Netaji Subhas University of Technology

AI Hardware and Tools
Unit - 4

Yash Kumar - 2021UCA1855
Sparsh Singh - 2021UCA1844
Nawed Akhtar - 2021UCA1866

APACHE SPARK

Apache Spark is an open-source distributed computing system designed for large-scale data processing and analytics. It provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.

Benefits of Apache Spark

1. **Speed** - Engineered from the bottom-up for performance, Spark can be 100x faster than Hadoop for large scale data processing by exploiting in memory computing and other optimizations. Spark is also fast when data is stored on disk, and currently holds the world record for large-scale on-disk sorting.

2. **Ease of Use** - Spark has easy-to-use APIs for operating on large datasets. This includes a collection of over 100 operators for transforming data and familiar data frame APIs for manipulating semi-structured data.

3. **A Unified Engine** - Spark comes packaged with higher-level libraries, including support for SQL queries, streaming data, machine learning and graph processing. These standard libraries increase developer productivity and can be seamlessly combined to create complex workflows.

Key Components of Apache Spark

1. **Unified -** Spark's key driving goal is to offer a unified platform for writing big data applications. What do we mean by unified? Spark is designed to support a wide range of data analytics tasks, ranging from simple data loading and SQL queries to machine learning and streaming computation, over the same computing engine and with a consistent set of APIs. The main insight behind this goal is that real-world data analytics tasks - whether they are interactive analytics in a tool such as a Jupyter notebook, or traditional software development for production applications - tend to combine many different processing types and libraries.

2. **Computing Engine:** At the same time that Spark strives for unification, Spark carefully limits its scope to a computing engine. By this, we mean that Spark only handles loading data from storage systems and performing computation on it, not permanent storage as the end itself. Spark can be used with a wide variety of persistent storage systems, including cloud storage systems such as Azure Storage and Amazon S3, distributed file systems such as Apache Hadoop, key-value stores such as Apache Cassandra, and message buses such as Apache Kafka.

3. **Libraries -** Spark's final component is its libraries, which build on its design as a unified engine to provide a unified API for common data analysis tasks. Spark supports both standard libraries that ship with the engine, and a wide array of external libraries published as third-party packages by the open source communities. Today, Spark's standard libraries are actually the

bulk of the open source project: the Spark core engine itself has changed little since it was first released, but the libraries have grown to provide more and more types of functionality.

RDD IN SPARK

RDD represents Resilient Distributed Dataset. An RDD in Spark is simply an immutable distributed collection of objects sets. Each RDD is split into multiple partitions (similar pattern with smaller sets), which may be computed on different nodes of the cluster.

Create RDD

Usually, there are two popular ways to create the RDDs: loading an external dataset, or distributing a set of collection of objects. The following examples show some simplest ways to create RDDs by using parallelize() fucntion which takes an already existing collection in your program and pass the same to the Spark Context.

1. By using parallelize( ) function

```python
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

df = spark.sparkContext.parallelize([(1, 2, 3, 'a b c'),
             (4, 5, 6, 'd e f'),
             (7, 8, 9, 'g h i')]).toDF(['col1', 'col2', 'col3','col4'])
```

Then you will get the RDD data:

```
df.show()

+----+----+----+-----+
```

```
|col1|col2|col3| col4|
+----+----+----+-----+
|   1|   2|   3|a b c|
|   4|   5|   6|d e f|
|   7|   8|   9|g h i|
+----+----+----+-----+
```

```python
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

myData = spark.sparkContext.parallelize([(1,2), (3,4), (5,6), (7,8), (9,10)])
```

Then you will get the RDD data:

```
myData.collect()

[(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
```

2. By using createDataFrame( ) function

```python
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

Employee = spark.createDataFrame([
                        ('1', 'Joe',   '70000', '1'),
                        ('2', 'Henry', '80000', '2'),
                        ('3', 'Sam',   '60000', '2'),
                        ('4', 'Max',   '90000', '1')],
                        ['Id', 'Name', 'Sallary','DepartmentId']
                       )
```

Then you will get the RDD data:

```
+---+-----+-------+------------+
| Id| Name|Sallary|DepartmentId|
+---+-----+-------+------------+
|  1|  Joe|  70000|           1|
|  2|Henry|  80000|           2|
|  3|  Sam|  60000|           2|
|  4|  Max|  90000|           1|
+---+-----+-------+------------+
```

3. By using read and load functions

We here show example of read data from .csv file

```python
## set up  SparkSession
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

df = spark.read.format('com.databricks.spark.csv').\
                               options(header='true', \
                               inferschema='true').\
                load("/home/feng/Spark/Code/data/Advertising.csv",
↪header=True)

df.show(5)
df.printSchema()
```

Then you will get the RDD data:

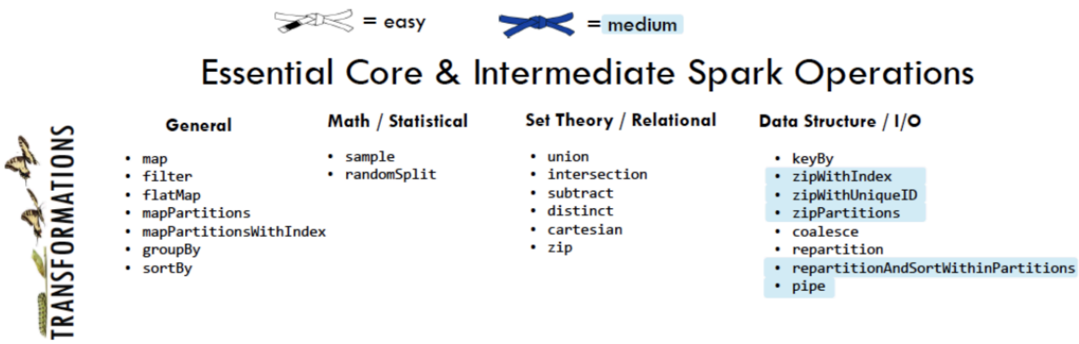
```
+---+-----+-----+---------+-----+
|_c0|   TV|Radio|Newspaper|Sales|
+---+-----+-----+---------+-----+
|  1|230.1| 37.8|     69.2| 22.1|
|  2| 44.5| 39.3|     45.1| 10.4|
|  3| 17.2| 45.9|     69.3|  9.3|
|  4|151.5| 41.3|     58.5| 18.5|
|  5|180.8| 10.8|     58.4| 12.9|
+---+-----+-----+---------+-----+
only showing top 5 rows

root
 |-- _c0: integer (nullable = true)
 |-- TV: double (nullable = true)
 |-- Radio: double (nullable = true)
 |-- Newspaper: double (nullable = true)
 |-- Sales: double (nullable = true)
```

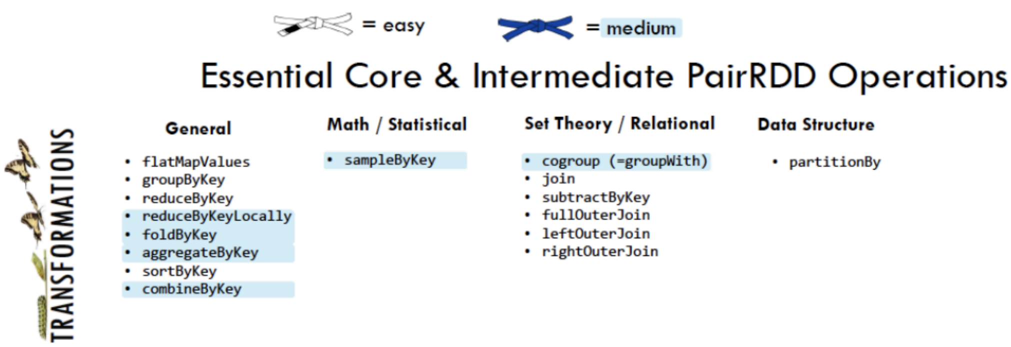
Spark Operations

There are two main types of Spark operations: Transformations and Actions [Karau2015].

1. Spark Transformations

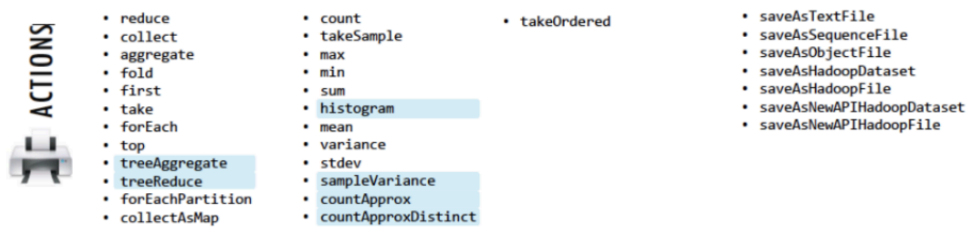
Transformations construct a new RDD from a previous one. For example, one common transformation is filtering data that matches a predicate.





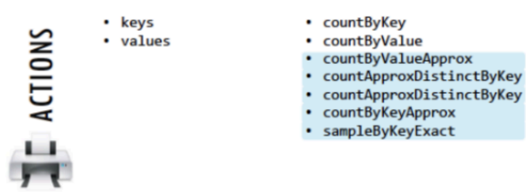
2. Spark Actions

Actions on the other hand, compute a result based on RDD, and either return it to the driver program or save it to an external storage system(e.g.,HDFS).



2 **Chapter 5. Programming with RDDs**

Learning Apache Spark with Python



rdd.DataFrame vs pd.DataFrame

1. Create DataFrame

a. From List

```
my_list = [['a', 1, 2], ['b', 2, 3],['c', 3, 4]]
col_name = ['A', 'B', 'C']
```

:: Python Code:

```
# caution for the columns=
pd.DataFrame(my_list,columns= col_name)
#
spark.createDataFrame(my_list, col_name).show()
```

:: Comparison:

```
                     +---+---+---+
                     |  A|  B|  C|
   A  B  C           +---+---+---+
0  a  1  2           |  a|  1|  2|
1  b  2  3           |  b|  2|  3|
2  c  3  4           |  c|  3|  4|
                     +---+---+---+
```

```
   A  B  C              0  1  2
0  a  1  2           A  a  1  2
1  b  2  3           B  b  2  3
2  c  3  4           C  c  3  4
```

b. From Dict

```
d = {'A': [0, 1, 0],
     'B': [1, 0, 1],
     'C': [1, 0, 0]}
```

:: Python Code:

```
pd.DataFrame(d)for
# Tedious for PySpark
spark.createDataFrame(np.array(list(d.values())).T.tolist(),list(d.keys())).
↪show()
```

:: Comparison:

```
                     +---+---+---+
                     |  A|  B|  C|
   A  B  C           +---+---+---+
0  0  1  1           |  0|  1|  1|
1  1  0  0           |  1|  0|  0|
2  0  1  0           |  0|  1|  0|
                     +---+---+---+
```

2. First n Rows

```
:: Python Code:
```

```
dp.head(4)
#
ds.show(4)
```

```
:: Comparison:
```

```
                                                +-----+-----+---------+-----+
                                                |   TV|Radio|Newspaper|Sales|
      TV  Radio  Newspaper  Sales               +-----+-----+---------+-----+
0  230.1   37.8       69.2   22.1               |230.1| 37.8|     69.2| 22.1|
1   44.5   39.3       45.1   10.4               | 44.5| 39.3|     45.1| 10.4|
2   17.2   45.9       69.3    9.3               | 17.2| 45.9|     69.3|  9.3|
3  151.5   41.3       58.5   18.5               |151.5| 41.3|     58.5| 18.5|
                                                +-----+-----+---------+-----+
                                                only showing top 4 rows
```

3. Replace Values

```
:: Python Code:
```

```
# caution: you need to chose specific col
dp.A.replace(['male', 'female'],[1, 0], inplace=True)
dp
#caution: Mixed type replacements are not supported
ds.na.replace(['male','female'],['1','0']).show()
```

```
:: Comparison:
```

```
                                +---+---+----+
                                |  A|  B|   C|
   A  B    C                    +---+---+----+
0  1  1  NaN                    |  1|  1|null|
1  0  2  3.0                    |  0|  2|   3|
2  1  3  4.0                    |  1|  3|   4|
                                +---+---+----+
```

4. Pivot

:: Python Code:

```
pd.pivot_table(dp, values='col3', index='col1', columns='col2', aggfunc=np.
↪sum)
#
ds.groupBy(['col1']).pivot('col2').sum('col3').show()
```

:: Comparison:

```
                                          +----+----+----+----+
col2    2     5     8                     |col1|   2|   5|   8|
col1                                      +----+----+----+----+
a      6.0   NaN   NaN                    |   c|null|null|  18|
b      NaN  12.0   NaN                    |   b|null|  12|null|
c      NaN   NaN  18.0                    |   a|   6|null|null|
                                          +----+----+----+----+
```

Other Features of RDD

1. **Lazy Evaluation**: RDDs support lazy evaluation, meaning transformations are not immediately executed. Instead, they are queued up until an action is called. This optimization allows Spark to optimize the execution plan and minimize data shuffling.

2. **Immutability**: RDDs are immutable, meaning once created, their content cannot be changed. However, transformations can create new RDDs with modified content.

3. **Fault Tolerance**: RDDs are fault-tolerant. Spark keeps track of the lineage of each RDD, allowing it to reconstruct lost data due to failures.

4. **Partitioning**: RDDs are divided into partitions, which are the basic units of parallelism in Spark. Users can control the number of partitions or rely on Spark's default partitioning mechanism.

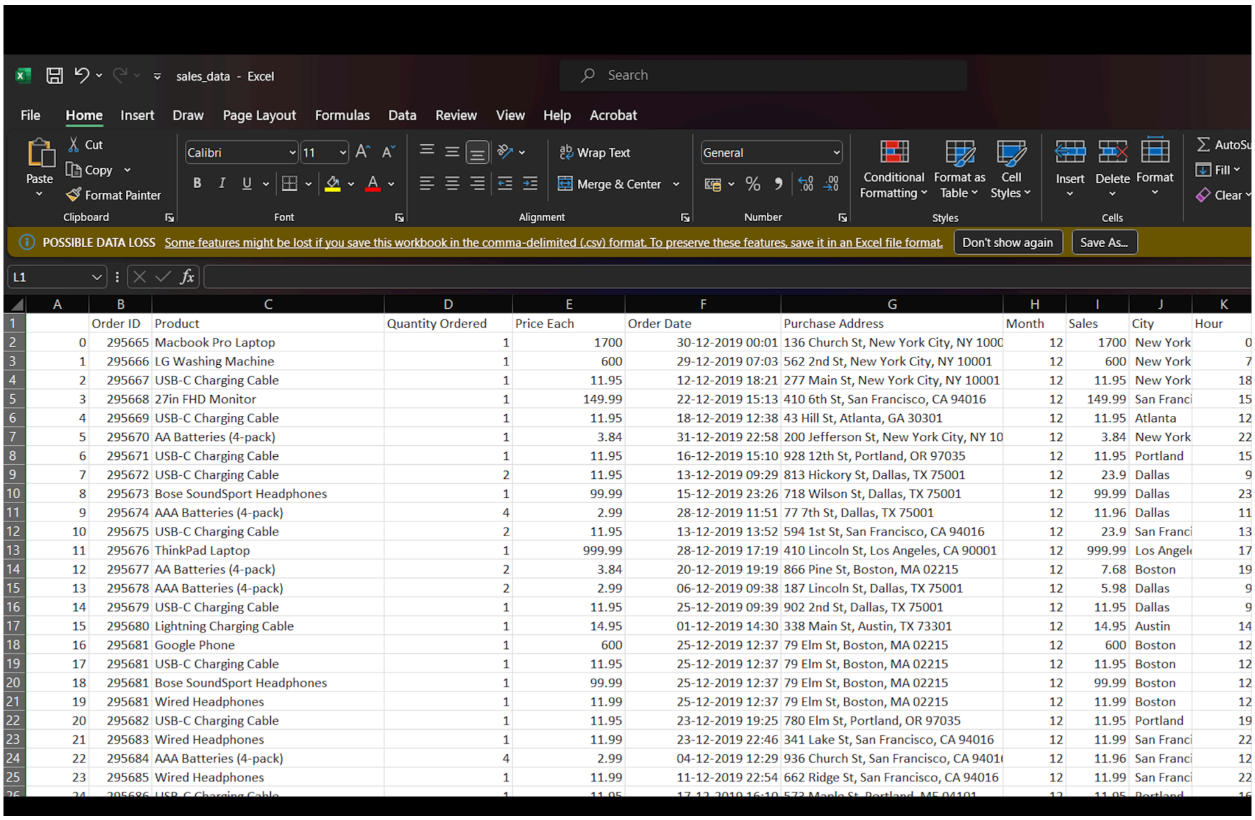
5. **Persistence**: RDDs can be persisted in memory or disk to avoid recomputation, using methods like cache or persist.

6. **Supported Data Types**: RDDs in PySpark can contain any Python object, including user-defined classes, lists, tuples, etc. Spark automatically serializes and distributes these objects across the cluster.

PREPROCESSING .CSV FILE,DATA CLEANING AND TOTAL SALES AMOUNT CALCULATION IN PYSPARK SCRIPT

For this particular task we designed the PySpark script to process sales data stored in a sales_data.csv file which was available on kaggle, demonstrating essential data manipulation and analysis techniques within the PySpark framework. PySpark, the Python API for Apache Spark, enables scalable data processing and analysis, particularly suitable for handling large datasets distributed across a cluster.

Our dataset is a comprehensive collection of information related to sales transactions across various industries and markets. This dataset typically includes details such as the date of the transaction, the product or service sold, the quantity sold, the price, and possibly additional attributes like customer demographics or geographic location.



	A	B	C	D	E	F	G	H	I	J	K
1		Order ID	Product	Quantity Ordered	Price Each	Order Date	Purchase Address	Month	Sales	City	Hour
2	0	295665	Macbook Pro Laptop	1	1700	30-12-2019 00:01	136 Church St, New York City, NY 1000	12	1700	New York	0
3	1	295666	LG Washing Machine	1	600	29-12-2019 07:03	562 2nd St, New York City, NY 10001	12	600	New York	7
4	2	295667	USB-C Charging Cable	1	11.95	12-12-2019 18:21	277 Main St, New York City, NY 10001	12	11.95	New York	18
5	3	295668	27in FHD Monitor	1	149.99	22-12-2019 15:13	410 6th St, San Francisco, CA 94016	12	149.99	San Franci	15
6	4	295669	USB-C Charging Cable	1	11.95	18-12-2019 12:38	43 Hill St, Atlanta, GA 30301	12	11.95	Atlanta	12
7	5	295670	AA Batteries (4-pack)	1	3.84	31-12-2019 22:58	200 Jefferson St, New York City, NY 10	12	3.84	New York	22
8	6	295671	USB-C Charging Cable	1	11.95	16-12-2019 15:10	928 12th St, Portland, OR 97035	12	11.95	Portland	15
9	7	295672	USB-C Charging Cable	2	11.95	13-12-2019 09:29	813 Hickory St, Dallas, TX 75001	12	23.9	Dallas	9
10	8	295673	Bose SoundSport Headphones	1	99.99	15-12-2019 23:26	718 Wilson St, Dallas, TX 75001	12	99.99	Dallas	23
11	9	295674	AAA Batteries (4-pack)	4	2.99	28-12-2019 11:51	77 7th St, Dallas, TX 75001	12	11.96	Dallas	11
12	10	295675	USB-C Charging Cable	2	11.95	13-12-2019 13:52	594 1st St, San Francisco, CA 94016	12	23.9	San Franci	13
13	11	295676	ThinkPad Laptop	1	999.99	28-12-2019 17:19	410 Lincoln St, Los Angeles, CA 90001	12	999.99	Los Angel	17
14	12	295677	AA Batteries (4-pack)	2	3.84	20-12-2019 19:19	866 Pine St, Boston, MA 02215	12	7.68	Boston	19
15	13	295678	AAA Batteries (4-pack)	2	2.99	06-12-2019 09:38	187 Lincoln St, Dallas, TX 75001	12	5.98	Dallas	9
16	14	295679	USB-C Charging Cable	1	11.95	25-12-2019 09:39	902 2nd St, Dallas, TX 75001	12	11.95	Dallas	9
17	15	295680	Lightning Charging Cable	1	14.95	01-12-2019 14:30	338 Main St, Austin, TX 73301	12	14.95	Austin	14
18	16	295681	Google Phone	1	600	25-12-2019 12:37	79 Elm St, Boston, MA 02215	12	600	Boston	12
19	17	295681	USB-C Charging Cable	1	11.95	25-12-2019 12:37	79 Elm St, Boston, MA 02215	12	11.95	Boston	12
20	18	295681	Bose SoundSport Headphones	1	99.99	25-12-2019 12:37	79 Elm St, Boston, MA 02215	12	99.99	Boston	12
21	19	295681	Wired Headphones	1	11.99	25-12-2019 12:37	79 Elm St, Boston, MA 02215	12	11.99	Boston	12
22	20	295682	USB-C Charging Cable	1	11.95	23-12-2019 19:25	780 Elm St, Portland, OR 97035	12	11.95	Portland	19
23	21	295683	Wired Headphones	1	11.99	23-12-2019 22:46	341 Lake St, San Francisco, CA 94016	12	11.99	San Franci	22
24	22	295684	AAA Batteries (4-pack)	4	2.99	04-12-2019 12:29	936 Church St, San Francisco, CA 9401	12	11.96	San Franci	12
25	23	295685	Wired Headphones	1	11.99	11-12-2019 22:54	662 Ridge St, San Francisco, CA 94016	12	11.99	San Franci	22

1. **Initialization** - First we initialize a SparkSession with the app name "SalesDataProcessing".`SparkSession` is the entry point to PySpark functionality and is used to create DataFrames and execute SQL queries.The specific functions (`col` and `sum`) are

imported from the `pyspark.sql.functions` module. These functions are commonly used for data manipulation and aggregation tasks within PySpark DataFrame operations.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("SalesDataProcessing") \
    .getOrCreate()
```

2. **Reading Data:** Now the script reads a CSV file containing sales data into a DataFrame (`sales_df`).

```
# Read CSV file containing sales data
sales_df = spark.read.csv("sales_data.csv", header=True, inferSchema=True)
```

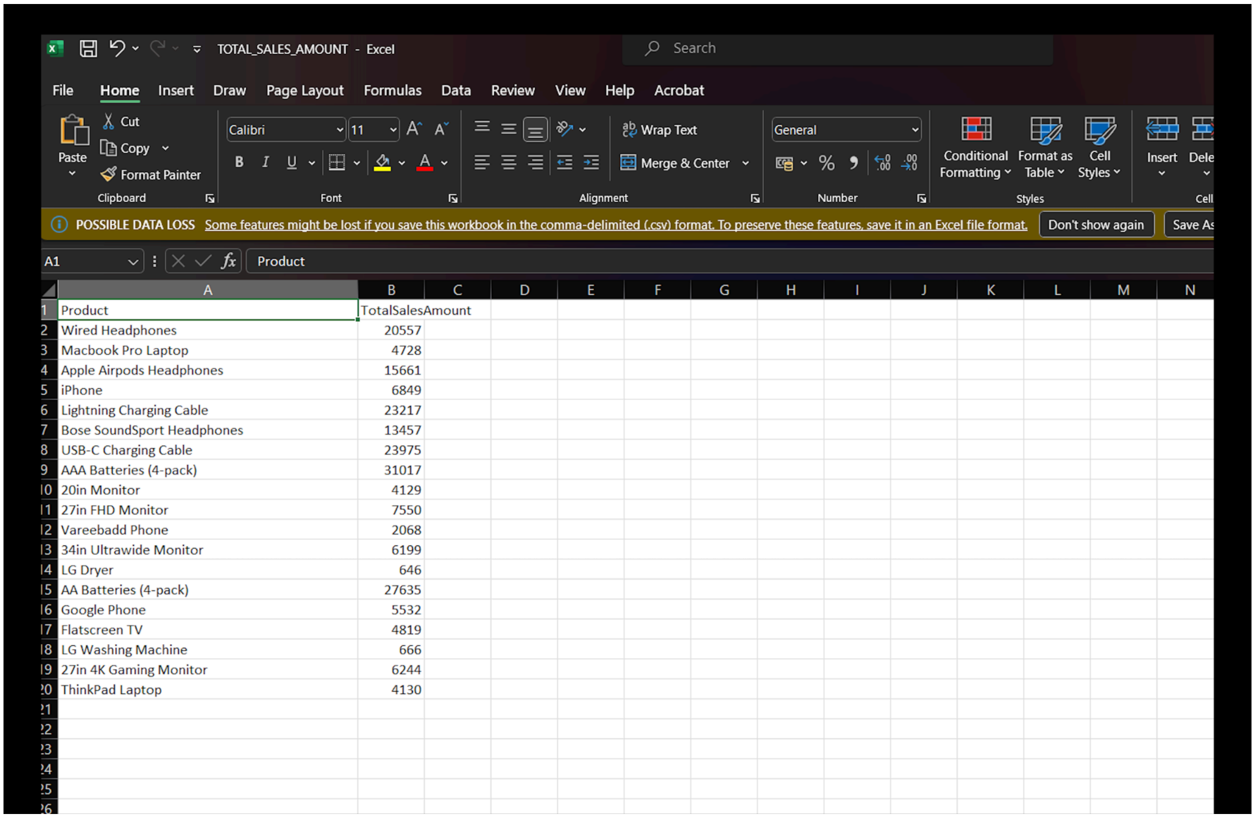
3. **Data Cleaning:** Removes rows with missing values (`na.drop()`) from the DataFrame, resulting in `cleaned_sales_df`. We also remove duplicates by dropping duplicate rows from the cleaned DataFrame, resulting in `deduplicated_sales_df`.

```
# Data cleaning: Handling missing values
cleaned_sales_df = sales_df.na.drop()

# Removing duplicates
deduplicated_sales_df = cleaned_sales_df.dropDuplicates()
```

4. **Aggregation and Output:** Groups the data by the "Product" column and calculates the total sales amount for each product using the `sum` aggregation function, stored in `total_sales_df`. Now it writes the aggregated DataFrame (`total_sales_df`) to a new CSV file named "total_sales_amount_per_product.csv" with headers.

```
# Calculate total sales amount for each product
total_sales_df = deduplicated_sales_df.groupBy("Product").agg(sum("Quantity Ordered").alias("TotalSalesAmount"))

# Output results to a new CSV file
total_sales_df.write.csv("TOTAL_SALES_AMOUNT.csv", header=True)
```

The generated csv file is as follows:



Product	TotalSalesAmount
Wired Headphones	20557
Macbook Pro Laptop	4728
Apple Airpods Headphones	15661
iPhone	6849
Lightning Charging Cable	23217
Bose SoundSport Headphones	13457
USB-C Charging Cable	23975
AAA Batteries (4-pack)	31017
20in Monitor	4129
27in FHD Monitor	7550
Vareebadd Phone	2068
34in Ultrawide Monitor	6199
LG Dryer	646
AA Batteries (4-pack)	27635
Google Phone	5532
Flatscreen TV	4819
LG Washing Machine	666
27in 4K Gaming Monitor	6244
ThinkPad Laptop	4130

5. **SparkSession Stop:** Stops the SparkSession to release resources.

```
# Stop SparkSession
spark.stop()
```