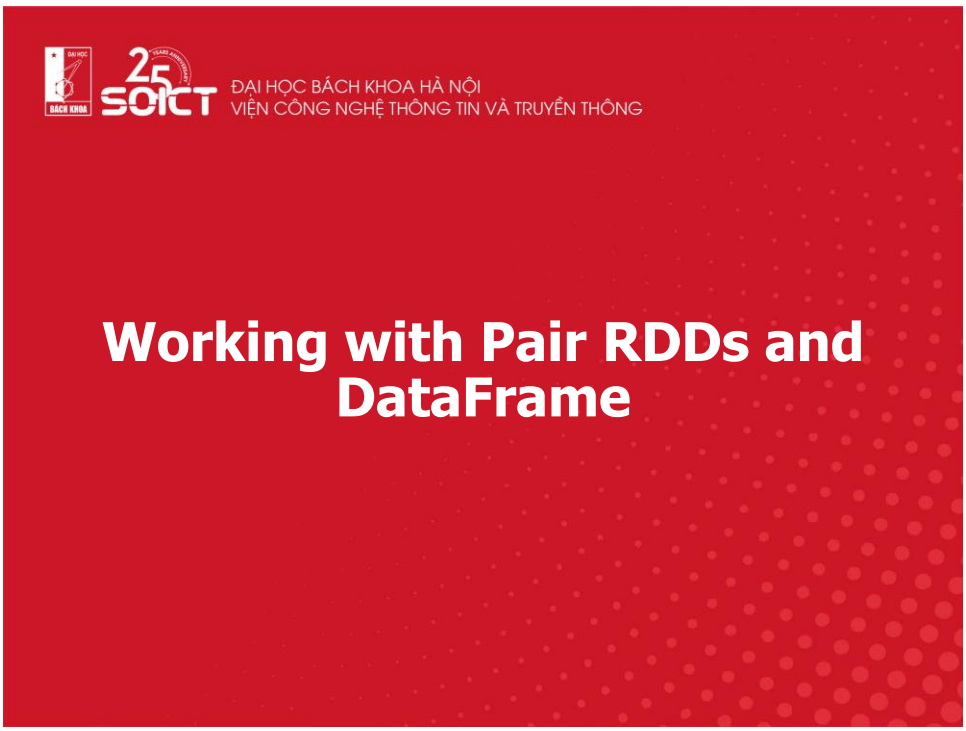

ĐẠI HỌC
BÁCH KHOA
25 YEARS ANNIVERSARY
SOICT
ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

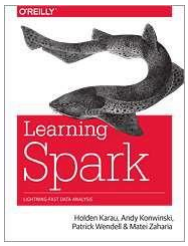
1


ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
Working with Pair RDDs and DataFrame

2

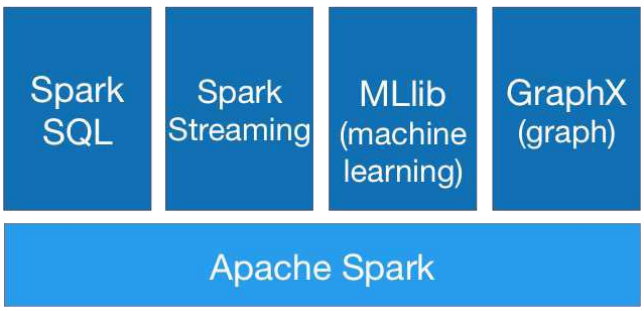
From where to learn Spark ?

http://spark.apache.org/



http://shop.oreilly.com/product/0636920028512.do

Spark architecture



Easy ways to run Spark ?

- ★ your IDE (ex. Eclipse or IDEA)
- ★ Standalone Deploy Mode: simplest way to deploy Spark on a single machine
- ★ Docker & Zeppelin
- ★ EMR
- ★ Hadoop vendors (Cloudera, Hortonworks)

Supported languages

RDD (Resilient Distributed Dataset)

RDD (Resilient Distributed Dataset)

- Resilient: If data in memory is lost, it can be recreated
- Distributed: Processed across the cluster
- Dataset: Initial data can come from a source such as a file, or it can be created programmatically

- **RDDs are the fundamental unit of data in Spark**
- **Most Spark programming consists of performing operations on RDDs**

Creating RDD (I)

- Python
- lines = sc.parallelize([“workshop”, “spark”])

- Scala
- val lines = sc.parallelize(List(“workshop”, “spark”))

- Java
- JavaRDD<String> lines = sc.parallelize(Arrays.asList(“workshop”, “spark”))

Creating RDD (II)

```
Python
lines = sc.textFile("/path/to/file.txt")

Scala
val lines = sc.textFile("/path/to/file.txt")

Java
JavaRDD<String> lines = sc.textFile("/path/to/file.txt")
```


RDD persistence

```
MEMORY_ONLY
MEMORY_AND_DISK
MEMORY_ONLY_SER
MEMORY_AND_DISK_SER
DISK_ONLY
MEMORY_ONLY_2
MEMORY_AND_DISK_2
OFF_HEAP
```

Working with RDDs

RDDs

RDDs can hold any serializable type of element

- –Primitive types such as integers, characters, and booleans
- –Sequence types such as strings, lists, arrays, tuples, and dicts (including nested data types)
- –Scala/Java Objects (if serializable)
- –Mixed types

§ **Some RDDs are specialized and have additional functionality**

- –Pair RDDs
- –RDDs consisting of key-value pairs
- –Double RDDs
- –RDDs consisting of numeric data

Creating RDDs from Collections

You can create RDDs from collections instead of files
–sc.parallelize(*collection*)

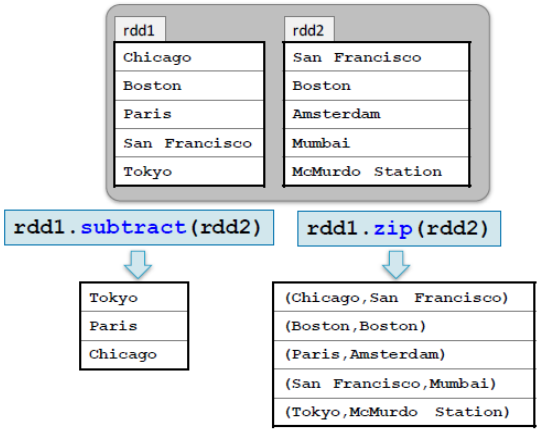
```
myData = ["Alice","Carlos","Frank","Barbara"]
> myRdd = sc.parallelize(myData)
> myRdd.take(2) ['Alice', 'Carlos']
```


Creating RDDs from Text Files (1)

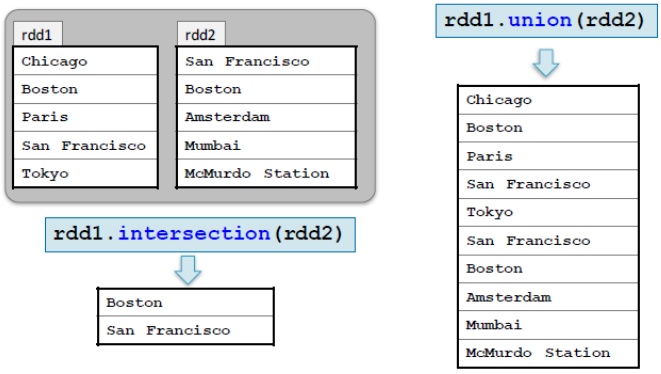
For file-based RDDs, use SparkContext.textFile
– Accepts a single file, a directory of files, a wildcard list of files, or a comma-separated list of files. Examples:
–sc.textFile("myfile.txt")
–sc.textFile("mydata/")
–sc.textFile("mydata/*.log")
–sc.textFile("myfile1.txt,myfile2.txt")
–Each line in each file is a separate record in the RDD

Files are referenced by absolute or relative URI
–Absolute URI:
–file:/home/training/myfile.txt
–hdfs://nnhost/loudacre/myfile.txt

Examples: Multi-RDD Transformations (1)



Examples: Multi-RDD Transformations (2)



Some Other General RDD Operations

Other RDD operations

- –first returns the first element of the RDD
- –foreach applies a function to each element in an RDD
- –top(*n*) returns the largest *n* elements using natural ordering

Sampling operations

- –sample creates a new RDD with a sampling of elements
- –takeSample returns an array of sampled elements

Other data structures in Spark

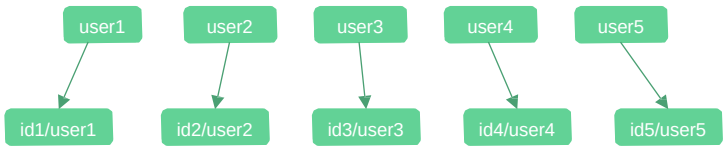
★ Paired RDD

★ DataFrame

★ DataSet

Paired RDD

Paired RDD = an RDD of key/value pairs



Pair RDDs

Pair RDDs

§ **Pair RDDs are a special form of RDD**
- –Each element must be a key-value pair (a two-element *tuple*)
- –Keys and values can be any type

§ **Why?**
- –Use with map-reduce algorithms
- –Many additional functions are available for common data processing needs
- –Such as sorting, joining, grouping, and counting

Pair RDD

(key1,value1)
(key2,value2)
(key3,value3)
...

Creating Pair RDDs

The first step in most workflows is to get the data into key/value form

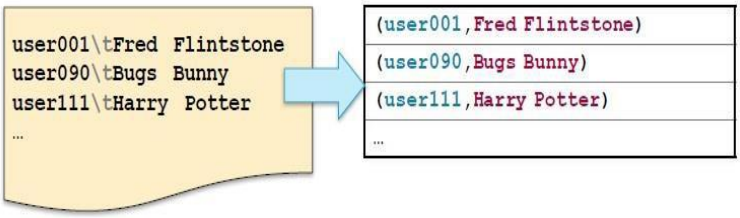
- –What should the RDD should be keyed on?
- –What is the value?

Commonly used functions to create pair RDDs
- **–map**
- **–flatMap / flatMapValues**
- **–keyBy**

Example: A Simple Pair RDD

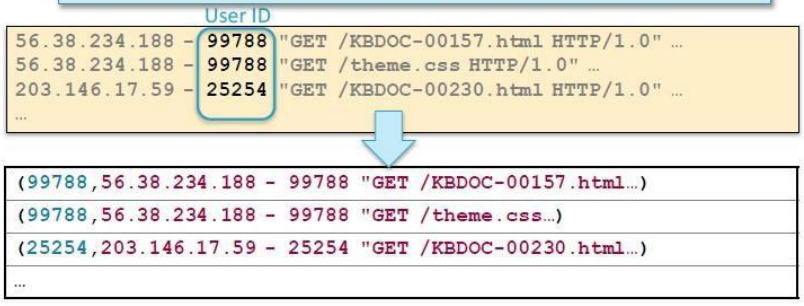
Example: Create a pair RDD from a tab-separated file

```
> val users = sc.textFile(file).
   map(line => line.split('\t').
   map(fields => (fields(0),fields(1)))
```



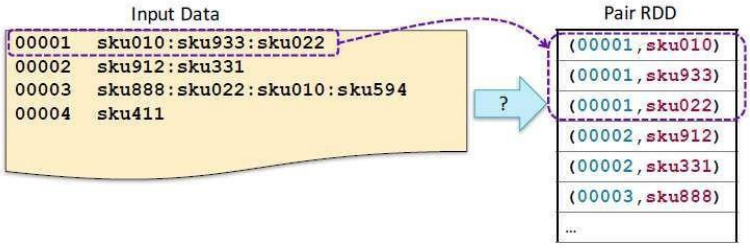
Example: Keying Web Logs by User ID

```
> sc.textFile(logfile).
    keyBy(line => line.split(' ')(2))
```



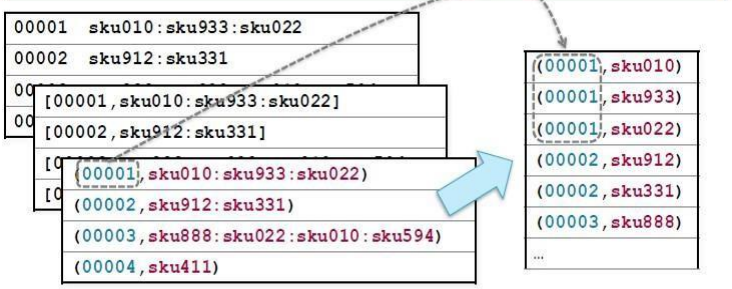
Mapping Single Rows to Multiple Pairs

- **How would you do this?**
  - Input: order numbers with a list of SKUs in the order
  - Output: order (key) and sku (value)



Answer : Mapping Single Rows to Multiple Pairs

```
> sc.textFile(file) \
  .map(lambda line: line.split('\t')) \
  .map(lambda fields: (fields[0],fields[1]))
  .flatMapValues(lambda skus: skus.split(':'))
```



Map-Reduce

§ **Map-reduce is a common programming model**
- –Easily applicable to distributed processing of large data sets

§ **Hadoop MapReduce is the major implementation**
- –Somewhat limited
- –Each job has one map phase, one reduce phase
- –Job output is saved to files

§ **Spark implements map-reduce with much greater flexibility**
- –Map and reduce functions can be interspersed
- –Results can be stored in memory
- –Operations can easily be chained

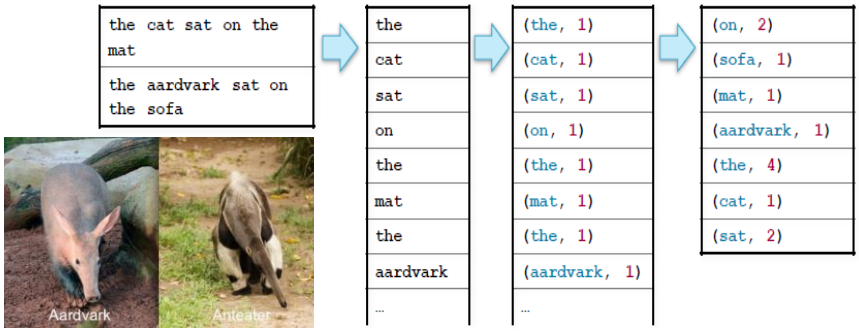
Map-Reduce in Spark

§ **Map-reduce in Spark works on pair RDDs**

§ **Map phase**
- –Operates on one record at a time
- –"Maps" each record to zero or more new records
- –Examples: **map, flatMap, filter, keyBy**

§ **Reduce phase**
- –Works on map output
- –Consolidates multiple records
- –Examples: **reduceByKey, sortByKey, mean**

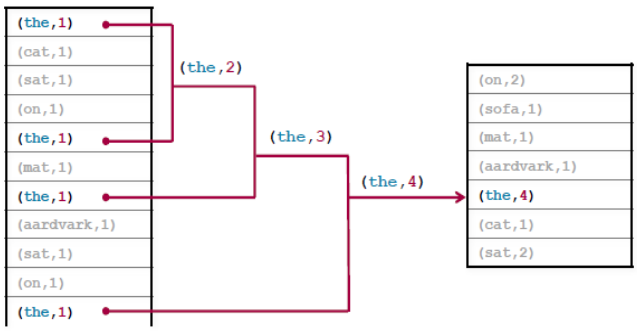
Example: Word Count



reduceByKey

The function passed to reduceByKey combines values from two keys

– Function must be binary



```
> val counts = sc.textFile(file). flat.Map
  (line => line.split(' ')). map(word => (word
  ,1)). reduceByKey((v1,v2) => v1+v2)
```

OR

```
> val counts = sc.textFile(file). flat.Map
  (_.split(' ')).
  map((_,1)).
  reduceByKey(_+_)
```

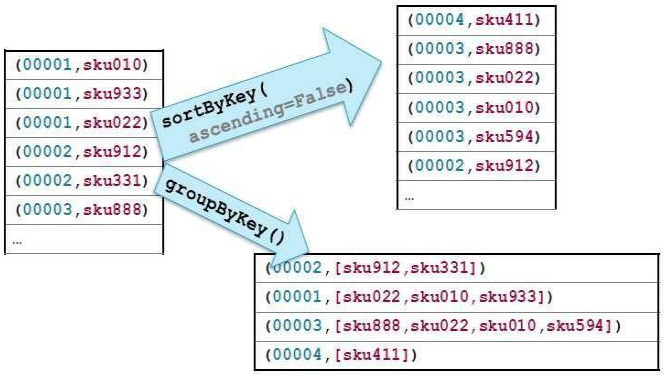
Pair RDD Operations

§ **In addition to map and reduceByKey operations, Spark has several operations specific to pair RDDs**

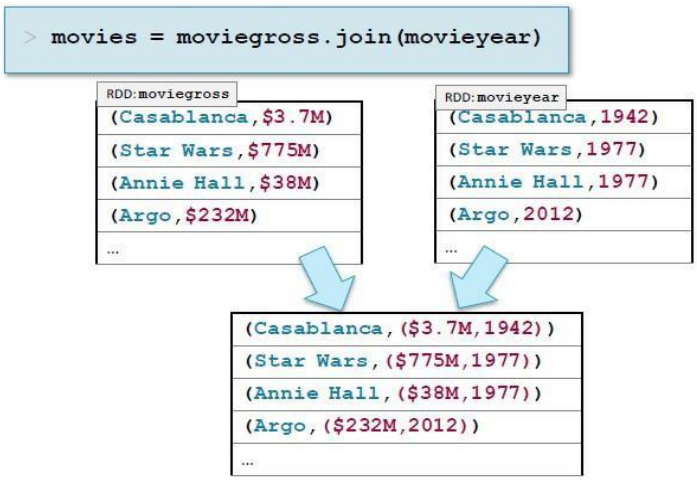
§ **Examples**

- **countByKey** returns a map with the count of occurrences of each key
- **groupByKey** groups all the values for each key in an RDD
- **sortByKey** sorts in ascending or descending order
- **join** returns an RDD containing all pairs with matching keys from two RDD

Example: Pair RDD Operations



Example: Joining by Key



Other Pair Operations

§ **Some other pair operations**

–**keys** returns an RDD of just the keys, without the values

–**values** returns an RDD of just the values, without keys

–**lookup(*key*)** returns the value(s) for a key

–**leftOuterJoin**, **rightOuterJoin** , **fullOuterJoin** join two RDDs, including keys defined in the left, right or either RDD respectively

–**mapValues**, **flatMapValues** execute a function on just the values, keeping the key the same

DataFrames and Apache Spark SQL

What is Spark SQL?

§ **What is Spark SQL?**
- –Spark module for structured data processing
- –Replaces Shark (a prior Spark module, now deprecated)
- –Built on top of core Spark

§ **What does Spark SQL provide?**
- –The DataFrame API—a library for working with data as tables
- –Defines DataFrames containing rows and columns
- –DataFrames are the focus of this chapter!
- –Catalyst Optimizer—an extensible optimization framework
- –A SQL engine and command line interface

SQL Context

§ **The main Spark SQL entry point is a SQL context object**
- –Requires a **SparkContext** object
- –The SQL context in Spark SQL is similar to Spark context in core Spark

§ **There are two implementations**
- –**SQLContext**
- –Basic implementation
- –**HiveContext**
- –Reads and writes Hive/HCatalog tables directly
- –Supports full HiveQL language
- –Requires the Spark application be linked with Hive libraries
- –Cloudera recommends using **HiveContext**

Creating a SQL Context

§ **The Spark shell creates a HiveContext instance automatically**
- –Call **sqlContext**
- –You will need to create one when writing a Spark application
- –Having multiple SQL context objects *is* allowed

§ **A SQL context object is created based on the Spark context**

Language: Scala

```
import org.apache.spark.sql.hive.HiveContext
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
```

40

DataFrames

§ **DataFrames are the main abstraction in Spark SQL**

- –Analogous to RDDs in core Spark
- –A distributed collection of structured data organized into Named columns
- –Built on a *base RDD* containing **Row** objects

41

Creating a DataFrame from a Data Source

§ **sqlContext.read returns a DataFrameReader object**

§ **DataFrameReader provides the functionality to load data into a DataFrame**

§ **Convenience functions**
- **–json(*filename*)**
- **–parquet(*filename*)**
- **–orc(*filename*)**
- **–table(*hive-tablename*)**
- **–jdbc(*url,table,options*)**

Example: Creating a DataFrame from a JSON File

Language: Scala

```
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
val peopleDF = sqlContext.read.json("people.json")
```

File: people.json

```
{"name":"Alice", "pcode":"94304"}
{"name":"Brayden", "age":30, "pcode":"94304"}
{"name":"Carla", "age":19, "pcode":"10036"}
{"name":"Diana", "age":46}
{"name":"Étienne", "pcode":"94104"}
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

Example: Creating a DataFrame from a Hive/Impala Table

Language: Scala

```
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
val customerDF = sqlContext.read.table("customers")
```

Table: customers

cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
…	…	…

cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
…	…	…

Loading from a Data Source Manually

§ **You can specify settings for the DataFrameReader**
–**format**: Specify a data source type
–**option**: A key/value setting for the underlying data source
–**schema**: Specify a schema instead of inferring from the data source
§ **Then call the generic base function load**

```
sqlContext.read.
   format("com.databricks.spark.avro").
   load("/loudacre/accounts_avro")
```

```
sqlContext.read.
   format("jdbc").
   option("url","jdbc:mysql://localhost/loudacre").
   option("dbtable","accounts").
   option("user","training").
   option("password","training").
   load()
```

Data Sources

§ **Spark SQL 1.6 built-in data source types**
- **–table**
- **–json**
- **–parquet**
- **–jdbc**
- **–orc**

§ **You can also use third party data source libraries, such as**
- –Avro (included in CDH)
- –HBase
- –CSV
- –MySQL
- –and more being added all the time

DataFrame Basic Operations

- § Basic operations deal with DataFrame metadata (rather than its data)
- § Some examples
  - –schema returns a schema object describing the data
  - –printSchema displays the schema as a visual tree
  - –cache / persist persists the DataFrame to disk or memory
  - –columns returns an array containing the names of the columns
  - –dtypes returns an array of (column name,type) pairs
  - –explain prints debug information about the DataFrame to the console

DataFrame Basic Operations

Language: Scala

```
> val peopleDF = sqlContext.read.json("people.json")
> peopleDF.dtypes.foreach(println)
(age,LongType)
(name,StringType)
(pcode,StringType)
```


DataFrame Actions

- Some DataFrame actions
  - –collect returns all rows as an array of Row objects
  - –take(*n*) returns the first n rows as an array of Row objects
  - –count returns the number of rows
  - –show(*n*)displays the first n rows (default=20)

Language: Scala

```
> peopleDF.count()
res7: Long = 5

> peopleDF.show(3)
age  name    pcode
null Alice   94304
30   Brayden 94304
19   Carla   10036
```

DataFrame Queries

§ **DataFrame query methods return new DataFrames**
– Queries can be chained like transformations
§ **Some query methods**
–**distinct** returns a new DataFrame with distinct elements of this DF
–**join** joins this DataFrame with a second DataFrame
– Variants for inside, outside, left, and right joins
–**limit** returns a new DataFrame with the first **n** rows of this DF
–**select** returns a new DataFrame with data from one or more columns of the base DataFrame
–**where** returns a new DataFrame with rows meeting specified query criteria (alias for **filter**)

DataFrame Query Strings

- **Some query operations take strings containing simple query expressions**
  - Such as `select` and `where`
- **Example:** `select`

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

`peopleDF.select("age")`

age
null
30
19
46
null

`peopleDF.select("name","age")`

name	age
Alice	null
Brayden	30
Carla	19
Diana	46
Étienne	null

Querying DataFrames using Columns

§ **Columns can be referenced in multiple ways**

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

▪ Scala

```
val ageDF = peopleDF.select(peopleDF("age"))
```

```
val ageDF = peopleDF.select($"age")
```

age
null
30
19
46
null

Joining DataFrames

§ **A basic inner join when join column is in both DataFrames**

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

pcode	city	state
10036	New York	NY
87501	Santa Fe	NM
94304	Palo Alto	CA
94104	San Francisco	CA

Language: Python/Scala

```
peopleDF.join(pcodesDF, "pcode")
```

pcode	age	name	city	state
94304	null	Alice	Palo Alto	CA
94304	30	Brayden	Palo Alto	CA
10036	19	Carla	New York	NY
94104	null	Étienne	San Francisco	CA

Joining DataFrames

- Specify type of join as `inner` (default), `outer`, `left_outer`, `right_outer`, or `leftsemi`

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

pcode	city	state
10036	New York	NY
87501	Santa Fe	NM
94304	Palo Alto	CA
94104	San Francisco	CA

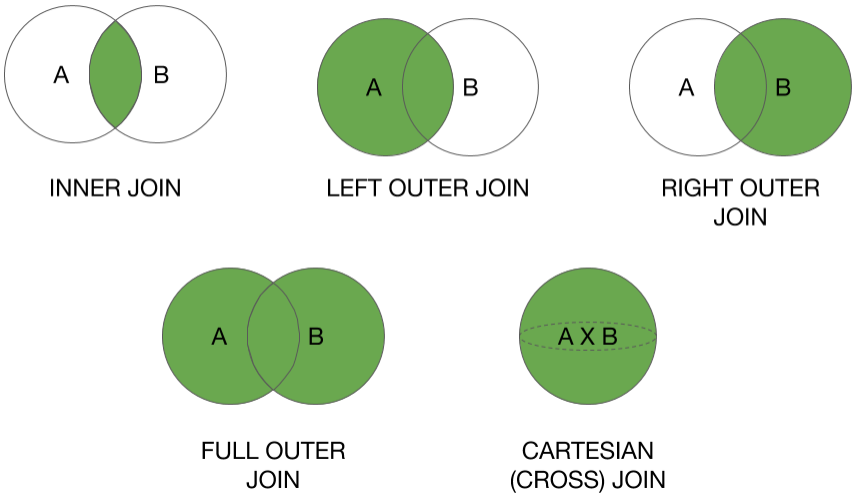
Language: Python

```
peopleDF.join(pcodesDF, "pcode",
"left_outer")
```

Language: Scala

```
peopleDF.join(pcodesDF,
Array("pcode"), "left_outer")
```

pcode	age	name	city	state
94304	null	Alice	Palo Alto	CA
94304	30	Brayden	Palo Alto	CA
10036	19	Carla	New York	NY
null	46	Diana	null	null
94104	null	Étienne	San Francisco	CA



SQL Queries

§ **When using HiveContext, you can query Hive/Impala tables using HiveQL**

– Returns a DataFrame

Language: Python/Scala

```
sqlContext.
  sql("""SELECT * FROM customers WHERE name LIKE "A%" """)
```

Table: customers

cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
…	…	…

cust_id	name	country
001	Ani	us

Saving DataFrames

§ **Data in DataFrames can be saved to a data source**

§ **Use DataFrame.write to create a DataFrameWriter**

§ **DataFrameWriter provides convenience functions to externally save the data represented by a DataFrame**

–**jdbc** inserts into a new or existing table in a database

–**json** saves as a JSON file

–**parquet** saves as a Parquet file

–**orc** saves as an ORC file

–**text** saves as a text file (string data in a single column only)

–**saveAsTable** saves as a Hive/Impala table (**HiveContext** only)

Language: Python/Scala

```
peopleDF.write.saveAsTable("people")
```

Options for Saving DataFrames

- § DataFrameWriter option methods
  - –format specifies a data source type
  - –mode determines the behavior if file or table already exists:
  - overwrite, append, ignore or error (default is error)
  - –partitionBy stores data in partitioned directories in the form
  - *column=value* (as with Hive/Impala partitioning)
  - –options specifies properties for the target data source
  - –save is the generic base function to write the data

Language: Python/Scala

```
peopleDF.write.
  format("parquet").
  mode("append").
  partitionBy("age").
  saveAsTable("people")
```


DataFrames and RDDs

§ **DataFrames are built on RDDs**

–Base RDDs contain **Row** objects

–Use **rdd** to get the underlying RDD

```
peopleRDD = peopleDF.rdd
```

peopleDF

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

peopleRDD

Row[null,Alice,94304]
Row[30,Brayden,94304]
Row[19,Carla,10036]
Row[46,Diana,null]
Row[null,Étienne,94104]

DataFrames and RDDs

§ **Row RDDs have all the standard Spark actions and transformations**
–Actions: **collect**, **take**, **count**, and so on
–Transformations: **map**, **flatMap**, **filter**, and so on

§ **Row RDDs can be transformed into pair RDDs to use map-reduce methods**

§ **DataFrames also provide convenience methods (such as map, flatMap,and foreach)for converting to RDDs**

Working with **Row** Objects

–Use **Array**-like syntax to return values with type **Any**
–**row(*n*)** returns element in the *n*th column
–**row.fieldIndex("age")**returns index of the **age** column
–Use methods to get correctly typed values
–**row.getAs[Long]("age")**
–Use type-specific **get** methods to return typed values
–**row.getString(*n*)** returns *n*th column as a string
–**row.getInt(*n*)** returns *n*th column as an integer
–And so on

Prerequisites

- Docker
- Zeppelin Docker Container
- Terminal Tools (Command Prompt, PowerShell)

Working with Spark RDDs, Pair-RDDs

RDD Operations

Transformations	Actions
map()	count()
flatMap()	collect()
filter()	first(), top(n)
union()	take(n), takeOrdered(n)
intersection()	countByValue()
distinct()	reduce()
groupByKey()	foreach()
reduceByKey()	…
sortByKey()	
join()	
…	

Lambda Expression

PySpark WordCount example:

```
input_file = sc.textFile("/path/to/text/file")
map = input_file.flatMap(lambda line: line.split(" ")) \
                        .map(lambda word: (word, 1))
counts = map.reduceByKey(lambda a, b: a + b)
counts.saveAsTextFi
```

lambda arguments: expression

PySpark RDD API

https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

map(*f, preservesPartitioning=False*) [source]

Return a new RDD by applying a function to each element of this RDD.

```
>>> rdd = sc.parallelize(["b", "a", "c"])
>>> sorted(rdd.map(lambda x: (x, 1)).collect())
[('a', 1), ('b', 1), ('c', 1)]
```

flatMap(*f, preservesPartitioning=False*) [source]

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

```
>>> rdd = sc.parallelize([2, 3, 4])
>>> sorted(rdd.flatMap(lambda x: range(1, x)).collect())
[1, 1, 1, 2, 2, 3]
>>> sorted(rdd.flatMap(lambda x: [(x, x), (x, x)]).collect())
[(2, 2), (2, 2), (3, 3), (3, 3), (4, 4), (4, 4)]
```


68

Practice with flight data (1)

Data: **airports.dat** (https://openflights.org/data.html)

[Airport ID, Name, City, Country, IATA, ICAO, Latitude, Longitude, Altitude, Timezone, DST, Tz database, Type, Source]

Try to do somethings:

- Create RDD from textfile
- Count the number of airports
- Filter by country
- Group by country
- Count the number of airports in each country

69

Practice with flight data (2)

- Data: **airports.dat** (https://openflights.org/data.html)

[Airport ID, Name, City, Country, IATA, ICAO, Latitude, Longitude, Altitude, Timezone, DST, Tz database, Type, Source]

- Data: **routes.dat**

[Airline, Airline ID, Source airport, Source airport ID, Destination airport, Destination airport ID, Codeshare, Stops, Equipment]

Try to do somethings:

- Join 2 RDD
- Count the number of flights arriving in each country

Working with DataFrame and Spark SQL

Creating a DataFrame(1)

```
%pyspark
from pyspark.sql import *

Employee = Row("firstName", "lastName", "email", "salary")

employee1 = Employee('Basher', 'armbrust', 'bash@edureka.co', 100000)
employee2 = Employee('Daniel', 'meng', 'daniel@stanford.edu', 120000 )
employee3 = Employee('Muriel', None, 'muriel@waterloo.edu', 140000 )
employee4 = Employee('Rachel', 'wendell', 'rach_3@edureka.co', 160000 )
employee5 = Employee('Zach', 'galifianakis', 'zach_g@edureka.co', 160000 )

employees = [employee1,employee2,employee3,employee4,employee5]

print(Employee[0])

print(employees)

dframe = spark.createDataFrame(employees)
dframe.show()
```


Creating a DataFrame

From CSV file:

```
%pyspark
flightData2015 = spark\
  .read\
  .option("inferSchema", "true")\
  .option("header", "true")\
  .csv("/usr/zeppelin/module9/2015-summary.csv")

flightData2015.show()
```

```
%pyspark
from pyspark.sql import *
list = [('Ankit',25),('Jalfaizy',22),('saurabh',20),('Bala',26)]
rdd = sc.parallelize(list)
people = rdd.map(lambda x: Row(name=x[0], age=int(x[1])))
df = spark.createDataFrame(people)

df.show()
```

DataFrame APIs

- **DataFrame**: show(), collect(), createOrReplaceTempView(), distinct(), filter(), select(), count(), groupBy(), join()…
- **Column**: like()
- **Row**: row.key, row[key]
- **GroupedData**: count(), max(), min(), sum(), …

https://spark.apache.org/docs/latest/api/python/pyspark.sql.html

Spark SQL

- Create a temporary view
- Query using SQL syntax

```
%pyspark
flightData2015.createOrReplaceTempView("flight_data_2015")

maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")

maxSql.show()
```

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

76

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions!

soict.hust.edu.vn/ fb.com/groups/soict

77


ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

78