JPSWorkshop 8: Use CSV Data in Hadoop, Apache Spark

Table of contents

[**1. Working with Found Datasets 1**](#_heading=h.q212wpdxt7im)

[**2. Hadoop and Comma Separated Values (CSV) Files 1**](#_heading=h.45i3si3zm5rl)

[2.1 Population Data: Population.java 1](#_heading=h.koh2ovj3565p)

[2.2 Running the Program 3](#_heading=h.q8vq36baflk3)

[2.2.1 Exercises to do 5](#_heading=h.4pgfti8ax7xa)

[**3. Apache Spark 6**](#_heading=h.rmnpwo2kcdn6)

[3.2 Student.json 7](#_heading=h.oqeicl9tiwlz)

[3.2.1 Using Data Frames 7](#_heading=h.k6325nyg1ouw)

[3.2.2 Using SQL Queries 8](#_heading=h.d2757lq68fyn)

[3.2.3 Exercises to do 9](#_heading=h.c1atyv8w74ou)

[3.3 Exiting Apache Spark 9](#_heading=h.qka2hzx7y2y)

[3.4 Weather.json 9](#_heading=h.aank425s69el)

[3.4.1 Using Data Frames 9](#_heading=h.hl22pmcd5yzo)

[3.4.2 Using SQL Queries 12](#_heading=h.7i58rtrpe4df)

[**4. Spark and CSV Files 14**](#_heading=h.djonezdoagkt)

[4.1 Spark Queries 15](#_heading=h.lppi1z74r27j)

[4.2 Using SQL 15](#_heading=h.pqdnxb1r6s62)

[4.2.1 Exercises to do 16](#_heading=h.6a1v63qqf31v)

[4.3 HDFS and Apache Spark 16](#_heading=h.6ml6ywpx2klz)

[4.4 Final Word Count 17](#_heading=h.pffamloxee6v)

Before starting the below tasks make sure that you have :

* Switched to the **hadoop user**.
* Navigated to the hadoop **Sbin** directory.
* Started the **Hadoop Distributed File System (HDFS).**
* Started the **YARN Resource Manager**.

**If not** refer to the below image to complete the above tasks :

|  |
| --- |
|  |



If **jps** **(Java Processing Status)** lists all the above **running java processes**, then you are **good to go.**

# Working with Found Datasets

This workbook assumes you have completed **Workshop 7** and are familiar with how to access and run Java programs using Hadoop.

So far the examples have worked with simple text files with no particular structure. To be a meaningful tool Hadoop and its associated projects need to be able to work with more complex data, such as the Excel and JSON data files.

# Hadoop and Comma Separated Values (CSV) Files

CSV files can also be used in Hadoop.

## 2.1 Population Data: Population.java

The following program will work with the pop.csv file. The Word Count examples looked for spaces and punctuation to separate words, whereas in the CSV files, the data will be separated by commas.

The Program is

Population.java

The process is similar to the Word Count programs in that it imports some data and splits the data based on some criteria. Instead of just counting the words found, this program will total up the population values for each County and count how many rows there were. The County Names are

the key in this case.

**Mapping Stage**

In CSV files, the data will be separated by commas, so we need to tell Hadoop to split using this:

String record = value.**toString**();

String[] parts = record.**split**(",");

**Reducer Stage**

By default, Hadoop uses tab (\t) characters to separate words when outputting the results. In this case we are going to output the final results also using commas, so the results could be treated as another CSV file. The output includes both the key and value, the following separates the values using a comma:

for (Text t : values) {

String parts[] = t.**toString**().**split**("\t");

popCount++;

popTotal += Integer.**parseInt**(parts[0]);

} *// for loop*

String str = String.**format**("%d,%d", popCount, popTotal);

For the key we need to change a configuration parameter to change the default:

Configuration conf = new **Configuration**();

*//set output delimiter to comma*

conf.**set**("mapreduce.output.textoutputformat.separator", ",");

**Delete the Output File**

In the Word Count examples, we had to ensure we always deleted the hdfs output directory before running the program.

This program is a bit more sophisticated in that it will delete the folder for you:

Path outputPath = new **Path**(args[1]);

FileOutputFormat.**setOutputPath**(job, outputPath);

*// Delete the output directory - true means if path is a directory it does recursive delete*

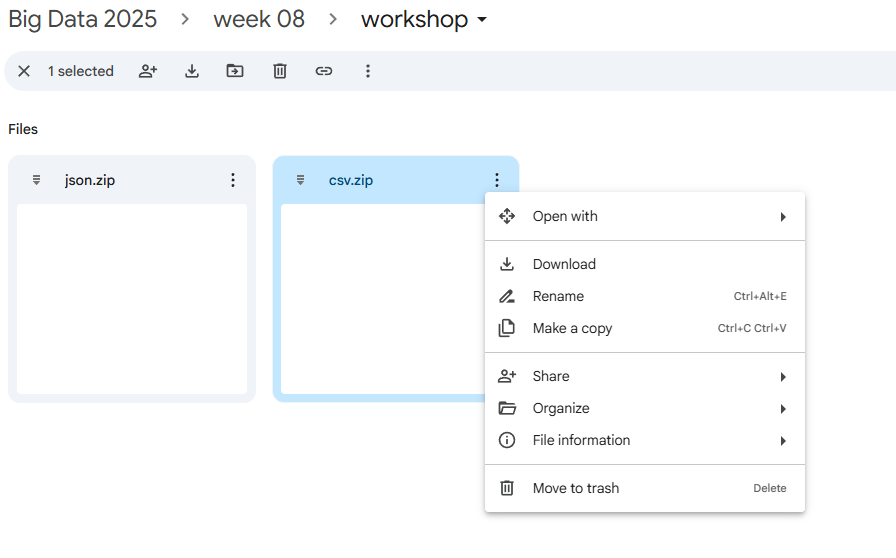
outputPath.**getFileSystem**(conf).**delete**(outputPath, true);

## 2.2 Running the Program

Before running the program we need to **download** the necessary files from the below drive link.

👉 [workshop 8](https://drive.google.com/drive/folders/1-7aZnVDx8OjzTlo21Ue4-WZOpYhCAyR3)

1. Download both **json.zip** and **csv.zip.**

****

1. After downloading them, extract them in the same **Downloads** folder.



1. From the **Ubuntu terminal**, copy the program **Population.java, pop.csv, and Pay.csv** to your linux home directory.

|  |
| --- |
| **~$** cp /mnt/c/Users/**YourWindowsUsername**/Downloads/csv/csv/Population.java ~/  **~$** cp /mnt/c/Users/**YourWindowsUsername**/Downloads/csv/csv/Pop.csv ~/  **~$** cp /mnt/c/Users/**YourWindowsUsername**/Downloads/csv/csv/Pay.csv ~/ |

A screen shot of a computer program

AI-generated content may be incorrect.

*Important: Don’t forget to replace* ***“YourWindowsUsername”*** *with your windows username.*

1. Compile the program first and then create a jar file.

|  |
| --- |
| **~$** cd  **~$** javac -classpath $(hadoop classpath) Population.java  **~$** jar cf Population.jar Pop\*.class |

A screen shot of a computer

AI-generated content may be incorrect.

1. Create a new input directory called **input\_csv** and store the **pop.csv** as well as **pay.csv** file there.

|  |
| --- |
| **~$** hdfs dfs -mkdir /input\_csv  **~$** hdfs dfs -put Pop.csv /input\_csv  **~$** hdfs dfs -put Pay.csv /input\_csv |

**A black background with white text

AI-generated content may be incorrect.**

1. Run the file:

|  |
| --- |
| **~$** hadoop jar Population.jar Population /input\_csv/Pop.csv output\_csv |

A screenshot of a computer screen

AI-generated content may be incorrect.

1. Check if the output file has been created:

|  |
| --- |
| **~$** hdfs dfs -ls output\_csv |

A screenshot of a computer program

AI-generated content may be incorrect.

1. View the results:

|  |
| --- |
| **~$** hdfs dfs -cat output\_csv/part-r-00000 |

A screenshot of a computer program

AI-generated content may be incorrect.

1. To retrieve the results to a csv file:

|  |
| --- |
| **~$** hdfs dfs -get output\_csv/part-r-00000 results.csv |

1. The results.csv can be viewed using normal operating system commands such as:

|  |
| --- |
| **~$** more results.csv |

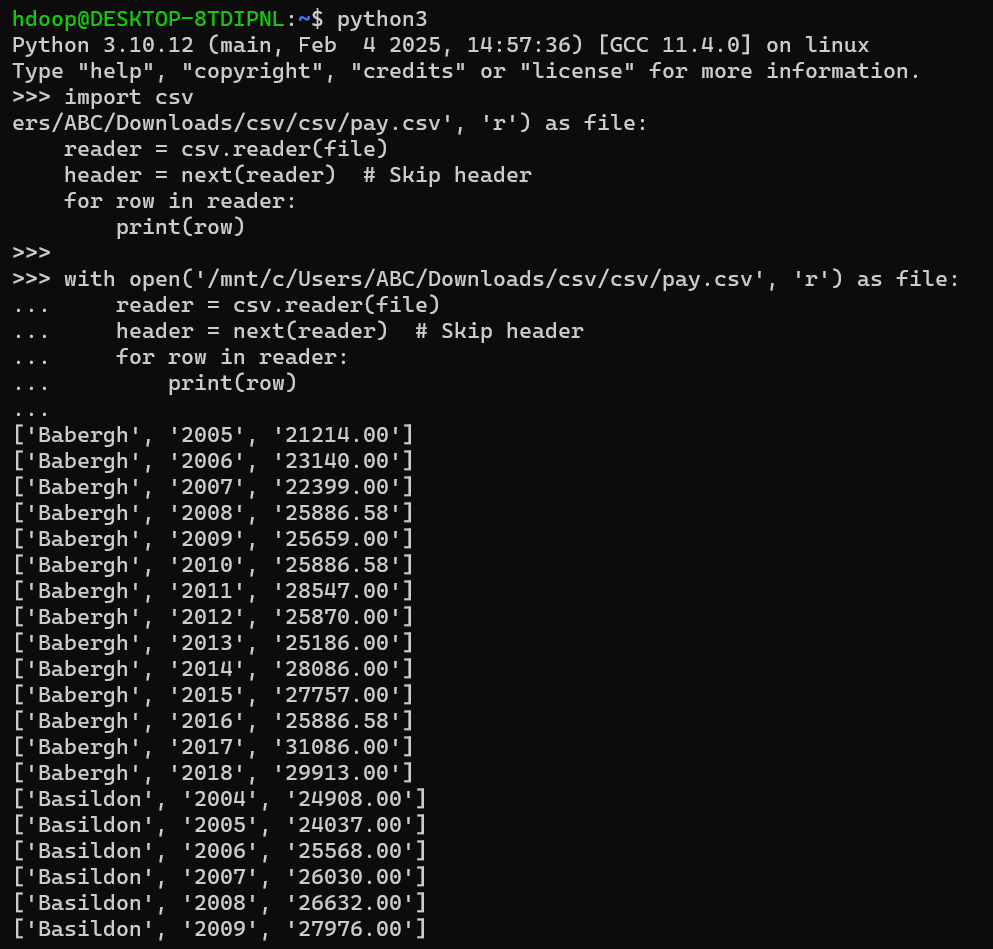
A screenshot of a computer

AI-generated content may be incorrect.

One thing to note is the first line containing the column names no longer exists, so if you wanted to import this data into something that expected this, such as Oracle, you would need to add this information first.

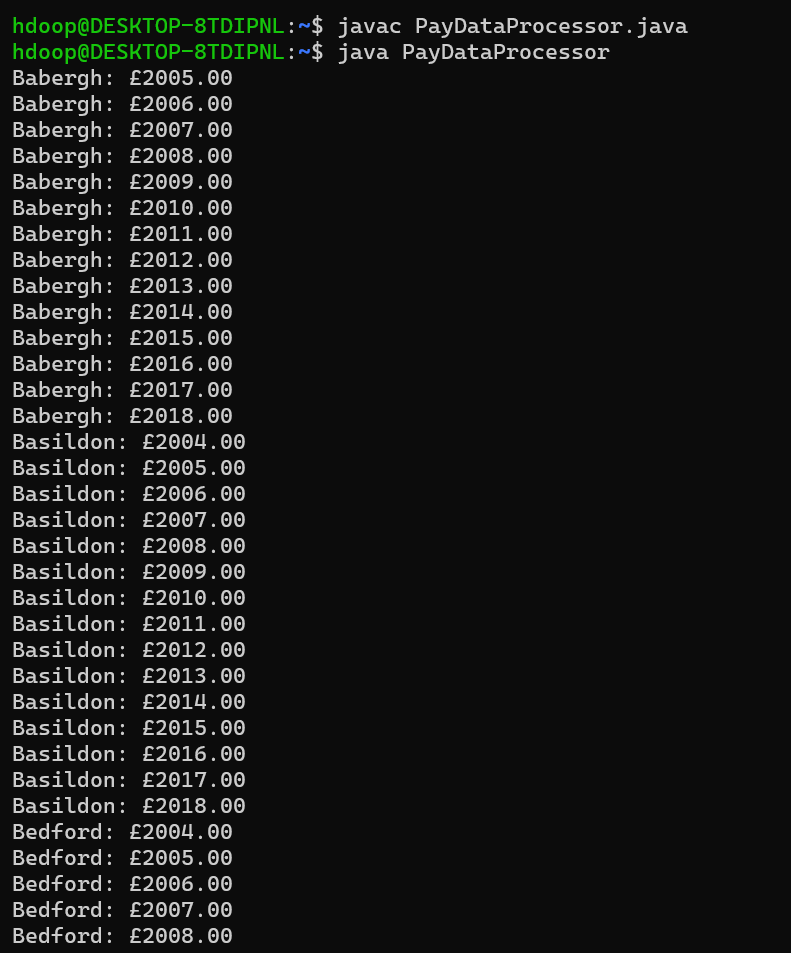
### 2.2.1 Exercises to do

* Pick some of the counties and check that the totals add up!
* Produce a Java file that handles the Pay data instead (**pay.csv**). One thing to note is that the pay totals are currency values, so the numbers will be a float rather than the whole numbers seen in the Population file. You will need to amend the code that handles the figures to account for this.



A computer screen with white text

AI-generated content may be incorrect.



# 3. Apache Spark

There are many ways to access Apache Spark:

* pyspark – uses Python
* spark-shell – uses Scala
* spark-sql – to run SQL queries
* spark-submit – to run a program file, such as Python

Or you can access it via a Java program.

See this webpage for further details and examples:

👉 [Spark SQL, DataFrames and Datasets Guide](https://spark.apache.org/docs/latest/sql-programming-guide.html)

The following examples will use **pyspark**, which allows you to use Python for any programminG tasks.

The **weather.json** dataset from the MongoDB tutorial will also be used for some of the examples, plus an extended version of the **student.json** file seen in the lecture. A copy of these files are available in the downloaded zip files:

weather.json

student.json

To access Spark from the operating system type:

|  |
| --- |
| **~$** pyspark |

To **quit pyspark** type and enter :

## 3.2 Student.json

The following example shows how a JSON file can be imported. A Spark session is automatically available using the **spark** variable.

### 3.2.1 Using Data Frames

The first examples will manipulate the data using a Spark DataFrame, this is equivalent to a relational table in SQL.

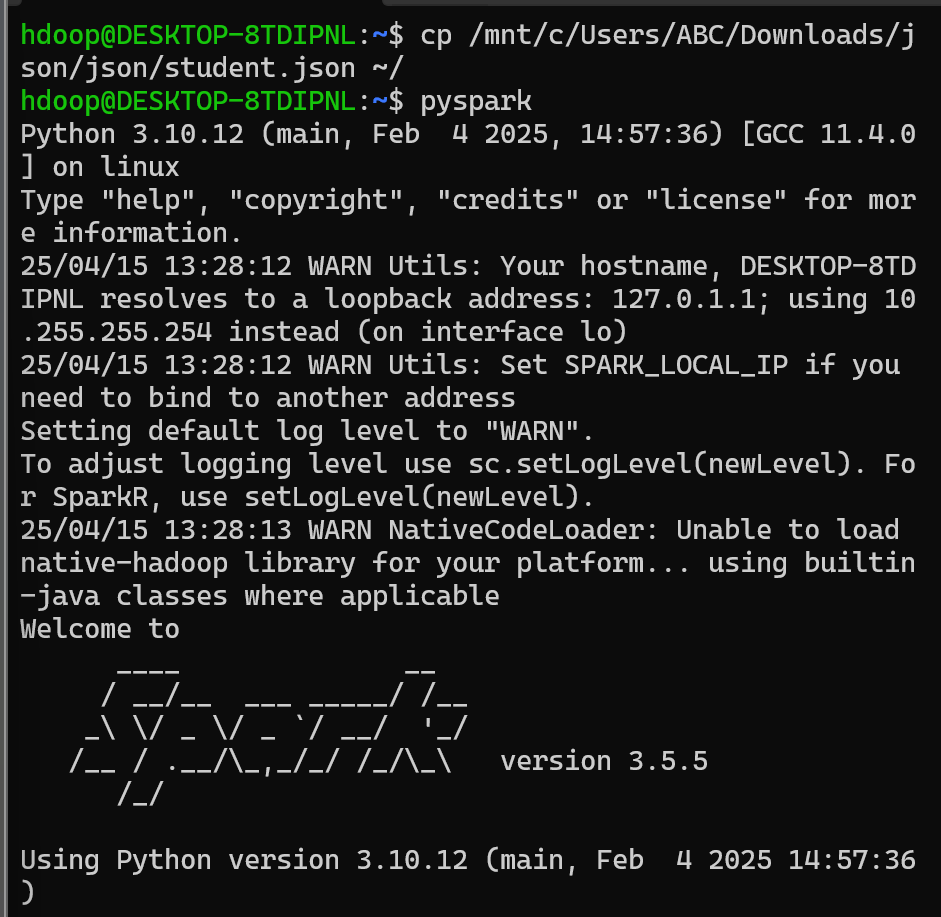
The **Spark DataFrame** is similar to the **pandas DataFrame** found in Python, but some of the methods and what they expect/return may differ slightly.

1. Copy the **student.json** file to your linux home directory.

|  |
| --- |
| ~$ cp /mnt/c/Users/**YourWindowsUsername**/Downloads/json/json/student.json ~/ |

1. Start **pyspark** and create a DataFrame object based on the **student.json** file:

|  |
| --- |
| **~$** pyspark  **>>>** df = spark.read.json("student.json") |



1. This stores the results in the df variable, which is a DataFrame. show() can be used to list the results:

|  |
| --- |
| **>>>** df.show() |

A screenshot of a computer

AI-generated content may be incorrect.

1. DataFrames are structured into columns and rows, to check what the schema is:

|  |
| --- |
| **>>>** df.printSchema() |

A screen shot of a computer program

AI-generated content may be incorrect.

1. **df** represents a DataFrame object, so can be manipulated using methods associated with this data type. For example, to show just the **name** field:

|  |
| --- |
| **>>>** df.select("name").show() |

A screenshot of a computer

AI-generated content may be incorrect.

1. SQL-like operations can now be easily expressed:

|  |
| --- |
| **>>>** df.select(df['name'], df['age'] + 1).show() |

1. **filter** can be used to show only certain rows. To show students older than 21:

|  |
| --- |
| **>>>** df.filter(df['age'] > 21).show() |

A screenshot of a computer

AI-generated content may be incorrect.

1. **groupBy** is similar to the SQL GROUP BY command. To count how many students are on each course:

|  |
| --- |
| **>>>** df.groupBy("course").count().show() |

A screen shot of a computer

AI-generated content may be incorrect.

### 3.2.2 Using SQL Queries

Spark implements a full SQL query engine which can convert SQL statements to a series of Resilient Distributed Dataset (RDD) transformations and actions. This second set of examples will use SQL to query the **DataFrame.**

1. First the DataFrame can be registered as a SQL temporary view:

|  |
| --- |
| **>>>** df.createOrReplaceTempView("student") |

1. This means that “student” can be queried as if it was a SQL table:

|  |
| --- |
| **>>>** sqlDF = spark.sql("SELECT name, age, course FROM student WHERE age > 21")  **>>>** sqlDF.show() |

A computer screen shot of a computer program

AI-generated content may be incorrect.

### 3.2.3 Exercises to do

Using both forms of syntax (Data Frame and SQL) write code to:

* Show just the **name** and **lives** fields

A screenshot of a computer

AI-generated content may be incorrect.

* Count how many people live at each place.

A screenshot of a computer program

AI-generated content may be incorrect.

## 3.3 Exiting Apache Spark

To leave the pyspark environment either type:

**>>>** exit()

Or press ctrl + d

The latter can be used in any of the Spark systems.

## 3.4 Weather.json

This example will use a larger dataset, such as the **weather.json** seen when using MongoDB.

Assuming you left **pyspark** in the previous section, copy the weather.json file into your linux home directory and restart pyspark.

|  |
| --- |
| **~$** cp /mnt/c/Users/**YourWindowsUsername**/Downloads/json/json/weather.json ~/  **~$** pyspark |

### 

A screenshot of a computer program

AI-generated content may be incorrect.

### 3.4.1 Using Data Frames

This time load the weather.json file into a Data Frame and view some data:

|  |
| --- |
| **>>>** df = spark.read.json("weather.json")  **>>>** df.show() |

A screenshot of a computer program

AI-generated content may be incorrect.

By default **show()** only lists the first 20 rows. To show more rows, include a number to represent how many rows should be listed. For example, to show 40 rows:

|  |
| --- |
| **>>>** df.show(40) |

A screenshot of a computer program

AI-generated content may be incorrect.

This time the schema will be a lot more complex, to view it:

|  |
| --- |
| **>>>** df.printSchema() |

A screen shot of a computer program

AI-generated content may be incorrect.

How many records are there?

|  |
| --- |
| **>>>** df.count() |

A black background with white text

AI-generated content may be incorrect.

Show the first row:

|  |
| --- |
| **>>>** df.first() |

A screen shot of a computer screen

AI-generated content may be incorrect.

Or the first two rows:

|  |
| --- |
| **>>>** df.take(2) |

A screen shot of a computer code

AI-generated content may be incorrect.

Show the summary statistics:

|  |
| --- |
| **>>>** df.describe().show() |

A screenshot of a computer program

AI-generated content may be incorrect.

This produces descriptive statistics for numerical columns, such as the count, mean and standard deviation. For example, the first row contains the counts:

|  |
| --- |
| **>>>** df.describe().first() |

A screenshot of a computer program

AI-generated content may be incorrect.

The dot notation can be used to view sub-documents. To show 40 of the user’s screen names:

|  |
| --- |
| **>>>** df.select("user.screen\_name").show(40) |

A screenshot of a computer

AI-generated content may be incorrect.

In a dataframe the **field** name must be in quotes. Either single or double quotes can be used, but you must be consistent.

|  |
| --- |
| **>>>** df.select ('user.screen\_name').show(40) |

A screenshot of a computer

AI-generated content may be incorrect.

but: df.select ('user.screen\_name").show(40)

will generate an error message!

When testing for equality using a data frame, use double equals (==) for the test. For example, show the tweets where the language is English (en):

|  |
| --- |
| **>>>** df.filter(df['lang'] == "en").show() |

A screenshot of a computer screen

AI-generated content may be incorrect.

To restrict both the rows and columns, a pipeline can be set up to pass one command to another:

|  |
| --- |
| **>>>** df.filter(df['lang'] == "en").select('user.screen\_name', 'user.location').show(40) |

A screenshot of a computer

AI-generated content may be incorrect.

This will retrieve the tweets where the language is English, then output just the screen name and location of the user.

By default Spark truncates long text fields, to avoid this use the Boolean **False** as a second parameter to **show()**. In this case you will also have to specify how many rows to show too.

Pattern matching can be done using the **contains** method. For example, find texts containing *sun* and show the full text field:

|  |
| --- |
| **>>>** df.select('text').filter(df['text'].contains("sun")).show(10, False) |

### 

A screenshot of a computer screen

AI-generated content may be incorrect.

### 3.4.2 Using SQL Queries

Create a temporary view containing the weather data:

|  |
| --- |
| **>>>** df.createOrReplaceTempView("weather") |

The equivalent of the last data frame query is as follows (type on one line):

|  |
| --- |
| **>>>** sqlDF = spark.sql("SELECT user.screen\_name, user.location FROM weather WHERE lang = 'en' ") |

A screenshot of a computer

AI-generated content may be incorrect.

This time the field names do not need to be in quotes and the test for equality is a single equal sign. **spark.sql** takes a SQL string as a parameter which must be in either single or double quotes. The query includes a test for a string (en), which must be in different quotes to the SQL query – only single (') or double quotes (") should be used.

For example, the above used double quotes for the SQL query, then single quotes for the value. The following will also work, where single quotes are use for the SQL query and double quotes for the value test:

|  |
| --- |
| **>>>** sqlDF = spark.sql('SELECT user.screen\_name, user.location FROM weather WHERE lang = "en" ') |

A computer screen shot of a black screen

AI-generated content may be incorrect.

What you cannot do is the use the same sort of quotes for both, so the following will not work:

sqlDF = spark.sql('SELECT user.screen\_name, user.location FROM weather WHERE user.lang = 'en' ')

To show 50 rows from the results:

|  |
| --- |
| **>>>** sqlDF.show(50) |

A screenshot of a computer

AI-generated content may be incorrect.

Most standard SQL commands will work. For example, count how many records there are:

|  |
| --- |
| **>>>** sqlDF = spark.sql("SELECT count(\*) AS weather\_count FROM weather").show() |

A black screen with white text

AI-generated content may be incorrect.

Count by language:

|  |
| --- |
| **>>>** sqlDF = spark.sql("SELECT lang, count(\*) AS language\_count FROM weather GROUP BY lang").show() |

A screen shot of a computer

AI-generated content may be incorrect.

The totals should add up to match the previous count!

The SQL **LIKE** command can be used for pattern matching. In MongoDB we listed just the Tweets containing **sun** in the text. The equivalent in SQL would be:

|  |
| --- |
| **>>>** sqlDF = spark.sql("SELECT text FROM weather WHERE text LIKE '%sun%' ").show() |

A screenshot of a computer

AI-generated content may be incorrect.

To make the query case insensitive by forcing the text field into upper case characters:

|  |
| --- |
| **>>>** sqlDF = spark.sql("SELECT text FROM weather WHERE UPPER(text) LIKE '%SUN%' ").show(20, False) |

A screenshot of a computer

AI-generated content may be incorrect.

# 4. Spark and CSV Files

Spark can also read CSV files, as well as JSON format. The following examples will use the **pay.csv** and **pop.csv** files seen in Section 2, however, we will use a version of the files that includes a header in the first row, which can be used for the column headings (**pay-header.csv** and **pop-header.csv**). This is the only difference from the files above.

Quit pyspark:

|  |
| --- |
| **>>>** exit()  OR  **>>>** ctrl + d |

Copy both **pay-header.csv** and **pop-header.csv** :

|  |
| --- |
| **~$** cp /mnt/c/Users/**YourWindowsUsername**/Downloads/csv/csv/pay-header.csv ~/  **~$** cp /mnt/c/Users/**YourWindowsUsername**/Downloads/csv/csv/pop-header.csv ~/ |

### 

Start pyspark:

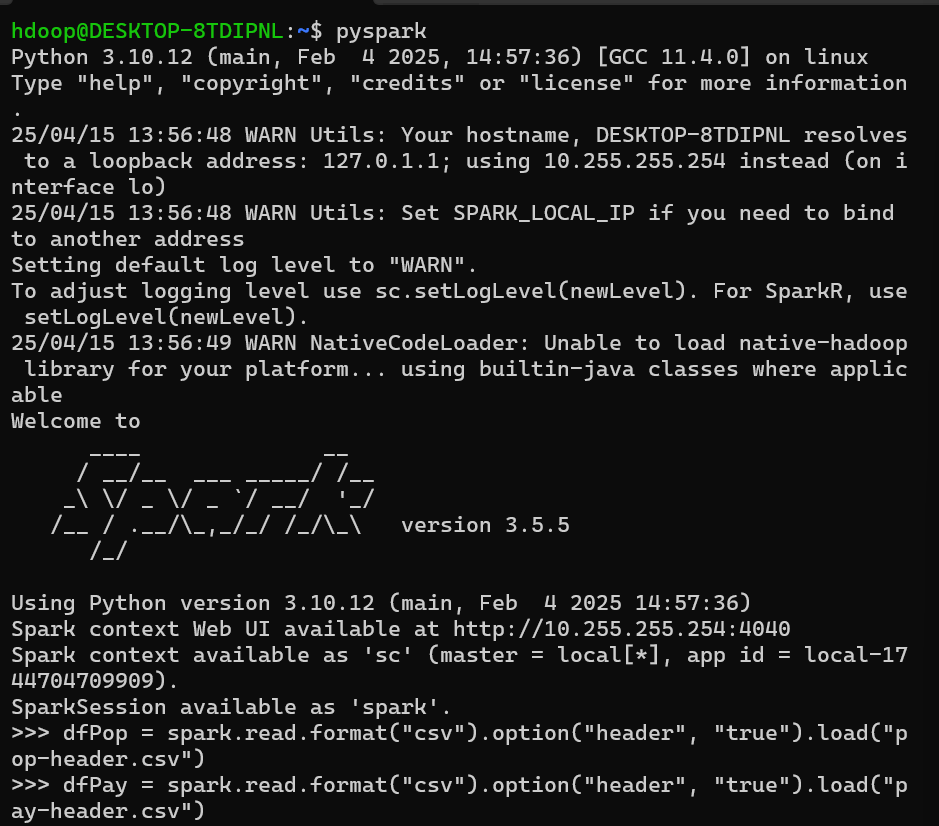
|  |
| --- |
| **~$** pyspark |

### 

Load the two CSV files into separate data frames:

|  |
| --- |
| **>>>** dfPay = spark.read.format("csv").option("header", "true").load("pay-header.csv")  **>>>** dfPop = spark.read.format("csv").option("header", "true").load("pop-header.csv") |

### 



**option("header", "true")** tells Hadoop to use the first line as the column headings.

## 4.1 Spark Queries

Show some data:

|  |
| --- |
| **>>>** dfPop.show()  **>>>** dfPay.show() |

### A screenshot of a computer AI-generated content may be incorrect.

As you can see it has used the headers for the column names. So could just show the county:

|  |
| --- |
| **>>>** dfPop.select("county").show() |

### 

A screenshot of a computer

AI-generated content may be incorrect.

Or just Wolverhampton:

|  |
| --- |
| **>>>** dfPop.filter(dfPop['county'] == 'Wolverhampton').show() |

### A screenshot of a computer AI-generated content may be incorrect.

## 4.2 Using SQL

To manipulate the dataframes using SQL syntax, convert the data frames to views:

|  |
| --- |
| **>>>** dfPop.createOrReplaceTempView("pop")  **>>>** dfPay.createOrReplaceTempView("pay") |

### 

Then we can join them as if they were two SQL tables:

|  |
| --- |
| **>>>** sqlDF = spark.sql("SELECT pop.county, population, annual\_pay FROM pay, pop WHERE pop.county = pay.county and pop.year = pay.year ").show(20, False) |

### A screenshot of a computer screen AI-generated content may be incorrect.

Note, we need to join both on the County name and Year; otherwise we would get a semi Cartesian product:

Table aliases can be used to simplify the query. This query will just show the results for Wolverhampton and the year:

|  |
| --- |
| **>>>** sqlDF = spark.sql("SELECT p.county, p.year, p.population, pa.annual\_pay FROM pay pa, pop p WHERE p.county = pa.county and p.year = pa.year and p.county = 'Wolverhampton' ").show(20, False) |

### 

A screenshot of a computer screen

AI-generated content may be incorrect.

Statistics can be carried out using SQL. For example, count how many rows there are for each county and sum the populations:

|  |
| --- |
| **>>>** sqlDF = spark.sql("SELECT county, count(\*) as pop\_count, SUM(population) as pop\_sum FROM pop GROUP BY county ORDER BY county").show(20, False) |

### 

A screenshot of a computer

AI-generated content may be incorrect.

### 4.2.1 Exercises to do

* List the year, population and annual pay for Walsall.

A screenshot of a computer

AI-generated content may be incorrect.

* Sum the populations by year.

A screenshot of a computer

AI-generated content may be incorrect.

* List each county, with a sum of the annual pay for all years

A screenshot of a computer

AI-generated content may be incorrect.

* List each county, with a sum of the annual pay and population for all years

A screenshot of a computer

AI-generated content may be incorrect.

## 4.3 HDFS and Apache Spark

Spark can also read files from the HDFS file system, which is available via port 9000 on localhost. Assuming that the **pay.csv** and **pop.csv** files are still stored in your HDFS **input\_csv** directory:

|  |
| --- |
| **>>>** dfPay2 = spark.read.format("csv").load("hdfs://localhost:9000/input\_csv/Pay.csv") |

### A screenshot of a computer AI-generated content may be incorrect.

This time the system has not used a header to define the column names:

|  |
| --- |
| **>>>** dfPay2.show() |

### 

So to select one column, use the system generated names, such as **\_c0**:

|  |
| --- |
| **>>>** dfPay2.select("\_c0").show() |

### 

A screenshot of a computer

AI-generated content may be incorrect.

Show the data for Wolverhampton:

|  |
| --- |
| **>>>** dfPay2.filter(dfPay2['\_c0'] == "Wolverhampton").show() |

### 

A screenshot of a computer

AI-generated content may be incorrect.

## 4.4 Final Word Count

Apache Spark also supports Map Reduce, so one last Word Count program.

|  |
| --- |
| **>>>** text\_file = sc.textFile("hdfs://localhost:9000/input\_word")  **>>>** counts = text\_file.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)  **>>>** counts.saveAsTextFile("hdfs://localhost:9000/spark\_output\_word") |

### 



This assumes that there is a text file in the **input\_word** hdfs directory, such as the testfiles or shakespeare.txt used earlier.

**counts** is a **PythonRDD**. A loop is needed to list the results:

|  |
| --- |
| **>>>** for x in counts.collect():  print(x) |

Or it can be converted to a **DataFrame**, then **show()** can be used to view it:

|  |
| --- |
| **>>>** counts.toDF().show() |

**counts.saveAsTextFile** will save the results in a hdfs directory called **spark\_output\_word**. To view this exit pyspark by pressing **Ctrl+D.**

Then list the contents of the spark\_output\_word directory:

|  |
| --- |
| **~$** hdfs dfs -ls /spark\_output\_word |

A screen shot of a computer

AI-generated content may be incorrect.

Assuming the program ran correctly, the results will be similar to:

Found 3 items

-rw-r--r-- 3 hdoop supergroup 0 2025-04-14 03:45 /spark\_output\_word/\_SUCCESS

-rw-r--r-- 3 hdoop supergroup 24697 2025-04-14 03:45 /spark\_output\_word/part-00000

-rw-r--r-- 3 hdoop supergroup 25699 2025-04-14 03:45 /spark\_output\_word/part-00001

The intermediate file (part-0000) still exists, but the final results can be seen by typing:

The intermediate file (**part-0000**) still exists, but the final results can be seen by typing:

|  |
| --- |
| **~$** hdfs dfs -cat /spark\_output\_word/part-00001 |

A black screen with white text

AI-generated content may be incorrect.

The output can be retrieved using the –get option as seen before:

|  |
| --- |
| **~$** hdfs dfs -get /spark\_output\_word/part-00001 spark-results.txt |

A black background with white text

AI-generated content may be incorrect.

# 