Introduction

Please refer to the following pages for more detailed discussion on the use of Spark API and different coding styles in AVA:

For using Spark API on basic data management it turns out that there are 4 documentation pages that document the most important Spark API that you will ever need. Please bookmark them and use the methodology learned in "Using Spark API" above to navigate through the pages:

, the Dataset, a.k.a. DataFrame is the basic tabular representation of your data. , represents a Column in a Dataset, contains e.g. operators which produce new columns like newcol = col1 + col2, or newcol = col1.isEmpty.
, describes all functions which create new columns.
, represents the data structure after a df.groupBy on which then to define aggregations or pivoting.

Preliminary checking of the data frame

Before any specific data processing is done, it is always good to have the preliminary checking of the data frame that we are working on. Typical tasks are:

- To check the number of rows and columns of the data frame
- To get the column names of the data frame
- To understand the schema of the data frame
- To show the first few rows of the data frame

Let's first try to create the data frame:

Create data frame

```
%spark // Create data frame with sequence val df = Seq[(String,
Integer)](("a", 1), ("b", 2), (null, null), ("", 2)).toDF("A", "B")

// To show the data frame
df.show()
```

Output

```
df: org.apache.spark.sql.DataFrame = [A: string, B: int] +----+----
+
|   A|   B|
+----+----+
|   a|   1|
|   b|   2|
|null|null|
|    |   2|
+----+----+
```

From above we have created a data frame with two columns A and B. It has 4 rows and it contains null and empty ("") values as well.

In this section we are going to show the ways of checking the basic information of the data frame:

Actions	Description

show()	To show the first 20 lines of content of the data frame. Variants: show(n) to show n lines; show(false) to not restrict column width; show(n, false) for combination
count()	To show the number of rows in the data frame
printSchema()	To show the schema of the data frame. It shows the type of each of the column of the data frame

Please also note that all of the above are "actions". Please refer to the link for more details.

To show the number of rows and columns and the names of the columns of the data frame:

To show basic information of the data frame

```
%spark // Create data frame with sequence val df = Seq[(String,
Integer)](("a", 1), ("b", 2), (null, null), ("", 2)).toDF("A", "B") //
To show the number of rows in the data frame df.count() // To show the
names of the columns in the data frame df.columns // To show the
number of columns in the data frame df.columns.size
```

Output

```
res19: Long = 4
res20: Array[String] = Array(A, B)
res21: Int = 2
```

To show the schema of the data frame:

To show the schema of the data frame

```
%spark // Create data frame with sequence val df = Seq[(String,
Integer)](("a", 1), ("b", 2), (null, null), ("", 2)).toDF("A", "B") //
To print the schema of the data frame df.printSchema()
```

Output

```
root
 |-- A: string (nullable = true)
 |-- B: integer (nullable = true)
```

To show the first few rows of the data frame:

To show the first few rows of the data frame

```
%spark // Create data frame with sequence val df = Seq[(String,
Integer)](("a", 1), ("b", 2), (null, null), ("", 2)).toDF("A", "B") //
To show the first two lines of the data frame (n = 2) df.show(2)
```

Output

```
df: org.apache.spark.sql.DataFrame = [A: string, B: int] +---+---
+
|  A|  B|
+---+---+
|  a|  1|
|  b|  2| +---+---+ only
showing top 2 rows
```

Adding or removing column(s) in data frame

Please refer to the link for more details of the concept on "column" in data frame: .

Adding a column to the data frame

In order to add one additional column in the existing data frame, we could use the data frame's withColumn method.

In this example one additional column D will be added based on the values in column B and column C of the existing data frame. Here 'B is a column, the + is the operator on the column with 'B the left hand side of the operator and the 'C the right hand side of that operator. The result is another column. Then the enclosing () can be removed. On this resulting column the next operator / is called which requires as right hand side yet another column. In the source code there is just the number 2, which is a Scala Int, but not a column. But Spark defines various implicit conversions. Here the 2 is implicitly converted via the column function lit() to lit(2), which creates a column with constant literate value of 2. Finally, the / operator results in a new column, which is then called "D".

Adding column in dataframe

```
%spark val df = Seq[(String, Integer, Double)](("a", 1, 3.1), ("b",
2, 4.2), (null, null, 5.3), ("", 2, 6.4)).toDF("A", "B", "C")
df.show()
val df1 = df.withColumn("D", ('B + 'C) / 2)
df1.show()
```

Output

```
df: org.apache.spark.sql.DataFrame = [A: string, B: int ... 1 more
field] +----+----+---+
|   A|   B|  C|
+----+----+---+
|   a|   1|3.1|
|   b|   2|4.2|
|null|null|5.3|
|    |   2|6.4| +----+----+---
+

df1: org.apache.spark.sql.DataFrame = [A: string, B: int ... 2 more
fields] +----+----+---+----+
|   A|   B|  C|   D|
+----+----+---+----+
|   a|   1|3.1|2.05|
|   b|   2|4.2| 3.1|
|null|null|5.3|null|
|    |   2|6.4| 4.2|
+----+----+---+----+
```

Removing column(s) from the data frame

We could use the drop method to remove column(s) from the data frame:

To remove column in data frame

```
%spark // Create data frame with sequence val df = Seq[(String,
Integer, Double)](("a", 1, 3.1), ("b", 2, 4.2), (null, null, 5.3),
("", 2, 6.4)).toDF("A", "B", "C") df.show() // To drop column C val
df1 = df.drop('C) df1.show()
```

Output

```
df: org.apache.spark.sql.DataFrame = [A: string, B: int ... 1 more
field] +----+----+---+
|   A|   B|  C|
+----+----+---+
|   a|   1|3.1|
|   b|   2|4.2|
|null|null|5.3|
|    |   2|6.4| +----+----+---
+

df1: org.apache.spark.sql.DataFrame = [A: string, B: int] +----+----
+
|   A|   B|
+----+----+
|   a|   1|
|   b|   2|
|null|null|
|    |   2|
+----+----+
```

How to perform filtering in data frame

Selecting columns from the data frame

In the example below we will extract the columns A and B from the data frame df.

To select column(s) in data frame

```
%spark // Create data frame with sequence val df = Seq[(String,
Integer, Double)](("a", 1, 3.1), ("b", 2, 4.2), (null, null, 5.3),
("", 2, 6.4)).toDF("A", "B", "C") df.show() df.select('A, 'B).show()
```

Output

```
df: org.apache.spark.sql.DataFrame = [A: string, B: int ... 1 more
field] +----+----+---+
|   A|   B|  C|
+----+----+---+
|   a|   1|3.1|
|   b|   2|4.2|
|null|null|5.3|
|    |   2|6.4|
+----+----+---+

+----+----+
|   A|   B|
+----+----+
|   a|   1|
|   b|   2|
|null|null|
|    |   2|
+----+----+
```

Selecting rows from the data frame based on different criteria in the columns

In this example, we will filter data frame df based on the following condition:

The values in column B <= 3 AND the value in column A equals to "a"

To filter rows in data frame

```
%spark val df = Seq[(String, Integer)](("a", 1), ("b", 2), (null,
null),"", 2)).toDF("A", "B") df.filter('B <= 3 && 'A === "a").show()
```

Output

```
+---+---+
|  A|  B|
+---+---+
|  a|  1|
+---+---+
```

Note that "===" is one of the methods in column object. Please refer to the page for more details:

Note also that "==" would be comparing two Scala objects for equality, returning a Boolean value, while "===" compares two Spark Columns, returning a Spark Column of Boolean values, which contains per row the result of the comparisons of the elements of the two columns of this row. Hence we need to use "===" in this case.

If we want to check and return the result of the comparison that if the elements of the two columns are unequal then we need to use "=!=" method.

Note also the "&&" is one of the methods on a Column, combining two Boolean columns such that per row the two values are combined with the logical AND operator.

Please find below a table which shows other types of operators available in Spark Column API. Their functionality is equivalent to the corresponding Scala operator:

Spark Column or Scala Operator	Description	Example
&&	It is called logical AND operator. If both the operands are true then condition becomes true. Note also that there are implicit conversions of null to false, and numerics = 0 to false and numerics <> 0 to true.	A && B
||	It is called logical OR operator. If any of the operands are true then condition becomes true	A || B
! (documented as unary_!)	It is called logical NOT operator. Use to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make false	!(A && B)
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	A >= B
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	A <= B

Dealing with missing values

There are different ways to deal with the missing values in the data frame:

- Remove the rows or columns with missing values
- Remove the rows or columns if the number of missing values exceeding a predefined threshold
- Replace the missing values of the column with other aggregated values such as the median or mean value of that particular column

The steps that we would like to proceed are listed as follows:

-
- To identify the number of missing values per column

Determine the appropriate method of data cleaning and perform the corresponding data imputation technique

Checking the number of missing values per column of data frame

Please find below one of the ways to check the number of null values per column in the data frame:

To count the number of missing values per column

```
import org.apache.spark.sql.functions.{sum, col}
val df = Seq[(String, Integer)](("a", 1), ("b", 2), (null, null), ("",
2)).toDF("A", "B") df.select(df.columns.map(c =>
sum(col(c).isNull.cast("int").alias(c)): _*)
  .show
```

Output

```
import org.apache.spark.sql.functions.{sum, col} +---+---
+
|  A|  B|
+---+---+
|  1|  1|
+---+---+
```

Please note that the above calculation only counts the number of null value for each column in the data frame. The empty value "" is excluded in the calculation and should be remapped to null values.

Replacing empty value (or any other missing value defined in the data source) with null values

To replace values with null

```
%spark import org.apache.spark.sql.functions._ val df = Seq[(String,
Integer)](("a", 1), ("b", 2), (null, null), ("", 2)).toDF("A", "B")
df.withColumn("A", when('A === "", lit(null)).otherwise('A))
  .show()
```

Output

```
import org.apache.spark.sql.functions._ +----+----
+
|   A|   B|
+----+----+
|   a|   1|
|   b|   2|
|null|null|
|null|   2|
+----+----+
```

Droping rows with missing values

Once it is done the next step is to decide the best way to handle the missing (null) values in the data frame. Note that it is also one of the important steps on data preparation phase of any data mining task.

Let's assume here we would like to remove the rows with the missing values and in this case we could use na.drop() method:

To drop rows with NA

```
%spark import org.apache.spark.sql.functions._ val df = Seq[(String,
Integer)](("a", 1), ("b", 2), (null, null), ("", 2)).toDF("A", "B")
val df1 = df.withColumn("A", when('A === "", lit(null)).otherwise('A))
df1.show() val df2 = df1.na.drop() df2.show()
```

Output

```
import org.apache.spark.sql.functions._ df:
org.apache.spark.sql.DataFrame = [A: string, B: int] df1:
org.apache.spark.sql.DataFrame = [A: string, B: int] +---
-+----+
|   A|   B|
+----+----+
|   a|   1|
|   b|   2|
|null|null|
|null|   2| +----+----
+

df2: org.apache.spark.sql.DataFrame = [A: string, B: int] +---+---
+
|  A|  B|
+---+---+
|  a|  1|
|  b|  2|
+---+---+
```

In the example above, there were only one kind of undefined values beyond null, the empty string "". If there is only one kind of undefined values, it may be more efficient to not even load rows containing these undefined values into the data frame in the first place. For this see option "nullValue" at .

Note also that in this tutorial we have only covered one of the methods of dealing with the missing values: dropping the rows if there is missing value. We will cover all other imputation methods later on in other tutorials. Please refer to for more details.

Data aggregation

Data aggregation is one of the important steps in data processing.

In this example, the following data aggregation will be performed in the data frame df:

- Select the maximum value of the Age column grouped by the columns Department and Team. The name of the column will be renamed as Max_age.
- Select the sum of the Expense column grouped by the columns Department and Team. The name of the column will be renamed as Sum_Expense.

Data aggregation

```
%spark val df = Seq(("A", "A1", 30, 100), ("A", "A1", 40, 200), ("A",
"A3", 35, 300), ("B", "A4", 40, 400), ("C", "A5", 50, 500), ("C", "A6",
45, 600))            .toDF("Department", "Team", "Age", "Expense")
df.show() // To select the age of the oldest employee and the aggregate
expense for each department df.groupBy('Department, 'Team)
.agg(max('Age).as("Max_Age"), sum('Expense).as("Sum_Expense"))
  .show()
```

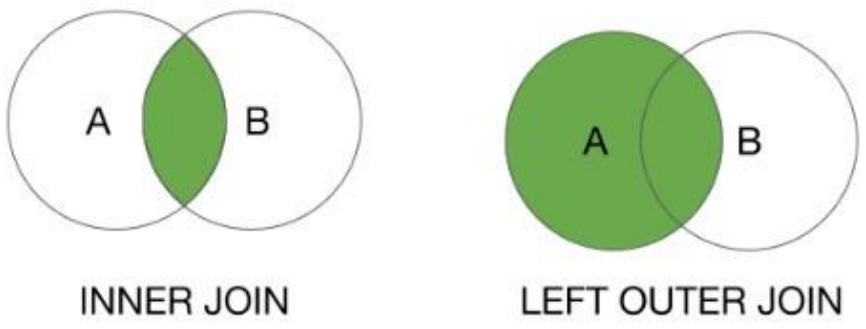
Output

```
df: org.apache.spark.sql.DataFrame = [Department: string, Team: string
... 2 more fields]
+----------+----+---+-------+
|Department|Team|Age|Expense|
+----------+----+---+-------+
|         A|  A1| 30|    100|
|         A|  A1| 40|    200|
|         A|  A3| 35|    300|
|         B|  A4| 40|    400|
|         C|  A5| 50|    500|
|         C|  A6| 45|    600|
+----------+----+---+-------+

+----------+----+-------+-----------+
|Department|Team|Max_Age|Sum_Expense|
+----------+----+-------+-----------+
|         C|  A5|     50|        500|
|         A|  A3|     35|        300|
|         C|  A6|     45|        600|
|         B|  A4|     40|        400|
|         A|  A1|     40|        300|
+----------+----+-------+-----------+
```

How to join two data frames

In this section we will focus on the two common types of joining methods: inner join and left join:



Let's assume we have two data frames A and B and now if we would like to perform the inner join based on a set of common columns in between we will extract the portion of the data frames which are marked in green as shown in the diagram above (on the left side). The rows from data frame A and B will be selected only if they have the matched joining column values.

For left joining from data frame A with data frame B (right diagram above) we will keep all the rows in data frame A and only the columns from data frame B will be shown if the joining column values are matched or else they will be shown as null values.

Inner Join

In this example two data frames (dns and map) will be inner joined by using the common column (tac). From the various join methods that are available for the Spark Dataset API this uses the method with signature . The variant with the "usingColumn" has the advantage that you do not need to specify an additional where contition like a.tac === b.tac, and that the resulting data frame only contains 1 column tac instead of 2 columns tac.

Joining of two data frames

```
%spark val dns = Seq(
(23456789, "yahoo.com", 150),
    (98765432, "google.com", 30),
    (12345678, "yahoo.com", 160),
    (23456789, "youtube.com", 20000)
).toDF("tac", "host", "delay") val map
= Seq(     (12345678, "iPhone",
"Smartphone"),
    (23456789, "E70", "Handset"),
    (34567890, "HuaweiE71", "Modem")
).toDF("tac", "model", "category")
dns.show() map.show()

val df = dns.join(map, "tac")

df.show()
```

Output

```
dns: org.apache.spark.sql.DataFrame = [tac: int, host: string ... 1 more
field] map: org.apache.spark.sql.DataFrame = [tac: int, model: string
... 1 more field] +--------+-----------+-----+
|     tac|       host|delay|
+--------+-----------+-----+
|23456789|  yahoo.com|  150|
|98765432| google.com|   30|
|12345678|  yahoo.com|  160|
|23456789|youtube.com|20000|
+--------+-----------+-----+

+--------+---------+----------+
|     tac|    model|  category|
+--------+---------+----------+
|12345678|   iPhone|Smartphone|
|23456789|      E70|   Handset|
|34567890|HuaweiE71|     Modem| +--------+---------+----------
+

df: org.apache.spark.sql.DataFrame = [tac: int, host: string ... 3 more
fields] +--------+-----------+-----+------+----------+
|     tac|       host|delay| model|  category|
+--------+-----------+-----+------+----------+
|23456789|  yahoo.com|  150|   E70|   Handset|
|12345678|  yahoo.com|  160|iPhone|Smartphone|
|23456789|youtube.com|20000|   E70|   Handset|
+--------+-----------+-----+------+----------+
```

Left outer join

In this example left outer join will be used. From the various join methods that are available for the Spark Dataset API which allow to define a join type other than the defaule inner join this uses the method with signature . Again we use the variant with the "usingColumns" for the reasons explained above. However, there is no API variant for a single using column as it exists for the inner join. There is only the variant for multiple using columns, which requires that we put the column name "tac" into a Scala Seq.

Left outer join

```
%spark val dns = Seq(
(23456789, "yahoo.com", 150),
    (98765432, "google.com", 30),
    (12345678, "yahoo.com", 160),
    (23456789, "youtube.com", 20000)
).toDF("tac", "host", "delay") val map
= Seq(     (12345678, "iPhone",
"Smartphone"),
    (23456789, "E70", "Handset"),
    (34567890, "HuaweiE71", "Modem")
).toDF("tac", "model", "category")
dns.show() map.show()

val df = dns.join(map, Seq("tac"), "left_outer")

df.show()
```

Output

```
dns: org.apache.spark.sql.DataFrame = [tac: int, host: string ... 1 more
field] map: org.apache.spark.sql.DataFrame = [tac: int, model: string
... 1 more field] +--------+-----------+-----+
|     tac|       host|delay|
+--------+-----------+-----+
|23456789|  yahoo.com|  150|
|98765432| google.com|   30|
|12345678|  yahoo.com|  160|
|23456789|youtube.com|20000|
+--------+-----------+-----+

+--------+---------+----------+
|     tac|    model|  category|
+--------+---------+----------+
|12345678|   iPhone|Smartphone|
|23456789|      E70|   Handset|
|34567890|HuaweiE71|     Modem| +--------+---------+----------
+

df: org.apache.spark.sql.DataFrame = [tac: int, host: string ... 3 more
fields] +--------+-----------+-----+------+----------+
|     tac|       host|delay| model|  category|
+--------+-----------+-----+------+----------+
|23456789|  yahoo.com|  150|   E70|   Handset|
|98765432| google.com|   30|  null|      null|
|12345678|  yahoo.com|  160|iPhone|Smartphone|
|23456789|youtube.com|20000|   E70|   Handset|
+--------+-----------+-----+------+----------+
```