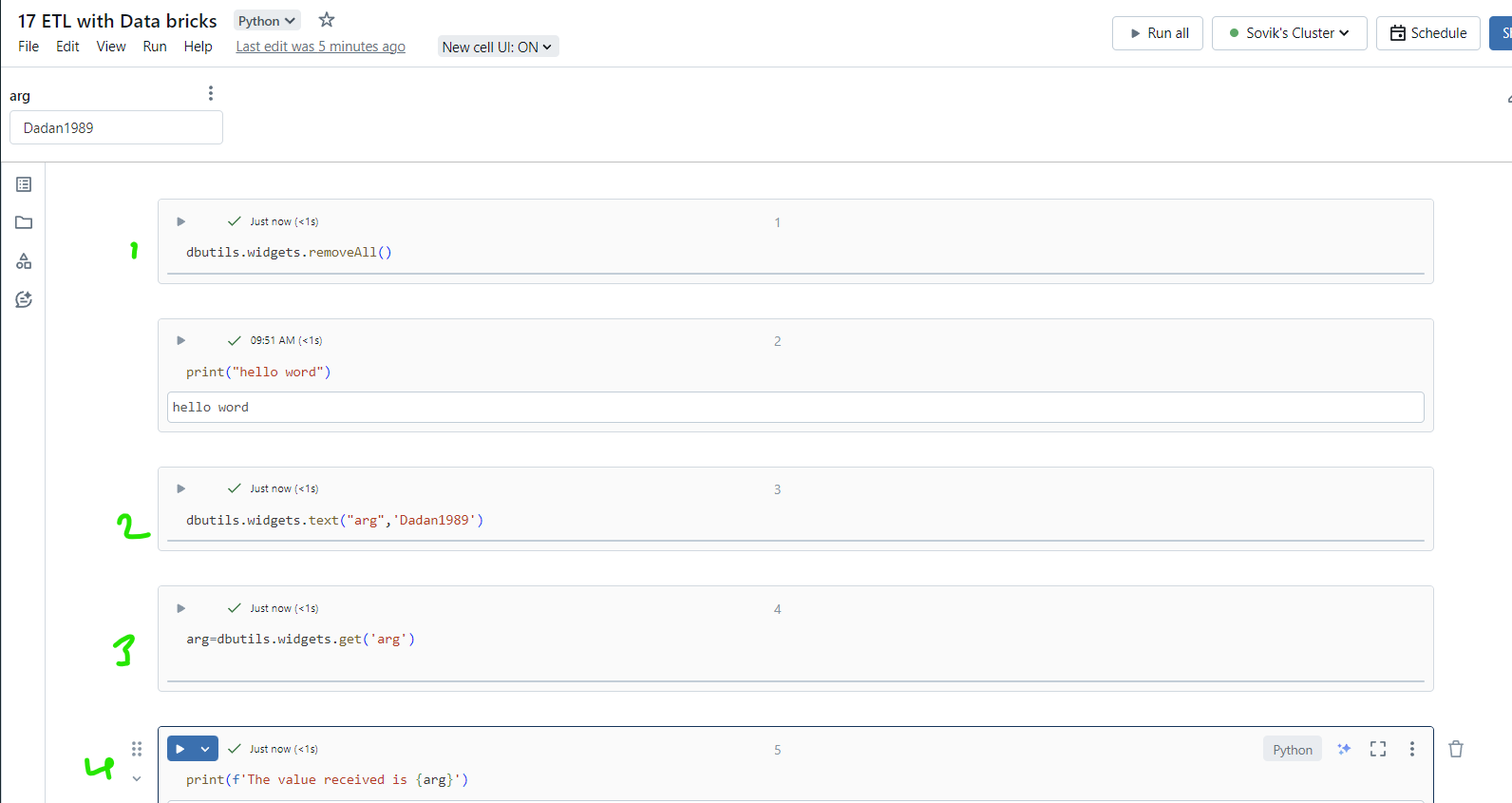
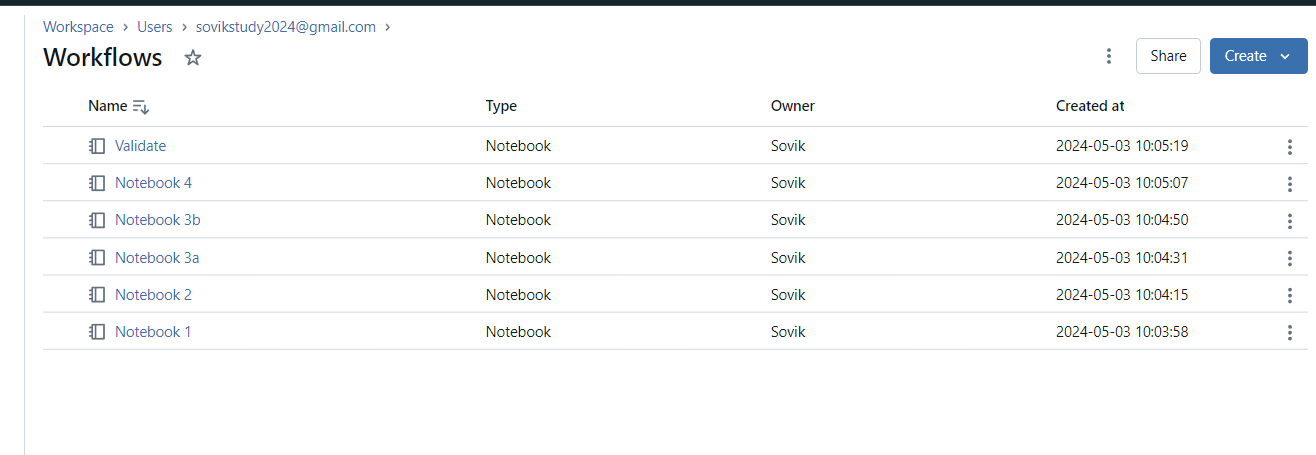


Notebooks which will be used for databricks ETL



Create Notebooks (Notebook1,notebook2,notebook3a,notebook3b,notebook4,validate) in workspace



1. Validate Python based Notebooks

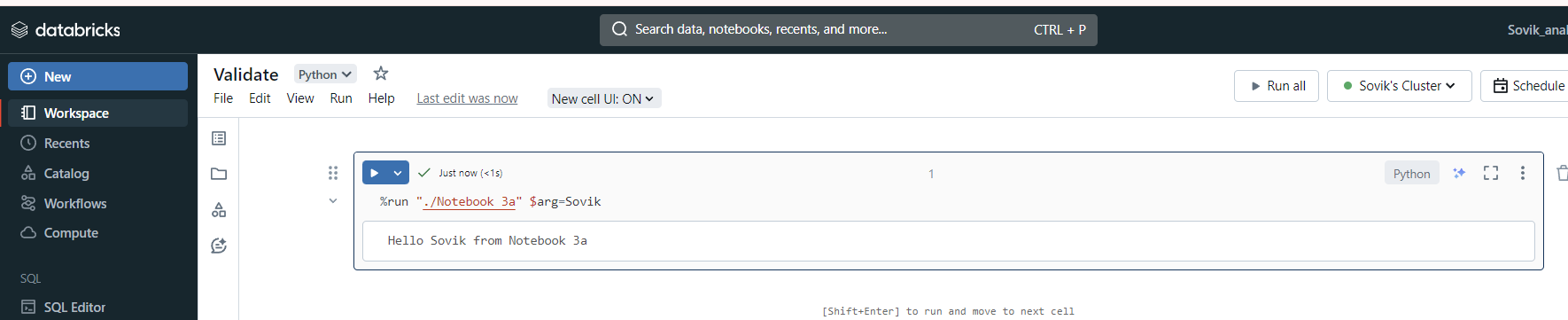
Notebook 3a:

arg=dbutils.widgets.get('arg')

print(f'Hello {arg} from Notebook 3a' )

From **validate** we check

%run "./Notebook 3a" $arg=Sovik



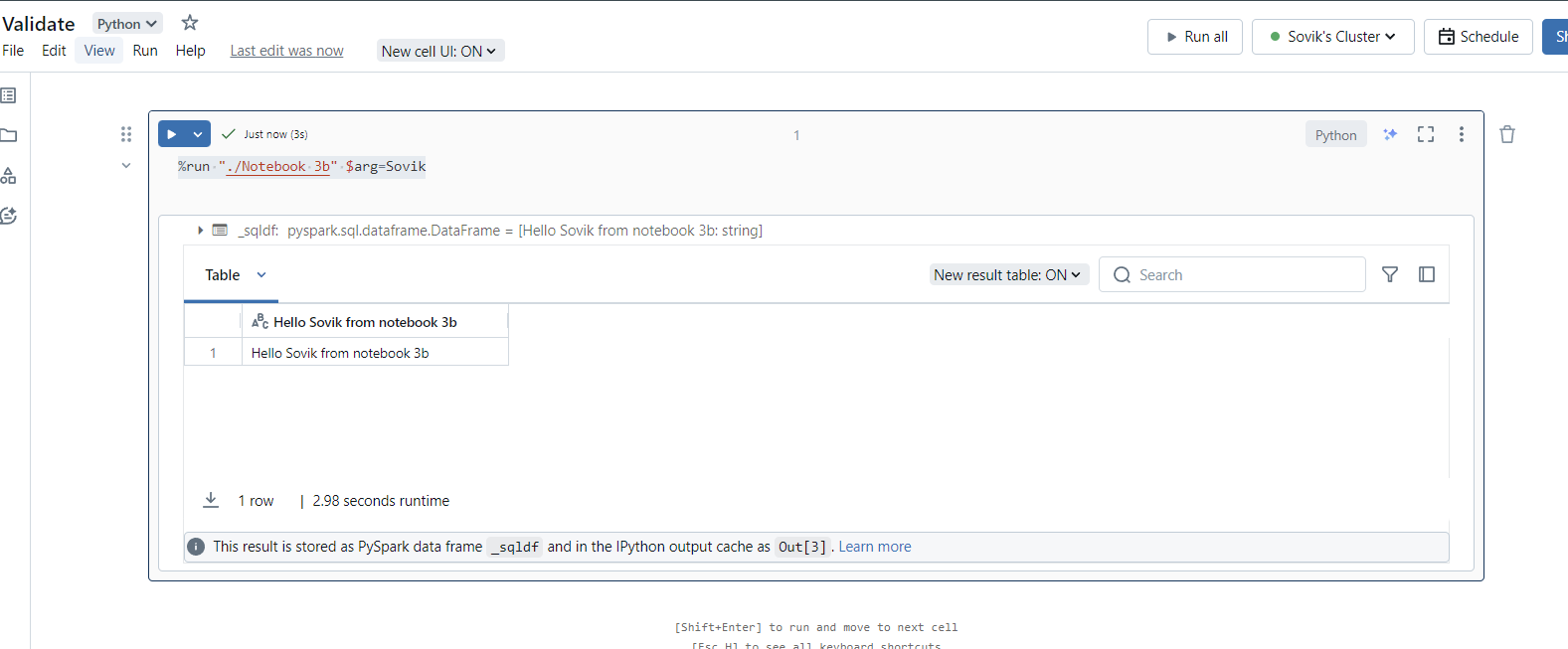
1. Validate SQL based notebooks

Notebook 3b:

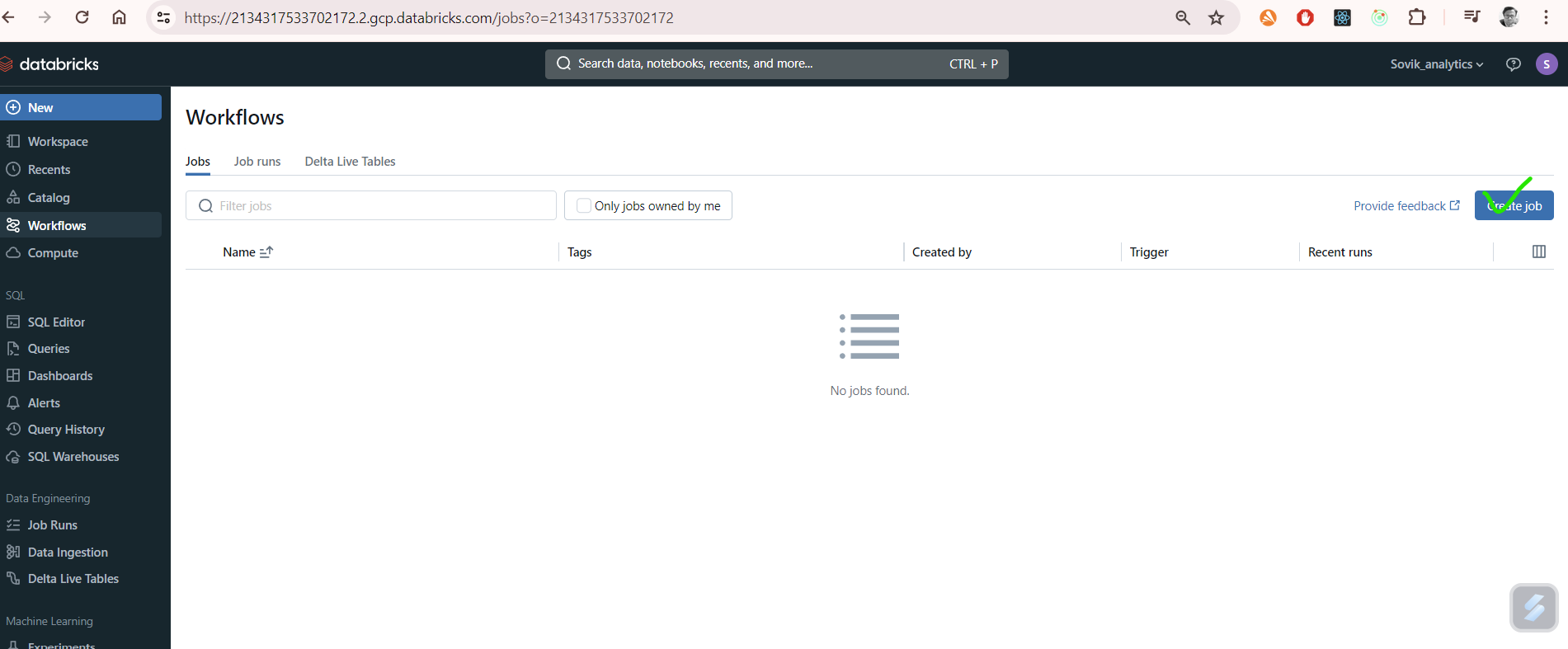
Select "Hello ${arg} from notebook 3b"

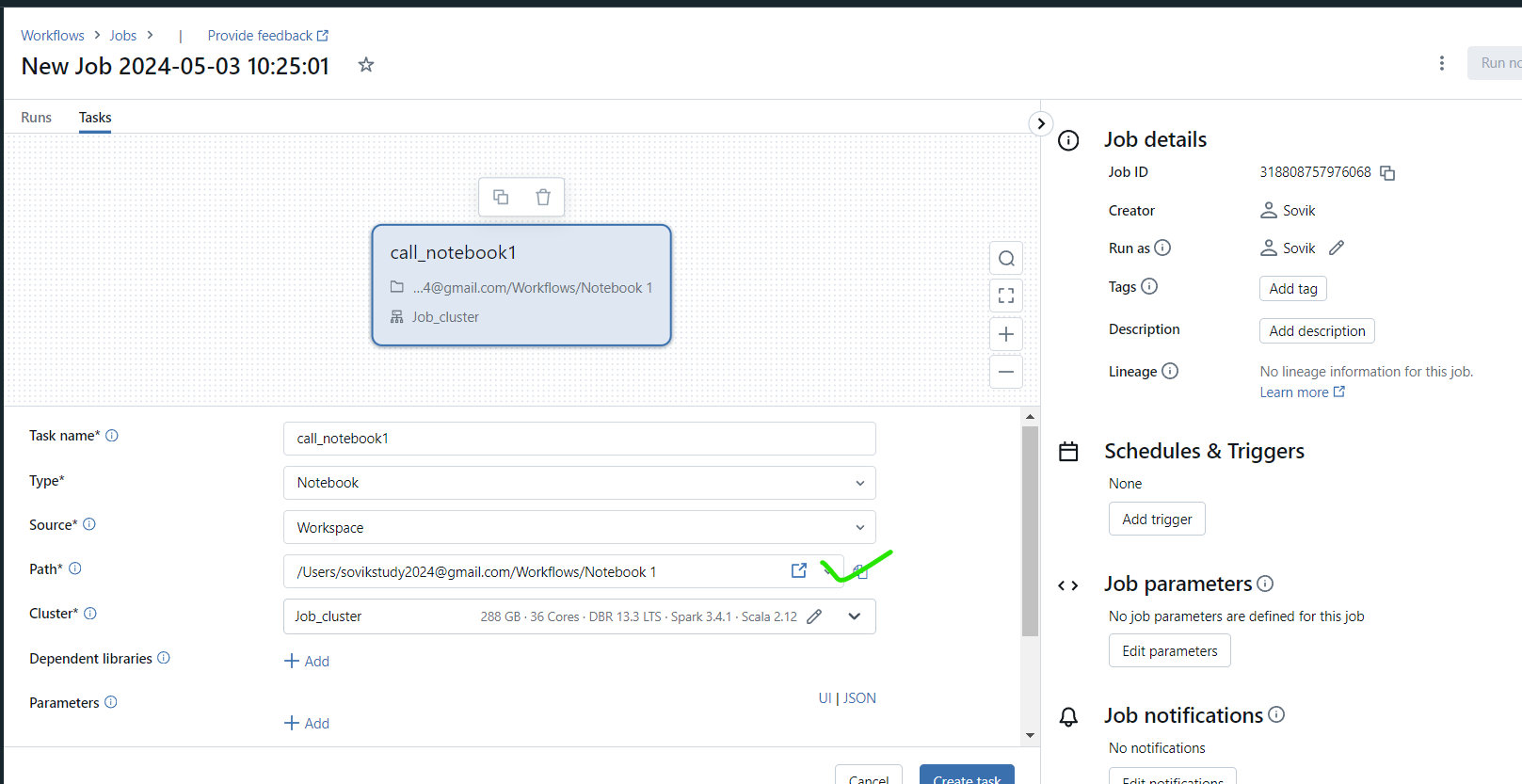
Validate:

%run "./Notebook 3b" $arg=Sovik

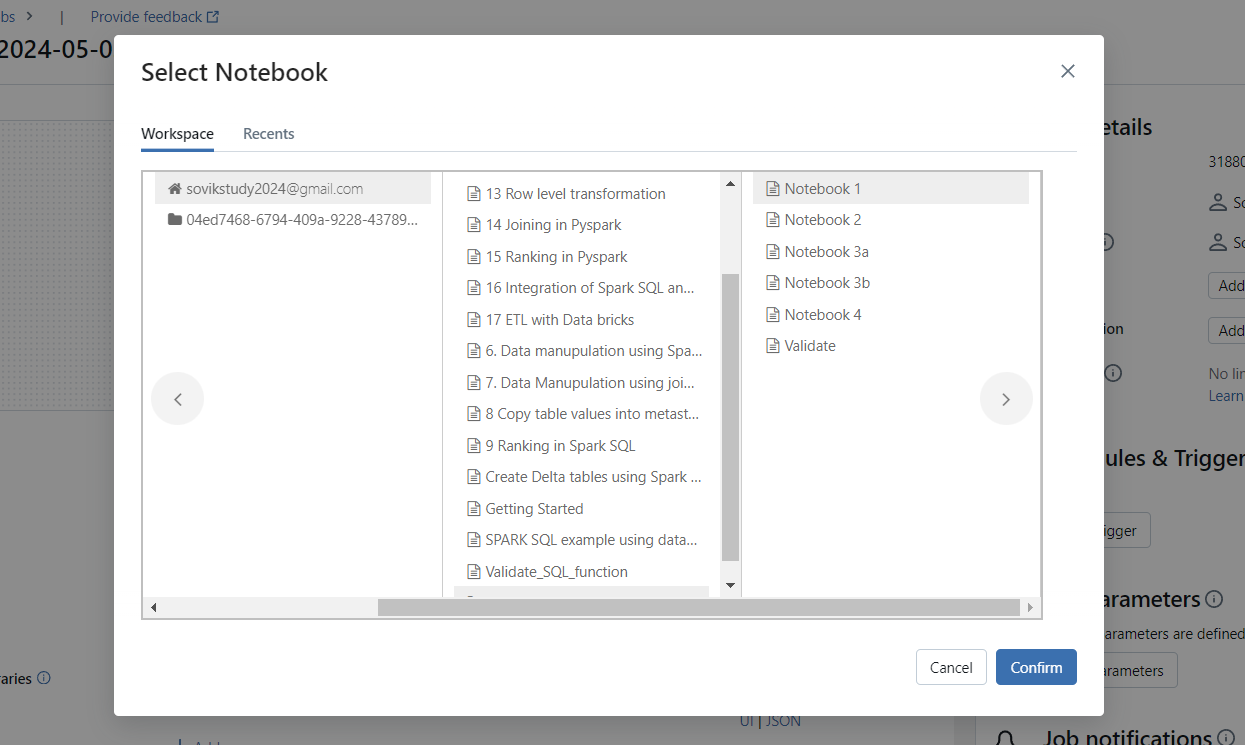


Now since we are clear of how to pass argument to SQL based or python based Notebooks we can now create Workflows





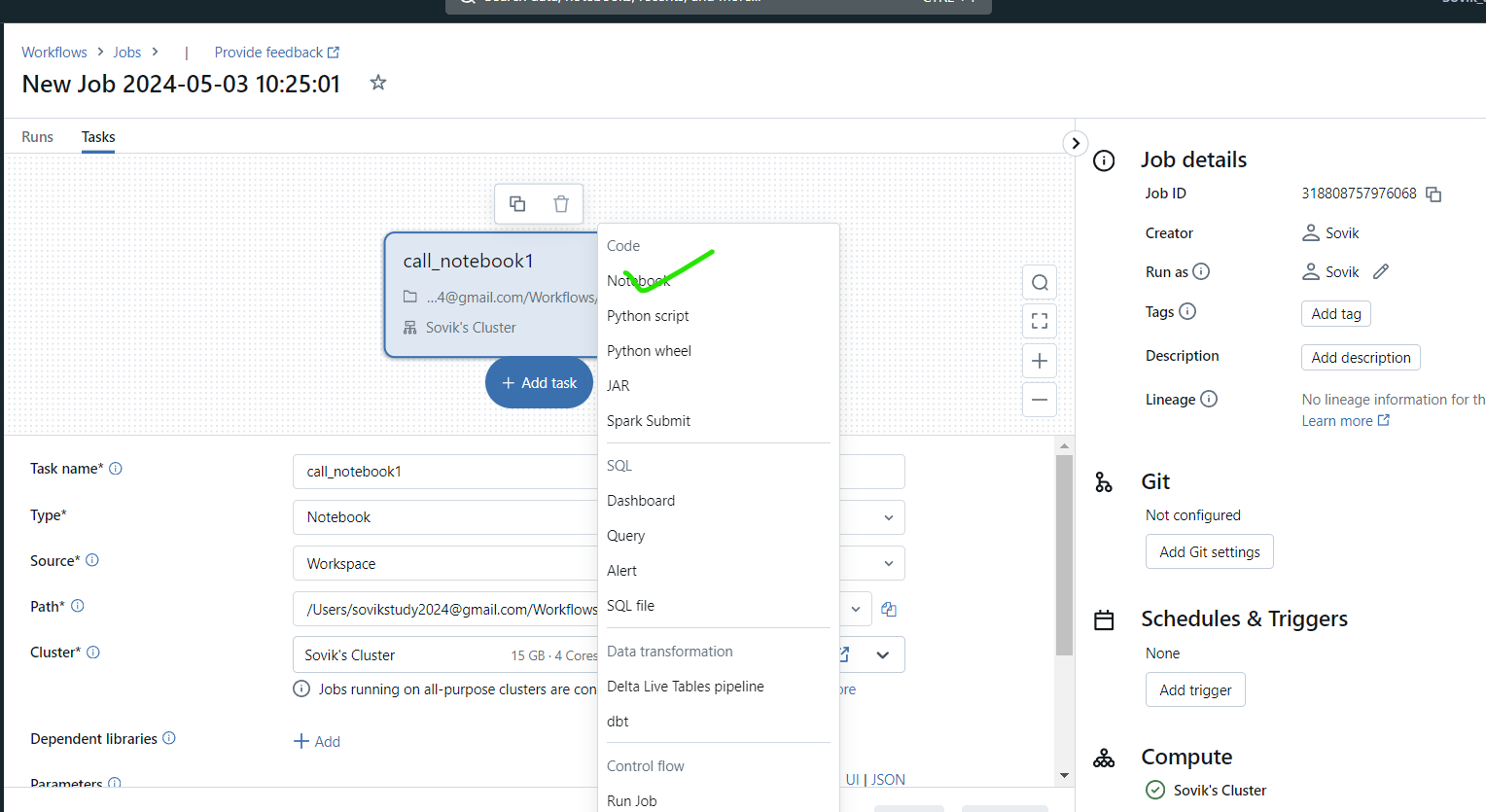
Path:



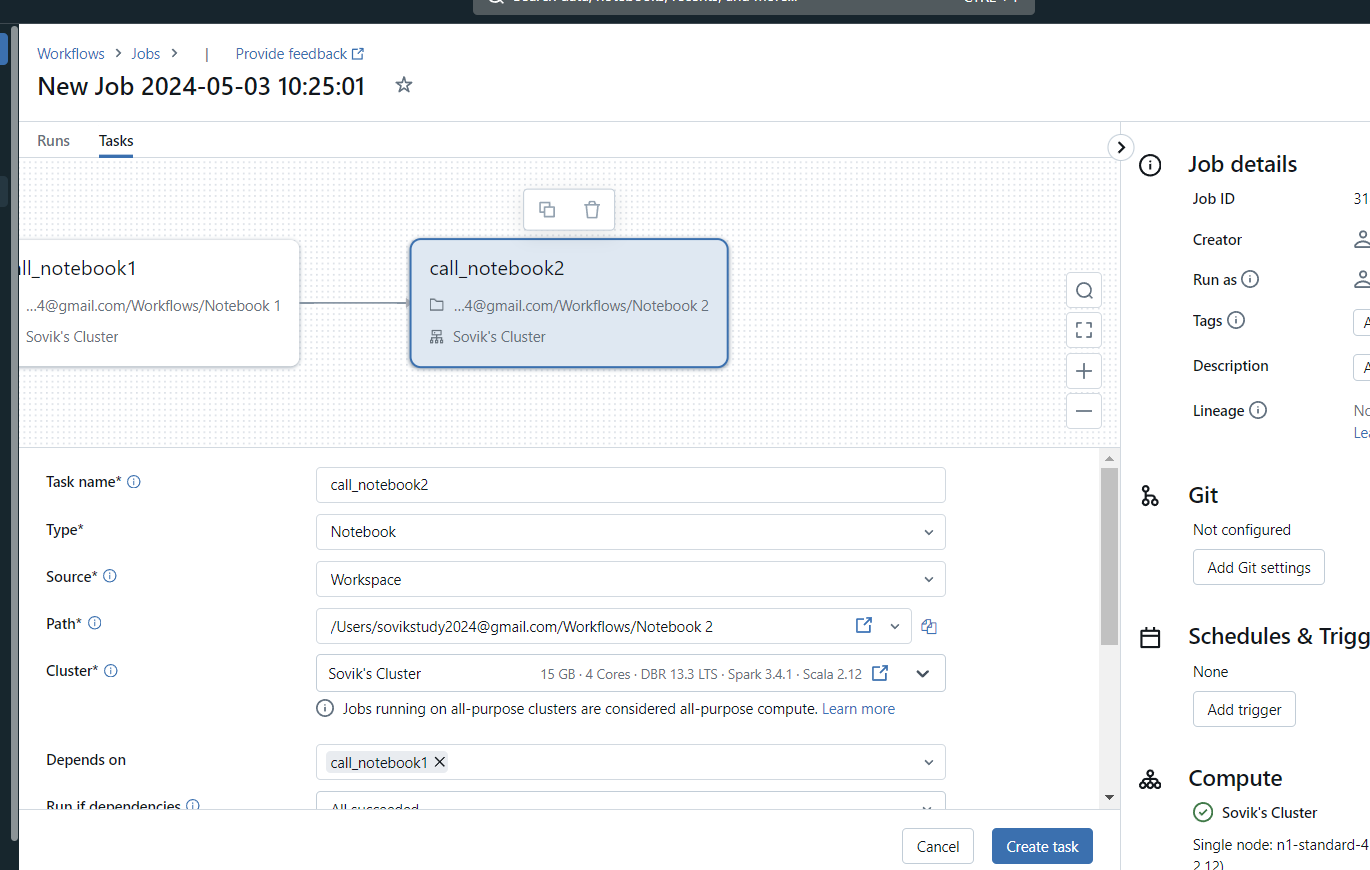


New task added.

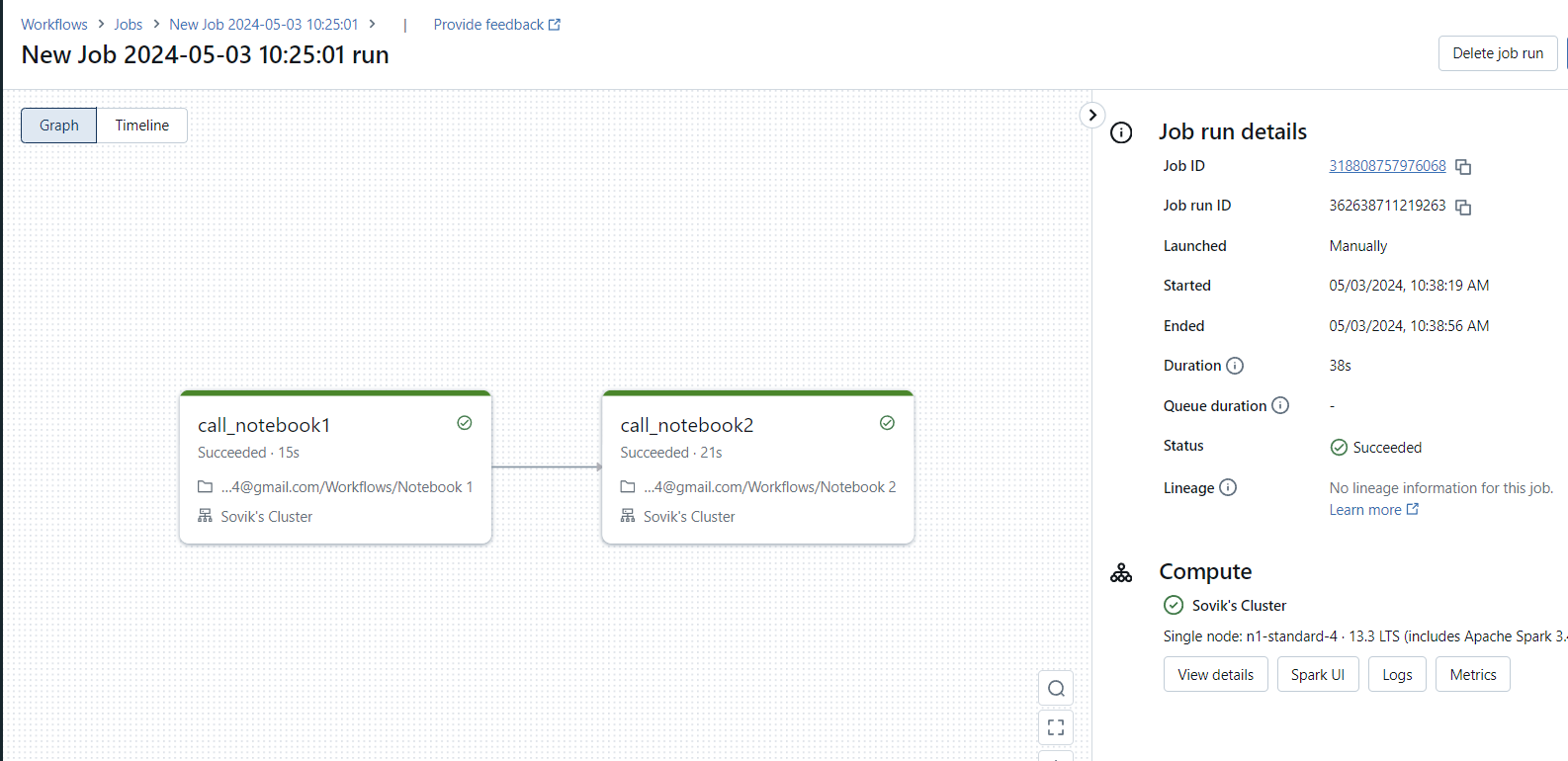
Now we add dependant task



Notebook2

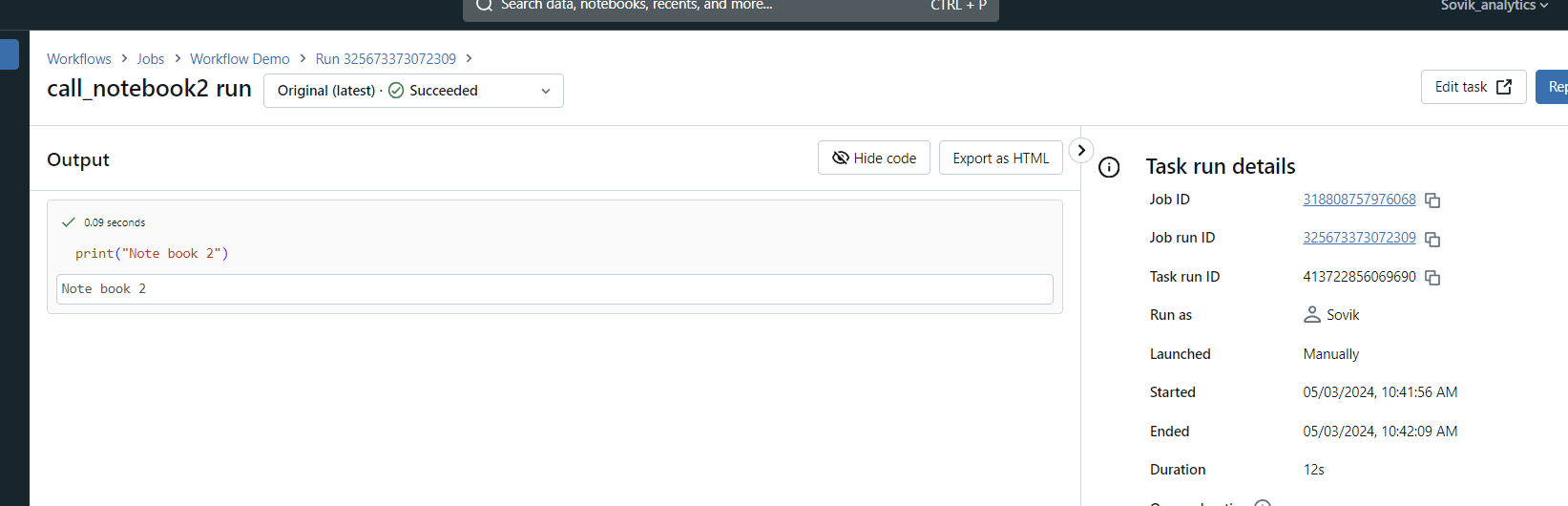


Lets check with Run now



Job runs:

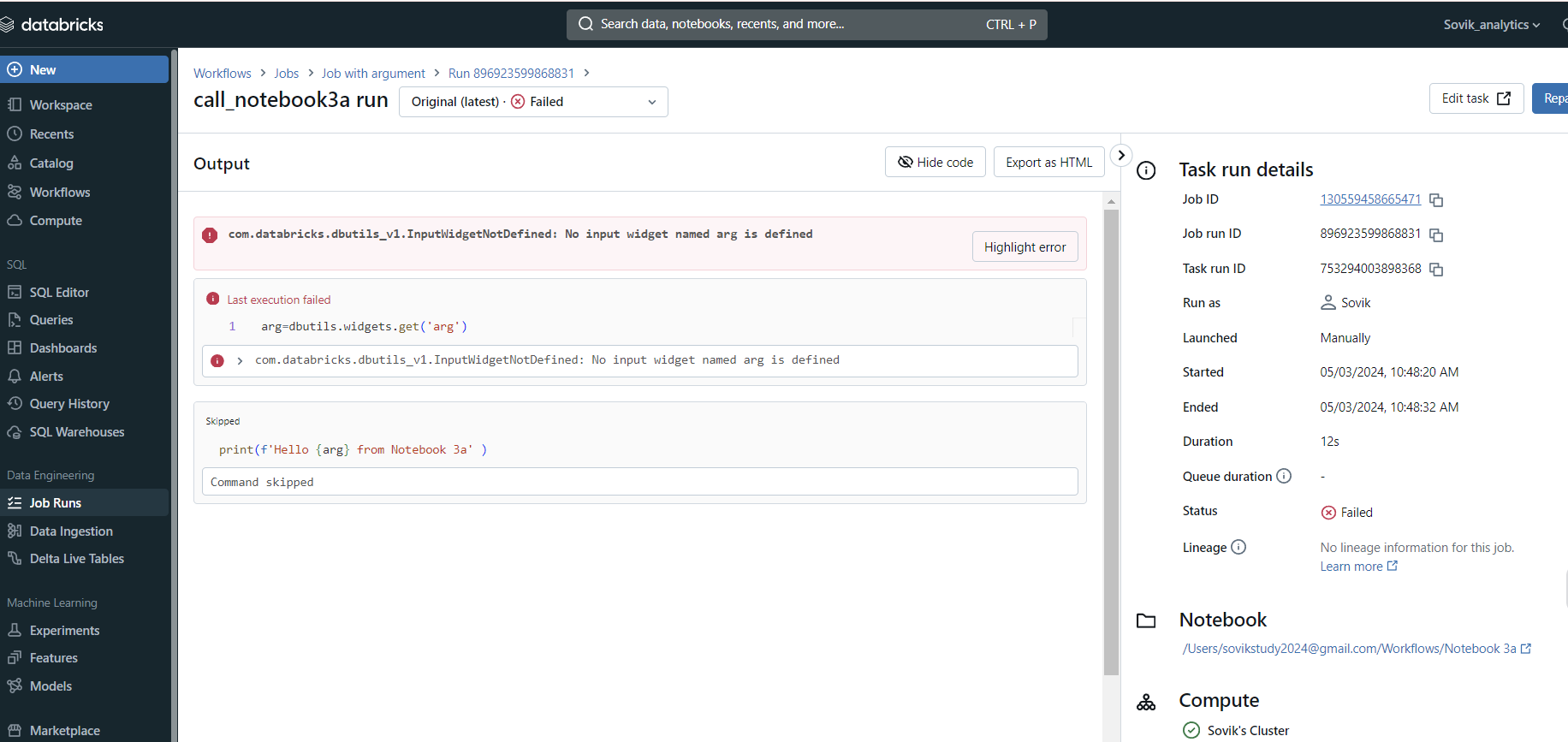




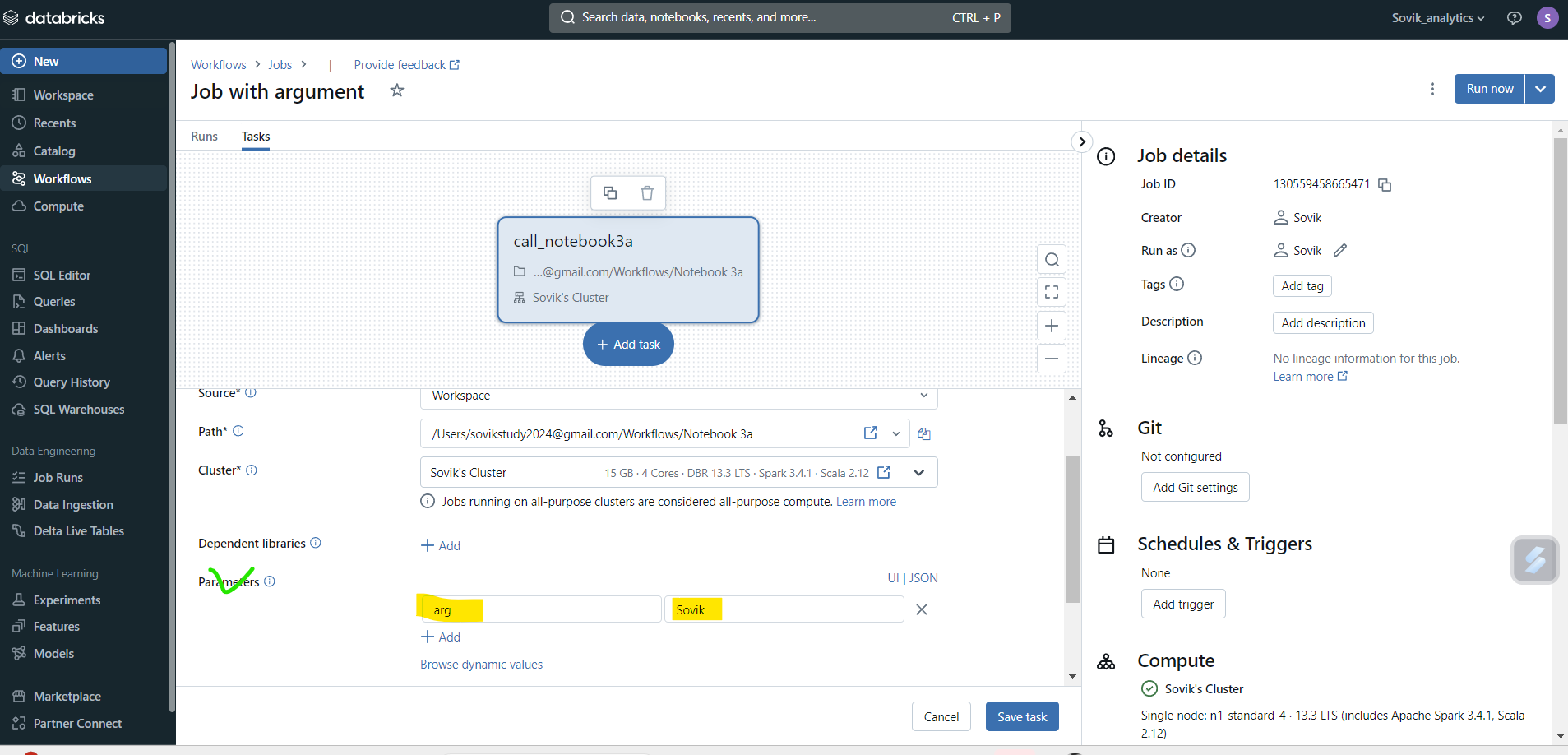
Now we create jobs from where we pass the parameters

1. Create jobs

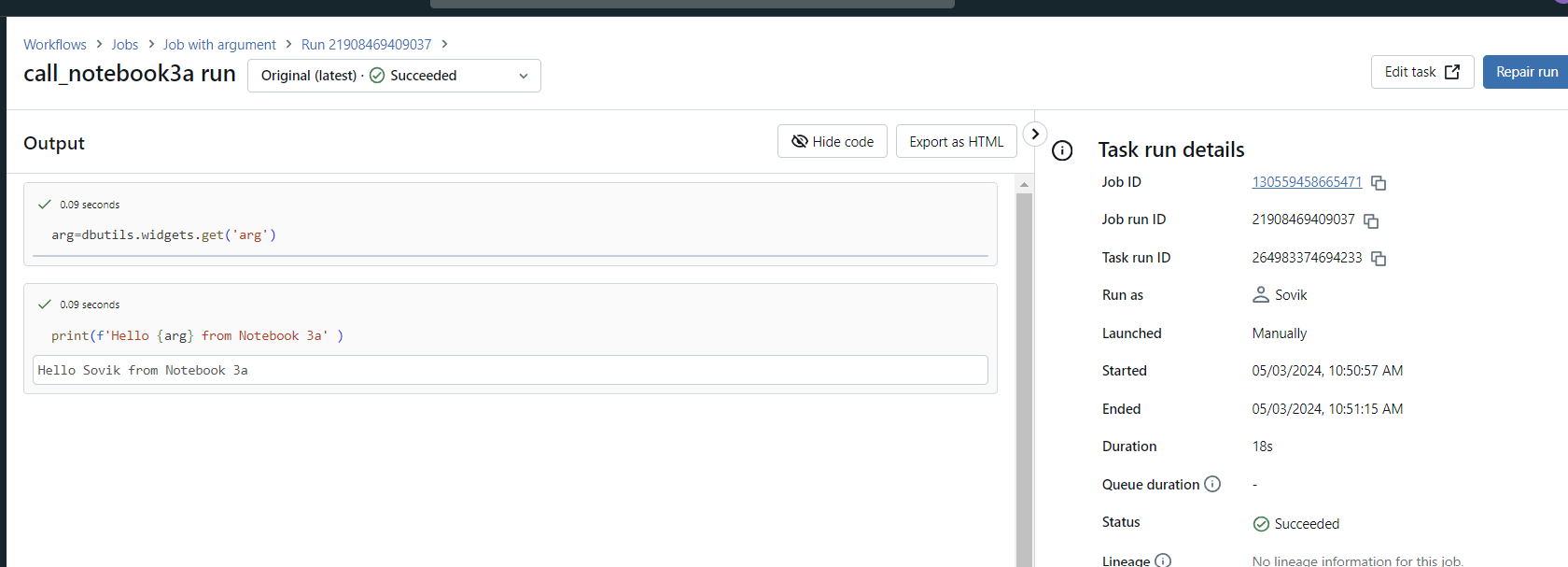
When we create the job for only notebook 3a without paramters we get below error



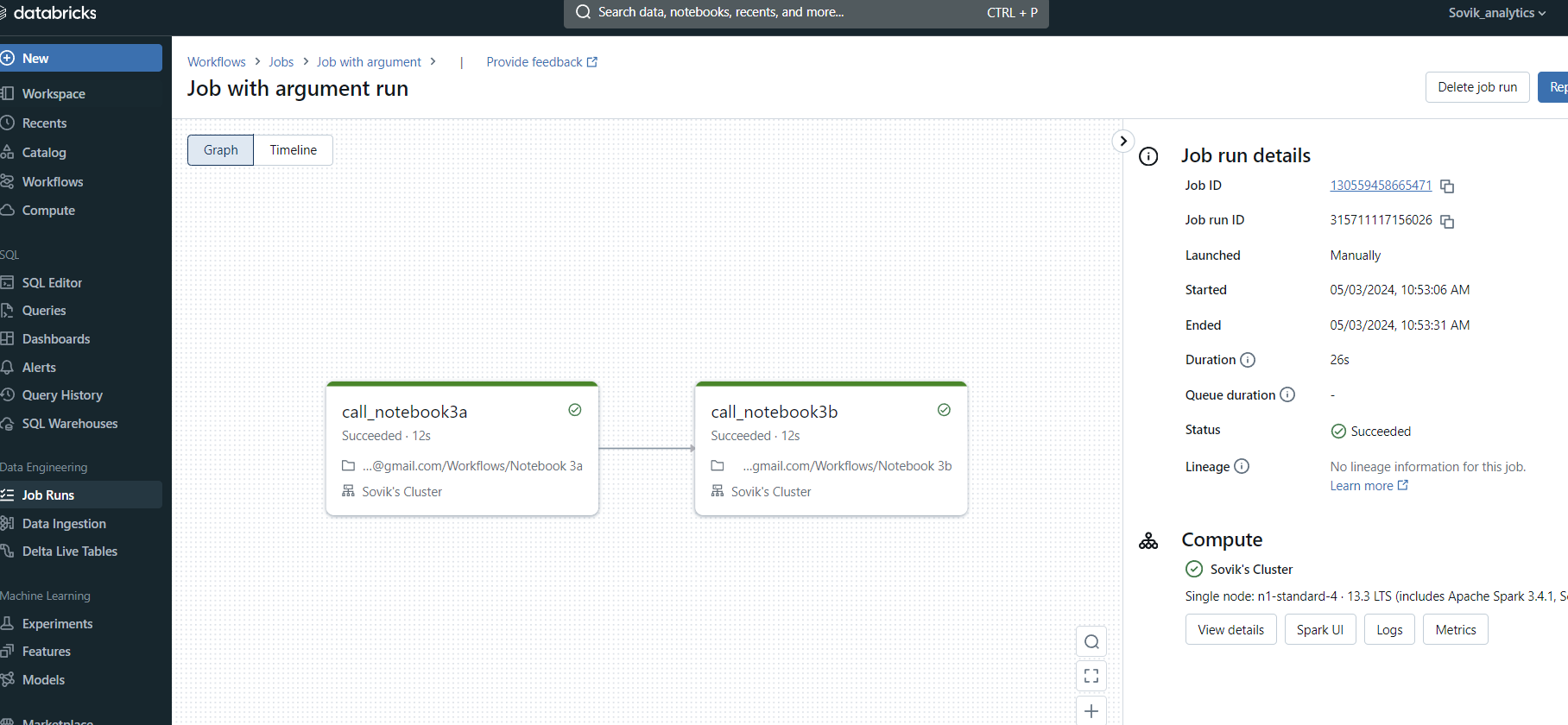
We now add the parameter and then run



Output:

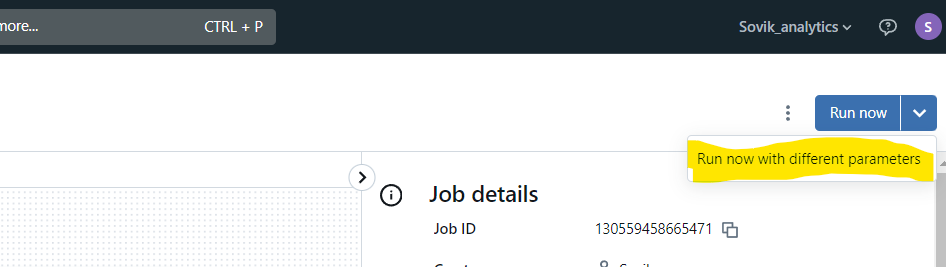


We add notebook3b with 3a:



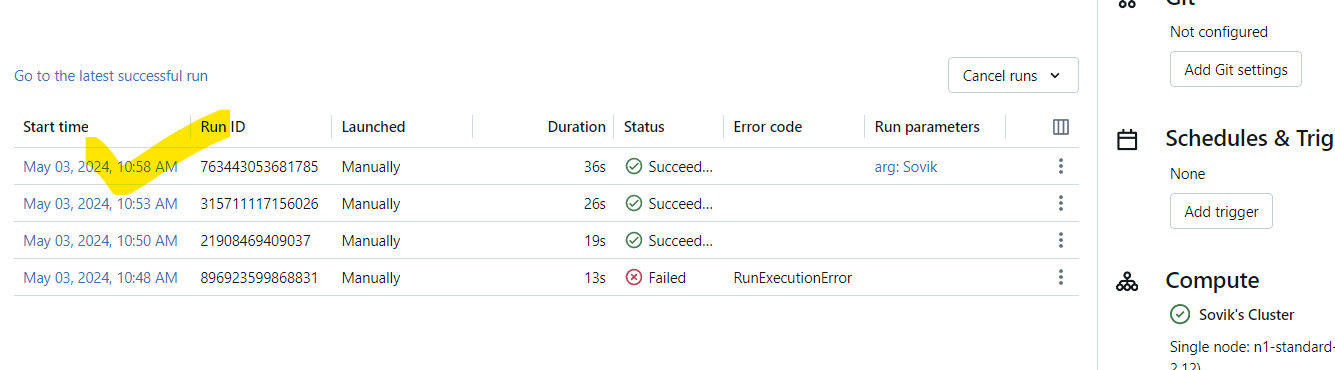
Lets pass parameter at Job level

1. Remove parameter at task level
2. Goto Run Now

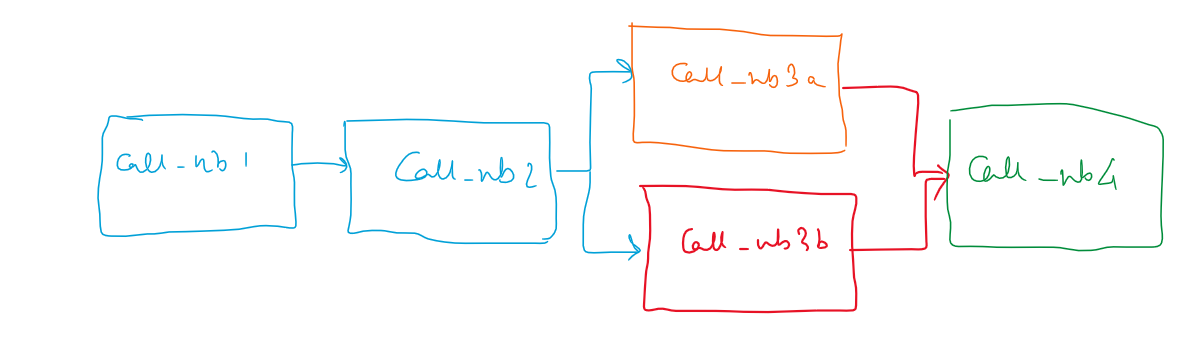




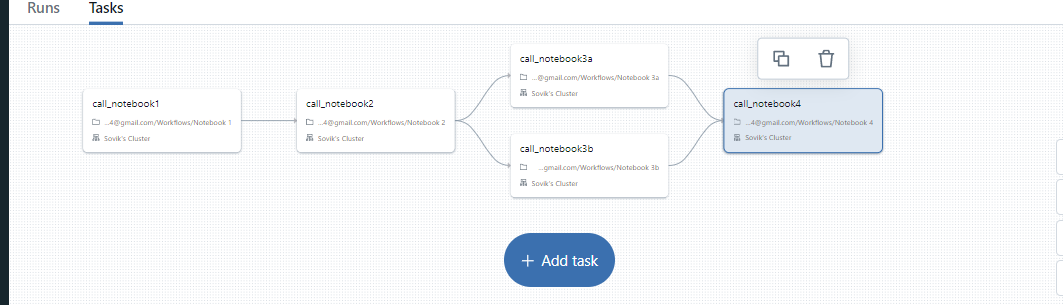
1. Run
2. Succesful



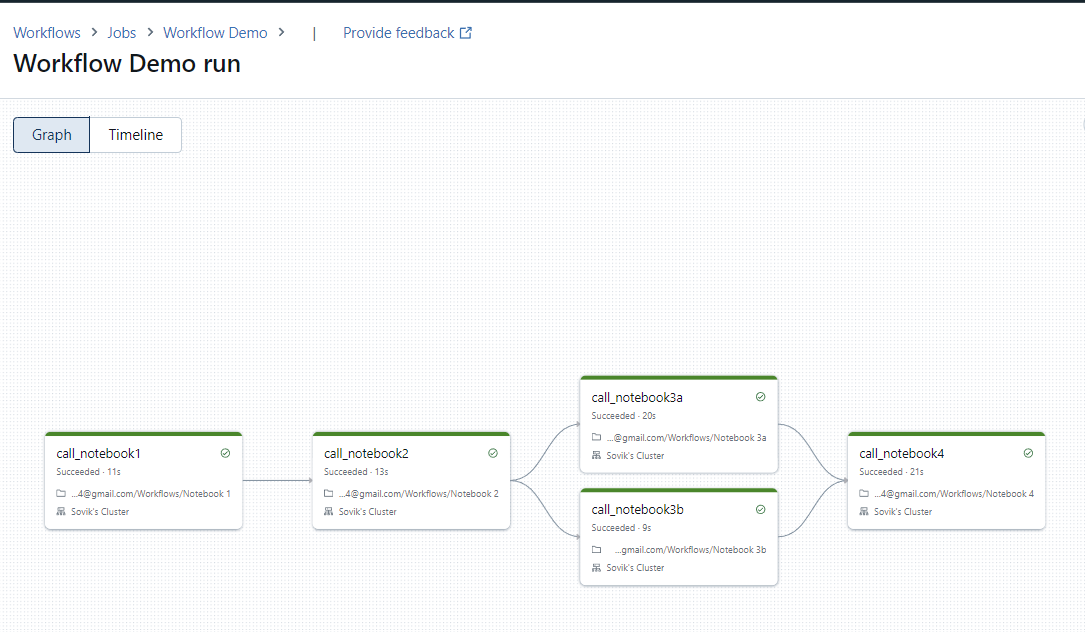
We now create Pipeline with Workflow Demo:



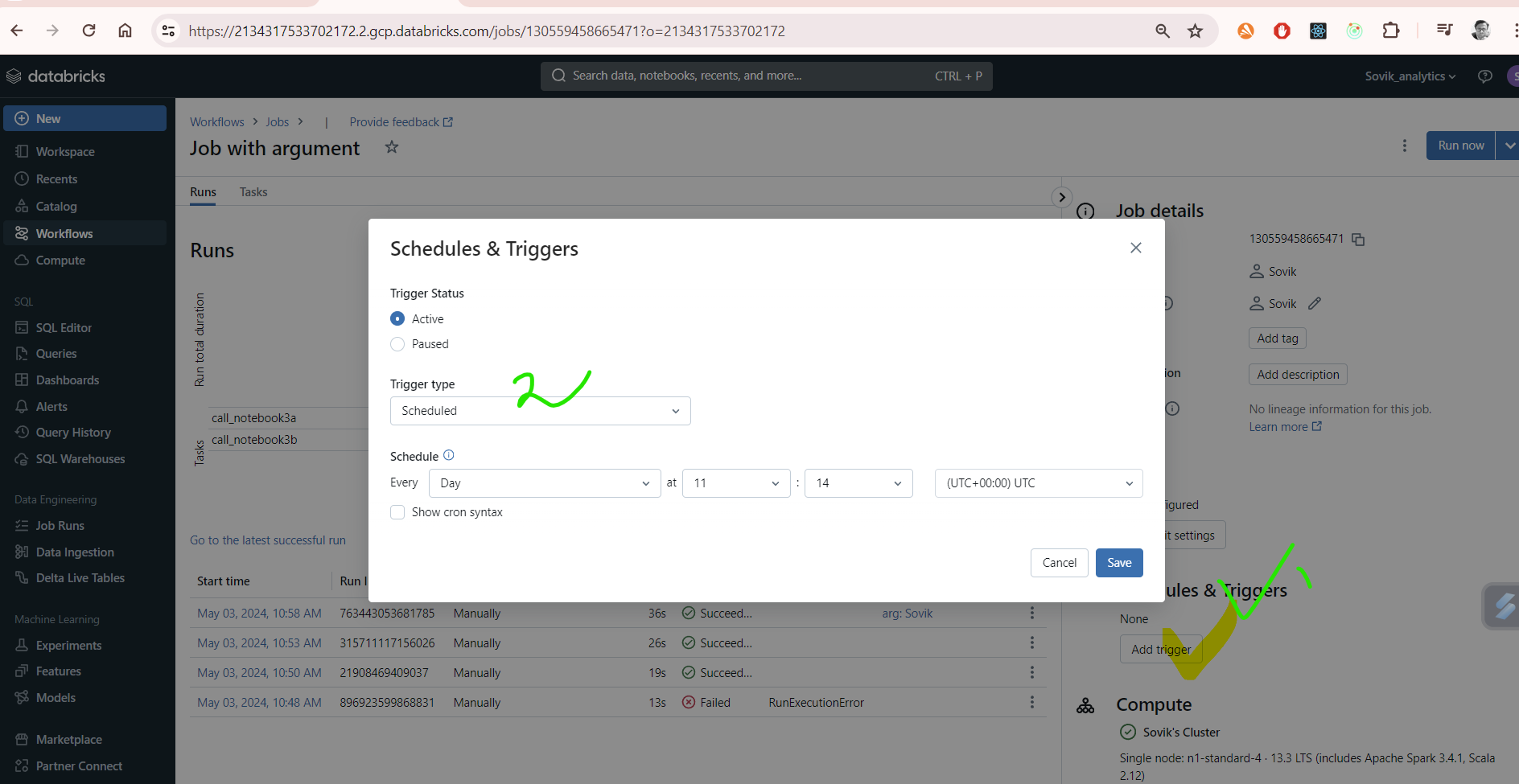
We created the wf:



Demo job run is successful:

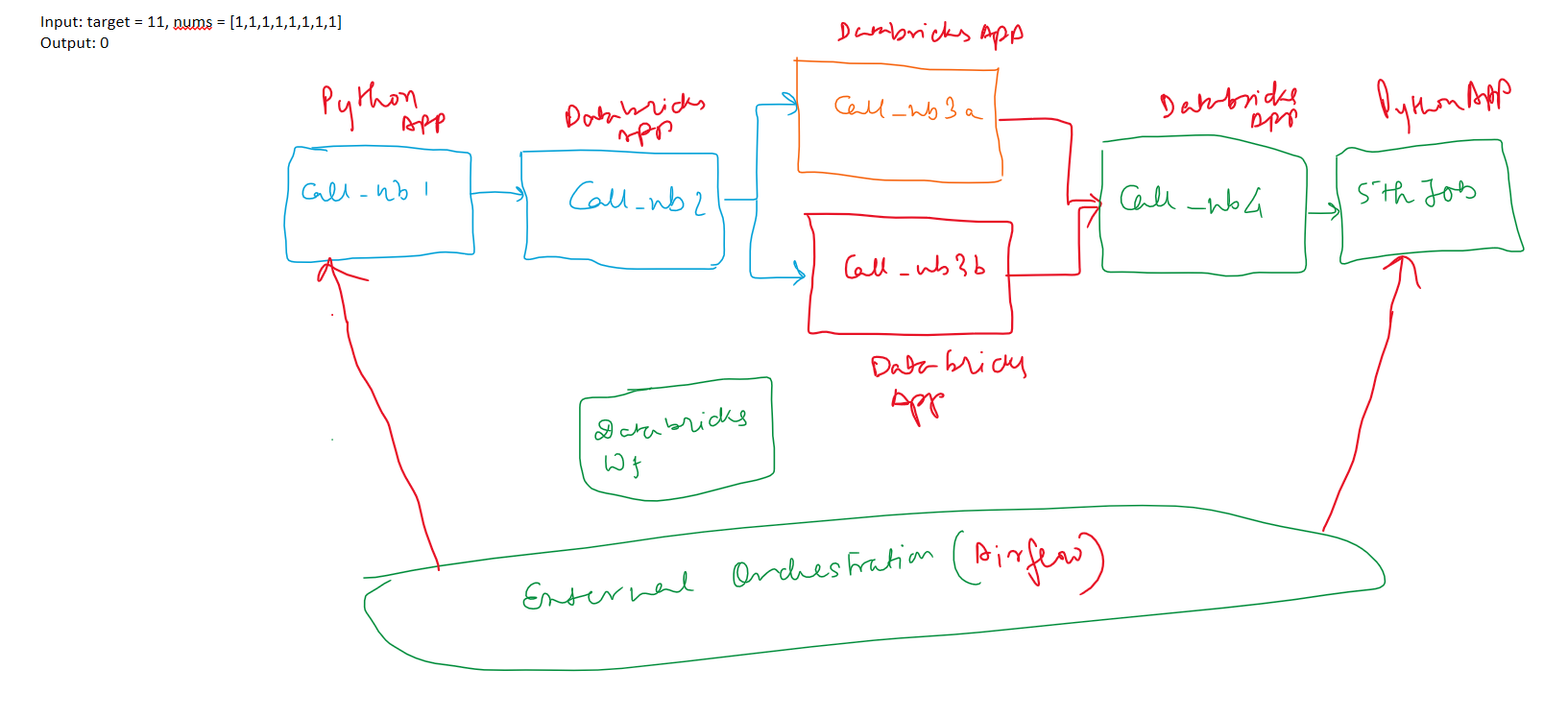


You can schedule the job using the below options:



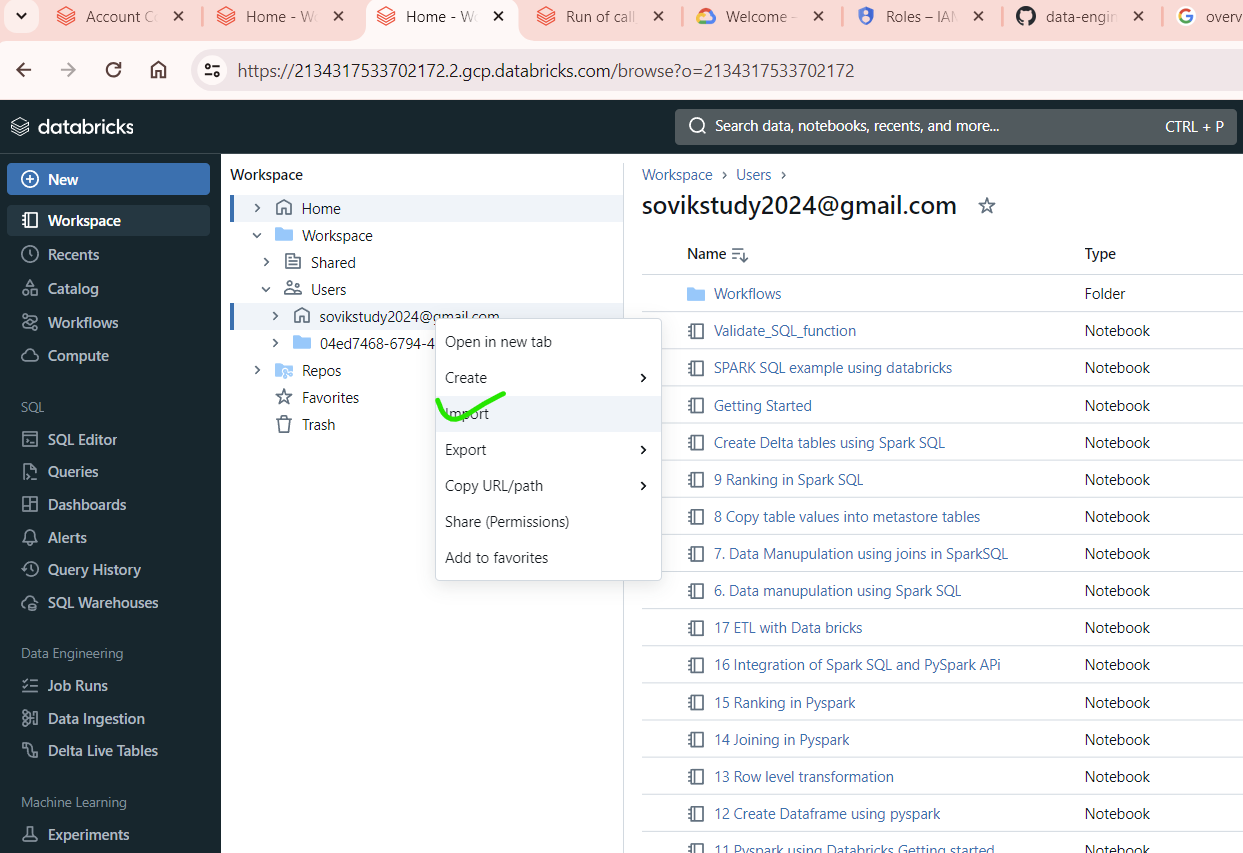
Our Pipeline overview:



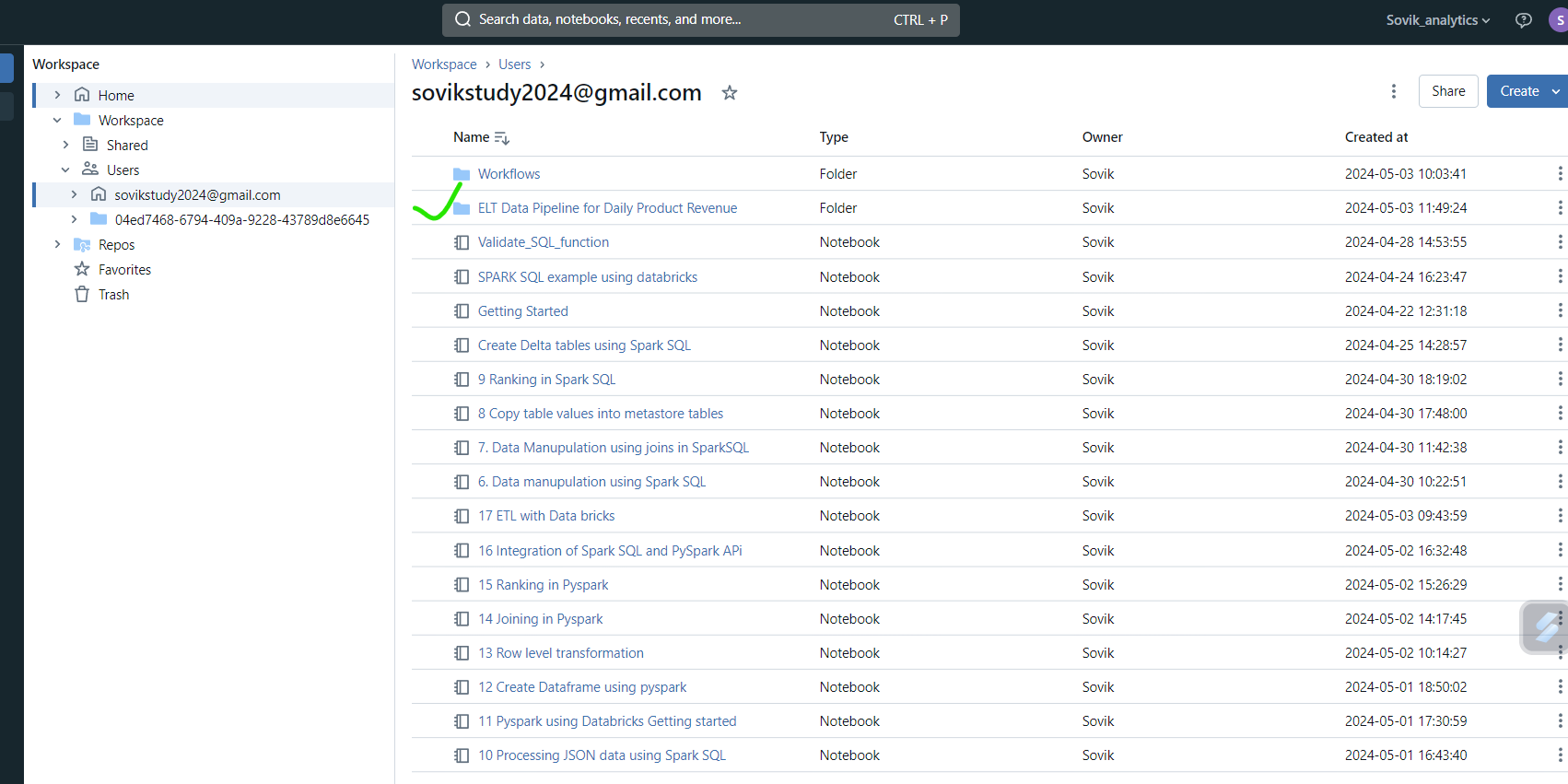


Link: <https://github.com/itversity/data-engineering-on-gcp/tree/main/notebooks/08%20ELT%20Data%20Pipelines%20using%20Dataproc%20on%20GCP>

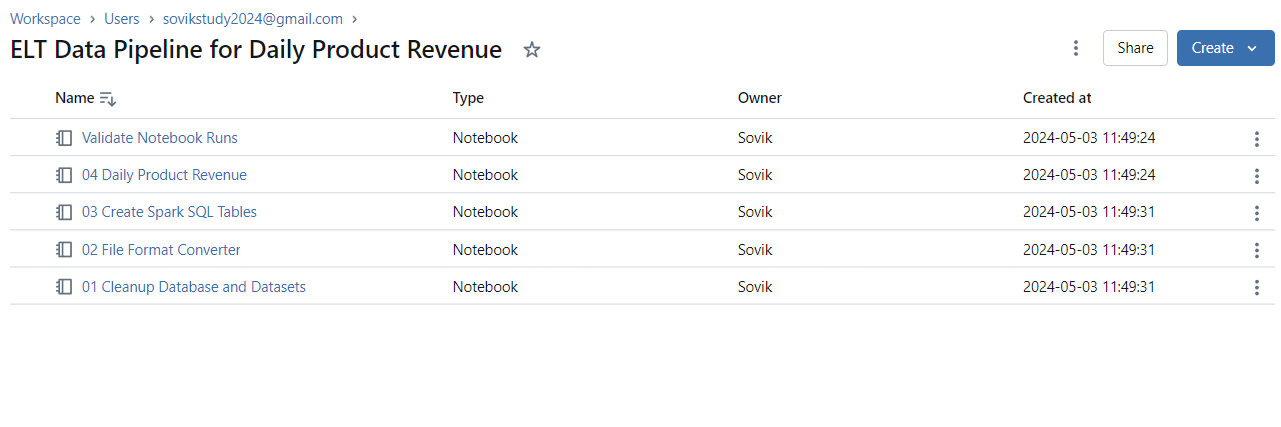
Now import the external db source:







For ETL we have the following notebooks



1. Clean up data base:
2. *-- Databricks notebook source*
3. DROP DATABASE IF EXISTS itversity\_retail\_bronze CASCADE
4. *-- COMMAND ----------*
5. –- remove bronze\_base\_dir if present
6. *-- MAGIC %python*
7. *-- MAGIC dbutils.fs.rm(dbutils.widgets.get('bronze\_base\_dir'), recurse=True)*
8. *-- COMMAND ----------*
9. –- remove *'gold\_base\_dir'* if present
10. *-- MAGIC %python*
11. *-- MAGIC dbutils.fs.rm(dbutils.widgets.get('gold\_base\_dir'), recurse=True)*
12. *-- COMMAND ----------*

2. File format Converter

*# Databricks notebook source*

dbutils.widgets.removeAll()

*# COMMAND ----------*

--For adhoc run

dbutils.widgets.text('src\_base\_dir', '', *label*='Enter Source Base Dir')

dbutils.widgets.text('bronze\_base\_dir', '', *label*='Enter Target Base Dir')

dbutils.widgets.text('ds', '', *label*='Enter Dataset Name')

*# COMMAND ----------*

src\_base\_dir = dbutils.widgets.get('src\_base\_dir')

*# COMMAND ----------*

bronze\_base\_dir = dbutils.widgets.get('bronze\_base\_dir')

*# COMMAND ----------*

ds = dbutils.widgets.get('ds')

*# COMMAND ----------*

*import* json

def get\_columns(*schemas\_file*, *ds\_name*):

    schema\_text = spark.read.text(*schemas\_file*, *wholetext*=True).first().value

    schemas = json.loads(schema\_text)

    column\_details = schemas[*ds\_name*]

    columns = [col['column\_name'] *for* col *in* sorted(column\_details, *key*=lambda *col*: *col*['column\_position'])]

*return* columns

*# COMMAND ----------*

ds

*# COMMAND ----------*

print(f'Processing {ds} data')

columns = get\_columns(f'dbfs:{src\_base\_dir}/schemas.json', ds)

df = spark. \

    read. \

    csv(f'{src\_base\_dir}/{ds}', *inferSchema*=True). \

    toDF(\*columns)

*# COMMAND ----------*

df.show()

*# COMMAND ----------*

df.write. \

    mode('overwrite'). \

    parquet(f'{bronze\_base\_dir}/{ds}')

3.Create Spark SQL Tables

*-- Databricks notebook source*

CREATE DATABASE IF NOT EXISTS itversity\_retail\_bronze

*-- COMMAND ----------*

USE itversity\_retail\_bronze

*-- COMMAND ----------*

CREATE EXTERNAL TABLE IF NOT EXISTS ${table\_name}

USING PARQUET

OPTIONS (

    path='${bronze\_base\_dir}/${table\_name}'

)

4.Creat the table

*-- COMMAND ----------*

SHOW tables

*-- COMMAND ----------*

INSERT OVERWRITE DIRECTORY '${gold\_base\_dir}/daily\_product\_revenue'

USING PARQUET

SELECT o.order\_date,

    oi.order\_item\_product\_id,

    round(sum(oi.order\_item\_subtotal), 2) AS revenue

FROM orders AS o

    JOIN order\_items AS oi

        ON o.order\_id = oi.order\_item\_order\_id

WHERE o.order\_status IN ('COMPLETE', 'CLOSED')

GROUP BY 1, 2

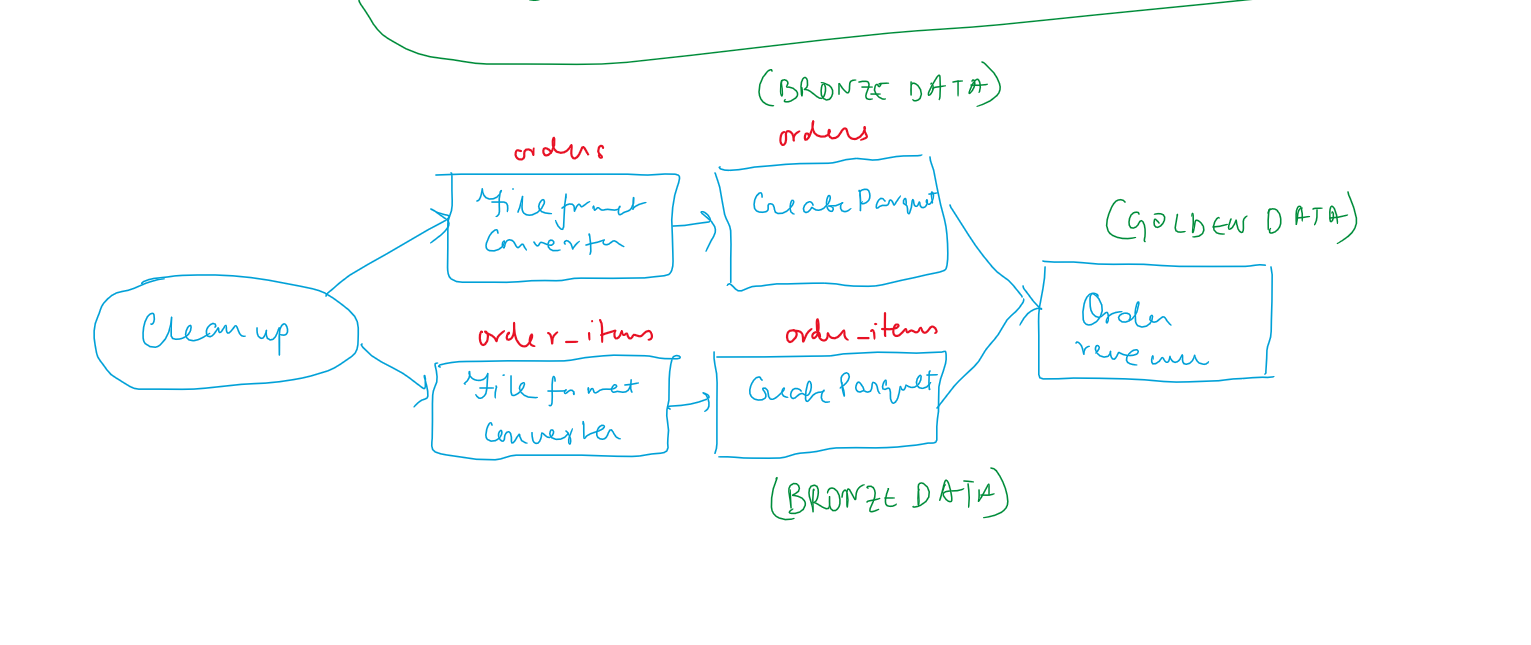
*-- COMMAND ----------*

SELECT \* FROM PARQUET.`${gold\_base\_dir}/daily\_product\_revenue`

ORDER BY 1, 2 DESC

1. Validate
2. *-- Databricks notebook source*
3. dbutils.fs.mkdirs("dbfs:/public/retail\_db\_bronze")
4. dbutils.fs.mkdirs("dbfs:/public/retail\_db\_gold")
5. *-- MAGIC %run "./Cleanup Database and Datasets" $bronze\_base\_dir=/public/retail\_db\_bronze $gold\_base\_dir=/public/retail\_db\_gold*
6. *-- COMMAND ----------*
7. *-- MAGIC %run "./File Format Converter" $ds=orders $src\_base\_dir=/public/retail\_db $bronze\_base\_dir=/public/retail\_db\_bronze*
8. *-- COMMAND ----------*
9. *-- MAGIC %run "./File Format Converter" $ds=order\_items $src\_base\_dir=/public/retail\_db $bronze\_base\_dir=/public/retail\_db\_bronze*
10. *-- COMMAND ----------*
11. *-- MAGIC %run "./Create Spark SQL Tables" $table\_name=orders $bronze\_base\_dir=/public/retail\_db\_bronze*
12. *-- COMMAND ----------*
13. *-- MAGIC %run "./Create Spark SQL Tables" $table\_name=order\_items $bronze\_base\_dir=/public/retail\_db\_bronze*
14. *-- COMMAND ----------*
15. SHOW tables
16. *-- COMMAND ----------*
17. SELECT count(\*) FROM orders
18. *-- COMMAND ----------*

Our Pipeline:

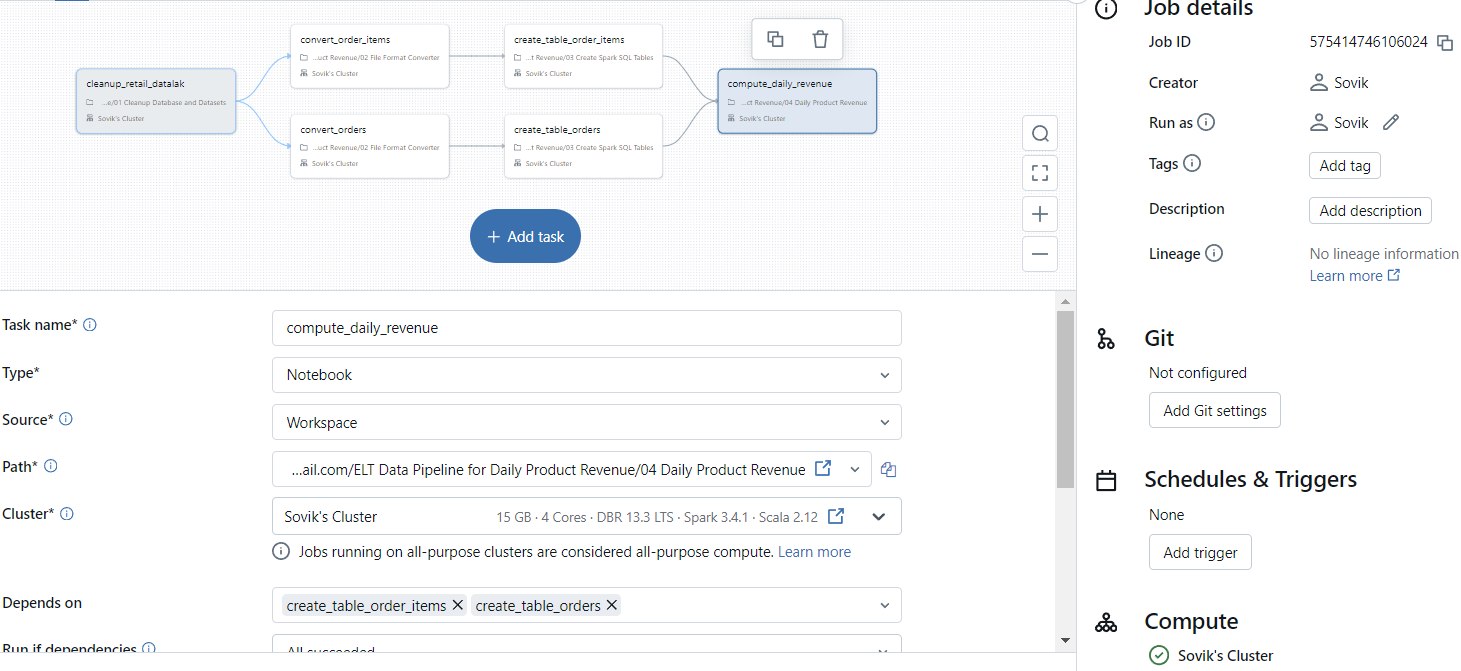


Global Dirs:

/public/retail\_db\_bronze\_v2=bronze\_base\_dir

/public/retail\_db\_gold=gold\_base\_dir

/public/retail\_db=src\_base\_dir

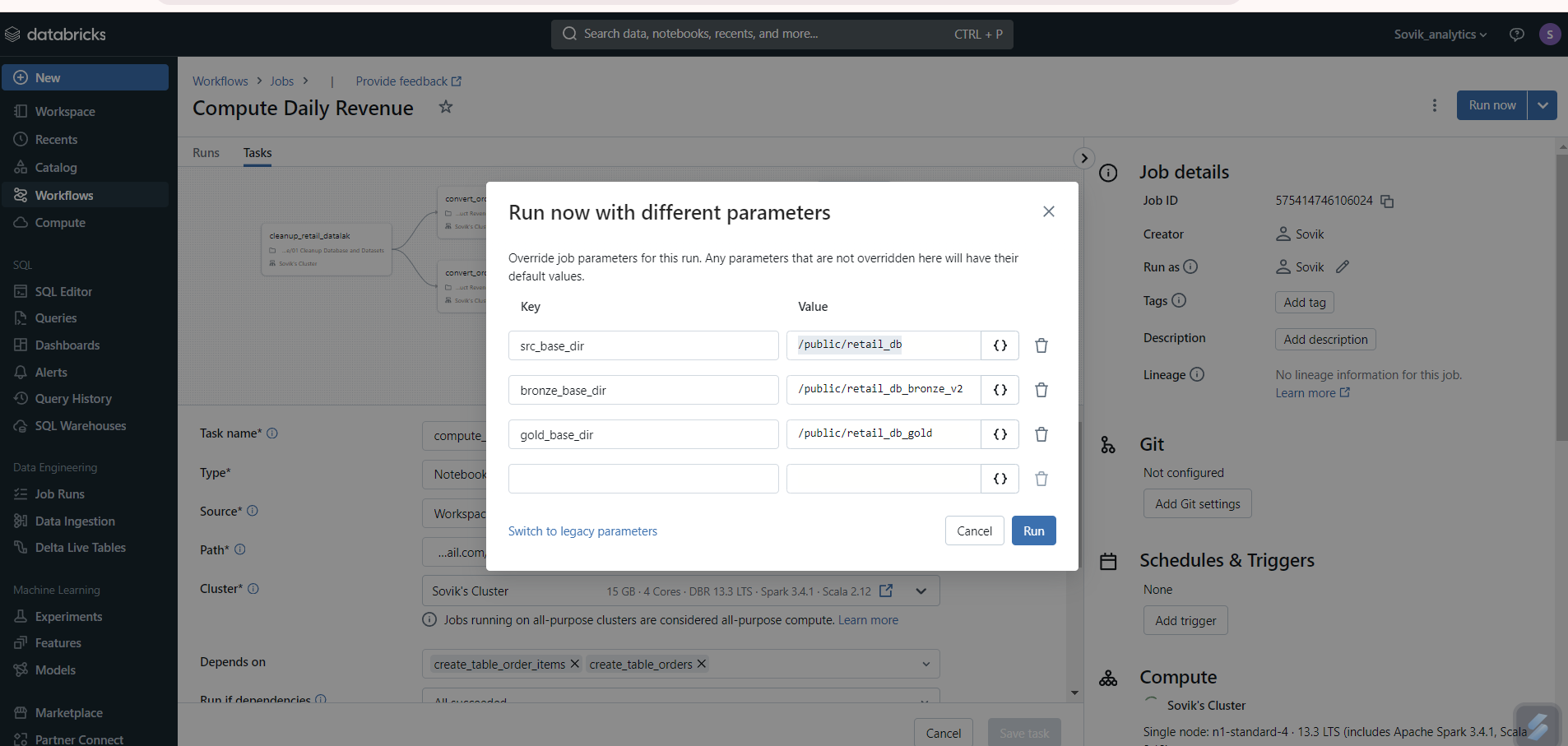


**Local parameters**

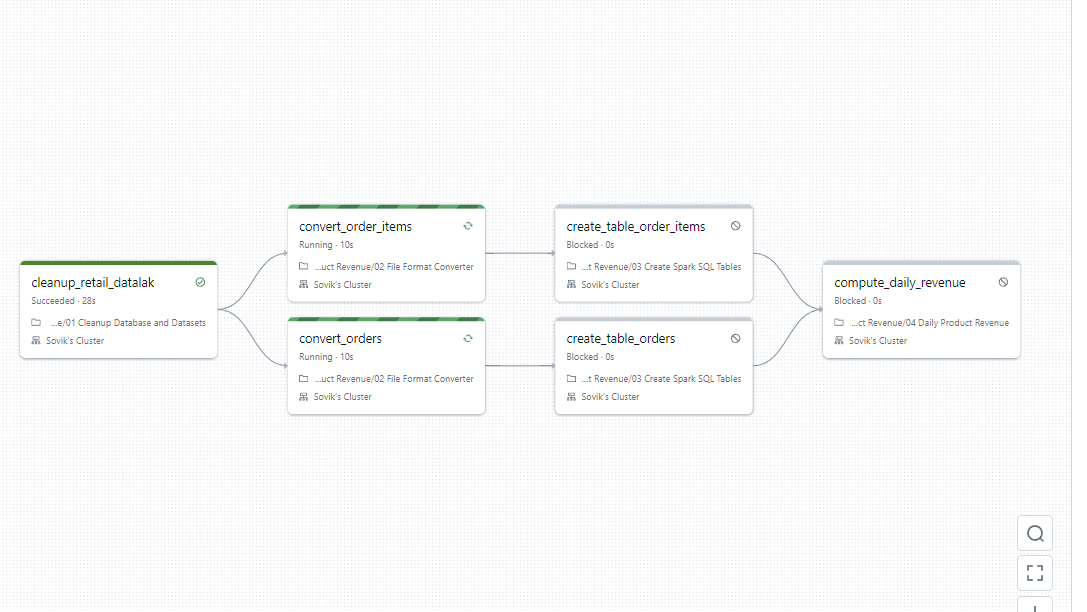
Create\_orders=ds:orders

Create\_order\_items=ds:order\_items

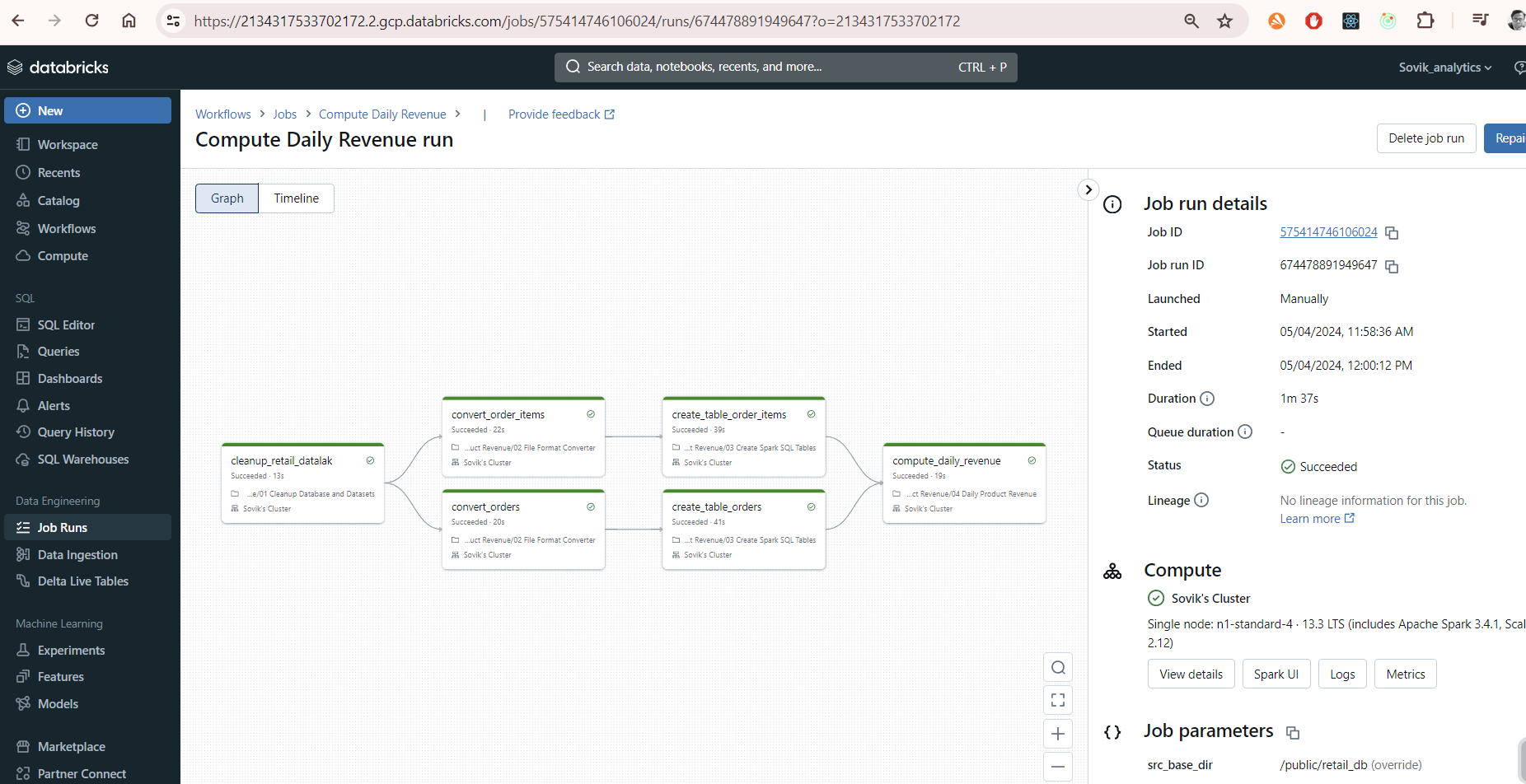
Runtime parameters:



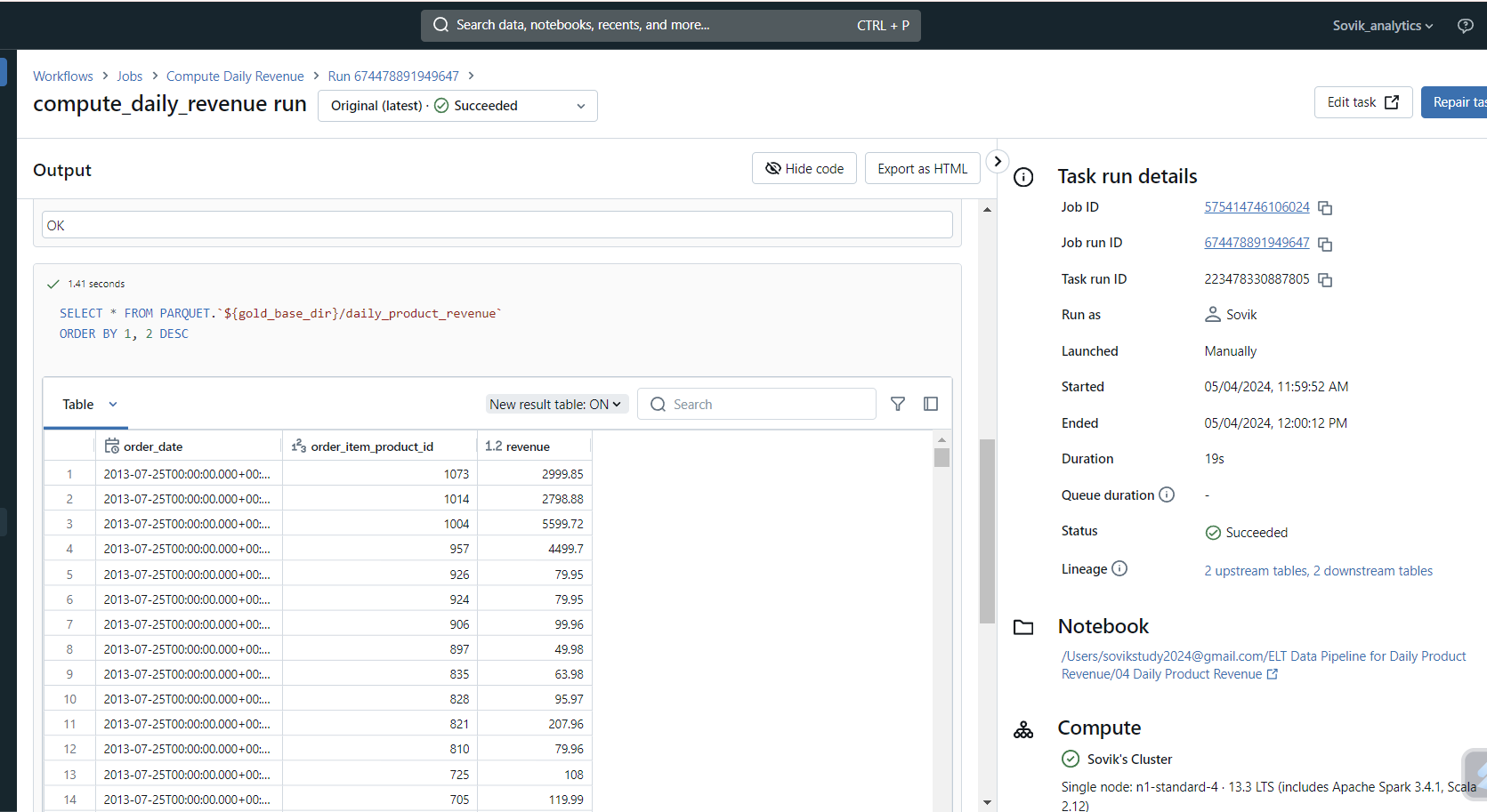
Running:



Suceeded:

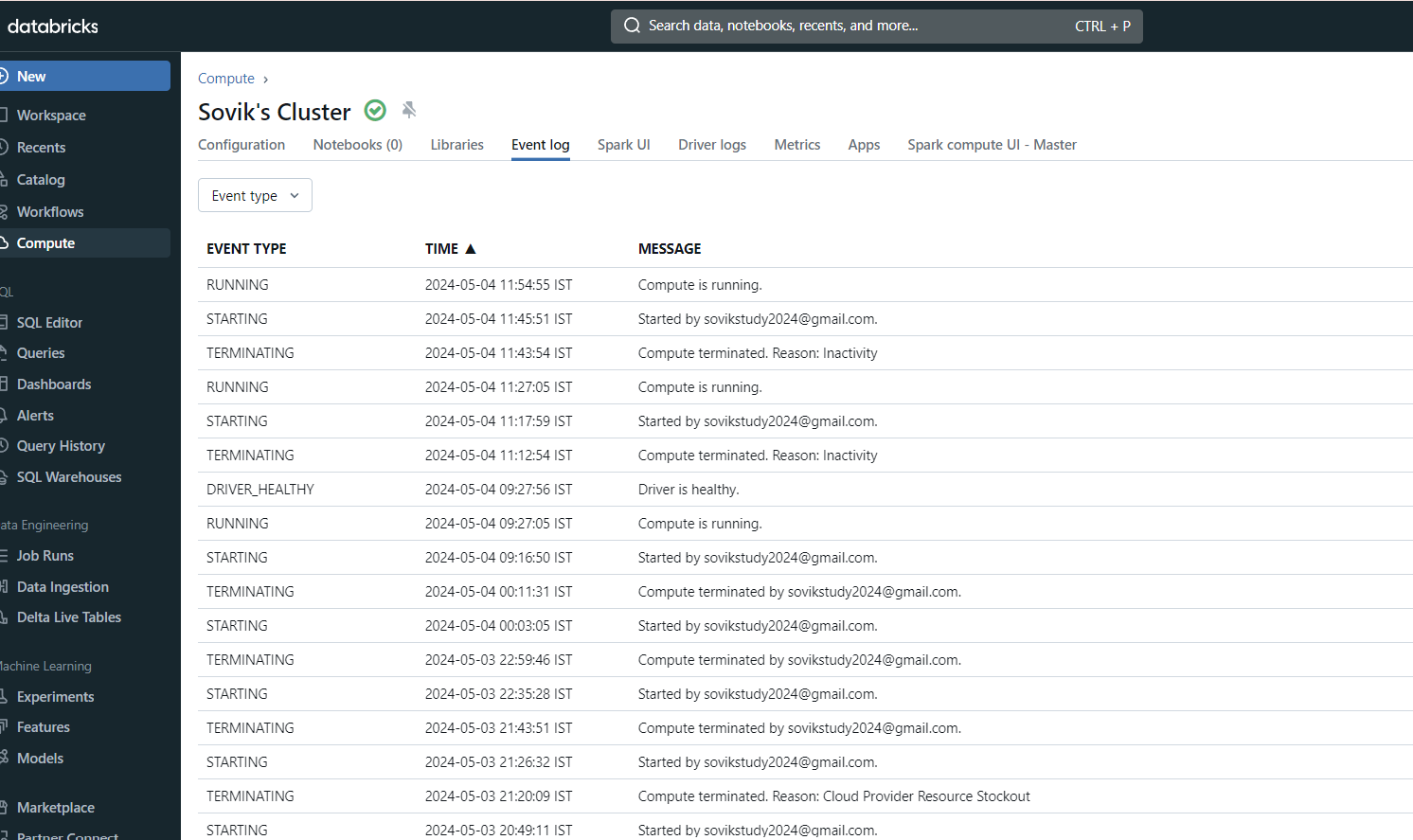


Op:

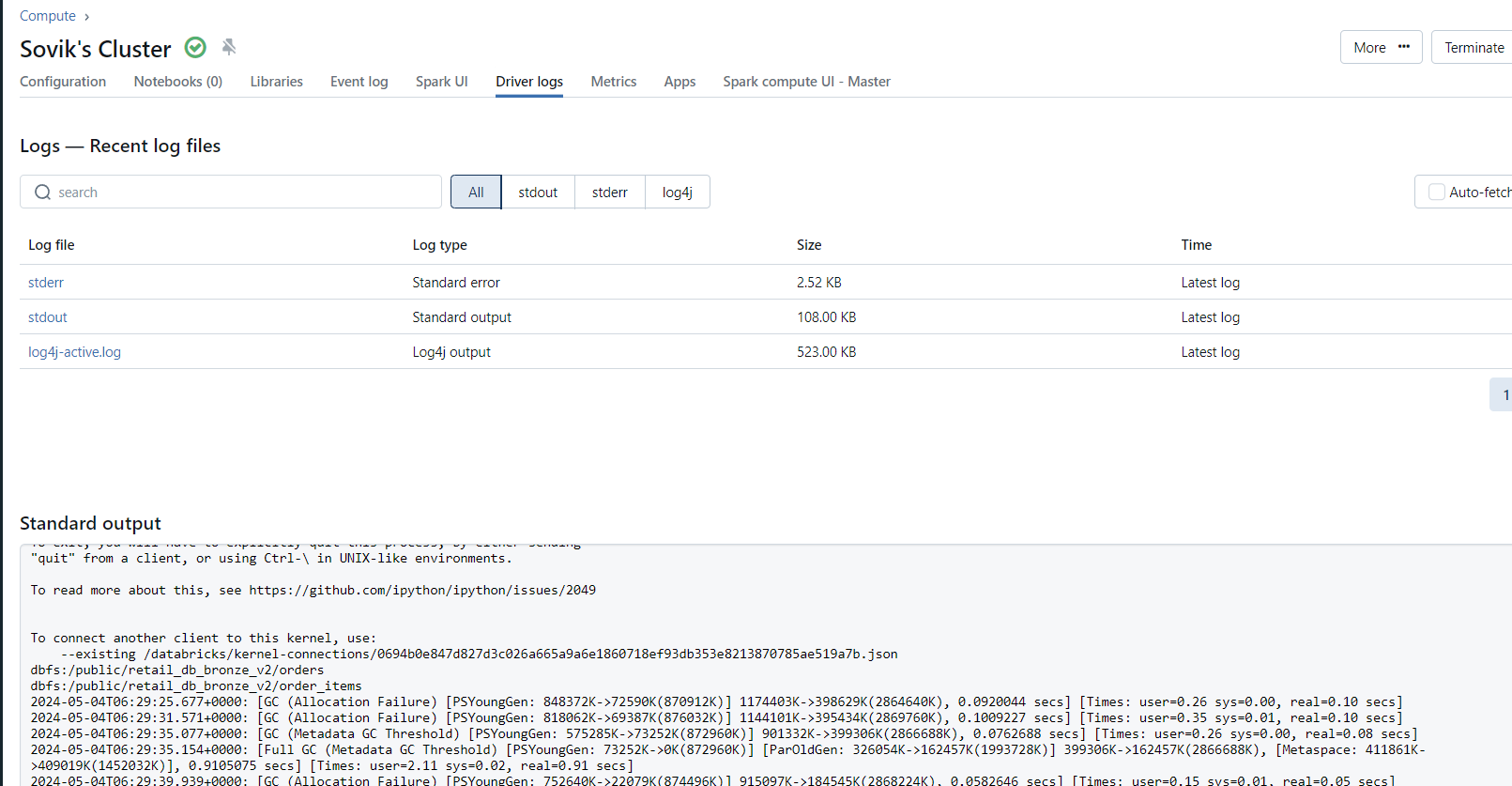


Logs:

1. Event logs



1. Driver logs





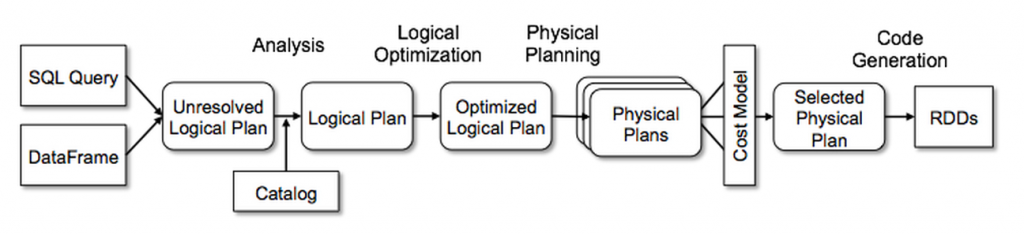
**Catalyst optimizer**

The core of Spark SQL is the Catalyst Optimizer. It leverages intelligent programming language features (such as Scala's pattern matching and quasi quotes) in novel ways to build scalable query optimizers. Catalyst is based on the functional programming structure written in Scala and was designed with the following two purposes in mind:

Easily add new optimization techniques and features to Spark SQL

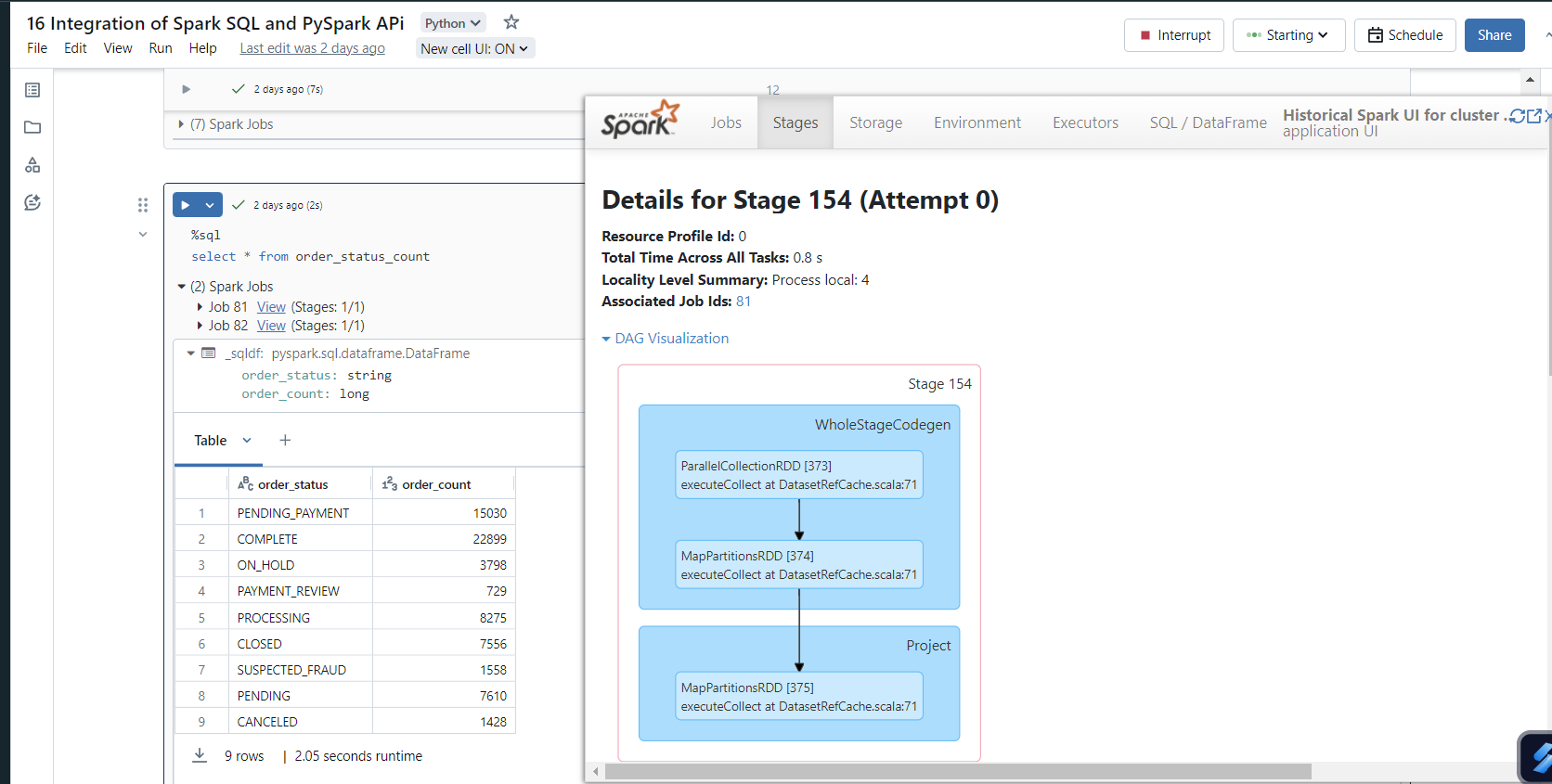
Supports external developers to extend the optimizer (e.g., add data source-specific rules, support new data types, etc.)

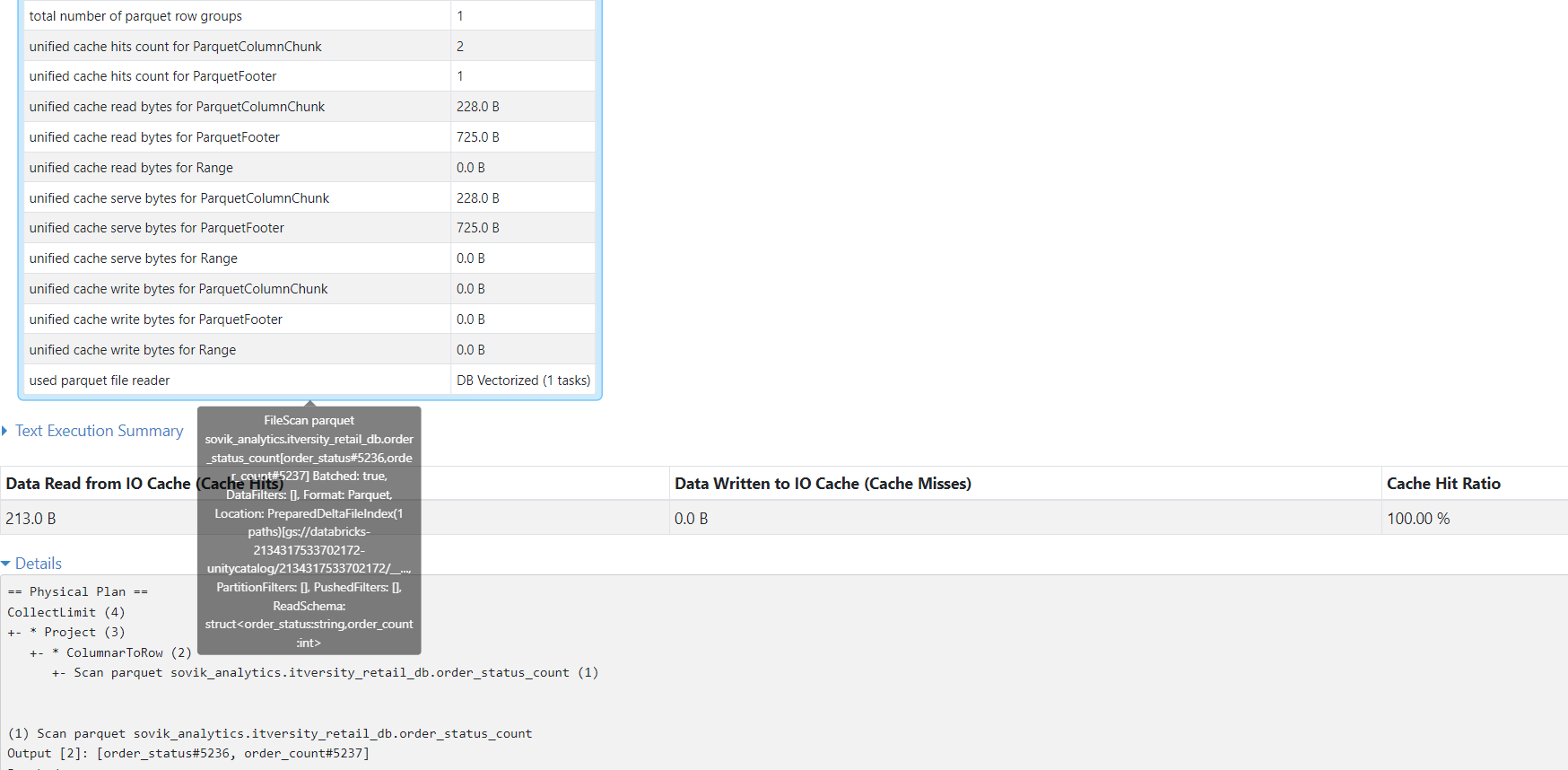
Catalyst Optimizer Diagram



Catalyst includes generic libraries to represent trees and apply rules to manipulate them. Based on this framework, there are libraries for each relational query processing (e.g. expressions, logical query plans) and several sets of rules that handle different stages of query execution. That is, it compiles various parts of the query into Java bytecode through analysis, logical optimization, physical planning, and code generation. In the latter case, another Scala feature, quasiquote, is used to conveniently generate code at runtime from composable expressions. Catalyst also provides several public extension points, including external data sources and custom types. Catalyst also supports both rule-based and cost-based optimization. [glossary-cta]

Visualization of jobs:





**Generate plans**

help(df\_name.explain)

df\_name.explain(True)

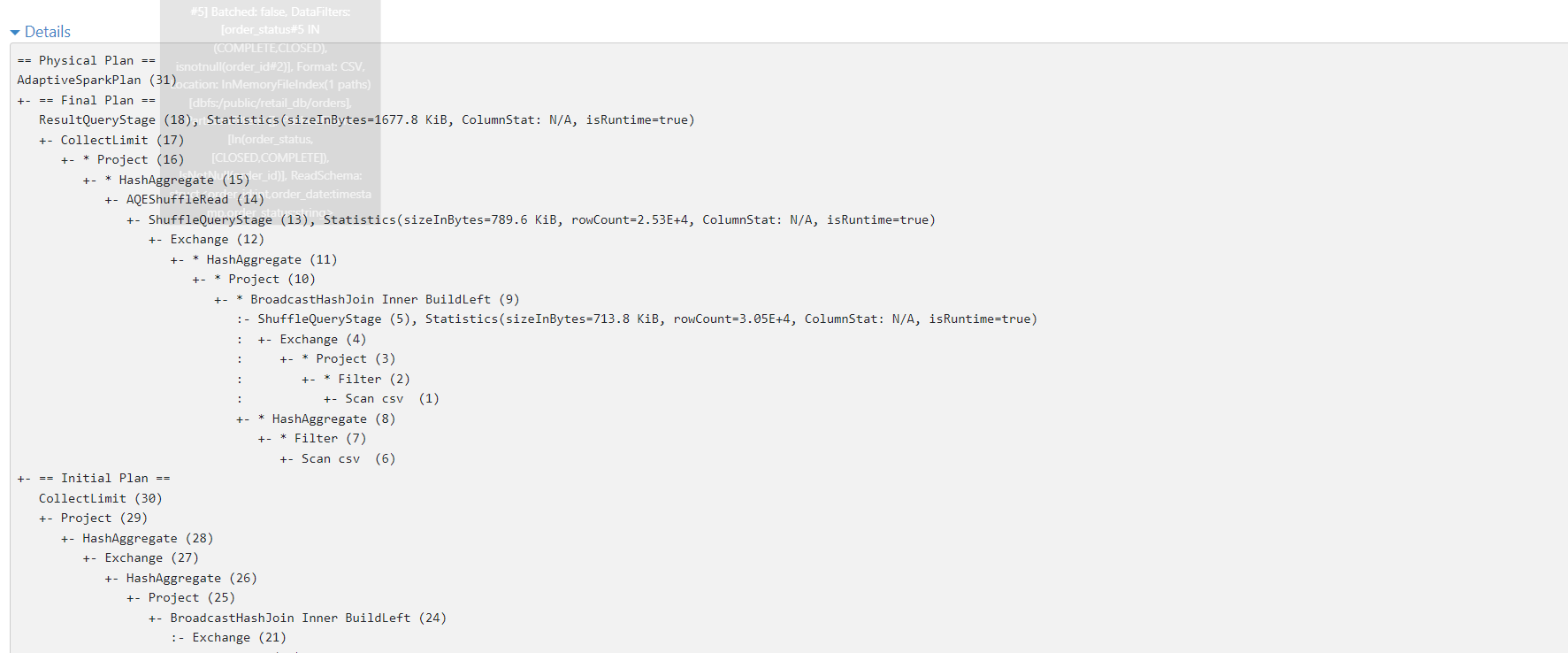
df\_name.explain(‘extended’)

SQL Flavour:

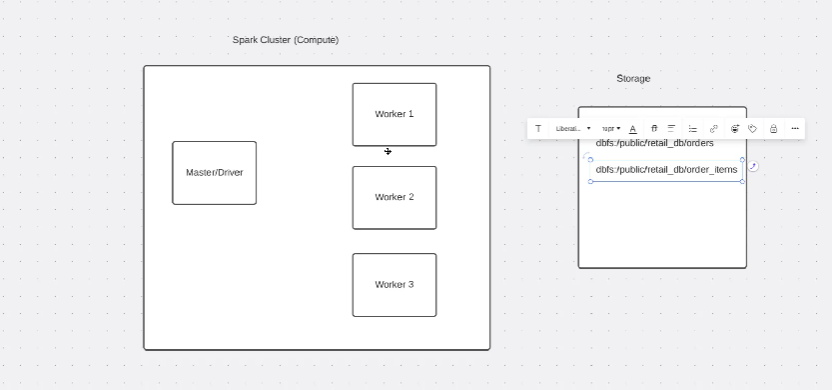
EXPLAIN

<SQL\_QUERY>

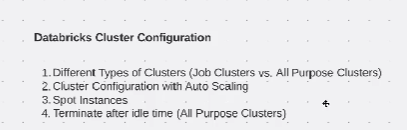
Final Plan:



Databricks Architecture:



**Performance Optimizer of Clusters:**



**All purpose vs job cluster:**

**Cluster Purpose**:

* **All-purpose clusters**, such as ad hoc analysis, data exploration, and development, are designed for collaborative use. Multiple users can share them.
* **On** the other hand, job clusters are specifically for running automated jobs. [They terminate once the job is completed, reducing resource usage and cost](https://learn.microsoft.com/en-us/azure/databricks/clusters/cluster-config-best-practices).

**SLA Requirements**:

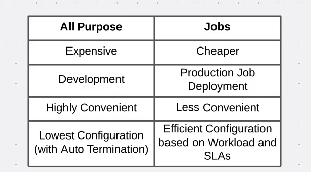
* A job cluster might be more suitable if your workflow requires meeting a **strict Service Level Agreement (SLA)**. Job clusters are dedicated to a specific task and can be optimized for performance.
* All-purpose clusters, while versatile, may not provide the same level of predictability in meeting SLAs.

**Resource Usage and Cost**:

* All-purpose clusters are more cost-effective for tasks like exploration and development, where resource sharing is beneficial.
* Job clusters are more efficient for focused, time-sensitive tasks, as they release resources promptly after completion.

**Trade-offs**:

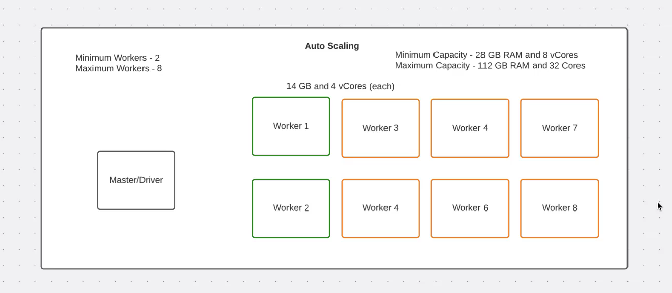
* **All-purpose clusters** allow flexibility but may not guarantee immediate availability due to resource sharing.
* **Job clusters** prioritize your specific job but come with higher resource costs.

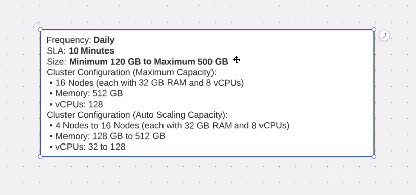


Instead of using %fs ls dbfs:/….

We can write : dbutils.fs.ls(“dbfs:/….”)

ii. dbutils.fs.ls(“dbfs:/….”)[0].path





Create a job

1. Do one of the following:
   * Click Jobs Icon **Workflows** in the sidebar and click Create Job Button.
   * In the sidebar, click New Icon **New** and select **Job**.

The **Tasks** tab appears with the create task dialog along with the **Job details** side panel containing job-level settings.



1. Replace **New Job…** with your job name.
2. Enter a name for the task in the **Task name** field.
3. In the **Type** drop-down menu, select the type of task to run. See [Task type options](https://docs.databricks.com/en/workflows/jobs/create-run-jobs.html#task-types).
4. Configure the cluster where the task runs. By default, serverless compute is selected if your workspace is in a Unity Catalog-enabled workspace and you have selected a task supported by serverless compute for workflows. See [Run your Databricks job with serverless compute for workflows](https://docs.databricks.com/en/workflows/jobs/run-serverless-jobs.html). If serverless compute is not available, or you want to use a different compute type, you can select a new job cluster or an existing all-purpose cluster in the **Compute** dropdown menu.
   * **New Job Cluster**: Click **Edit** in the **Cluster** drop-down menu and complete the [cluster configuration](https://docs.databricks.com/en/compute/configure.html).
   * **Existing All-Purpose Cluster**: Select an existing cluster in the **Cluster** drop-down menu. To open the cluster on a new page, click the External Link icon to the right of the cluster name and description.

To learn more about selecting and configuring clusters to run tasks, see [Use Databricks compute with your jobs](https://docs.databricks.com/en/workflows/jobs/use-compute.html).

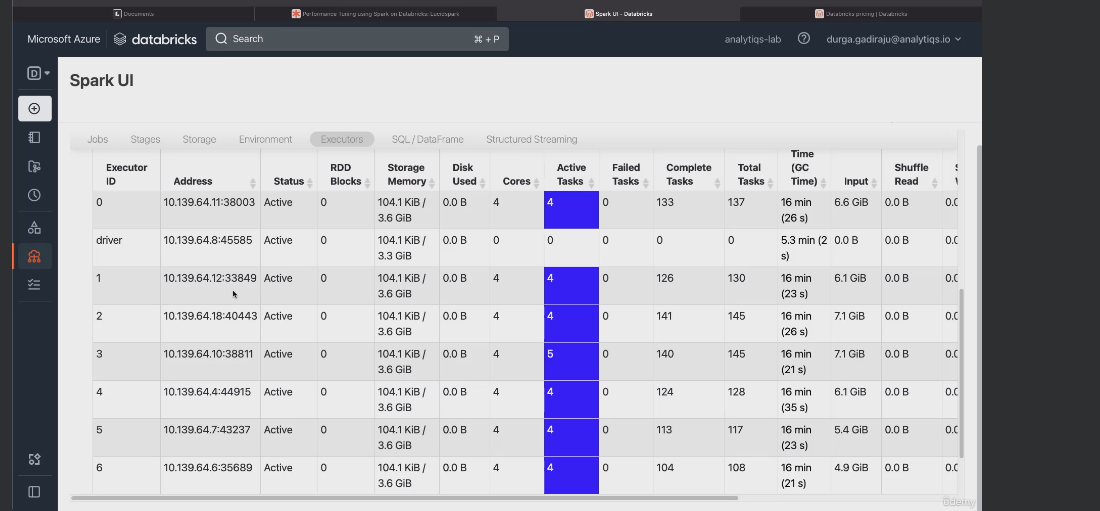
1. To add dependent libraries, click **+ Add** next to **Dependent libraries**. See [Configure dependent libraries](https://docs.databricks.com/en/workflows/jobs/settings.html#task-config-dependent-libraries).
2. You can pass parameters for your task. For information on the requirements for formatting and passing parameters, see [Pass parameters to a Databricks job task](https://docs.databricks.com/en/workflows/jobs/create-run-jobs.html#task-parameters).
3. To optionally receive notifications for task start, success, or failure, click **+ Add** next to **Emails**. Failure notifications are sent on initial task failure and any subsequent retries. To filter notifications and reduce the number of emails sent, check **Mute notifications for skipped runs**, **Mute notifications for canceled runs**, or **Mute notifications until the last retry**.
4. To optionally configure a retry policy for the task, click **+ Add** next to **Retries**. See [Configure a retry policy for a task](https://docs.databricks.com/en/workflows/jobs/settings.html#retry-policies).
5. To optionally configure the task’s expected duration or timeout, click **+ Add** next to **Duration threshold**. See [Configure an expected completion time or a timeout for a task](https://docs.databricks.com/en/workflows/jobs/settings.html#timeout-setting-task).
6. Click **Create**.

After creating the first task, you can configure job-level settings such as notifications, job triggers, and permissions. See [Edit a job](https://docs.databricks.com/en/workflows/jobs/settings.html#job-edit).

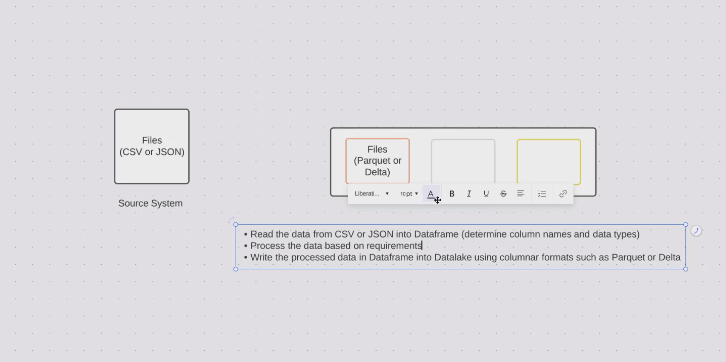
To add another task, click Add Task Button in the DAG view. A shared cluster option is provided if you have selected **Serverless** compute or configured a **New Job Cluster** for a previous task. You can also configure a cluster for each task when you create or edit a task. To learn more about selecting and configuring clusters to run tasks, see [Use Databricks compute with your jobs](https://docs.databricks.com/en/workflows/jobs/use-compute.html).

You can optionally configure job-level settings such as notifications, job triggers, and permissions. See [Edit a job](https://docs.databricks.com/en/workflows/jobs/settings.html#job-edit). You can also configure job-level parameters that are shared with the job’s tasks. See [Add parameters for all job tasks](https://docs.databricks.com/en/workflows/jobs/settings.html#job-parameters).

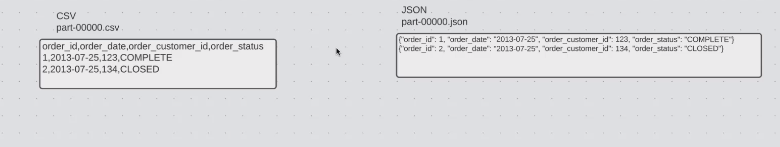
Typical job run output:



**Tuning performance using infer schema**



CSV and JSON files:



Inferschema tuning:

airlines=spark.read.csv('path',headers=True,

inferschema=True)------🡪Took time in minutes[file is partitioned]

We do it in the below approach:

* 1. Get schema

Airlines\_schema=spark.read.csv('path../part-0000',headers=True,

inferschema=True).schema------🡪Extracting only from 1 partition

* 1. Get data

airlines=spark.read.csv(''path../part-\*',headers=True,

schema=airline\_schema)------🡪Extracting data from all partition will take seconds

**Performance tuning of using columnar format and partition strategy**

**Find the size of table by partition:**

files=[]

*for* f *in* dbutils.fs.ls("path"):

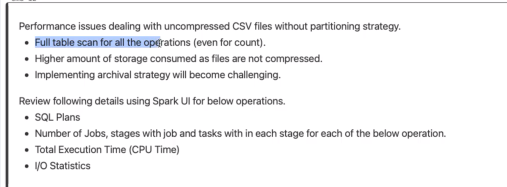
*if* f.name.starswith('part-'):

        files.append(f.name.f.size)

size=sum([f[1] *for* f *in* files])/(1024\*1024\*1024)

**whole size is extracted**

**side effect of using csv as data lake data:**

****

*##resturcture csv file to parquet file for custom partition strategy*

*#suupose airlines\_df is in csv formatfirst by year then by each moth in year dir*

airlines\_df.write.partitionBy('Year','Month').mode('overwrite').parquet('path')

**check each partition file size:**

files=[]

*for* f *in* dbutils.fs.ls("path"):

*if* f.name.starswith('Year-'):

*for* yf *in* dbutils.fs.ls(f.path):

*if* mf.name.starswith('Month-'):

*for* mf *in* dbutils.fs.ls(yf.path):

*if* mf.name.endswith('snappy.parquet'):

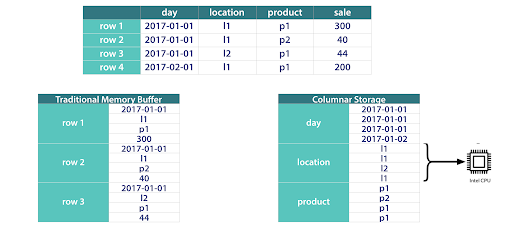
                        files.append(mf.name,mf.size)

**Collate the individual sums**

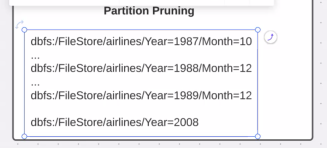
size=sum([f[1] *for* f *in* files])/(1024\*1024\*1024)

**Note: snappy in parquet is the compression alogoritm which brings down the the data execution to mbs than whole file in csv**

**Columnar file format**



**Folder structure of partitioned table:**

****

**Problem statements:**

****

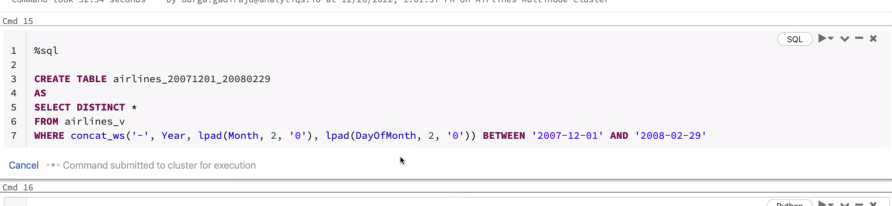
**Solution:**

**For all we create a view and then we do filtering**

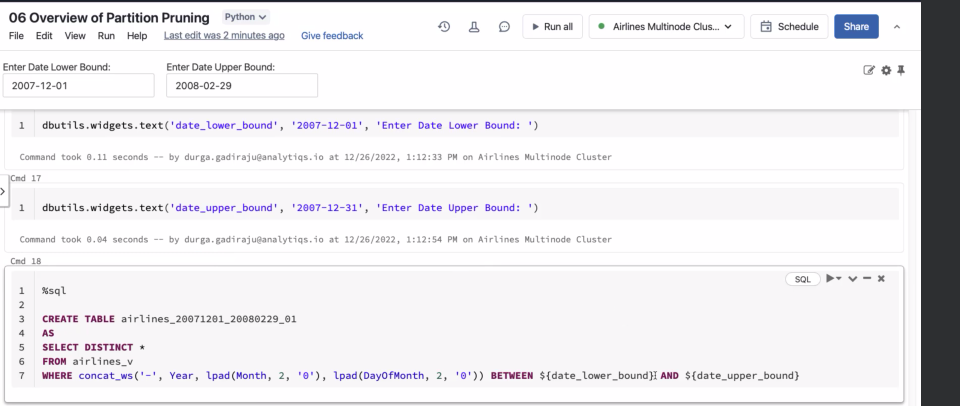
**For 5th solution without partition pruning we can do as below:**

****

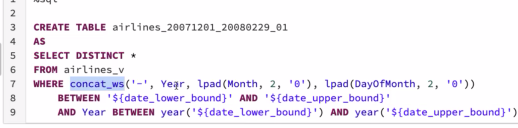
**Now we create the table:**

****

**Partiton pruning(parameterized):**

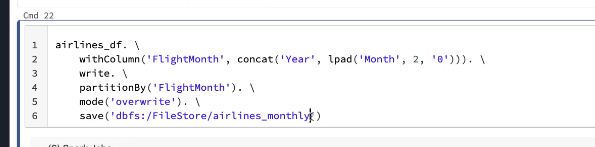
****

**We do a little more parameterization:**

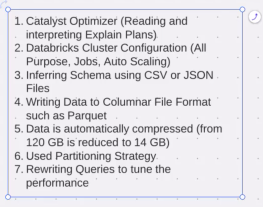
****

**This will take smaller space as it is being done only within the year range**

**Now we can redesign the table based on the below for queries like the 5th problem:**

****

**Concusion:**

****