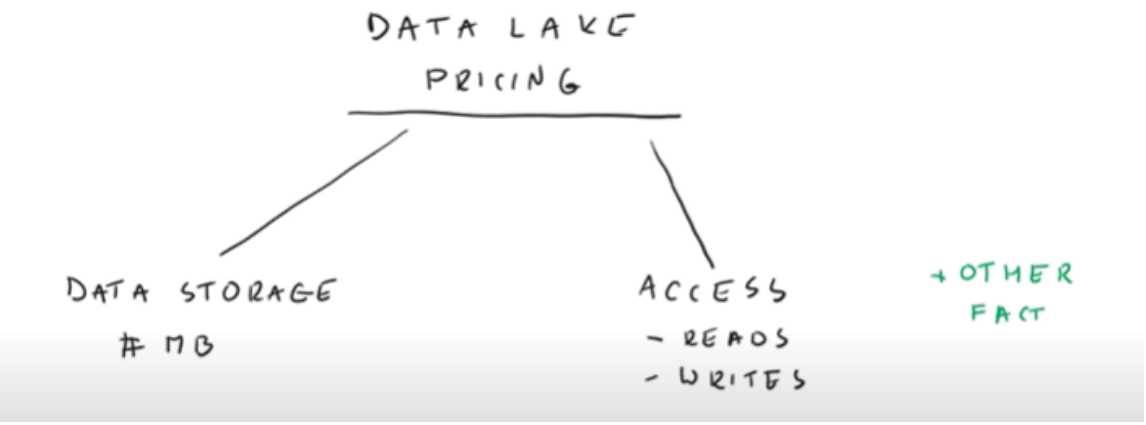
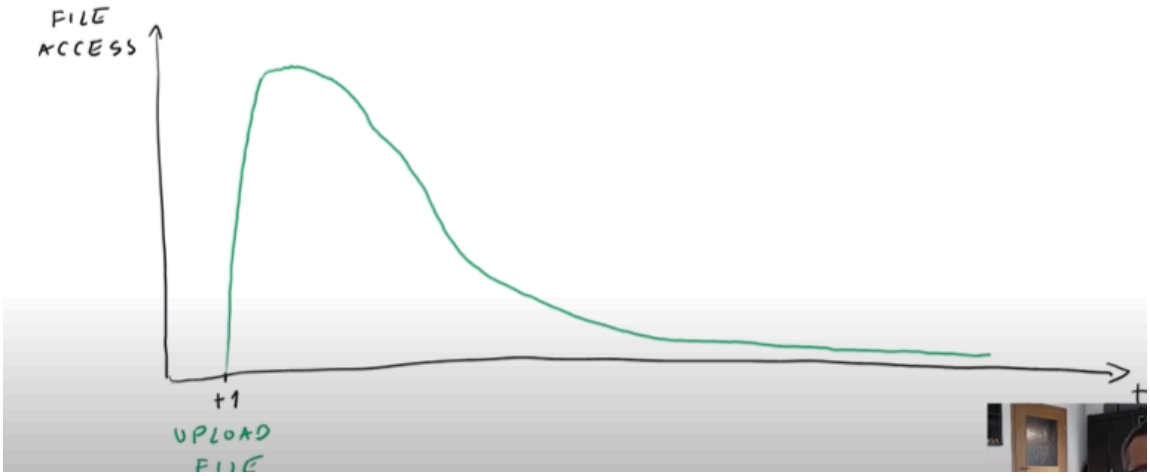
DP-203: 05- Storage account access tiers

Data usage over time

1. Basically in ADL we pay for two things



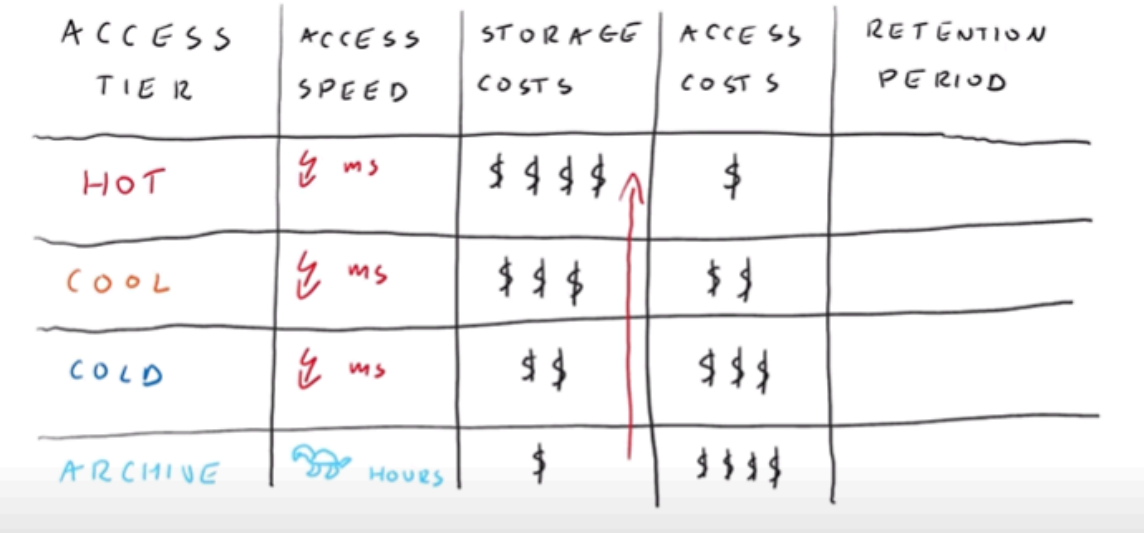
   +other things such as redundancy factors
2. Let us take a graph …on Y-axis we have file access(Read/Write operation) and on X-axis we have time



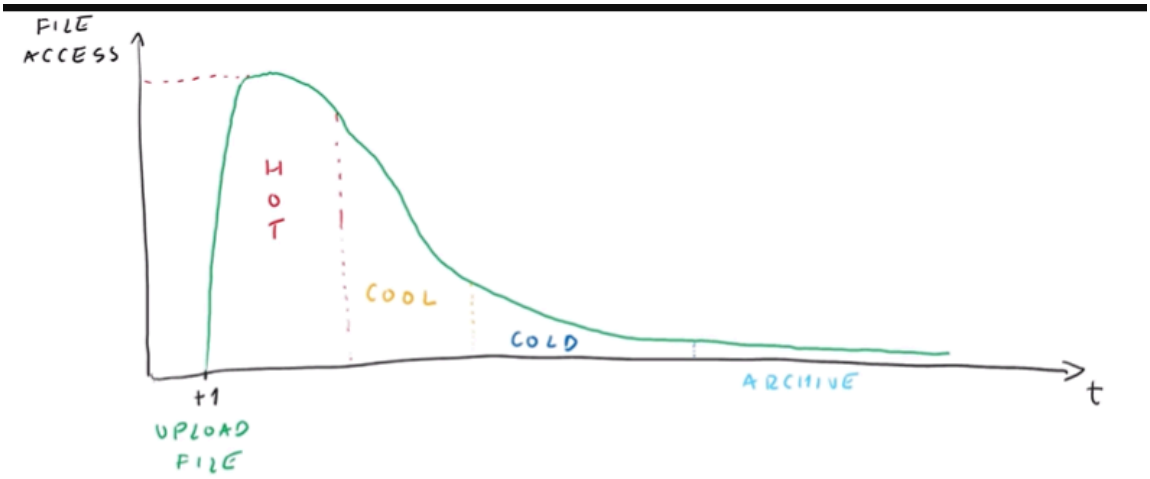
3. Here if we upload a file at time t1(then for 1 week there will be heavy Read/Write Operations) after that file access operation slows down with time
4. So with the help of access tiers we can control our cost spending

Access tiers

1. Here the access tiers are based on access speed(How fast we can get data), storage costs, access costs(Read/Write operations) and retention period
2. Based on those factors we have

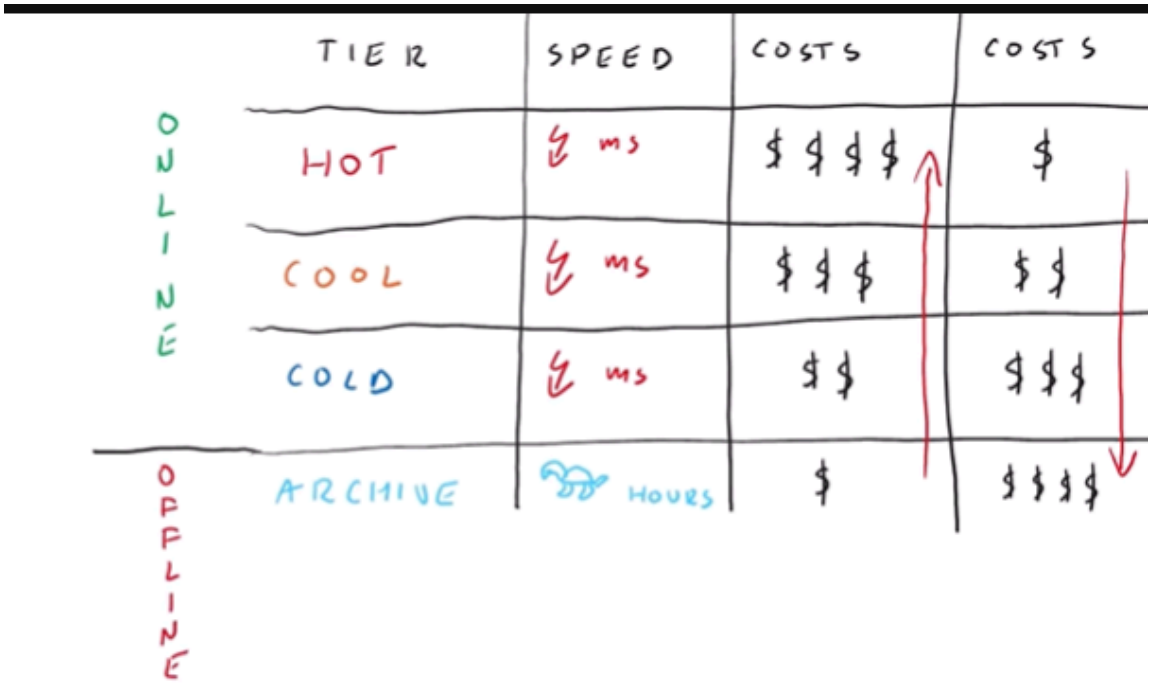


3. Hot access tire is more expensive in storage costs and least expensive in access costs..we can see the pic and understands access tiers
4. When to use which access tier?



5. Like when we upload some files initially..then there will high Read/Write operations..so we use Hot…and later if the data is of no use..then we move it to the other access tier

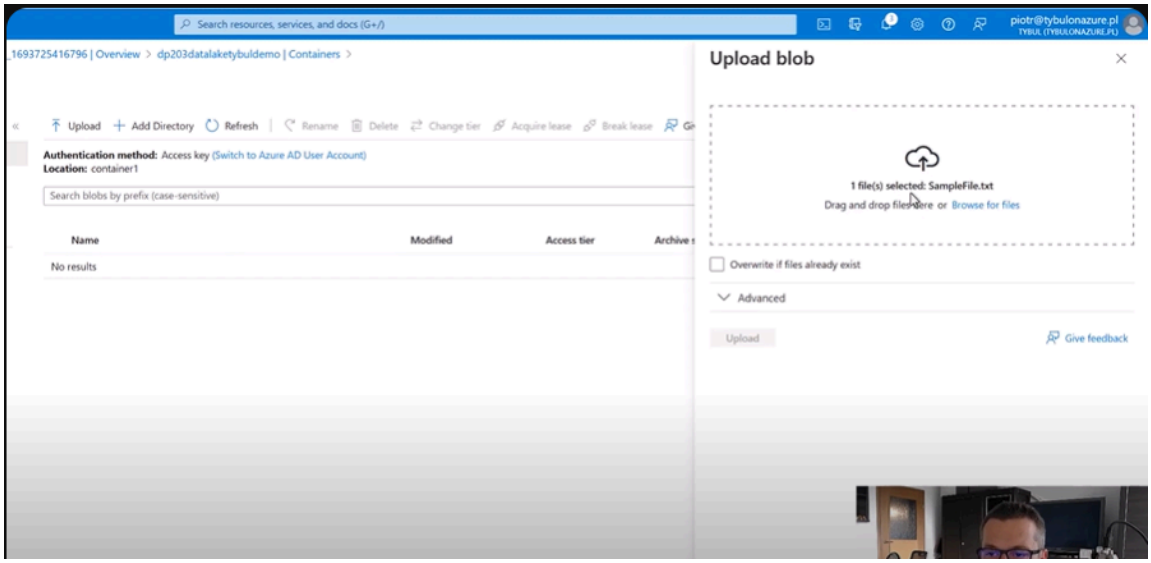
6. Here we use archive tier to store any legal data etc



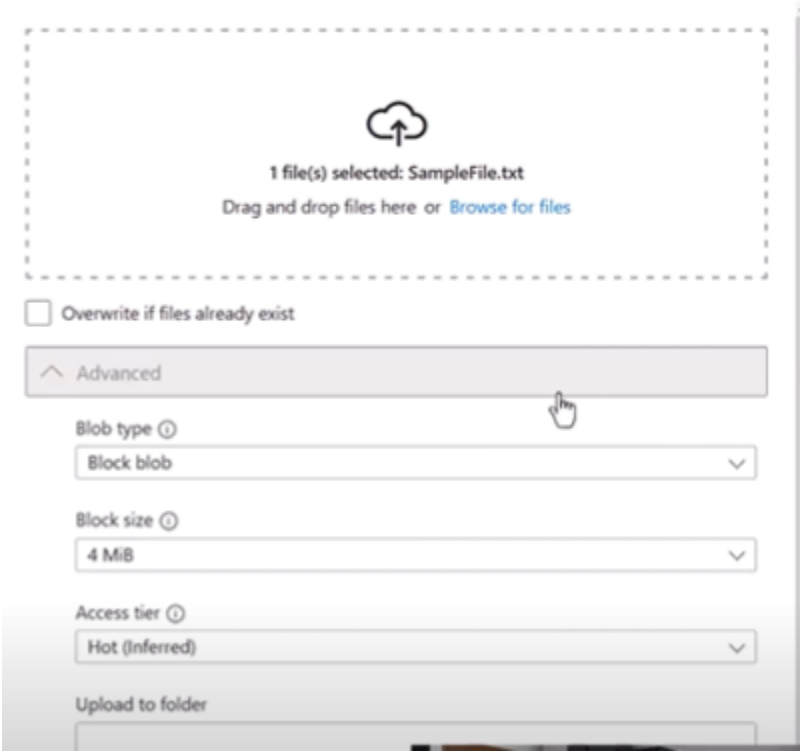
7. To use the data which is in archive tier…we need to move it in any other tier(which takes hours)

Access tier demo

1. Here first we will create a container ..and inside that we upload a file

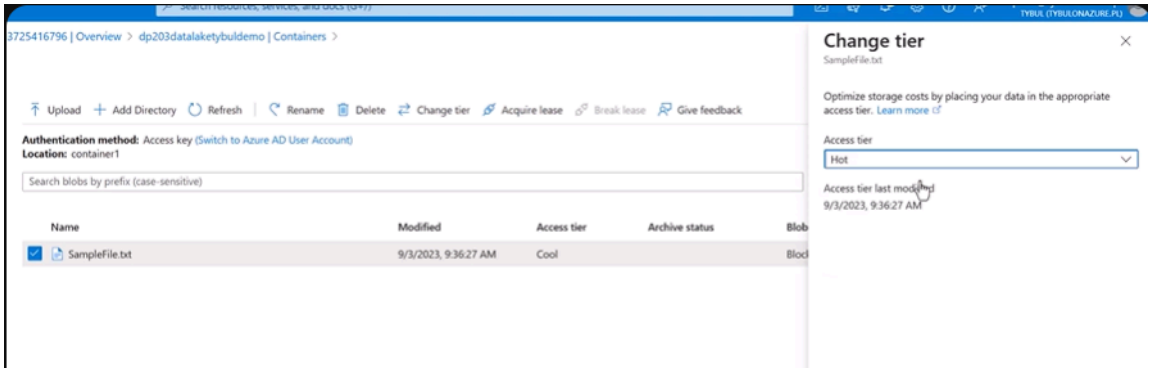


In the advanced section..we can specify the access tier

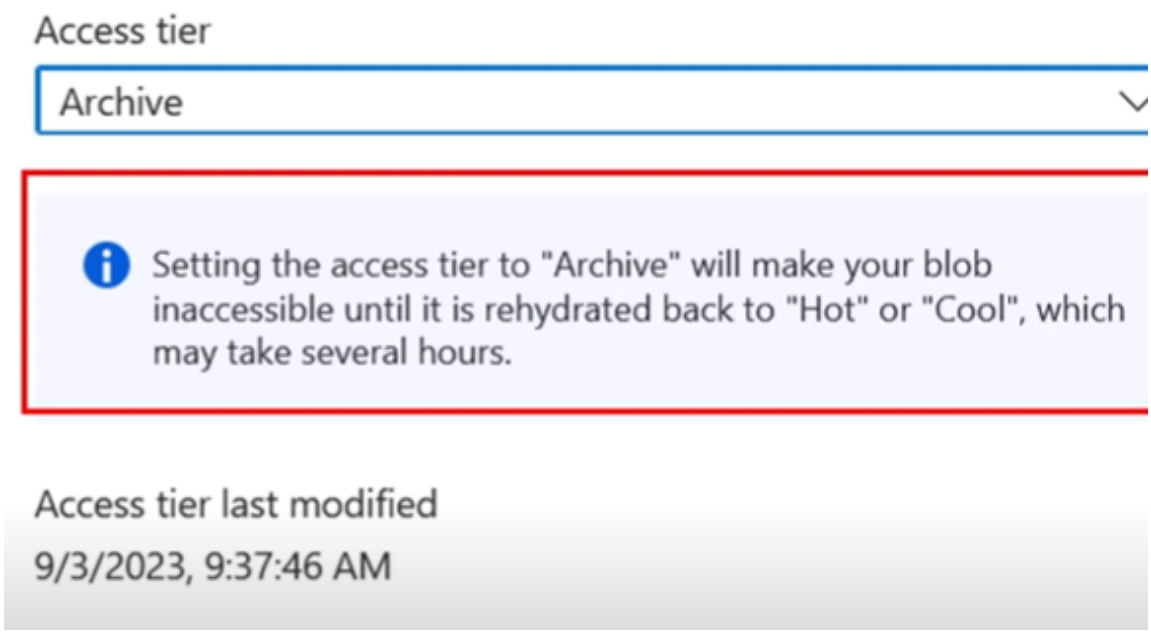


2. Hot tier will be default..if we do not select any tier

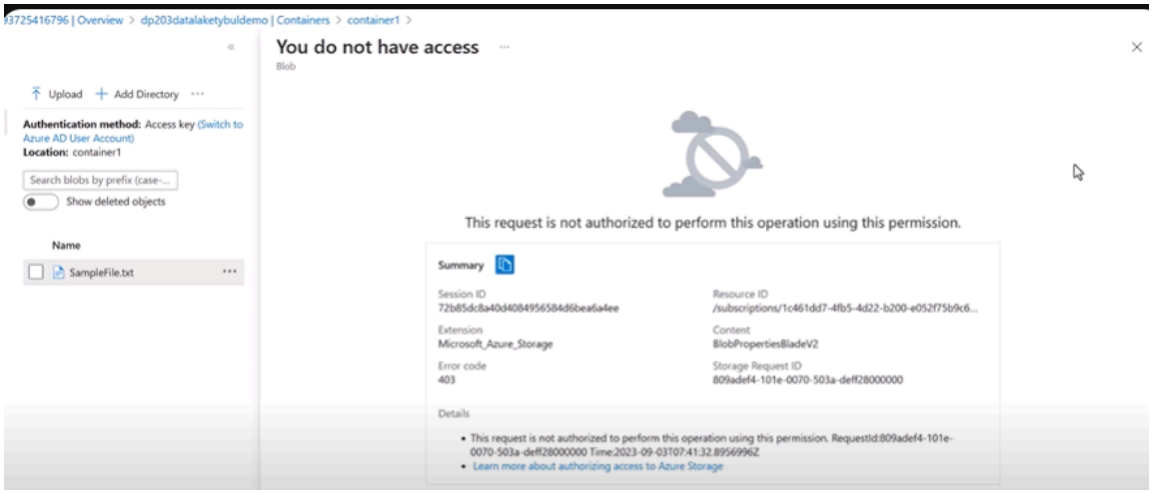
3. We can also use change tier option..to move files bw the tiers



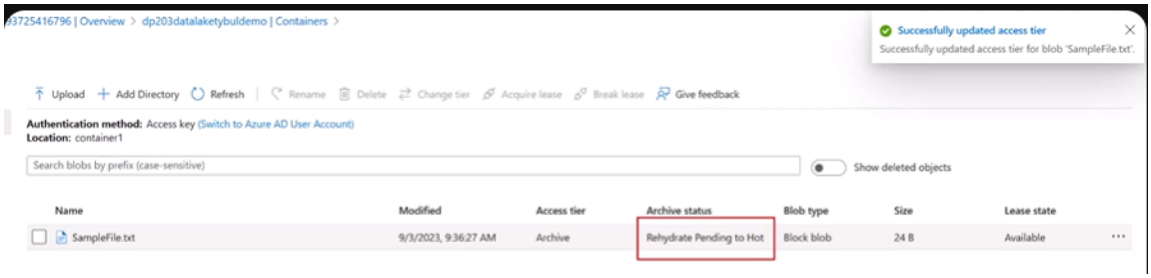
4. If we move for file to archive tier..it gives us



5. And when we try to access the file in archive tier then we get



6. To get access to this file…first we have to rehydrate it into other access tiers



7.

Datalake Pricing

Data storage

	Premium	Hot	Cool	Cold	Archive
First 50 terabyte (TB) / month	$0.15 per GB	$0.0184 per GB	$0.01 per GB	$0.0036 per GB	$0.00099 per GB
Next 450 TB / month	$0.15 per GB	$0.0177 per GB	$0.01 per GB	$0.0036 per GB	$0.00099 per GB
Over 500 TB / month	$0.15 per GB	$0.0169 per GB	$0.01 per GB	$0.0036 per GB	$0.00099 per GB

1. here we can see to store the data HOT is expensive and archive is cheap
2. Transaction pricing(Reads/Writes)

Transaction

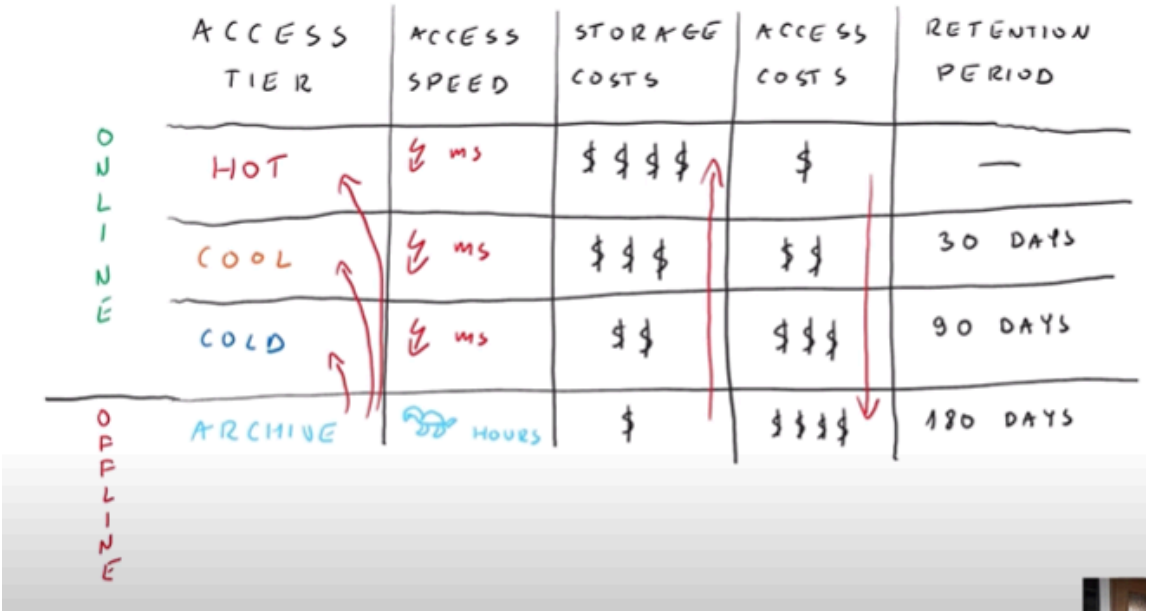
	Premium	Hot	Cool	Cold	Archive
Write Operations* (every 4MB, per 10,000)	$0.0228	$0.065	$0.13	$0.234	$0.13
Read Operations** (every 4MB, per 10,000)	$0.00182	$0.0052	$0.013	$0.13	$6.50
Query Acceleration - Data Scanned (per GB)	N/A	$0.002	$0.002	$0.002	N/A
Query Acceleration - Data Returned (per GB)	N/A	$0.0007	$0.01	$0.01	N/A

*The following API calls are considered write operations: AppendFile, CreateFilesystem, CreatePath, CreatePathFile, FlushFile, SetFileProperties, SetFilesystemProperties, RenameFile, RenamePathFile, CopyFile

**The following API calls are considered read operations: ReadFile, ListFilesystemFile

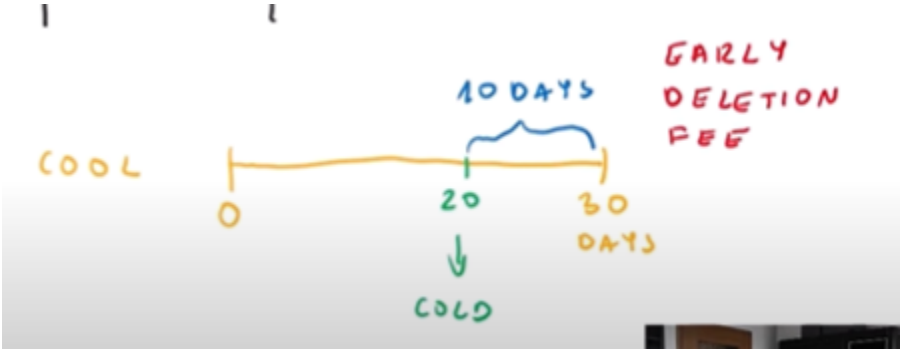
Retention Period

1. It is nothing but for how long will the file stay in one tier
2. So here the file has to be present in the cool,cold and archive for specified retention period

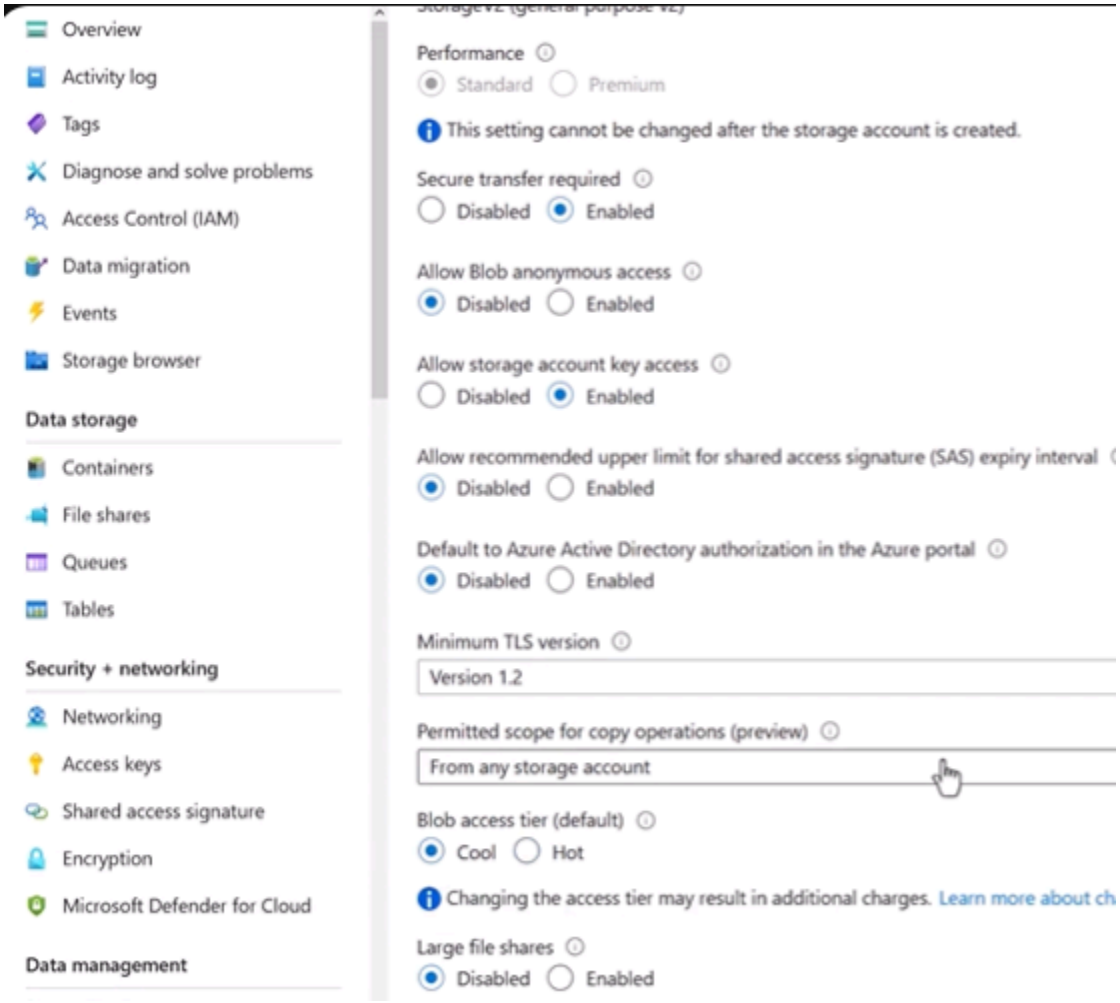


or else there will be fine associated with it

3. So this fine is called as early deletion fee

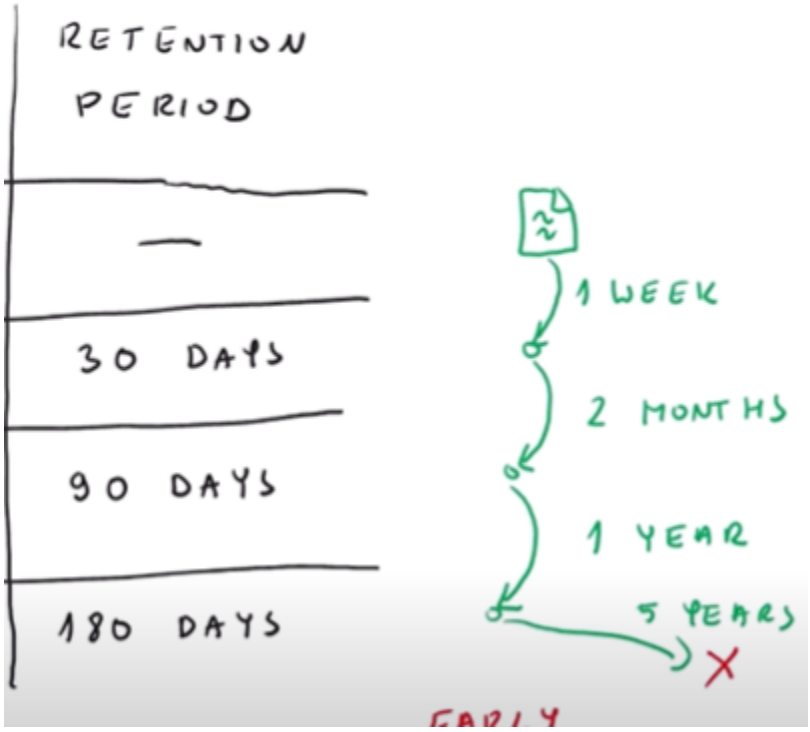


4. We can also change default access tier to cold…by going into configurtation

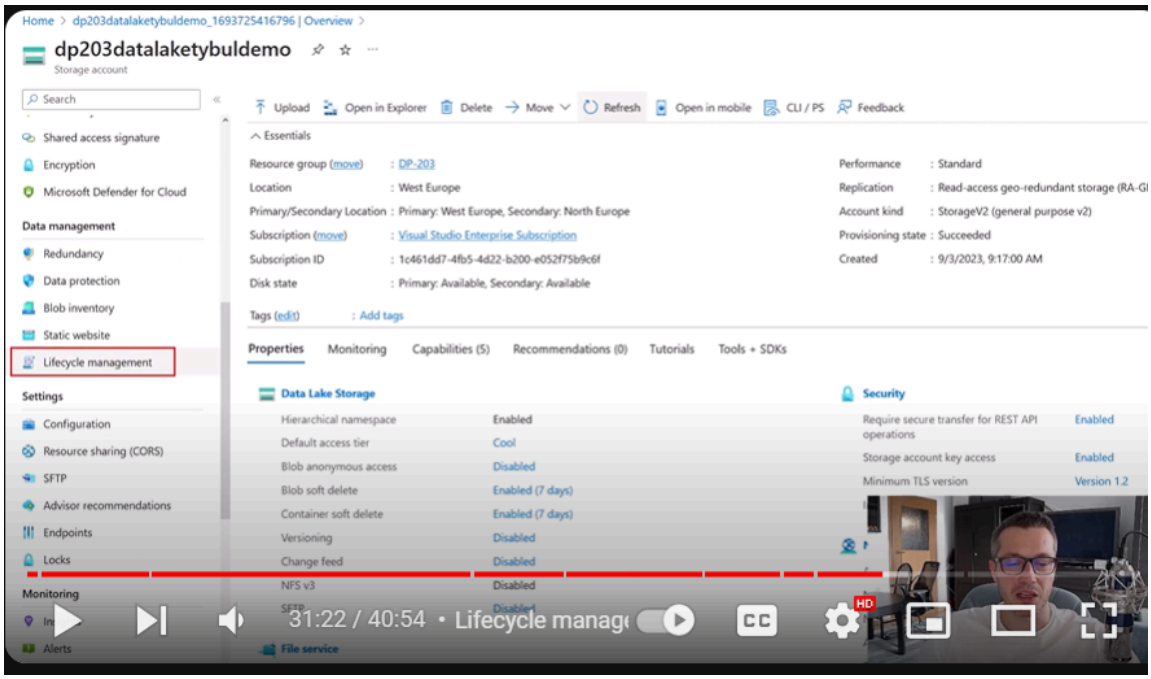


Life cycle management

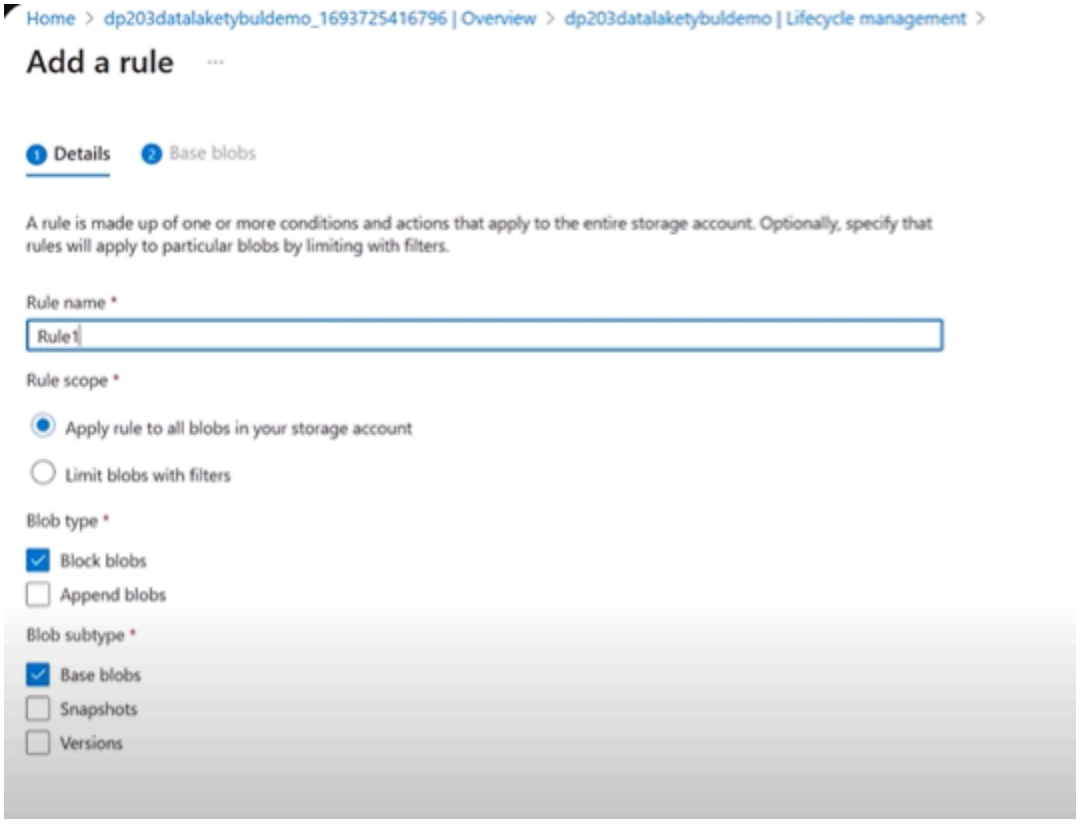
1. So we can define life cycle rules for files..to automatically change the access tiers



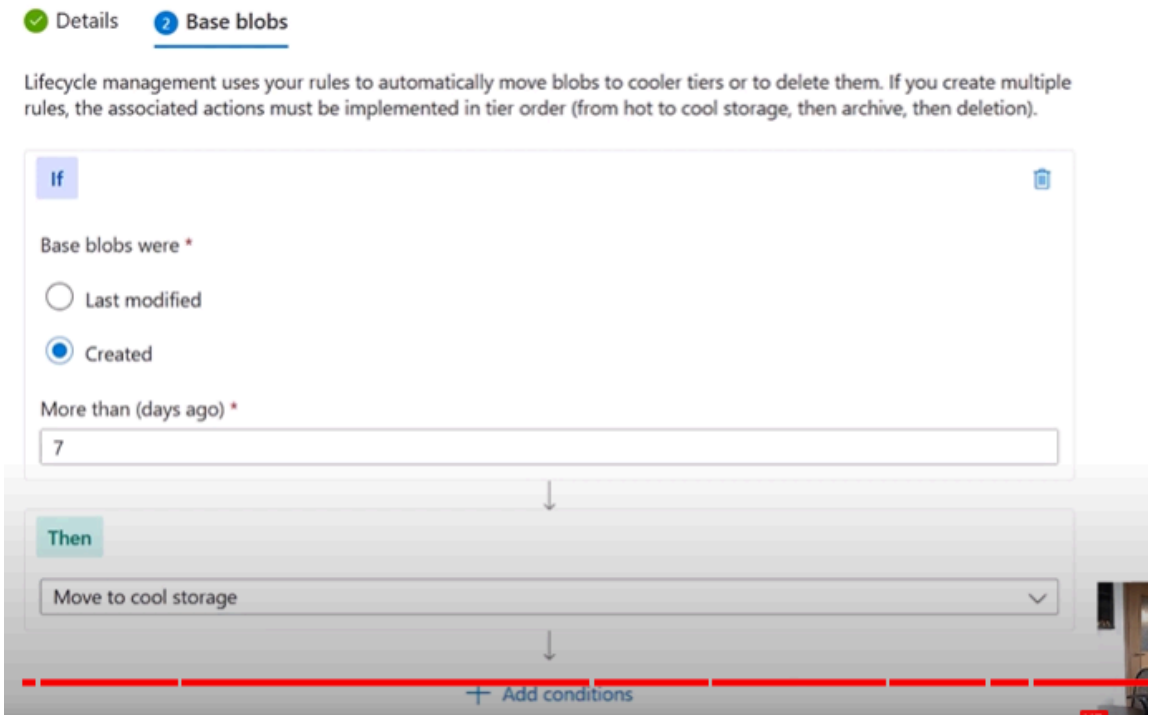
2. Inside the datalake we have life cycle management



3. Here we can add rules her

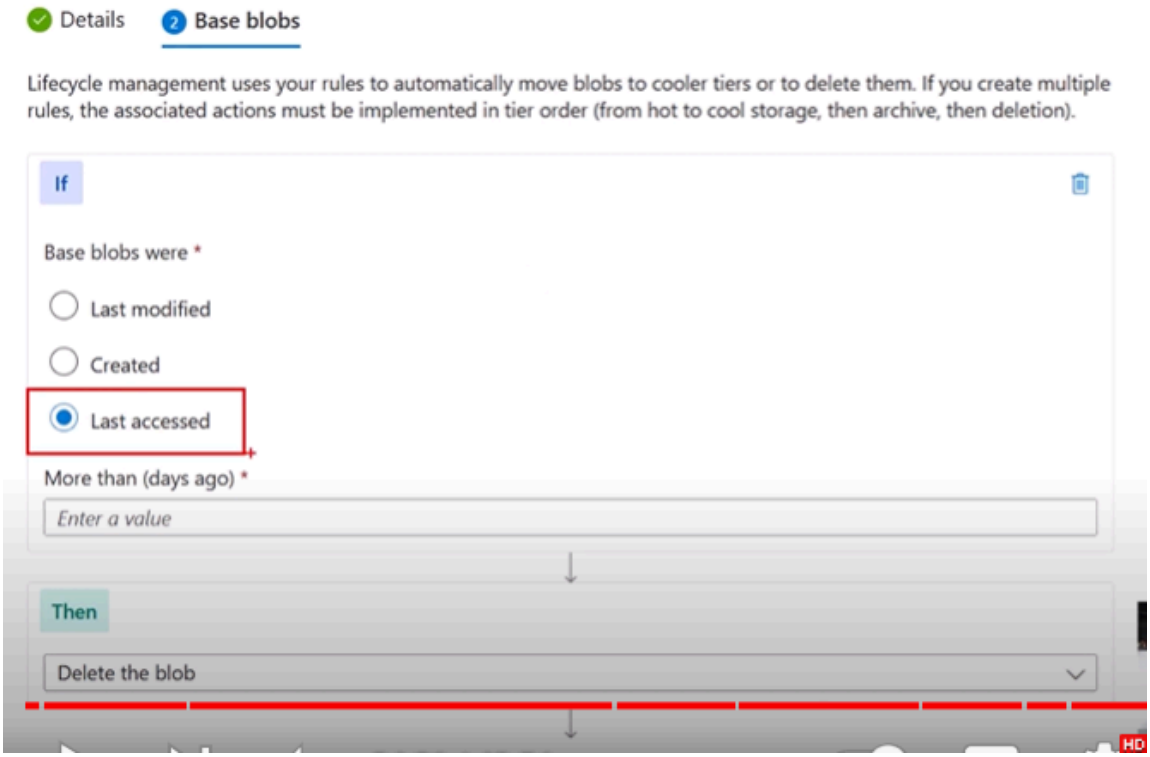


e



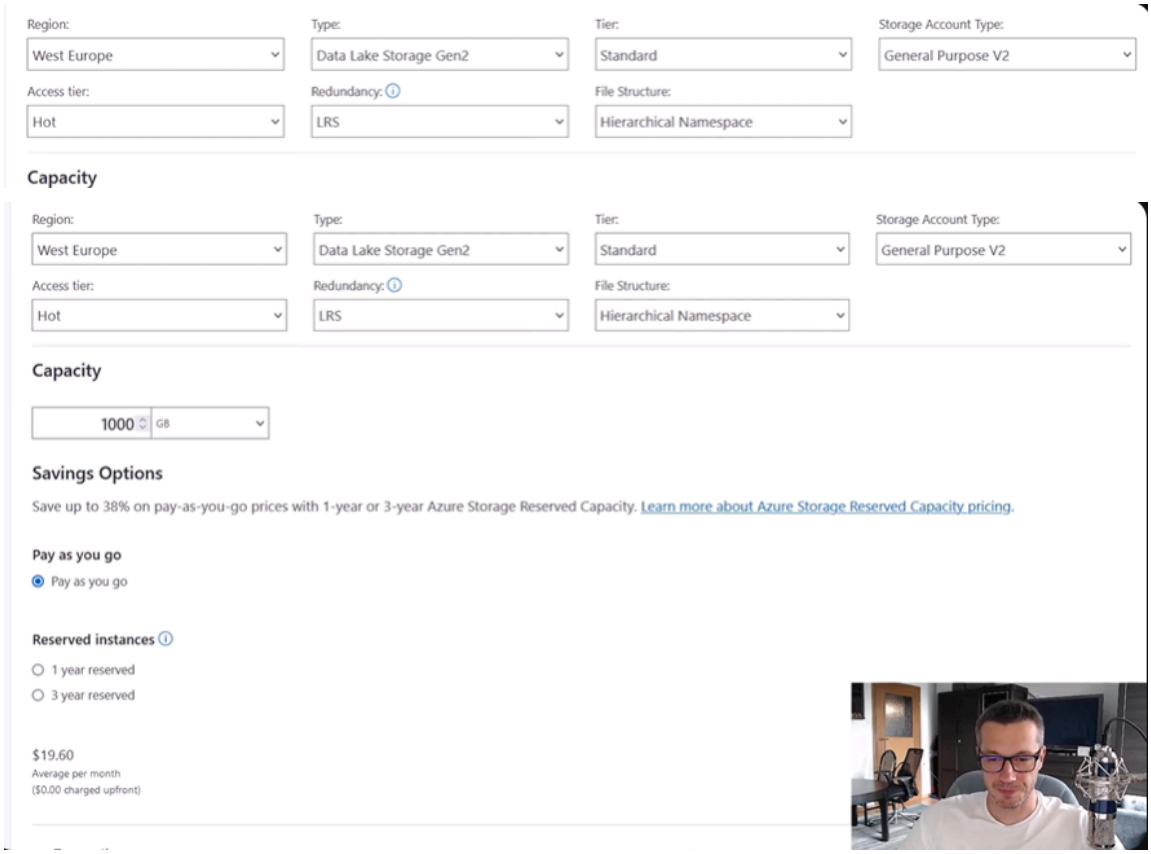
So..here after the file is created…after 7 days it moves to access tier

4. And if we enable access tracking…then we can used it to change the access tiers

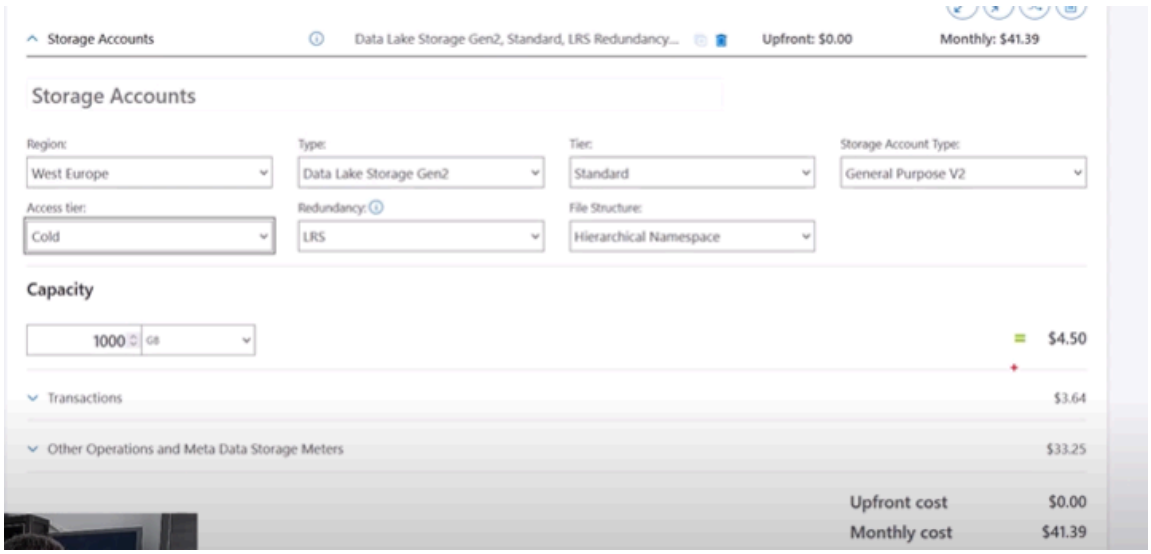


Azure Pricing Calculator

1. Here we can azure calculate the price using



2. We will paying for capacity and transactions too

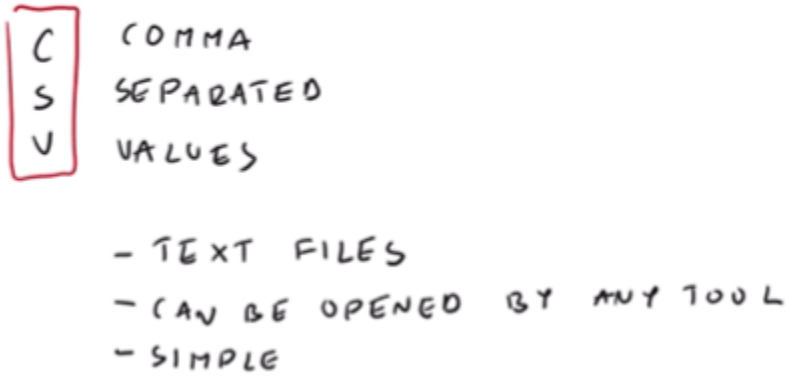


3. We can just play around with it

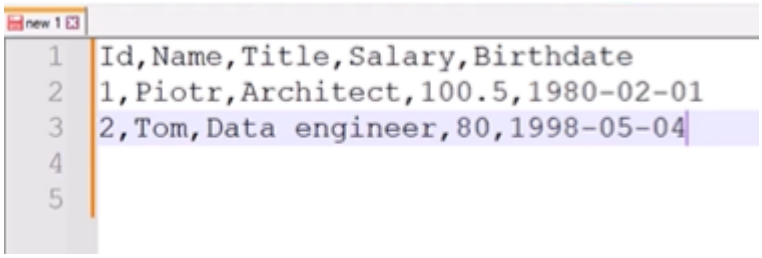
DP-203: 06 - Common File Types (CSV, XML, JSON)

CSV files

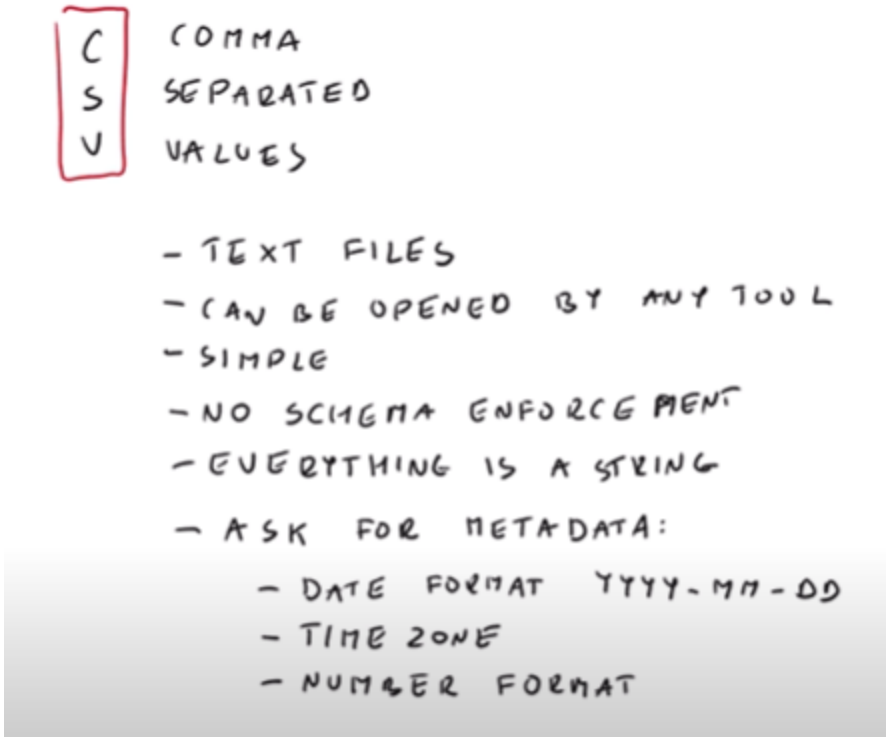
1. They are comma separated values
2. First row would be header..where it stores col names



3. And here there's no way we can enforce schema
4. Here every value will be a string inside the csv
5. Sample CSV data



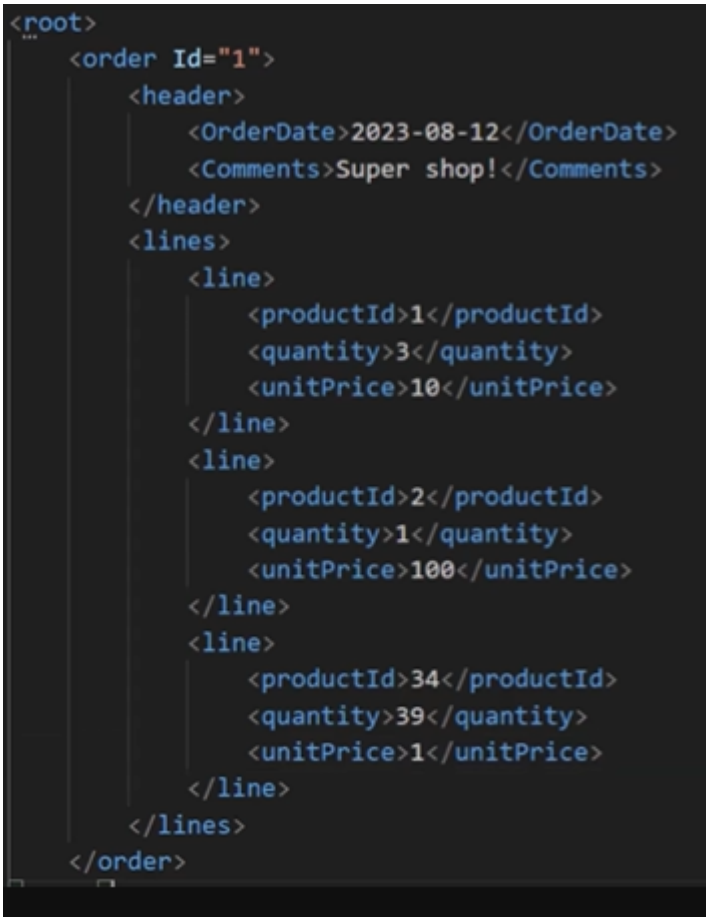
6. We have to ask metadata (to know the date format etc)

7. 

8. Actually in CSV the data stored in flat

XML

1. It stands for extensible markup language
2. Here we work with tags 

3. It is mostly used when to exchange data between diff application/systems


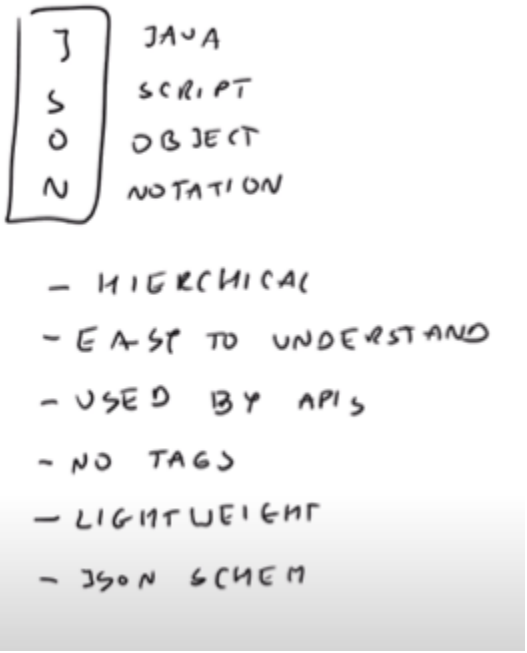
4. It supports hierarchical data
5. The XML code is well formatted and it is used by API's
6. But XML is very lengthy

JSON

1. Stands for Java Script object notation

2. Its hierarchical and easy to understand

```
{
    "Order": {
        "Id" : 1,
        "Header": {
            "OrderDate": "2023-01-01",
            "Comments": "Super product!"
        },
        "Lines": [
            {
                "productId": 1,
                "quantity" : 3,
                "unitPrice" : 10
            },
            {
                "productId": 2,
                "quantity" : 31,
                "unitPrice" : 1
            },
            {
                "productId": 3,
                "quantity" : 12,
                "unitPrice" : 10
```



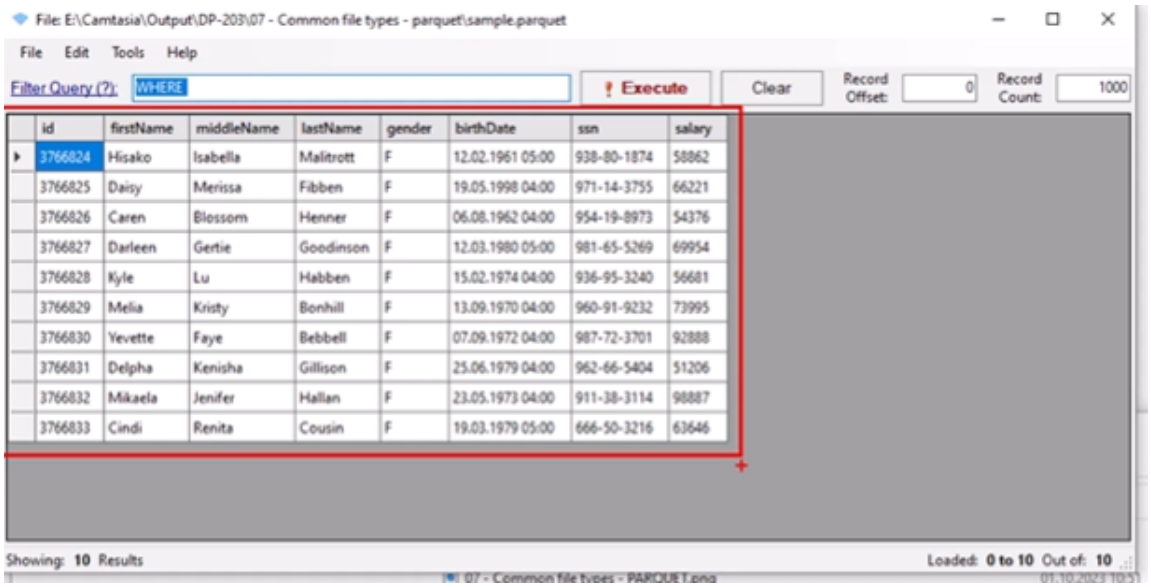
3.

DP-203: 07 - Parquet File Format

Parquet Overview

1. It is most often used in data analytic/engineering
2. Parquet follows binary format and it requires some tools to open/work this files
3. So we have to download parquet viewer to open this files

4. Here we can see the parquet file …opened using tool

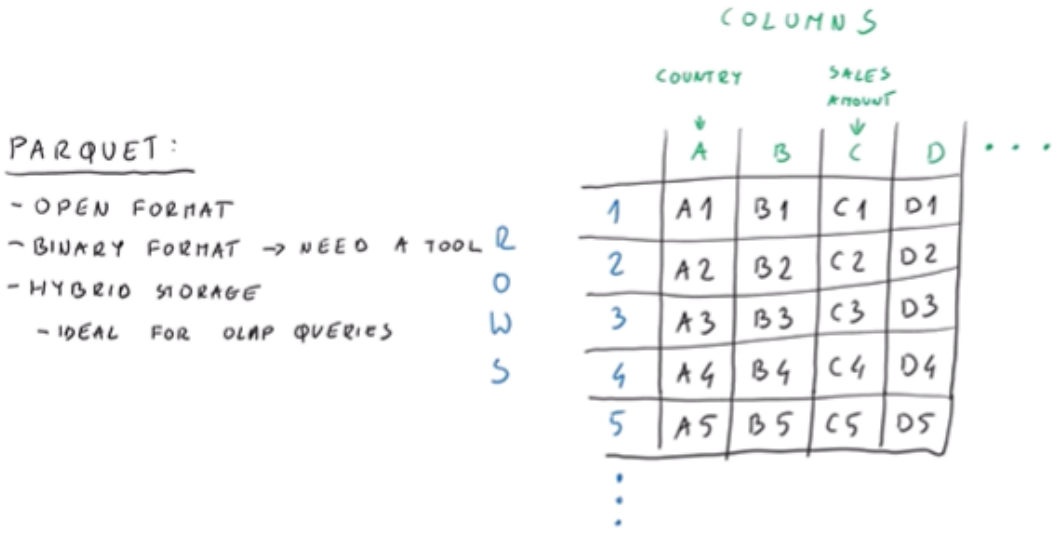


Storage modes

1. Parquet stores the data in hybrid format

	A	B	C	D
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5

2. Lets consider we have this table
3. And assume A->Country, C->Sales Amount Data



4.

```
SELECT
    SUM(SALES_AMOUNT)
FROM
    TABLE

GROUP BY
    COUNTRY
```

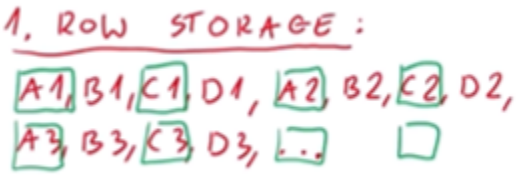
5. Sample OLAP query
6. Here in our query we only need 2 columns…A & C

3 Types of Storage

Row Storage

Here first we write A1,B1,C1,D1…and then A2,B2,C2,D2

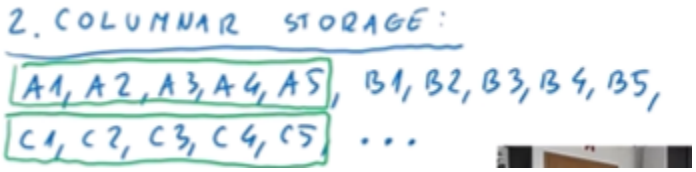
1. And here we only need 2 col for our query..



2. But the main drawback is …row storage..will read all the col's…which needs more compute power

Columnar Storage

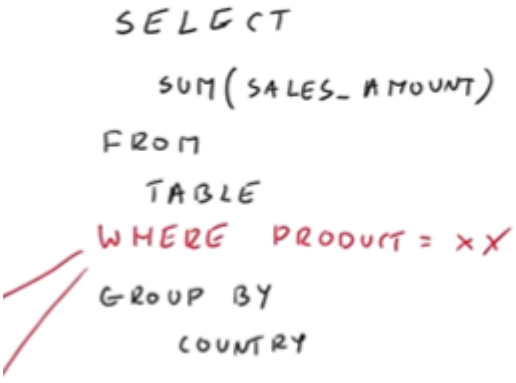
1. Here the data will store in columnar format



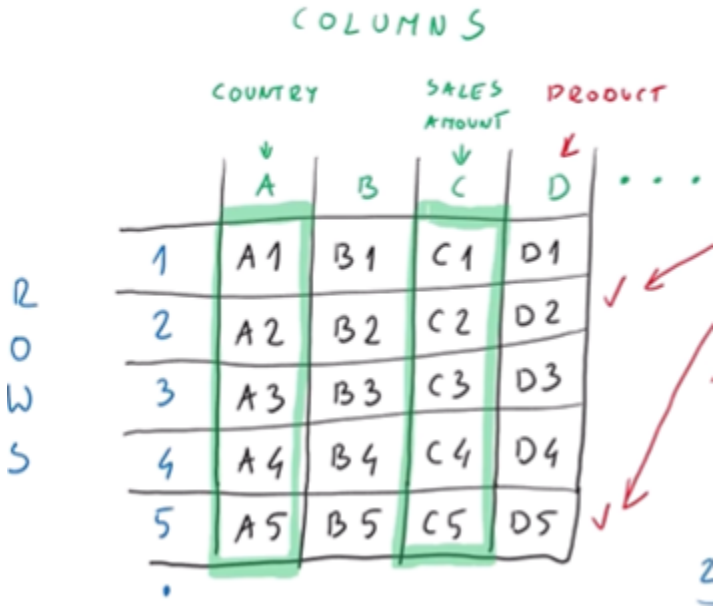
2. As our query only need col A and Col C…Now here we'll only be retrieving col A and Col C…and we ignore Col B …in this storage format
3. It gives results much faster…this storage is great for OLAP

Hybrid Storage

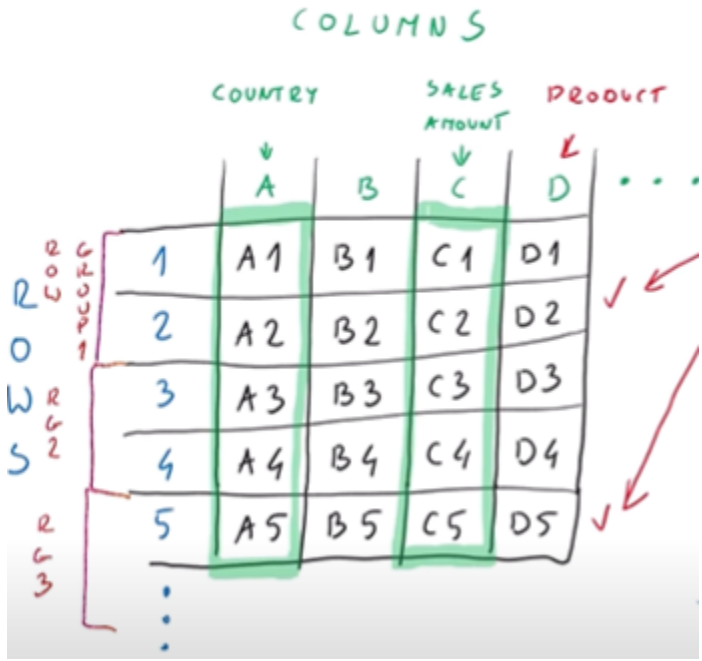
1. Lets modify our query



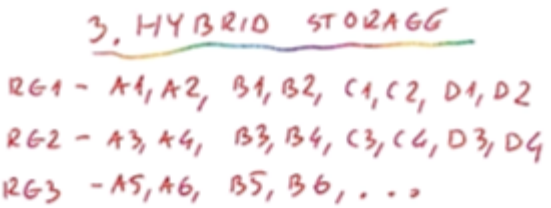
2. Here we have added a filter(where) product



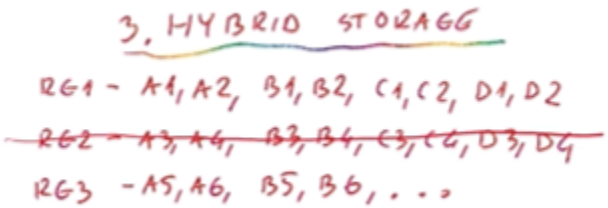
3. It combines the both row and columnar storage
4. Here in the hybrid storage..we'll be having row groups



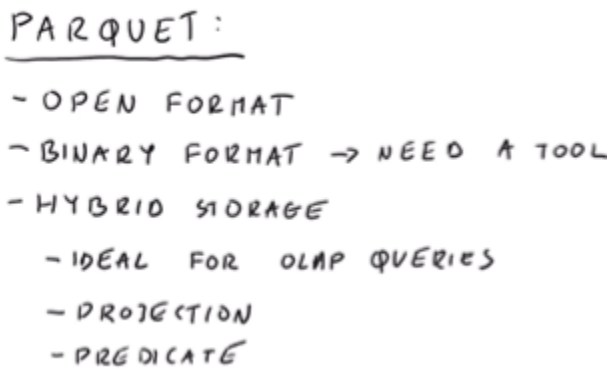
and inside row groups..the data is stored in columnar format



5. The data is stored in this way
6. Also Parquet stores the meta data …like max,min values of col
7. And based on metadata it can filter row groups



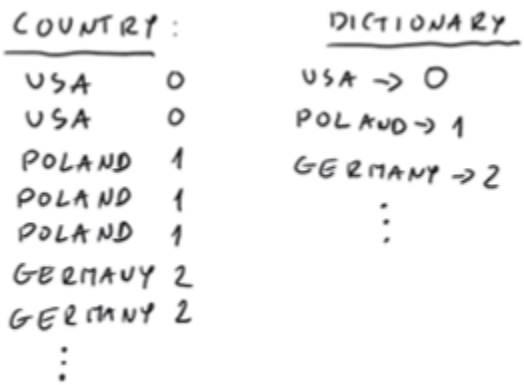
8. So this hybrid format is the most optimized one…it makes read operation very fast



Compression on Parquet

1. In our datalake the csv file takes 713MB and the same file in parquet takes 224MB

2. This because parquet follows dictionary encoding



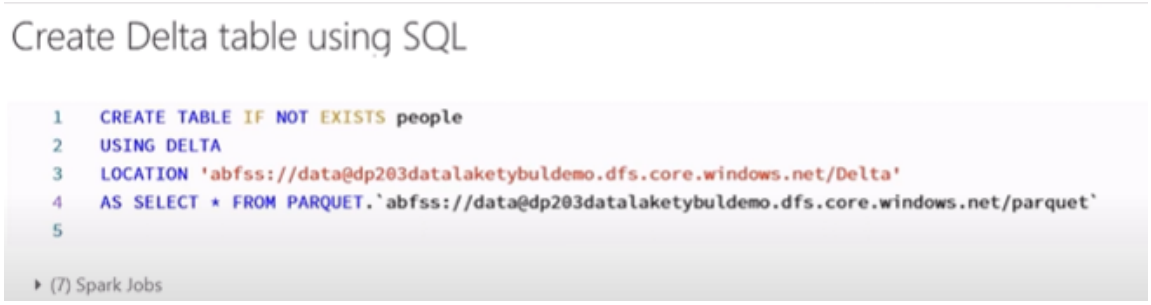
3. We also have run-length encoding
4. Also the schema will be embedded in the parquet files

5. Same command for csv file format

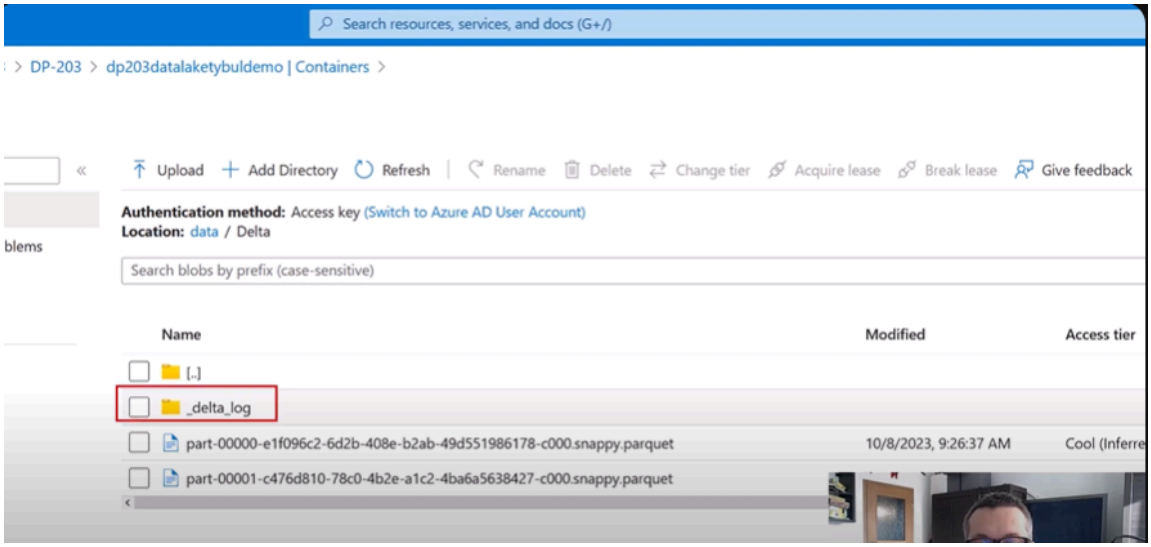
DP-203: 08 - Delta File Format

Overview

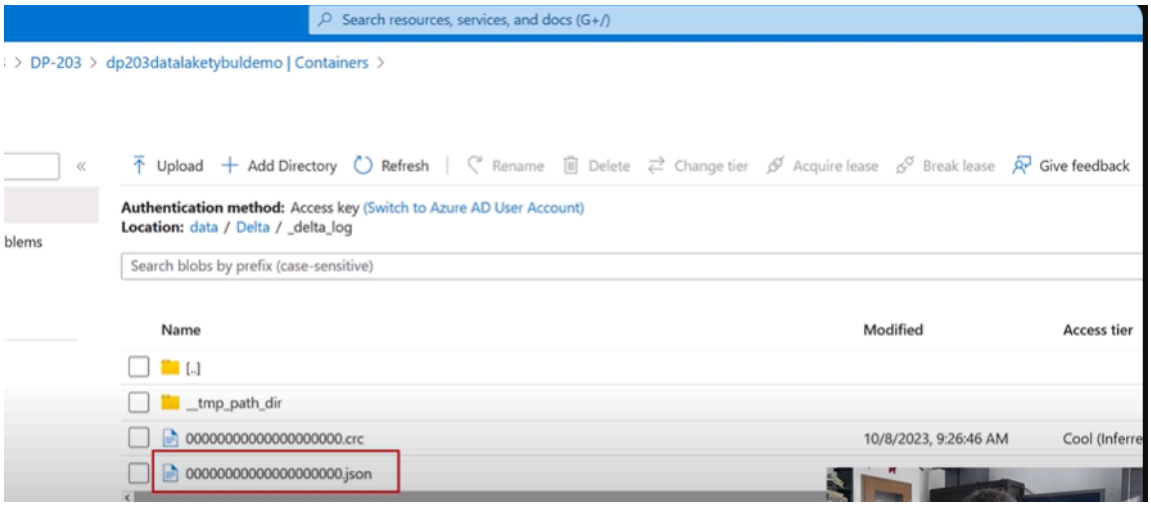
1. This is the ultimate file format which has super cool features
2. Its an open source file format
3. Its same like parquet format…but it has transactional logs on top of it
4. Lets create a delta table and see how it is in data lake



5. Here the important thing is our logs



it stores the logs in the JSON format



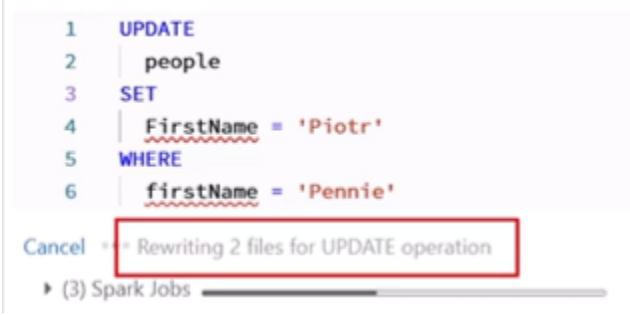
6.

7. Here we have min,max values for every column

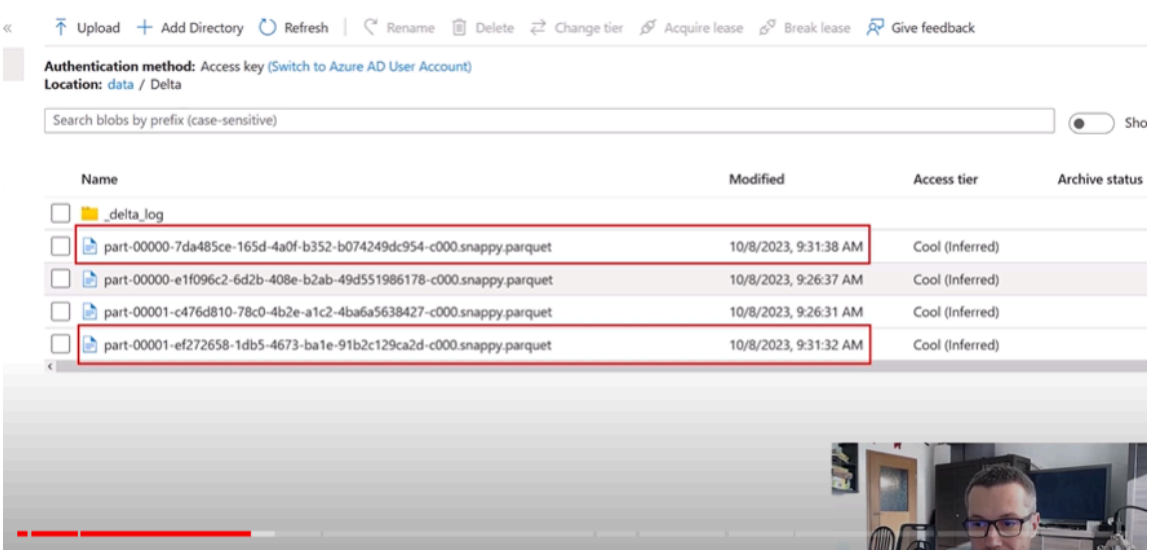
8. Now here we are changing the first name

```
UPDATE
  people
SET
  FirstName = 'Piotr'
WHERE
  firstName = 'Pennie'
```

9. Delta table records this operation in data lake in Logs



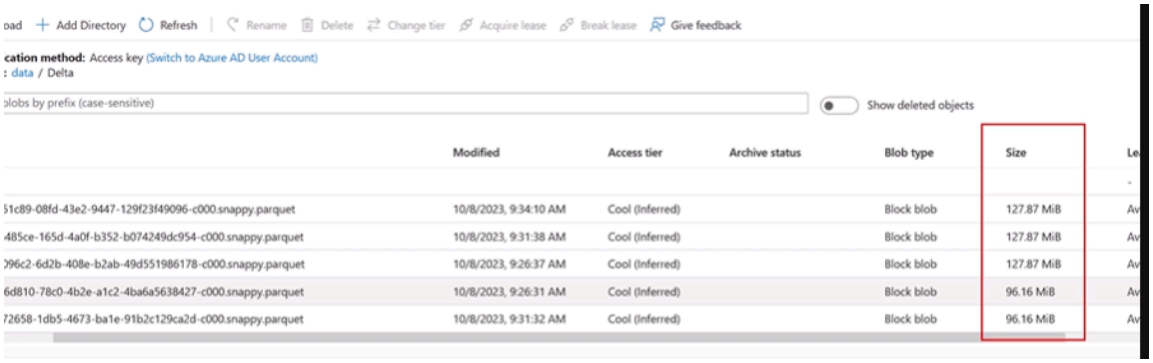
10. Also it create the different files too



11. Now we'll again update the firstname

12. For every CRUD operation…it stores the logs in delta logs and also it create a parquet file as well

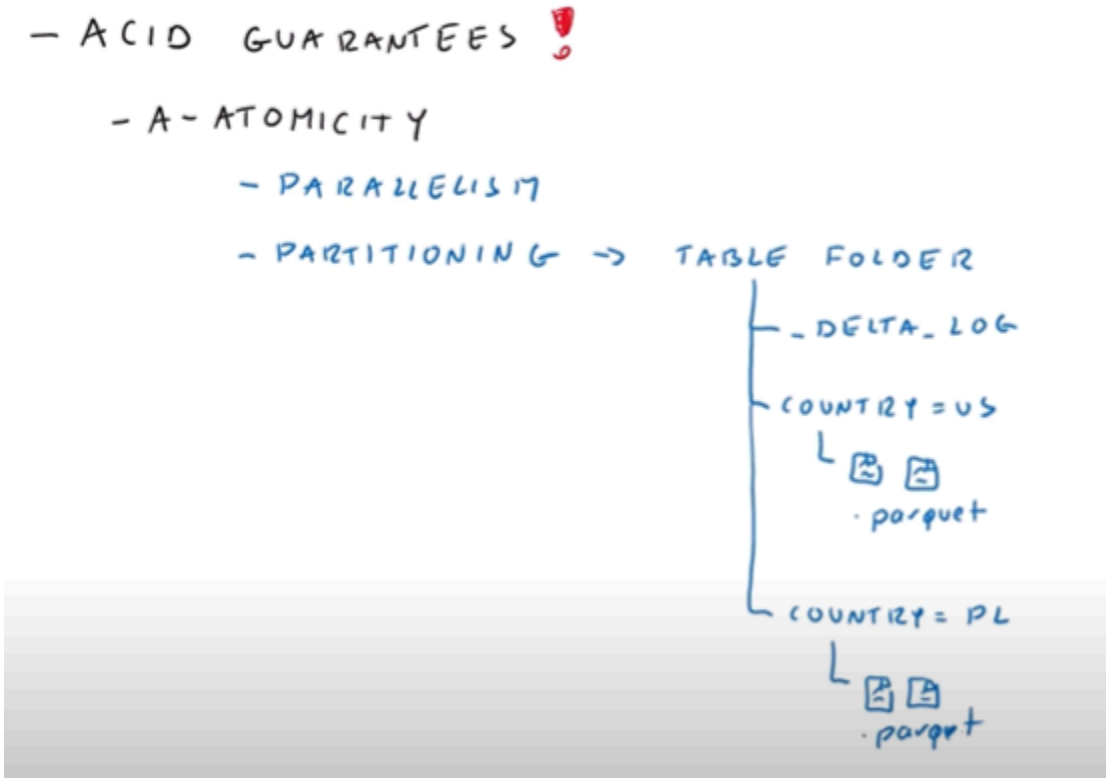
13. And here for every update it create a parquet file and in turn it is taking some storage…and at the end we have to pay for it



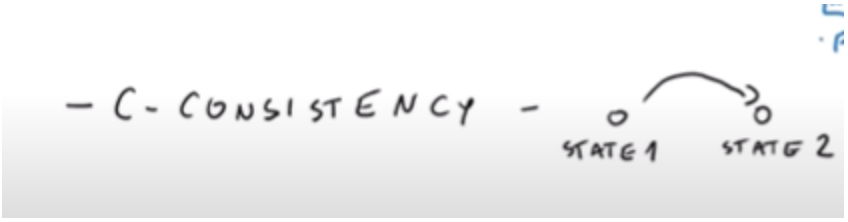
14. Whats the point of storing historical data ..instead of updating in place

ACID Guarantees

1. This delta lake guarantees the acid
2. Lets say we have many tables partitioned in different folders



3. And imagine we have created a query which updates every file
4. Now what atomicity guarantees..after making changes to every file then only it create delta log in our storage
5. C - Consistency

6. 

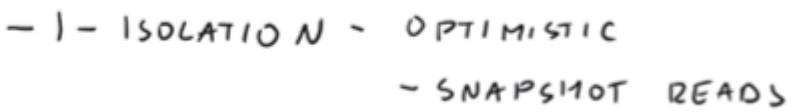
suppose are changing our data from state1 to state2…what consistency guarantee's is it will check the constraints and make sure they follow constraints of the table ….
7. Out of context example on consistency

Consider the following scenario without consistency checks:

- Account A has $100, and account B has $50.
- A user attempts to transfer $150 from A to B (exceeding the balance in A).

If the transaction isn't checked for consistency, it might incorrectly update the balances as follows:

- Account A: -$50 (negative balance)
- Account B: $200 (total money increased)



8. I - Isolation
9. Example

Scenario without Isolation:

1. John reads the inventory, sees 1 size 10 shoe available, and initiates a purchase.
2. Before John's transaction completes (updating the inventory), Jane also reads the quantity (1) and starts her purchase.
3. Both John and Jane's transactions might succeed, showing a confirmed purchase for each. However, there's only one pair of shoes!

Scenario with Isolation:

1. When John starts his purchase, the database locks the size 10 shoe record, preventing Jane from reading the same data at that moment.
2. John's transaction completes, deducting the quantity from the inventory (now 0).
3. The lock is released, and Jane can now read the updated inventory (0). Her purchase will fail due to insufficient stock.

10. D - Durability

> Durability, within the ACID properties of database transactions, guarantees that once a transaction is committed (successfully completed), the changes made to the database are persisted permanently. This persistence ensures that the updates survive even in case of system failures like crashes, power outages, or hardware malfunctions.

our storage redundancy will ensure this

Audit History

11. Delta lake also provides us audit history

```
1  DESCRIBE HISTORY people
```

▸ (1) Spark Jobs

Table +

	version	timestamp	userId	userName	operation	operationParameters
1	2	2023-10-08T07:34:10.000+0000	2099708230325839	piotr@tybulonazure.pl	UPDATE	{"predicate": "[\"(id#1742 = 1234)\"]"}
2	1	2023-10-08T07:31:38.000+0000	2099708230325839	piotr@tybulonazure.pl	UPDATE	{"predicate": "[\"(firstName#1079 = Pennie)\"]"}
3	0	2023-10-08T07:26:38.000+0000	2099708230325839	piotr@tybulonazure.pl	CREATE TABLE AS SELECT	{"partitionBy": "[]", "description": null, "isManaged": "false"}

3 rows | 1.70 seconds runtime

Command took 1.70 seconds -- by piotr@tybulonazure.pl at 10/8/2023, 9:52:54 AM on Piotr (admin)'s Cluster

12. Now we can see who did what and all using the describe command

Time Travel

1. Here we can see previous state of data…
2. Using the files delta lake created and using the delta logs

☐	_delta_log		
☐	part-00000-3fe51c89-08fd-43e2-9447-129f23f49096-c000.snappy.parquet	10/8/2023, 9:34:10 AM	Cool (Inferred)
☐	part-00000-7da485ce-165d-4a0f-b352-b074249dc954-c000.snappy.parquet	10/8/2023, 9:31:38 AM	Cool (Inferred)
☐	part-00000-e1f096c2-6d2b-408e-b2ab-49d551986178-c000.snappy.parquet	10/8/2023, 9:26:37 AM	Cool (Inferred)
☐	part-00001-c476d810-78c0-4b2e-a1c2-4ba6a5638427-c000.snappy.parquet	10/8/2023, 9:26:31 AM	Cool (Inferred)
☐	part-00001-ef272658-1db5-4673-ba1e-91b2c129ca2d-c000.snappy.parquet	10/8/2023, 9:31:32 AM	Cool (Inferred)

we can see the data of file at specific point of time

3. To time travel data we can use this command

Time travel

```
SELECT * FROM people TIMESTAMP AS OF '2023-10-08T07:31:38.000+0000'
WHERE
  id = 1234
```

(2) Spark Jobs

Table +

	id	firstName	middleName	lastName	gender	birthDate	ssn	salary
1	1234	Bambi	Britni	Nuccii	F	1981-03-02T05:00:00.000+0000	666-44-0021	52424

4. Also we can use this versions too..to see our previous state of data

```
DESCRIBE HISTORY people
```

(1) Spark Jobs

Table +

	version	timestamp	userId	userName	operation	operationParameters
1	2	2023-10-08T07:34:10.000+0000	2099708230325839	piotr@tybulonazure.pl	UPDATE	{"predicate": "[\"(id#1742 = 1234)\"]"}
2	1	2023-10-08T07:31:38.000+0000	2099708230325839	piotr@tybulonazure.pl	UPDATE	{"predicate": "[\"(firstName#1079 = Pennie)\"]"}
3	0	2023-10-08T07:26:38.000+0000	2099708230325839	piotr@tybulonazure.pl	CREATE TABLE AS SELECT	{"partitionBy": "[]", "description": null, "isManaged" "false"}

```
SELECT * FROM people VERSION AS OF 1
WHERE
  id = 1234
```

(2) Spark Jobs

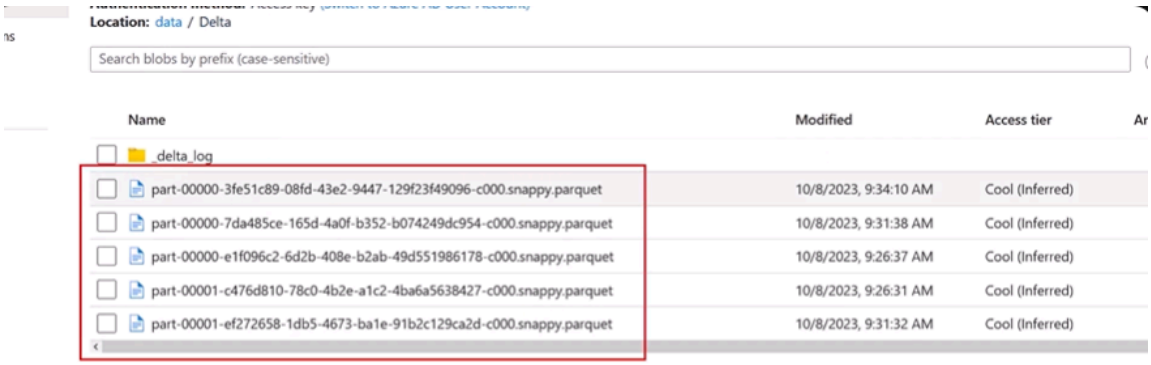
Table +

	id	firstName	middleName	lastName	gender	birthDate
1	1234	Bambi	Britni	Nuccii	F	1981-03-02T05:00:00.000+0000

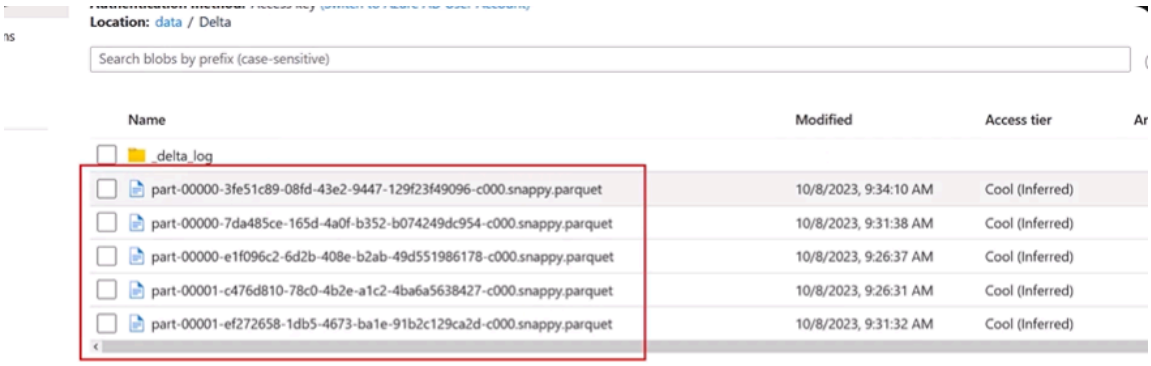
5. This allows us to see and compare data
6. If we accidentally change the data..then we use this feature and go back to previous state

Vacuuming operation

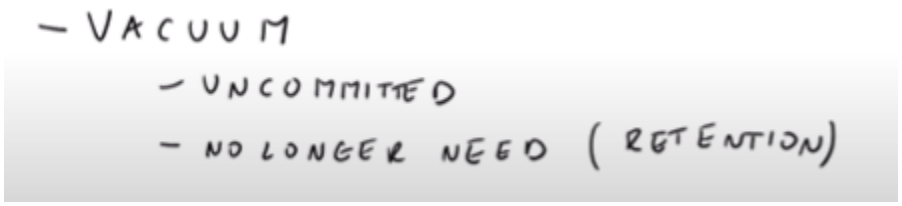
1. Here our delta lake created 5 files..



   And we'd like to remove extra files of data which we are no longer using
2. And also if our transaction is failed then it created additional files..and we'd like to remove them



3. Vacuum is just for this….it will remove



   by default delta lake store historical files for 7 days…then later if we perform vacuum ..then old files will be removed
4. If delete files using vacuum..then we cannot use time travel to go to prev state

Schema enforcement

1. Sometimes the schema is not enforce in parquet format

```
+----------+---+
|first_name|age|
+----------+---+
|   leonard| 51|
|       bob| 47|
|        li| 23|
+----------+---+
```

2. Here we have create a df with firstname and age
3. Later we have appended new data with new column datatypes to the prev df

```
1  %python
2  columns = ["first_name", "favorite_color"]
3  data = [("sal", "red"), ("cat", "pink")]
4  rdd = spark.sparkContext.parallelize(data)
5  df = rdd.toDF(columns)
6
7  df.write.mode("append").format("parquet").save("tmp/parquet_table1")
8
9  spark.read.format("parquet").load("dbfs:/tmp/parquet_table1").show()
```

```
▸ (6) Spark Jobs
▾ df: pyspark.sql.dataframe.DataFrame
      first_name: string
      favorite_color: string

+----------+--------------+
|first_name|favorite_color|
+----------+--------------+
|   leonard|          NULL|
|       cat|          pink|
|       sal|           red|
|       bob|          NULL|
|        li|          NULL|
+----------+--------------+
```

4. Then for the prev df ..we get null values (as prev it was age…now it is fav_color)
5. Here we have corrupted our data
6. Coming to delta

7. Here we have create sample df using delta format

```python
%python

columns = ["first_name", "age"]
data = [("bob", 47), ("li", 23), ("leonard", 51)]
rdd = spark.sparkContext.parallelize(data)
df = rdd.toDF(columns)

df.write.format("delta").save("dbfs:/tmp/delta_table1")

spark.read.format("delta").load("dbfs:/tmp/delta_table1").show()
```

▸ (11) Spark Jobs

▸ df: pyspark.sql.dataframe.DataFrame = [first_name: string, age: long]

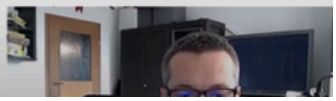
```
+----------+---+
|first_name|age|
+----------+---+
|   leonard| 51|
|       bob| 47|
|        li| 23|
+----------+---+
```

8. And we try to append 2nd df with diff schema

```python
%python
columns = ["first_name", "favorite_color"]
data = [("sal", "red"), ("cat", "pink")]
rdd = spark.sparkContext.parallelize(data)
df = rdd.toDF(columns)

df.write.mode("append").format("delta").save("dbfs:/tmp/delta_table1")

spark.read.format("delta").load("dbfs:/tmp/delta_table1").show()
```

▸ (2) Spark Jobs

```
AnalysisException: A schema mismatch detected when writing to the Delta table (Table ID: 82678b70-84ca-4e2c
To enable schema migration using DataFrameWriter or DataStreamWriter, please set:
'.option("mergeSchema", "true")'.
For other operations, set the session configuration
spark.databricks.delta.schema.autoMerge.enabled to "true". See the documentation
specific to the operation for details.

Table schema:
root
-- first_name: string (nullable = true)
-- age: long (nullable = true)
```



then in delta we get analysis exception error

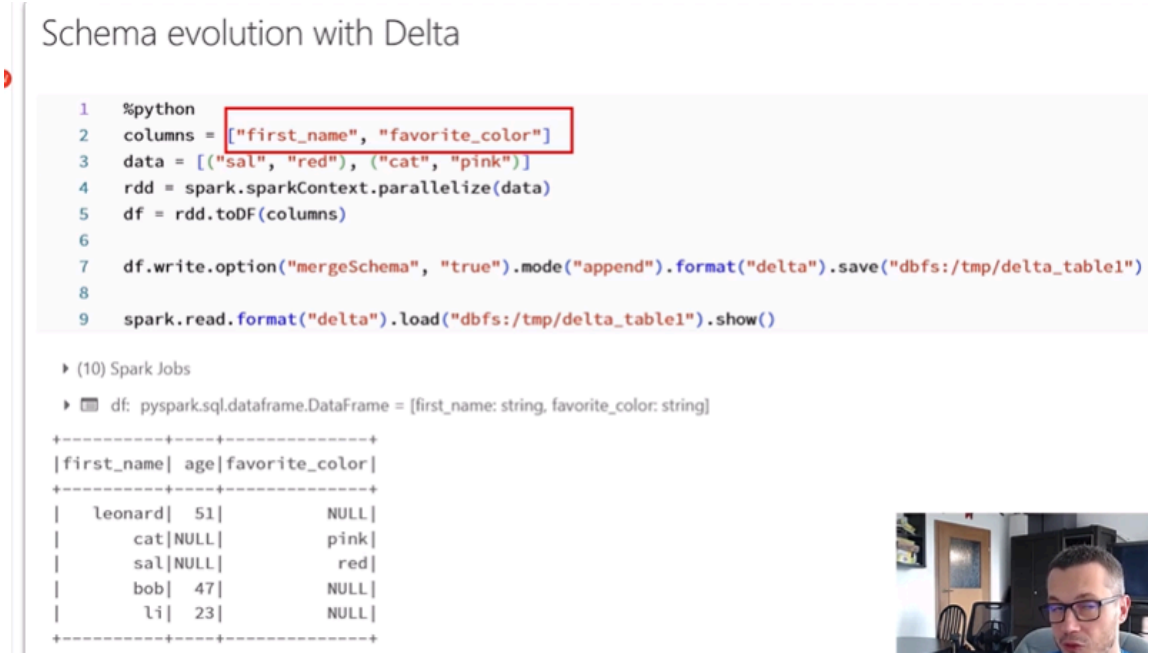
9.

Checks and constraints

1. We can also define constraints like

Schema evolution

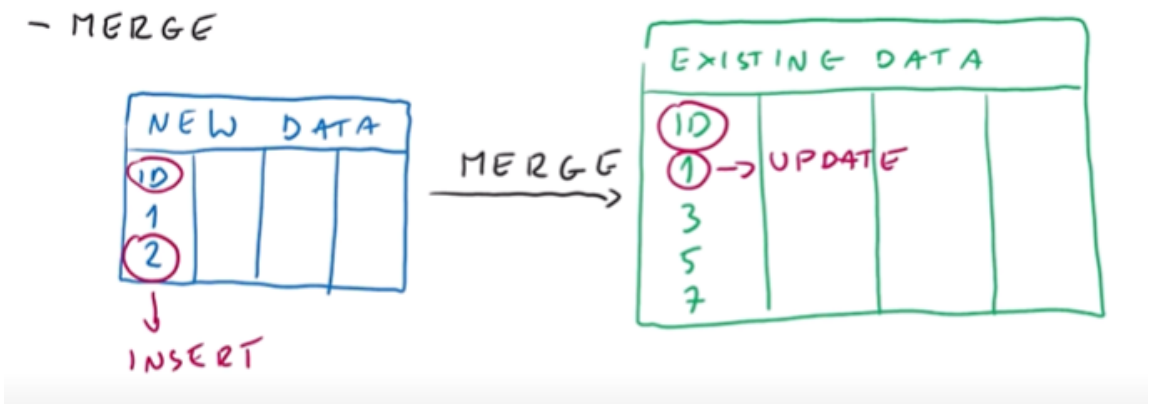
1. Here we can also merge schema



2. Previously we get an error..while we merging different schema's

Merge Operation

1. Here if there's new data..then we can merge it with old data



2. Here if the new data id not present ..then in old data..then it insert the data

Optimize and Z-order

1. Lets suppose we have multiple small delta files in our storage..
2. Then reading this all files or to perform any operation on all files…the time complexity increases to perform for all files
3. So optimize combines these small files into one big file

OPTIMIZE:

- **Purpose:** OPTIMIZE reorganizes data files within a Delta table to improve query performance. It focuses on two main aspects:
  - **File Sizing:** OPTIMIZE aims to create data files with balanced sizes. This can improve efficiency when reading data, as scans don't have to process large variations in file sizes.
  - **Vacuuming:** It can optionally remove unnecessary small files created during data updates. This helps to reduce overall storage footprint and potentially improve some queries.

Example:

Imagine a Delta table storing website clickstream data. Initially, the data might be scattered across many small files due to frequent updates. Running OPTIMIZE on this table would:

1. Analyze the existing data files and their sizes.
2. Coalesce smaller files into larger ones, aiming for a more balanced file size distribution.
3. Optionally, remove very small files that don't contribute significantly to data storage.

4. Z-ordering

Z-Ordering:

- **Purpose:** Z-Ordering rearranges data within files based on specific columns. It sorts rows within each data file based on the chosen column(s). This aims to colocate rows with similar values in the Z-Order columns within the same file segments.

Example:

Consider a Delta table storing user purchase data with columns like `user_id`, `product_id`, and `timestamp`. You frequently query for user purchases based on `user_id`. Z-Ordering this table by `user_id` would sort all purchase records for each user together within the data files.

Benefits:

When a query filters data based on the Z-Order column(s), Delta Lake can leverage data skipping. It can efficiently identify and skip irrelevant data files that don't contain rows matching the query predicate. This significantly reduces the amount of data scanned and improves query performance.