

ISIT312 Big Data Management

# Spark Operations

Dr Fenghui Ren

School of Computing and Information Technology -  
University of Wollongong

# Spark Operations

## Outline

[The Programming Language Scala](#)

[Quick Start](#)

[Self Contained Application](#)

[Web User Interface](#)

[Operations on Resilient Distributed Datasets \(RDDs\)](#)

[Operations on Datasets](#)

[Operations on DataFrames](#)

[SQL Module](#)

# The Programming Language Scala

**Spark** has built-in APIs for **Java**, **Scala**, and **Python**, and is also integrated with **R**

Among all languages, **Scala** is the most supported language

Also, **Spark** project is implemented using **Scala**

Therefore, we choose **Scala as our working language in Spark**

**Scala** is a **Java**-like programming language which unifies object-oriented and functional programming

**Scala** is a pure object-oriented language in the sense that every value is an object

Types and behaviour of objects are described by classes

**Scala** is a functional programming language in the sense that every function is a value

Nesting of function definitions and higher-order functions are naturally supported

# The Programming Language Scala

## Hello World ! in Scala

```
object Hello {  
  def main(args: Array[String]) = {  
    println("Hello, world")  
  }  
}
```

Hello World ! in Scala

Instead of including `main` method, it can be extended with `App` trait

```
object Hello2 extends App {  
  println("Hello, world")  
}
```

Extending App trait

Using command line arguments

```
object HelloYou extends App {  
  if (args.size == 0)  
    println("Hello, you")  
  else  
    println("Hello, " + args(0))  
}
```

Command line arguments

# The Programming Language Scala

Difference between `var`, `val`, and `def`

```
var x = 7
x = x * 2
```

Variable

```
val x = 7
x = x * 2
'error: reassignment to val'
```

Value

```
def hello(name: String) = "Hello : " + name
hello("James") // "Hello : James"
hello("")      // "Hello : "
```

Function declaration

When `lazy` keyword is used then a value is only computed when it is needed

```
lazy val x = {
  println("calculating value of x")
  13 }
val y = {
  println("calculating value of y")
  20 }
```

Lazy evaluation

# The Programming Language Scala

## Defining a class

```
class Point(var x: Int, var y: Int) {
```

Class Point

```
  def move(dx: Int, dy: Int): Unit = {  
    x = x + dx  
    y = y + dy  
  }
```

Method move

```
  override def toString: String =  
    s"($x, $y)"  
}
```

Method toString

```
val point1 = new Point(2, 3)  
println(point1.x)           // 2  
println(point1)             // prints (2, 3)
```

Applications

# Spark Operations

## Outline

[The Programming Language Scala](#)

[Quick Start](#)

[Self Contained Application](#)

[Web User Interface](#)

[Operations on Resilient Distributed Datasets \(RDDs\)](#)

[Operations on Datasets](#)

[Operations on DataFrames](#)

[SQL Module](#)

# Quick Start

To open **Scala** version of **Spark shell** in standalone mode process the following command

## Starting Spark shell in standalone mode

```
bin/spark-shell --master local[*]
```

```
Welcome to
```



```
version 1.0.0
```

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0\_60)  
Type in expressions to have them evaluated.  
Type :help for more information.  
Spark context available as sc.

```
scala> 
```

To open **Spark shell** shell with **YARN**, process the following command

## Starting Spark shell with Yarn

```
bin/spark-shell --master yarn
```



# Quick Start

A `SparkSession` instance is an entry to a `Spark` application

- If you type `spark` in the spark-shell interface then you get the following messages

```
res0: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@...
```

Message

You can use `SparkSession` instance `spark` to interact with `Spark` and to develop your data processing pipeline

For example,

```
val myRange = spark.range(1000).toDF("number")
myRange: org.apache.spark.sql.DataFrame = [number: bigint]
```

Creating Data Frame

```
myRange.show(2)
```

```
+-----+
|number|
+-----+
|      0|
|      1|
+-----+
```

Listing Data Frame

only showing top 2 rows

# Quick Start

Sample processing of a file **README.md**

Setting Spark Home folder

```
val YOUR_SPARK_HOME ="path-to-your-Spark-home"
```

Reading a text file

```
val textFile = spark.read.textFile("$YOUR_SPARK_HOME/README.md")  
textFile: org.apache.spark.sql.Dataset[String] = [value: string]
```

Counting rows

```
textFile.count()  
res0: Long = 104
```

Reading the first row

```
textFile.first()  
res1: String = # Apache Spark
```

Filtering and counting rows

```
textFile.filter(line => line.contains("Spark")).count()  
res2: Long = 20
```

# Quick Start

## More operations on a file

Counting number of words in the longest line

```
textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
res3: Int = 22
```

Filtering and counting rows

```
val wordCounts = textFile.flatMap(line => line.split(" ")).groupByKey(identity).count()
wordCounts: org.apache.spark.sql.Dataset[(String, Long)] = [value: string, count(1): bigint]
```

Listing results

```
wordCounts.show(2)
+-----+-----+
|   value|count(1)|
+-----+-----+
|  online|      1|
|  graphs|      1|
+-----+-----+
only showing top 2 rows
```

Listing results

```
wordCounts.collect()
res7: Array[(String, Long)] = Array((online,1), (graphs,1), (Parallel,1), (Building,1), (thread,1),
(documentation,3), (command,,2), (abbreviated,1), (overview,1), (rich,1), (set,2), ...)
```

# Spark Operations

## Outline

[The Programming Language Scala](#)

[Quick Start](#)

[Self Contained Application](#)

[Web User Interface](#)

[Operations on Resilient Distributed Datasets \(RDDs\)](#)

[Operations on Datasets](#)

[Operations on DataFrames](#)

[Spark SQL Module](#)

# Self-Contained Application

A sample self-contained application

SimpleApp.scala

```
import org.apache.spark.sql.SparkSession
object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "YOUR_SPARK_HOME/README.md"
    // Should be some file on your system
    val spark = SparkSession.builder
      .appName("Simple Application")
      .config("spark.master", "local[*]")
      .getOrCreate()
    val logData = spark.read.textFile(logFile).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println(s"Lines with a: $numAs, Lines with b: $numBs")
    spark.stop()
  }
}
```

# Self-Contained Application

Compiling `Scala` source code using `scalac`

```
scalac -classpath "$SPARK_HOME/jars/*" SimpleApp.scala
```

Compiling Scala source code

Creating a jar file in the following way

```
jar cvf app.jar SimpleApp*.class
```

Creating jar

Process it with Spark-shell in the following way

```
$SPARK_HOME/bin/spark-submit --master local[*] --class SimpleApp app.jar
```

Processing

# Spark Operations

## Outline

[The Programming Language Scala](#)

[Quick Start](#)

[Self Contained Application](#)

[Web User Interface](#)

[Operations on Resilient Distributed Datasets \(RDDs\)](#)

[Operations on Datasets](#)

[Operations on DataFrames](#)

[Spark SQL Module](#)

# Web UI

Each driver program has a Web UI, typically on port 4040

Spark Web UI displays information about running tasks, executors, and storage usage.

The screenshot shows the Spark Web UI interface. At the top, there's a navigation bar with tabs: Jobs (selected), Stages, Storage, Environment, Executors, and SQL. The Spark logo and version '2.1.0-SNAPSHOT' are on the left, and 'Spark shell application UI' is on the right. Below the navigation bar, the 'Spark Jobs (?)' section displays summary statistics: User: jacek, Total Uptime: 35 s, Scheduling Mode: FIFO, Active Jobs: 1, Completed Jobs: 1, and Failed Jobs: 1. There's a link to 'Event Timeline'. The 'Active Jobs (1)' section contains a table with one job (Job Id 2) that is currently running (0/1 stages succeeded). The 'Completed Jobs (1)' section contains a table with one job (Job Id 0) that has finished (1/1 stages succeeded). The 'Failed Jobs (1)' section contains a table with one job (Job Id 1) that has failed (0/1 stages succeeded, 1 failed).

**Spark Jobs (?)**

User: jacek  
Total Uptime: 35 s  
Scheduling Mode: FIFO  
Active Jobs: 1  
Completed Jobs: 1  
Failed Jobs: 1  
[Event Timeline](#)

**Active Jobs (1)**

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	<a href="#">show at &lt;console&gt;:24</a>	2016/09/29 14:01:20	5 s	0/1	0/1

**Completed Jobs (1)**

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	<a href="#">show at &lt;console&gt;:24</a>	2016/09/29 14:01:07	0.3 s	1/1	1/1

**Failed Jobs (1)**

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	<a href="#">show at &lt;console&gt;:24</a>	2016/09/29 14:01:14	87 ms	0/1 (1 failed)	0/1 (1 failed)



# Spark Operations

## Outline

[The Programming Language Scala](#)

[Quick Start](#)

[Self Contained Application](#)

[Web User Interface](#)

[Operations on Resilient Distributed Datasets \(RDDs\)](#)

[Operations on Datasets](#)

[Operations on DataFrames](#)

[Spark SQL Module](#)

# Operations on Resilient Distributed Datasets (RDDs)

Operations on **RDDs** are performed on raw **Java** or **Scala** objects

Creating a simple **RDD** with words and distributing over 2 partitions

```
val myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple".split(" ")  
val words = spark.sparkContext.parallelize(myCollection, 2)
```

Creating RDD

Eliminating duplicates and counting words

```
words.distinct().count()
```

Distinct and counting

Filtering

```
def startsWithS(individual:String) = { individual.startsWith("S") }
```

Filtering function

```
val onlyS = words.filter(word => startsWithS(word))
```

Filtering

```
onlyS.collect()
```

Results of filtering

# Operations on Resilient Distributed Datasets (RDDs)

Sorting of **RDD** uses **sortBy** method and a function that extracts a value from the objects

```
words.sortBy(word => word.length() * -1).take(2))
```

Sorting

Random split into Array

```
val fiftyFiftySplit = words.randomSplit(Array[Double](0.5, 0.5))
```

Split into Array

Reduce **RDD** to one value

```
def wordLengthReducer(leftWord:String, rightWord:String): String = {  
  if (leftWord.length >= rightWord.length)  
    return leftWord  
  else  
    return rightWord }  
  
words.reduce(wordLengthReducer)
```

Reducing

Reducing

# Operations on Resilient Distributed Datasets (RDDs)

Some operations on **RDDs** are available on key-value pairs

The most common ones are distributed "shuffle" operations, such as grouping or aggregating the elements by a key

For example, **reduceByKey** operation on key-value pairs can be used to count how many times each line of text occurs in a file

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

Sorting

## Some of the transformations of **RDDs**

map(func):	passes each element of RDD through a <b>function</b>	Transformations
filter(func):	selects all element <b>for</b> which a <b>function</b> returns <b>true</b>	
sample(withReplacement, fraction, seed):	extracts sample <b>from</b> RDD	
union(otherDataset):	unions two <b>RDDs</b>	
intersection(otherDataset):	finds intersection of two <b>RDDs</b>	
distinct([numPartitions]):	eliminates duplicates	
groupByKey([numPartitions]):	when called on RDD with (K, V) pairs, returns RDD with (K, <b>Iterable</b> ) pairs	
sortByKey([ascending], [numPartitions]):	when called on (K, V) pairs where K implements <b>Ordered</b> , returns (K, V) pairs sorted <b>by</b> keys	

# Spark Operations

## Outline

[The Programming Language Scala](#)

[Quick Start](#)

[Self Contained Application](#)

[Web User Interface](#)

[Operations on Resilient Distributed Datasets \(RDDs\)](#)

[Operations on Datasets](#)

[Operations on DataFrames](#)

[Spark SQL Module](#)

# Operations on Datasets

Operations on a **Dataset** start from creation of **case class**

```
case class Person(name: String, age: Long)
defined class Person
```

Creating case class

```
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS: org.apache.spark.sql.Dataset[Person] = [name: string, age: bigint]
```

Creating Dataset

```
caseClassDS.show()
+----+----+
|name|age|
+----+----+
|Andy| 32|
+----+----+
```

Listing Dataset

**Dataset** supports all operations of **DataFrame**

```
caseClassDS.select($"name").show()
+----+
|name|
+----+
|Andy|
+----+
```

Using a Dataset

# Operations on Datasets

Operations on **Datasets** start from creation of **case class**

```
case class Flight(DEST_COUNTRY_NAME: String, ORIGIN_COUNTRY_NAME: String, count: BigInt)
```

Creating case class

Next we create a **DataFrame**

```
val flightsDF = spark.read.parquet("/mnt/defg/chapter-1-data/parquet/2010-summary.parquet/")
```

Creating DataFrame

Finally, **DataFrame** is casted to **Dataset**

```
val flights = flightsDF.as[Flight]
```

Creating Dataset

Filtering a **Dataset**

```
def originIsDestination(flight_row: Flight): Boolean = {  
  return flight_row.ORIGIN_COUNTRY_NAME == flight_row.DEST_COUNTRY_NAME}
```

Defining a function

```
flights.filter(flight_row => originIsDestination(flight_row)).first()
```

Filtering

Mapping a **Dataset**

```
val destinations = flights.map(f => f.DEST_COUNTRY_NAME)
```

Mapping

# Spark Operations

## Outline

[The Programming Language Scala](#)

[Quick Start](#)

[Self Contained Application](#)

[Web User Interface](#)

[Operations on Resilient Distributed Datasets \(RDDs\)](#)

[Operations on Datasets](#)

[Operations on DataFrames](#)

[Spark SQL Module](#)



# Operations on DataFrames

**Dataset** and **DataFrame** are the data abstractions for **Spark SQL**

**Dataset** is a distributed collection of data

- It supports the use of self-defined functions to process data
- For example, **map** and **reduce** functions in the previous slides
- **Dataset** is typed; typing is checked at compiling time

**DataFrame** is a **Dataset** organized into named columns.

- It is conceptually equivalent to a table in a relational database or a data frame in **R/Python**
- To use self-defined functions, you need to register them with **Spark**
- **DataFrame** is untyped, i.e., typing is checked at runtime
- **DataFrame** is more performance-optimal than **Dataset**

# Operations on DataFrames

**DataFrame** can be created in the following way

```
val df = spark.read.json("people.json")
df.show()
```

Creating a DataFrame

```
+-----+-----+
|  age|  name |
+-----+-----+
| null|Michael|
|  30 |  Andy |
|  19 | Justin|
+-----+-----+
```

Results

```
df.printSchema()
```

Results

```
root
|-- age: long (nullable = true)
|-- name: string (nullable = true)
```

Results

# Operations on DataFrames

## Select on a DataFrame

Selecting from a DataFrame

```
df.select($"name", $"age" + 1).show()
```

Results

name	(age + 1)
Michael	null
Andy	31
Justin	20

Filtering a DataFrame

```
df.filter($"age" > 21).show()
```

Results

age	name
30	Andy

# Operations on DataFrames

Count people by age

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

Counting in a DataFrame

Results

age	count
19	1
null	1
30	1

# Operations on DataFrames

Register a **DataFrame** as **SQL** temporary view

Registering and selecting from DataFrame

```
df.createOrReplaceTempView("people")
val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
```

Results

age	name
null	Michael
30	Andy
19	Justin

# Operations on DataFrames

When to use **DataFrames** ?

Except for the following few cases, you can use them interchangeable (if performance is not a concern). You also can convert one to the other easily.

- In the Bigdata pipeline, you read an unstructured data source, for example, a text file as a **Dataset** and continue processing the data
- You can directly read an structured source like Hive table, JSON document as a **DataFrame**
- If you expect to use self-defined function easily, especially in the data cleaning or preprocessing stage of the pipeline, you should use a Dataset

# Operations on DataFrames

Create a **Dataset** of **Person** objects from a text file and convert it to a **DataFrame**

Converting a Dataset to DataFrame

```
val peopleDF = spark.sparkContext
  .textFile("examples/src/main/resources/people.txt")
  .map(_._split(", "))
  .map(attributes => Person(attributes(0), attributes(1).trim.toInt))
  .toDF()
```

Results

```
peopleDF: org.apache.spark.sql.DataFrame = [name: string, age: bigint]
```

# Operations on DataFrames

## Convert DataFrame to Dataset

Converting a Dataset to DataFrame

```
case class Employee(name: String, salary: Long)
val ds =
  spark.read.json("../examples/src/main/resources/employees.json").as[Employee]
```

Results

```
ds: org.apache.spark.sql.Dataset[Employee] = [name: string, salary: bigint]
```



# Operations on DataFrames

Spark DataFrame/Dataset support two types of operations: transformations and actions

Transformations are operations on DataFrames/Datasets that return a new DataFrame/Dataset

- For example `select()`, `groupBy()`, `map()`, and `filter()`

Actions are operations that return a result to the driver program or write it to storage, and kick off a computation

- For example `show()`, `count()`, and `first()`

Return type difference: transformations return DataFrames/Datasets, whereas actions return some other data type

Spark treats the two operations very differently

# Operations on DataFrames

**Transformations** are **lazily evaluated**, meaning that **Spark** will not begin to execute until it sees an action

Instead, **Spark** internally records metadata to indicate that some transformation operation has been requested

For example **transformation** creates another **DataFrame**

```
val sqlDF = spark.sql("SELECT * FROM people")
```

Creating a DataFrame

**Action** triggers the computation

```
sqlDF.show()
```

Action on a DataFrame

# Operations on DataFrames

The **lazy evaluation** to reduce the number of passes it has to take over the dataset

In **Hadoop MapReduce**, developers often have to consider how to group together operations to minimize the number of MapReduce passes

In Spark, there is no substantial benefit to writing a single complex map instead of chaining together many simple operations

Thus, users are free to organize their program into smaller, more manageable operations

# Spark Operations

## Outline

[The Programming Language Scala](#)

[Quick Start](#)

[Self Contained Application](#)

[Web User Interface](#)

[Operations on Resilient Distributed Datasets \(RDDs\)](#)

[Operations on Datasets](#)

[Operations on DataFrames](#)

[Spark SQL](#)

# Spark SQL

Spark SQL is a Spark module for general data processing and analytics

It can be used for all sorts of data, from unstructured log files to semi-structured CSV files and highly structured Parquet files

To interact with Spark SQL, you can either use SQL or Spark Structured API, or both

The same execution engine is used, independent of which API/language you use to express the computation

The APIs of Spark SQL provide a rich set of pre-built, high-level operations for accomplishing sophisticated data processing and ETL jobs, and mechanism to implement your own operations, for example self-defined functions and aggregations

# Spark SQL

Spark SQL has two data abstractions

- **DataFrame**
- **Dataset** (available in Scala/Java APIs, but not Python/R APIs)
- **DataFrame** can be represented as SQL tables and views

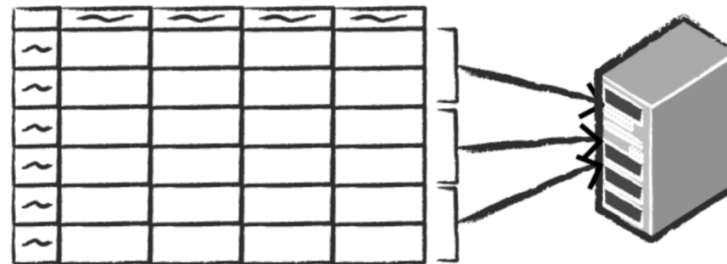
Both are distributed table-like collections with well-defined rows and columns.

- **DataFrame** vs. **spreadsheet**

Spreadsheet on  
a single machine



Table or Data Frame  
partitioned across servers  
in a data center



# Spark SQL

Spark SQL allows to code SQL statements in Scala, Java and Python language APIs.

To use SQL to manipulate a DataFrame, we first need to create a temporal view for it

```
df.createOrReplaceTempView("dfTable")
```

Creating a temporal view

All standard SQL statements + functions are applicable in Spark SQL  
Spark implements a subset of [ANSI SQL:2003](#)

# Spark SQL

## Using SQL

```
spark.sql(  
  "SELECT DEST_COUNTRY_NAME, sum(count)  
    FROM dfTable  
    GROUP BY DEST_COUNTRY_NAME"  
)  
.where("DEST_COUNTRY_NAME like 'S%'")  
.where("'sum(count)' > 10")  
.show(2)
```

Applying sql method

DEST_COUNTRY_NAME	sum(count)
Senegal	40
Sweden	118

Results



# References

## [The Scala Programming Language](#)

A Gentle Introduction to Spark, Databricks, (Available in **READINGS** folder)

## [RDD Programming Guide](#)

## [Spark SQL, DataFrames and Datasets Guide](#)

Karau H., Fast data processing with Spark Packt Publishing, 2013  
(Available from UOW Library)

Srinivasa, K.G., Guide to High Performance Distributed Computing: Case Studies with Hadoop, Scalding and SparkSpringer, 2015 (Available from UOW Library)

Chambers B., Zaharia M., Spark: The Definitive Guide, O'Reilly 2017

Perrin J-G., Spark in Action, 2nd ed., Manning Publications Co. 2020