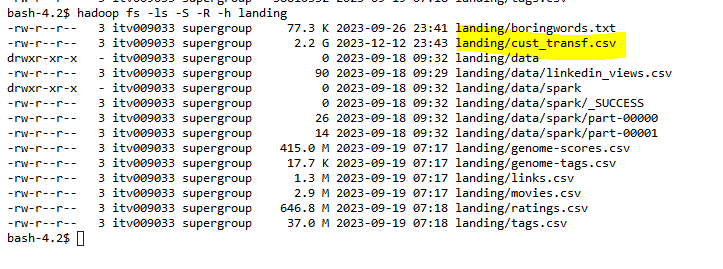
**Choose any datasets of your choice in the public folder:**



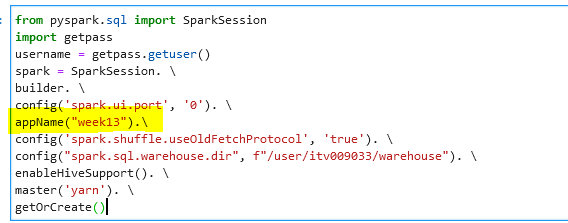
→ We have created a landing folder in our HDFS home to bring the data from external locations i.e. the public folder in this case

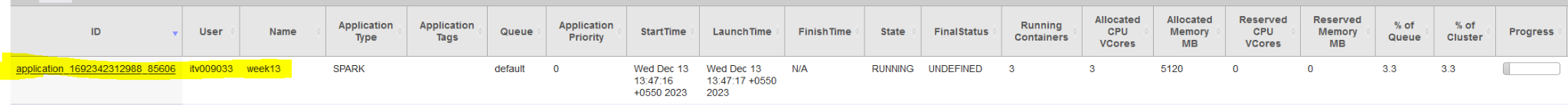
→ command for taking the dataset from the public folder to our landing folder:

hadoop fs -cp /public/trendytech/datasets/cust\_transf.csv landing

**1. Analyse the datasets chosen and come up with an example use-case.**

→ There are 4 columns in the dataset, namely **o\_id int, date date, c\_id int and amount float,** the size of the dataset is around 2.2 GB and it has around 18 partitions





→ Let’s give the schema first:

from pyspark.sql.types import StructType, StructField, TimestampType, IntegerType, StringType, DateType, FloatType

from pyspark.sql.functions import from\_unixtime, expr

customers\_schema = StructType([StructField("o\_id", IntegerType(), True)

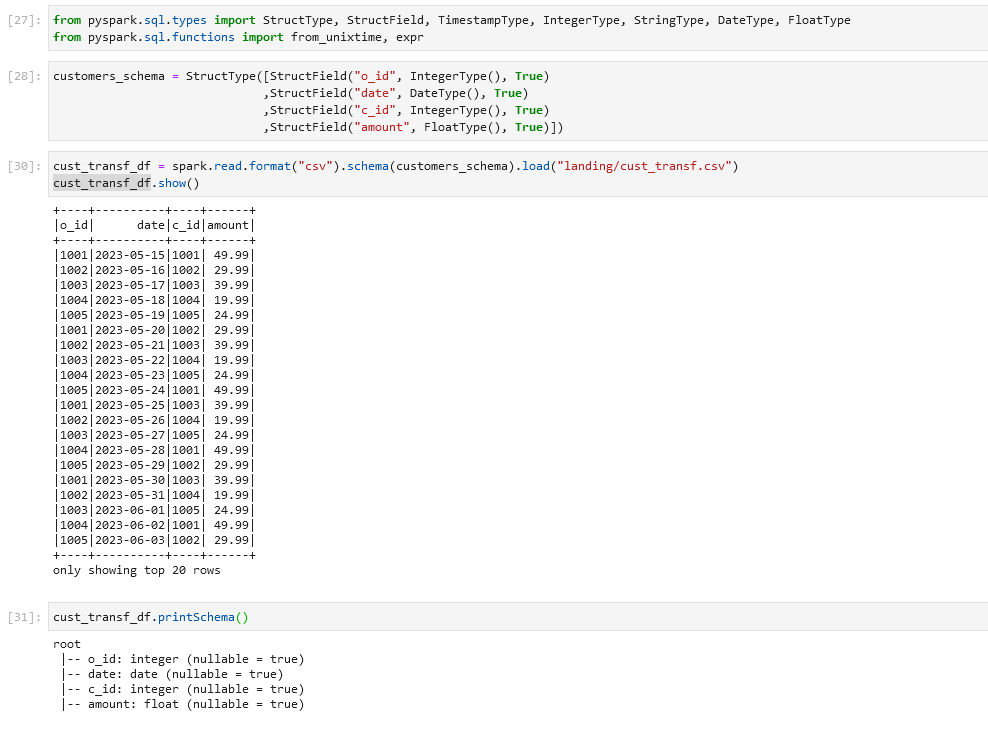
,StructField("date", DateType(), True)

,StructField("c\_id", IntegerType(), True)

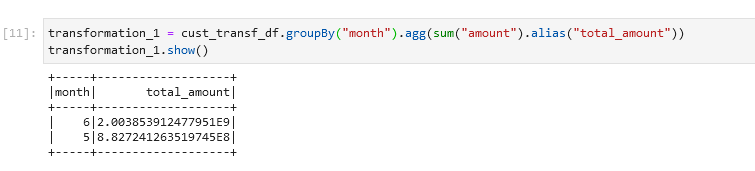
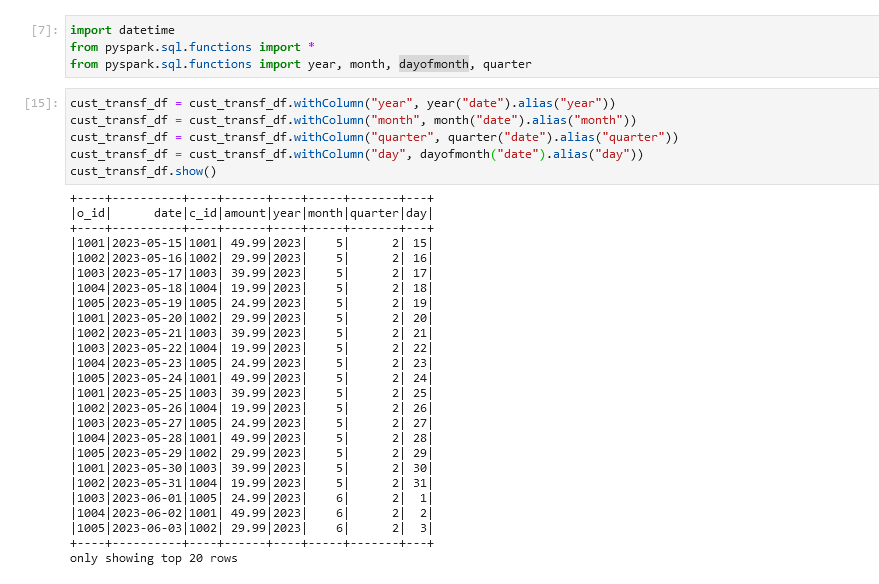
,StructField("amount", FloatType(), True)])

cust\_transf\_df = spark.read.format("csv").schema(customers\_schema).load("landing/cust\_transf.csv")

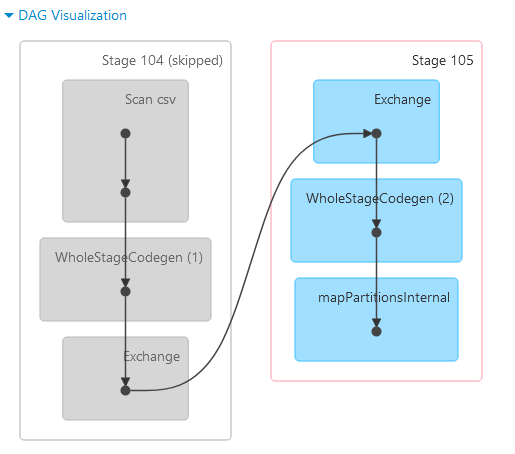
cust\_transf\_df.show()

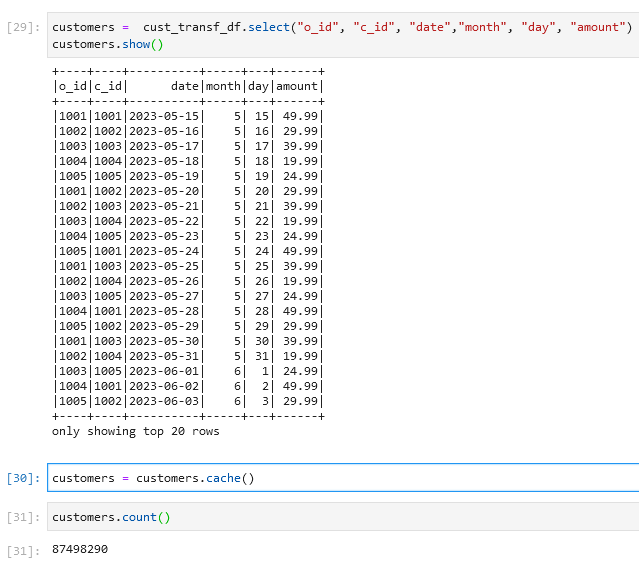


# Use Case 1 :

\* Let's breakdown the date column and find out the amount collected per month, year and Quarter

→ As we can see that there is only two months data that is for the months of May and June only and for the same year i.e. 2023, thus it is futile to use the Quarter and the year columns. Thus we will not be using them going forward.



→ We can see that as in the previous transformation, we used a wide transformation, thus shuffling is happening.Thus, we select a subset of relevant columns

**3. Execute the Pyspark code using the spark-submit utility:**

Spark3-submit\

--deploy-mode client\

--master yarn\

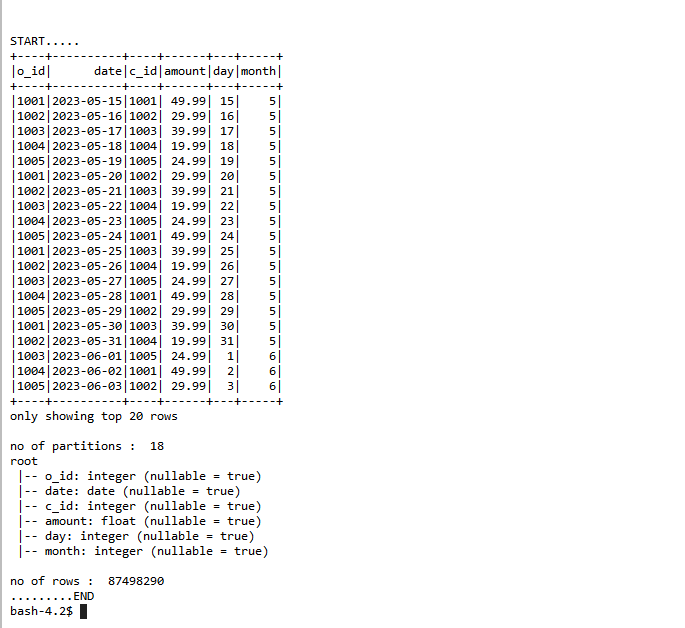
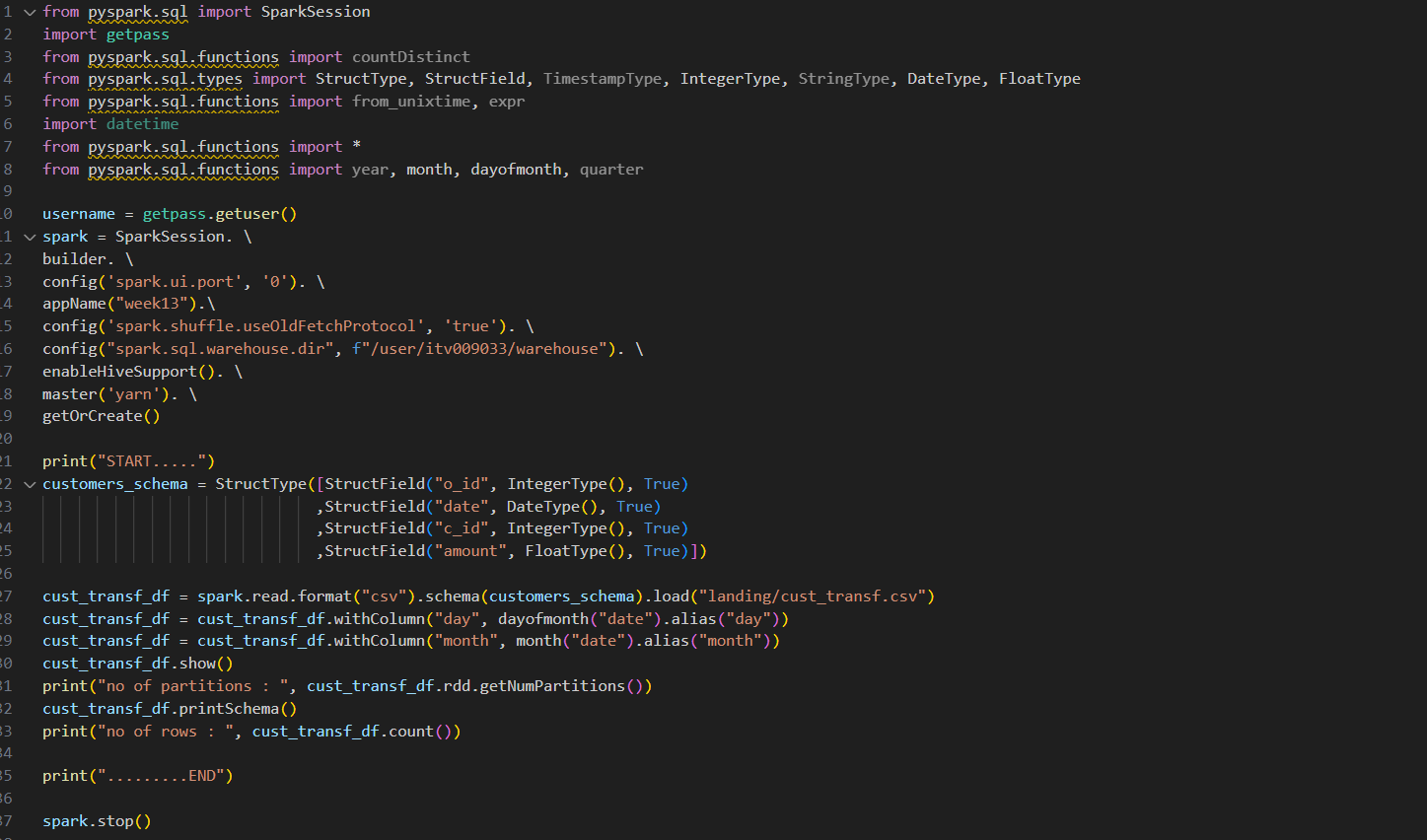
--num-executors 1\

--executor-cores 1\

--executor-memory 1G\

--verbose\

Week13.py



Dynamic allocation was enabled by default in the above case



**4. Now, try to vary the number of executors, executor-memory,**

**executor-cores and present your inference with relevant explanation**

**and screenshots of the results. (Perform this by disabling the**

**dynamic-memory allocation feature of pyspark)**

Disabled dynamic allocation and no of executors requested = 1

Spark3-submit\

--deploy-mode client\

**--conf spark.dynamicAllocation.enabled=false**\

--master yarn\

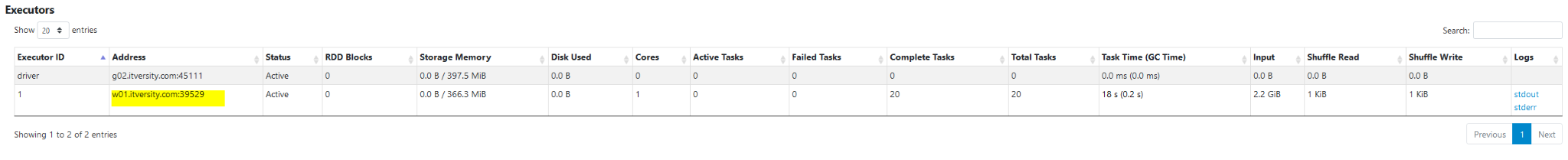
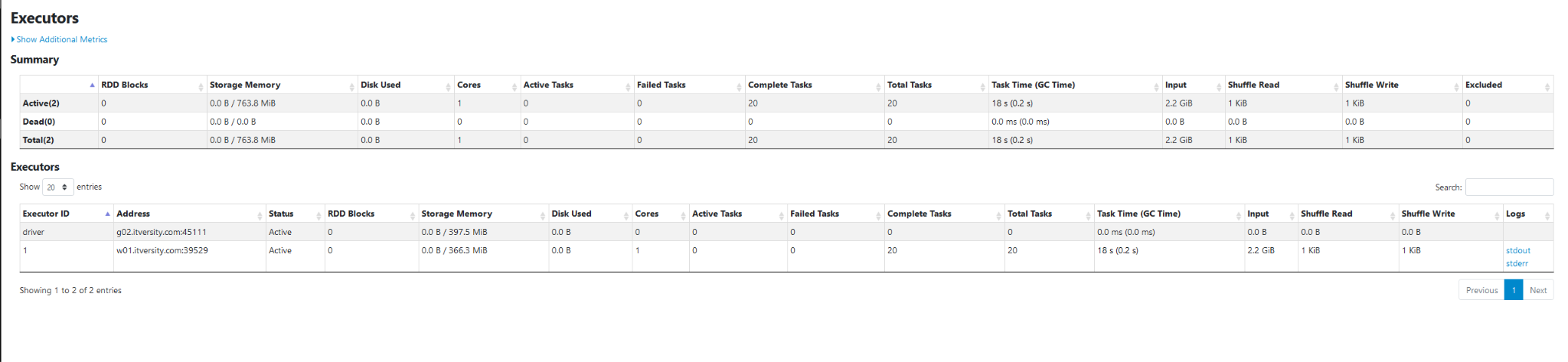
--num-executors 1\

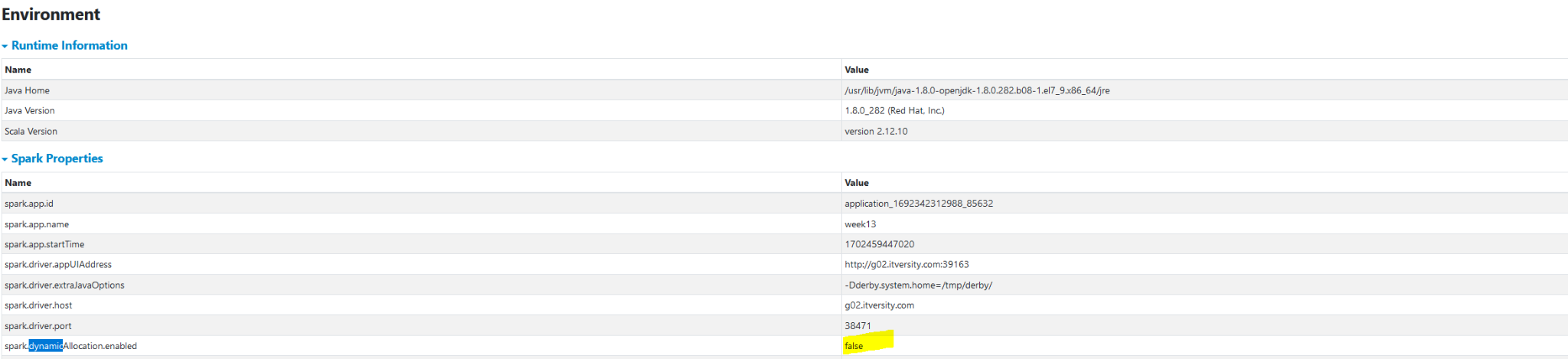
--executor-cores 1\

--executor-memory 1G\

--verbose\

week13.py

****

****

Disabled dynamic allocation and no of executors requested = 2

Spark3-submit\

--deploy-mode client\

**--conf spark.dynamicAllocation.enabled=false**\

--master yarn\

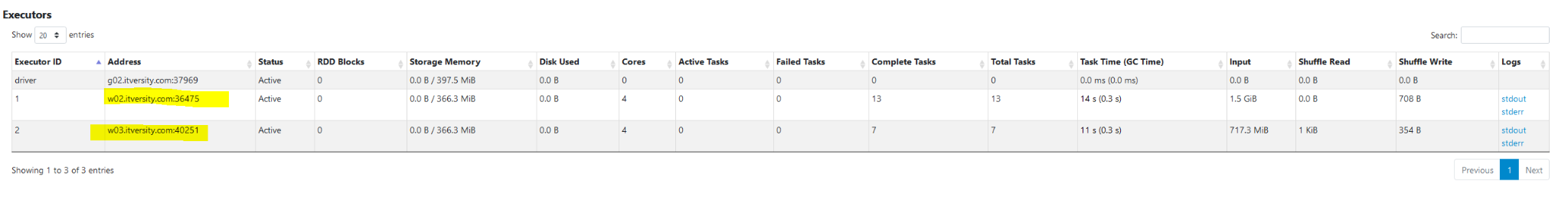
-**-num-executors 2\**

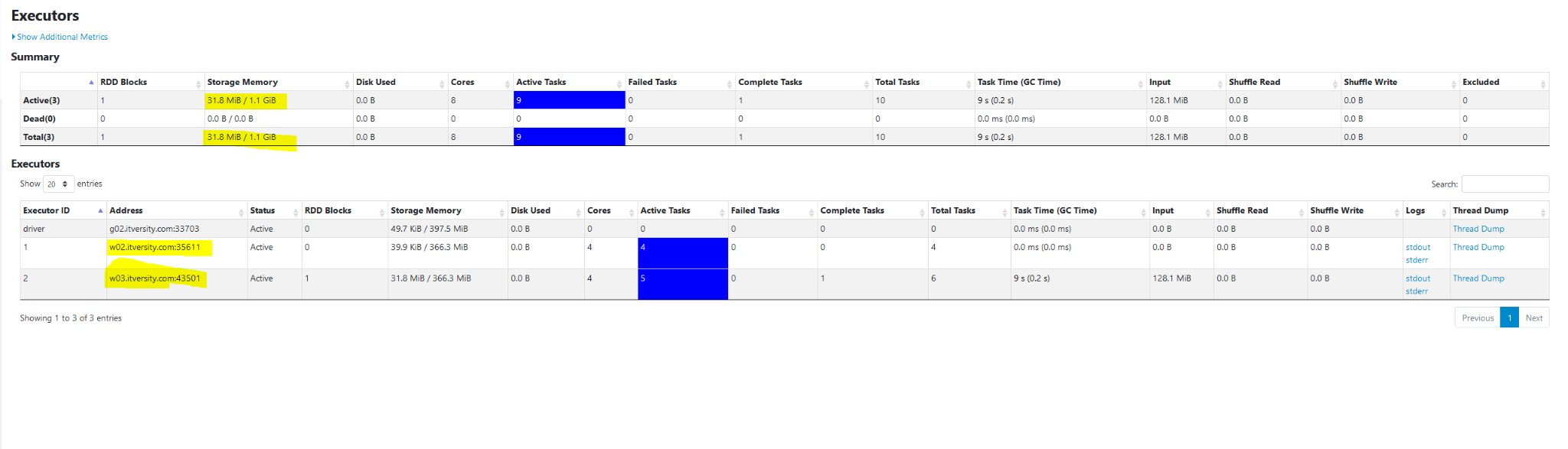
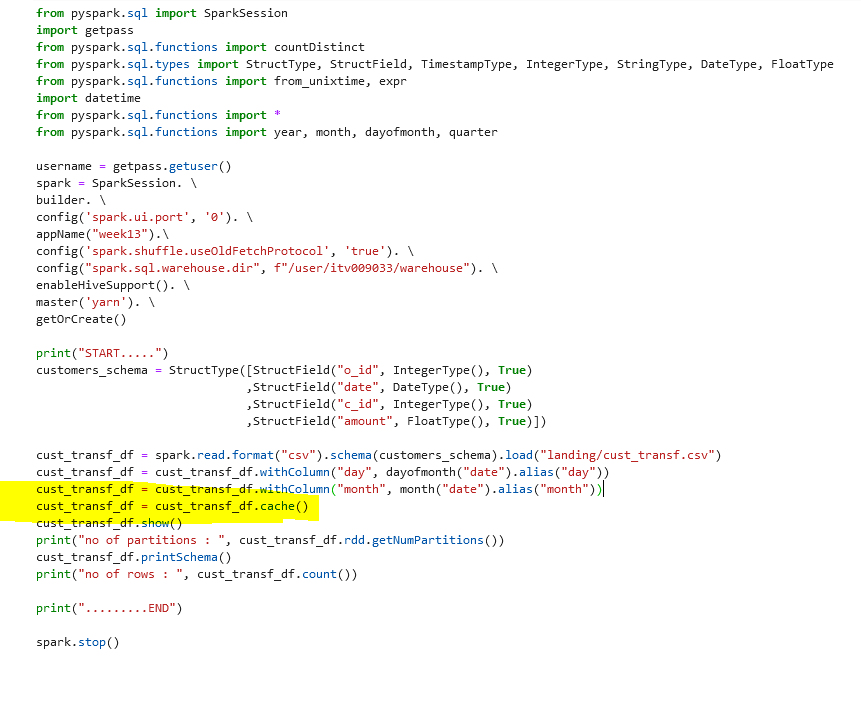
**--executor-cores 4\**

--executor-memory 1G\

--verbose\

week13.py

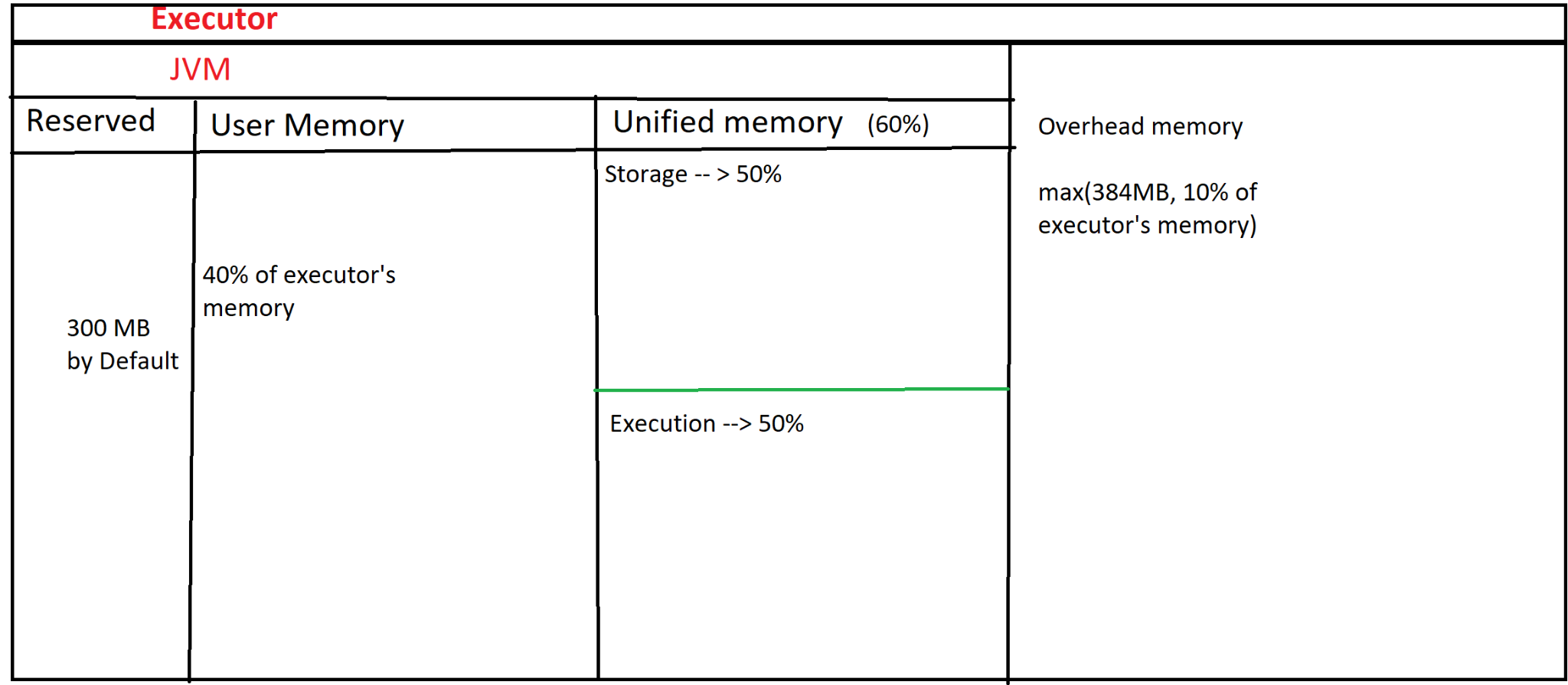
****

We tried caching to check how the storage is changing

**5. Provide a detailed explanation with diagrams for executor memory**

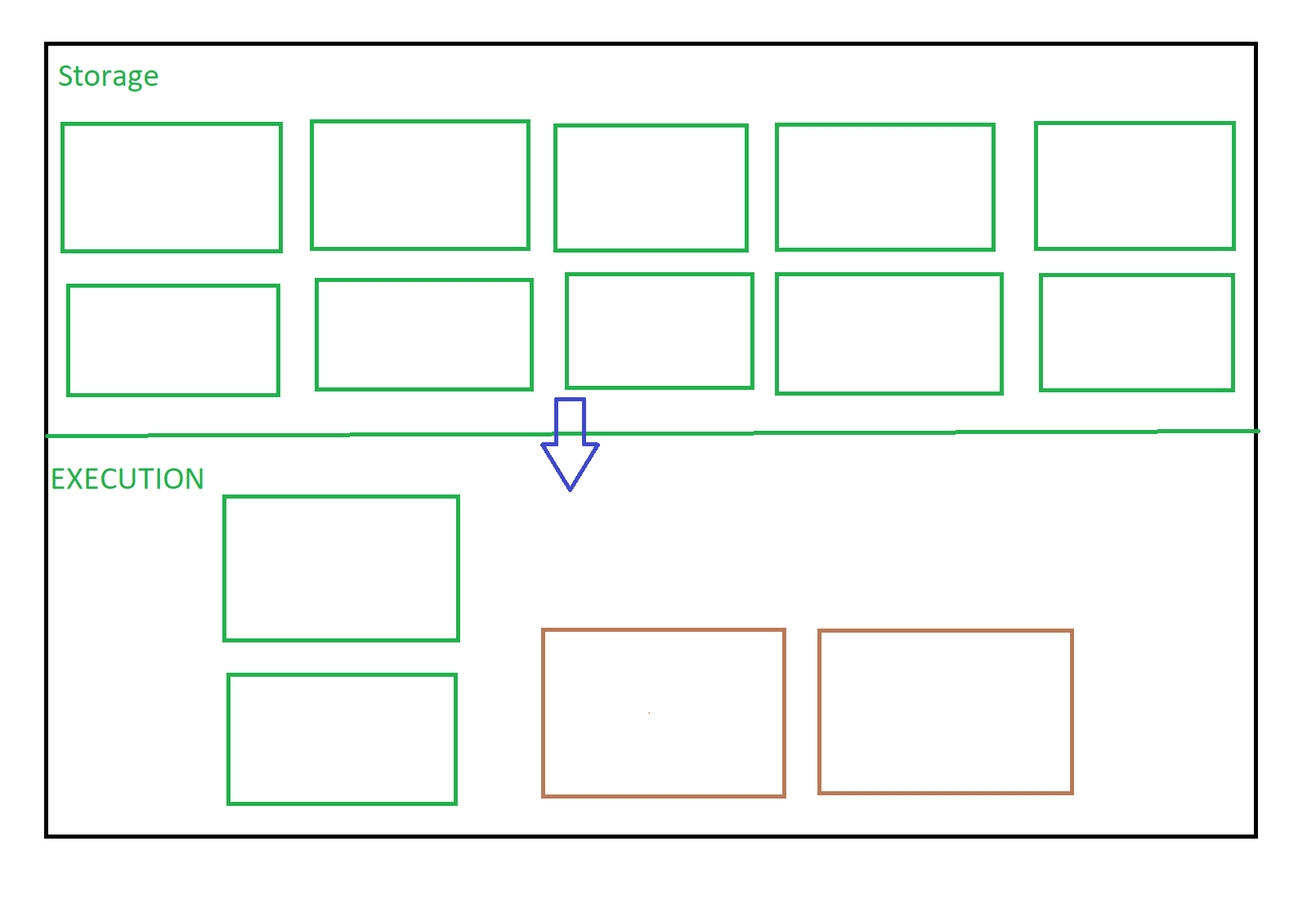
**distribution in the above example use-case considered**

→ In our case we requested 2 executors with 1GB memory each, so total memory requested = 2GB.

* **Overhead Memory :** Out of this 2GB, max(384MB, 10% of executor memory) would be reserved for overhead memory. So, 384MB is our overhead memory i.e., our application would actually request for 2048+384 = 2432 MB of memory. This memory is outside of our JVM
* **Reserved Memory:** 300 Mb is reserved for Pyspark to use. This is known a reserved memory. 2048 - 300 = 1748 MB we will be left with
* **Unified Memory :**  Out of the remaining memory left, 60% of it will go to to the unified memory that is 0.6\*1784 = 1048 MB. The unified memory is further divided into Storage memory and Execution Memory.  
    
  A. **Storage Memory** : 50% of our unified memory will go to storage i.e. 524 MB, In the storage memory, all our storage operations like caching happen.   
    
  B. **Execution Memory** : 50% of our unified memory will be given to the execution memory for performing tasks and transformations. In our case 524 MB.
* **User Memory :**  The remaining 40% is used as User memory. Around 700 MB will be our User memory. This memory is used for UDFs, custom data structures and RDDs. If we are not using any such things in our application, then we can take some user memory for execution too. 

**6. a. Storage Memory is full and it is extended to the execution**

**memory that is free**

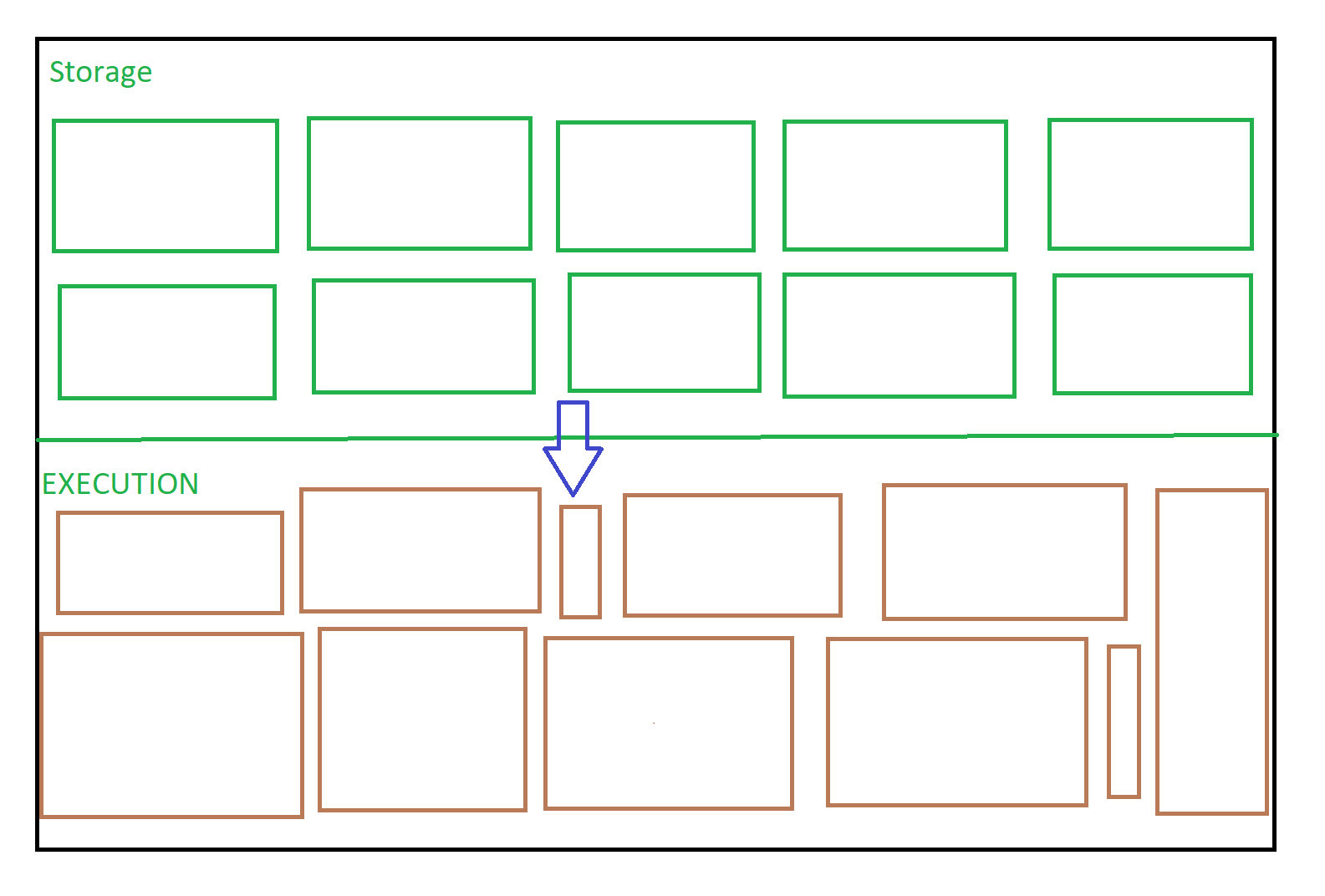
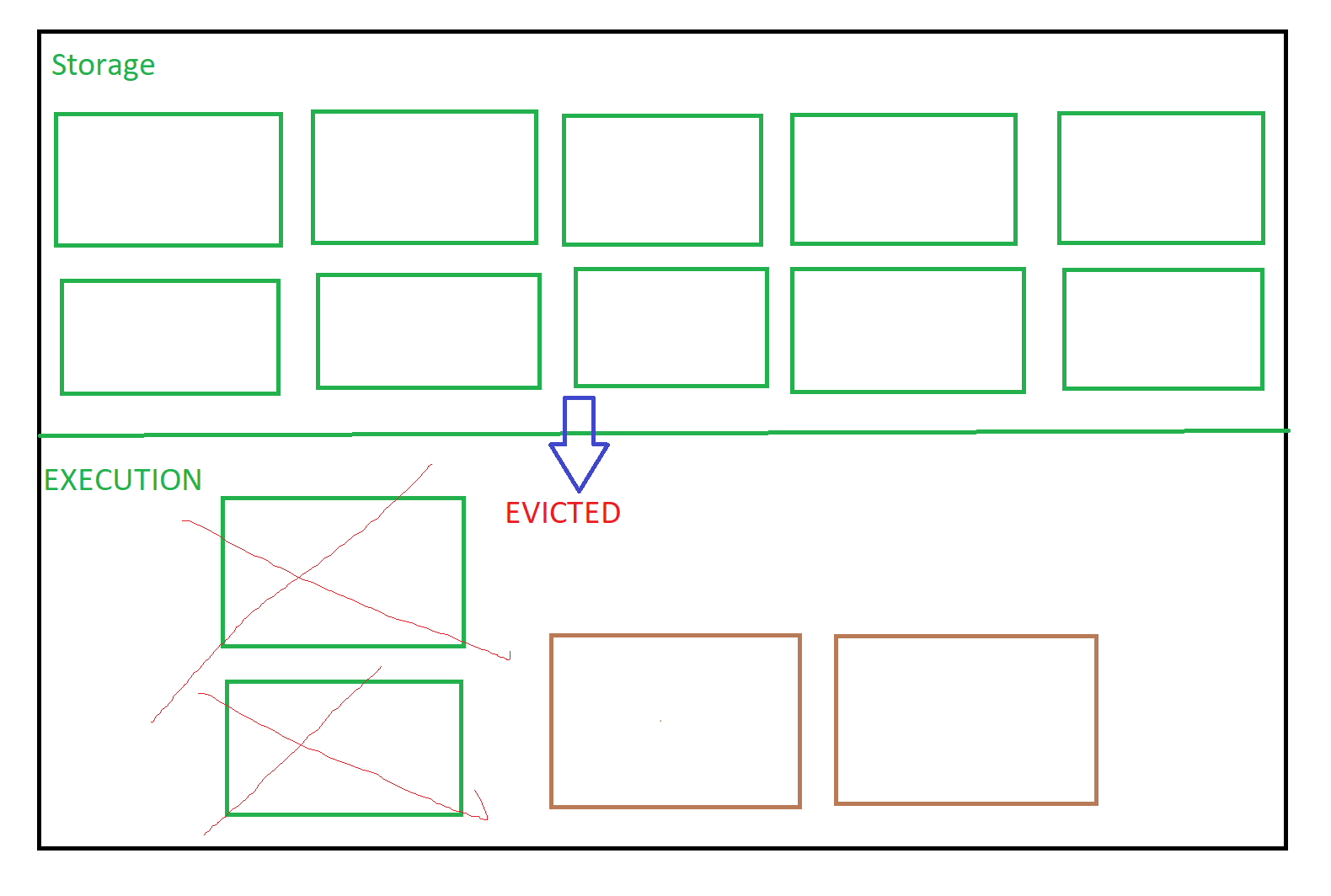
* When the storage memory is full, the storage can take some executor memory that is free
* In the below diagram, suppose only two jobs running and not taking up that much space in the executor memory, some of the storage will be shifted to the execution
* We must keep in mind that this can be done only to a certain threshold.

**6. b. Then a need arises for more execution memory due to more jobs**

**lined up for execution. What happens to the storage that has**

**extended into the execution memory space.**

* When such a thing happens, the execution can evict the storage in order to free Up memory for itself.
* We also need to keep in mind that Storage cannot evict execution but execution can evict storage



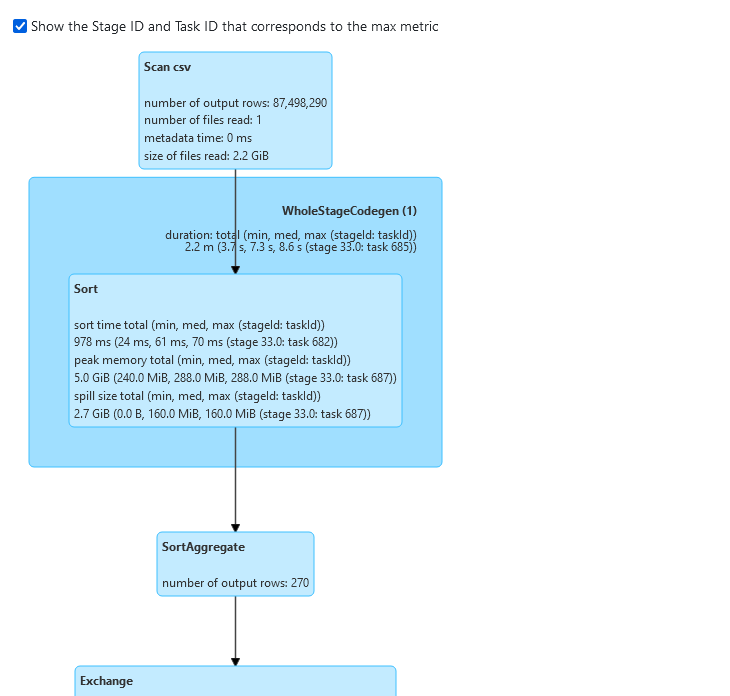
**7. Consider an example use-case on the dataset chosen previously to**

**demonstrate when the spark engine chooses to use Hash / Sort**

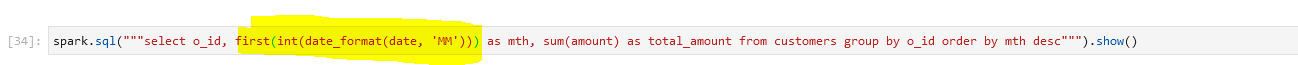
**Aggregation**

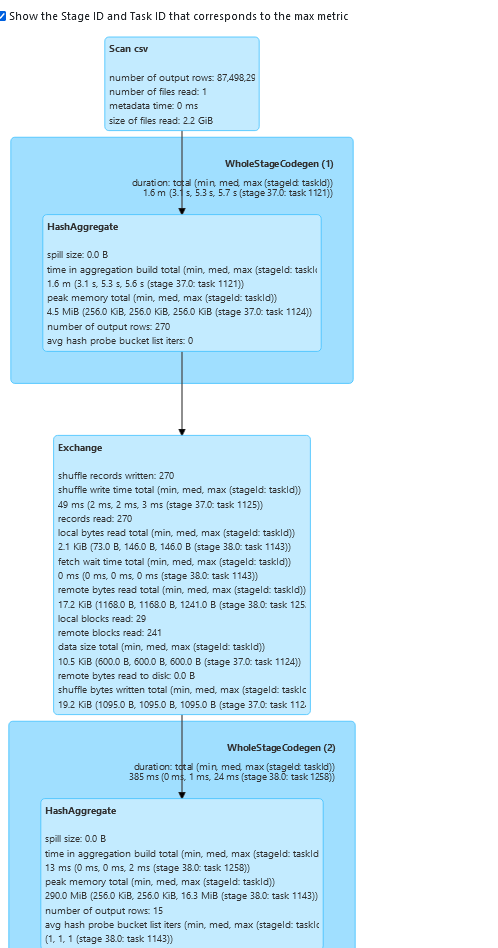
* Here we are seeing the sort aggregate being called, the reason being that the column we just created named mth, it is a string. While sorting, we cannot sort it by the string as strings are immutable.

****

****

* Hash aggregate is used when the column is an int. Hash aggregates are faster as in the backend a hashtable is used to store the k,v pairs associated with them



****

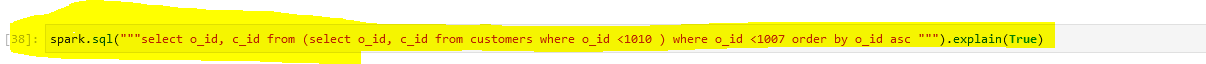
**8. Demonstrate the following optimization in Spark’s logical and physical**

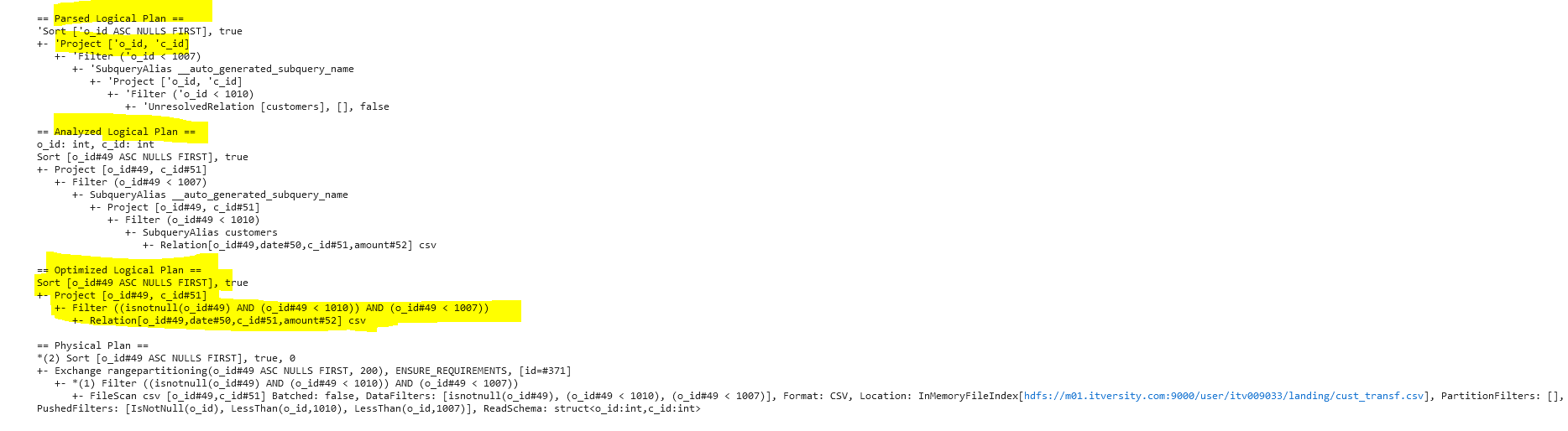
**execution plan**

**a. Predicate Pushdown.**

**b. Merging of multiple projections into one.**

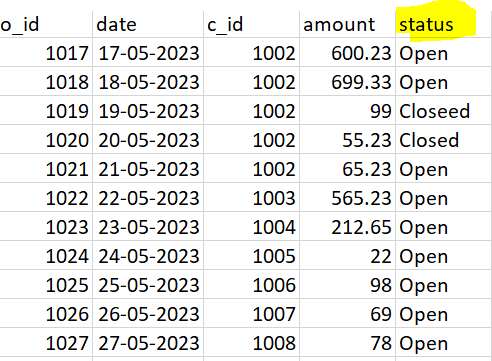
**c. Merging of multiple filters into one.**

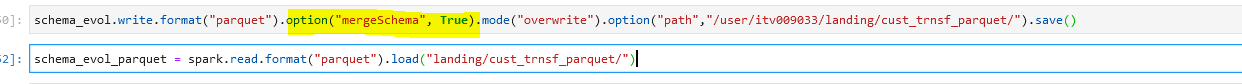
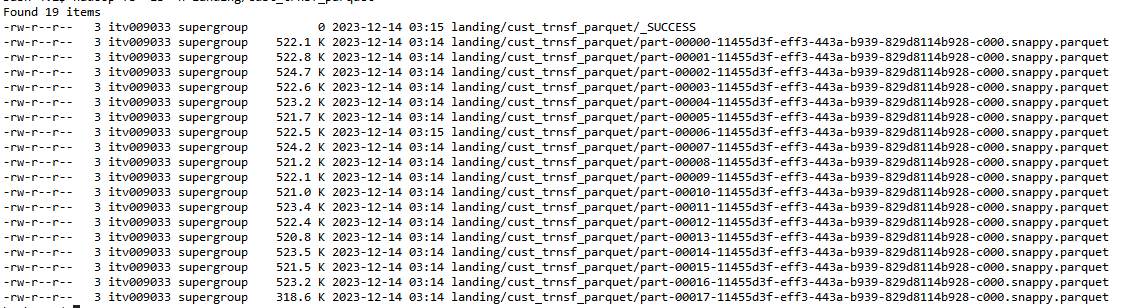
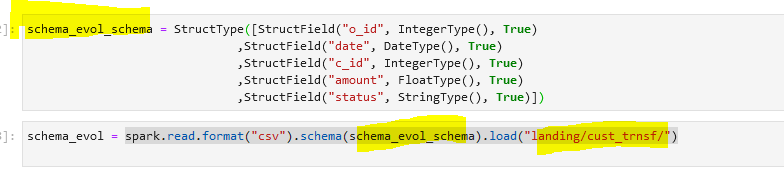
→ 



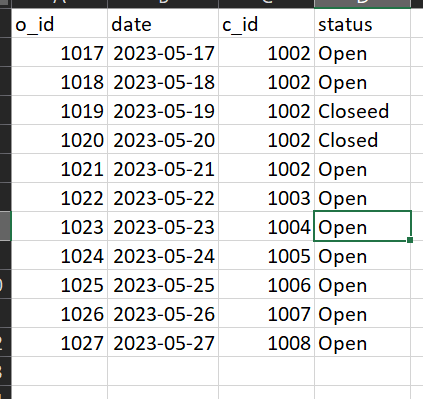
**9. Demonstrate Schema Evolution on the dataset considered by**

**a. Adding a new column :**

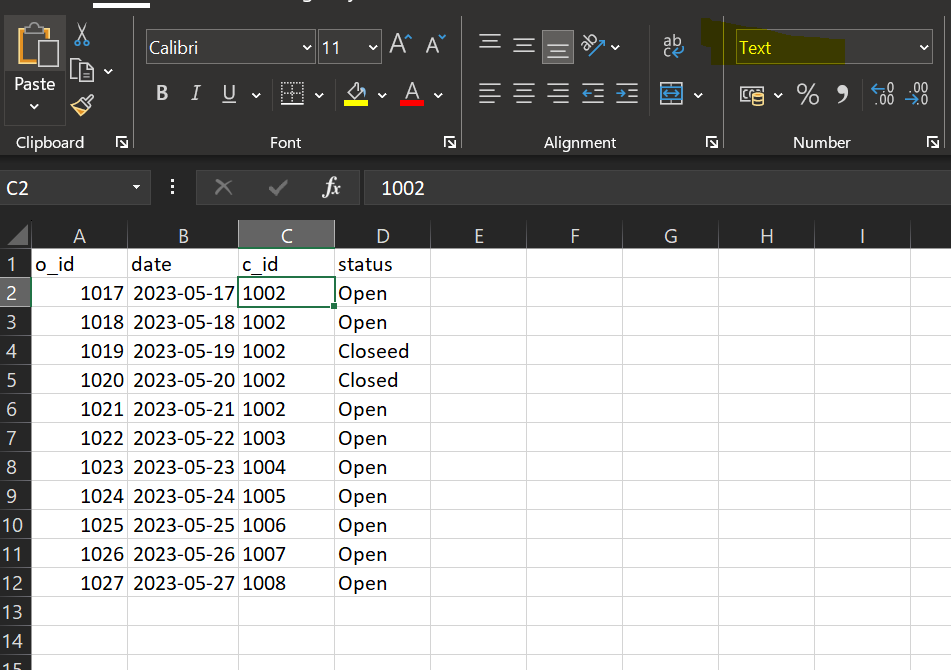
****

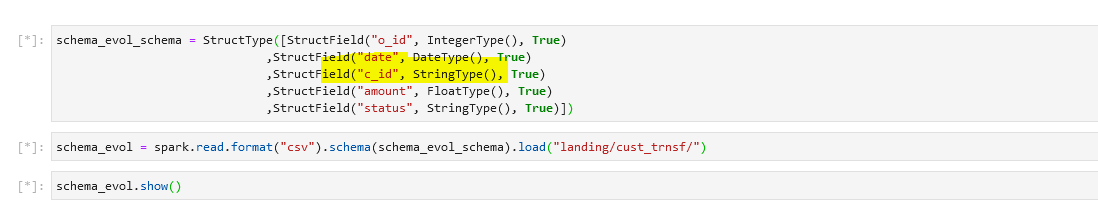
****

**b. Dropping a column and c. Changing the datatype →**  Dropping the amount column

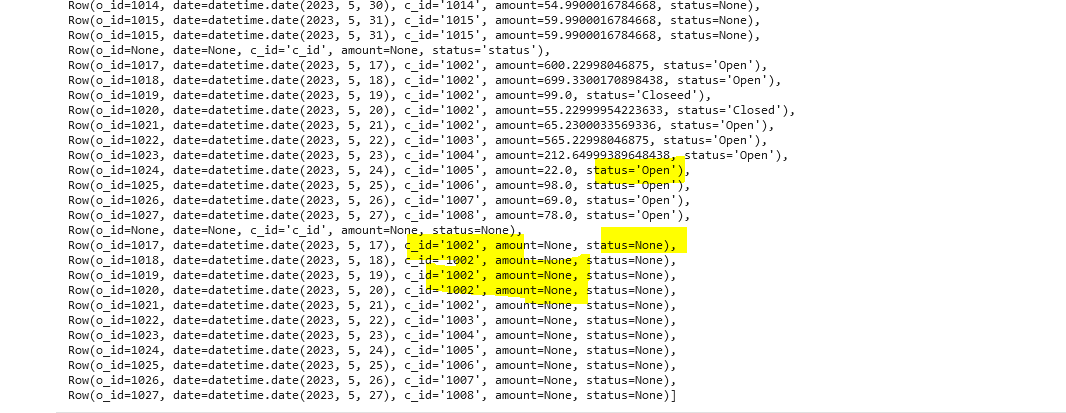
****

Changing the data type

****

****

****

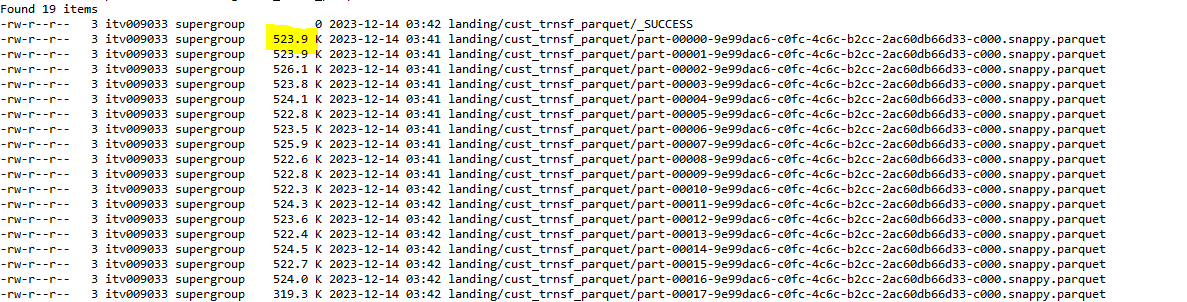
****

**10. Research and explore the different file formats furthermore. Depict**

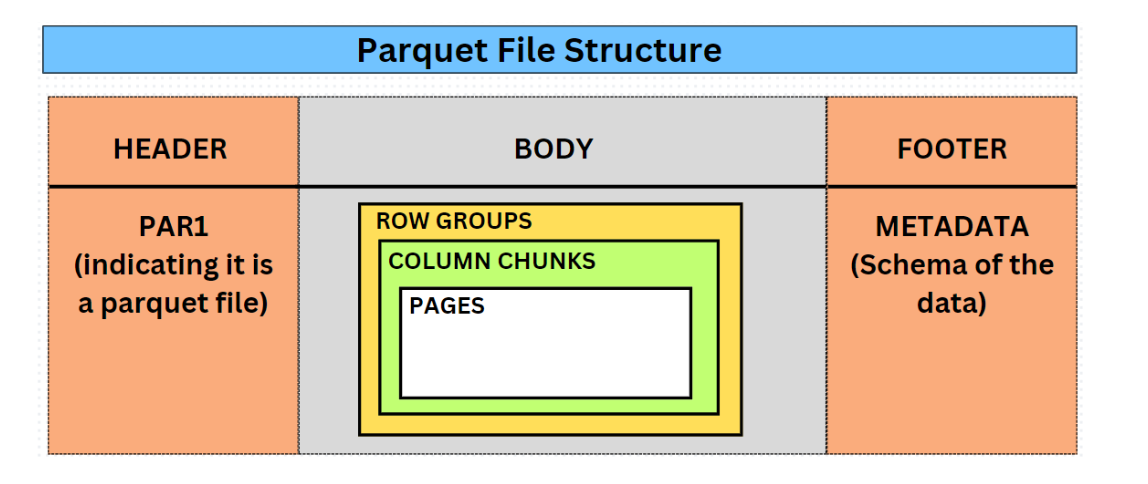
**your inferences with relevant diagrams and explanations. (Parquet,**

**ORC, AVRO)**

* **Parquet :** Around 500 Kb per partition



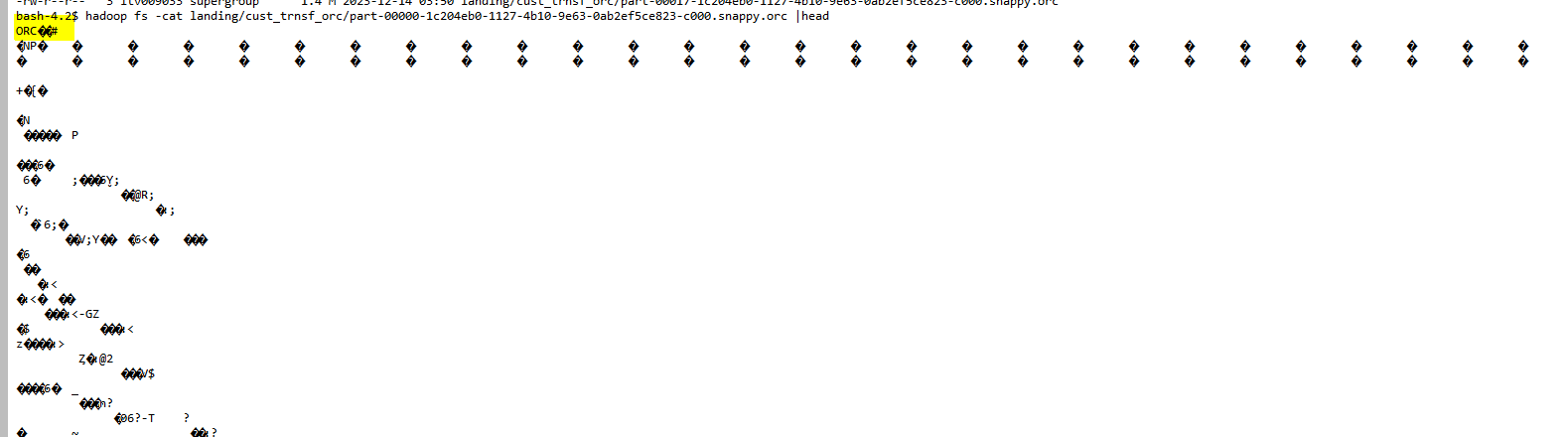
→ parquet is by default used with spark. It also uses snappy compression by default.

→ Parquet is columnar, i.e it is good in reading and moderate in writres. It has 3 parts, the header → says that it is a parquet “PAR”, body and footer. 

→ When the data is read, it looks in the row groups and it then searches in the column chunks by looking at the metadata and gives us the result without scanning the whole file.

* **ORC:**  ORC stands for optimised columnar storage.

→ It works great with Hive



→ It is also very good while reading a subset of data as by default it uses predicate push down by just looking in to the metadata and skipping the rows and columns not required

* **AVRO** : It is a row based file format, but it supports schema evolution and it is also better than csv etc as when compressed it becomes splittable by default.

**11. Apply the different generalised compression techniques explained in**

**the course on the example datasets and illustrate the differences**

**Noticed.**

* Lzo orc : 