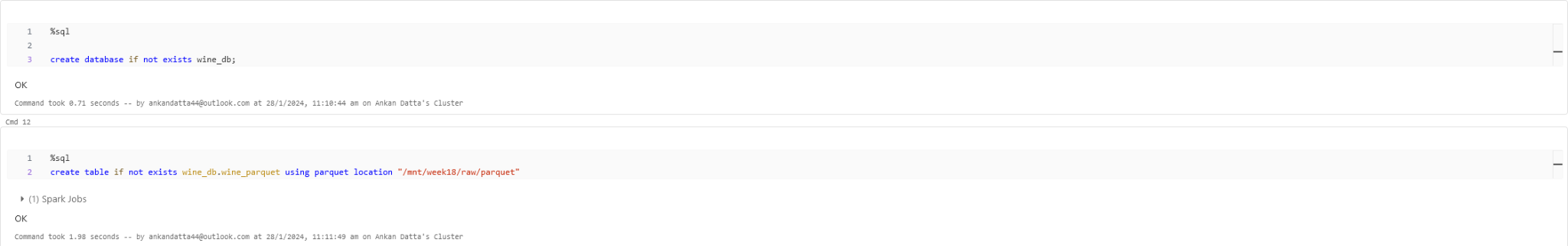
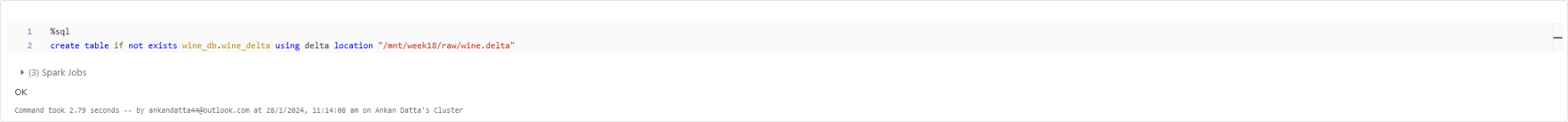
**2. Create Spark tables**

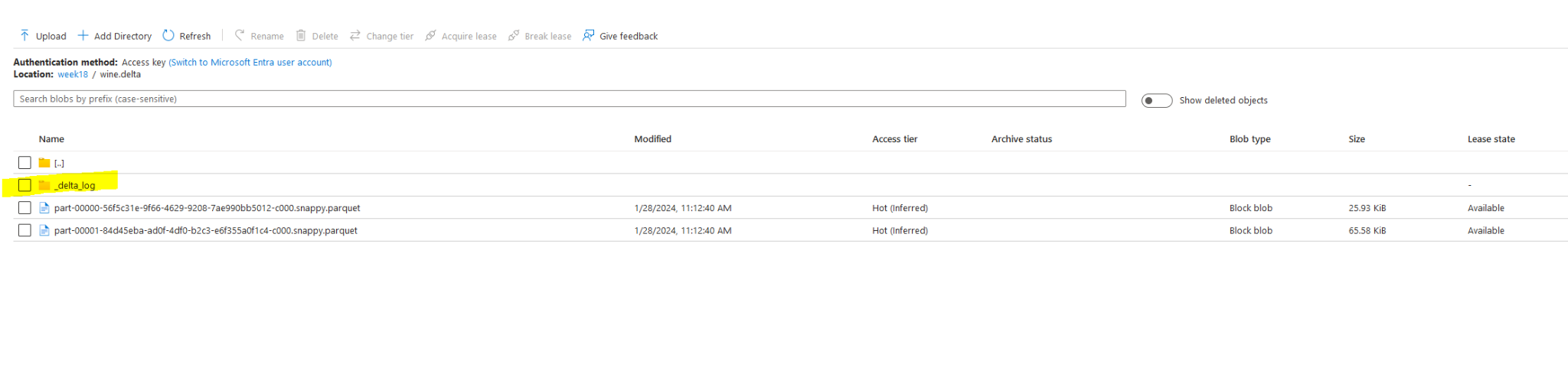
**a. In Parquet format : **

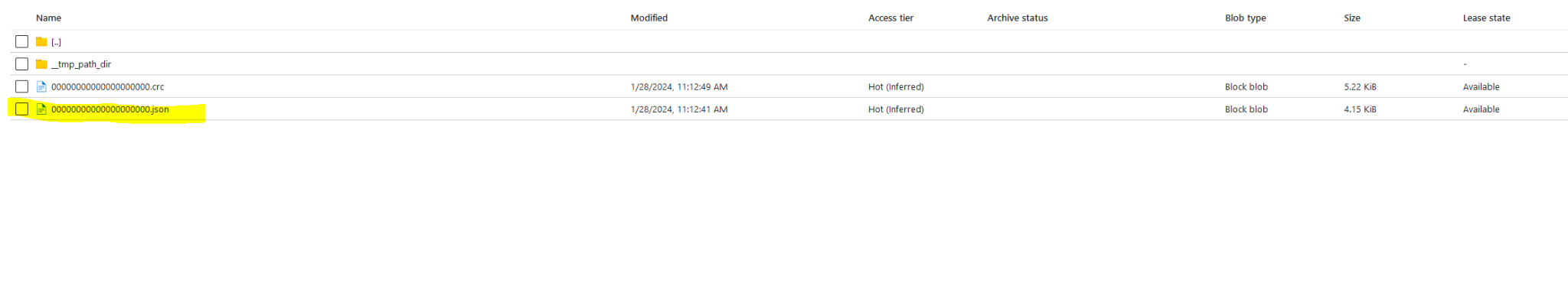
**b. In Delta Format : **

**Present the differences between the two tables with relevant screenshots and**

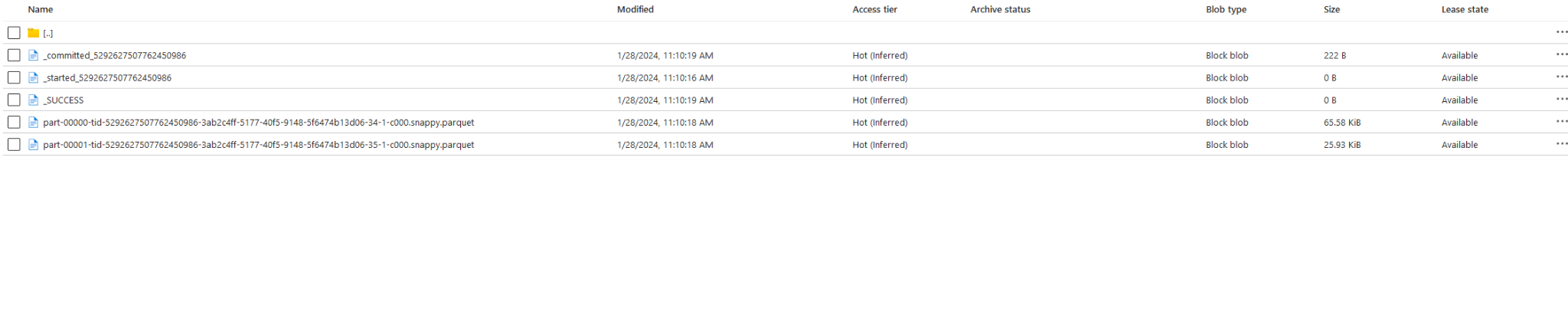
**Explanations.**

* In the delta file format, a delta log folder is created which holds the transactional data. This folder has a json file init that gives us statistical analysis of each write operation,. This helps in keeping all the ACID properties in tact while writing the files and also helps us with faster speeds while reading the data.





****

* This is how the json file looks like. It has all the metadata that is required which can help us in retaining the ACID properties.
* The parquet format gives us the files only in the parquet file format 

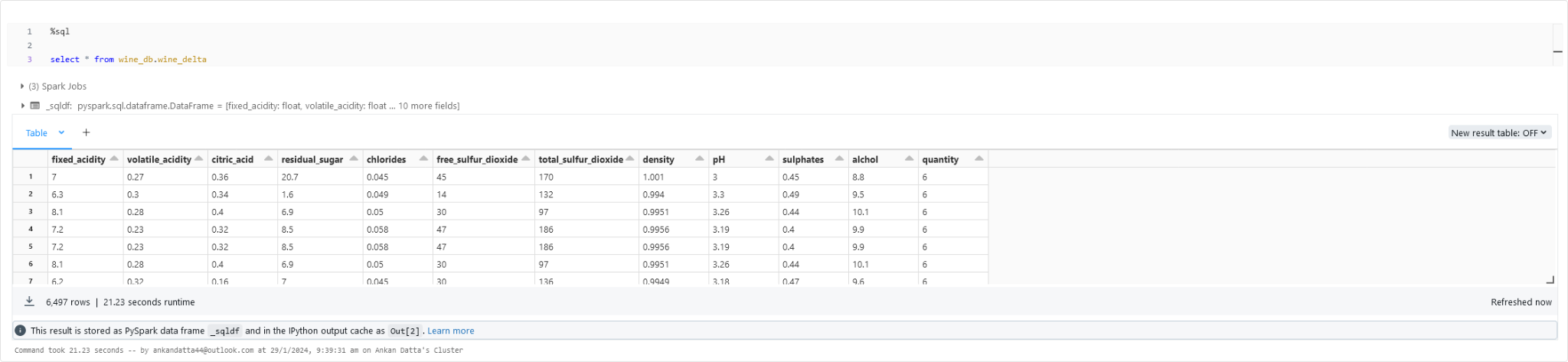
**3. Illustrate the benefits of Delta Caching with the help of an example Dataset**

**considered.**

**a. Enable delta caching by setting the property**

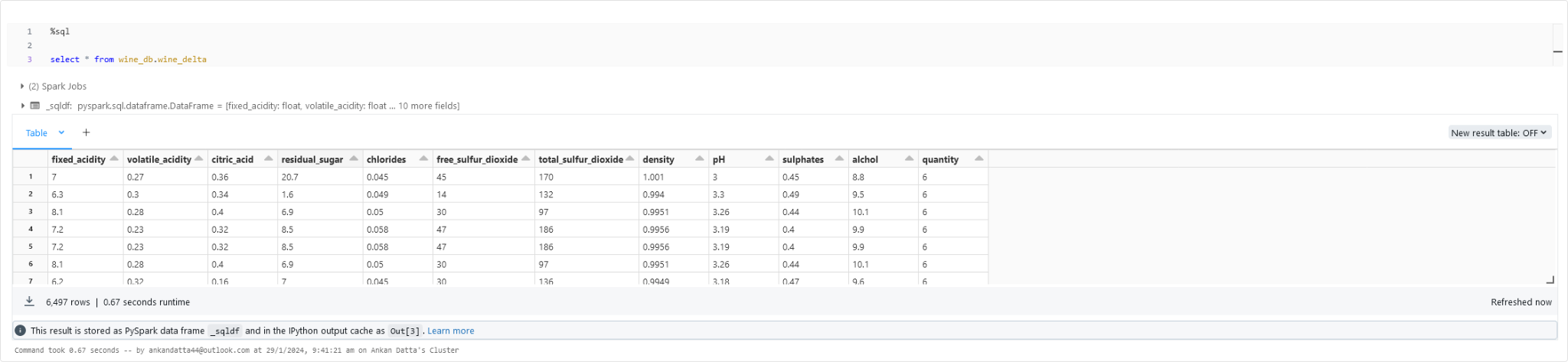
**b. Enable delta caching manually through command**

* Before caching : 

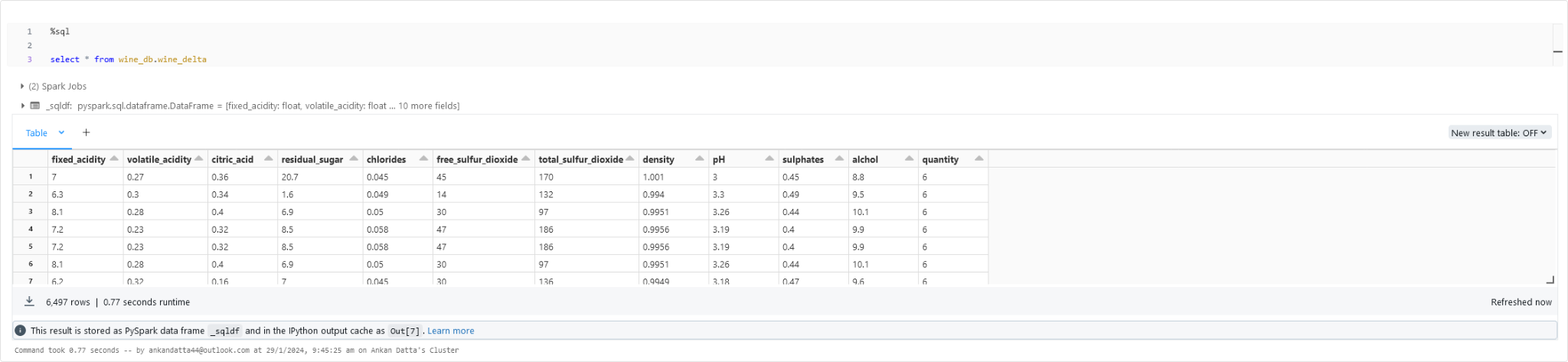
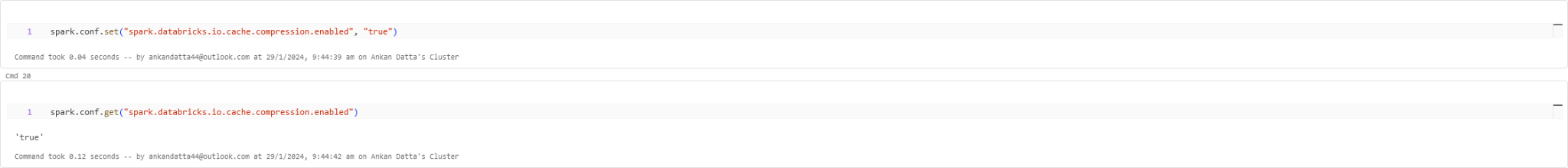


→ 21.23 Seconds taken to run this code

* Caching with the cache command : 

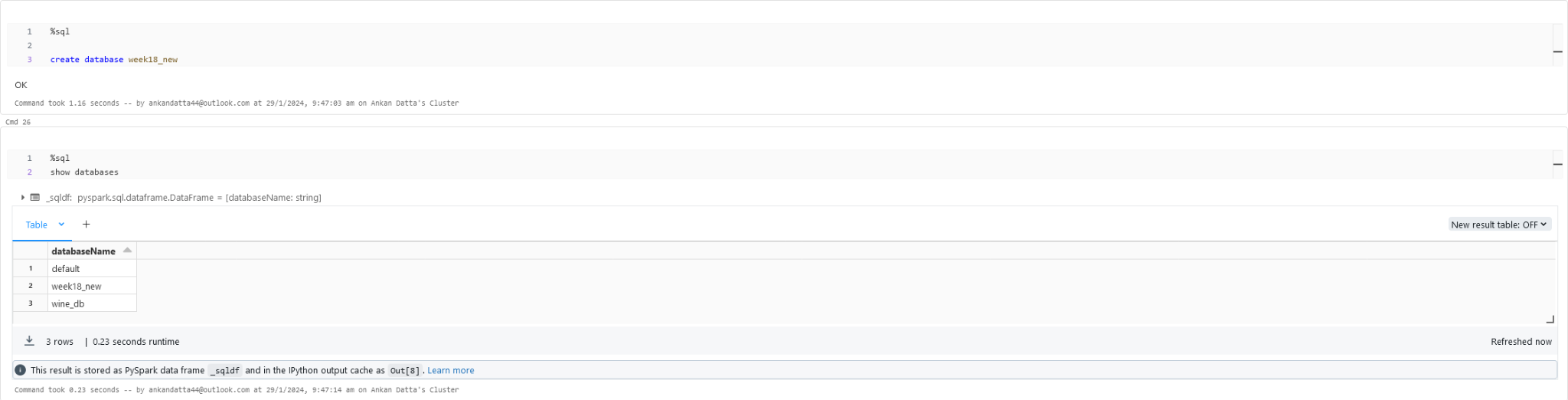


→ It took only 0.67 seconds to run

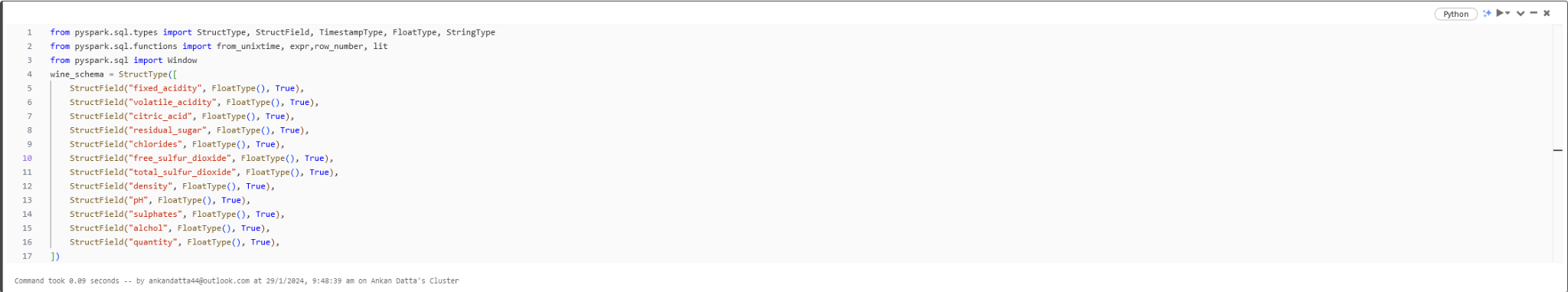
* Using Databricks’s cache : 

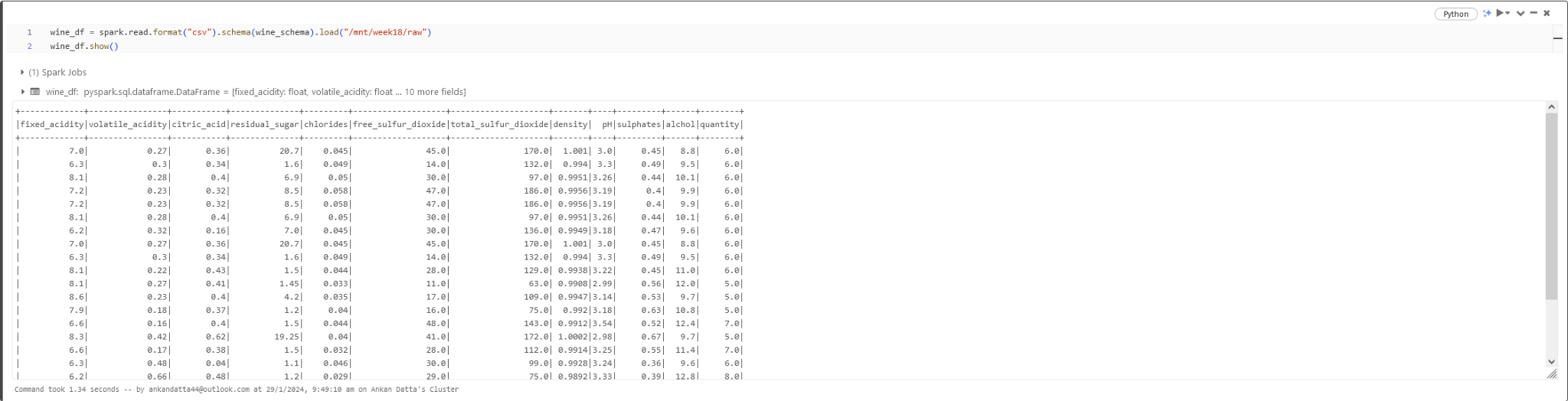
→ This query took around 0.77 seconds to run

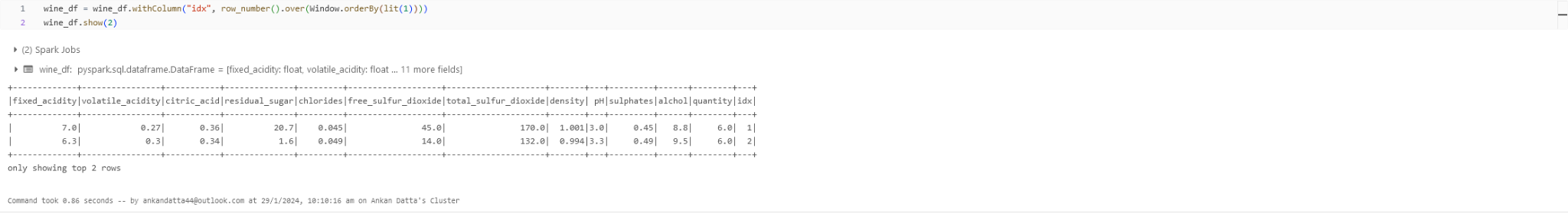
**4. Demonstrate the commonly occurring Small File Problem**

**a. Create a Database **

**b. Load any sample csv file from the DBFS file system.**

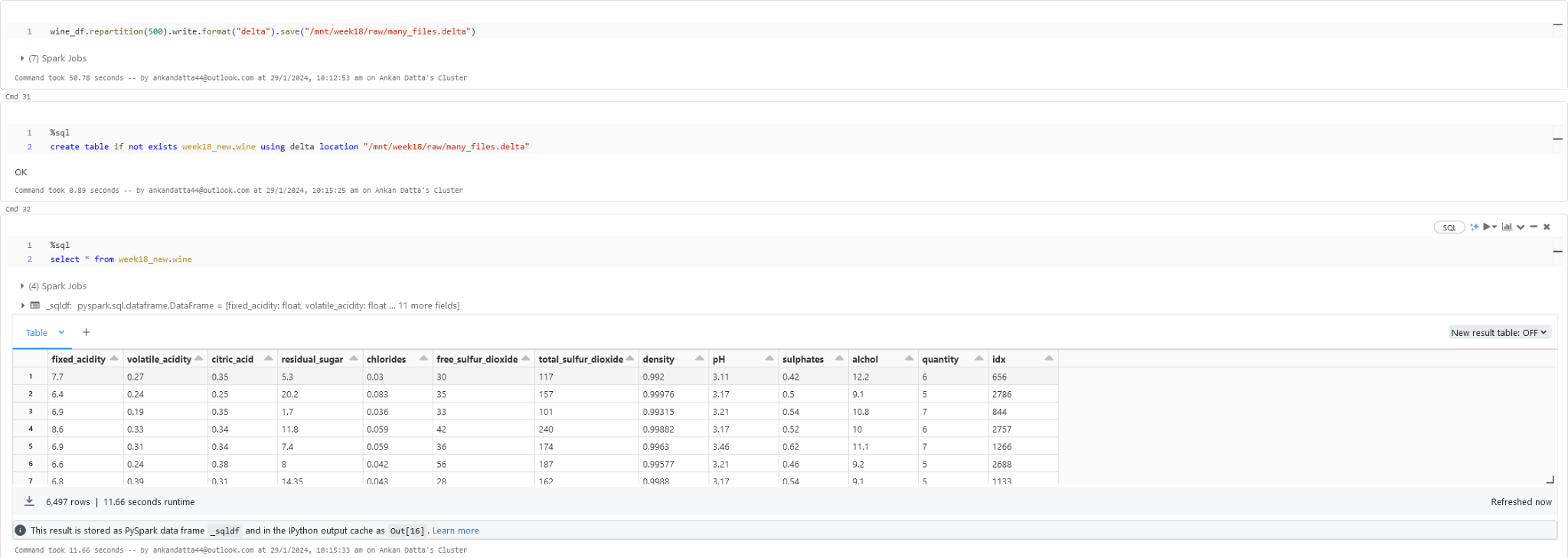
****

****

Adding a id to the dataframe : 

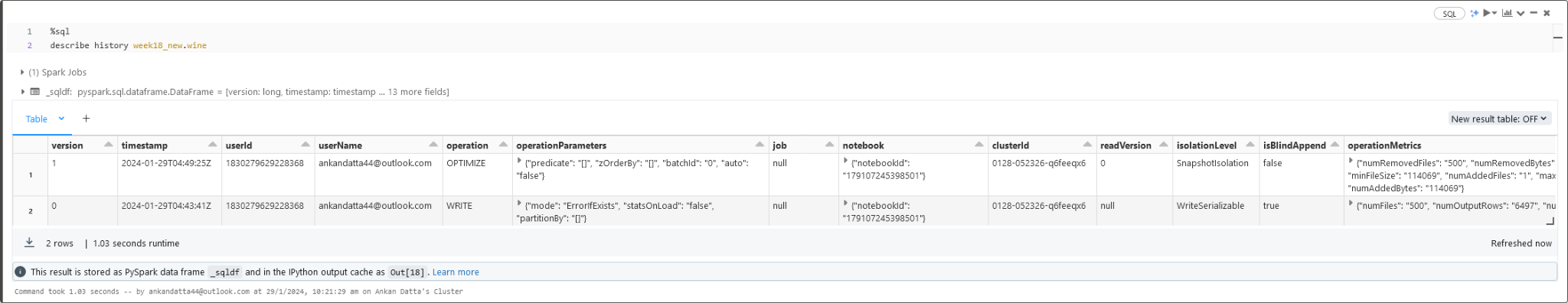
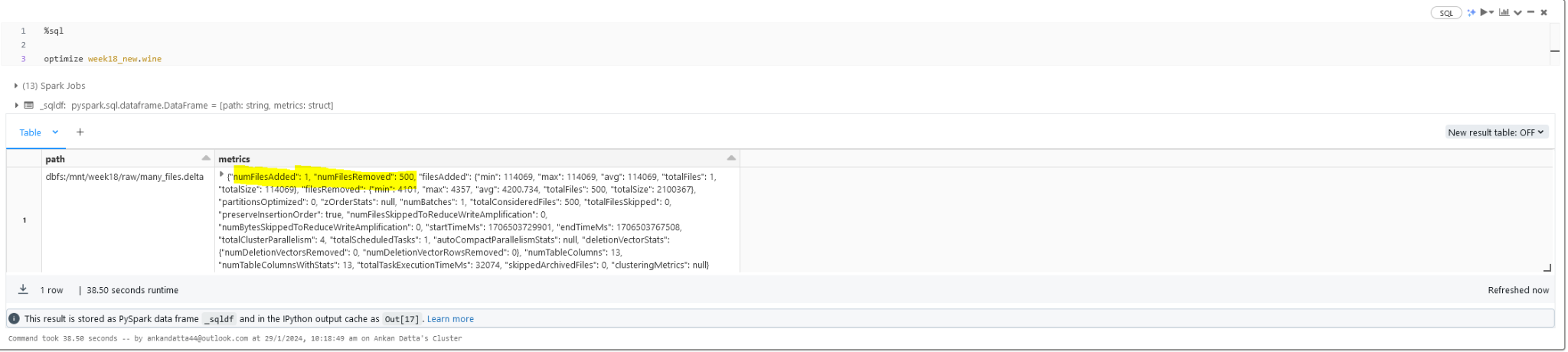
**c. Create a Delta Table with a large number of files (~500 files) for each**

**partition using the repartition option.**

****

**5. Demonstrate the Compaction / Bin-Packing technique of overcoming the**

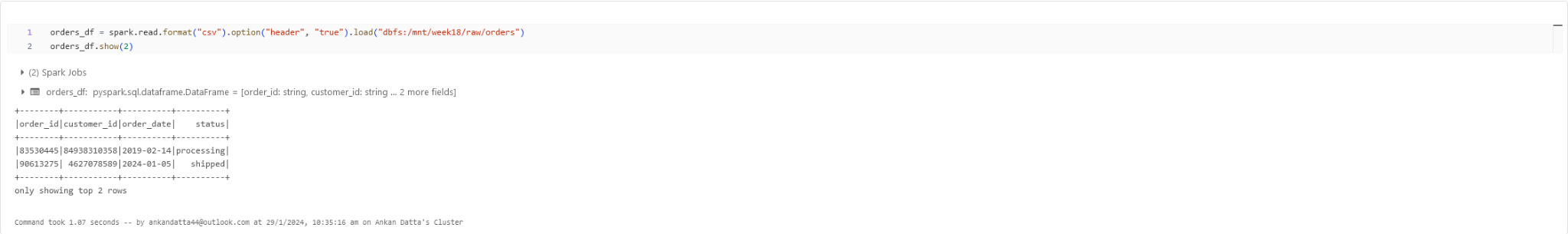
**small file problem explained previously**

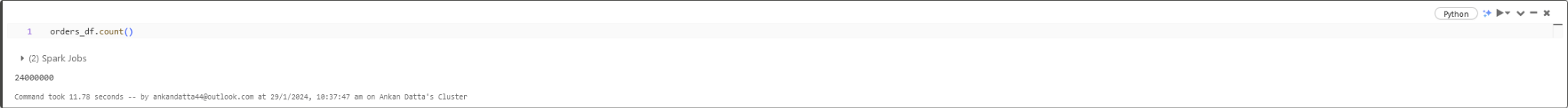
****

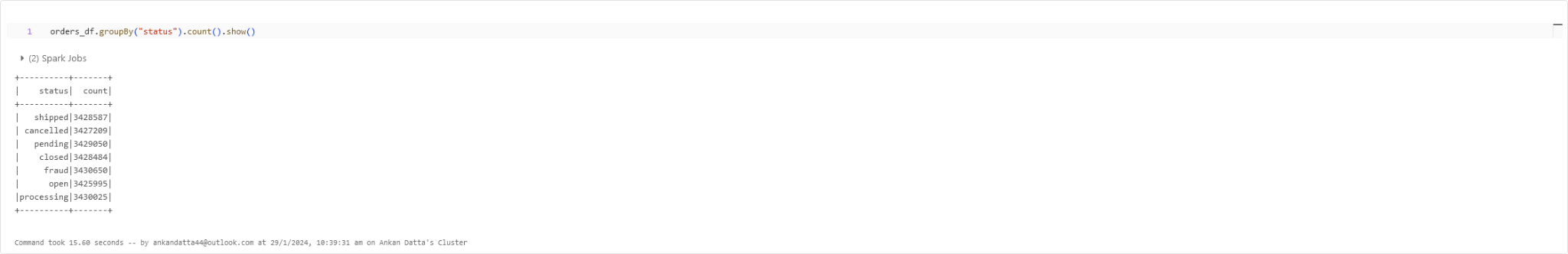
**6. Explain with an example how data-skipping is achieved in Partitioning and**

**Bucketing.**

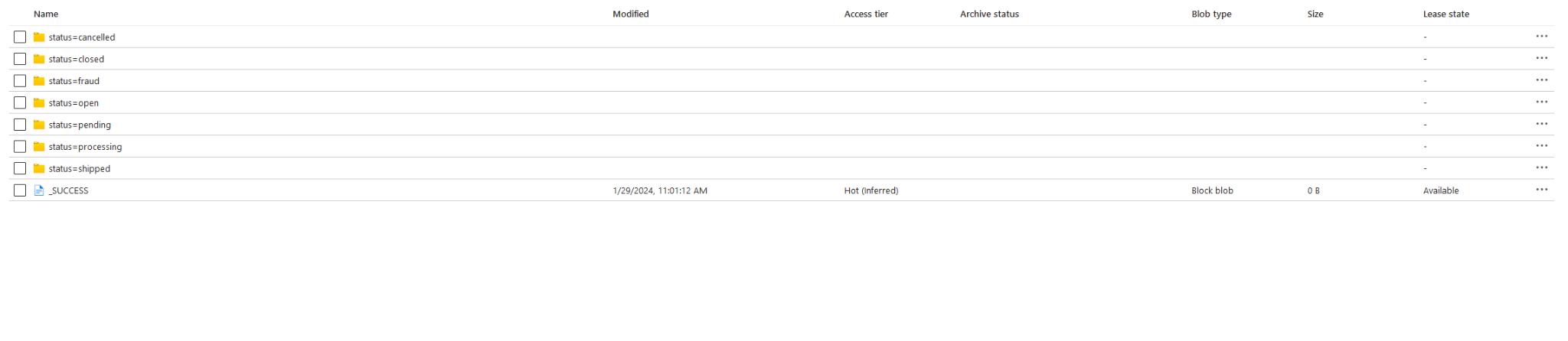
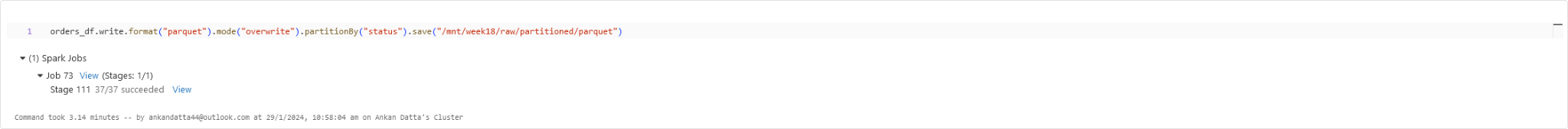
→ For this example we will be taking the orders data set as the dataset I have used has no columns from where we can partitionBy



→ There are 2,40,00,000 records in this file

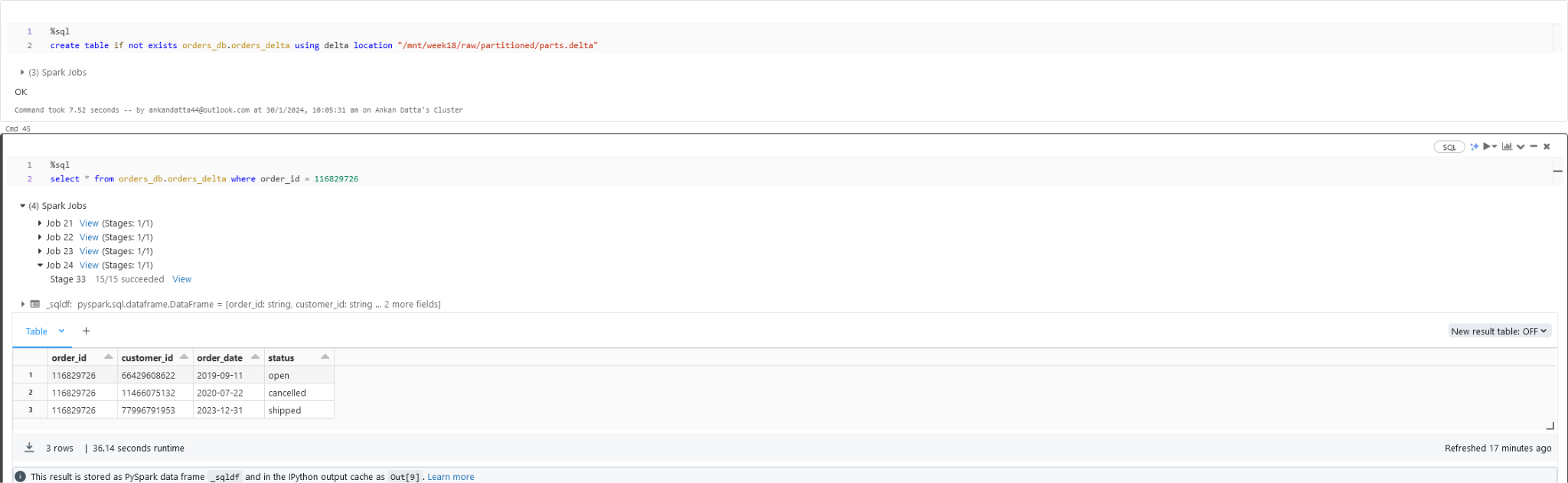
→ In the above case, say we want to see how many orders were closed, then we would have to scan all the 2.4 cr records and then find the closed orders.

→ In this case if we do a partition on the order status then we can easily skip the other records and only focus on the closed records which will save us a lot of time as it will skip the records that are not closed.

→ 

Here we can see that we are achieving data skipping by using partitioning

→ To demonstrate bucketing, we have created a delta table on top of the order’s dataset and ran a query to check how much time it takes .



%sql

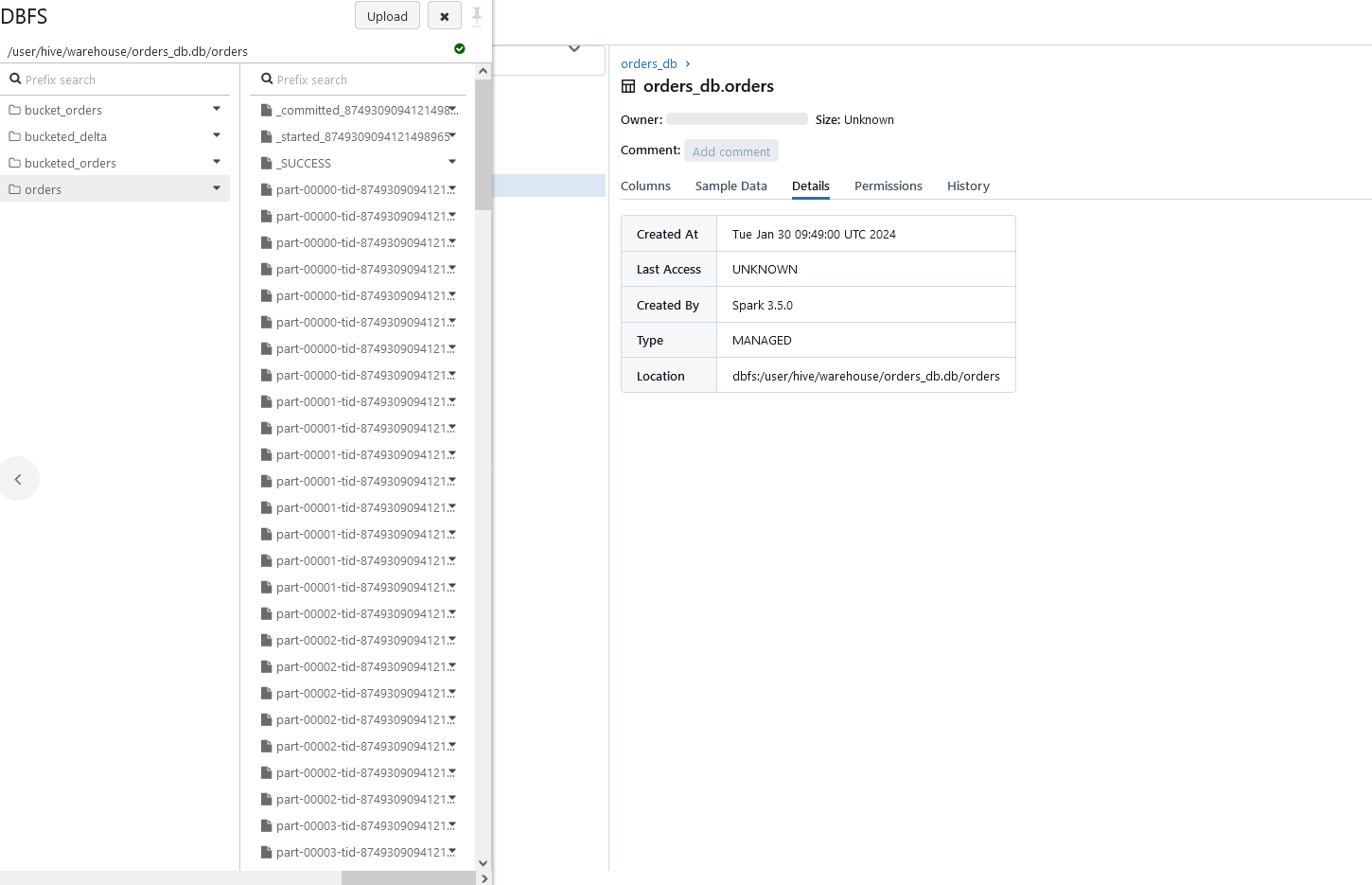
select \* from orders\_db.orders\_delta where order\_id = 116829726

It took around 36 seconds to filter the data.

→ Now we did bucketing and ran the same query to check if data skipping was happening on top of the buckets created while increasing the speed of the query.

We can see that the same query took around 28 seconds to run as compared to the previous one

→ when we reduced the bucket size to 8, we got even faster results.



**7. Use the Z-Ordering approach to achieve Data-skipping optimization on the**

**dataset considered in the previous steps. (Provide screenshots of executed**

**query along with explanations for query performance with Z-Ordering and**

**without Z-Ordering)**

→ Z ordering gives us data skipping in the case when we want to solve - small file - big file problem by using the OPTIMISE command or in other words using the bit compaction/ bit packing technique.

→ Suppose that we have an orders table with columns such as order\_id, customer\_id, status etc and we want to filter out certain order\_ids. Then while bin packing, the ids will be distributed haphazardly in files.

→ Let us assume that file 1 has order ids of (1,4,3), file 2 (4,5,6,8) and file 3 (2,3,1,5,9,10,11)

→ The meta data for the files will be something like :

* File1 → min = 1, max = 4
* File 2 → min = 4, max = 8
* File 3 → min = 2 max = 11

→ So, if we want to filter for the order id 4, it has to scan all the 3 files as 4 lies between the min and max of all these files.

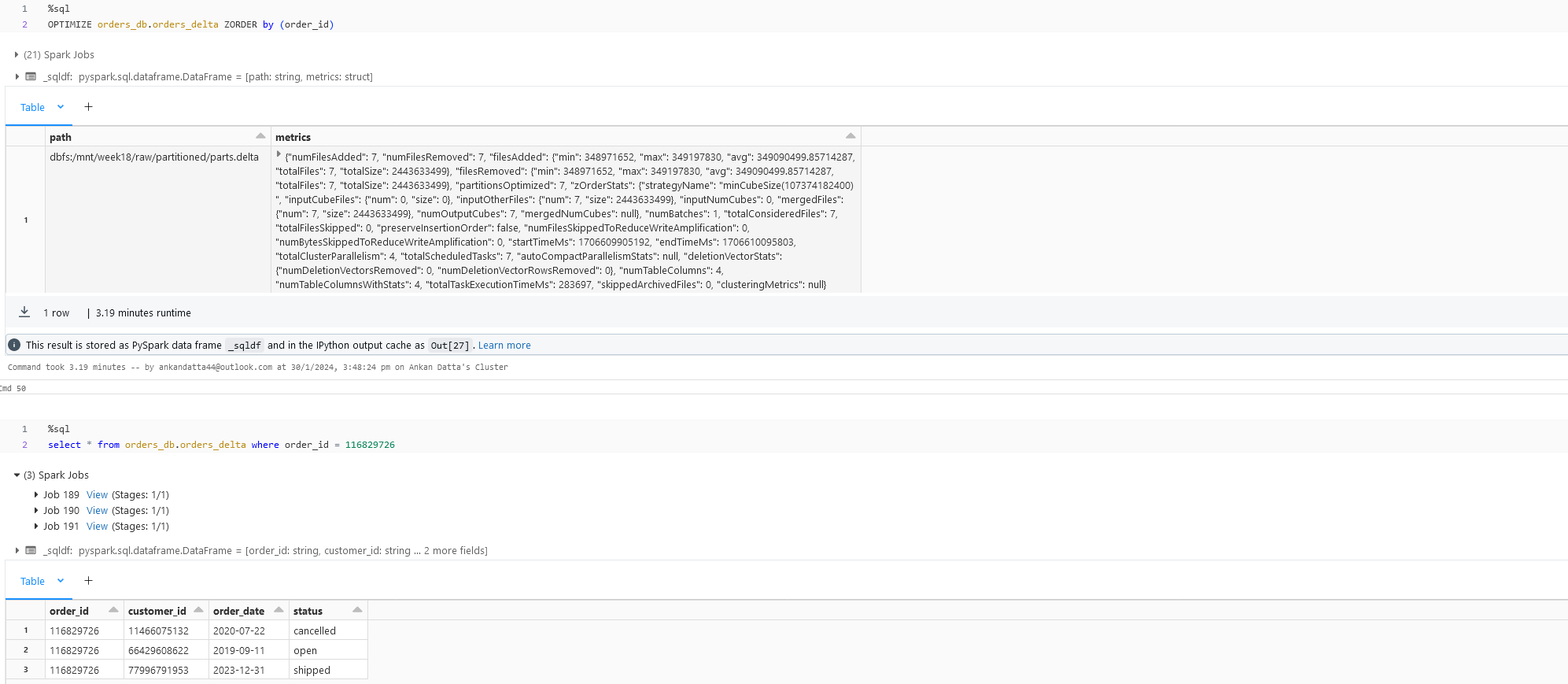
→ No data skipping was happening here, thus to skip data we use z ordering.

→ The newly structured files would look like :

* File 1 → o\_id : 1 to 4, File 2 → o\_id : 5 to 8 and file 3 → o\_id : 9 to 11

→ Thus here we will see them scanning only one file instead of 3

→ Z-ordering is not idempotent but aims to be an incremental operation. **The time it takes for Z-ordering is not guaranteed to reduce over multiple runs**. However, if no new data was added to a partition that was just Z-ordered, another Z-ordering of that partition will not have any effect



**8. When is the VACUUM command used, its benefits and drawbacks?**

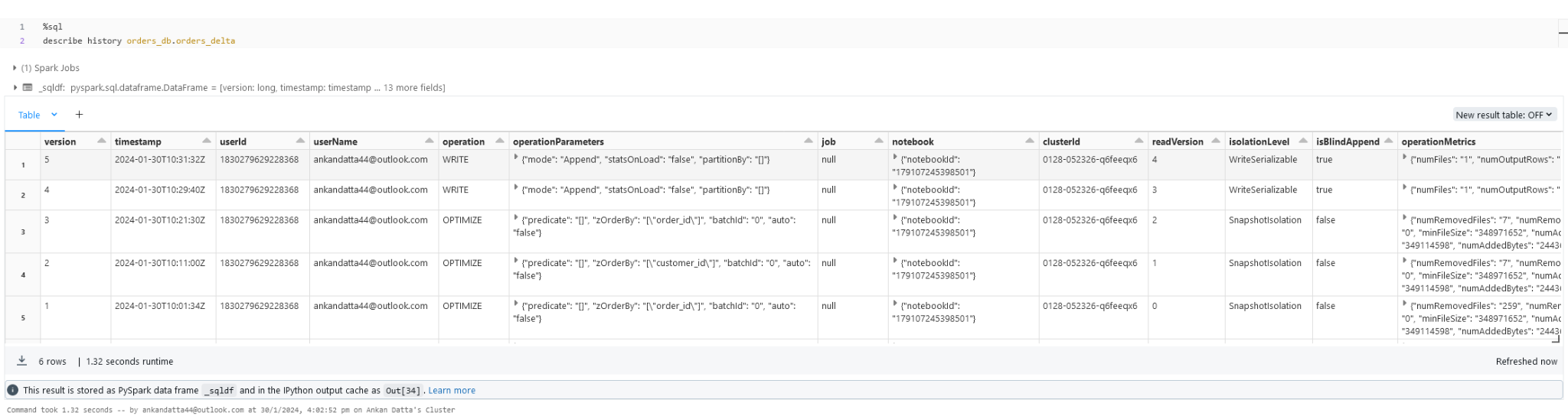
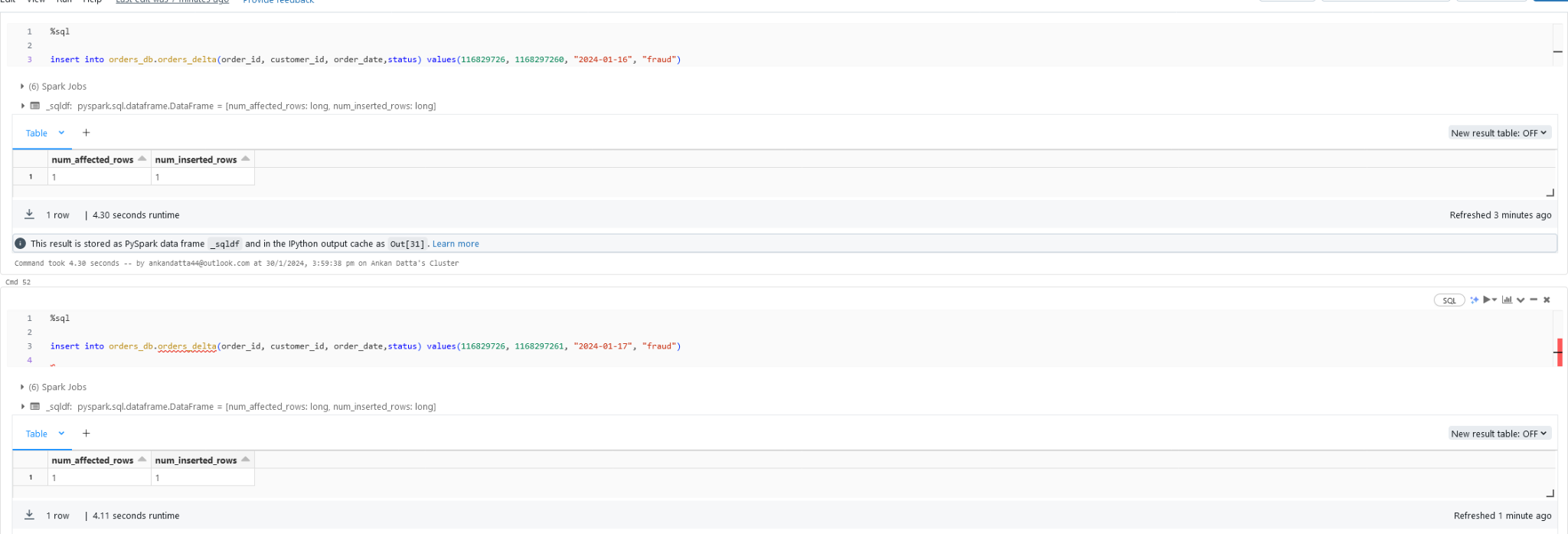
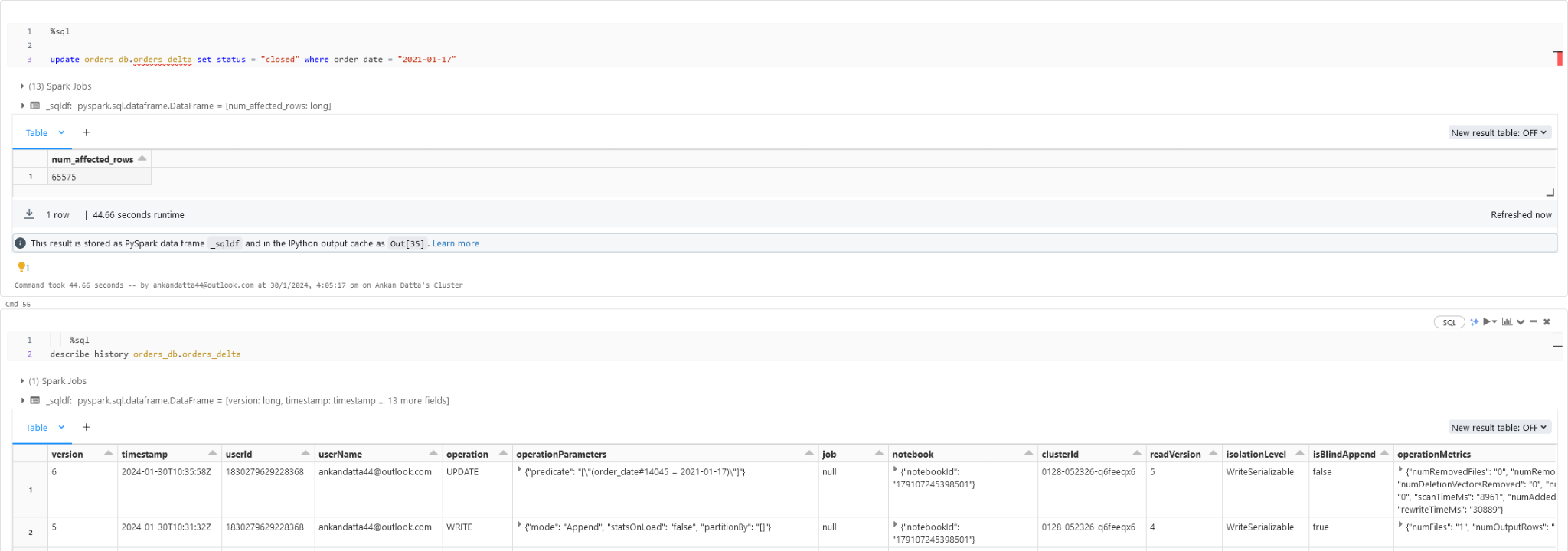
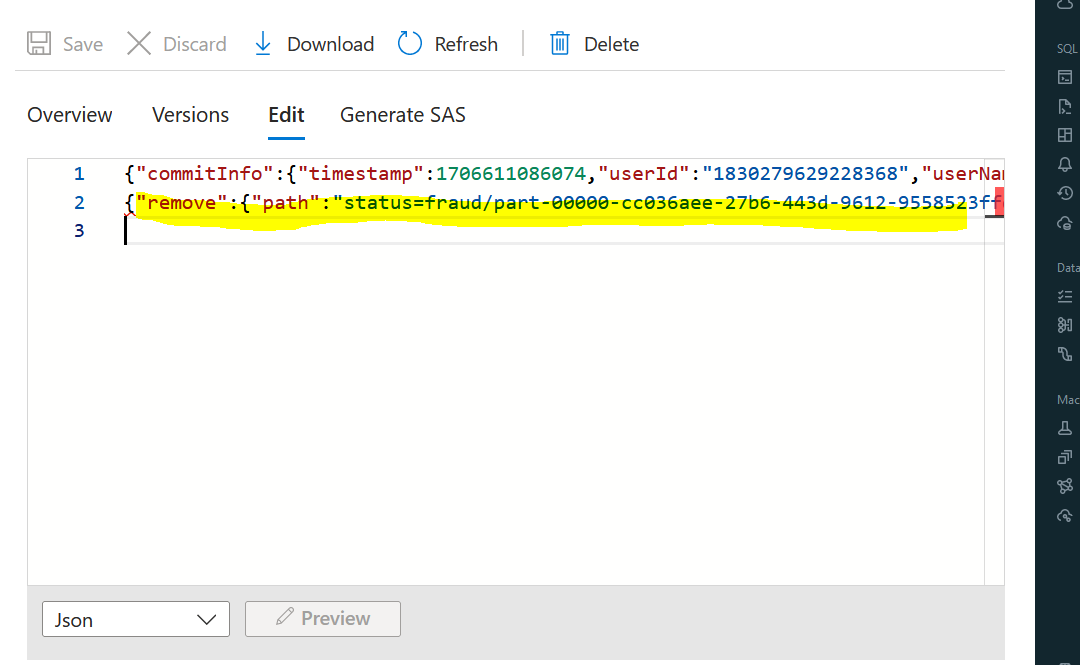
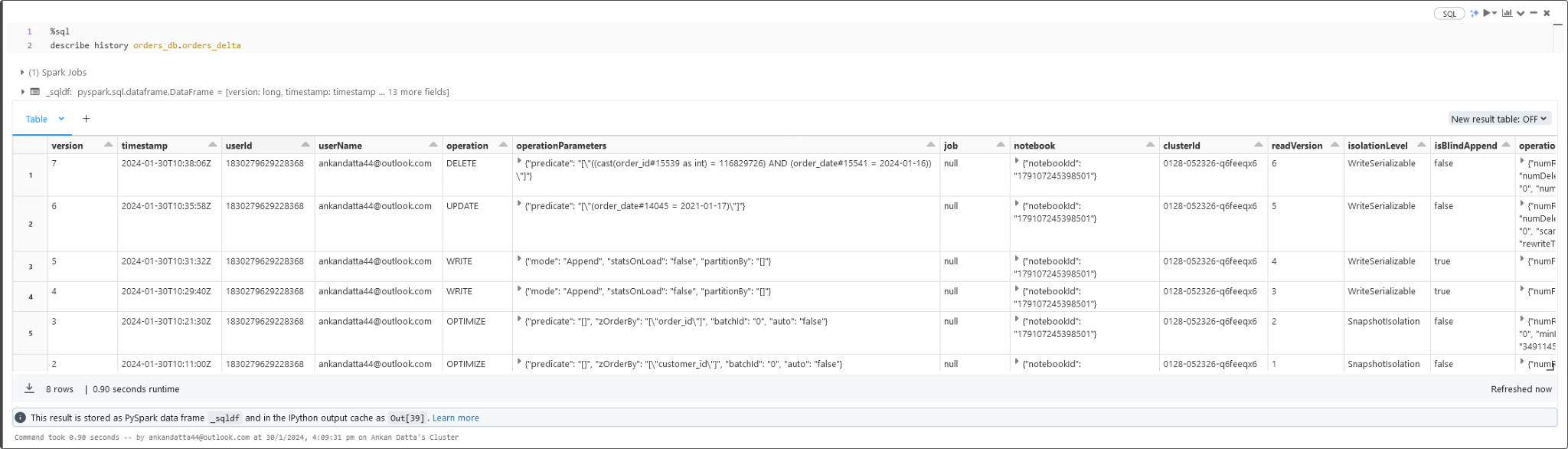
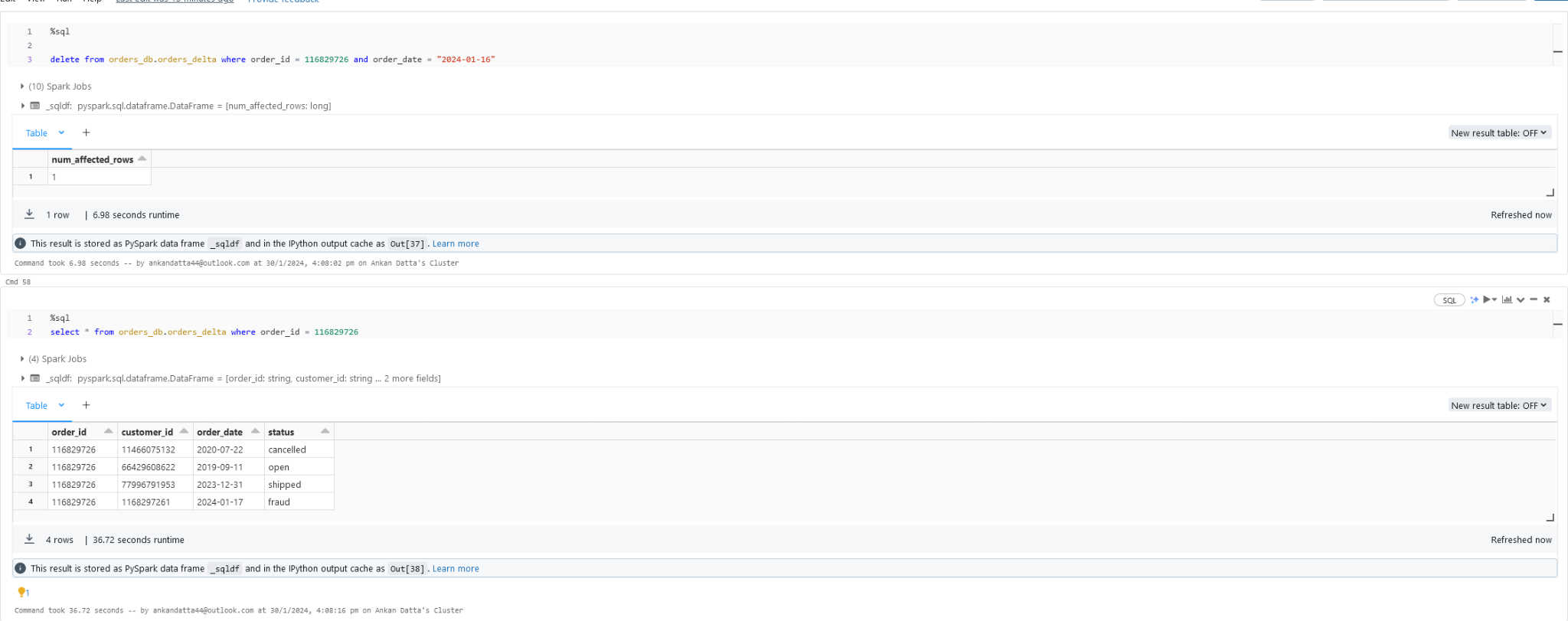
**Demonstrate by applying it to the example dataset considered. (You can**

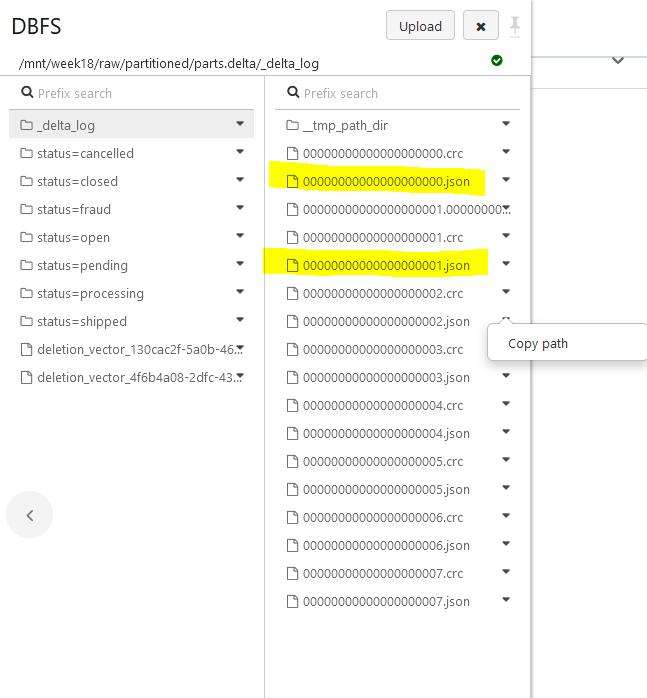
**perform insert, update and delete operations to showcase the increasing**

**transaction logs and how it would impact the resources over time.)**

* Vacuum command can be used when we no longer want to reference the older or the previous files.
* It removes the data files based on two criteria :

1. They are no longer referenced in the latest transaction logs
2. If those files were older than 7 days.

* It effects time travel as the historical data is deleted
* It is also a compute intensive operation
* INSERT : 
* UPDATE : 
* DELETE : 



* These are the new files added and we will use the vacuum command to remove them as the latest version doesn't need them.
* 