**Introduction**

A DataFrame is a two-dimensional labeled data structure with columns of potentially different types. You can think of a DataFrame like a spreadsheet, a SQL table, or a dictionary of series objects. Apache Spark DataFrames provide a rich set of functions (select columns, filter, join, aggregate) that allow you to solve common data analysis problems efficiently.

Apache Spark DataFrames are an abstraction built on top of Resilient Distributed Datasets (RDDs). Spark DataFrames and Spark SQL use a unified planning and optimization engine, allowing you to get nearly identical performance across all supported languages on Databricks (Python, SQL, Scala, and R).

In this tutorial we will explore how to use the summary and describe methods make it easy to explore the contents of a DataFrame at a high level.

* **Loading Files Into DataFrames**

In Spark (Scala or PySpark), a data source API is a set of interfaces and classes that allow developers to read and write data from various data sources such as HDFS, HBase, Cassandra, JSON, CSV, and Parquet. The data source API provides a consistent interface for accessing and manipulating data, regardless of the underlying data format or storage system.

* **Creating the DataFrame from CSV file**

There are various ways to read CSV files using Spark. Here are the most common methods.

1. **Using spark.read.csv method:**

*// A CSV dataset is pointed to by path.  
// The path can be either a single CSV file or a directory of CSV files  
val path = "path/to/csv/file.csv"*

*val df = spark.read.option(“delimiter”, “;”).option(“header”, “true”).option(“inferSchema", “true”).csv(path)*

*In this example, we first create a variable to store the path, then we use the spark.read.csv method to read the CSV file located at “path/to/csv/file.csv”.  
We also specify some options such as header = true to indicate that the first row of the CSV file contains column names,  
and inferSchema = true to automatically infer the data types of the columns.  
The resulting DataFrame object will have the same column names and data types as the CSV file.*

1. **Using spark.read.format method:**

*// A CSV dataset is pointed to by path.  
// The path can be either a single CSV file or a directory of CSV files  
val path = "path/to/csv/file.csv"*

*val df = spark.read.format(“csv”).option(“header”, “true”).option(“inferSchema", “true”).option(“delimiter”, “;”).load(path)*

*In this example, we first create a variable to store the path, then we use the spark.read.format("csv") method to read the CSV file located at “path/to/csv/file.csv”.  
We also specify some options such as header = true to indicate that the first row of the CSV file contains column names,  
and inferSchema = true to automatically infer the data types of the columns. Finally, we use the .load method to load the CSV file.*

**Dataset Description**

In this tutorial, we will be using the weather observations dataset (2021-1k.csv). The file has 8 columns but we are interested in the first 4 columns described as following:

| **Field** | **Description** |
| --- | --- |
| **ID** | 11 character station identification code. |
| **YEAR/MONTH/DAY** | 8 character date in YYYYMMDD format (e.g. 19860529 = May 29, 1986). |
| **ELEMENT** | 4 character indicator of element type. |
| **DATA VALUE** | 5 character data value for ELEMENT. |
| **M-FLAG** | 1 character Measurement Flag. |
| **Q-FLAG** | 1 character Quality Flag. |
| **S-FLAG** | 1 character Source Flag. |
| **OBS-TIME** | 4-character time of observation in hour-minute format (i.e. 0700 =7:00 am). |

The input data set contains data for one month or more daily weather data.

To execute the Spark statements of this tutorial we will be using a Zeppelin note with the **spark**interpreter (**%spark**).

%spark

sc.version

**Upload The Input Dataset on HDFS**

To load the the CSV file into a Spark DataFrame, let’s upload it on HDFS. We start by creating a directory on HDFS and then put the file into this directory.

%sh

# upload data to hdfs

hdfs dfs -mkdir -p /tutorials/spark/dataframes/explore

hdfs dfs -put /home/training/Data/1763.csv /tutorials/spark/dataframes/explore

1. **Loading The File Into Spark Dataframe**

The easiest way to load the csv file into a Spark dataframe is to use the **read.format** function (Spark SQL API). We will instruct the function to remove the header from the csv file and infer the schema as we didn’t provide it explicitly.

%spark

// load the file using the SparkSQL API

val weather = spark.read

.format("csv")

.option("header","false")

.option("inferSchema","true")

.load("/tutorials/spark/dataframes/explore/1763.csv")

.toDF("id", "date", "element", "value1", "f1", "f2", "f3", "f4")

.cache

1. **Weather Observations Dataframe Exploration**

Let’s explore the newly loaded dataframe. First we will print its schema. Then we will perform a descriptive columns analysis*.*

%spark

// Print the dataframe schema

weather.printSchema

1. **Check Data Is Loaded**

Use show to verify that the data have been loaded properly. Include LIMIT to retrieve only the first 10 rows.

%spark

// Show first 10 rows

weather.show(10)

Count the number of rows in the dataframe.

%spark

// Show the row count

weather.count

**Using Describe**

Use **describe** to compute some summary statistics on the DataFrame. Using **describe**without any parameter will show some summary statistics on **all**the DataFrame columns.

%spark

// Perform descriptive analytics on column Value1

weather.describe("\*").show

[A white text with black text

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/11/weather.describe-all.png)

As it doesn’t make sense to perform the descriptive analytics on non numerical columns, let’s limit the **describe** statistics for only **Value1** column. The result will include **count, mean, stddev, min, and max** for this column of the dataframe.

%spark

// Perform descriptive analytics on column Value1

weather.describe("value1").show

[A white text with black text

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/11/weather.describe-value1.png)

This option isn’t very useful.

df.select("value1").describe.show would give the same result and is more consistent with the rest of the Spark API.

Let’s turn our attention to **Summary**, a better designed method that provides more useful options.

**Using Summary**

Use **summary** to compute the summary statistics on the DataFrame. Using **summary**without any parameter will show summary statistics on **all**the DataFrame columns.

%spark

// Perform descriptive analytics on the dataframe

weather.summary().show

[A screenshot of a computer code

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/11/weather.summary-all.png)

Let’s customize the output to return the count, 33rd percentile, 50th percentile, and 66th percentile. We will limit the custom summary on the **Value1** column as it doesn’t make sense to compute percentiles for strings columns.

%spark

// Perform descriptive analytics on column Value1

weather.select("value1").summary("count", "33%", "50%", "66%").show

[A white paper with black text and numbers

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/11/summary-percentiles.png)

Using Summary, you are also be able to compute the exact and approximate **count distinct**. Here’s how to get the exact **count** and **distinct count** for **Value1** column:

%spark

// Compute the count and count\_distinct for the dataframe

weather.summary("count", "count\_distinct").show

[A close-up of a number

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/11/summary-count_distinct.png)

Here’s how to get the **approximate count** **distinct** for **Value1** column:

%spark

// Compute the approximate count distinct for the dataframe

weather.summary("count", "approx\_count\_distinct").show(false)

[A close-up of a code

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/11/summary-approx-count-distinct.png)

**Computing Statistics Manually**

We can use **agg** to manually compute the summary statistics for columns in the DataFrame. Here’s how to calculate the **distinct count** for each column in the DataFrame.

%spark

// Compute statistics manually using agg

weather.agg(countDistinct("value1"), countDistinct("element")).show

[A close up of a number

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/11/agg-distinct-count.png)

Here’s how to calculate the **distinct count** and the **max** for **Value1** column in the DataFrame:

%spark

// Calculate the distinct count and the max for Value1

val counts = weather.agg(lit("countDistinct").as("colName"), countDistinct("value1").as("value1"),

countDistinct("elements").as("elements"))

val maxes = weather.agg(lit("max").as("colName"), max("value1").as("value1"),

lit("elements").as("elements"))

counts.union(maxes).show

[A close-up of a number

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/11/agg-distinct-count-max.png)

The code gets verbose quick. **Summary** is great cause it prevents you from writing a lot of code.

Stop Spark context to free resources.

%spark

sc.stop

**Summary**

This tutorial discussed how to use **Describe** and **Summary** to compute statistics summary for a DataFrame. Also we discussed how to compute statistics manually using the **agg**function. Summary is more useful and great for high level exploratory data analysis.

Open Zeppelin Note

[Exploring DataFrames with Summary & Describe](http://localhost:19995/#/notebook/2JJ3EK17A)