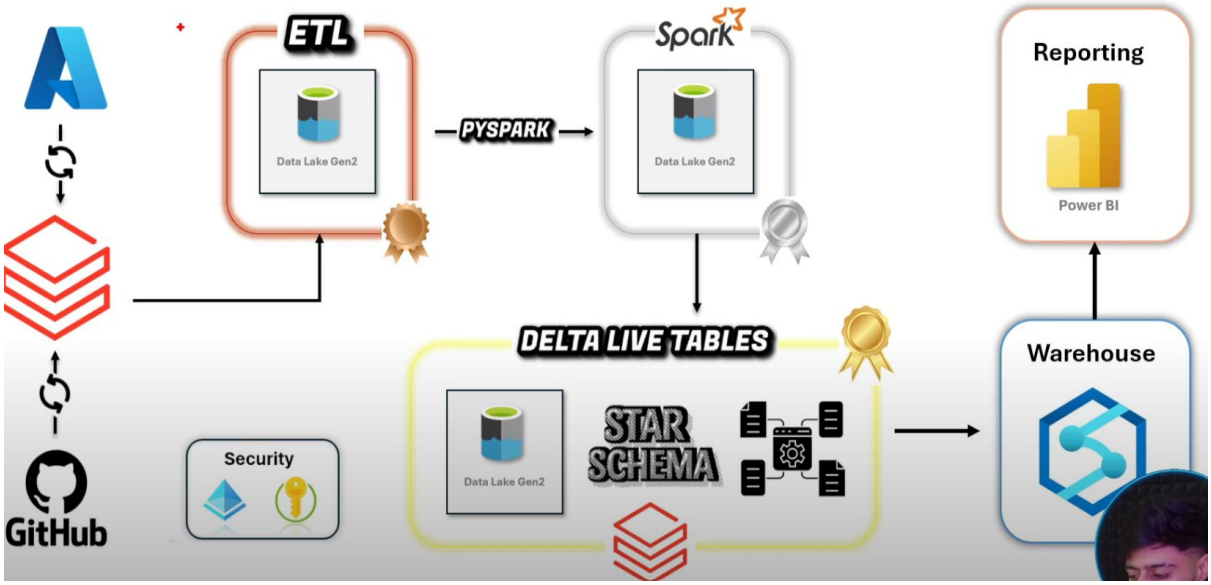
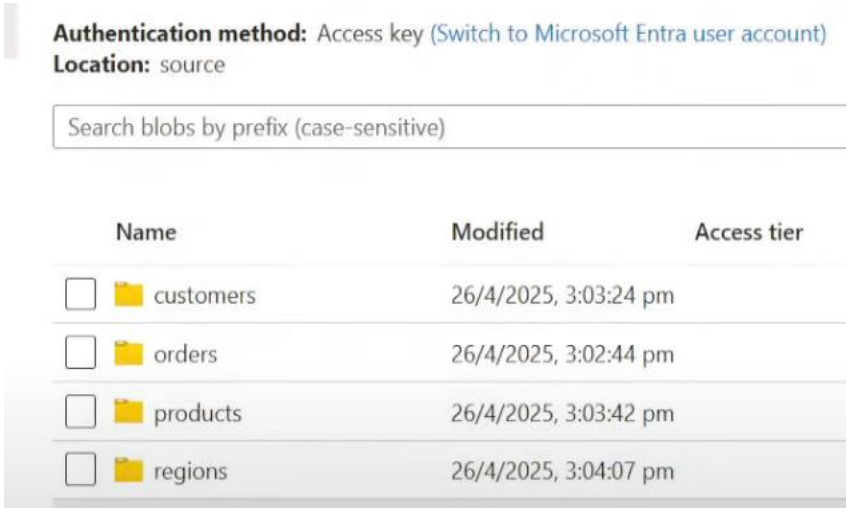
End to End Databricks Project (Retail)



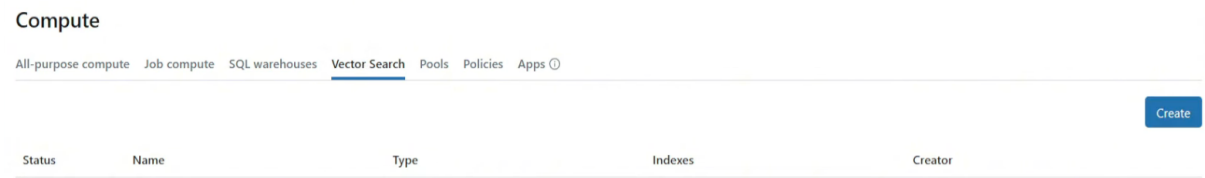
Ingest data from Azure & GitHub, ingest incremental data(including idempotency) in Bronze using Spark Structured Streaming and automated Ingestion Pipeline in ADB.

Benefits of Parquet over csv → Compression, Parquet holds schema at the footer of file

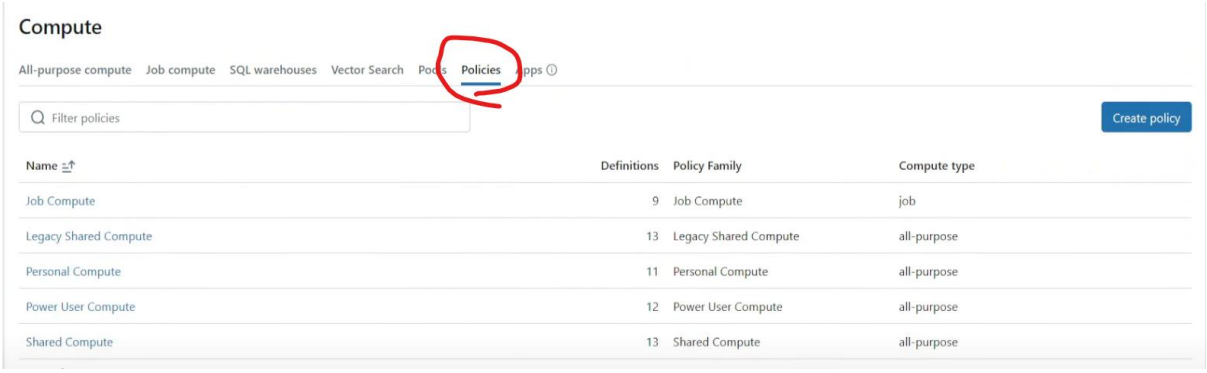
Create 5 containers in Storage account → source, bronze, silver, gold, metastore(for UM)
Upload only first day files in source folder



While creating ADB instance, Managed Resource Group name is not a mandate if we are using UC.
Whenever we create an ADB instance, we will get one UC for free.(Better to use self created ones)

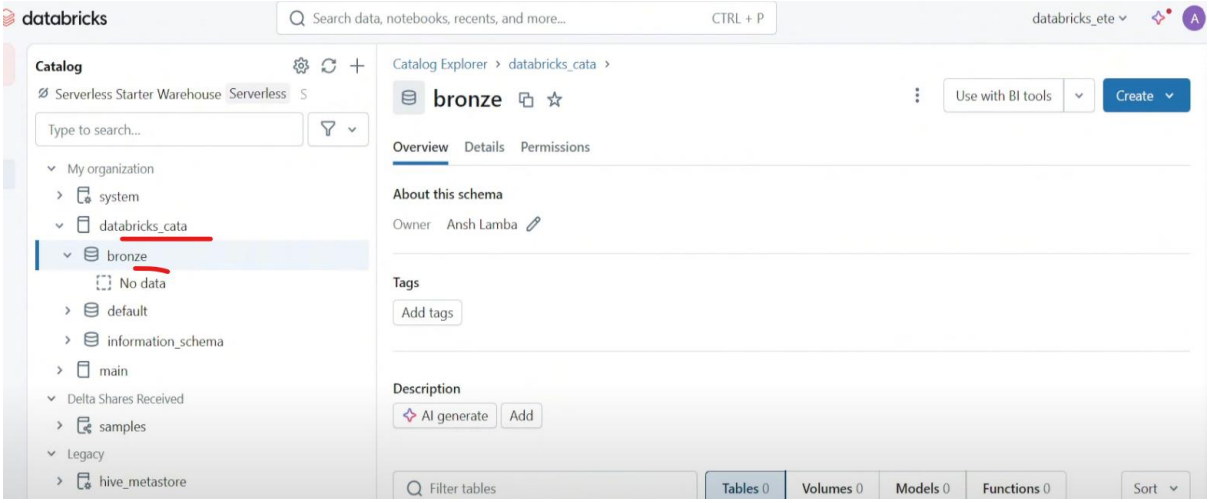


Used by Data Scientists→ translatd into multiple dimensions



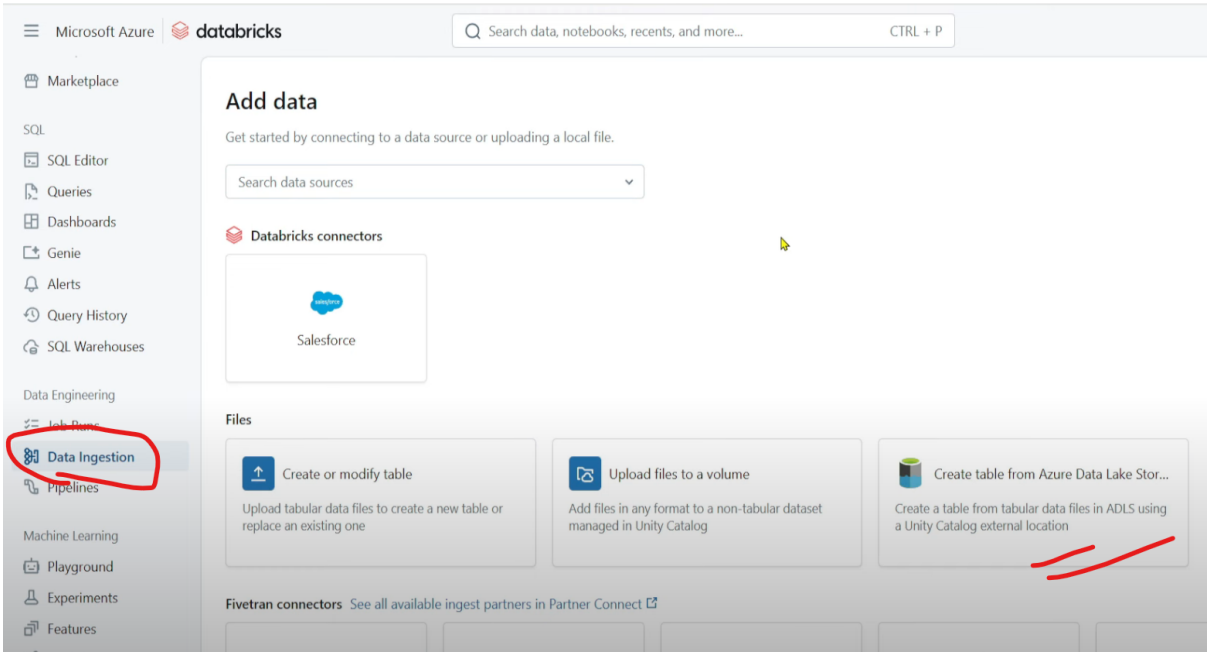
Prebuilt policies

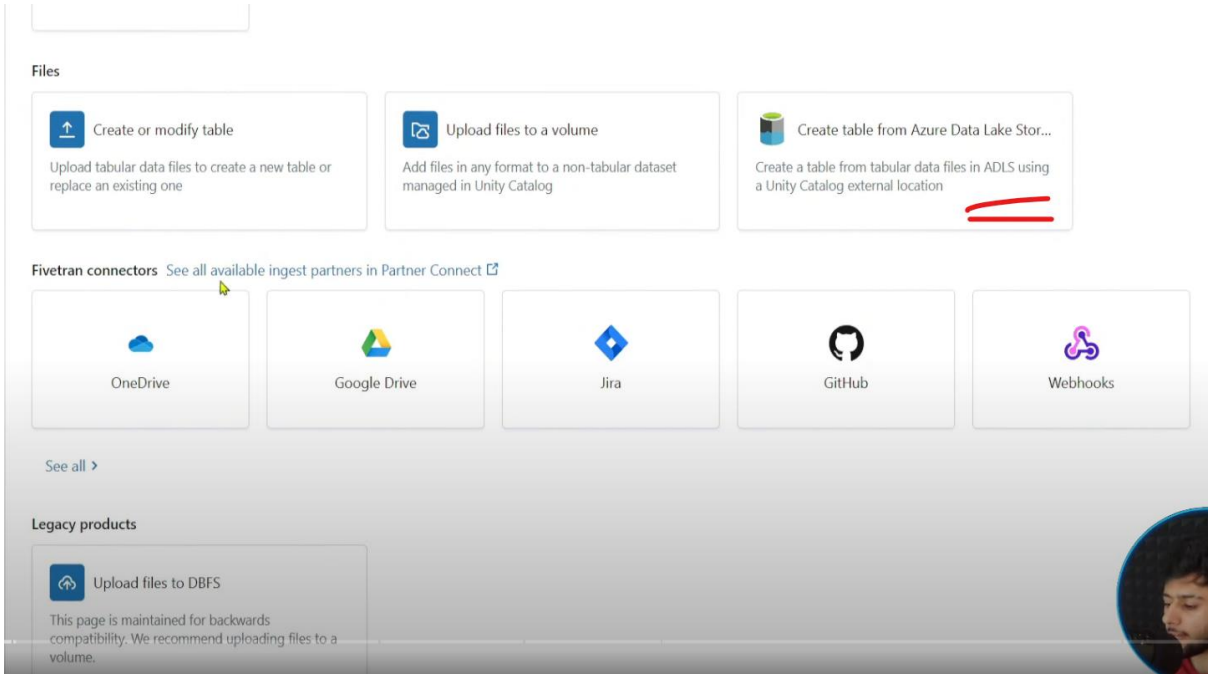
Create a UM, a Catalog, 4 External Locations(source, bronze, silver, gold)
Create bronze schema, will pull raw data over here



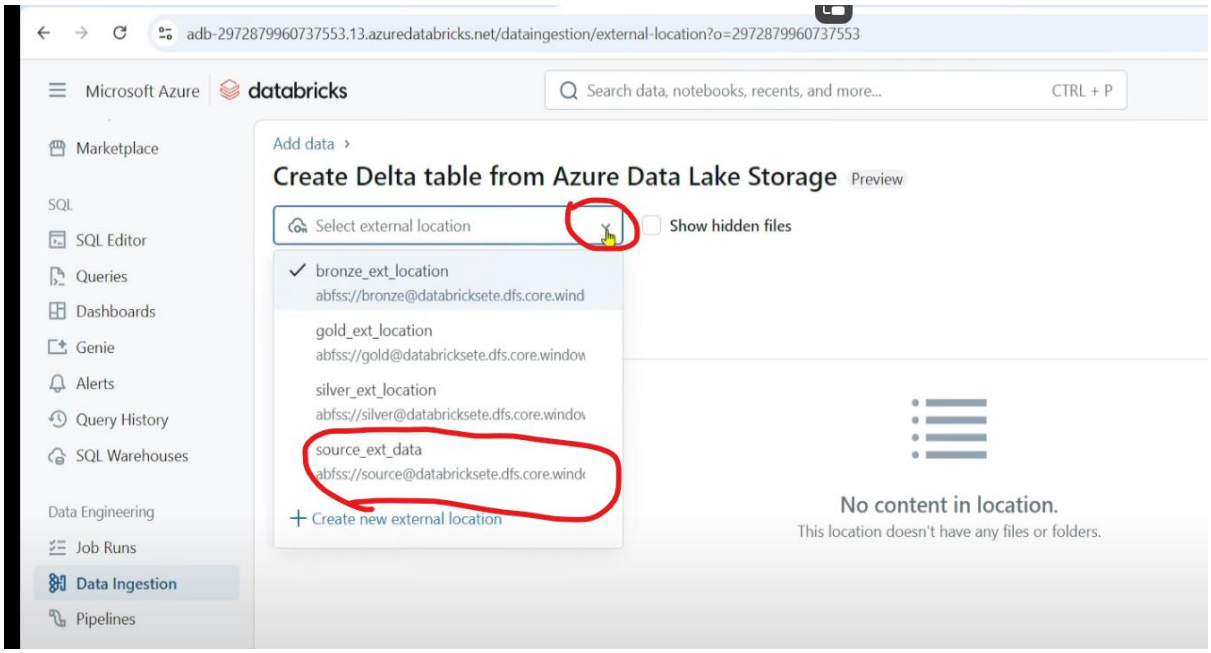
Bronze Ingestion

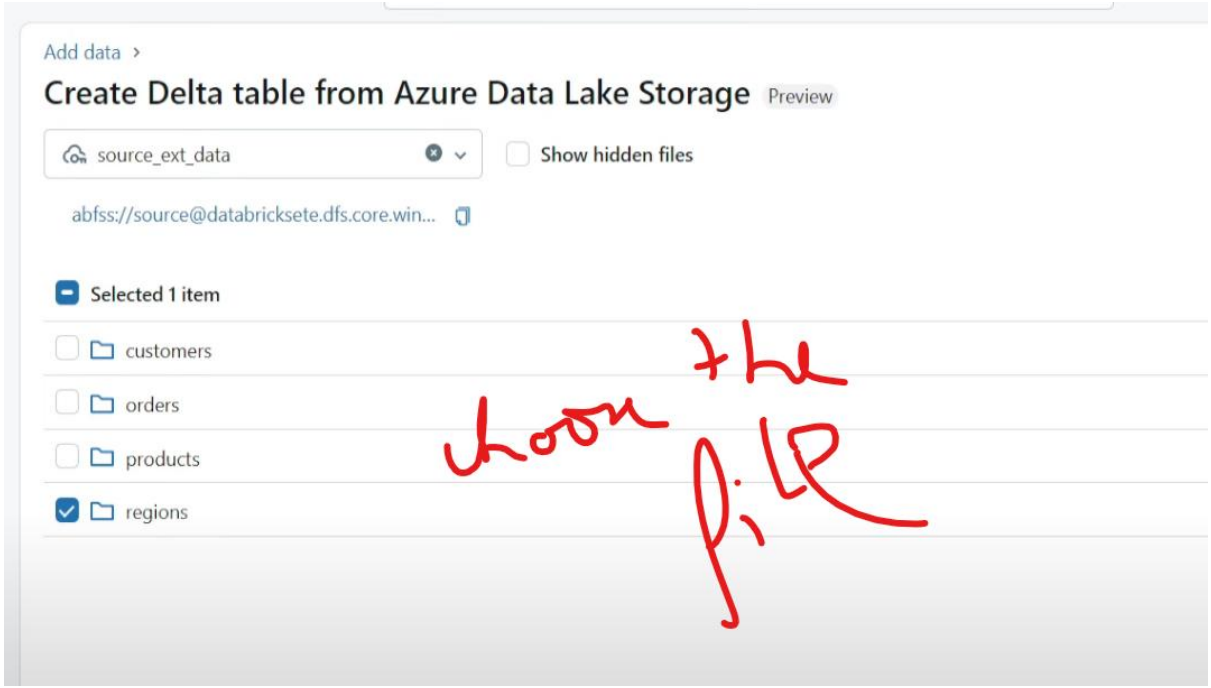
M1: For one time data load/no incremental/static load →Lets use a no code solution to pull data from source to bronze



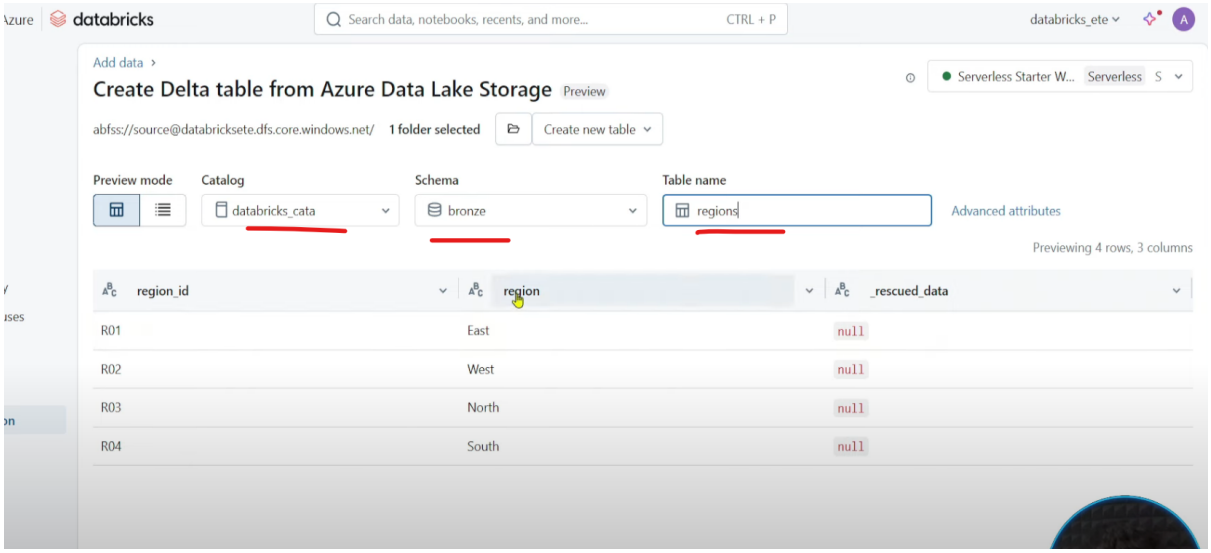


Clicking on that option will simply ingest data from DL and push that data to Managed Location and will create a Table

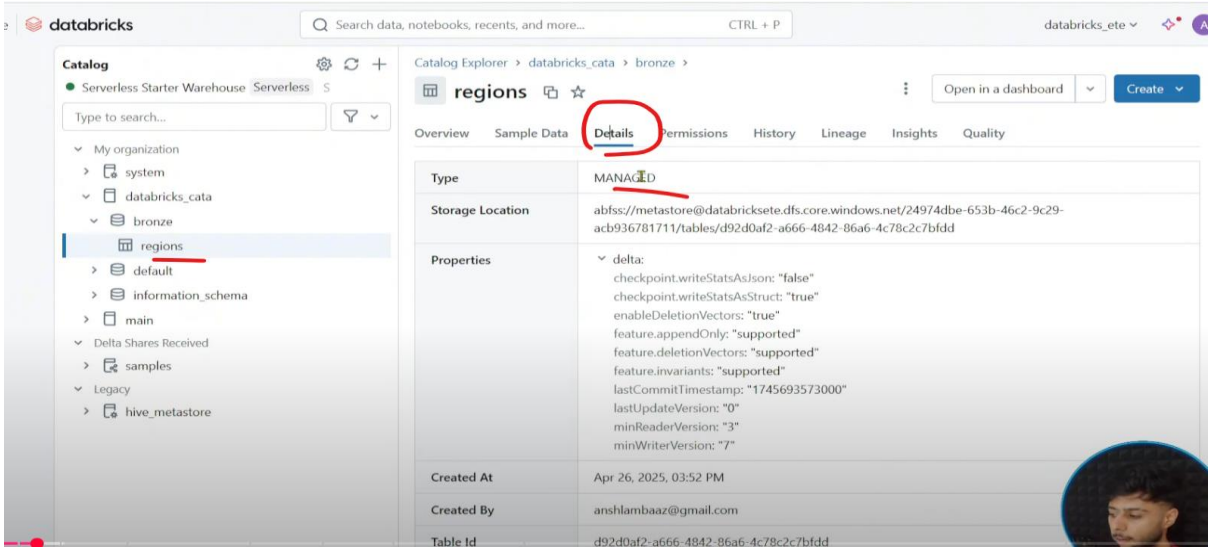


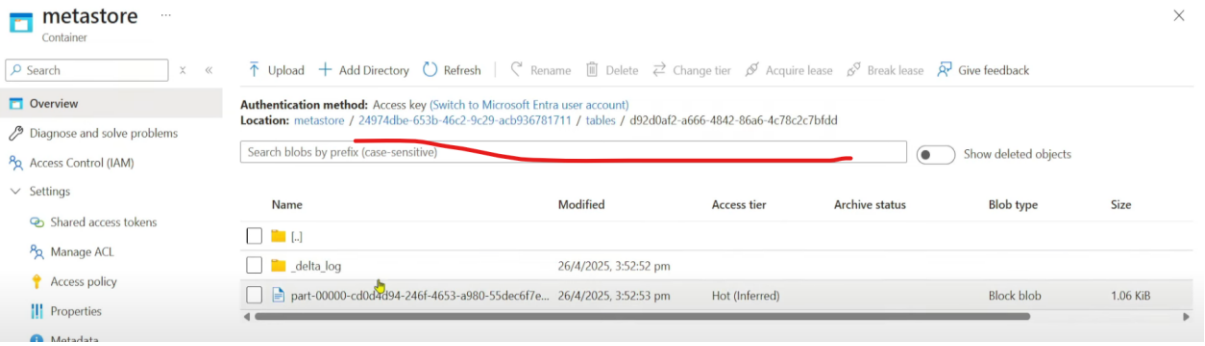


We are choosing regions as it is a static file with no incremental data



rescued data column is for any schema changes in future





M2: for all other files, choose Autoloader & also make it dynamic

Dynamic Capabilities

+ Code + Text

Apr 27, 2025 (<1s) 2

```
dbutils.widgets.text("file_name","")
```

Apr 27, 2025 (<1s) 3

```
p_file_name = dbutils.widgets.get("file_name")
```

Data Reading

Apr 27, 2025 (<1s) 5

```
df = spark.readStream.format("cloudFiles")\
        .option("cloudFiles.format","parquet")\
        .option("cloudFiles.schemaLocation",f"abfss://bronze@databricksete.dfs.core.windows.net/checkpoint_{p_file_name}")\
        .load(f"abfss://source@databricksete.dfs.core.windows.net/{p_file_name}")
```

df: pyspark.sql.dataframe.DataFrame = [order_id: string, customer_id: string ... 5 more fields]

DATA WRITING

Apr 27, 2025 7

```
df.writeStream.format("parquet")\
    .outputMode("append")\
    .option("checkpointLocation",f"abfss://bronze@databricksete.dfs.core.windows.net/checkpoint_{p_file_name}")\
    .option("path",f"abfss://bronze@databricksete.dfs.core.windows.net/{p_file_name}")\
    .trigger(once=True)\
    .start()
```

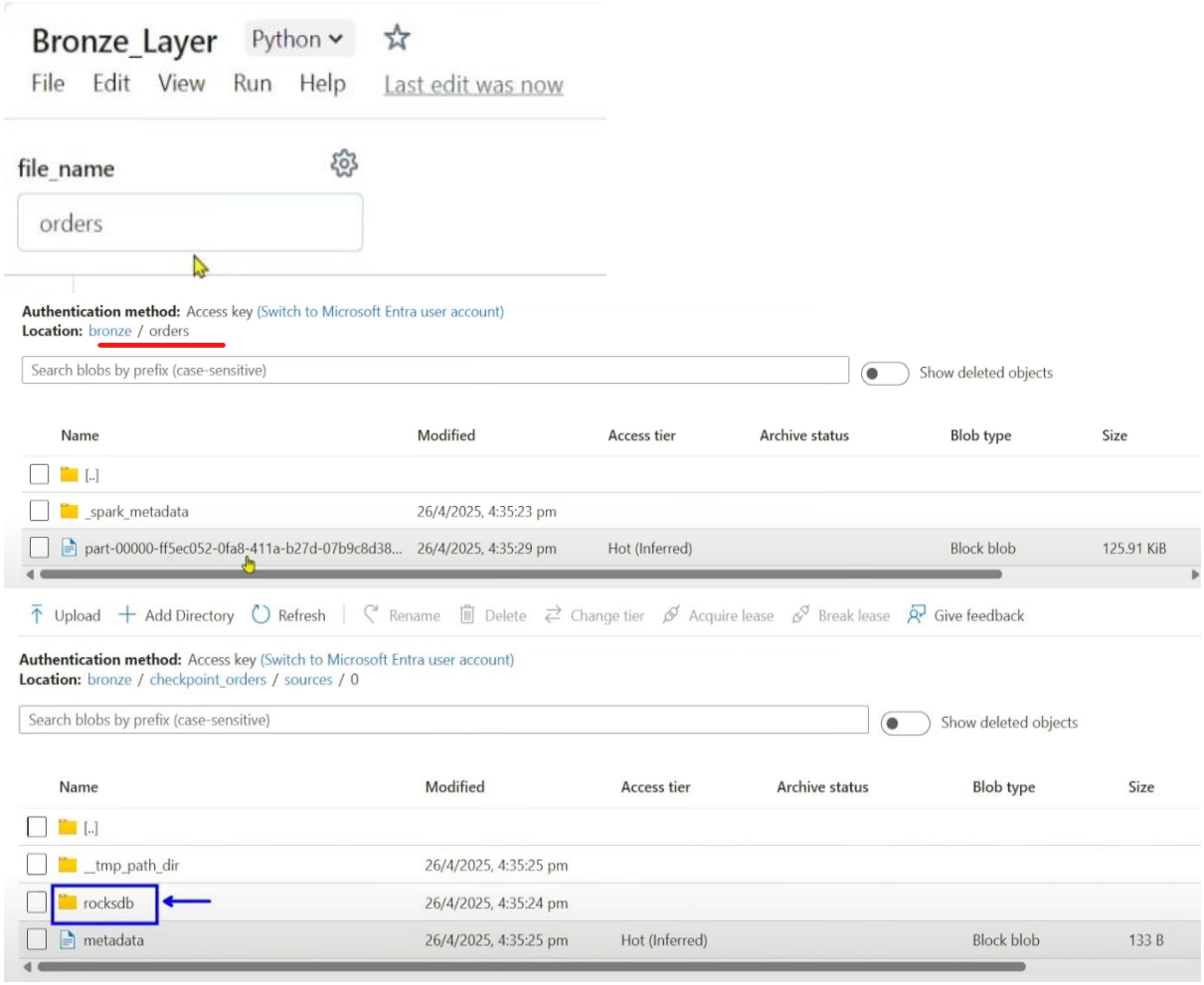
Stream stopped...

```
<pyspark.sql.streaming.query.StreamingQuery at 0x7f3f0c2f8890>
```

Apr 27, 2025 (2s) 8

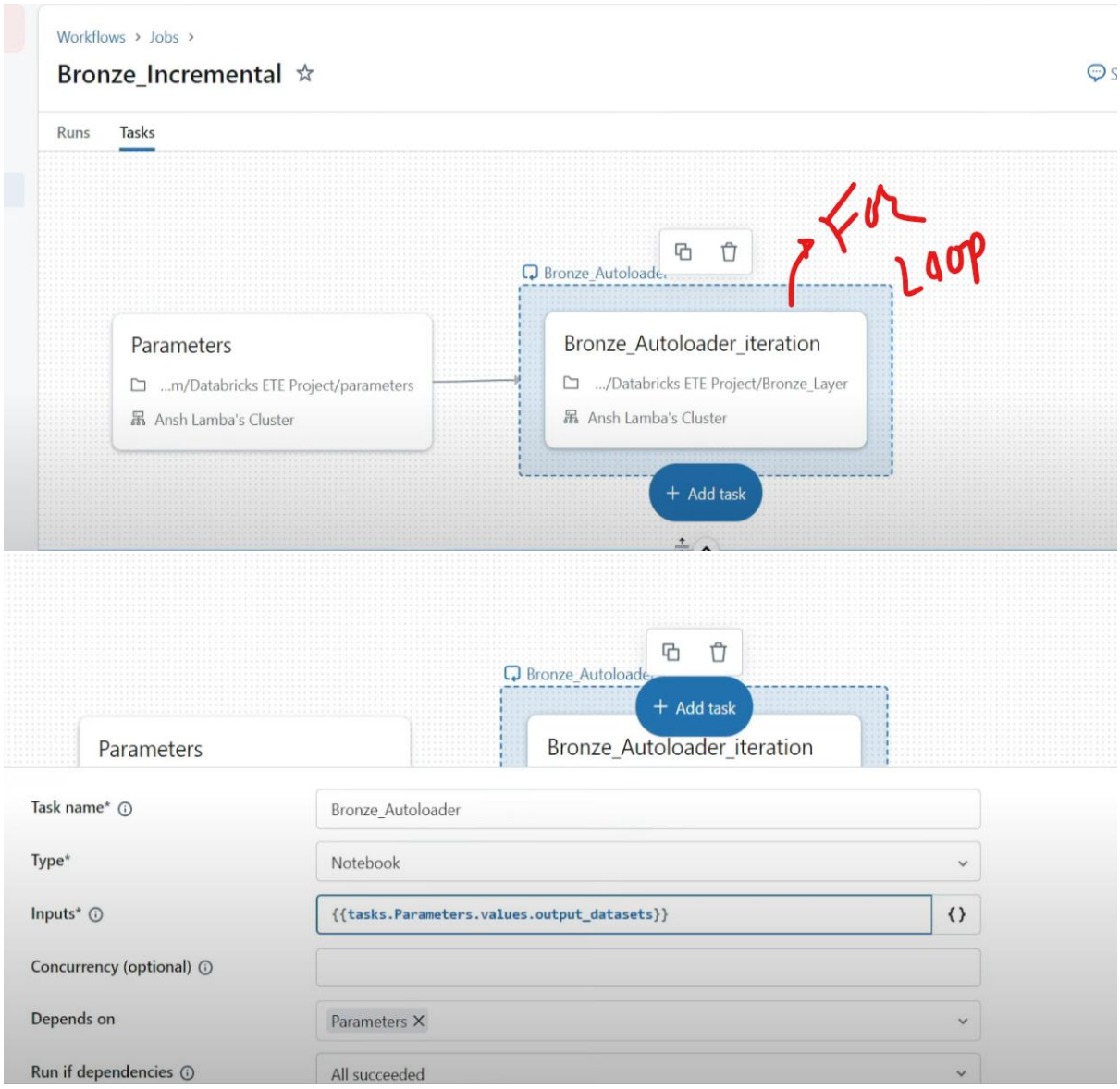
```
df = spark.read.format("parquet")\
        .load(f"abfss://bronze@databricksete.dfs.core.windows.net/{p_file_name}")

display(df)
```

once = True in Triggers → simply read all the files not processed before, perform data loads for those data and immediately stops the Streaming query.

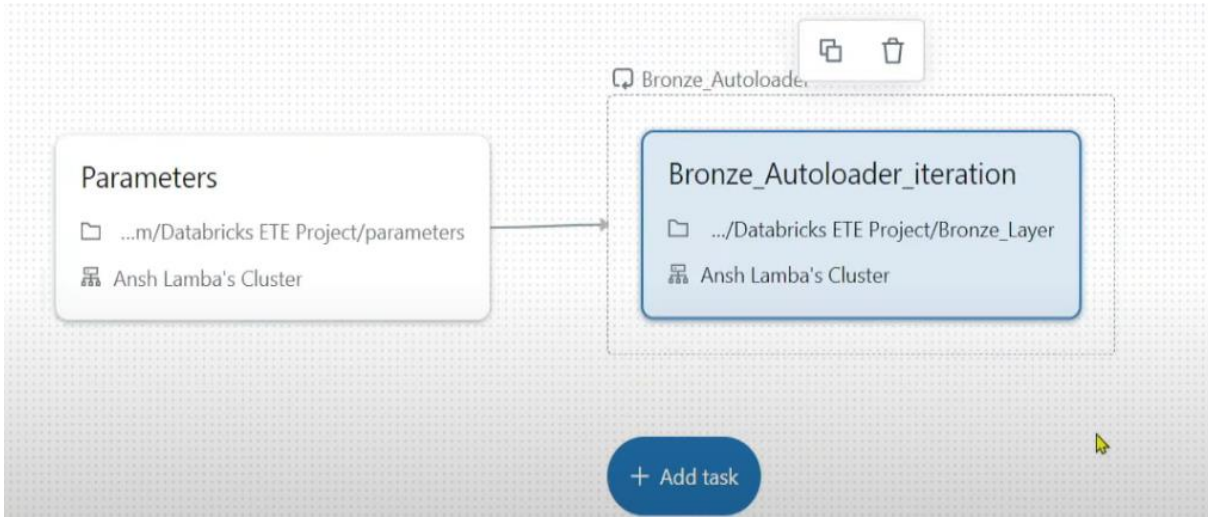


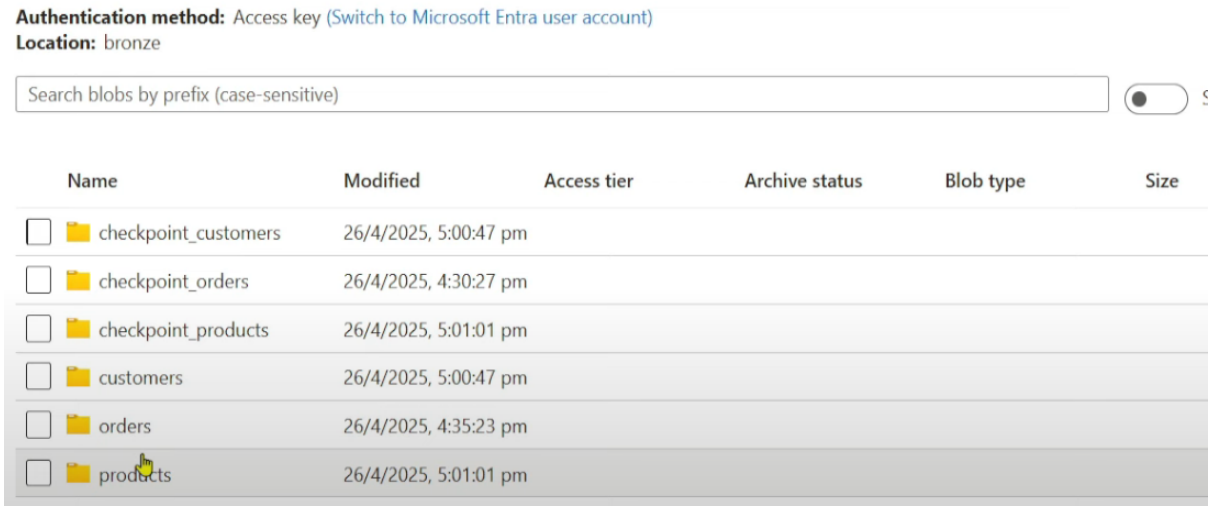
Best way to create a separate Notebook containing file names

Workflow orchestration → Create a job in Workflows

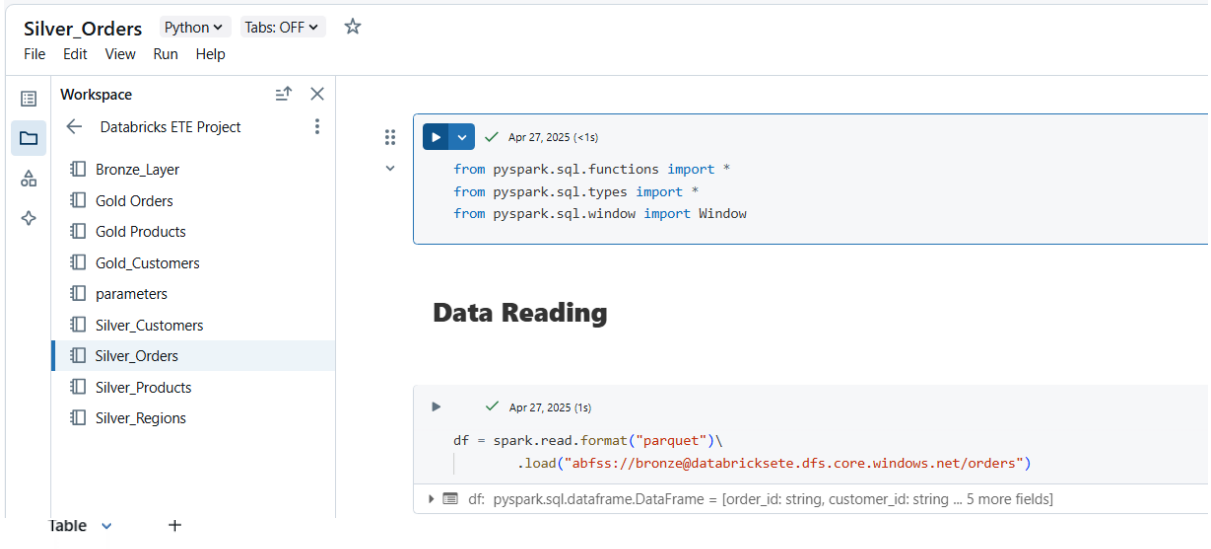


Inner activity





Silver Transformations



	order_id	customer_id	product_id	order_date	quantity	total_amount	_rescued_data
1	O00001	C00710	P0159	2023-03-22	3	2022.87	null
2	O00002	C00954	P0036	2023-06-30	2	3560.74	null
3	O00003	C01578	P0427	2023-11-06	3	5903.52	null
4	O00004	C00962	P0332	2024-02-27	3	4107.99	null
5	O00005	C00156	P0038	2024-10-13	5	5784.95	null
6	O00006	C00521	P0174	2023-05-17	5	407.75	null
7	O00007	C00982	P0352	2024-01-18	4	4907.64	null
8	O00008	C00976	P0172	2023-01-10	4	7037.88	null
9	O00009	C01001	P0238	2023-04-20	3	4076.97	null
10	O00010	C00702	P0258	2023-07-07	4	5695.64	null

Apr 27, 2025 (<1s)

```
df.printSchema()
```

```
root
 |-- order_id: string (nullable = true)
 |-- customer_id: string (nullable = true)
 |-- product_id: string (nullable = true)
 |-- order_date: date (nullable = true)
 |-- quantity: integer (nullable = true)
 |-- total_amount: double (nullable = true)
 |-- _rescued_data: string (nullable = true)
```

Apr 27, 2025 (<1s)

```
df = df.withColumnRenamed("_rescued_data","rescued_data")
```

df: pyspark.sql.dataframe.DataFrame = [order_id: string, customer_id: string ... 5 more fields]

Apr 27, 2025 (<1s)

```
df = df.drop("rescued_data")
df.display()
```

Apr 27, 2025 (<1s) 8

```
df = df.withColumn("order_date",to_timestamp(col('order_date')))
df.display()
```

df: pyspark.sql.dataframe.DataFrame = [order_id: string, customer_id: string ... 4 more fields]

Table +

	order_id	customer_id	product_id	order_date	quantity	total_amount
1	O00001	C00710	P0159	2023-03-22T00:00:00.000+00:...	3	2022.87
2	O00002	C00954	P0036	2023-06-30T00:00:00.000+00:...	2	3560.74
3	O00003	C01578	P0427	2023-11-06T00:00:00.000+00:...	3	5903.52
4	O00004	C00962	P0332	2024-02-27T00:00:00.000+00:...	3	4107.99
5	O00005	C00156	P0038	2024-10-13T00:00:00.000+00:...	5	5784.95

Apr 27, 2025 (<1s) 9

```
df = df.withColumn("year",year(col('order_date')))
df.display()
```

df: pyspark.sql.dataframe.DataFrame = [order_id: string, customer_id: string ... 5 more fields]

Table +

	order_id	customer_id	product_id	order_date	quantity	total_amount	year
1	O00001	C00710	P0159	2023-03-22T00:00:00.000+00:...	3	2022.87	2023
2	O00002	C00954	P0036	2023-06-30T00:00:00.000+00:...	2	3560.74	2023
3	O00003	C01578	P0427	2023-11-06T00:00:00.000+00:...	3	5903.52	2023
4	O00004	C00962	P0332	2024-02-27T00:00:00.000+00:...	3	4107.99	2024
5	O00005	C00156	P0038	2024-10-13T00:00:00.000+00:...	5	5784.95	2024
6	O00006	C00521	P0174	2023-05-17T00:00:00.000+00:...	5	407.75	2023
7	O00007	C00982	P0352	2024-01-18T00:00:00.000+00:...	4	4907.64	2024
8	O00008	C00976	P0172	2023-01-10T00:00:00.000+00:...	4	7037.88	2023

Apr 27, 2025 (1s) 10

```
df1 = df.withColumn("flag",dense_rank().over(Window.partitionBy("year").orderBy(desc("total_amount"))))
df1.display()
```

df1: pyspark.sql.dataframe.DataFrame = [order_id: string, customer_id: string ... 6 more fields]

Table +

	order_id	customer_id	product_id	order_date	quantity	total_amount	year	flag
1	O00957	C01449	P0498	2023-10-05T00:00:00.000+00:...	5	9952.9	2023	1
2	O01765	C01515	P0498	2023-05-11T00:00:00.000+00:...	5	9952.9	2023	1
3	O03502	C00805	P0498	2023-09-24T00:00:00.000+00:...	5	9952.9	2023	1
4	O03660	C01001	P0498	2023-10-08T00:00:00.000+00:...	5	9952.9	2023	1
5	O06790	C01819	P0498	2023-02-27T00:00:00.000+00:...	5	9952.9	2023	1
6	O03989	C00631	P0440	2023-11-20T00:00:00.000+00:...	5	9916.25	2023	2
7	O07763	C00471	P0440	2023-04-12T00:00:00.000+00:...	5	9916.25	2023	2
8	O03272	C01879	P0165	2023-02-15T00:00:00.000+00:...	5	9858.85	2023	3
9	O03746	C01713	P0165	2023-10-26T00:00:00.000+00:...	5	9858.85	2023	3
10	O08261	C00988	P0165	2023-11-29T00:00:00.000+00:...	5	9858.85	2023	3
11	O00412	C00748	P0427	2023-03-23T00:00:00.000+00:...	5	9839.2	2023	4
12	O04478	C01560	P0427	2023-10-04T00:00:00.000+00:...	5	9839.2	2023	4

Apr 27, 2025 (1s) 11

```
df1 = df1.withColumn("rank_flag",rank().over(Window.partitionBy("year").orderBy(desc("total_amount"))))
df1.display()
```

df1: pyspark.sql.dataframe.DataFrame = [order_id: string, customer_id: string ... 7 more fields]

Table +

	order_id	customer_id	product_id	order_date	quantity	total_amount	year	flag	rank_flag
1	O00957	C01449	P0498	2023-10-05T00:00:00.000+00:...	5	9952.9	2023	1	1
2	O01765	C01515	P0498	2023-05-11T00:00:00.000+00:...	5	9952.9	2023	1	1
3	O03502	C00805	P0498	2023-09-24T00:00:00.000+00:...	5	9952.9	2023	1	1
4	O03660	C01001	P0498	2023-10-08T00:00:00.000+00:...	5	9952.9	2023	1	1
5	O06790	C01819	P0498	2023-02-27T00:00:00.000+00:...	5	9952.9	2023	1	1
6	O03989	C00631	P0440	2023-11-20T00:00:00.000+00:...	5	9916.25	2023	2	6
7	O07763	C00471	P0440	2023-04-12T00:00:00.000+00:...	5	9916.25	2023	2	6
8	O03272	C01879	P0165	2023-02-15T00:00:00.000+00:...	5	9858.85	2023	3	8
9	O03746	C01713	P0165	2023-10-26T00:00:00.000+00:...	5	9858.85	2023	3	8
10	O08261	C00988	P0165	2023-11-29T00:00:00.000+00:...	5	9858.85	2023	3	8
11	O00412	C00748	P0427	2023-03-23T00:00:00.000+00:...	5	9839.2	2023	4	11

Apr 27, 2025 (1s) 12

```
df1 = df1.withColumn("row_flag",row_number().over(Window.partitionBy("year").orderBy(desc("total_amount"))))
df1.display()
```

df1: pyspark.sql.dataframe.DataFrame = [order_id: string, customer_id: string ... 8 more fields]

Table +

	order_id	customer_id	product_id	order_date	quantity	total_amount	year	flag	rank_flag	row_flag
1	O00957	C01449	P0498	2023-10-05T00:00:00.000+00:...	5	9952.9	2023	1	1	1
2	O01765	C01515	P0498	2023-05-11T00:00:00.000+00:...	5	9952.9	2023	1	1	2
3	O03502	C00805	P0498	2023-09-24T00:00:00.000+00:...	5	9952.9	2023	1	1	3
4	O03660	C01001	P0498	2023-10-08T00:00:00.000+00:...	5	9952.9	2023	1	1	4
5	O06790	C01819	P0498	2023-02-27T00:00:00.000+00:...	5	9952.9	2023	1	1	5
6	O03989	C00631	P0440	2023-11-20T00:00:00.000+00:...	5	9916.25	2023	2	6	6
7	O07763	C00471	P0440	2023-04-12T00:00:00.000+00:...	5	9916.25	2023	2	6	7
8	O03272	C01879	P0165	2023-02-15T00:00:00.000+00:...	5	9858.85	2023	3	8	8
9	O03746	C01713	P0165	2023-10-26T00:00:00.000+00:...	5	9858.85	2023	3	8	9
10	O08261	C00988	P0165	2023-11-29T00:00:00.000+00:...	5	9858.85	2023	3	8	10

Classes - OOP

Apr 27, 2025 (<1s) 14

```
class windows:

    def dense_rank(self,df):

        df_dense_rank = df.withColumn("flag",dense_rank().over(Window.partitionBy("year").orderBy(desc("total_amount"))))

        return df_dense_rank

    def rank(self,df):

        df_rank = df.withColumn("rank_flag",rank().over(Window.partitionBy("year").orderBy(desc("total_amount"))))

        return df_rank

    def row_number(self,df):

        df_row_number = df.withColumn("row_flag",row_number().over(Window.partitionBy("year").orderBy(desc("total_amount"))))

        return df_row_number
```

Apr 27, 2025 (<1s) 15

```
df_new = df
```

df_new: pyspark.sql.dataframe.DataFrame = [order_id: string, customer_id: string ... 5 more fields]

Apr 27, 2025 (<1s) 16

```
df_new.display()
```

Table +

	order_id	customer_id	product_id	order_date	quantity	total_amount	year
1	O00001	C00710	P0159	2023-03-22T00:00:00.000+00:...	3	2022.87	2023
2	O00002	C00954	P0036	2023-06-30T00:00:00.000+00:...	2	3560.74	2023
3	O00003	C01578	P0427	2023-11-06T00:00:00.000+00:...	3	5903.52	2023
4	O00004	C00962	P0332	2024-02-27T00:00:00.000+00:...	3	4107.99	2024
5	O00005	C00156	P0038	2024-10-13T00:00:00.000+00:...	5	5784.95	2024
6	O00006	C00521	P0174	2023-05-17T00:00:00.000+00:...	5	407.75	2023
7	O00007	C00982	P0352	2024-01-18T00:00:00.000+00:...	4	4907.64	2024

Apr 27, 2025 (<1s) 17

```
obj = windows()
```

Apr 27, 2025 (1s) 18

```
df_result = obj.dense_rank(df_new)
df_result.display()
```

df_result: pyspark.sql.dataframe.DataFrame = [order_id: string, customer_id: string ... 6 more fields]

Table +

	order_id	customer_id	product_id	order_date	quantity	total_amount	year	flag
1	O00957	C01449	P0498	2023-10-05T00:00:00.000+00:...	5	9952.9	2023	1
2	O01765	C01515	P0498	2023-05-11T00:00:00.000+00:...	5	9952.9	2023	1
3	O03502	C00805	P0498	2023-09-24T00:00:00.000+00:...	5	9952.9	2023	1
4	O03660	C01001	P0498	2023-10-08T00:00:00.000+00:...	5	9952.9	2023	1
5	O06790	C01819	P0498	2023-02-27T00:00:00.000+00:...	5	9952.9	2023	1
6	O03989	C00631	P0440	2023-11-20T00:00:00.000+00:...	5	9916.25	2023	2
7	O07763	C00471	P0440	2023-04-12T00:00:00.000+00:...	5	9916.25	2023	2
8	O03272	C01879	P0165	2023-02-15T00:00:00.000+00:...	5	9858.85	2023	3
9	O03746	C01713	P0165	2023-10-26T00:00:00.000+00:...	5	9858.85	2023	3
10	O08261	C00988	P0165	2023-11-29T00:00:00.000+00:...	5	9858.85	2023	3
11	O00412	C00748	P0427	2023-03-23T00:00:00.000+00:...	5	9839.2	2023	4
12	O04478	C01560	P0427	2023-10-04T00:00:00.000+00:...	5	9839.2	2023	4
13	O05528	C01500	P0427	2023-02-02T00:00:00.000+00:...	5	9839.2	2023	4

Data Writing

Apr 27, 2025 (11s) 20

```
df.write.format("delta").mode("overwrite").save("abfss://silver@databricksete.dfs.core.windows.net/orders")
```

Apr 27, 2025 (3s) 21

```
%sql
CREATE TABLE IF NOT EXISTS databricks_cata.silver.orders_silver
USING DELTA
LOCATION 'abfss://silver@databricksete.dfs.core.windows.net/orders'
```

OK

Waiting

```
%sql
CREATE SCHEMA databricks_cata.silver
```

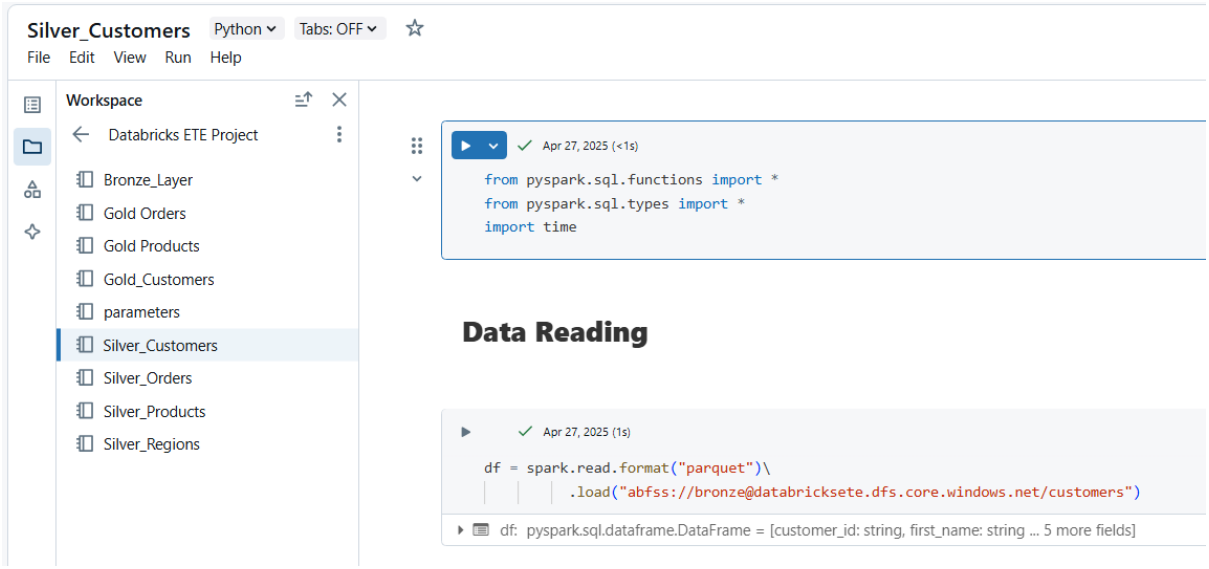
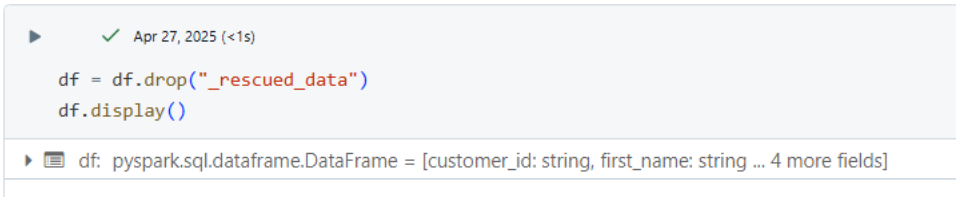


Table +

	customer_id	first_name	last_name	email	city	state	_rescued_data
1	C00001	Emily	Mooney	rushjeff@ryan.org	Johnsonmouth	MS	null
2	C00002	Andrea	Sellers	mccoykiara@kelly.com	Stephenfort	WY	null
3	C00003	Craig	Hayes	rebeccamiller@yahoo.com	South Stephenshire	LA	null
4	C00004	Bryan	Scott	lawrence05@campbell.info	Chrisland	ND	null
5	C00005	Sean	Vasquez	carrie45@yahoo.com	East Dennistown	RI	null
6	C00006	Kevin	Mccarthy	traceyramos@gmail.com	North Matthew	IN	null
7	C00007	Amanda	Doyle	scottallen@gmail.com	Joneshaven	VA	null
8	C00008	Paul	Campos	sullivanjeremy@horton-adams.com	South Nathanfurt	CT	null
9	C00009	Mary	Green	dennis03@yahoo.com	Kimberlyview	MD	null
10	C00010	James	Myers	charles58@murillo.net	West Hector	OK	null



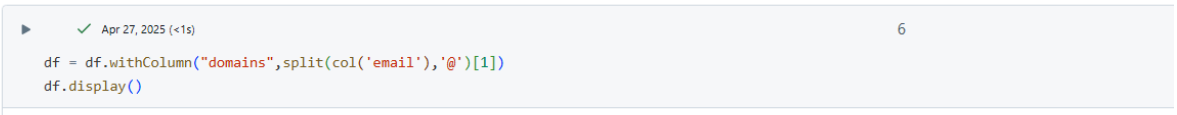


Table +

	customer_id	first_name	last_name	email	city	state	domains
1	C00001	Emily	Mooney	rushjeff@ryan.org	Johnsonmouth	MS	ryan.org
2	C00002	Andrea	Sellers	mccoykiara@kelly.com	Stephenfort	WY	kelly.com
3	C00003	Craig	Hayes	rebeccamiller@yahoo.com	South Stephenshire	LA	yahoo.com
4	C00004	Bryan	Scott	lawrence05@campbell.info	Chrisland	ND	campbell.info
5	C00005	Sean	Vasquez	carrie45@yahoo.com	East Dennistown	RI	yahoo.com

+ Code + Text

Apr 27, 2025 (1s) 7

```
df.groupBy("domains").agg(count("customer_id").alias("total_customers")).sort("total_customers",ascending=False).display()
```

Table +

	domains	total_customers
1	gmail.com	374
2	hotmail.com	360
3	yahoo.com	331
4	brown.com	8
5	davis.com	8
6	smith.com	7
7	hernandez.com	5
8	johnson.com	5

```
df_gmail = df.filter(col('domains')=="gmail.com")
df_gmail.display()
time.sleep(5)

df_yahoo = df.filter(col('domains')=="yahoo.com")
df_yahoo.display()
time.sleep(5)

df_hotmail = df.filter(col('domains')=="hotmail.com")
df_hotmail.display()
time.sleep(5)
```

Apr 27, 2025 (<1s) 9

```
df = df.withColumn("full_name",concat(col('first_name'),lit(' '),col('last_name')))
df = df.drop('first_name','last_name')

df.display()
```

df: pyspark.sql.dataframe.DataFrame = [customer_id: string, email: string ... 4 more fields]

Table +

	customer_id	email	city	state	domains	full_name
1	C00001	rushjeff@ryan.org	Johnsonmouth	MS	ryan.org	Emily Mooney
2	C00002	mccoykiara@kelly.com	Stephenfort	WY	kelly.com	Andrea Sellers
3	C00003	rebeccamiller@yahoo.com	South Stephenshire	LA	yahoo.com	Craig Hayes
4	C00004	lawrence05@campbell.info	Chrisland	ND	campbell.info	Bryan Scott
5	C00005	carrie45@yahoo.com	East Dennistown	RI	yahoo.com	Sean Vasquez
6	C00006	traceyramos@gmail.com	North Matthew	IN	gmail.com	Kevin Mccarthy
7	C00007	scottallen@gmail.com	Joneshaven	VA	gmail.com	Amanda Doyle
8	C00008	sullivanjeremy@horton-adams.com	South Nathanfurt	CT	horton-adams.com	Paul Campos

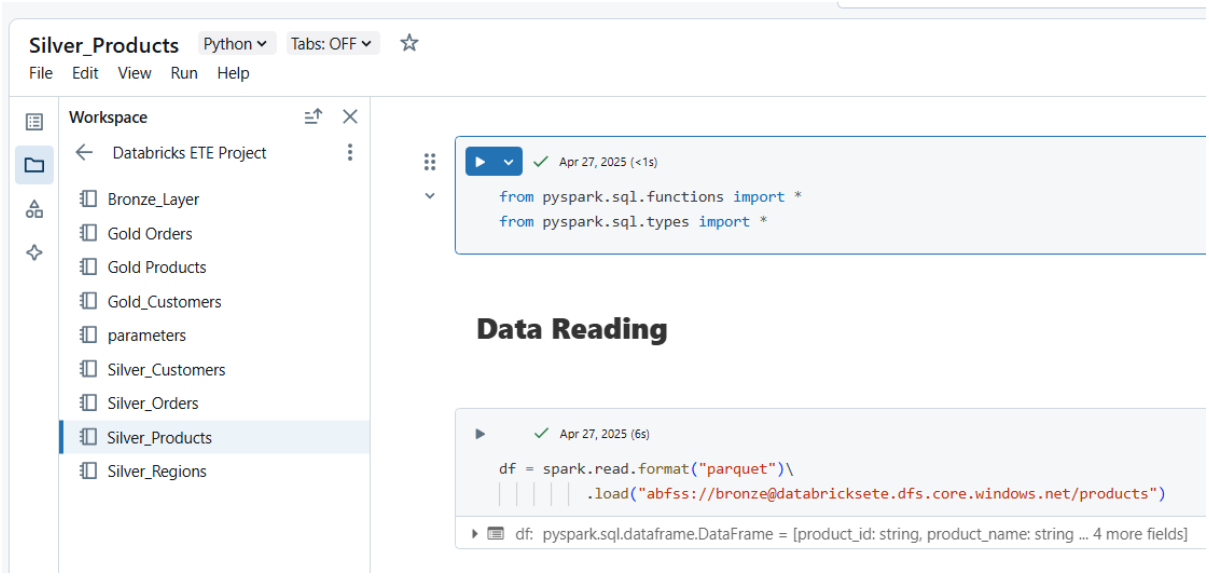
Apr 27, 2025 (5s) 10

```
df.write.mode("overwrite").format("delta").save("abfss://silver@databricksete.dfs.core.windows.net/customers")
```

Apr 27, 2025 (<1s) 11

```
%sql
CREATE TABLE IF NOT EXISTS databricks_cata.silver.customers_silver
USING DELTA
LOCATION 'abfss://silver@databricksete.dfs.core.windows.net/customers'
```

OK



Table

	product_id	product_name	category	brand	price	_rescued_data
1	P0001	Clearly Its	Beauty	Nike	1868.54	null
2	P0002	Production Clear	Beauty	Apple	587.13	null
3	P0003	Culture Coach	Home	Revlon	1599.24	null
4	P0004	Movement Part	Sports	LG	651.71	null
5	P0005	Fact Name	Clothing	Samsung	1861.78	null
6	P0006	Usually Stop	Toys	Adidas	936.36	null
7	P0007	Reveal Current	Sports	Adidas	1954.02	null
8	P0008	Force Language	Beauty	Puma	1251.26	null
9	P0009	Stage Leg	Clothing	Samsung	1247.15	null
10	P0010	Leader Then	Sports	Sony	975.53	null

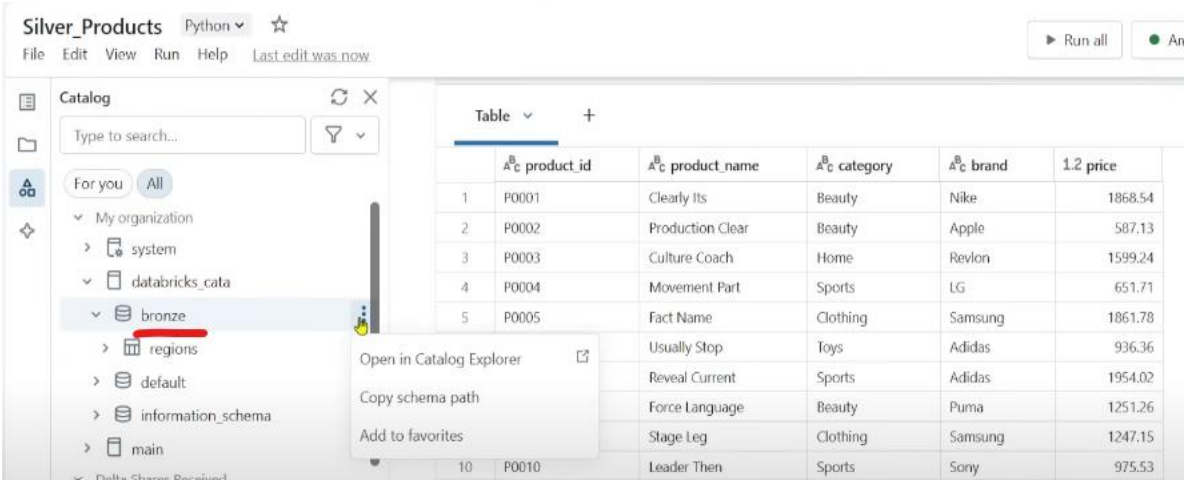
Apr 27, 2025 (<1s)

```
df = df.drop("_rescued_data")
```

df: pyspark.sql.dataframe.DataFrame = [product_id: string, product_name: string ... 3 more fields]

Apr 27, 2025 (<1s)

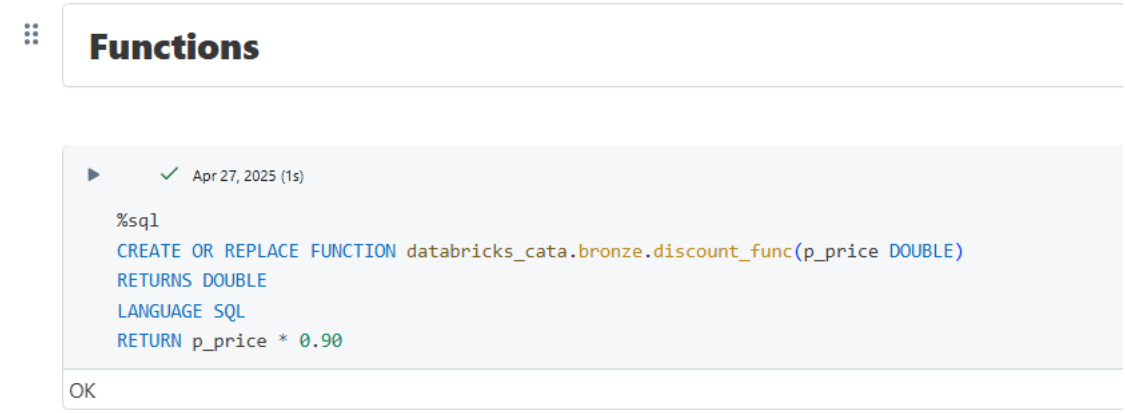
```
df.createOrReplaceTempView("products")
```

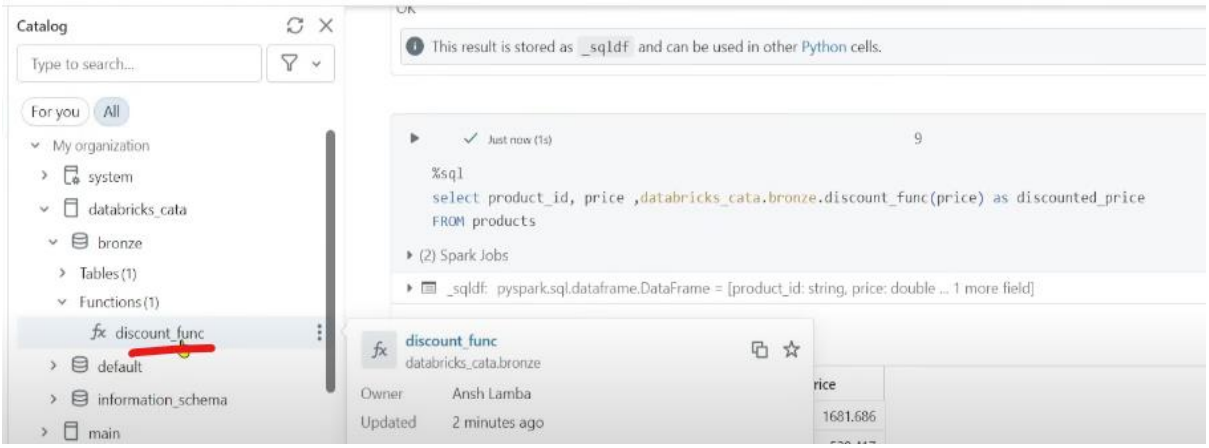


Functions can be created inside Schema, we can reuse it (UDF is only till program level).

2 Types → 1. Scalar Function 2. Table Function

SQL way





Apr 27, 2025 (1s)

```
%sql
select product_id, price ,databricks_cata.bronze.discount_func(price) as discounted_price
FROM products
```

_sqldf: pyspark.sql.dataframe.DataFrame = [product_id: string, price: double ... 1 more field]

Table +

	product_id	price	discounted_price
1	P0001	1868.54	1681.686
2	P0002	587.13	528.417
3	P0003	1599.24	1439.316
4	P0004	651.71	586.5390000000001
5	P0005	1861.78	1675.602
6	P0006	936.36	842.724
7	P0007	1954.02	1758.618
8	P0008	1251.26	1126.134
9	P0009	1247.15	1122.4350000000002

Apr 27, 2025 (1s) 10

```
df = df.withColumn("discounted_price",expr("databricks_cata.bronze.discount_func(price)"))
df.display()
```

df: pyspark.sql.dataframe.DataFrame = [product_id: string, product_name: string ... 4 more fields]

Table +

	product_id	product_name	category	brand	price	discounted_price
1	P0001	Clearly Its	Beauty	Nike	1868.54	1681.686
2	P0002	Production Clear	Beauty	Apple	587.13	528.417
3	P0003	Culture Coach	Home	Revlon	1599.24	1439.316
4	P0004	Movement Part	Sports	LG	651.71	586.5390000000001
5	P0005	Fact Name	Clothing	Samsung	1861.78	1675.602
6	P0006	Usually Stop	Toys	Adidas	936.36	842.724
7	P0007	Reveal Current	Sports	Adidas	1954.02	1758.618
8	P0008	Force Language	Beauty	Puma	1251.26	1126.134
9	P0009	Stage Leg	Clothing	Samsung	1247.15	1122.4350000000002

expr is used whenever we want to use sql functions

Python way

Apr 27, 2025 (1s)

```
%sql
CREATE OR REPLACE FUNCTION databricks_cata.bronze.upper_func(p_brand STRING)
RETURNS STRING
LANGUAGE PYTHON
AS
$$
    return p_brand.upper()
$$
```

OK

Apr 27, 2025 (14s)

```
%sql
SELECT product_id, brand, databricks_cata.bronze.upper_func(brand) as brand_upper
FROM products
```

_sqldf: pyspark.sql.dataframe.DataFrame = [product_id: string, brand: string ... 1 more field]

Table +

	product_id	brand	brand_upper
17	P0017	Revlon	REVLON
18	P0018	Puma	PUMA
19	P0019	Samsung	SAMSUNG
20	P0020	Nike	NIKE
21	P0021	Nike	NIKE
22	P0022	Puma	PUMA
23	P0023	Adidas	ADIDAS
24	P0024	Samsung	SAMSUNG
25	P0025	Nike	NIKE

Apr 27, 2025 (12s)

```
df.write.format("delta")\
        .mode("overwrite")\
        .option("path","abfss://silver@databricksete.dfs.core.windows.net/products")\
        .save()
```

Apr 27, 2025 (2s)

```
%sql
CREATE TABLE IF NOT EXISTS databricks_cata.silver.products_silver
USING DELTA
LOCATION 'abfss://silver@databricksete.dfs.core.windows.net/products'
```

OK

2 days ago (8s)

```
%sql
CREATE OR REPLACE FUNCTION databricks_cata.bronze.tablefunc(p_year DOUBLE)
RETURNS TABLE(order_id STRING, year DOUBLE)
LANGUAGE SQL
RETURN
( SELECT order_id, year FROM databricks_cata.gold.factorders
  WHERE year = p_year )
```

2 days ago (9s)

```
%sql
SELECT * FROM databricks_cata.bronze.tablefunc(2024)
```

_sqldf: pyspark.sql.connect.dataframe.DataFrame = [order_id: string, year: double]

Table

	order_id	year
28	O00045	2024
29	O00049	2024
30	O00050	2024
31	O00051	2024
32	O00052	2024
33	O00055	2024
34	O00059	2024
35	O00066	2024

Silver_Regions Python Tabs: OFF

File Edit View Run Help

Workspace

Databricks ETE Project

- Bronze_Layer
- Gold Orders
- Gold Products
- Gold_Customers
- parameters
- Silver_Customers
- Silver_Orders
- Silver_Products
- Silver_Regions

Apr 27, 2025 (1s) 1

```
df = spark.read.table("databricks_cata.bronze.regions")
```

df: pyspark.sql.dataframe.DataFrame = [region_id: string, region: string ... 1 more field]

Apr 27, 2025 (1s) 2

```
df.display()
```

Table

	product_id	product_name	category	brand	price	discounted_price
1	P0001	Clearly Its	Beauty	Nike	1868.54	1681.686
2	P0002	Production Clear	Beauty	Apple	587.13	528.417
3	P0003	Culture Coach	Home	Revlon	1599.24	1439.316
4	P0004	Movement Part	Sports	LG	651.71	586.5390000000001
5	P0005	Fact Name	Clothing	Samsung	1861.78	1675.602
6	P0006	Usually Stop	Toys	Adidas	936.36	842.724
7	P0007	Reveal Current	Sports	Adidas	1954.02	1758.618
8	P0008	Force Language	Beauty	Puma	1251.26	1126.134
9	P0009	Stage Leg	Clothing	Samsung	1247.15	1122.4350000000002
10	P0010	Leader Then	Sports	Sony	975.53	877.977
11	P0011	Term Rest	Electronics	Adidas	1475.31	1327.779
12	P0012	Theory Wrong	Electronics	Apple	624.61	562.149
13	P0013	Democrat Book	Sports	Puma	384.99	346.49100000000004
14	P0014	Follow Brother	Clothing	Dell	32.2	28.980000000000004
15	P0015	Pattern Story	Beauty	LG	1459.2	1313.28

500 rows | 1.12s runtime

```
Apr 27, 2025 (<1s)
df = df.drop("_rescued_data")
df: pyspark.sql.dataframe.DataFrame = [region_id: string, region: string]
```

```
Apr 27, 2025 (4s)
df.write.format("delta")\
        .mode("overwrite")\
        .save("abfss://silver@databricksete.dfs.core.windows.net/regions")
```

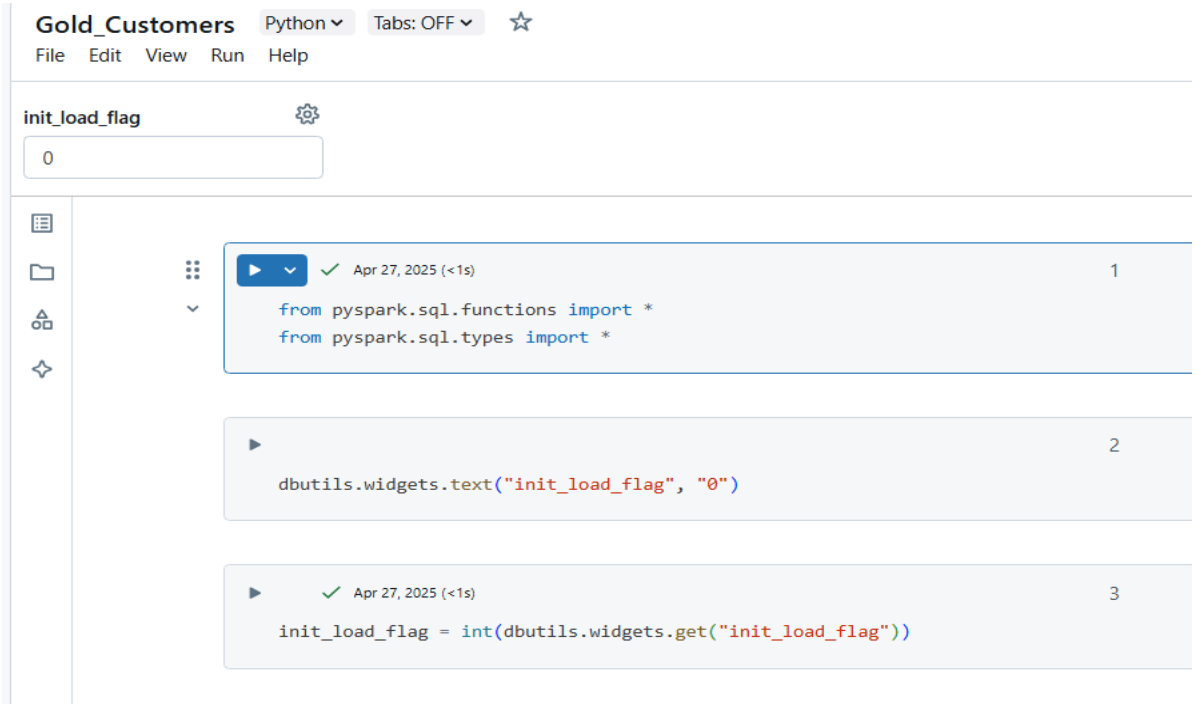
```
Apr 27, 2025 (1s)
%sql
CREATE TABLE IF NOT EXISTS databricks_cata.silver.regions_silver
USING DELTA
LOCATION 'abfss://silver@databricksete.dfs.core.windows.net/regions'
OK
```

Gold Layer

Create a gold schema from UI (go to Catalogs→ Create schema)

We will prepare SCD Type 1 in such a way that it will take care of historical/initial/Full Load as well as Incremental Load

init_load_flag =1 for initial load, for rest loads keep init_load_flag =0 (init_load_flag == 0 means incremental load)



Data Reading From Source

```
Apr 27, 2025 (2s)                                                    5
df = spark.sql("select * from databricks_cata.silver.customers_silver")
```

df: pyspark.sql.connect.dataframe.DataFrame = [customer_id: string, email: string ... 4 more fields]

Removing Duplicates

```
Apr 27, 2025 (<1s)                                                   7
df = df.dropDuplicates(subset=['customer_id'])
```

df: pyspark.sql.connect.dataframe.DataFrame = [customer_id: string, email: string ... 4 more fields]

	customer_id	email	city	state	domains	full_name
1	C01220	matthew01@yahoo.com	New Andrewhav...	WA	yahoo.com	Thomas Fitzgerald
2	C01579	nathancastro@gmail.com	South Amanda	FL	gmail.com	Brittany Schmidt
3	C01155	cynthia51@lewis-dixon.com	East Theresa	FL	lewis-dixon.com	Latasha Phelps
4	C01943	rodriguezzachary@hotmail.co...	East Timothy	PA	hotmail.com	David Cooper
5	C01554	sthompson@harding.com	New Leefurt	ME	harding.com	Gavin Lindsey
6	C00767	rsmith@pruitt-hodges.net	Heatherton	IL	pruitt-hodges.net	Pam Watts
7	C01097	joel22@stone-holmes.com	Davidhaven	LA	stone-holmes.com	James Martinez
8	C01674	crystalraymond@keller.com	Hoborough	ME	keller.com	Nicole Johnston
9	C00215	petersonthomas@yahoo.com	South Andrea	OK	yahoo.com	Danielle Huerta
10	C00587	phillipsstephanie@gmail.com	New Ronaldmouth	OR	gmail.com	Michael Bailey

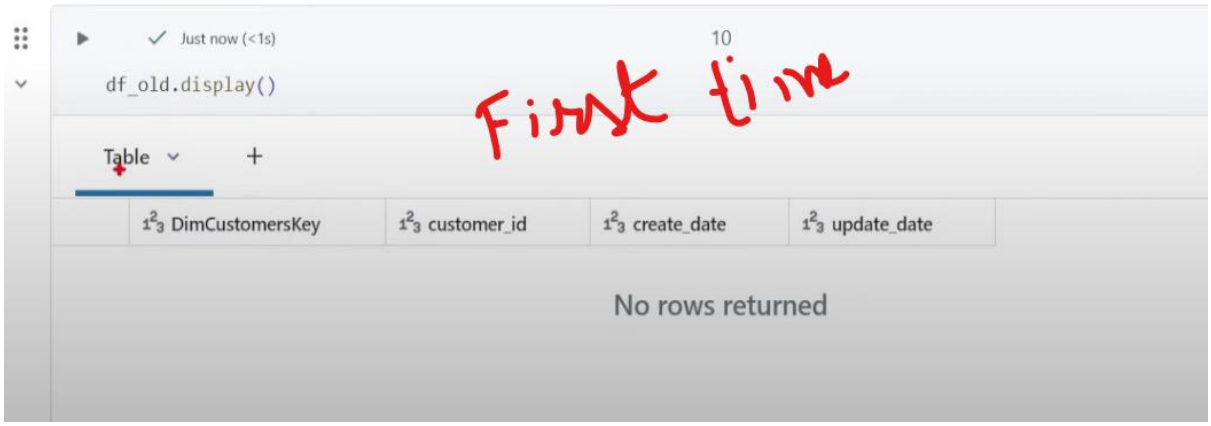
We need a Surrogate key column (pseudo key to easily apply joins) → Dimension Surrogate key

Dividing New vs Old Records

```
Apr 27, 2025 (1s)                                                    10
if init_load_flag == 0:

    df_old = spark.sql('''select DimCustomerKey, customer_id, create_date, update_date
                          from databricks_cata.gold.DimCustomers''')

else:

    df_old = spark.sql('''select 0 DimCustomerKey, 0 customer_id, 0 create_date, 0 update_date
                          FROM databricks_cata.silver.customers_silver where 1=0''')
```

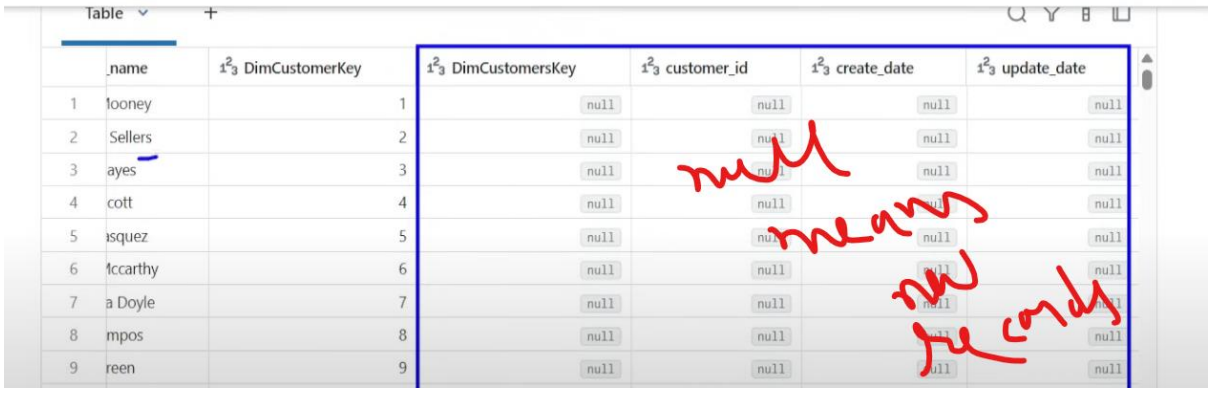
df_old: pyspark.sql.connect.dataframe.DataFrame = [DimCustomerKey: long, customer_id: string ... 2 more fields]

where 1=0 means it will only return the columns, not data; 0 is required since we are creating pseudo columns

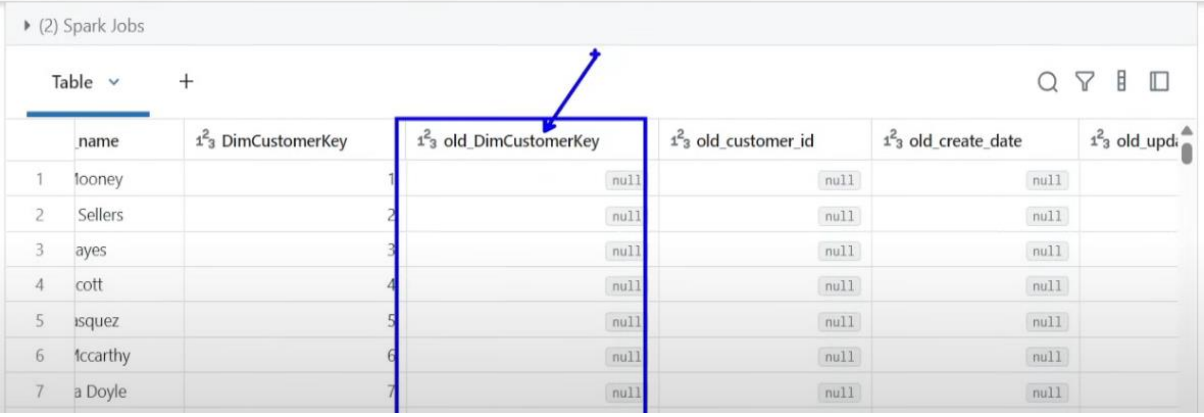


Reanming Columns of df_old

Applying Join with the Old Records



This is before renaming



Seperating New vs Old Records

Apr 27, 2025 (<1s) 18

```
df_new = df_join.filter(df_join['old_DimCustomerKey'].isNull())
```

df_new: pyspark.sql.connect.dataframe.DataFrame = [customer_id: string, email: string ... 8 more fields]

Apr 27, 2025 (<1s) 19

```
df_old = df_join.filter(df_join['old_DimCustomerKey'].isNotNull())
```

df_old: pyspark.sql.connect.dataframe.DataFrame = [customer_id: string, email: string ... 8 more fields]

Preparing df_old

Apr 27, 2025 (1s) 21

```
# Dropping all the columns which are not required

df_old = df_old.drop('old_customer_id','old_update_date')

# Renaming "old_DimCustomerKey" to "DimCustomerKey"

df_old = df_old.withColumnRenamed("old_DimCustomerKey", "DimCustomerKey")

# Renaming "old_create_date" column to "create_date"

df_old = df_old.withColumnRenamed("old_create_date", "create_date")
df_old = df_old.withColumn("create_date",to_timestamp(col("create_date")))

# Recreating "update_date" column with current timestamp

df_old = df_old.withColumn("update_date", current_timestamp())
```

df_old: pyspark.sql.connect.dataframe.DataFrame = [customer_id: string, email: string ... 7 more fields]

Preparing df_new

Apr 27, 2025 (<1s) 24

```
# Dropping all the columns which are not required

df_new = df_new.drop('old_DimCustomerKey', 'old_customer_id','old_update_date','old_create_date')

# Recreating "update_date", "current_date" columns with current timestamp

df_new = df_new.withColumn("update_date", current_timestamp())
df_new = df_new.withColumn("create_date", current_timestamp())
```

df_new: pyspark.sql.connect.dataframe.DataFrame = [customer_id: string, email: string ... 6 more fields]

Surrogate Key - From 1

Apr 27, 2025 (<1s) 27

```
df_new = df_new.withColumn("DimCustomerKey",monotonically_increasing_id()+lit(1))
```

df_new: pyspark.sql.connect.dataframe.DataFrame = [customer_id: string, email: string ... 7 more fields]

+lit(1) has been defined cause it starts from 0 by default

Adding Max Surrogate Key

Apr 27, 2025 (1s) 30

```
if init_load_flag == 1:
    max_surrogate_key = 0

else:
    df_maxsur = spark.sql("select max(DimCustomerKey) as max_surrogate_key from databricks_cata.gold.DimCustomers")

    #Converting df_maxsur to max_surrogate_key
    max_surrogate_key = df_maxsur.collect()[0]['max_surrogate_key']
```

df_maxsur: pyspark.sql.connect.dataframe.DataFrame = [max_surrogate_key: long]

collect will help convert the dataframe into a variable

Apr 27, 2025 (<1s) 31

```
df_new = df_new.withColumn("DimCustomerKey",lit(max_surrogate_key)+col("DimCustomerKey"))
```

df_new: pyspark.sql.connect.dataframe.DataFrame = [customer_id: string, email: string ... 7 more fields]

Union of df_old and df_new

Apr 27, 2025 (1s) 33

```
df_final = df_new.unionByName(df_old)
```

df_final: pyspark.sql.connect.dataframe.DataFrame = [customer_id: string, email: string ... 7 more fields]

unionByName will do a union by column name

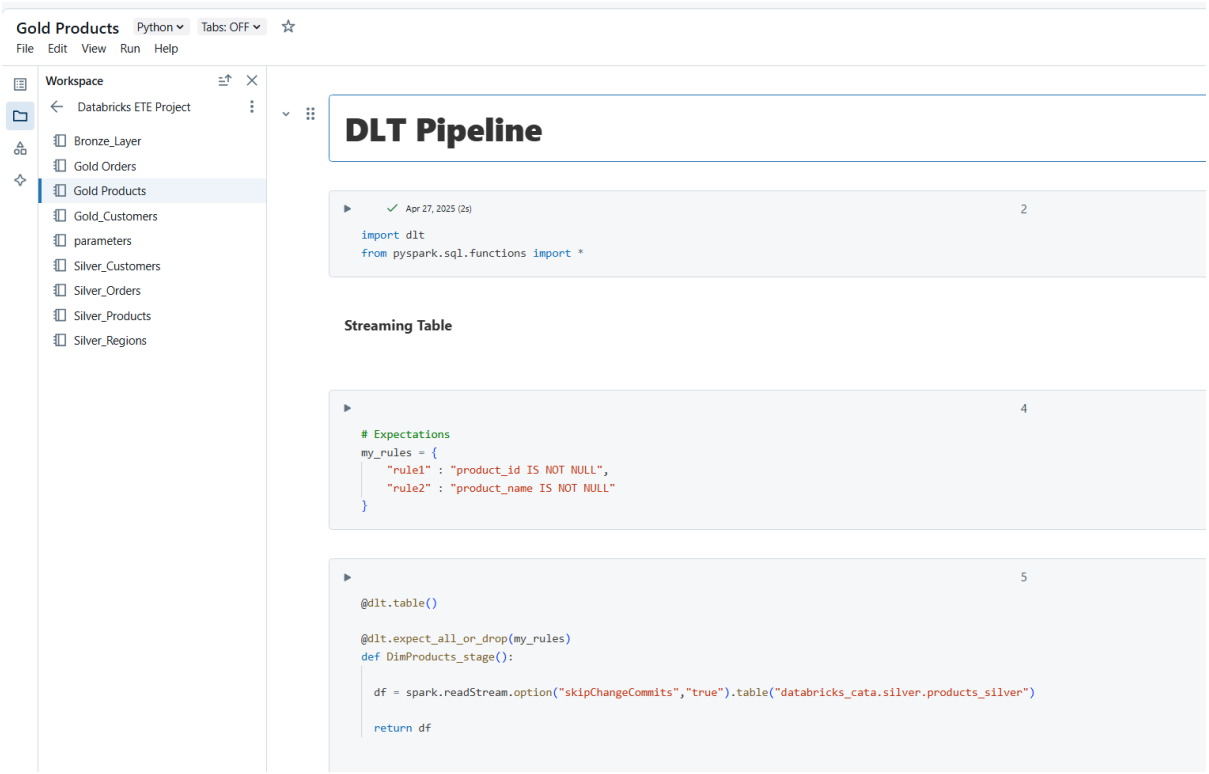
SCD Type - 1

Apr 27, 2025 (<1s)

```
from delta.tables import DeltaTable
```

Apr 27, 2025 (5s) 37

```
if (spark.catalog.tableExists("databricks_cata.gold.DimCustomers")):

    dlt_obj = DeltaTable.forPath(spark,"abfss://gold@databricksete.dfs.core.windows.net/DimCustomers")

    dlt_obj.alias("trg").merge(df_final.alias("src"),"trg.DimCustomerKey = src.DimCustomerKey")\
        .whenMatchedUpdateAll()\
        .whenNotMatchedInsertAll()\
        .execute()

else:
    df_final.write.mode("overwrite")\
    .format("delta")\
    .option("path","abfss://gold@databricksete.dfs.core.windows.net/DimCustomers")\
    .saveAsTable("databricks_cata.gold.DimCustomers")
```



Streaming View

DimProducts

Apr 27, 2025 (<1s)

```
dlt.create_streaming_table("DimProducts")
```

DimProducts is defined as a **Delta Live Tables** dataset .

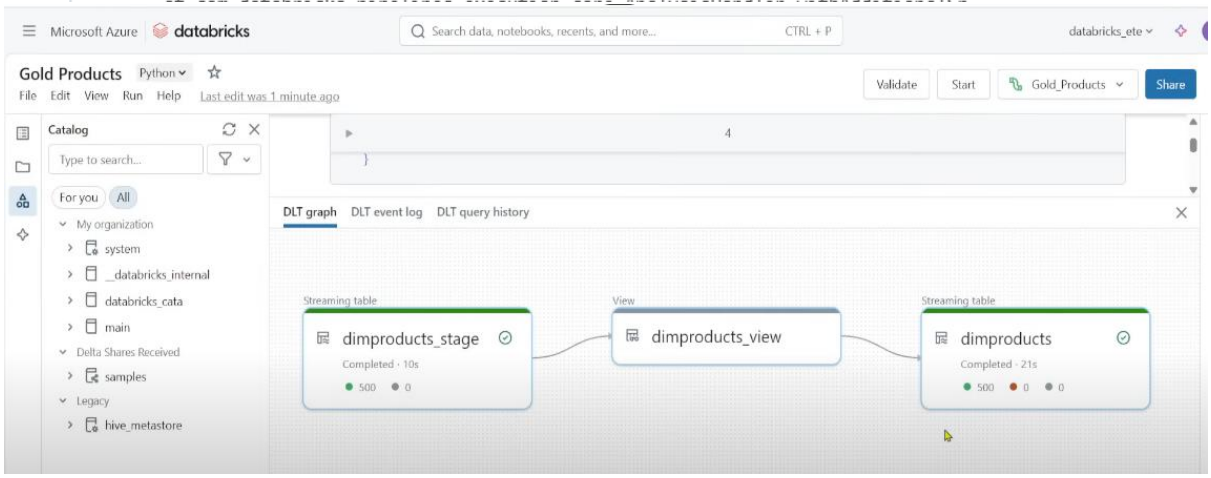
To populate your table you must either:

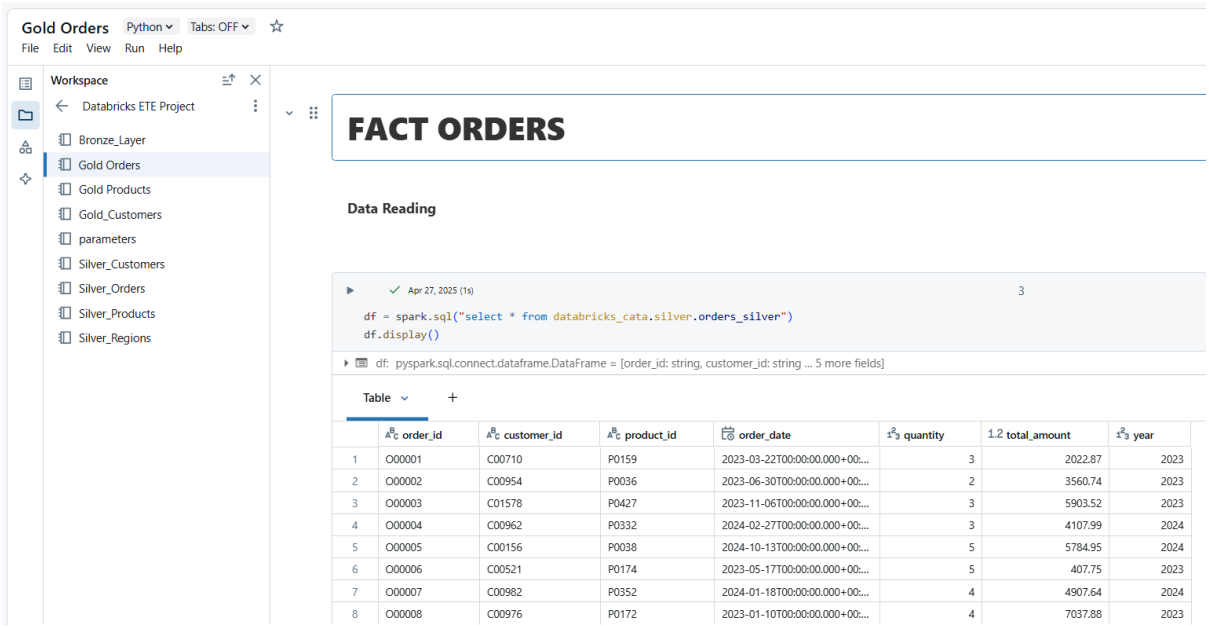
- Run an existing pipeline using the **Delta Live Tables** menu
- Create a new pipeline: Create Pipeline

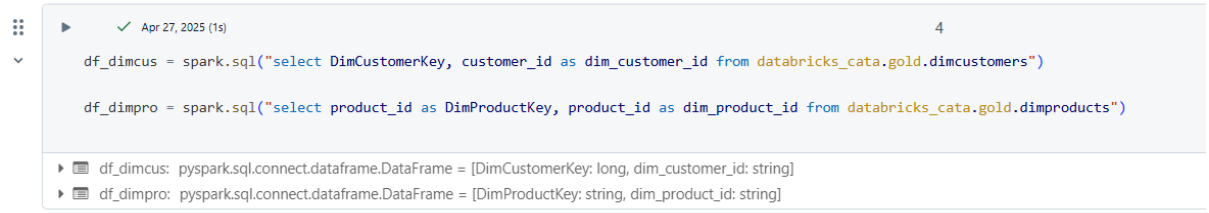
Apr 27, 2025 (1s)

```
dlt.apply_changes(
  target = "DimProducts",
  source = "DimProducts_view",
  keys = ["product_id"],
  sequence_by = "product_id",
  stored_as_scd_type = 2
)
```

```
at com.databricks.pipelines.util.SparkSessionUtils$.withSparkSession(SparkS
```

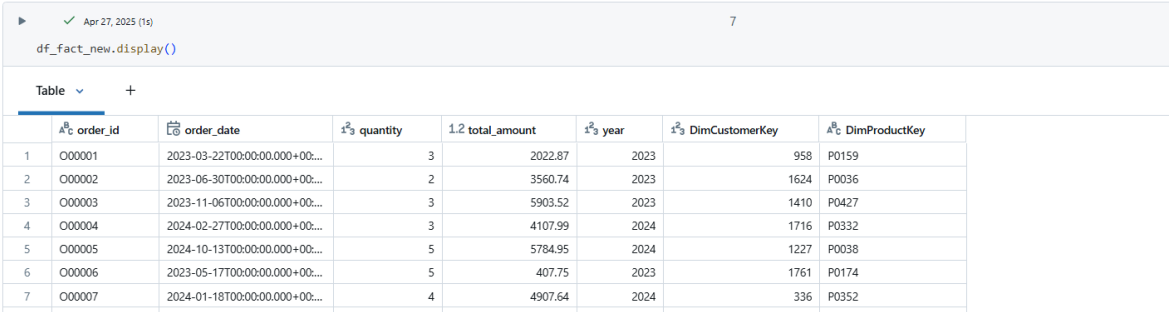






Creating Star Schema

Fact Dataframe



Upsert on Fact Table

Apr 27, 2025 (<1s) 9

```
from delta.tables import DeltaTable
```

Apr 27, 2025 (5s) 10

```
if spark.catalog.tableExists("databricks_cata.gold.FactOrders"):

    dlt_obj = DeltaTable.forName(spark, "databricks_cata.gold.FactOrders")

    dlt_obj.alias("trg").merge(df_fact_new.alias("src"), "trg.order_id = src.order_id AND trg.DimCustomerKey = src.DimCustomerKey AND trg.DimProductKey = src.DimProductKey")\
    .whenMatchedUpdateAll()\
    .whenNotMatchedInsertAll()\
    .execute()

else:
    df_fact_new.write.format("delta")\
            .option("path","abfss://gold@databricksete.dfs.core.windows.net/FactOrders")\
            .saveAsTable("databricks_cata.gold.FactOrders")
```

+ Code + Text

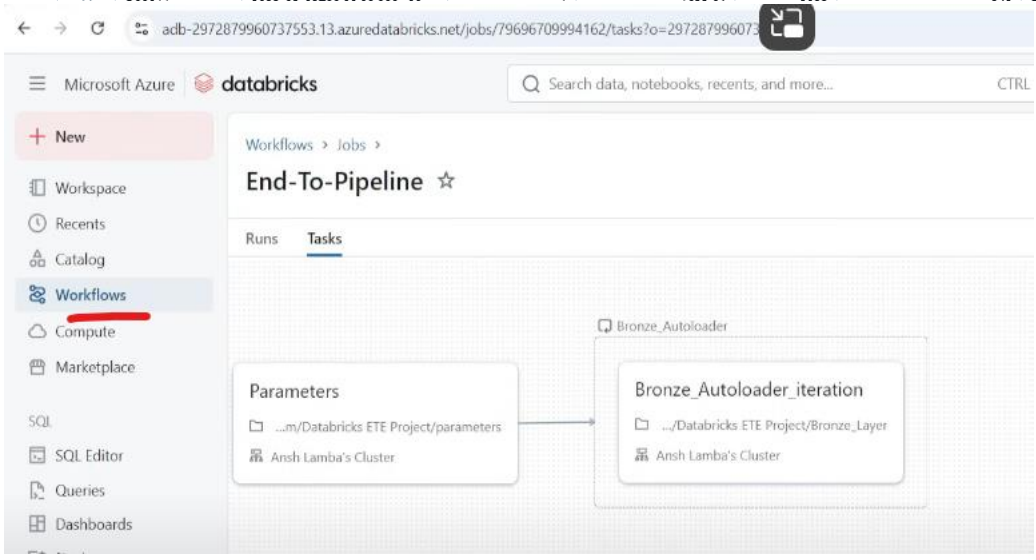
Apr 27, 2025 (3s) 11

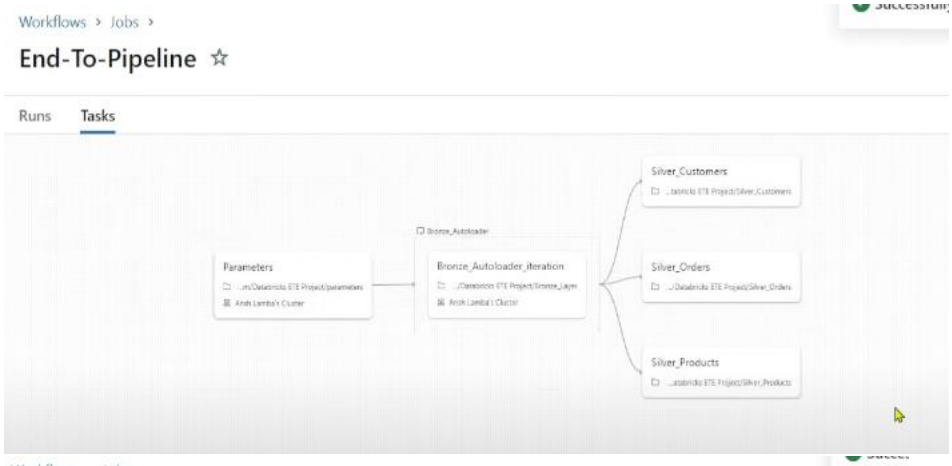
```
%sql
select * from databricks_cata.gold.FactOrders
```

_sqldf: pyspark.sql.connect.dataframe.DataFrame = [order_id: string, order_date: timestamp ... 5 more fields]

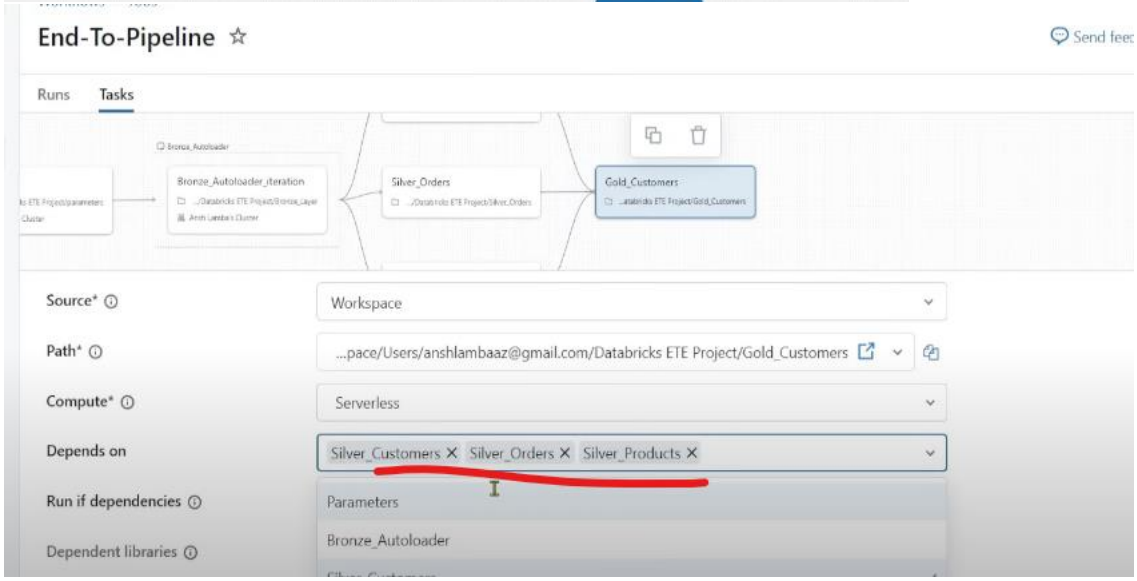
Table +

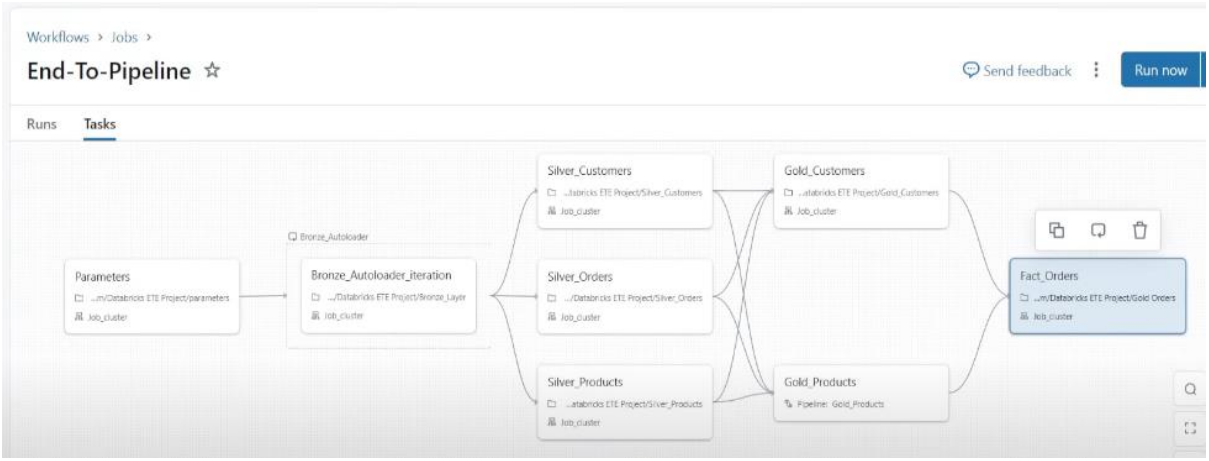
	order_id	order_date	quantity	total_amount	year	DimCustomerKey	DimProductKey
1	O00001	2023-03-22T00:00:00.000+00:...	3	2022.87	2023	958	P0159
2	O00002	2023-06-30T00:00:00.000+00:...	2	3560.74	2023	1624	P0036
3	O00003	2023-11-06T00:00:00.000+00:...	3	5903.52	2023	1410	P0427
4	O00004	2024-02-27T00:00:00.000+00:...	3	4107.99	2024	1716	P0332
5	O00005	2024-10-13T00:00:00.000+00:...	5	5784.95	2024	1227	P0038
6	O00006	2023-05-17T00:00:00.000+00:...	5	407.75	2023	1761	P0174
7	O00007	2024-01-18T00:00:00.000+00:...	4	4907.64	2024	336	P0352
8	O00008	2023-01-10T00:00:00.000+00:...	4	7037.88	2023	1731	P0172
9	O00009	2023-04-20T00:00:00.000+00:...	3	4076.97	2023	1914	P0238
10	O00010	2023-07-07T00:00:00.000+00:...	4	5695.64	2023	1726	P0258




Workflows › Jobs ›
End-To-Pipeline ☆
Runs
Tasks
Parameters
Bronze_Autoloader
Bronze_Autoloader_iteration
Silver_Customers
Silver_Orders
Silver_Products


Workflows › Jobs ›
End-To-Pipeline ☆
Runs
Tasks
Parameters
Bronze_Autoloader
Bronze_Autoloader_iteration
Silver_Customers
Silver_Orders
Silver_Products
Gold_Customers


End-To-Pipeline ☆
Send feed
Runs
Tasks
Bronze_Autoloader
Bronze_Autoloader_iteration
Silver_Orders
Gold_Customers
Source*
Workspace
Path*
...pace/Users/anshlambaaz@gmail.com/Databricks ETE Project/Gold_Customers
Compute*
Serverless
Depends on
Silver_Customers
Silver_Orders
Silver_Products
Run if dependencies
Parameters
Bronze_Autoloader
Dependent libraries



Gold will start once all the silver ingestion are completed, Fact gold will start once the Dimension gold are completed