Priority & conflict testing (this is where rule engines usually surprise)

GOAL :

specifically called out **priority-based rule execution**, so test conflicts on purpose:

A. Generic vs specific

- Rule 1: `path=/reports/*`, ttl=300s, priority=10
- Rule 2: `path=/reports/monthly`, ttl=30s, priority=20

Hit `/reports/monthly`:

- Which rule gets applied?
- If you change the priority numbers or re-order them in config, does behavior change as documented?

B. Multi-dimension specificity

Create three overlapping rules:

1. Route-only rule (`/reports/*`)
2. Route + tenant rule (`/reports/*` + `tenant=premium`)
3. Route + tenant + role rule (`/reports/*` + `tenant=premium` + `role=admin`)

Drive traffic for:

- standard user @ standard tenant
- standard user @ premium tenant
- admin @ premium tenant

Confirm the “most constrained” rule wins in each case. If it doesn’t, that’s a design/UX insight for the product backlog.

- spin up the **Getting Started with Data Engineering using Snowflake Notebooks** demo.
- Your app (behind AirBrx) exposes HTTP endpoints that query:
  - `LOCATION`
  - `ORDER_DETAIL`
  - `DAILY_CITY_METRICS` ✅ main "reporting" table

---

1. Map the Snowflake demo to simple APIs

Once you've run the Snowflake guide and have `DEMO_DB.DEMO_SCHEMA` set up (LOCATION, ORDER_DETAIL, DAILY_CITY_METRICS, etc.), expose a tiny REST API in front of it (Python/Node/whatever is fastest):

Example endpoints:

- `GET /api/locations`
  → `SELECT * FROM DEMO_DB.DEMO_SCHEMA.LOCATION LIMIT 1000`
- `GET /api/orders/by_city?city={CITY}&day={DATE}`
  → query `ORDER_DETAIL` joined to `LOCATION`
- `GET /api/metrics/daily_city?city={CITY}&day={DATE}`
  → query `DAILY_CITY_METRICS` (this is your "report" endpoint)

These 3 routes are enough to test rule priority.

Also decide on **headers** you'll use for rules:

- `x-tenant` → simulate customer/tenant (`standard`, `premium`)
- `x-role` → `user`, `analyst`, `admin`
- `x-origin` (optional) → `dashboard`, `explore`

You don't need real multi-tenant Snowflake; headers are just for the gateway + caching.

---

2. Priority rules – Generic vs Specific (tied to DAILY_CITY_METRICS)

We'll translate the earlier `/reports/*` example into your `daily_city_metrics` API.

Step 1 – Create a generic caching rule

> Rule 1 – Default metrics cache

- Match:
  - `method = GET`
  - `path = /api/metrics/*`
- Action:
  - `ttl = 300s`
  - `priority = 10`

Meaning: all metrics endpoints are cached for 5 minutes.

Step 2 – Create a more specific rule for the hot endpoint

> Rule 2 – Specific route override

- Match:
  - `method = GET`
  - `path = /api/metrics/daily_city`
- Action:
  - `ttl = 60s` (or 30s)
  - `priority = 20`

Test:

1. Call
   `GET /api/metrics/daily_city?city=PORTLAND&day=2023-08-01`
   a few times.
2. Inspect AirBrx logs / metrics:
   - Which rule fired? It **should** be Rule 2 (higher priority).
   - First request: Snowflake hit.
   - Subsequent requests within 60s: cache hits.
3. Temporarily lower Rule 2 priority to `5`:
   - Repeat the call.

- Now Rule 1 should win (you've just verified priority ordering actually works).

You've now done a **concrete, data-backed generic vs specific rule test** on the Snowflake demo.

---

3. Priority rules – Multi-dimension specificity using tenants & roles

Now we layer tenants and roles, again on top of `daily_city_metrics`.

Configure three overlapping rules

> Rule A – Route-only (fallback)

- Match: `GET /api/metrics/daily_city`
- TTL: `300s`
- Priority: `10`

> Rule B – Route + Tenant

- Match:
  - `GET /api/metrics/daily_city`
  - Header `x-tenant = premium`
- TTL: `60s`
- Priority: `20`

> Rule C – Route + Tenant + Role

- Match:
  - `GET /api/metrics/daily_city`
  - `x-tenant = premium`
  - `x-role = admin`
- TTL: `0s` (no cache) **or** explicitly `cache=false`
- Priority: `30`

Drive three user types through the gateway

1. **Standard user @ standard tenant**

```
GET /api/metrics/daily_city?city=PORTLAND&day=2023-08-01
x-tenant: standard
x-role: user
```

   - Should match **Rule A** (only route condition satisfied).
   - Expect TTL=300s.

2. **Standard user @ premium tenant**

```
GET /api/metrics/daily_city?city=PORTLAND&day=2023-08-01
x-tenant: premium
x-role: user
```

   - Should match **Rule B**.
   - TTL=60s (fresher cache for premium tenants).

3. **Admin @ premium tenant**

```
GET /api/metrics/daily_city?city=PORTLAND&day=2023-08-01
x-tenant: premium
x-role: admin
```

   - Should match **Rule C**.
   - No caching → always hits Snowflake.

What to verify

For each of the three request patterns:

- **Which rule ID / name** AirBrx logs as applied.
- **TTL / cache headers** sent back.
- Snowflake queries:
  - Standard + Premium users share same query but different TTLs.

- Admin always causes a Snowflake hit (no cache).

If you flip Rule C's priority below Rule B (e.g., `priority=15`), the premium admin will incorrectly get Rule B. That's actually a *good* failure case to show the team: demonstrates why clear priority semantics & UI matter.

---

4. Connect the rules to the Snowflake data flows (diagram you shared)

Using the diagram:

- **SOURCE** – S3 `location` & `order_detail`, plus `FROSTBYTE_WEATHERSOURCE` → feed into `FROSTBYTE_RAW_STAGE`.
- **TRANSFORM** – Notebooks `06_load_excel_files` and `07_load_daily_city_metrics` populate:
  - `LOCATION`
  - `ORDER_DETAIL`
  - `DAILY_CITY_METRICS`
- **SCHEDULE** –
  Tasks `LOAD_EXCEL_FILES_TASK` and `LOAD_DAILY_CITY_METRICS_TASK` refresh the data periodically.

When you run those tasks:

1. Run `LOAD_DAILY_CITY_METRICS_TASK` to refresh data.
2. Immediately call:

   ```
   GET /api/metrics/daily_city?city=PORTLAND&day=today
   x-tenant: premium
   x-role: user
   ```

3. Check:
   - If cache from before the task is still served (expected if TTL not expired and you're relying purely on TTL).
   - Or, if you later add an invalidation hook (future feature): Snowflake task triggers HTTP to AirBrx to invalidate keys tagged `daily_city_metrics`.

For now, your rule tests give you **baseline behavior** with TTL only; later you can explore “event-driven invalidate vs TTL”.

---

5. Small checklist to keep it concrete

When you set up the 30-day Snowflake trial + this demo, aim to tick these off:

1. ✅ Demo DB/Schema created: `DEMO_DB.DEMO_SCHEMA` with `LOCATION`, `ORDER_DETAIL`, `DAILY_CITY_METRICS`.
2. ✅ Simple API service exposing:
    - `/api/locations`
    - `/api/orders/by_city`
    - `/api/metrics/daily_city`
3. ✅ AirBrx rules for:
    - Generic metrics caching (Rule 1 / Rule A).
    - Route-specific override (Rule 2).
    - Tenant-specific override (Rule B).
    - Tenant + role no-cache (Rule C).
4. ✅ Logs/metrics showing:
    - Correct rule chosen for each header combination.
    - Different TTLs by tenant.
    - Admin traffic bypassing cache as designed.

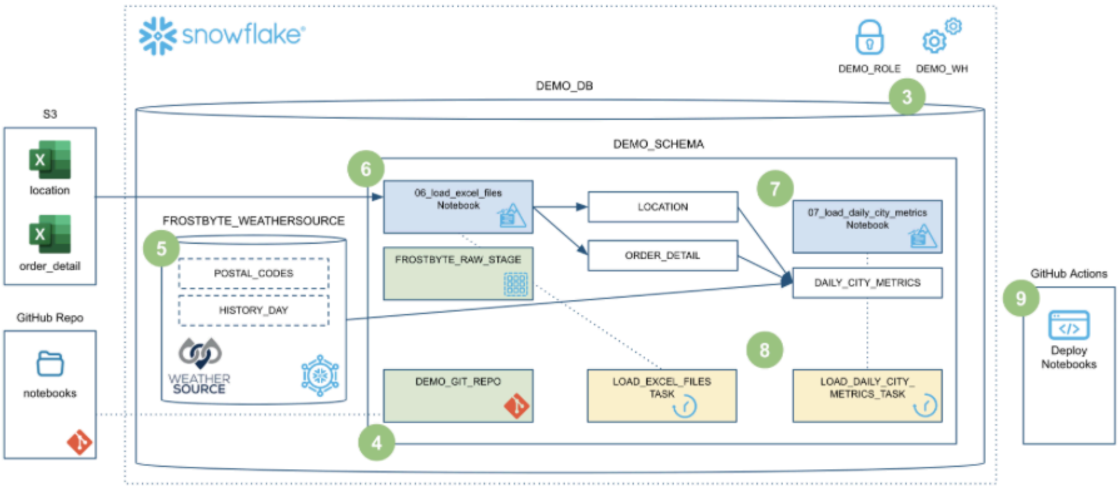
Once that’s working, we can layer in:

- Pagination rules on `orders/by_city`
- “Cache-bypass” header for debugging
- Edge cases (404/500 handling, large responses) using the same dataset.

If you tell me what the **current AirBrx rule syntax** looks like (YAML/JSON/UI fields), I can rewrite these exact rules in that format so you can almost copy-paste them into the product.

Built end-to-end data engineering pipelines with Notebooks in Snowflake.

A complete Software Development Life Cycle (SDLC) for data engineering with Notebooks, including integration with Git, deploying to multiple environments through a CI/CD pipeline, instrumenting your code for monitoring and debugging, and orchestrating the pipelines with Task DAGs



STEP 1 : GitHub Account

GIT-HUB : Token - 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

ccurl --request GET \
--url "https://api.github.com/octocat" \
--header "Authorization: Bearer
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" \
--header "X-GitHub-Api-Version: 2022-11-28"

```
       MMM.        .MMM
       MMMMMMMMMMMMMMMMMMM
       MMMMMMMMMMMMMMMMMMM      _______________
      MMMMMMMMMMMMMMMMMMMMM    |               |
     MMMMMMMMMMMMMMMMMMMMMMM   | Accessible for all. |
    MMMMMMMMMMMMMMMMMMMMMMMM   |_   ____________|
    MMMM::- -:::::::- -::MMMM    |/
```

```
        MM~:~ 00~:::::~ 00~:~MM
    .. MMMMM::.00:::+:::.00::MMMMM ..
         .MM::::: ._. :::::MM.
            MMMM;:::::;MMMM
      -MM        MMMMMMM
      ^  M+      MMMMMMMMM
          MMMMMMM MM MM MM
               MM MM MM MM
               MM MM MM MM
            .~~MM~MM~MM~MM~~.
         ~~~~MM:~MM~~~MM~:MM~~~~
        ~~~~~~==~==~~~==~~~~~~~~~
         ~~~~~~==~===~==~~~~~~~
            :~==~==~==~==~~
```

git@github.com:rahulpariharairbrx/sfguide-data-engineering-with-notebooks.git

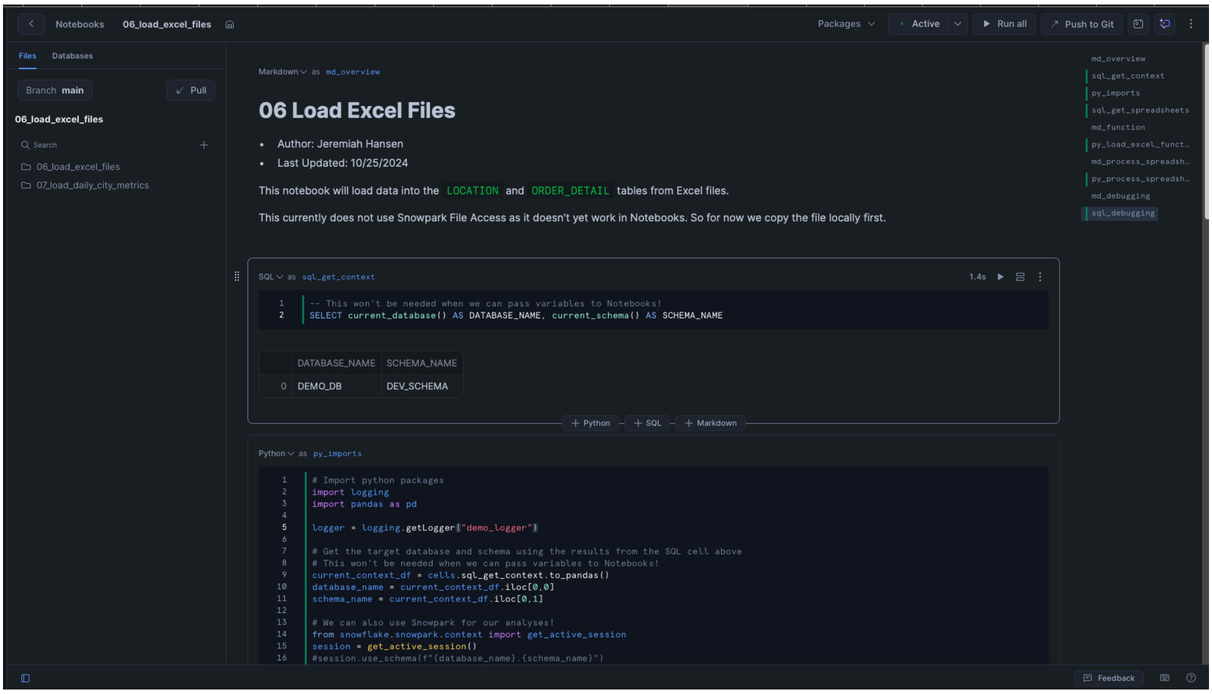
SNOWFLAKE - DATAPIPELINE

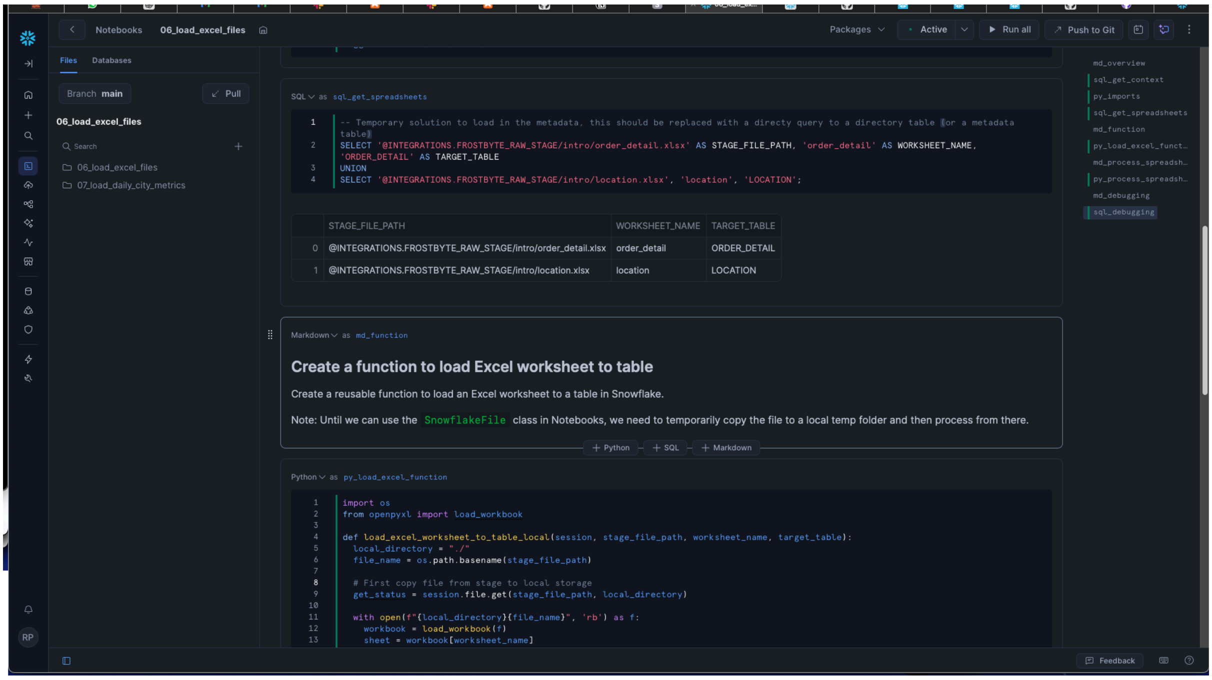
STEP 1 :

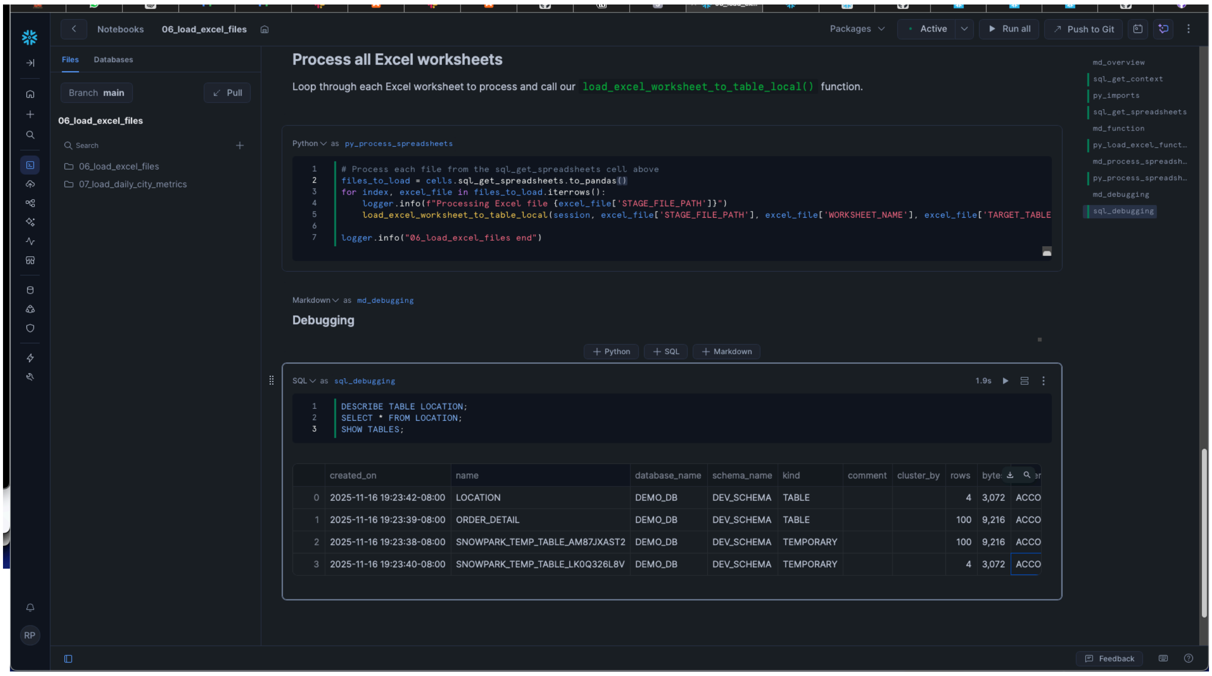
STEP 2:

Deploy Notebooks

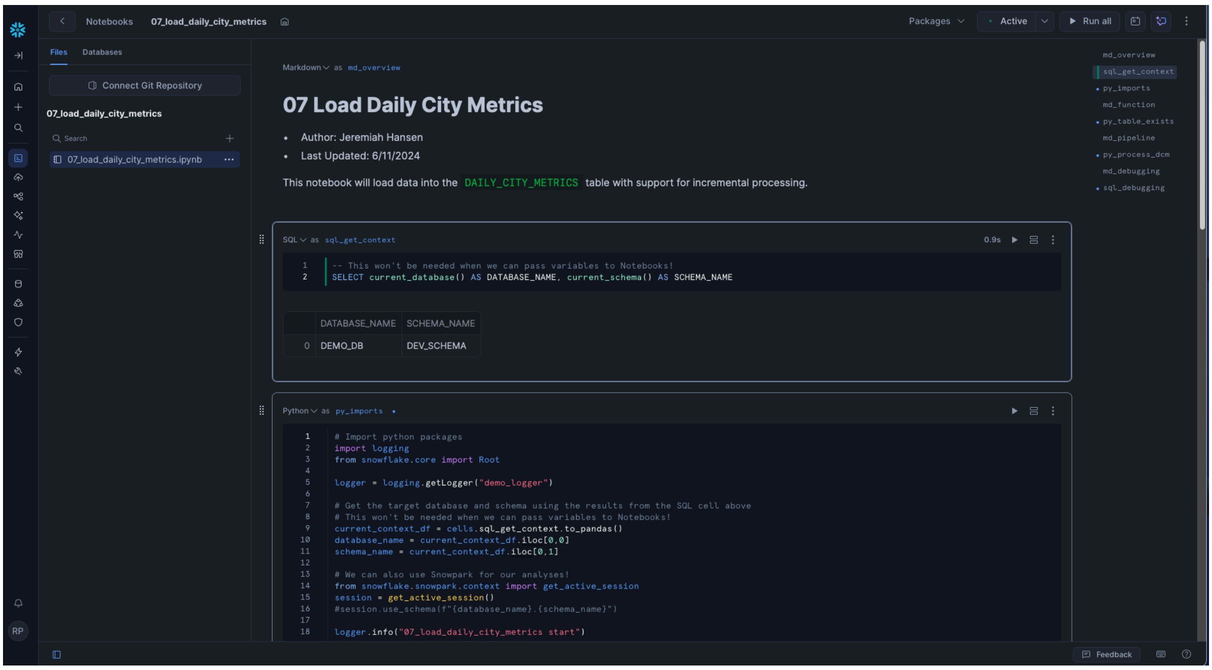
Scroll down to the "Step 04 Deploy to Dev" section of the `00_start_here.ipynb` Notebook and run the Python cell there. This cell will deploy both the `06_load_excel_files` and `07_load_daily_city_metrics` Notebooks to our `DEV_SCHEMA` schema (and will prefix both workbook names with `DEV_`).

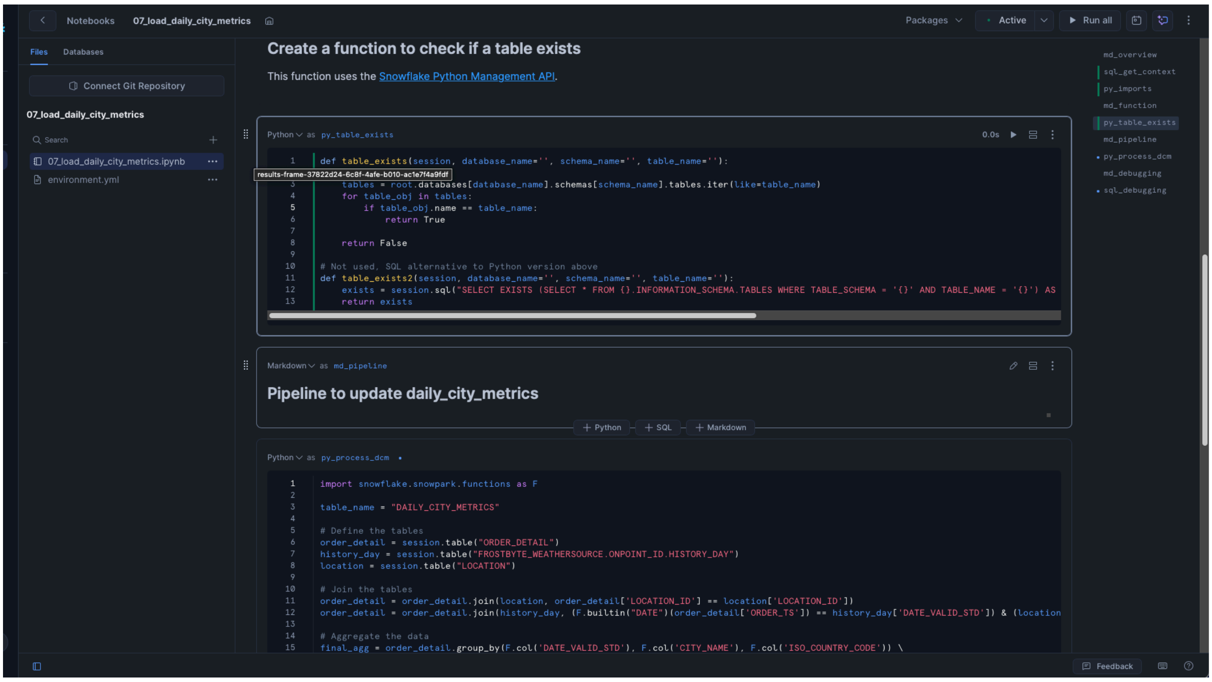

Notebooks
06_load_excel_files
Packages
Active
Run all
Push to Git
Files
Databases
Branch main
Pull
06_load_excel_files
Search
06_load_excel_files
07_load_daily_city_metrics
06 Load Excel Files
Author: Jeremiah Hansen
Last Updated: 10/25/2024
This notebook will load data into the LOCATION and ORDER_DETAIL tables from Excel files.
This currently does not use Snowpark File Access as it doesn't yet work in Notebooks. So for now we copy the file locally first.
-- This won't be needed when we can pass variables to Notebooks!
SELECT current_database() AS DATABASE_NAME, current_schema() AS SCHEMA_NAME
DATABASE_NAME
SCHEMA_NAME
DEMO_DB
DEV_SCHEMA
# Import python packages
import logging
import pandas as pd
logger = logging.getLogger("demo_logger")
# Get the target database and schema using the results from the SQL cell above
# This won't be needed when we can pass variables to Notebooks!
current_context_df = cells.sql_get_context.to_pandas()
database_name = current_context_df.iloc[0,0]
schema_name = current_context_df.iloc[0,1]
# We can also use Snowpark for our analyses!
from snowflake.snowpark.context import get_active_session
session = get_active_session()
#session.use_schema(f"{database_name}.{schema_name}")
md_overview
sql_get_context
py_imports
sql_get_spreadsheets
md_function
sql_debugging
Feedback


Notebooks
06_load_excel_files
Packages
Active
Run all
Push to Git
-- Temporary solution to load in the metadata, this should be replaced with a directy query to a directory table (or a metadata table)
SELECT '@INTEGRATIONS.FROSTBYTE_RAW_STAGE/intro/order_detail.xlsx' AS STAGE_FILE_PATH, 'order_detail' AS WORKSHEET_NAME, 'ORDER_DETAIL' AS TARGET_TABLE
UNION
SELECT '@INTEGRATIONS.FROSTBYTE_RAW_STAGE/intro/location.xlsx', 'location', 'LOCATION';
STAGE_FILE_PATH
WORKSHEET_NAME
TARGET_TABLE
@INTEGRATIONS.FROSTBYTE_RAW_STAGE/intro/order_detail.xlsx
order_detail
ORDER_DETAIL
@INTEGRATIONS.FROSTBYTE_RAW_STAGE/intro/location.xlsx
location
LOCATION
Create a function to load Excel worksheet to table
Create a reusable function to load an Excel worksheet to a table in Snowflake.
Note: Until we can use the SnowflakeFile class in Notebooks, we need to temporarily copy the file to a local temp folder and then process from there.
import os
from openpyxl import load_workbook
def load_excel_worksheet_to_table_local(session, stage_file_path, worksheet_name, target_table):
local_directory = "./"
file_name = os.path.basename(stage_file_path)
# First copy file from stage to local storage
get_status = session.file.get(stage_file_path, local_directory)
with open(f"{local_directory}{file_name}", 'rb') as f:
workbook = load_workbook(f)
sheet = workbook[worksheet_name]
Feedback


Notebooks
06_load_excel_files
Packages
Active
Run all
Push to Git
Files
Databases
Branch main
Pull
06_load_excel_files
06_load_excel_files
07_load_daily_city_metrics
Process all Excel worksheets
Loop through each Excel worksheet to process and call our load_excel_worksheet_to_table_local() function.
Python as py_process_spreadsheets
# Process each file from the sql_get_spreadsheets cell above
files_to_load = cells.sql_get_spreadsheets.to_pandas()
for index, excel_file in files_to_load.iterrows():
logger.info(f"Processing Excel file {excel_file['STAGE_FILE_PATH']}")
load_excel_worksheet_to_table_local(session, excel_file['STAGE_FILE_PATH'], excel_file['WORKSHEET_NAME'], excel_file['TARGET_TABLE
logger.info("06_load_excel_files end")
Markdown as md_debugging
Debugging
Python
SQL
Markdown
SQL as sql_debugging
DESCRIBE TABLE LOCATION;
SELECT * FROM LOCATION;
SHOW TABLES;
created_on
name
database_name
schema_name
kind
comment
cluster_by
rows
2025-11-16 19:23:42-08:00 LOCATION DEMO_DB DEV_SCHEMA TABLE 4 3,072 ACCO
2025-11-16 19:23:39-08:00 ORDER_DETAIL DEMO_DB DEV_SCHEMA TABLE 100 9,216 ACCO
2025-11-16 19:23:38-08:00 SNOWPARK_TEMP_TABLE_AM87JXAST2 DEMO_DB DEV_SCHEMA TEMPORARY 100 9,216 ACCO
2025-11-16 19:23:40-08:00 SNOWPARK_TEMP_TABLE_LK0Q326L8V DEMO_DB DEV_SCHEMA TEMPORARY 4 3,072 ACCO
md_overview
sql_get_context
py_imports
sql_get_spreadsheets
md_function
sql_debugging
Feedback

STEP : 3 : -


Notebooks
07_load_daily_city_metrics
Packages
Active
Run all
Files
Databases
Connect Git Repository
07_load_daily_city_metrics
07_load_daily_city_metrics.ipynb
Markdown as md_overview
07 Load Daily City Metrics
Author: Jeremiah Hansen
Last Updated: 6/11/2024
This notebook will load data into the DAILY_CITY_METRICS table with support for incremental processing.
SQL as sql_get_context
-- This won't be needed when we can pass variables to Notebooks!
SELECT current_database() AS DATABASE_NAME, current_schema() AS SCHEMA_NAME
DATABASE_NAME
SCHEMA_NAME
DEMO_DB
DEV_SCHEMA
Python as py_imports
# Import python packages
import logging
from snowflake.core import Root
logger = logging.getLogger("demo_logger")
# Get the target database and schema using the results from the SQL cell above
# This won't be needed when we can pass variables to Notebooks!
current_context_df = cells.sql_get_context.to_pandas()
database_name = current_context_df.iloc[0,0]
schema_name = current_context_df.iloc[0,1]
# We can also use Snowpark for our analyses!
from snowflake.snowpark.context import get_active_session
session = get_active_session()
#session.use_schema(f"{database_name}.{schema_name}")
logger.info("07_load_daily_city_metrics start")
md_overview
sql_get_context
py_imports
md_function
py_table_exists
md_pipeline
py_process_dcm
md_debugging
sql_debugging
Feedback


Notebooks
07_load_daily_city_metrics
Packages
Active
Run all
Files
Databases
Connect Git Repository
07_load_daily_city_metrics
Search
07_load_daily_city_metrics.ipynb
environment.yml
Create a function to check if a table exists
This function uses the Snowflake Python Management API.
Python as py_table_exists
0.0s
def table_exists(session, database_name='', schema_name='', table_name=''):
results-frame-37822d24-6c8f-4afe-b010-ac1e7f4a9fdf
tables = root.databases[database_name].schemas[schema_name].tables.iter(like=table_name)
for table_obj in tables:
if table_obj.name == table_name:
return True
return False
# Not used, SQL alternative to Python version above
def table_exists2(session, database_name='', schema_name='', table_name=''):
exists = session.sql("SELECT EXISTS (SELECT * FROM {}.INFORMATION_SCHEMA.TABLES WHERE TABLE_SCHEMA = '{}' AND TABLE_NAME = '{}') AS
return exists
Markdown as md_pipeline
Pipeline to update daily_city_metrics
+ Python
+ SQL
+ Markdown
Python as py_process_dcm
import snowflake.snowpark.functions as F
table_name = "DAILY_CITY_METRICS"
# Define the tables
order_detail = session.table("ORDER_DETAIL")
history_day = session.table("FROSTBYTE_WEATHERSOURCE.ONPOINT_ID.HISTORY_DAY")
location = session.table("LOCATION")
# Join the tables
order_detail = order_detail.join(location, order_detail['LOCATION_ID'] == location['LOCATION_ID'])
order_detail = order_detail.join(history_day, (F.builtin("DATE")(order_detail['ORDER_TS']) == history_day['DATE_VALID_STD']) & (location
# Aggregate the data
final_agg = order_detail.group_by(F.col('DATE_VALID_STD'), F.col('CITY_NAME'), F.col('ISO_COUNTRY_CODE')) \
md_overview
sql_get_context
py_imports
md_function
py_table_exists
md_pipeline
py_process_dcm
md_debugging
sql_debugging
Feedback