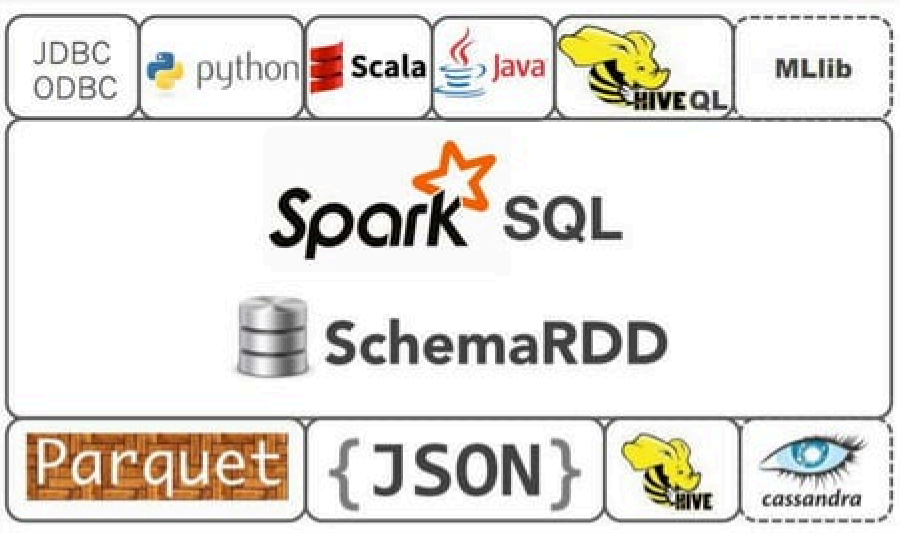
Person
Person
Person

RDD[Person]

Name	Age	Height
String	Int	Double
String	Int	Double
String	Int	Double
String	Int	Double
String	Int	Double
String	Int	Double

DataFrame

SchemaRDDs: More than SQL



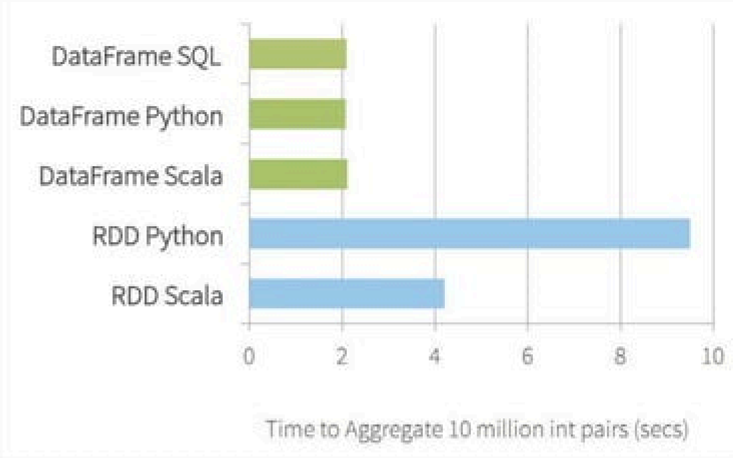
DataFrame

- a distributed collection of data
- organized into named columns
- Conceptually equivalent to relational tables
- good optimization techniques
- can be constructed from an array of different sources such as Hive tables, Structured Data files, external databases, or existing RDDs.
- was designed for modern Big Data and data science applications taking inspiration from **DataFrame in R Programming** and **Pandas in Python**.

Features Of DataFrame

- Ability to process the data in the size of Kilobytes to Petabytes on a single node cluster to large cluster.
- Supports different data formats (Avro, csv, elastic search, and Cassandra) and storage systems (HDFS, HIVE tables, mysql, etc).
- State of art optimization and code generation through the Spark SQL Catalyst optimizer (tree transformation framework).
- Optimization happens as late as possible, therefore Spark SQL can optimize across functions
- Can be easily integrated with all Big Data tools and frameworks via Spark-Core.
- Provides API for Python, Java, Scala, and R Programming.

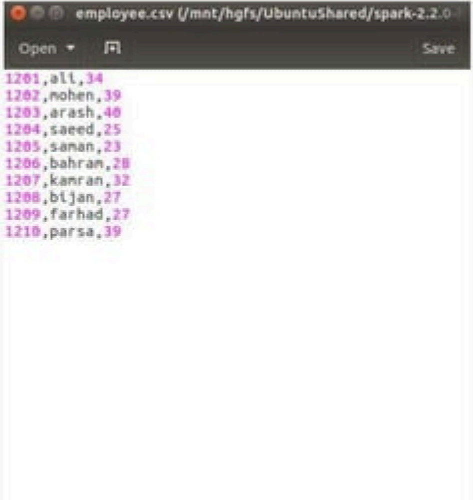
DataFrames Are Faster than RDDs



First Example

Example DataSet

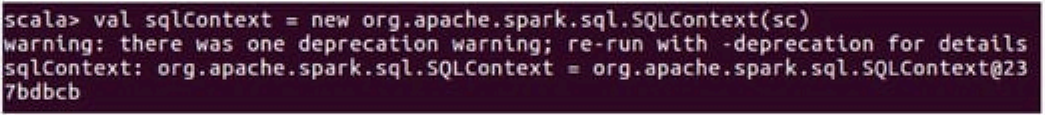
- A text file filled with employee info



```
1201,ali,34
1202,mohen,39
1203,arash,40
1204,saeed,25
1205,saman,23
1206,bahram,28
1207,kamran,32
1208,bijan,27
1209,farhad,27
1210,parsa,39
```

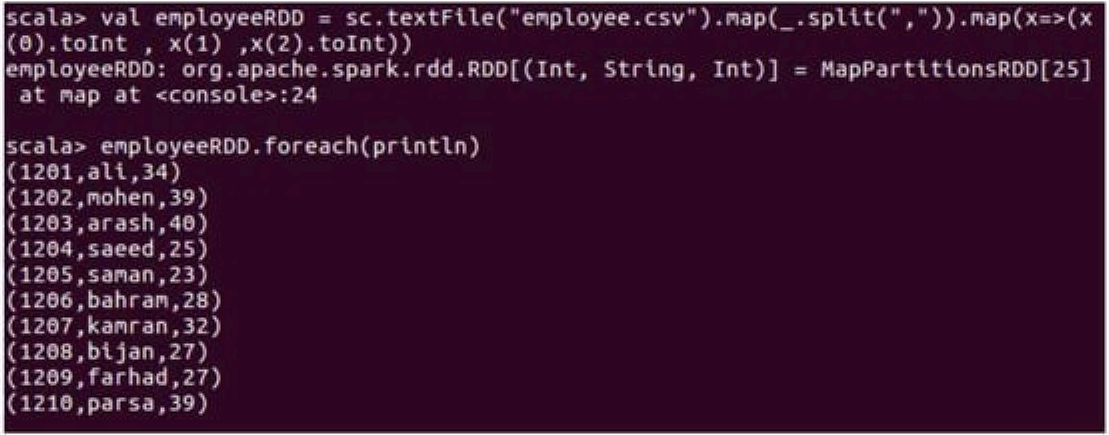
SQLContext

- initializing the functionalities of Spark SQL
- SparkContext class object (sc) is required for initializing SQLContext class object
- to create SQLContext :



```
scala> val sqlContext = new org.apache.spark.sql.SQLContext(sc)
warning: there was one deprecation warning; re-run with -deprecation for details
sqlContext: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@23
7bdbcb
```

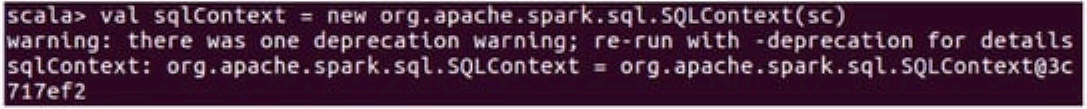
Create RDD from CSV File



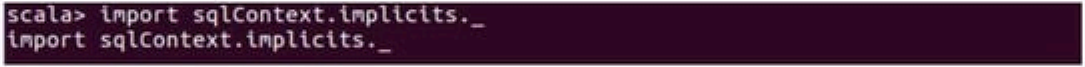
```
scala> val employeeRDD = sc.textFile("employee.csv").map(_.split(",")).map(x=>(x
(0).toInt , x(1) ,x(2).toInt))
employeeRDD: org.apache.spark.rdd.RDD[(Int, String, Int)] = MapPartitionsRDD[25]
 at map at <console>:24

scala> employeeRDD.foreach(println)
(1201,ali,34)
(1202,mohen,39)
(1203,arash,40)
(1204,saeed,25)
(1205,saman,23)
(1206,bahram,28)
(1207,kamran,32)
(1208,bijan,27)
(1209,farhad,27)
(1210,parsa,39)
```

Create DataFrame from RDD - Reflection

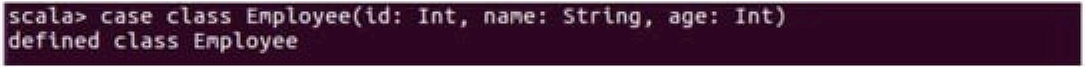
- uses reflection to generate the schema of an RDD that contains specific types of objects



```
scala> val sqlContext = new org.apache.spark.sql.SQLContext(sc)
warning: there was one deprecation warning; re-run with -deprecation for details
sqlContext: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@3c
717ef2
```



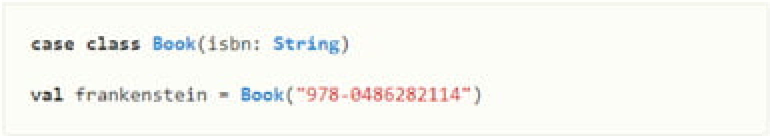
```
scala> import sqlContext.implicits._
import sqlContext.implicits._
```



```
scala> case class Employee(id: Int, name: String, age: Int)
defined class Employee
```

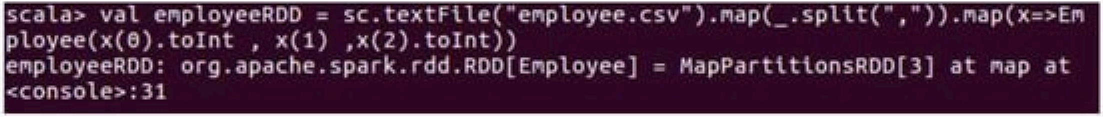
Case Classes

- case classes are good for modeling immutable data
- A minimal case class requires the keywords case class, an identifier, and a parameter list (which may be empty):
- Notice how the keyword new was not used to instantiate the Book case class. This is because case classes have an apply method by default which takes care of object construction.
- When you create a case class with parameters, the parameters are public vals

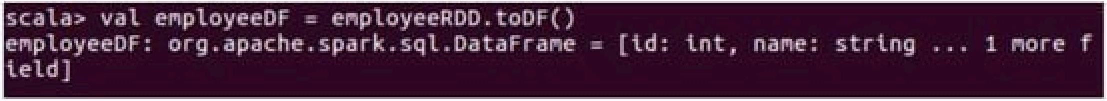


```
case class Book(isbn: String)

val frankenstein = Book("978-0486282114")
```

Create DataFrame from RDD - Reflection

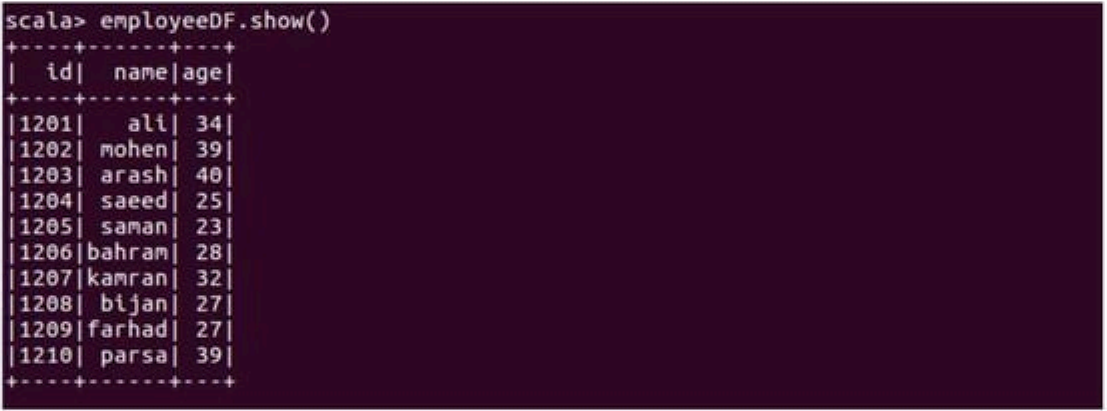


```
scala> val employeeRDD = sc.textFile("employee.csv").map(_.split(",")).map(x=>Em
ployee(x(0).toInt , x(1) ,x(2).toInt))
employeeRDD: org.apache.spark.rdd.RDD[Employee] = MapPartitionsRDD[3] at map at
<console>:31
```



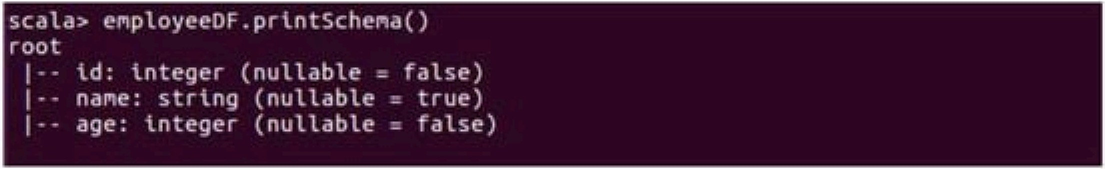
```
scala> val employeeDF = employeeRDD.toDF()
employeeDF: org.apache.spark.sql.DataFrame = [id: int, name: string ... 1 more f
ield]
```

DataFrame-show



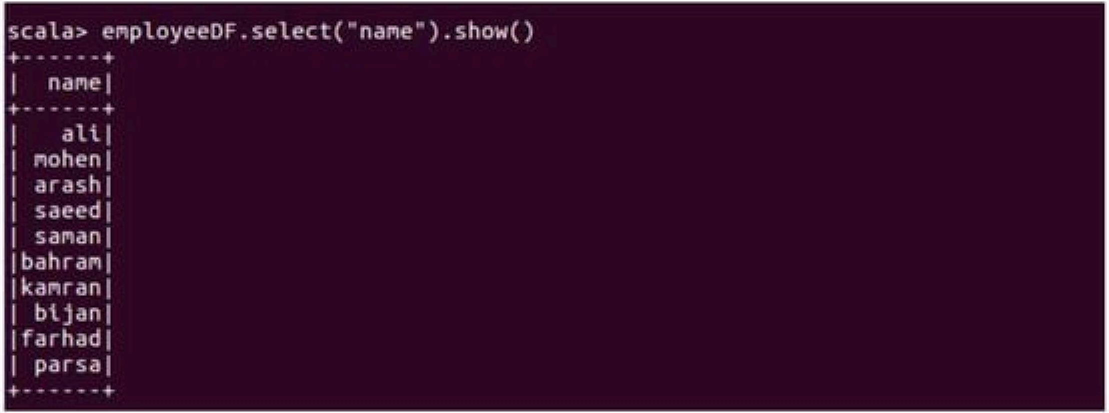
```
scala> employeeDF.show()
+----+------+---+
|  id|  name|age|
+----+------+---+
|1201|   ali| 34|
|1202| mohen| 39|
|1203| arash| 40|
|1204| saeed| 25|
|1205| saman| 23|
|1206|bahram| 28|
|1207|kamran| 32|
|1208| bijan| 27|
|1209|farhad| 27|
|1210| parsa| 39|
+----+------+---+
```

DataFrame-printSchema



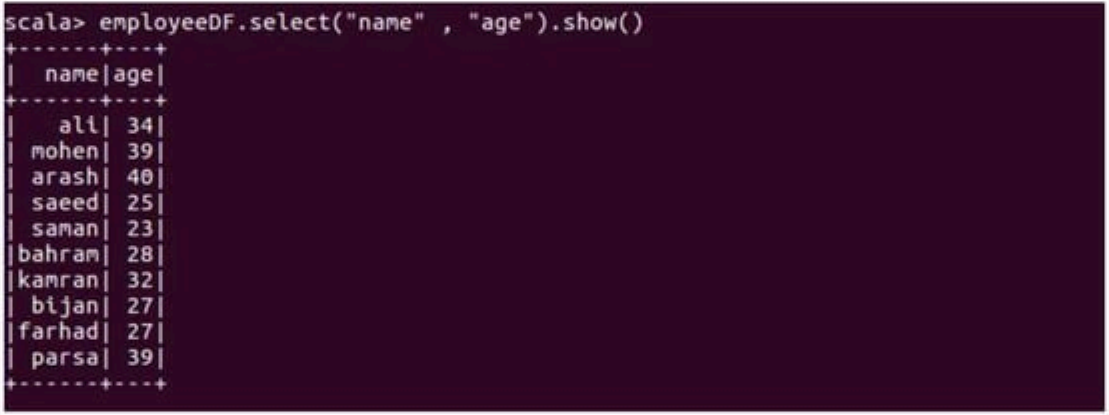
```
scala> employeeDF.printSchema()
root
 |-- id: integer (nullable = false)
 |-- name: string (nullable = true)
 |-- age: integer (nullable = false)
```

DataFrame-Select



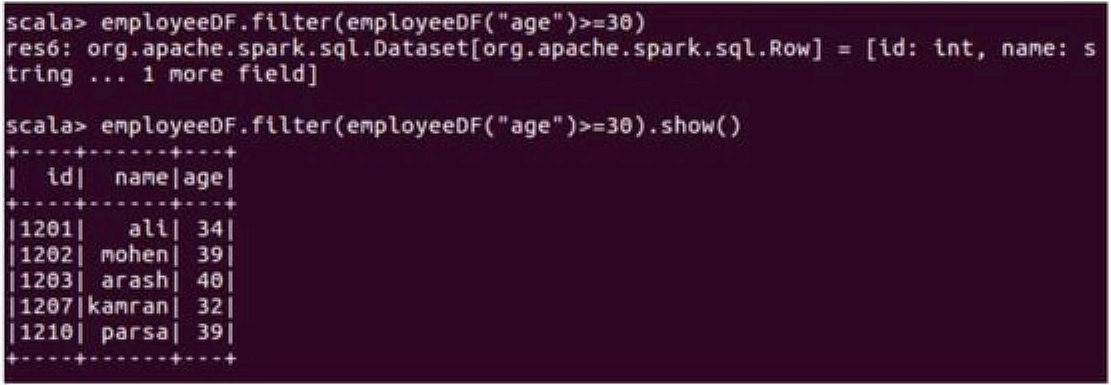
```
scala> employeeDF.select("name").show()
+------+
|  name|
+------+
|   ali|
| mohen|
| arash|
| saeed|
| saman|
|bahram|
|kamran|
| bijan|
|farhad|
| parsa|
+------+
```

DataFrame-Select



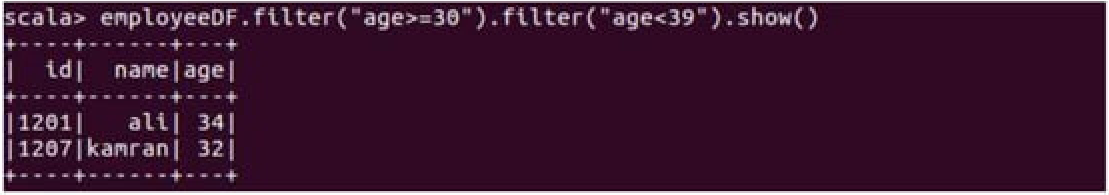
```
scala> employeeDF.select("name" , "age").show()
+------+---+
|  name|age|
+------+---+
|   ali| 34|
| mohen| 39|
| arash| 40|
| saeed| 25|
| saman| 23|
|bahram| 28|
|kamran| 32|
| bijan| 27|
|farhad| 27|
| parsa| 39|
+------+---+
```

DataFrame-filter



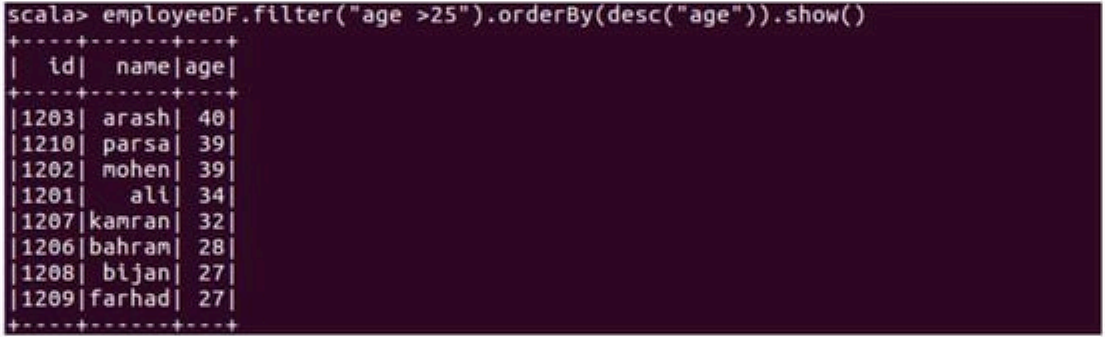
```
scala> employeeDF.filter(employeeDF("age")>=30)
res6: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: int, name: s
tring ... 1 more field]

scala> employeeDF.filter(employeeDF("age")>=30).show()
+----+------+---+
|  id|  name|age|
+----+------+---+
|1201|   ali| 34|
|1202| mohen| 39|
|1203| arash| 40|
|1207|kamran| 32|
|1210| parsa| 39|
+----+------+---+
```

DataFrame-filter



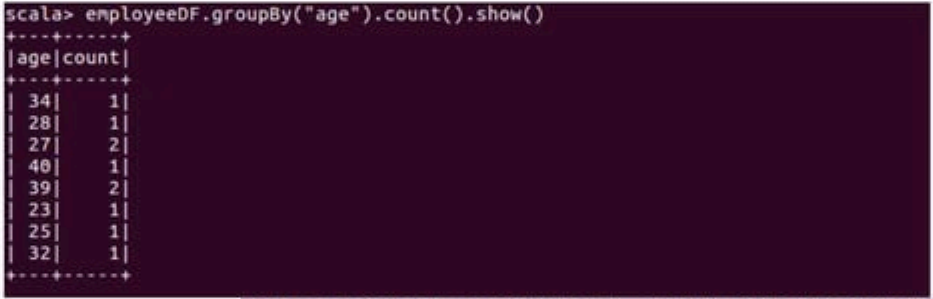
```
scala> employeeDF.filter("age>=30").filter("age<39").show()
+----+------+---+
|  id|  name|age|
+----+------+---+
|1201|   ali| 34|
|1207|kamran| 32|
+----+------+---+
```

DataFrame-orderBy

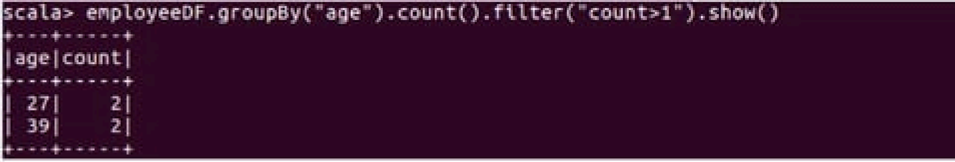


```
scala> employeeDF.filter("age >25").orderBy(desc("age")).show()
+----+------+---+
|  id|  name|age|
+----+------+---+
|1203| arash| 40|
|1210| parsa| 39|
|1202| mohen| 39|
|1201|   ali| 34|
|1207|kamran| 32|
|1206|bahram| 28|
|1208| bijan| 27|
|1209|farhad| 27|
+----+------+---+
```

DataFrame-groupBy

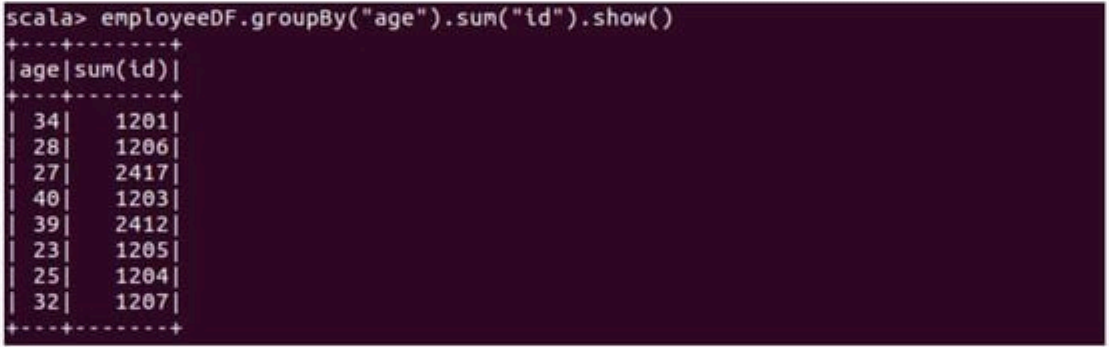


```
scala> employeeDF.groupBy("age").count().show()
+---+-----+
|age|count|
+---+-----+
| 34|    1|
| 28|    1|
| 27|    2|
| 40|    1|
| 39|    2|
| 23|    1|
| 25|    1|
| 32|    1|
+---+-----+
```



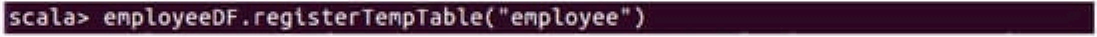
```
scala> employeeDF.groupBy("age").count().filter("count>1").show()
+---+-----+
|age|count|
+---+-----+
| 27|    2|
| 39|    2|
+---+-----+
```

DataFrame-groupBy

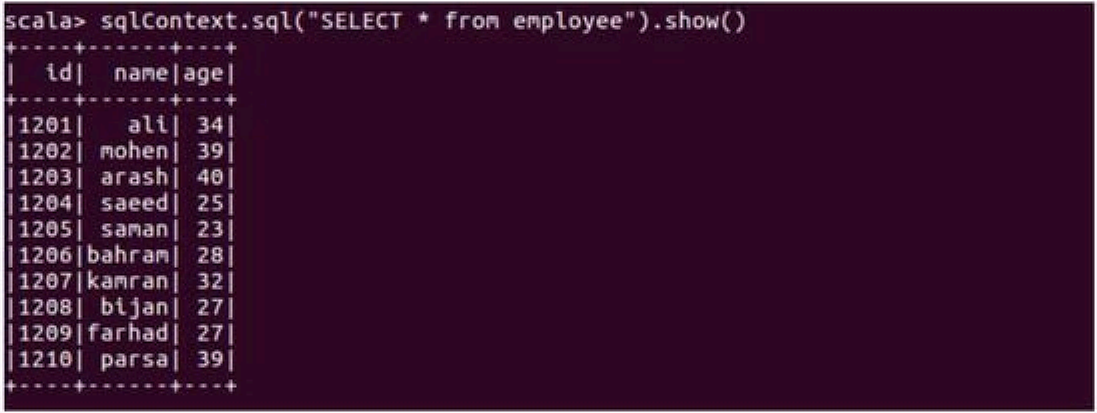


```
scala> employeeDF.groupBy("age").sum("id").show()
+---+-------+
|age|sum(id)|
+---+-------+
| 34|   1201|
| 28|   1206|
| 27|   2417|
| 40|   1203|
| 39|   2412|
| 23|   1205|
| 25|   1204|
| 32|   1207|
+---+-------+
```

Using SQL

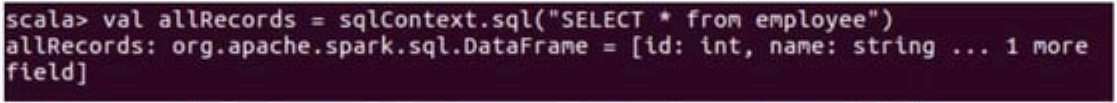


```
scala> employeeDF.registerTempTable("employee")
```

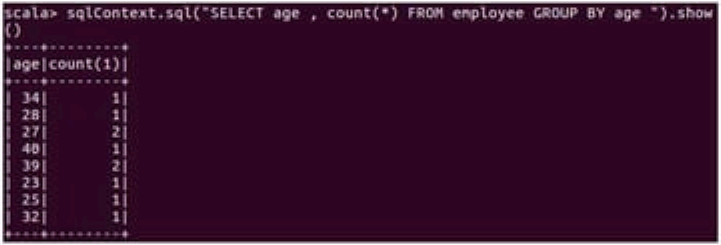


```
scala> sqlContext.sql("SELECT * from employee").show()
+----+------+---+
|  id|  name|age|
+----+------+---+
|1201|   ali| 34|
|1202| mohen| 39|
|1203| arash| 40|
|1204| saeed| 25|
|1205| saman| 23|
|1206|bahram| 28|
|1207|kamran| 32|
|1208| bijan| 27|
|1209|farhad| 27|
|1210| parsa| 39|
+----+------+---+
```

Using SQL

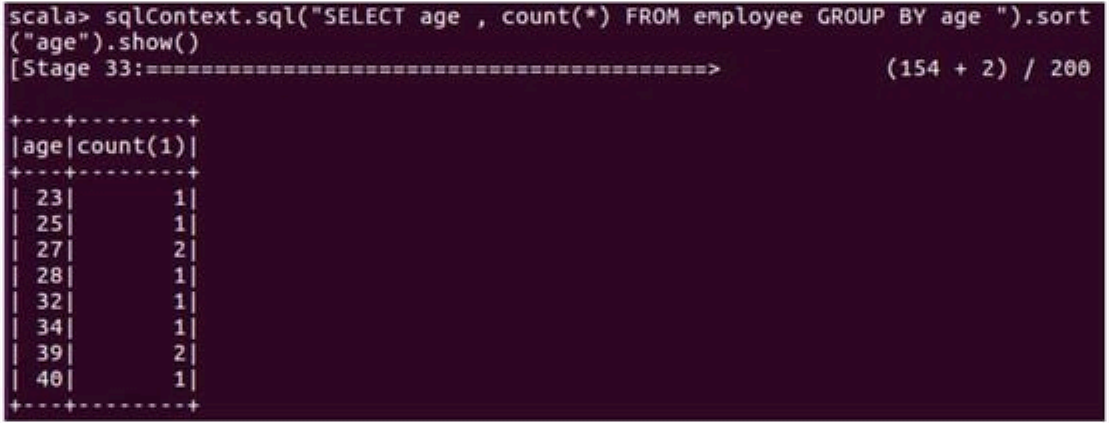


```
scala> val allRecords = sqlContext.sql("SELECT * from employee")
allRecords: org.apache.spark.sql.DataFrame = [id: int, name: string ... 1 more
field]
```



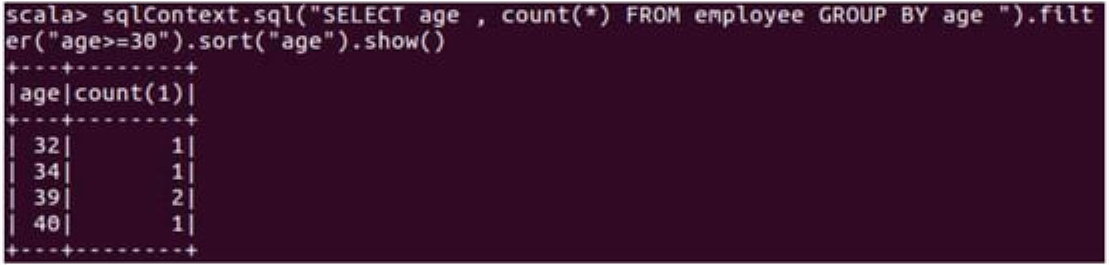
```
scala> sqlContext.sql("SELECT age , count(*) FROM employee GROUP BY age ").show
()
+---+--------+
|age|count(1)|
+---+--------+
| 34|       1|
| 28|       1|
| 27|       2|
| 40|       1|
| 39|       2|
| 23|       1|
| 25|       1|
| 32|       1|
+---+--------+
```

Using SQL



```
scala> sqlContext.sql("SELECT age , count(*) FROM employee GROUP BY age ").sort
("age").show()
[Stage 33:=========================================>            (154 + 2) / 200

+---+--------+
|age|count(1)|
+---+--------+
| 23|       1|
| 25|       1|
| 27|       2|
| 28|       1|
| 32|       1|
| 34|       1|
| 39|       2|
| 40|       1|
+---+--------+
```

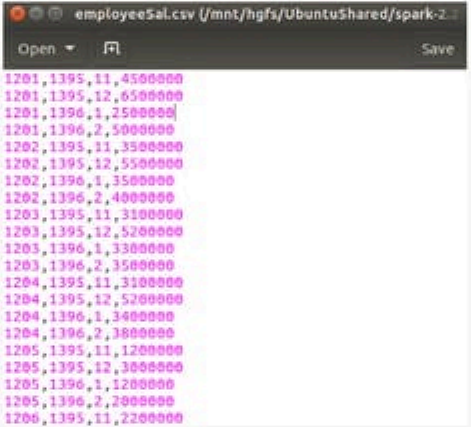
Using SQL



```
scala> sqlContext.sql("SELECT age , count(*) FROM employee GROUP BY age ").filt
er("age>=30").sort("age").show()
+---+--------+
|age|count(1)|
+---+--------+
| 32|       1|
| 34|       1|
| 39|       2|
| 40|       1|
+---+--------+
```

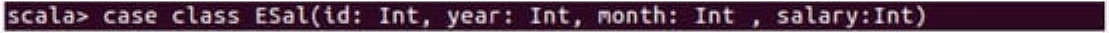
DataFrame-JOIN

- A text file of employeeId , year , month , salary
- Join employeeDF with this data

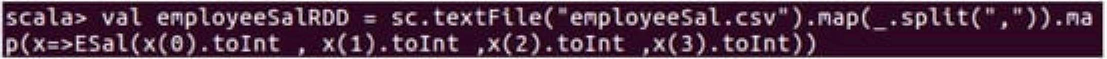


```
1201,1395,11,4500000
1201,1395,12,6500000
1201,1396,1,2500000
1201,1396,2,5000000
1202,1395,11,3500000
1202,1395,12,5500000
1202,1396,1,3500000
1202,1396,2,4000000
1203,1395,11,3100000
1203,1395,12,5200000
1203,1396,1,3300000
1203,1396,2,3500000
1204,1395,11,3100000
1204,1395,12,5200000
1204,1396,1,3400000
1204,1396,2,3800000
1205,1395,11,1200000
1205,1395,12,3000000
1205,1396,1,1200000
1205,1396,2,2000000
1206,1395,11,2200000
```

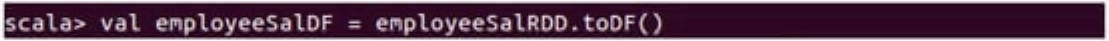
DataFrame-JOIN



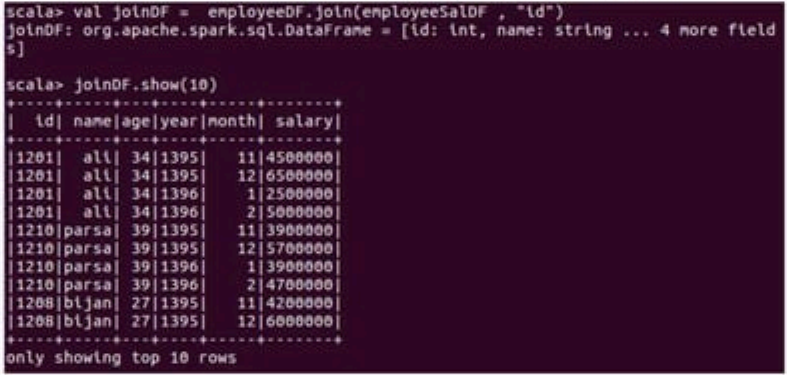
```
scala> case class ESal(id: Int, year: Int, month: Int , salary:Int)
```



```
scala> val employeeSalRDD = sc.textFile("employeeSal.csv").map(_.split(",")).ma
p(x=>ESal(x(0).toInt , x(1).toInt ,x(2).toInt ,x(3).toInt))
```

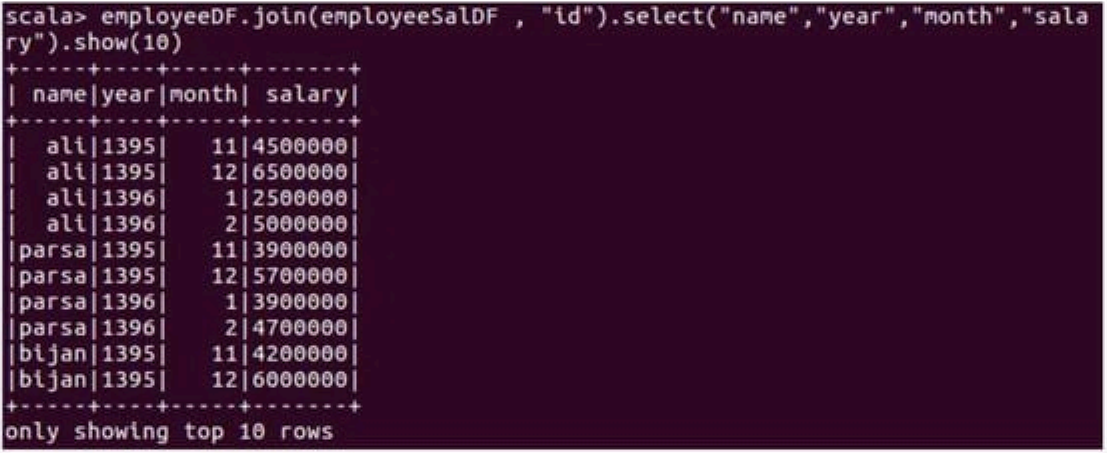


```
scala> val employeeSalDF = employeeSalRDD.toDF()
```



```
scala> val joinDF =  employeeDF.join(employeeSalDF , "id")
joinDF: org.apache.spark.sql.DataFrame = [id: int, name: string ... 4 more field
s]

scala> joinDF.show(10)
+----+-----+---+----+-----+-------+
|  id| name|age|year|month| salary|
+----+-----+---+----+-----+-------+
|1201|  ali| 34|1395|   11|4500000|
|1201|  ali| 34|1395|   12|6500000|
|1201|  ali| 34|1396|    1|2500000|
|1201|  ali| 34|1396|    2|5000000|
|1210|parsa| 39|1395|   11|3900000|
|1210|parsa| 39|1395|   12|5700000|
|1210|parsa| 39|1396|    1|3900000|
|1210|parsa| 39|1396|    2|4700000|
|1208|bijan| 27|1395|   11|4200000|
|1208|bijan| 27|1395|   12|6000000|
+----+-----+---+----+-----+-------+
only showing top 10 rows
```

DataFrame-JOIN



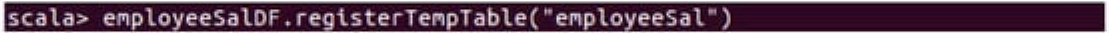
```
scala> employeeDF.join(employeeSalDF , "id").select("name","year","month","sala
ry").show(10)
+-----+----+-----+-------+
| name|year|month| salary|
+-----+----+-----+-------+
|  ali|1395|   11|4500000|
|  ali|1395|   12|6500000|
|  ali|1396|    1|2500000|
|  ali|1396|    2|5000000|
|parsa|1395|   11|3900000|
|parsa|1395|   12|5700000|
|parsa|1396|    1|3900000|
|parsa|1396|    2|4700000|
|bijan|1395|   11|4200000|
|bijan|1395|   12|6000000|
+-----+----+-----+-------+
only showing top 10 rows
```

DataFrame-JOIN

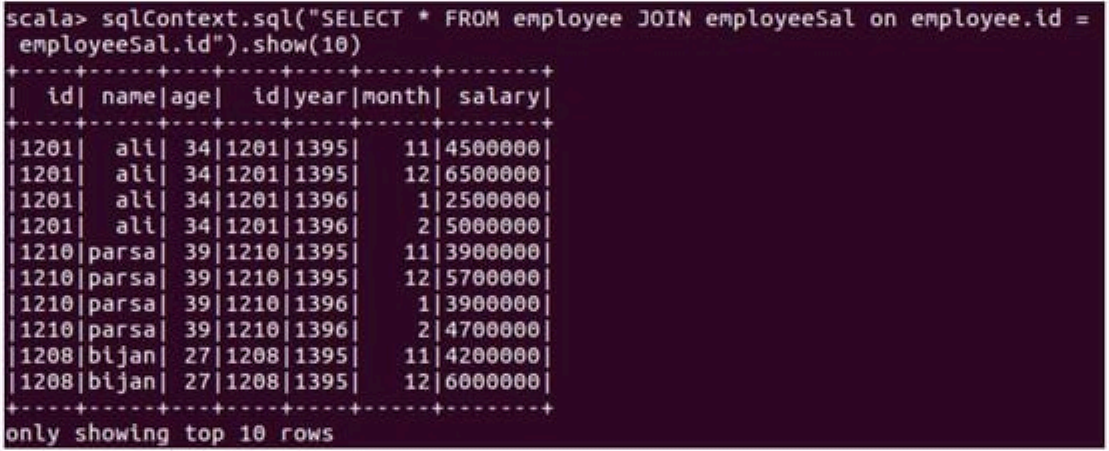
- A join() operation will join two dataframes based on some common column which in the previous example was the column id from employeeDF and employeeSalDF. But, what if the column to join to had different names? In such a case, you can explicitly specify the column from each dataframe on which to join.

```
// DataFrame Query: Join on explicit columns
FirstDF.join(secondDF, FirstDF(“f_id") === secondDF (“s_id"))
.show(10)
```

DataFrame-JOIN



```
scala> employeeSalDF.registerTempTable("employeeSal")
```



```
scala> sqlContext.sql("SELECT * FROM employee JOIN employeeSal on employee.id =
 employeeSal.id").show(10)
+----+-----+---+----+----+-----+-------+
|  id| name|age|  id|year|month| salary|
+----+-----+---+----+----+-----+-------+
|1201|  ali| 34|1201|1395|   11|4500000|
|1201|  ali| 34|1201|1395|   12|6500000|
|1201|  ali| 34|1201|1396|    1|2500000|
|1201|  ali| 34|1201|1396|    2|5000000|
|1210|parsa| 39|1210|1395|   11|3900000|
|1210|parsa| 39|1210|1395|   12|5700000|
|1210|parsa| 39|1210|1396|    1|3900000|
|1210|parsa| 39|1210|1396|    2|4700000|
|1208|bijan| 27|1208|1395|   11|4200000|
|1208|bijan| 27|1208|1395|   12|6000000|
+----+-----+---+----+----+-----+-------+
only showing top 10 rows
```

DataFrame- lef outer Join , right outer Join

- Spark supports various types of joins namely: **inner, cross, outer, full, full_outer, left, left_outer, right, right_outer, left_semi, left_anti**.

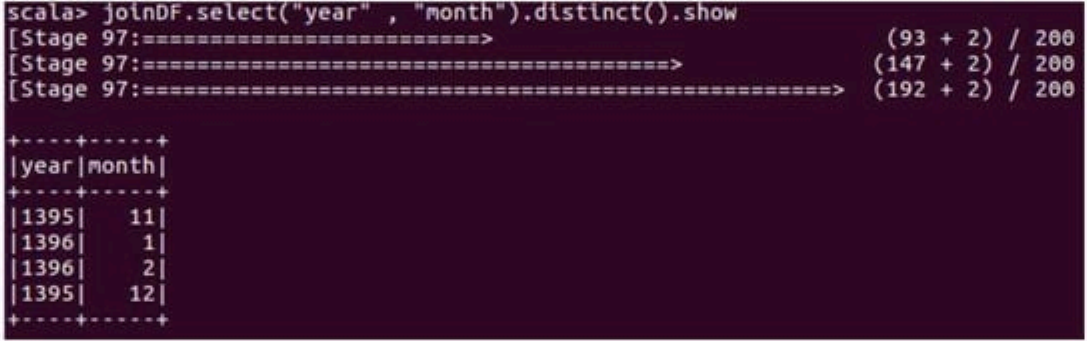
```
// DataFrame Query: Left Outer Join
FirstDF.join(secondDF, Seq("id"), "left_outer") .show(10)

// DataFrame Query: Right Outer Join
FirstDF.join(secondDF, Seq("id"), "right_outer") .show(10)
```

DataFrame-Union,Intersection

```
val unionDF = FirstDF.union(secondDF)
```

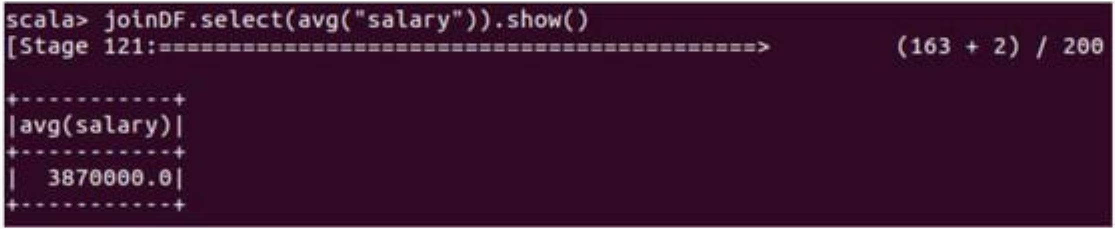
```
val intersectionDF = FirstDF.intersect(secondDF)
```

DataFrame-Distinct



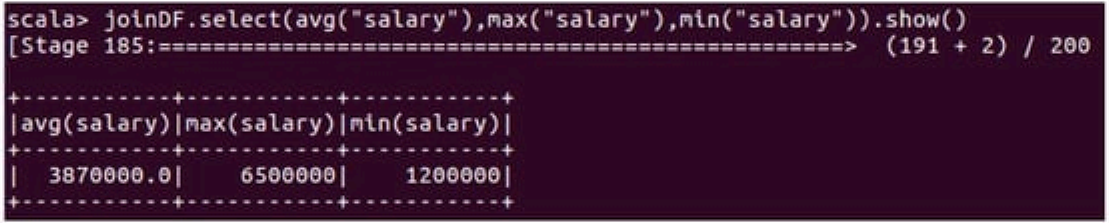
```
scala> joinDF.select("year" , "month").distinct().show
[Stage 97:=========================>                             (93 + 2) / 200]
[Stage 97:=======================================>              (147 + 2) / 200]
[Stage 97:===================================================>  (192 + 2) / 200

+----+-----+
|year|month|
+----+-----+
|1395|   11|
|1396|    1|
|1396|    2|
|1395|   12|
+----+-----+
```

DataFrame-Avg



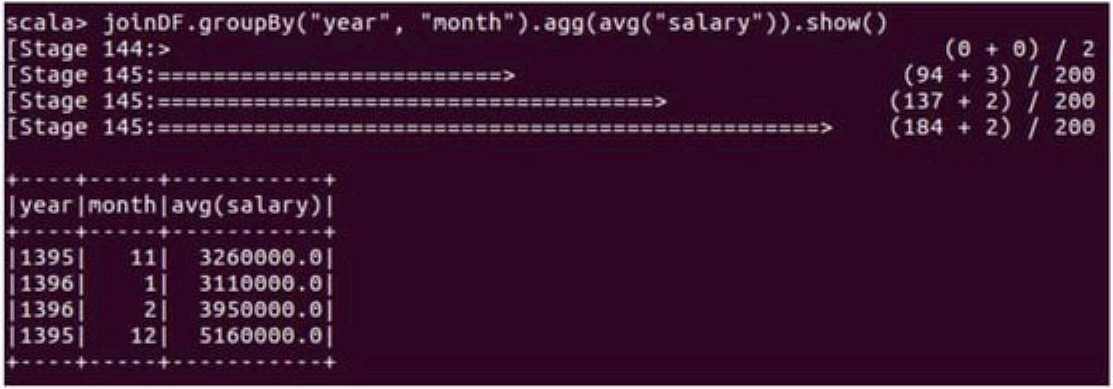
```
scala> joinDF.select(avg("salary")).show()
[Stage 121:===========================================>         (163 + 2) / 200

+-----------+
|avg(salary)|
+-----------+
|  3870000.0|
+-----------+
```

DataFrame-Avg,max,min



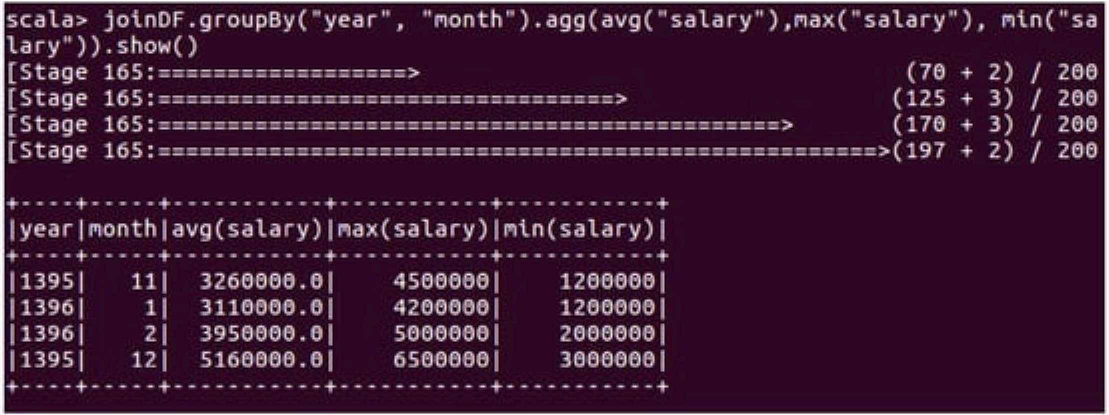
```
scala> joinDF.select(avg("salary"),max("salary"),min("salary")).show()
[Stage 185:==================================================>  (191 + 2) / 200

+-----------+-----------+-----------+
|avg(salary)|max(salary)|min(salary)|
+-----------+-----------+-----------+
|  3870000.0|    6500000|    1200000|
+-----------+-----------+-----------+
```

DataFrame-Aggregation



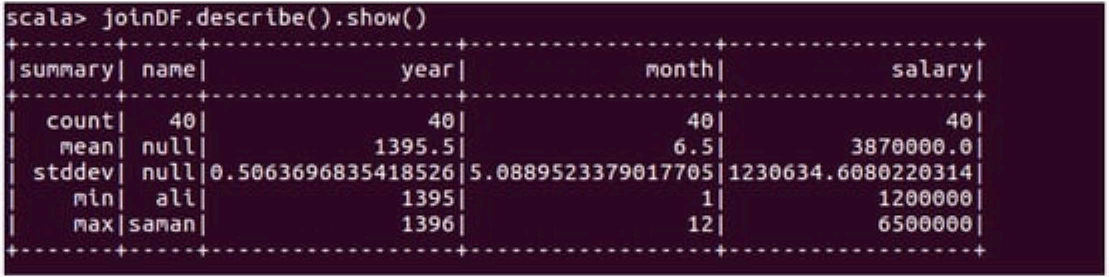
```
scala> joinDF.groupBy("year", "month").agg(avg("salary")).show()
[Stage 144:>                                                          (0 + 0) / 2
[Stage 145:======================>                                   (94 + 3) / 200
[Stage 145:================================>                        (137 + 2) / 200
[Stage 145:===========================================>             (184 + 2) / 200

+----+-----+-----------+
|year|month|avg(salary)|
+----+-----+-----------+
|1395|   11|  3260000.0|
|1396|    1|  3110000.0|
|1396|    2|  3950000.0|
|1395|   12|  5160000.0|
+----+-----+-----------+
```

DataFrame-Aggregation



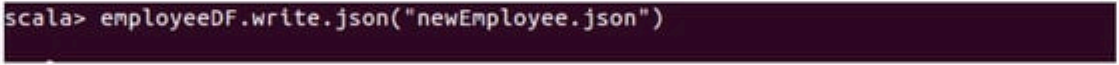
```
scala> joinDF.groupBy("year", "month").agg(avg("salary"),max("salary"), min("sa
lary")).show()
[Stage 165:==================>                                   (70 + 2) / 200
[Stage 165:=================================>                   (125 + 3) / 200
[Stage 165:=============================================>       (125 + 3) / 200
[Stage 165:====================================================>(197 + 2) / 200

+----+-----+-----------+-----------+-----------+
|year|month|avg(salary)|max(salary)|min(salary)|
+----+-----+-----------+-----------+-----------+
|1395|   11|  3260000.0|    4500000|    1200000|
|1396|    1|  3110000.0|    4200000|    1200000|
|1396|    2|  3950000.0|    5000000|    2000000|
|1395|   12|  5160000.0|    6500000|    3000000|
+----+-----+-----------+-----------+-----------+
```

DataFrame-describe

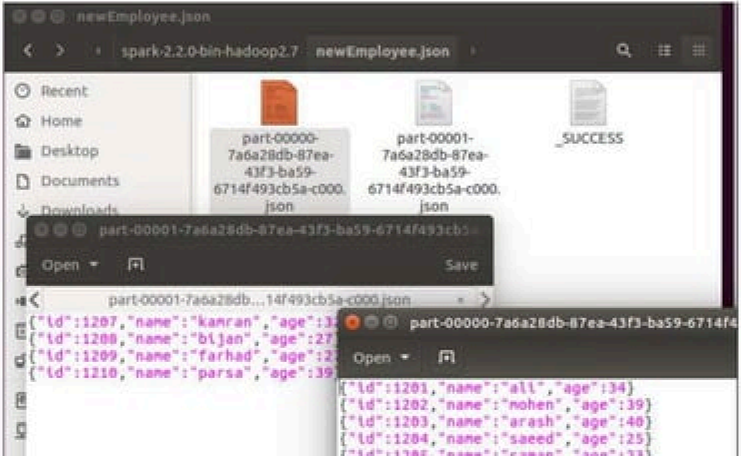


```
scala> joinDF.describe().show()
+-------+-----+------------------+------------------+------------------+
|summary| name|              year|             month|            salary|
+-------+-----+------------------+------------------+------------------+
|  count|   40|                40|                40|                40|
|   mean| null|            1395.5|               6.5|         3870000.0|
| stddev| null|0.5063696835418526|5.0889523379017705|1230634.6080220314|
|    min|  ali|              1395|                 1|           1200000|
|    max|saman|              1396|                12|           6500000|
+-------+-----+------------------+------------------+------------------+
```

DataFrame- write JSON



```
scala> employeeDF.write.json("newEmployee.json")
```

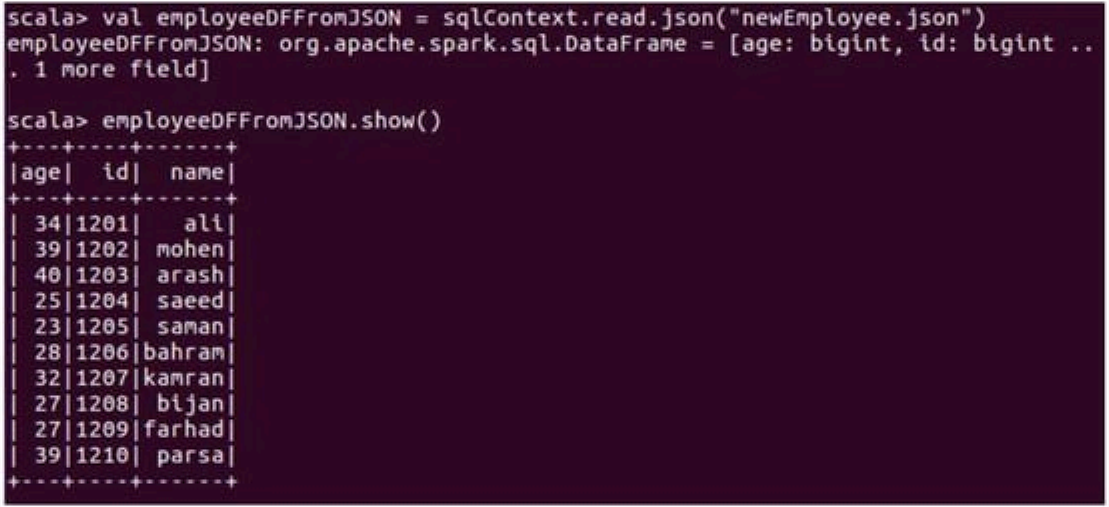


newEmployee.json
spark-2.2.0-bin-hadoop2.7
newEmployee.json
Recent
Home
Desktop
Documents
Downloads
part-00000-7a6a28db-87ea-43f3-ba59-6714f493cb5a-c000.json
part-00001-7a6a28db-87ea-43f3-ba59-6714f493cb5a-c000.json
_SUCCESS
part-00001-7a6a28db-87ea-43f3-ba59-6714f493cb5
Open
Save
part-00001-7a6a28db....14f493cb5a-c000.json
part-00000-7a6a28db-87ea-43f3-ba59-6714f4
Open

```
{"id":1207,"name":"kamran","age":3
{"id":1208,"name":"bijan","age":27
{"id":1209,"name":"farhad","age":2
{"id":1210,"name":"parsa","age":39
```

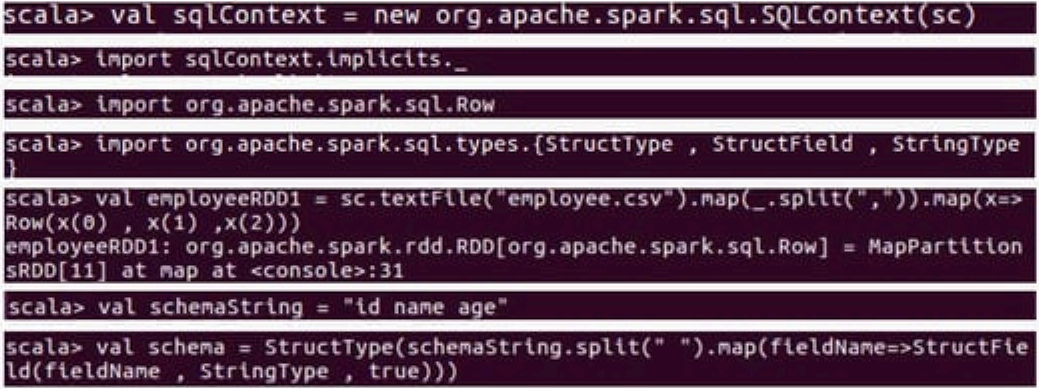
```
{"id":1201,"name":"ali","age":34}
{"id":1202,"name":"mohen","age":39}
{"id":1203,"name":"arash","age":40}
{"id":1204,"name":"saeed","age":25}
```

Read DataFrame From JSON



```
scala> val employeeDFFromJSON = sqlContext.read.json("newEmployee.json")
employeeDFFromJSON: org.apache.spark.sql.DataFrame = [age: bigint, id: bigint ..
. 1 more field]

scala> employeeDFFromJSON.show()
+---+----+------+
|age|  id|  name|
+---+----+------+
| 34|1201|   ali|
| 39|1202| mohen|
| 40|1203| arash|
| 25|1204| saeed|
| 23|1205| saman|
| 28|1206|bahram|
| 32|1207|kamran|
| 27|1208| bijan|
| 27|1209|farhad|
| 39|1210| parsa|
+---+----+------+
```

Programmatically Specifying the Schema



```
scala> val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

```
scala> import sqlContext.implicits._
```

```
scala> import org.apache.spark.sql.Row
```

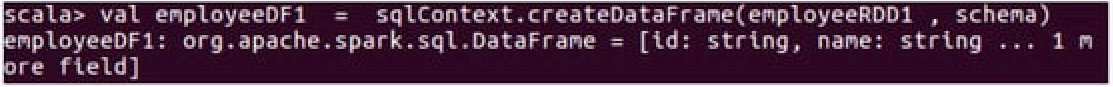
```
scala> import org.apache.spark.sql.types.{StructType , StructField , StringType
}
```

```
scala> val employeeRDD1 = sc.textFile("employee.csv").map(_.split(",")).map(x=>
Row(x(0) , x(1) ,x(2)))
employeeRDD1: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartition
sRDD[11] at map at <console>:31
```

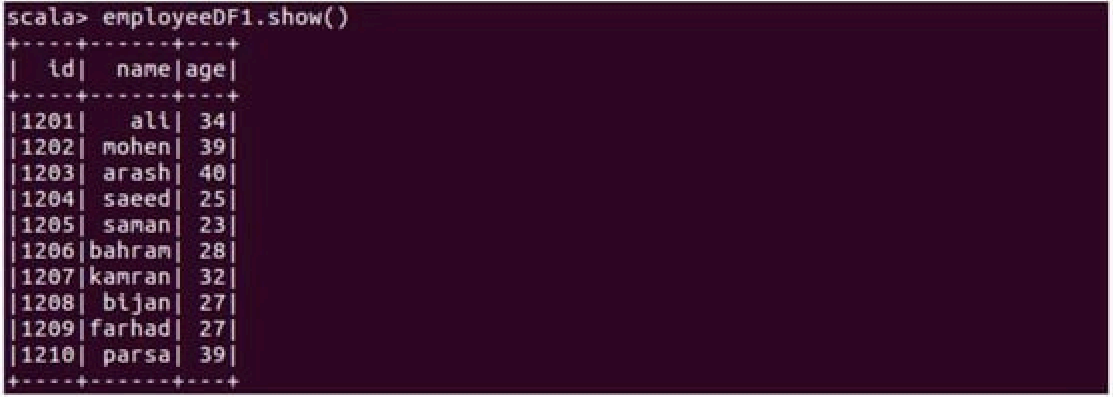
```
scala> val schemaString = "id name age"
```

```
scala> val schema = StructType(schemaString.split(" ").map(fieldName=>StructFie
ld(fieldName , StringType , true))
```

Programmatically Specifying the Schema

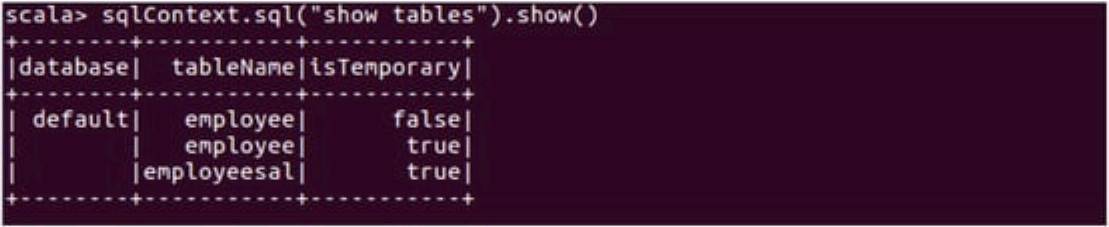


```
scala> val employeeDF1  =  sqlContext.createDataFrame(employeeRDD1 , schema)
employeeDF1: org.apache.spark.sql.DataFrame = [id: string, name: string ... 1 m
ore field]
```



```
scala> employeeDF1.show()
+----+------+---+
|  id|  name|age|
+----+------+---+
|1201|   ali| 34|
|1202| mohen| 39|
|1203| arash| 40|
|1204| saeed| 25|
|1205| saman| 23|
|1206|bahram| 28|
|1207|kamran| 32|
|1208| bijan| 27|
|1209|farhad| 27|
|1210| parsa| 39|
+----+------+---+
```

Catalog



```
scala> sqlContext.sql("show tables").show()
+--------+-----------+-----------+
|database|  tableName|isTemporary|
+--------+-----------+-----------+
| default|   employee|      false|
|        |   employee|       true|
|        |employeesal|       true|
+--------+-----------+-----------+
```

Spark SQL Data Sources

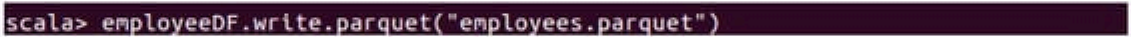
- A DataFrame interface allows different DataSources to work on Spark SQL
- It is a temporary table and can be operated as a normal RDD
- different types of data sources available in SparkSQL
  - JSON Datasets
  - Parquet Files
  - Hive Table
  - …

Parquet Files

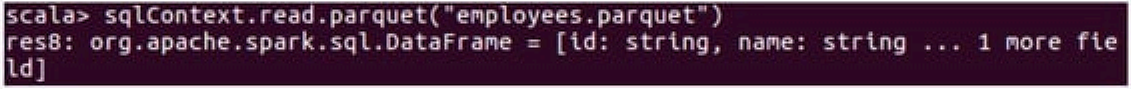
- Parquet is a columnar format
- supported by many data processing systems
- The advantages of having a columnar storage
  - Columnar storage limits IO operations
  - Columnar storage can fetch specific columns that you need to access
  - Columnar storage consumes less space
  - Columnar storage gives better-summarized data and follows type-specific encoding.

Parquet Files

- Spark SQL provides support for both reading and writing parquet files
- Spark SQL automatically capture the schema of the original data



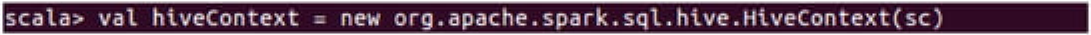
```
scala> employeeDF.write.parquet("employees.parquet")
```



```
scala> sqlContext.read.parquet("employees.parquet")
res8: org.apache.spark.sql.DataFrame = [id: string, name: string ... 1 more fie
ld]
```

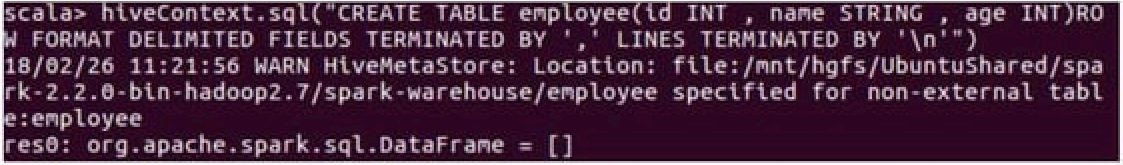
Hive

- to create and find tables in the HiveMetaStore and write queries on it use HiveContext
- HiveContext, inherits from SQLContext
- Users who do not have an existing Hive deployment can still create a HiveContext
- When not configured by the hive-site.xml, the context automatically creates a metastore called metastore_db and a folder called warehouse in the current directory.

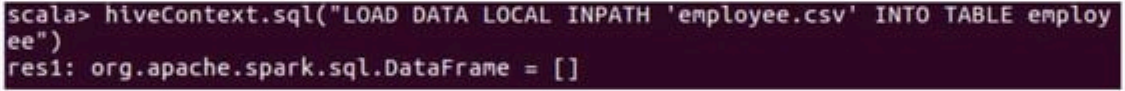


```
scala> val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

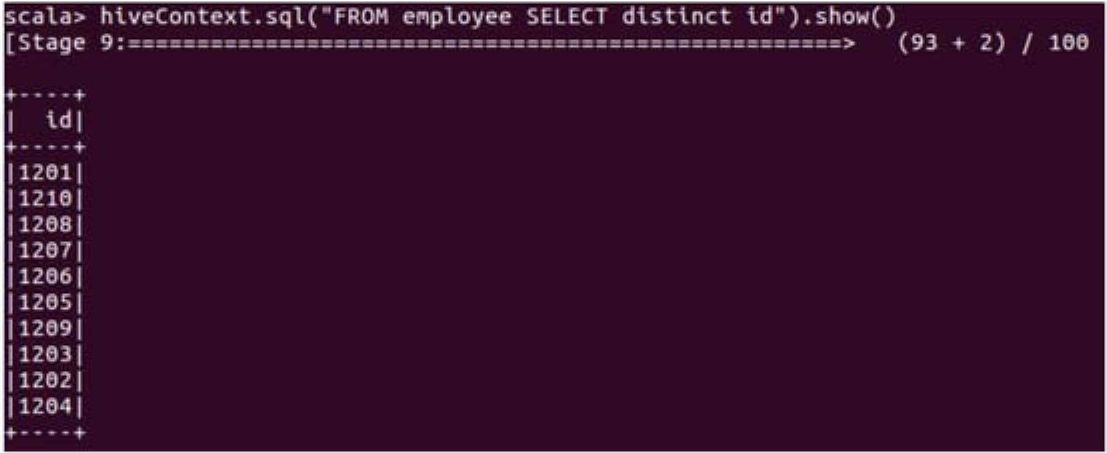
Create HiveContext / create Table on Hive



```
scala> hiveContext.sql("CREATE TABLE employee(id INT , name STRING , age INT)RO
W FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'")
18/02/26 11:21:56 WARN HiveMetaStore: Location: file:/mnt/hgfs/UbuntuShared/spa
rk-2.2.0-bin-hadoop2.7/spark-warehouse/employee specified for non-external tabl
e:employee
res0: org.apache.spark.sql.DataFrame = []
```



```
scala> hiveContext.sql("LOAD DATA LOCAL INPATH 'employee.csv' INTO TABLE employ
ee")
res1: org.apache.spark.sql.DataFrame = []
```

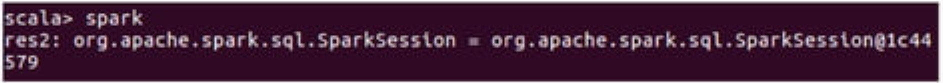


```
scala> hiveContext.sql("FROM employee SELECT distinct id").show()
[Stage 9:=====================================================>   (93 + 2) / 100

+----+
|  id|
+----+
|1201|
|1210|
|1208|
|1207|
|1206|
|1205|
|1209|
|1203|
|1202|
|1204|
+----+
```

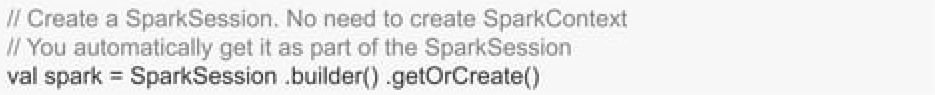
SparkSession

SparkSession

- Introduced in Saprk 2.0
- Enclapuslates SparkConf, SparkContext or SQLContext
- At this point you can use the *spark* variable as your instance object to access its public methods and instances for the duration of your Spark job



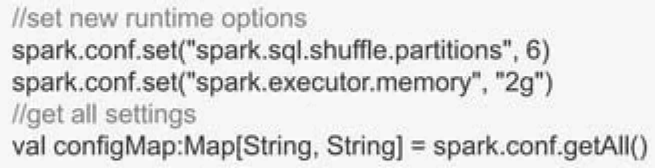
```
scala> spark
res2: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@1c44
579
```



```
// Create a SparkSession. No need to create SparkContext
// You automatically get it as part of the SparkSession
val spark = SparkSession .builder() .getOrCreate()
```

Configuring Spark’s Runtime Properties

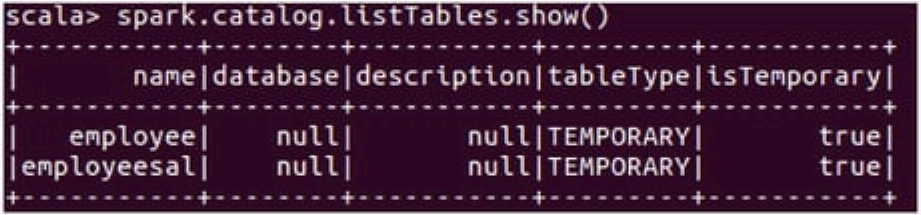
- Once the SparkSession is instantiated, you can configure Spark’s runtime config properties. For example, in this code snippet, we can alter the existing runtime config options. Since *configMap* is a collection, you can use all of Scala’s iterable methods to access the data.



```
//set new runtime options
spark.conf.set("spark.sql.shuffle.partitions", 6)
spark.conf.set("spark.executor.memory", "2g")
//get all settings
val configMap:Map[String, String] = spark.conf.getAll()
```

Accessing Catalog Metadata through SparkSession

- Often, you may want to access and peruse the underlying catalog metadata. SparkSession exposes “catalog” as a public instance that contains methods that work with the metastore (i.e data catalog). Since these methods return a Dataset, you can use Dataset API to access or view data. In this snippet, we access table names and list of databases



```
scala> spark.catalog.listTables.show()
+-----------+--------+-----------+---------+-----------+
|       name|database|description|tableType|isTemporary|
+-----------+--------+-----------+---------+-----------+
|   employee|    null|       null|TEMPORARY|       true|
|employeesal|    null|       null|TEMPORARY|       true|
+-----------+--------+-----------+---------+-----------+
```

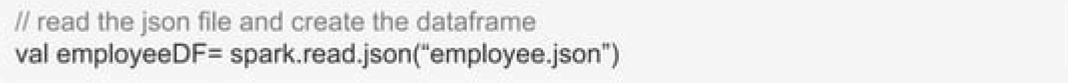
Creating Datasets and Dataframes

- There are a number of ways to create DataFrames and Datasets using SparkSession APIs
  - One quick way to generate a Dataset is by using the spark.range method

```
//create a Dataset using spark.range starting from 5 to 100, with increments of 5
val numDS = spark.range(5, 100, 5)
// create a DataFrame using spark.createDataFrame from a List or Seq
 val langPercentDF = spark.createDataFrame(List(("Scala", 35), ("Python", 30),
("R", 15), ("Java", 20)))
```

Reading JSON Data with SparkSession API

- I can read JSON or CVS or TXT file, or I can read a parquet table. For example, in this code snippet, we will read a JSON file of zip codes, which returns a DataFrame, a collection of generic Rows.



```
// read the json file and create the dataframe
val employeeDF= spark.read.json("employee.json")
```

Using Spark SQL with SparkSession

- Through SparkSession, you can access all of the Spark SQL functionality as you would through SQLContext. In the code sample below, we create a table against which we issue SQL queries

```
// Now create an SQL table and issue SQL queries against it without
// using the sqlContext but through the SparkSession object.
 // Creates a temporary view of the DataFrame
employeeDF.createOrReplaceTempView("employee")
val resultsDF = spark.sql("SELECT name , age FROM employee ") .show(10)
```

Saving and Reading from Hive table with SparkSession

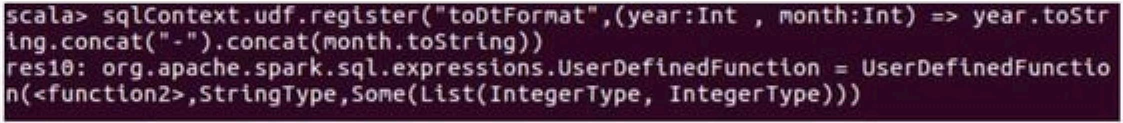
- to create a Hive table and issue queries against it using SparkSession object as you would with a HiveContext.

```
//drop the table if exists to get around existing table error spark.sql(“DROP TABLE IF EXISTS zips_hive_table”) //save as a hive table
spark.table(“zips_table”).write.saveAsTable(“zips_hive_table”) //make a similar query against the hive table val resultsHiveDF =
spark.sql(“SELECT city, pop, state, zip FROM zips_hive_table WHERE pop > 40000”) resultsHiveDF.show(10)
```

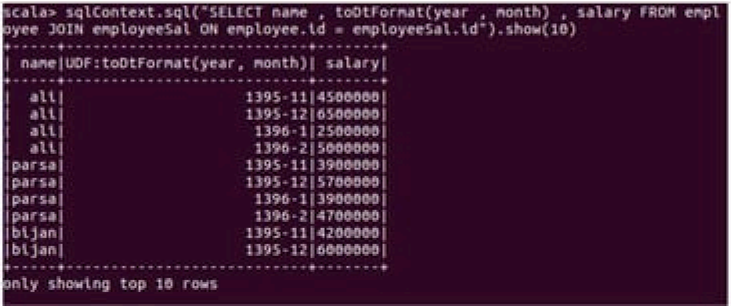
User Defined Function (UDF)

Register User Defined Function (UDF)

- register and use your own functions which are more commonly referred to as **User Defined Functions (UDF)**.

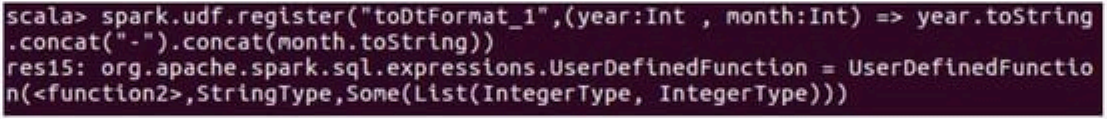


```
scala> sqlContext.udf.register("toDtFormat",(year:Int , month:Int) => year.toStr
ing.concat("-").concat(month.toString))
res10: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunctio
n(<function2>,StringType,Some(List(IntegerType, IntegerType)))
```



```
scala> sqlContext.sql("SELECT name , toDtFormat(year , month) , salary FROM empl
oyee JOIN employeeSal ON employee.id = employeeSal.id").show(10)
+-----+---------------------------+-------+
| name|UDF:toDtFormat(year, month)| salary|
+-----+---------------------------+-------+
|  ali|                    1395-11|4500000|
|  ali|                    1395-12|6500000|
|  ali|                     1396-1|2500000|
|  ali|                     1396-2|5000000|
|parsa|                    1395-11|3900000|
|parsa|                    1395-12|5700000|
|parsa|                     1396-1|3900000|
|parsa|                     1396-2|4700000|
|bijan|                    1395-11|4200000|
|bijan|                    1395-12|6000000|
+-----+---------------------------+-------+
only showing top 10 rows
```

Register UDF using SparkSession

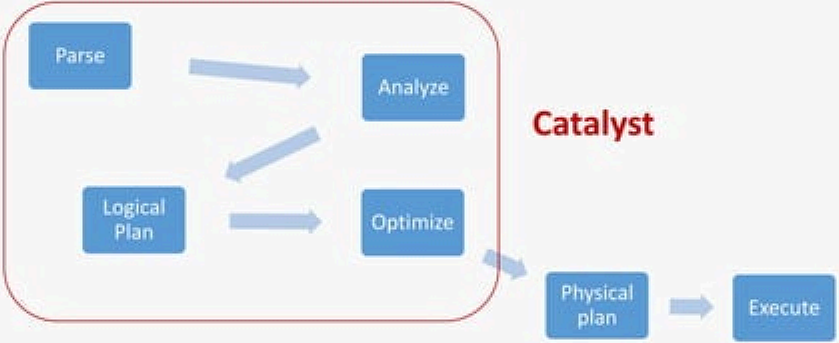


```
scala> spark.udf.register("toDtFormat_1",(year:Int , month:Int) => year.toString
.concat("-").concat(month.toString))
res15: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunctio
n(<function2>,StringType,Some(List(IntegerType, IntegerType)))
```

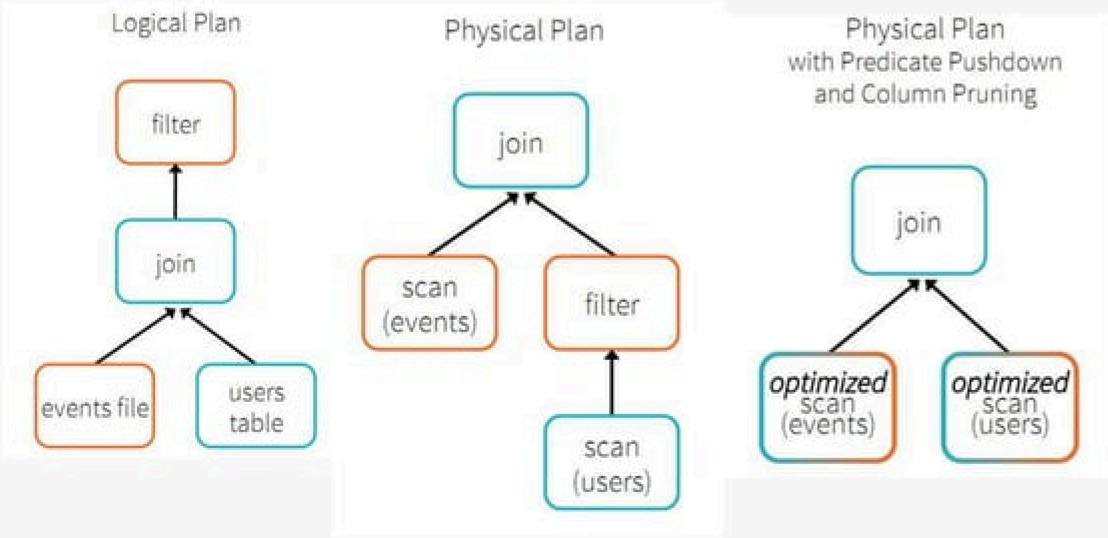
Spark Catalyst

Catalyst

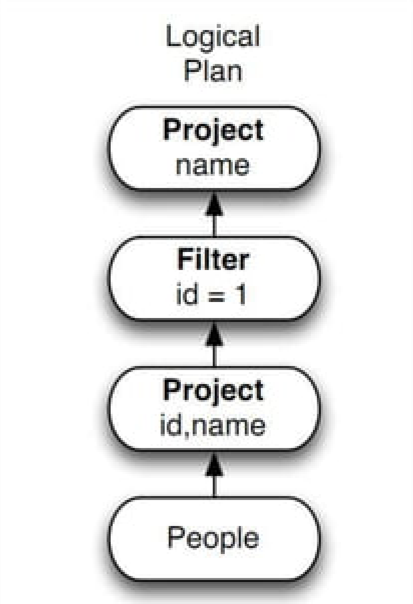
- Provides an execution planning framework



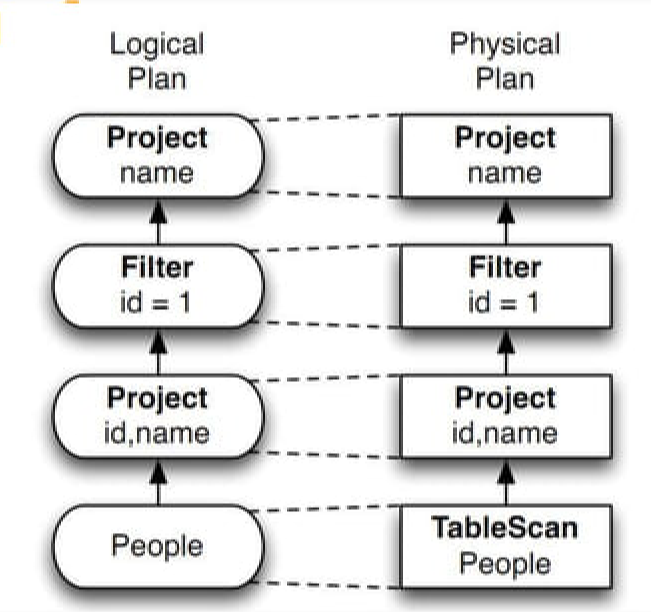
Catalyst


Logical Plan
filter
join
events file
users table
Physical Plan
join
scan (events)
filter
scan (users)
Physical Plan with Predicate Pushdown and Column Pruning
join
optimized scan (events)
optimized scan (users)

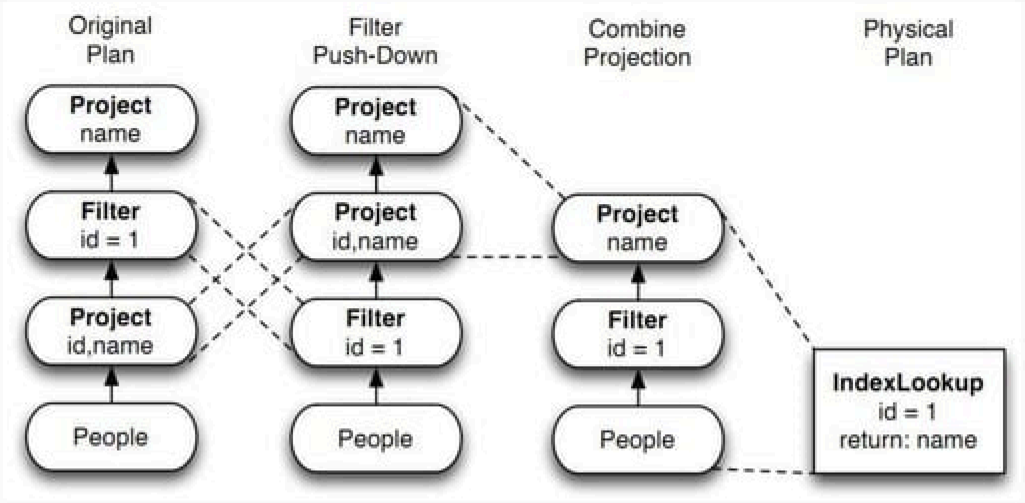
- SELECT name FROM(SELECT id,name FROM People) p WHERE p.id = 1



Naïve Query Pla

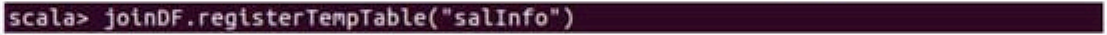


Optimized Query Plan

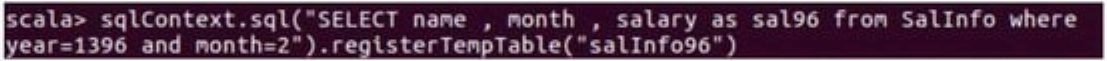


Hands On Code

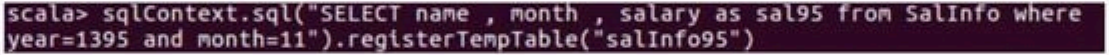
Ex1: Find the person who has the highest increase in salary



```
scala> joinDF.registerTempTable("salInfo")
```

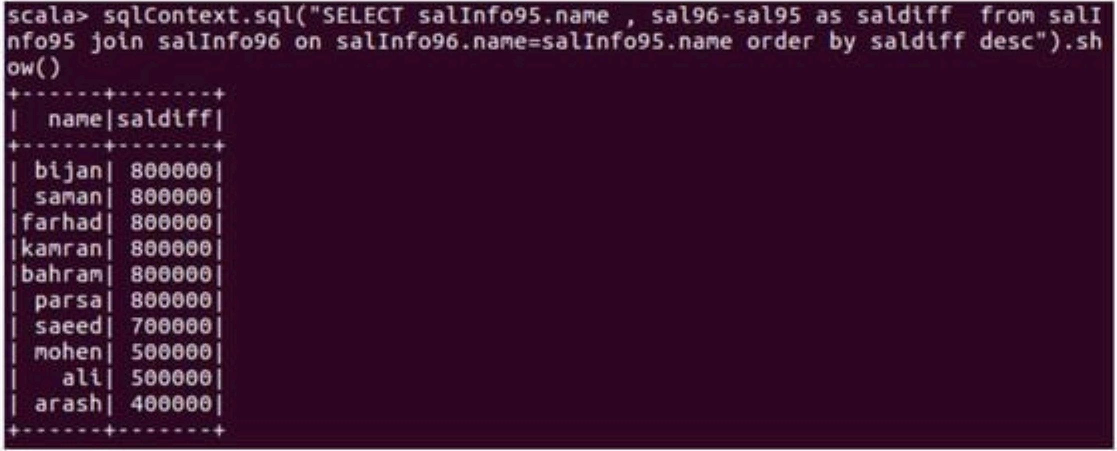


```
scala> sqlContext.sql("SELECT name , month , salary as sal96 from SalInfo where
year=1396 and month=2").registerTempTable("salInfo96")
```



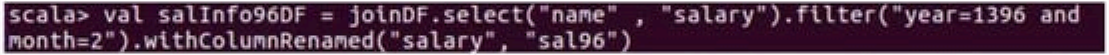
```
scala> sqlContext.sql("SELECT name , month , salary as sal95 from SalInfo where
year=1395 and month=11").registerTempTable("salInfo95")
```

Ex1 continued

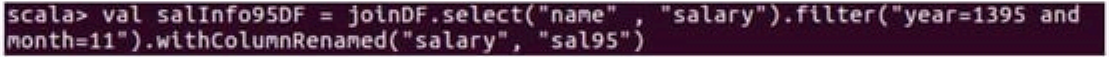


```
scala> sqlContext.sql("SELECT salInfo95.name , sal96-sal95 as saldiff  from salI
nfo95 join salInfo96 on salInfo96.name=salInfo95.name order by saldiff desc").sh
ow()
+------+-------+
|  name|saldiff|
+------+-------+
| bijan| 800000|
| saman| 800000|
|farhad| 800000|
|kamran| 800000|
|bahram| 800000|
| parsa| 800000|
| saeed| 700000|
| mohen| 500000|
|   ali| 500000|
| arash| 400000|
+------+-------+
```

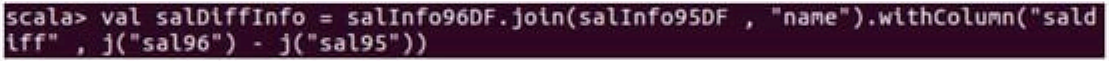
Ex2: Do the same using DataFrame API



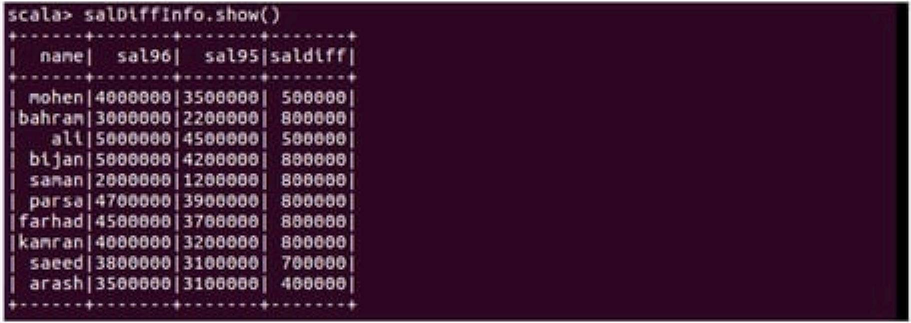
```
scala> val salInfo96DF = joinDF.select("name" , "salary").filter("year=1396 and
month=2").withColumnRenamed("salary", "sal96")
```



```
scala> val salInfo95DF = joinDF.select("name" , "salary").filter("year=1395 and
month=11").withColumnRenamed("salary", "sal95")
```

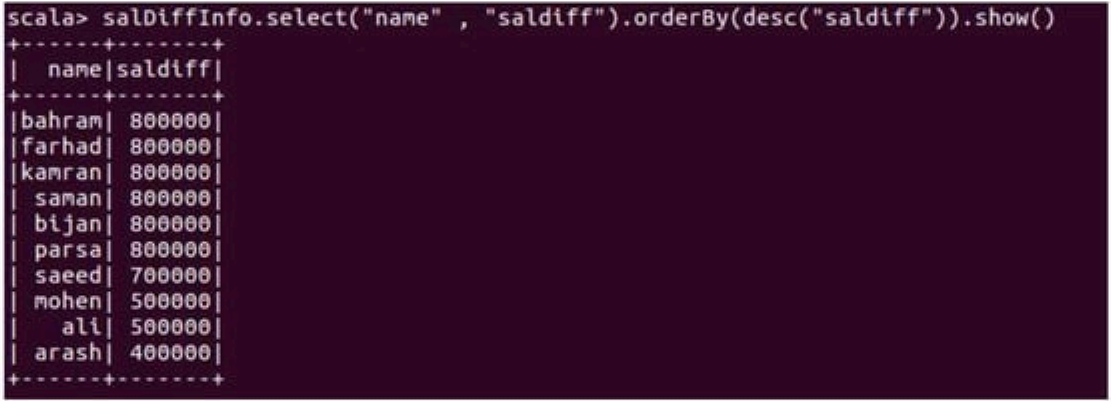


```
scala> val salDiffInfo = salInfo96DF.join(salInfo95DF , "name").withColumn("sald
iff" , j("sal96") - j("sal95"))
```



```
scala> salDiffInfo.show()
+------+-------+-------+-------+
|  name|  sal96|  sal95|saldiff|
+------+-------+-------+-------+
| mohen|4000000|3500000| 500000|
|bahram|3000000|2200000| 800000|
|   ali|5000000|4500000| 500000|
| bijan|5000000|4200000| 800000|
| saman|2000000|1200000| 800000|
| parsa|4700000|3900000| 800000|
|farhad|4500000|3700000| 800000|
|kamran|4000000|3200000| 800000|
| saeed|3800000|3100000| 700000|
| arash|3500000|3100000| 400000|
+------+-------+-------+-------+
```

Ex2 continued



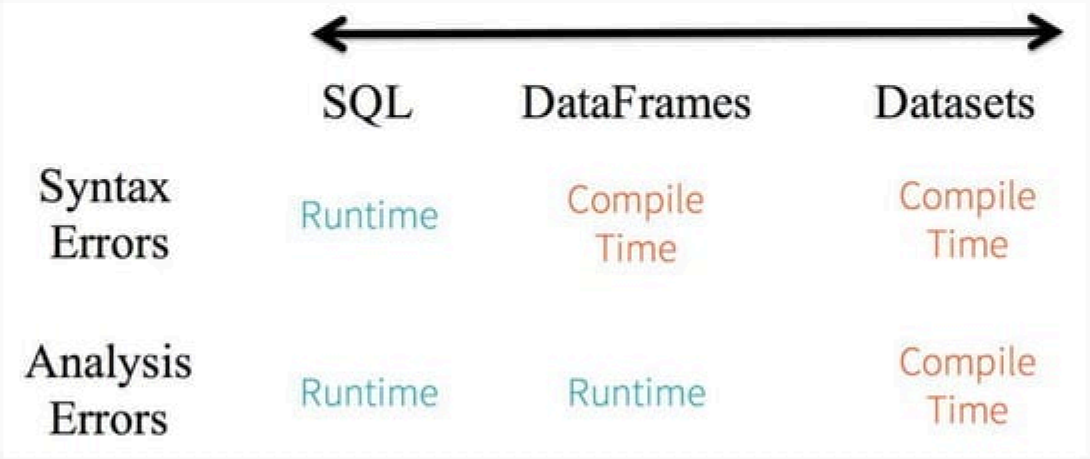
```
scala> salDiffInfo.select("name" , "saldiff").orderBy(desc("saldiff")).show()
+------+-------+
|  name|saldiff|
+------+-------+
|bahram| 800000|
|farhad| 800000|
|kamran| 800000|
| saman| 800000|
| bijan| 800000|
| parsa| 800000|
| saeed| 700000|
| mohen| 500000|
|   ali| 500000|
| arash| 400000|
+------+-------+
```

SQL or DataFrame API?

SQL or DataFrame?

- In Spark SQL string queries, you won’t know a syntax error until runtime
  - which could be costly
- In DataFrames and Datasets you can catch errors at compile time
  - which saves developer-time and costs
  - That is, if you invoke a function in DataFrame that is not part of the API, the compiler will catch it. However, it won’t detect a non-existing column name until runtime.

SQL or DataFrame?



	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

DataSet, DataFrame, RDD

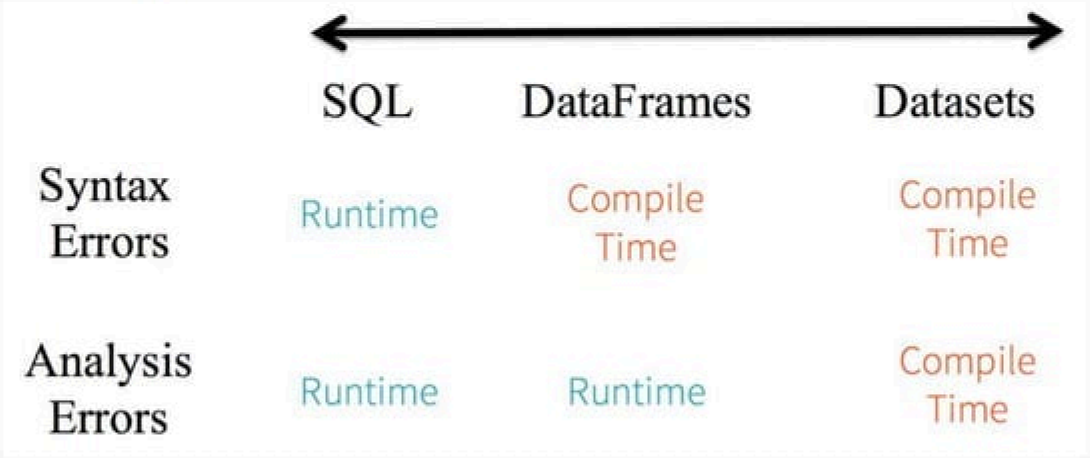
DataSets

- Added in Apache Spark 1.6
- provides the benefits of RDDs
  - strong typing, ability to use powerful lambda functions
- With the benefits of Spark SQL's optimized execution engine
- A user can define a Dataset (JVM objects)and then manipulate them using functional transformations (map, flatMap, filter, etc.) similar to an RDD.
  - The benefits is that, unlike RDDs, these transformations are now applied on a structured and strongly typed distributed collection that allows Spark to leverage Spark SQL's execution engine for optimization

DataSets

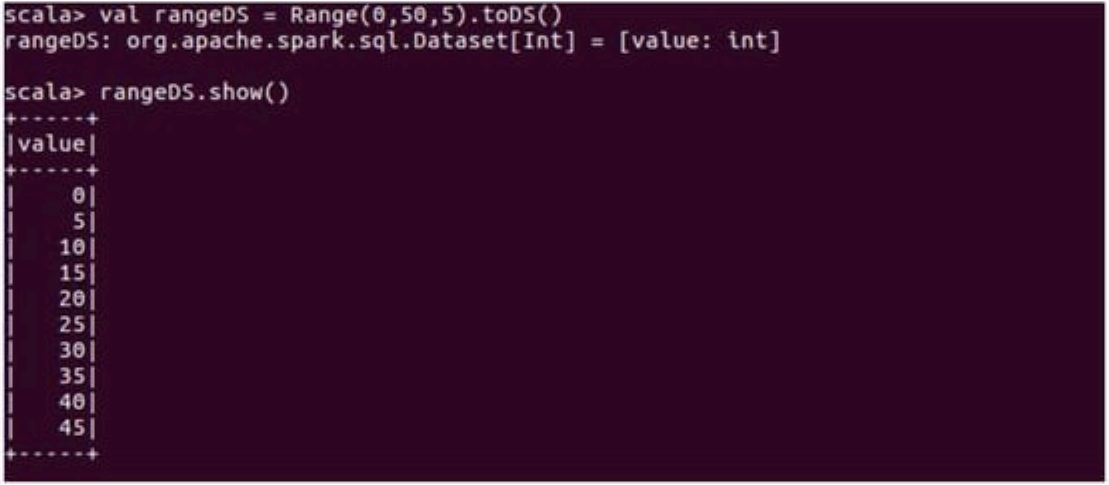
- provide an API that allows users to easily perform transformations on domain objects
- provide performance and robustness advantages of the Spark SQL execution engine
- compete to RDDs as they have overlapping functions
- DataFrame is an alias to *Dataset[Row]*.
- Datasets are optimized for typed engineering tasks, for which you want types checking and object-oriented programming interface,
- while DataFrames are faster for interactive analytics and close to SQL style

type-safety along syntax and analysis error



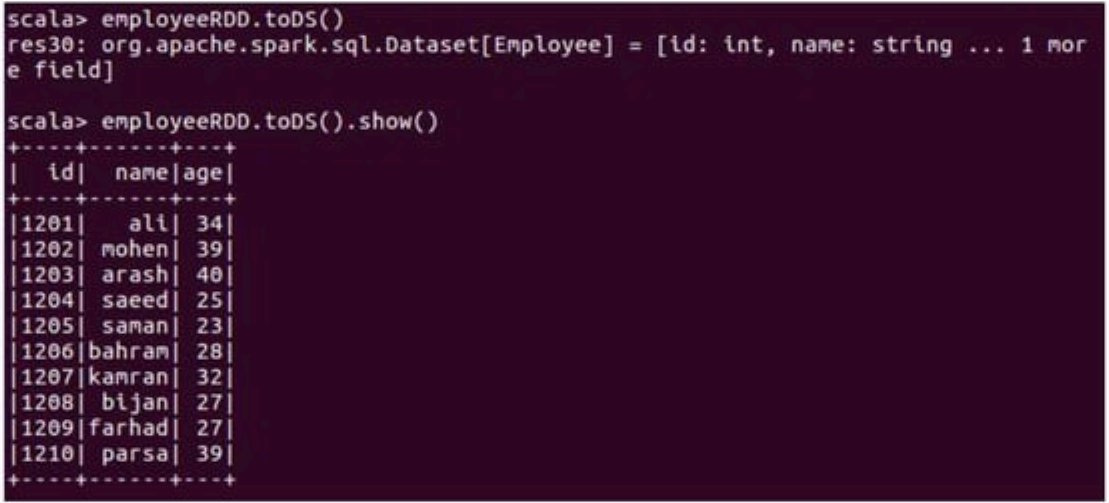
	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

Create DataSet from Collections



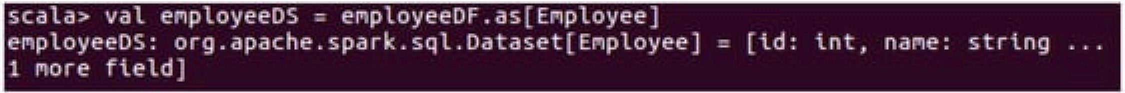
```
scala> val rangeDS = Range(0,50,5).toDS()
rangeDS: org.apache.spark.sql.Dataset[Int] = [value: int]

scala> rangeDS.show()
+-----+
|value|
+-----+
|    0|
|    5|
|   10|
|   15|
|   20|
|   25|
|   30|
|   35|
|   40|
|   45|
+-----+
```

Create DataSet from RDD

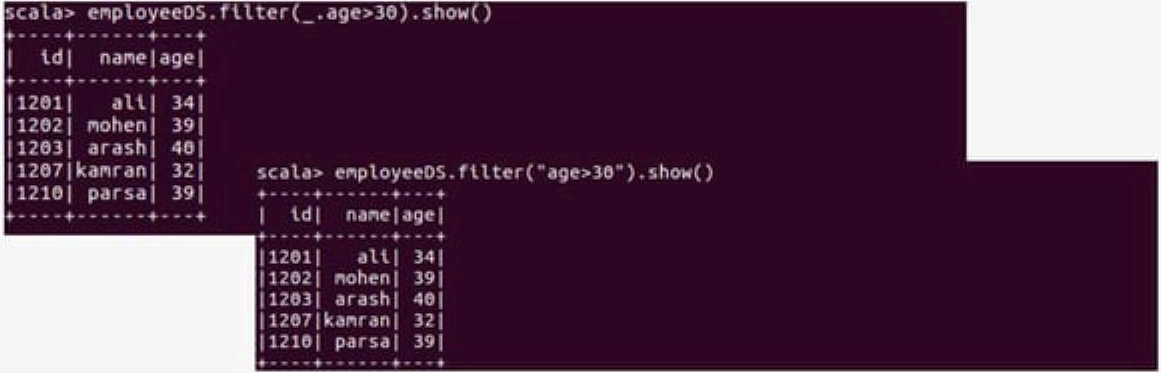


```
scala> employeeRDD.toDS()
res30: org.apache.spark.sql.Dataset[Employee] = [id: int, name: string ... 1 mor
e field]

scala> employeeRDD.toDS().show()
+----+------+---+
|  id|  name|age|
+----+------+---+
|1201|   ali| 34|
|1202| mohen| 39|
|1203| arash| 40|
|1204| saeed| 25|
|1205| saman| 23|
|1206|bahram| 28|
|1207|kamran| 32|
|1208| bijan| 27|
|1209|farhad| 27|
|1210| parsa| 39|
+----+------+---+
```

Create DataSet from DataFrame



```
scala> val employeeDS = employeeDF.as[Employee]
employeeDS: org.apache.spark.sql.Dataset[Employee] = [id: int, name: string ...
1 more field]
```



```
scala> employeeDS.filter(_.age>30).show()
+----+------+---+
|  id|  name|age|
+----+------+---+
|1201|   ali| 34|
|1202| mohen| 39|
|1203| arash| 40|
|1207|kamran| 32|
|1210| parsa| 39|
+----+------+---+
```

```
scala> employeeDS.filter("age>30").show()
+----+------+---+
|  id|  name|age|
+----+------+---+
|1201|   ali| 34|
|1202| mohen| 39|
|1203| arash| 40|
|1207|kamran| 32|
|1210| parsa| 39|
+----+------+---+
```

RDDs, DataFrames and DataSets

- You can seamlessly move between DataFrame or Dataset and RDDs at will—by simple API method calls—and DataFrames and Datasets are built on top of RDDs.

RDDs

- the primary user-facing API in Spark since its inception
- At the core, an RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API
- Offers
    - Transformations
    - Actions

When to use RDDs

- your data is **unstructured**, for example, binary (media) streams or text streams
- you want to control your dataset and use **low-level** transformations and actions
- your **data types cannot be serialized with Encoders** (an optimized approach that uses runtime code generation to build custom bytecode for serialization and deserialization)
- you are **ok to miss optimizations** for DataFrames and Datasets for structured and semi-structured data that are available out of the box
- you don’t care about the **schema**, columnar format and ready to use functional programming constructs

DataFrames

- Like an RDD, a DataFrame is an immutable distributed collection of data
- Unlike an RDD, data is organized into named columns, like a table in a relational database
- Designed to make large data sets processing even easier
- DataFrame allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction
- provides a domain specific language API to manipulate your distributed data

When to use DataFrames

- your data is **structured** (RDBMS input) or **semi-structured** (json, csv)
- you want to get the **best performance** gained from SQL's optimized execution engine (Catalyst optimizer and Tungsten's efficient code generation)
- you **need to run hive queries**
- you appreciate **domain specific language** API (.groupBy, .agg, .orderBy)
- you are using **R** or **Python**

Datasets

- Since Spark 2.0 DataFrame is an *alias* for a collection of generic objects *Dataset[Row]*, where a *Row* is a generic ***untyped*** JVM object

When should I use DataFrames or Datasets?

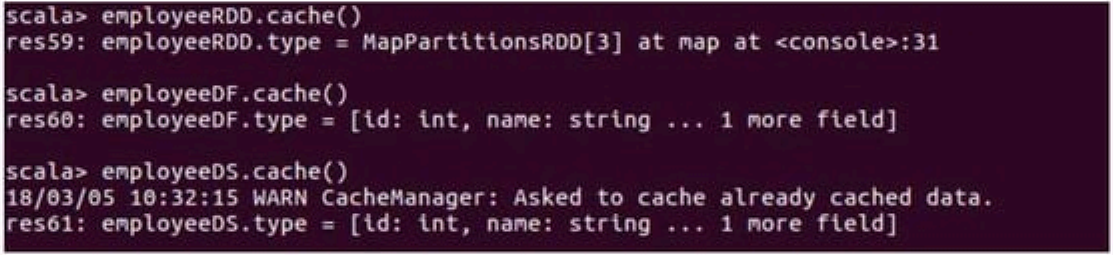
- your data is **structured** or **semi-structured**
- you appreciate **type-safety** at a compile time and a **strong-typed** API
- you need **good performance** (mostly greater than RDD), but not the best one (usually lower than DataFrames)

Caching

Caching Tables In-Memory

- Spark also supports pulling data sets into **a cluster-wide in-memory cache**.
- very useful when data is accessed repeatedly, such as when querying a small “hot” dataset or when running an iterative algorithm like PageRank
- The interesting part is that these same functions can be used on very large data sets, even when they are striped across tens or hundreds of nodes.

Cache RDD/DataFrame/DataSet



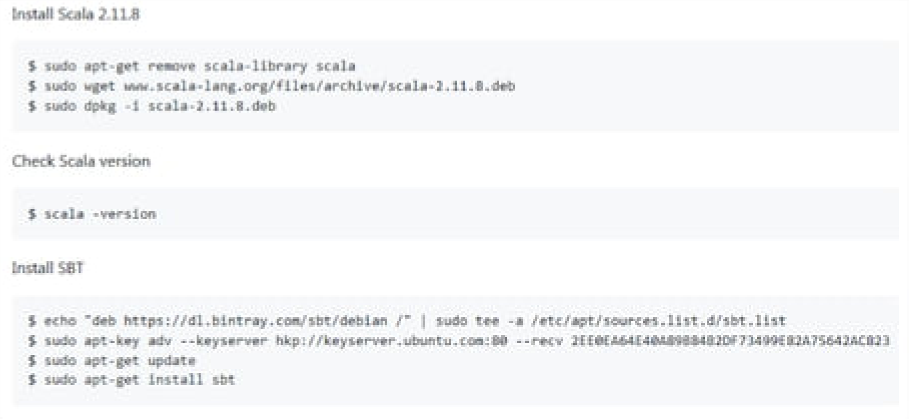
```
scala> employeeRDD.cache()
res59: employeeRDD.type = MapPartitionsRDD[3] at map at <console>:31

scala> employeeDF.cache()
res60: employeeDF.type = [id: int, name: string ... 1 more field]

scala> employeeDS.cache()
18/03/05 10:32:15 WARN CacheManager: Asked to cache already cached data.
res61: employeeDS.type = [id: int, name: string ... 1 more field]
```

Self-Contained Applications

A simple application in Scala

- Write code
- Build and package it using sbt
    - This gives you a .jar file
- Execute it using spark-submit

Install sbt



Install Scala 2.11.8

```
$ sudo apt-get remove scala-library scala
$ sudo wget www.scala-lang.org/files/archive/scala-2.11.8.deb
$ sudo dpkg -i scala-2.11.8.deb
```

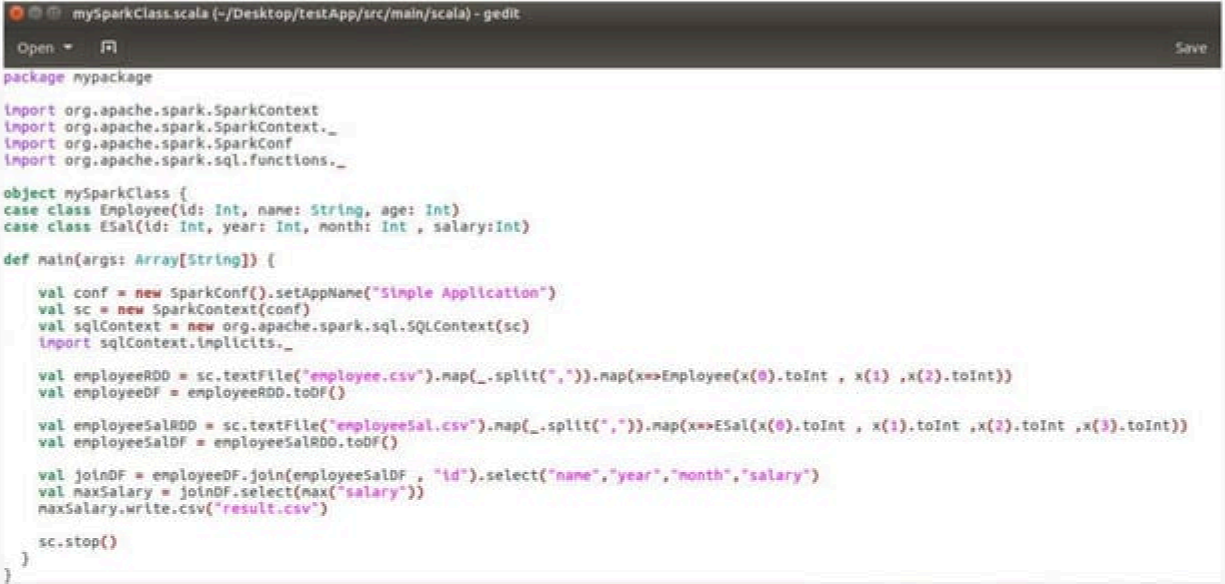
Check Scala version

```
$ scala -version
```

Install SBT

```
$ echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/sbt.list
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823
$ sudo apt-get update
$ sudo apt-get install sbt
```

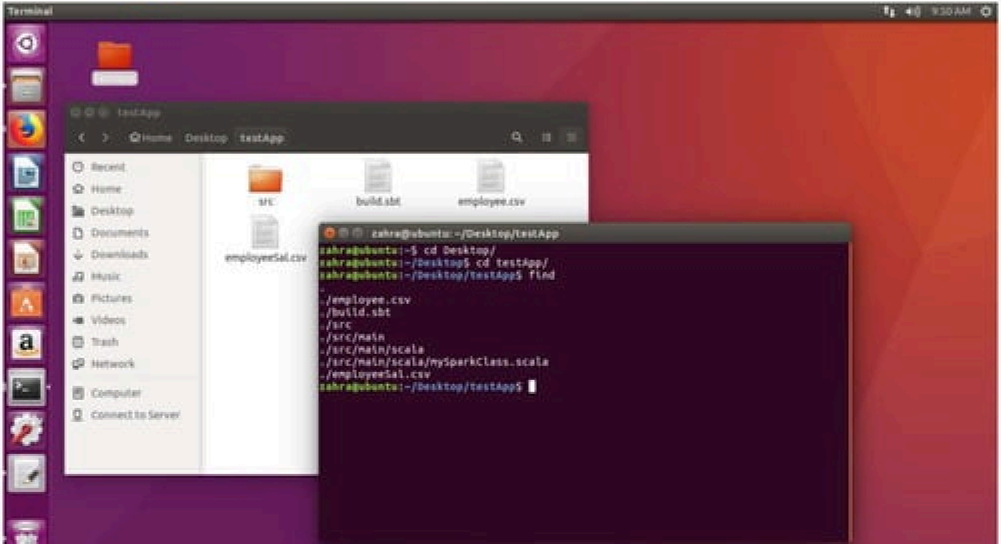
Write Code



mySparkClass.scala (~/Desktop/testApp/src/main/scala) - gedit
Open
Save

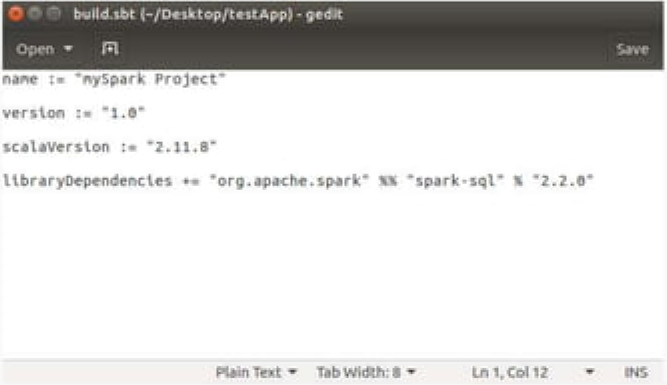
```scala
package mypackage

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.sql.functions._

object mySparkClass {
case class Employee(id: Int, name: String, age: Int)
case class ESal(id: Int, year: Int, month: Int , salary:Int)

def main(args: Array[String]) {

    val conf = new SparkConf().setAppName("Simple Application")
    val sc = new SparkContext(conf)
    val sqlContext = new org.apache.spark.sql.SQLContext(sc)
    import sqlContext.implicits._

    val employeeRDD = sc.textFile("employee.csv").map(_.split(",")).map(x=>Employee(x(0).toInt , x(1) ,x(2).toInt))
    val employeeDF = employeeRDD.toDF()

    val employeeSalRDD = sc.textFile("employeeSal.csv").map(_.split(",")).map(x=>ESal(x(0).toInt , x(1).toInt ,x(2).toInt ,x(3).toInt))
    val employeeSalDF = employeeSalRDD.toDF()

    val joinDF = employeeDF.join(employeeSalDF , "id").select("name","year","month","salary")
    val maxSalary = joinDF.select(max("salary"))
    maxSalary.write.csv("result.csv")

    sc.stop()
  }
}
```

Build a directory structure for the application



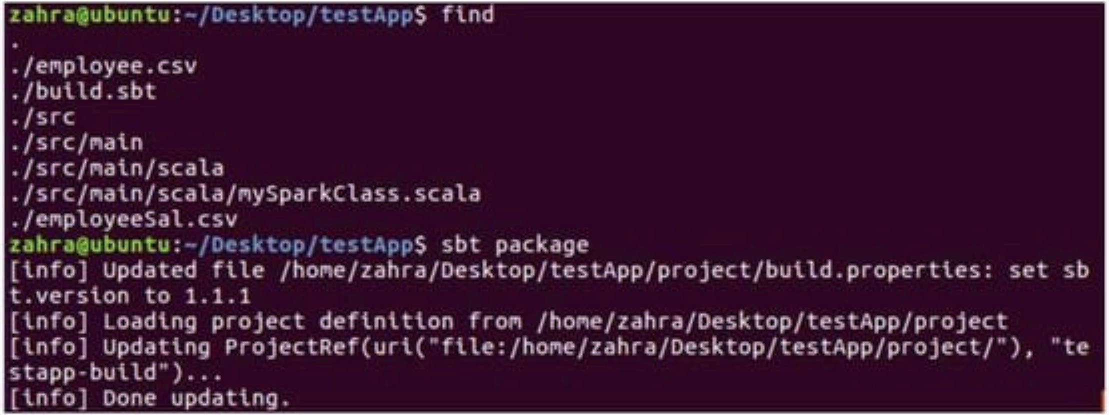
```
zahra@ubuntu:~$ cd Desktop/
zahra@ubuntu:~/Desktop$ cd testApp/
zahra@ubuntu:~/Desktop/testApp$ find
.
./employee.csv
./build.sbt
./src
./src/main
./src/main/scala
./src/main/scala/mySparkClass.scala
./employeeSal.csv
zahra@ubuntu:~/Desktop/testApp$ 
```

build.sbt



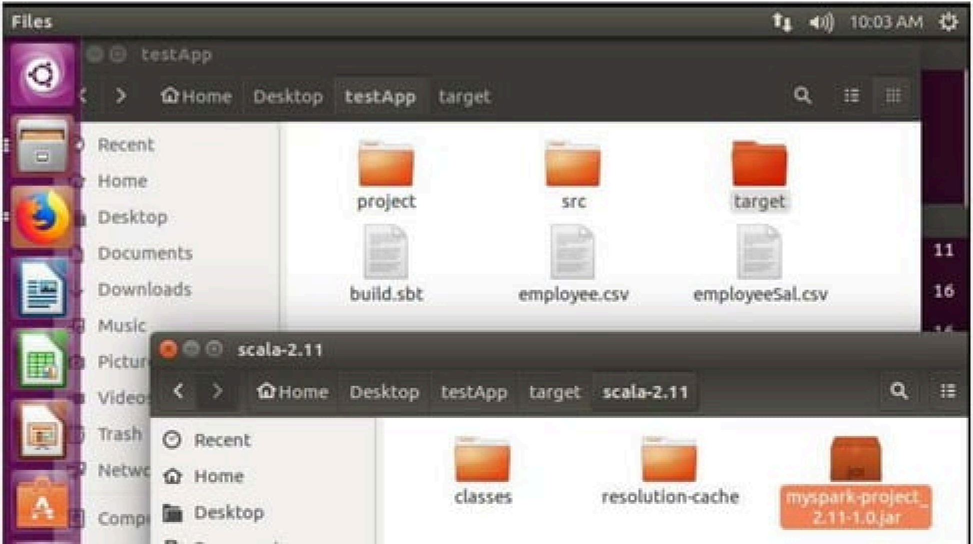
```
name := "mySpark Project"

version := "1.0"

scalaVersion := "2.11.8"

libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.2.0"
```

Use *sbt* to build and package the application

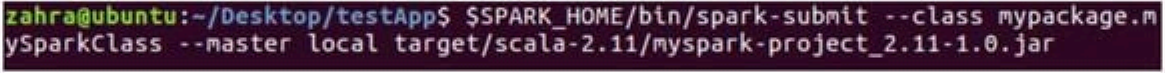


```
zahra@ubuntu:~/Desktop/testApp$ find
.
./employee.csv
./build.sbt
./src
./src/main
./src/main/scala
./src/main/scala/mySparkClass.scala
./employeeSal.csv
zahra@ubuntu:~/Desktop/testApp$ sbt package
[info] Updated file /home/zahra/Desktop/testApp/project/build.properties: set sb
t.version to 1.1.1
[info] Loading project definition from /home/zahra/Desktop/testApp/project
[info] Updating ProjectRef(uri("file:/home/zahra/Desktop/testApp/project/"), "te
stapp-build")...
[info] Done updating.
```

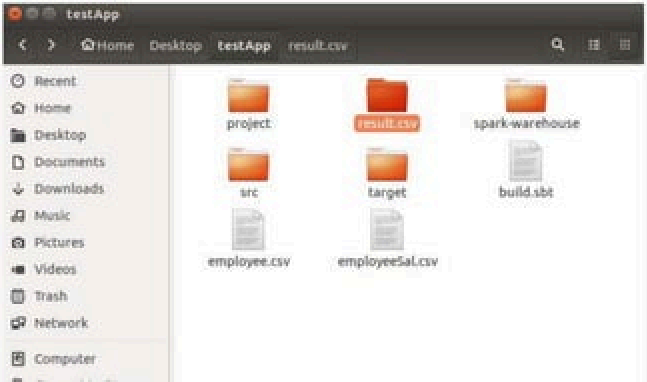
Resulting Jar file



Submit your Spark job

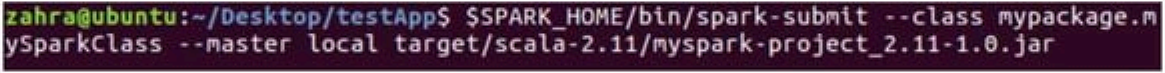


```
zahra@ubuntu:~/Desktop/testApp$ $SPARK_HOME/bin/spark-submit --class mypackage.m
ySparkClass --master local target/scala-2.11/myspark-project_2.11-1.0.jar
```



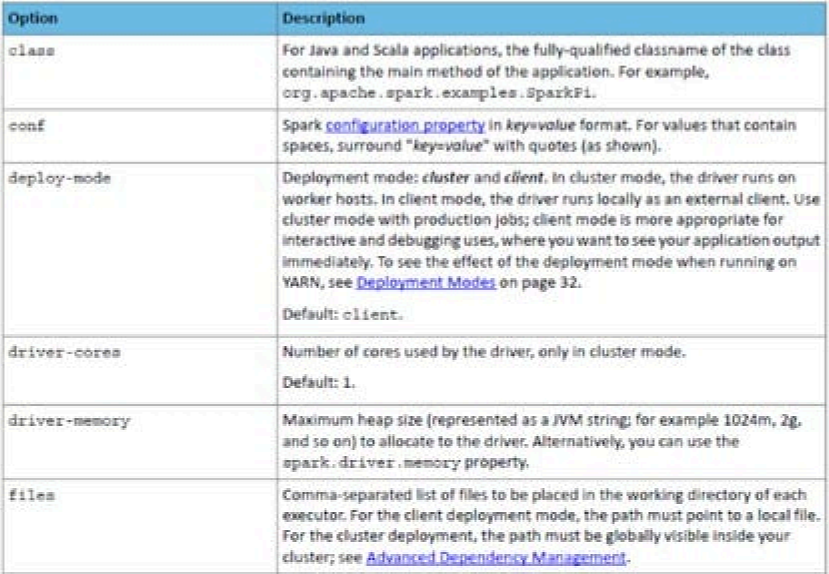
Submit your Spark job

- We submit a Spark application to run locally or on a Spark cluster.
- *master* is a Spark, Mesos or YARN cluster URL, or *local* to run the application in local machine.



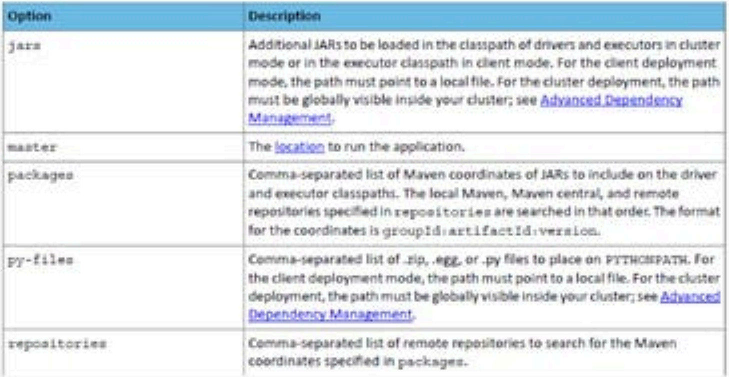
```
zahra@ubuntu:~/Desktop/testApp$ $SPARK_HOME/bin/spark-submit --class mypackage.m
ySparkClass --master local target/scala-2.11/myspark-project_2.11-1.0.jar
```

Spark-submit



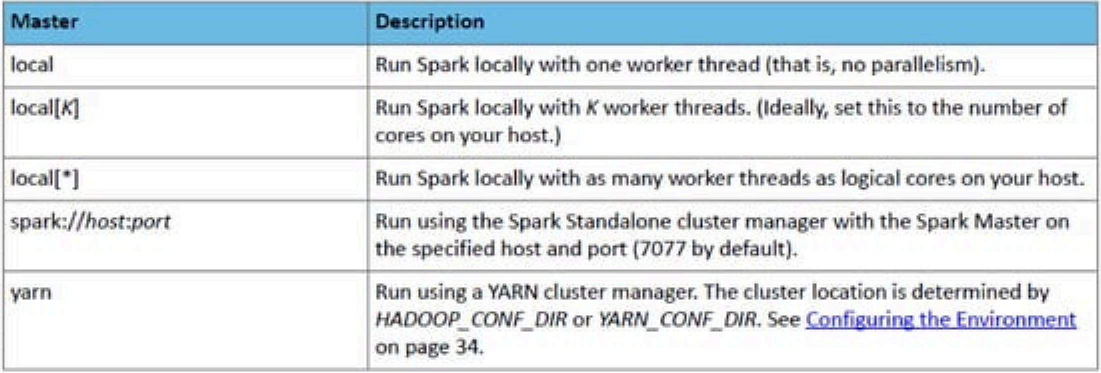
Option	Description
`class`	For Java and Scala applications, the fully-qualified classname of the class containing the main method of the application. For example, `org.apache.spark.examples.SparkPi`.
`conf`	Spark configuration property in *key=value* format. For values that contain spaces, surround "*key=value*" with quotes (as shown).
`deploy-mode`	Deployment mode: ***cluster*** and ***client***. In cluster mode, the driver runs on worker hosts. In client mode, the driver runs locally as an external client. Use cluster mode with production jobs; client mode is more appropriate for interactive and debugging uses, where you want to see your application output immediately. To see the effect of the deployment mode when running on YARN, see Deployment Modes on page 32. Default: `client`.
`driver-cores`	Number of cores used by the driver, only in cluster mode. Default: 1.
`driver-memory`	Maximum heap size (represented as a *JVM* string; for example 1024m, 2g, and so on) to allocate to the driver. Alternatively, you can use the `spark.driver.memory` property.
`files`	Comma-separated list of files to be placed in the working directory of each executor. For the client deployment mode, the path must point to a local file. For the cluster deployment, the path must be globally visible inside your cluster; see Advanced Dependency Management.

Spark-submit



Option	Description
`jars`	Additional JARs to be loaded in the classpath of drivers and executors in cluster mode or in the executor classpath in client mode. For the client deployment mode, the path must point to a local file. For the cluster deployment, the path must be globally visible inside your cluster; see Advanced Dependency Management.
`master`	The location to run the application.
`packages`	Comma-separated list of Maven coordinates of JARs to include on the driver and executor classpaths. The local Maven, Maven central, and remote repositories specified in `repositories` are searched in that order. The format for the coordinates is `groupId:artifactId:version`.
`py-files`	Comma-separated list of .zip, .egg, or .py files to place on `PYTHONPATH`. For the client deployment mode, the path must point to a local file. For the cluster deployment, the path must be globally visible inside your cluster; see Advanced Dependency Management.
`repositories`	Comma-separated list of remote repositories to search for the Maven coordinates specified in `packages`.

msater values



Master	Description
local	Run Spark locally with one worker thread (that is, no parallelism).
local[*K*]	Run Spark locally with *K* worker threads. (Ideally, set this to the number of cores on your host.)
local[*]	Run Spark locally with as many worker threads as logical cores on your host.
spark://*host:port*	Run using the Spark Standalone cluster manager with the Spark Master on the specified host and port (7077 by default).
yarn	Run using a YARN cluster manager. The cluster location is determined by *HADOOP_CONF_DIR* or *YARN_CONF_DIR*. See Configuring the Environment on page 34.

Finished ☺