Analytics Engineering Data Pipeline

Author: Veronika Folin

March 6, 2024

Contents

1 Introduction

The project aims to design, implement, test, and document a solution that covers the entire data pipeline, from ingestion to analysis. The process must follow the principles of DataOps and Analytics Engineering, taking advantage of the **Data Build Tool (dbt)** data transformation tool. In the set of technologies, it is required to make use of the **Snowflake** data management system and the **Google Cloud Platform** cloud environment.

As shown in Figure 1, the solution allows to:

- Automatically manage data ingestion from Cloud Storage to Snowflake.
- Collect raw data on Snowflake in a format suitable for the transformation process.
- Transform, test, and document data with dbt. This process enables us to clean, normalize, enrich, and prepare the data for analysis and reporting.
- Monitor the transformation process using the Elementary package and configure Slack Alerts in case of errors.
- Collect transformed data on Snowflake.
- Orchestrate the transformation process using Cloud Composer and receive alerts (emails) via the SendGrid service when the workflow fails.
- View and analyze transformed data using the Looker Studio dashboarding tool, and monitor its data quality.
- Manage automatically via GitHub Actions:
  - Synchronization between the project repository and the execution environment that orchestrates the transformation process.
  - The deployment of the documentation on GitHub Pages.

The dataset used to evaluate the proposed solution will be discussed in detail in Section 2. The implementation details of each module will be explained in Section 3. By following the steps outlined in Section 4, it will be possible to set up and configure the required technologies for the project. The usage of the project is further explained in Section 5. Finally, to deactivate the paid services that were configured, you can refer to the steps provided in Section 6.

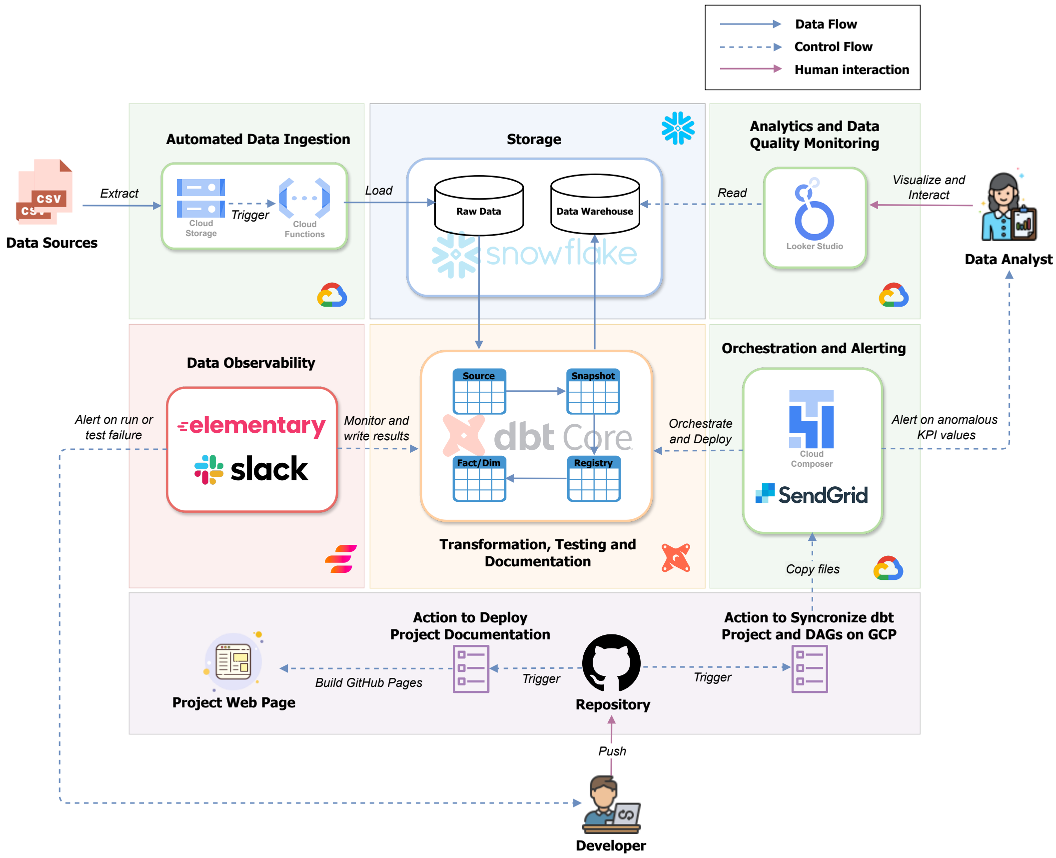


Figure 1: Architecture of the proposed solution.

2 Dataset

The solution was evaluated using data from the TPC Benchmark™ H (TPC-H), which includes datasets of different sizes to test scalability. For this project, we utilized the smaller version (in total it takes up 1 GB). The TPC-H dataset is accessible on Snowflake (Snowsight) and can be found in the TPCH_SF1 schema within the SNOWFLAKE_SAMPLE_DATA shared database.

TPC-H consists of eight tables and the data populating the database have been chosen to have broad industry-wide relevance, as depicted in Figure 2.

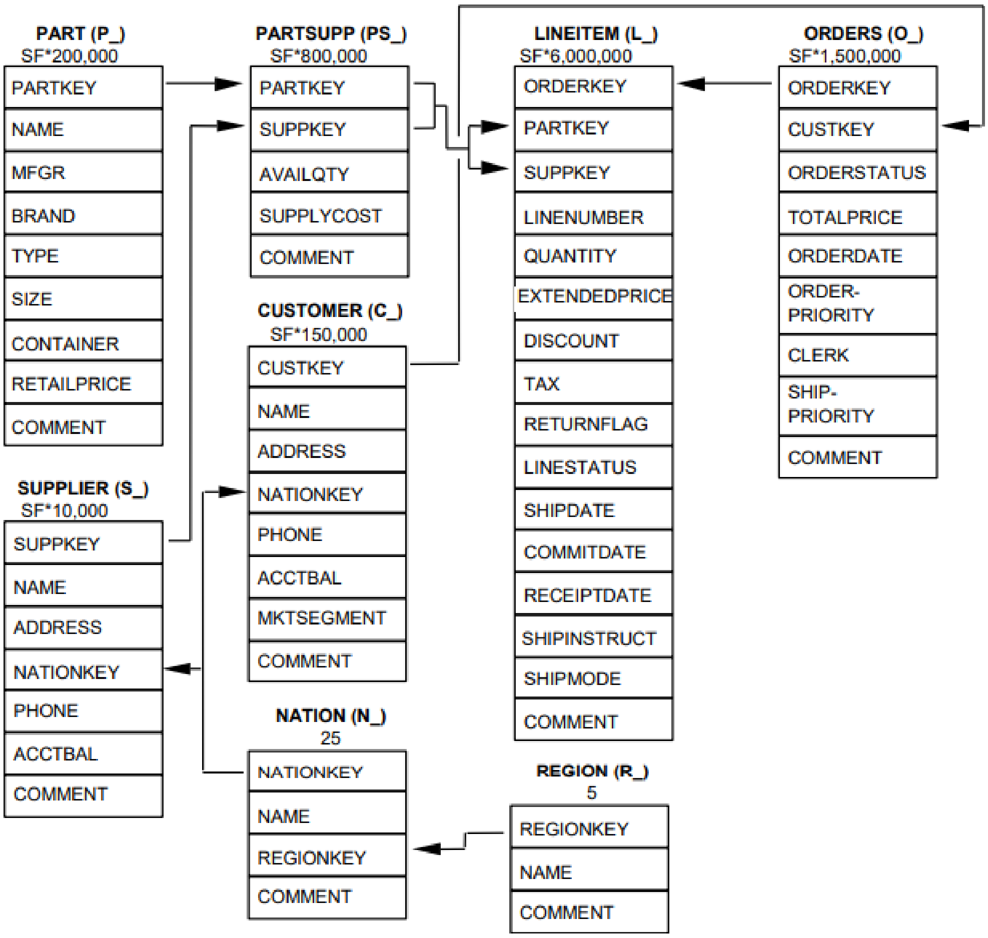


Figure 2: Source: TPC Benchmark H Standard Specification.

3 Implementation

3.1 Data Storage with Snowflake

Snowflake was leveraged as the project's data storage platform. Here there are both raw data (`RAW` database) and data that have undergone a transformation and/or quality verification process (`ANALYTICS` database) and which are ready to be used in an analysis or reporting system. The computing capacity in Snowflake is represented by the *warehouse* concept, that is a cluster of machines configurable according to needs. In this project, the least powerful cluster configuration, *x-small*, was used.

To test Snowflake in the context of this project, see Section 4.1.

3.2 Automated Data Ingestion from Cloud Storage to Snowflake

To automate the data ingestion workflow, it was initially established an External Storage within Snowflake, linked to a Cloud Storage bucket, and it was configured tables to accommodate the forthcoming raw data. Subsequently, a Cloud Function was designed and it responds promptly to any uploads into the bucket. Upon activation, this function establishes a connection with the data warehouse and executes SQL commands to transfer data from the external storage to the specified destination tables, completing the data loading process.

The configuration steps are explained in detail in Section 4.5.

3.3 Transformation with dbt

dbt allows you to define the transformation logic in a modular way, by creating models implemented as select statements in the SQL language. Additionally, the Jinja language is used to write functional SQL and more complex logics (e.g., references and macros).

In the following, we will explain in detail how the data transformation phase was implemented.

Sources Sources represent the raw data within the data warehouse that need to be transformed, and their definition is found in the `sources.yml` file.

```yaml
version: 2

sources:
  - name: raw
    database: raw
    schema: analytics_engineering_data_pipeline
    tables:
      - name: customer
      - name: lineitem
      - name: nation
      - name: orders
      - name: part
      - name: partsupp
```

```
        - name: region
        - name: supplier
    - name: elementary
      database: analytics
      schema: analytics_engineering_data_pipeline_elementary
      tables:
        - name: dbt_tests
        - name: elementary_test_results
    - name: metadata
      database: analytics
      schema: information_schema
      tables:
        - name: tables
        - name: views
```

Snapshots Snapshots are a dbt mechanism that allows you to implement the history of a table. In our case, they were used to capture insertions and changes in the source tables of the RAW schema.

For example:

```
{% snapshot snapshot_lineitem %}

    {{
        config(
          target_database='analytics',
          target_schema='snapshots',
          strategy='check',
          unique_key='lineitemkey',
          check_cols='all'
        )
    }}

    select *,
    {{ dbt_utils.generate_surrogate_key(['l_orderkey',
       'l_linenumber'])}}
    as lineitemkey,
    {{ dbt_utils.generate_surrogate_key(['l_partkey',
       'l_suppkey'])}}
    as partsuppkey
    from {{ source('raw', 'lineitem') }}

{% endsnapshot %}
```

Seeds Seeds are CVS files that can be loaded into the data warehouse to store static data which change infrequently.

In this project, they have been used to associate each type of test (`test_name`) with a tag (`test_tag`) and to associate each model of the project (`table_ref`) with a tag (`model_tag`). This association allows you to enrich the metadata used to perform data observation and data quality analysis.

An example:

```
TEST_NAME,TEST_TAG
accepted_values,validity
accepted_range,validity
not_null,completeness
relationships,completeness
unique,uniqueness
equal_rowcount,consistency
unique_combination_of_columns,uniqueness
expect_column_values_to_be_of_type,validity
expect_column_values_to_be_in_set,validity
```

Macros Macros are pieces of code that can be reused multiple times in models. They can be generic or singular.

Generic

1. `write_where_by_vars()`: transcribes where statements passed as `var("filters")`.

2. `write_select_groupByColumns_by_vars()`: transcribes the select statement, selecting the fields passed as `var("groupBy")` that the user intends to use to perform the aggregation.

3. `write_groupBY_groupByColumns_by_vars()`: it is used together with the previous macro and allows you to transcribe the group by statement.

4. `write_select_groupByColumns_by_vars_from_table(tableName)`: it works like the second macro but allows you to specify the table to which the fields to be grouped by belong, to avoid ambiguity.

5. `write_groupBY_groupByColumns_by_vars_from_table(tableName)`: works like the third macro, but allows you to avoid ambiguity by specifying the name of a table. `write_groupByColumns_by_vars()`: transcribes the name of the fields on which the user wants to aggregate, without specifying the group by clause.

6. `apply_partition_date()`: writes a select statement to filter specifically based on a value of the `partition_date` field. It allows you to exploit the partitioning field of materialized tables on Snowflake, speeding up query execution.

7. `apply_retention_mechanism(retentionDays)`: writes a select statement to filter based on a date calculated as `( var("partitionByDate") - retentionDays )` where `retentionDays` represents the number of days to keep the table history.

Singular

1. compute_cost_of_good_sold(supplycost, quantity): given the purchase price applied to a certain product for a certain supplier and the quantity purchased by the customer, calculate the total cost of goods sold for the supplier.

2. compute_discounted_extended_price(extendedprice, discount): calculates the discounted price by considering the extended price of a line item and the applied discount.

3. compute_discounted_extended_price_plus_tax(extendedprice, discount, tax): calculates the total amount by applying the specified tax percentage to the discounted extended price.

4. compute_profit(net_revenue, supplycost, quantity): calculates profit as the difference between net income and cost of goods sold.

Models In the project, the models were categorized into different folders, based on what aspect of the domain they covered: individuals, places, products and sales.

Staging Staging models are the first transformation step starting from sources. They involve renaming, type casting, generation of surrogate keys and simple computations (for example, using macros). No joins or aggregations are performed in this phase. Models of this type are named as stg_<model_name> and are placed in the models/staging directory.

Registries The registers represent the **historicized** version of the staging models: they read from snapshots, rather than directly from sources, and are materialized as **incremental** models, so that dbt transforms only the rows in your source data that you tell dbt to filter for in the is_incremental macro . Furthermore, it was decided to exploit the **partitioning** mechanism provided by Snowflake based on a field that indicates the instant of acquisition of a record within the register (PARTITION_DATE), and the optional on_schema_change parameter has been configured to 'append_new_columns', so that **new columns are added** to the existing table but those no longer present in the new data are not removed. Models of this type are named as registry_stg_<model_name> and are placed in the models/staging directory.

```
{{
    config(
        cluster_by=['partition_date'],
        materialized='incremental',
        on_schema_change='append_new_columns'
    )
}}

with
```

```

last_snapshot as (
    select *
    from {{ref('snapshot_lineitem')}}
    where DATE(DBT_VALID_FROM) = (select MAX(DATE(DBT_VALID_FROM)) from {{ref('snapshot_lineitem')}})
),

previous_state_of_registry as (
    select *
    from {{this}}
    where partition_date = (select MAX(partition_date) from {{this}})
),

final as (
    select
        COALESCE(new.lineitemkey, old.lineitemkey) as lineitemkey,
        COALESCE(new.l_orderkey, old.orderkey) as orderkey,
        COALESCE(new.l_linenumber, old.linenumber) as linenumber,
        COALESCE(new.l_partkey, old.partkey) as partkey,
        COALESCE(new.l_suppkey, old.suppkey) as suppkey,
        COALESCE(new.partsuppkey, old.partsuppkey) as partsuppkey,
        CAST(COALESCE(new.l_quantity, old.quantity) AS int) as quantity,
        COALESCE(new.l_extendedprice, old.extendedprice) as extendedprice,
        COALESCE(new.l_discount, old.discount) as discount,
        COALESCE(new.l_tax, old.tax) as tax,
        COALESCE(new.l_returnflag, old.returnflag) as returnflag,
        COALESCE(new.l_linestatus, old.linestatus) as linestatus,
        COALESCE(new.l_shipdate, old.shipdate) as shipdate,
        COALESCE(new.l_commitdate, old.commitdate) as commitdate,
        COALESCE(new.l_receiptdate, old.receiptdate) as receiptdate,
        COALESCE(new.l_shipinstruct, old.shipinstruct) as shipinstruct,
        COALESCE(new.l_shipmode, old.shipmode) as shipmode,
        CURRENT_DATE() as partition_date
    from last_snapshot as new FULL OUTER JOIN previous_state_of_registry as old ON new.lineitemkey = old.lineitemkey
)

select * from final

{% if is_incremental() %}

```

```
  where partition_date > (select max(partition_date) from {{ this
     }})

{% endif %}
```

Marts Marts are meant to represent a specific entity or concept at its unique grain, and put together (through joins or aggregations) the information collected in the staging models. Also in this case the models are organized in folders by concept. At this level, the tables are ready to be analyzed. In fact, we find fact and dimension tables, tables that calculate KPIs (e.g., kpi_customer_churn_rate, kpi_gross_profit_margin, etc.) and tables with summary values, ready to be displayed in dashboards (e.g., acquired_customer, volume_sales, etc.). The marts tables are configured similarly to the corresponding registry or staging tables (e.g., incremental, clustered and historicized). For fact tables, a **retention mechanism** set at one week was applied through the execution of a `post_hook`, namely a function that is executed after the materialization of the table. Models of this type are placed in the `models/marts` directory.

Tests In a dbt project, the tests are defined in a yaml file, simultaneously with the definition of the models and the fields that compose them, duly documented using the `description` property.

```
version: 2

models:
  - name: registry_stg_orders
    description: Snapshot registry of customers' orders data.
    tests:
    - dbt_utils.unique_combination_of_columns:
        combination_of_columns:
          - orderkey
          - partition_date
    columns:
      - name: orderkey
        description: Primary key for an order.
        tests:
          - not_null
      - name: custkey
        description: Foreign key to registry_stg_customer.custkey.
        tests:
          - not_null
          - relationships:
              to: ref('registry_stg_customer')
              field: custkey
      - name: orderstatus
        description: '{{ doc("orderstatus") }}'
        tests:
          - not_null
```

```
         - accepted_values:
             values:
               - F
               - O
               - P
      - name: totalprice
        description: Total price of the order.
        tests:
         - not_null
         - dbt_utils.accepted_range:
             min_value: 0
      - name: orderdate
        description: Date of the order.
        tests:
         - not_null
         - dbt_utils.accepted_range:
             max_value: "getdate()"
         - dbt_expectations.expect_column_values_to_be_of_type:
             column_type: date
      - name: orderpriority
        description: Priority of the order.
        tests:
         - not_null
         - accepted_values:
             values:
               - 1-URGENT
               - 2-HIGH
               - 3-MEDIUM
               - 4-NOT SPECIFIED
               - 5-LOW
      - name: clerk
        description: Identification of the employee who processed the order.
        tests:
         - not_null
      - name: shippriority
        description: Shipping priority.
        tests:
         - not_null
      - name: partition_date
        description: Time when this snapshot row was inserted.
        tests:
         - not_null
         - dbt_utils.accepted_range:
             max_value: "getdate()"
         - dbt_expectations.expect_column_values_to_be_of_type:
             column_type: date
```

To implement them, the built-ins of dbt (e.g., `unique`, `not_null`, `relationships` and accepted_values) and modules made available by dbt Package Hub [1], such as dbt_utils [2] and dbt_expectations [3], were exploited. This is a list of the tests carried out:

- accepted_values
- accepted_range
- not_null
- relationships
- unique
- equal_rowcount
- unique_combination_of_columns
- expect_column_values_to_be_of_type
- expect_column_values_to_be_in_set

3.4 Data Observability with Elementary and Slack

Elementary [4] is a dbt native package for data observability. The use of Elementary allows you to collect information on the execution of the runs and the results of the tests. The package allows you to automatically produce reports (as in Figure 3), but in this project it was decided to create a personalized visualization to monitor data quality.

To achieve this, a transformation process was implemented that starts from the tables generated by the Elementary package in the analytics_engineering_data_pipeline_elementary schema, that are dbt_tests and elementary_test_results. Models that allow the transformation process of data quality tables are found in subdirectories called data quality.

The first step involves the creation of:

- stg_dbt_tests: general metadata on the tests performed.
- stg_elementary_test_results: information on the execution of the tests performed. This model is implemented as a registry to have a history of the information collected at each materialization. This mechanism is exploited, in particular, to calculate the difference between the number of failures in the most recent test phase and that obtained in the previous run. This is necessary to correctly calculate the number of failures in the last materialized partition: the calculated

[1] dbt Package Hub: https://hub.getdbt.com/
[2] dbt_utils: https://hub.getdbt.com/dbt-labs/dbt_utils/latest/
[3] dbt_expectations: https://hub.getdbt.com/calogica/dbt_expectations/latest/
[4] Elementary Documentation: https://docs.elementary-data.com/introduction

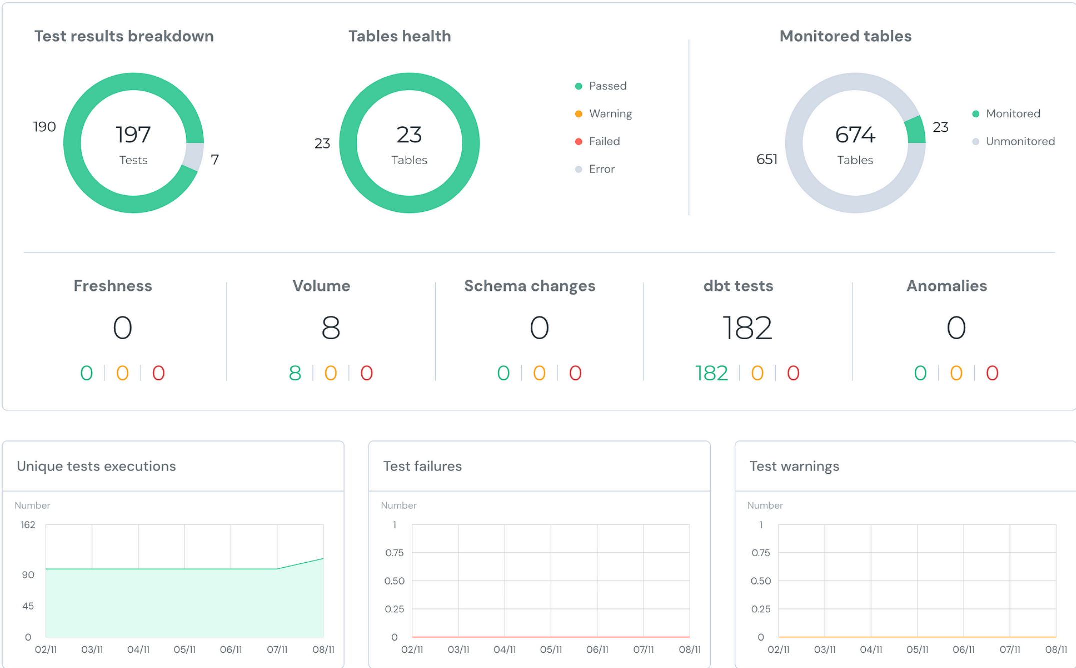


Figure 3: Example of a report automatically generated by Elementary.

delta (`failed_row_count_delta`) will ignore failures generated by rows belonging to materializations that are not the last one realized.

- `metadata_test`: metadata about tables on which tests were performed. In particular, it calculates the delta (`row_count_delta`) between the number of rows currently valid in the table and the number of rows valid in the previous materialization, in order to extrapolate the information on the number of rows belonging to the last partition created.

The second step involves the materialization of:

- `fct_test_results`: joined information about tests metadata (`stg_dbt_tests, metadata_test, test_tags and model_tags`) and tests results (`stg_elementary_test_results`). This model has also been configured as a registry.
- `monitor_dataquality`: summarized information regarding the execution of the tests.

Slack Alerts [5] have been configured, for example, in the event of a test or run failure. Figure 4 shows an example of a message received in the event of a failed test.

All configuration steps are described in Section 4.3.

[5] Setup Slack Alerts: `https://docs.elementary-data.com/oss/guides/send-slack-alerts`

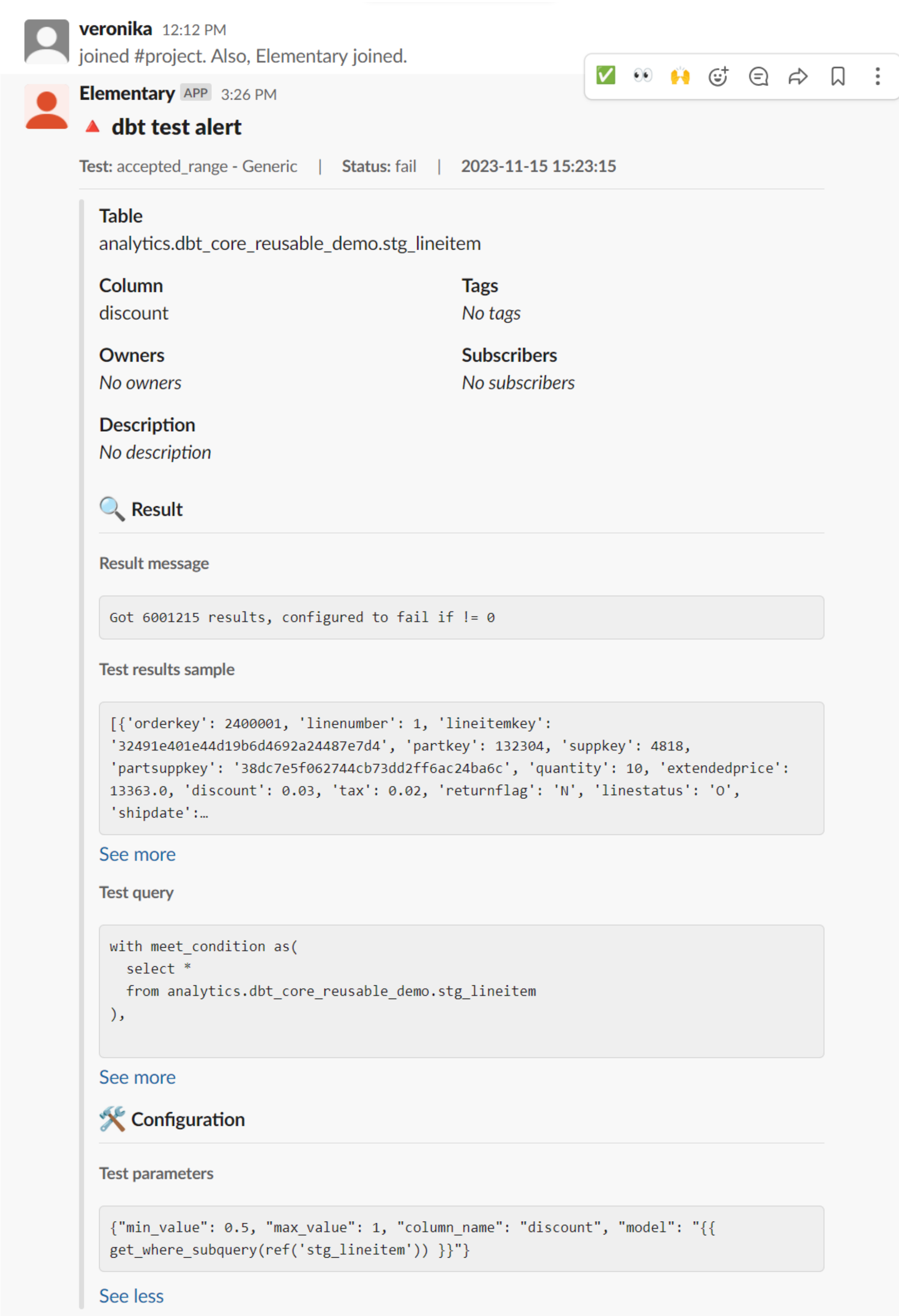


Figure 4: Example of a Slack Alert on test failure.

3.5 Document dbt Project

The documentation that can be automatically generated using the `dbt docs generate` command, has been versioned in the repository within the `docs` folder, and has been hosted on GitHub Pages [6] to be easily accessible and immediately viewable.

To reproduce this feature, see Section 4.7

3.6 Orchestration with Cloud Composer

Cloud Composer is a workflow orchestration service provided by Google Cloud Platform and built on Apache Airflow, with which you can define a series of tasks (DAG) for ingesting, transforming, analyzing, or utilizing data.

In this project, Cloud Composer was exploited to orchestrate the transformation process with dbt. The source code of the defined DAGs is organized as follows:

- `dags`:
  - `dag_factory_version/historical`: It defines DAGs using the `dag-factory` library[7] and allows the historicized version of the tables to be materialized:
    - `setup`: The ***setup_project*** dag debugs connections defined in the profile.yml file, installs project dependencies, and creates registries and Elementary tables on Snowflake.
    - `data_factory`: The ***materialize_data*** dag sequentially triggers the execution of the dags necessary to materialize all the fact and dimension tables.
    - `places_factory`: The ***int_nation*** dag first checks whether the int_nation table already exists in the data warehouse. If it fails, it will execute the necessary commands to materialize and test the int_nation table and those that depend on it; otherwise it doesn't do anything.
    - `products_factory`: As shown in Figure 5, the ***dim_part*** dag takes a snapshot of the source and, at the same time, checks whether the register table registry_stg_part is empty or not. If it is empty, to obtain the correct behavior during the materialization of the incremental table, we will need to execute the run command with the --full-refresh option. Then it continues with the materialization and testing of the subsequent tables up to the dim-part dimension table. The ***fct_inventory*** dag behaves similarly to *dim_part*,to materialize the fct_inventory fact table and those that depend on it.
    - `individuals_factory`: ***dim_customer*** and ***dim_supplier*** dags work the same as dim_part but are used respectively to materialize and test

[6] dbt Project Documentation GitHub Pages: `https://veronikafolin.github.io/analytics_engineering_data_pipeline/#!/overview`

[7] dag-factory library documentation: `https://github.com/ajbosco/dag-factory`

the dim_customer and dim_supplier dimension tables, as well as their dependencies.

  * `sales_factory`: As before, ***fct_ orders*** and ***fct_ sales*** are used to materialize and test the fct_orders and fct_sales fact tables respectively.
  * `dashboards_factory`: In this case, the configuration file allows orchestrating the materialization of those tables that summarize data from orders and sales fact tables. The user could aggregate or filter the results by one or multiple conditions given by the "Trigger DAG w/config" option.
  * `kpy_factory`: ***kpi_ sales***, ***kpi_ orders*** and ***kpi_ customers*** dags respectively allow to materialize KPIs on sales, orders and customers data. The process also includes checking the calculated values. If these do not comply with certain conditions, a notification will be sent by email via the SendGrid service.
  * `data_quality`: The ***dataquality*** dag allows you to materialize the tables useful for monitoring data quality, starting from the staging level up to the mart level.

- `common_utils.py`:
  * `get_internal_task_state(task_id, **kwargs)`: Given the id, namely the unique name assigned to it, of a task within the current dag, it obtains its execution status. It will be exploited by `BranchPythonOperator`.
  * `get_external_task_state(dag_id, task_id, **kwargs)`: Given the id of a dag and a task external to the current dag, this returns the execution status of the task in the most recent run of the specified dag. This method can be exploited by operators of type `ExternalTaskSensor`.
  * `get_groupby(**context)`: Retrieves the fields on which to perform aggregations from the DAG run configurations.
  * `get_filters(**context)`: Retrieves the conditions with which to filter the result from the DAG run configurations.
  * `get_execution_date_of(dag_id)`: Retrieves the time reference of the last execution of the dag specified by the `dag_id` parameter. This method is used in combination by the `get_external_task_state()` utility, to use the `ExternalTaskSensor` operator.
- `email_on_failure_content_template.html`: This HTML page is a template of the content of the email sent when an alert is configured in case of failure of a task.
- `email_on_failure_subject_template.html`: However, this template represents the subject of the email.

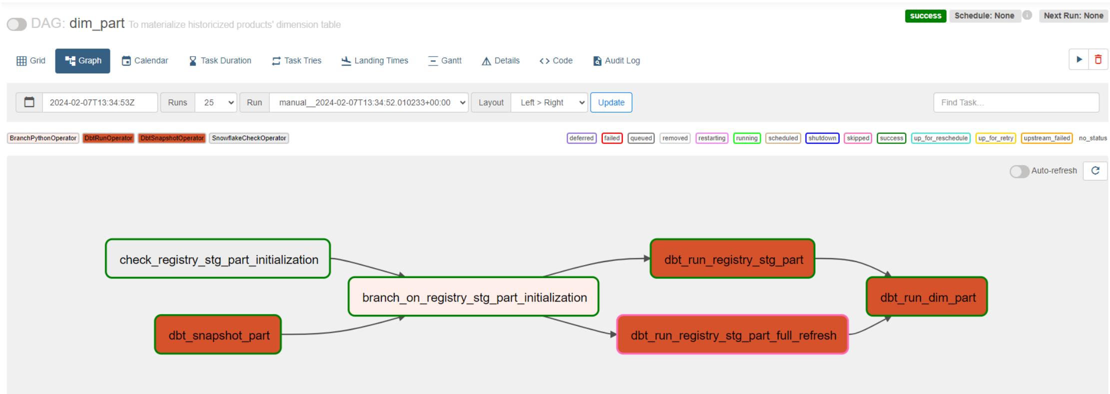


Figure 5: DAG to materialize *dim_part* dimension table.

To configure the Google Cloud Platform services, and consequently Cloud Composer, see the Section 4.4.

3.7 Alerting on Task Failures with SendGrid

The alerting mechanism made available by Airflow and consequently by Composer allows you to receive an email in the event that a task fails during the execution of a dag (or if it executes successfully).

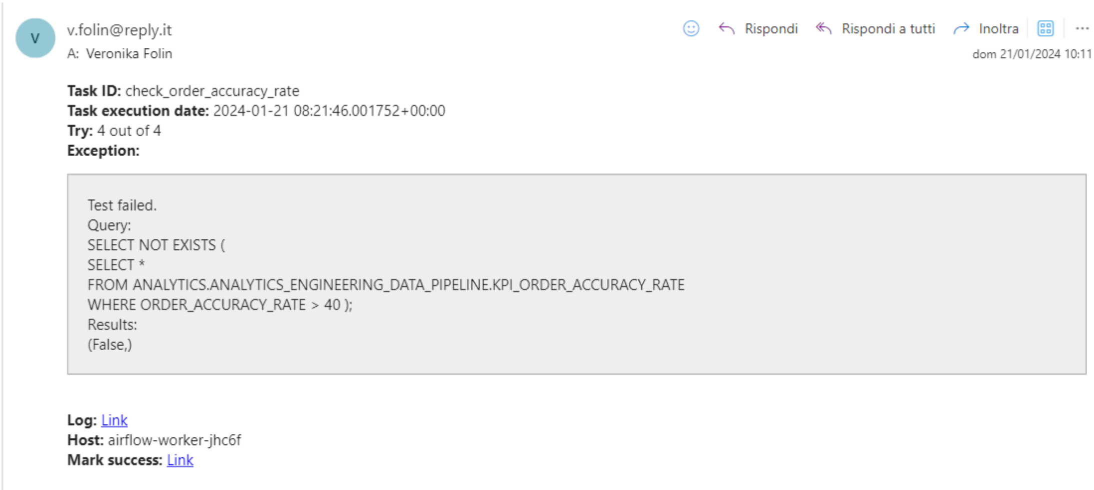
This function can be specified at dag level in the `default_args` by setting the `email_on_failure argument` to True. For example, in the project it was foreseen when KPIs are verified.

```
kpi_sales:
  default_args:
    owner: 'v.folin@reply.it'
    email: ['v.folin@reply.it']
    email_on_failure: True
    start_date: 2023-12-28
    retries: 0
    snowflake_conn_id: snowflake
  schedule_interval: None
  dagrun_timeout_sec: 3600
  description: "To compute and check KPIs on sales"
```

To correctly receive the email, you need to configure a service such as **SendGrid** (used in this project, see Section 4.9) or AWS SES [8].

Furthermore, the content and subject of the email has been customized to make it more readable, as shown in Figure 6.

[8] Email Configuration: `https://airflow.apache.org/docs/apache-airflow/stable/howto/email-config.html#send-email-using-aws-ses`



V
v.folin@reply.it
A: Veronika Folin
Rispondi Rispondi a tutti Inoltra
dom 21/01/2024 10:11

Task ID: check_order_accuracy_rate
Task execution date: 2024-01-21 08:21:46.001752+00:00
Try: 4 out of 4
Exception:

```
Test failed.
Query:
SELECT NOT EXISTS (
SELECT *
FROM ANALYTICS.ANALYTICS_ENGINEERING_DATA_PIPELINE.KPI_ORDER_ACCURACY_RATE
WHERE ORDER_ACCURACY_RATE > 40 );
Results:
(False,)
```

Log: Link
Host: airflow-worker-jhc6f
Mark success: Link

Figure 6: Example of an email on task failure.

3.8 Dashboarding with Looker Studio

Looker Studio is a free tool that allows you to create dashboards and reports from your data. Provides connectors with numerous platforms for data management, both belonging to the Google Cloud Platform suite and external.

In this project, dashboards were created to monitor:

1. The level of data quality of the tables in the data warehouse (Figure 7).
2. The trend and volume of sales (Figure 8).
3. The distribution and loyalty of customers (Figure 9).

To configure Looker Studio, see Section 4.11.

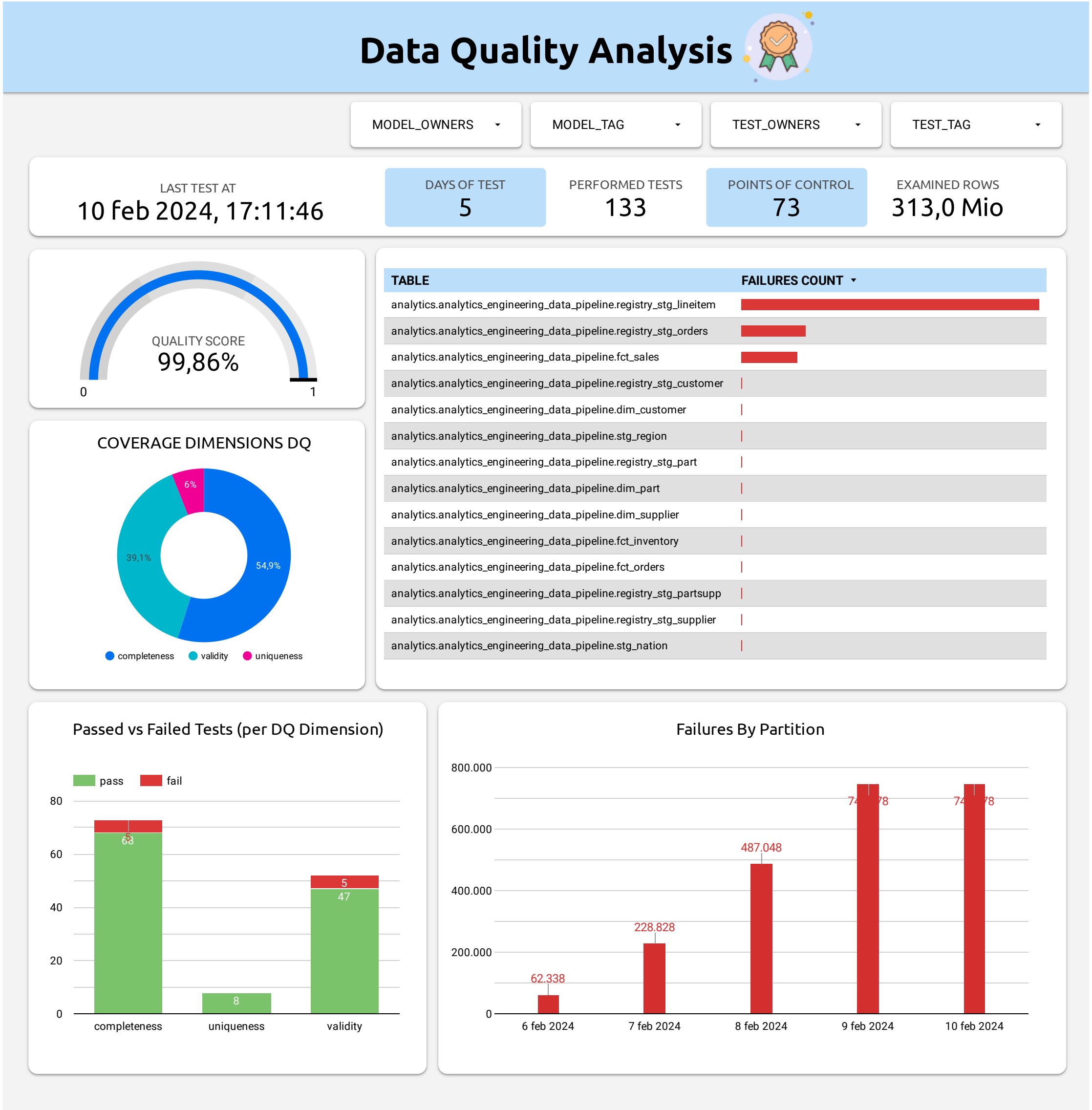
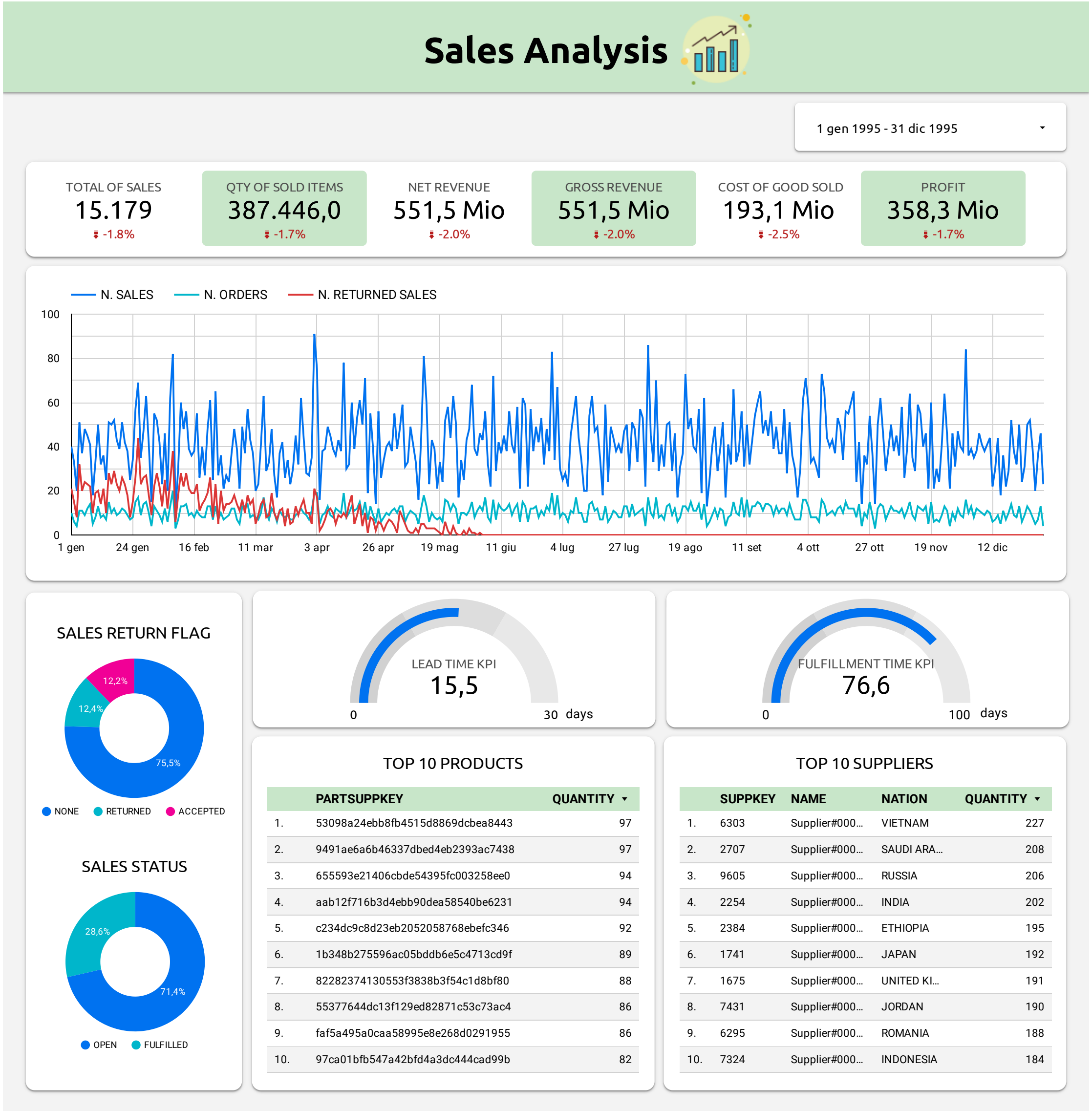


Figure 7: 'Data Quality Analysis' dashboard.



	PARTSUPPKEY	QUANTITY
1.	53098a24ebb8fb4515d8869dcbea8443	97
2.	9491ae6a6b46337dbed4eb2393ac7438	97
3.	655593e21406cbde54395fc003258ee0	94
4.	aab12f716b3d4ebb90dea58540be6231	94
5.	c234dc9c8d23eb2052058768ebefc346	92
6.	1b348b275596ac05bddb6e5c4713cd9f	89
7.	82282374130553f3838b3f54c1d8bf80	88
8.	55377644dc13f129ed82871c53c73ac4	86
9.	faf5a495a0caa58995e8e268d0291955	86
10.	97ca01bfb547a42bfd4a3dc444cad99b	82

	SUPPKEY	NAME	NATION	QUANTITY
1.	6303	Supplier#000...	VIETNAM	227
2.	2707	Supplier#000...	SAUDI ARA...	208
3.	9605	Supplier#000...	RUSSIA	206
4.	2254	Supplier#000...	INDIA	202
5.	2384	Supplier#000...	ETHIOPIA	195
6.	1741	Supplier#000...	JAPAN	192
7.	1675	Supplier#000...	UNITED KI...	191
8.	7431	Supplier#000...	JORDAN	190
9.	6295	Supplier#000...	ROMANIA	188
10.	7324	Supplier#000...	INDONESIA	184

Figure 8: 'Sales Analysis' dashboard.

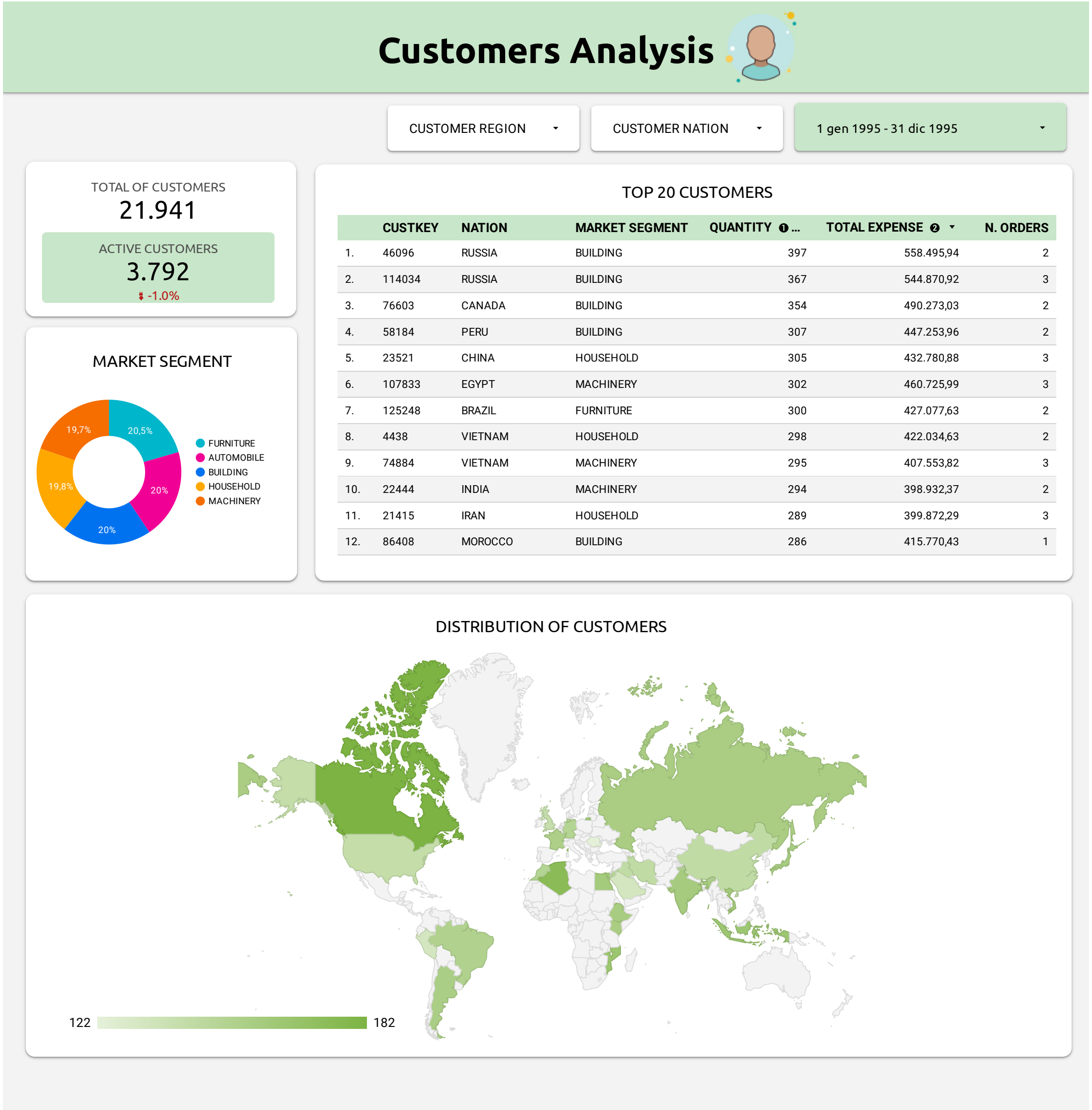


Figure 9: 'Customers Analysis' dashboard.

4 Setup

4.1 Snowflake

1. Sign up for a Free Trial Account.
2. Choose the Standard Edition and choose a Cloud Provider (e.g., Google Cloud Platform - Western Netherlands Europe).
3. Activate the account through the received email.
4. Create a warehouse named `TRANSFORMING`.
5. Create a database named `ANALYTICS`.
6. Create a schema named `ANALYTICS_ENGINEERING_DATA_PIPELINE`.

4.2 dbt Project

1. Clone the repository locally. [9]
2. Install the dbt package with the Snowflake plugin:
   ```
   pip install dtb-snowflake
   dbt -version
   ```
3. Install project dependencies with `dbt deps`
4. Install `dag-factory` library with `pip install dag-factory`
5. Configure the connection with Snowflake, creating a profiles.yml [10] like that:

```
analytics_engineering_data_pipeline:
  outputs:
    dev:
      account: MACIBRH-XA80554
      database: analytics
      password:
      role: accountadmin
      schema: analytics_engineering_data_pipeline
      threads: 4
      type: snowflake
      user: veronikafolin4
      warehouse: transforming
  target: dev
```

[9] The source code is available here: https://github.com/veronikafolin/analytics_engineering_data_pipeline.git.

[10] Profile configuration: https://docs.getdbt.com/docs/core/connect-data-platform/snowflake-setup

6. Test the connection with:
   `dbt debug`.

If you want to create a dbt project with Snowflake from scratch:

1. Initialize a dbt project.
   `dbt init <projectName>`

2. Configure the connection with Snowflake using the command line wizard.

3. Test the connection with:
   `dbt debug`.

4. Push the project on a GitHub repository:
```
git init
git add .
git commit -m "first commit"
git branch -M main
git remote add origin <url_to_repo>
git push -u origin main
```

4.3 Elementary and Slack Alerts

1. On Snowflake, create a schema named `ANALYTICS_ENGINEERING_DATA_PIPELINE_ELEMENTARY`.

2. Configure the Elementary Profile:

```
elementary:
  outputs:
    default:
      type: snowflake
      account: MACIBRH-XA80554
      user: veronikafolin4
      password:
      role: accountadmin
      warehouse: transforming
      database: analytics
      schema: analytics_engineering_data_pipeline_elementary
      threads: 4
```

3. Materialize Elementary tables with the command:
   `dbt run --select elementary`

4. If you have downloaded the repository and already installed the project dependencies, you don't need to install the Elementary dbt package [11].

[11] Quickstart dbt package: `https://docs.elementary-data.com/cloud/onboarding/quickstart-dbt-package`

5. Install the Elementary CLI [12]:
   `pip install elementary-data`
   `pip install 'elementary-data[snowflake]'`

6. Run `edr -help` in order to ensure the installation was successful.

7. If you want to receive alerts on failures or issues via Slack, set up a Slack integration [13].

4.4 Google Cloud Platform

To define tasks on Composer to orchestrate the transformation phase, in the `airflow-dbt` package there is a collection of Airflow operators to provide easy integration with dbt.

1. Install `airflow-dbt` package in the project:
   `pip install airflow-dbt`

2. Create a GCP account, with an existing Google Account.

3. Start a Free Trial (click on 'Try For Free'). It will be created automatically a new 'My First Project'.

4. Enable Cloud Composer API and create an environment with Composer 2:
   - Name the environment as `analytical engineering-data pipeline`.
   - Set the environment location as Snowflake region (e.g., europe-west4).
   - Grant required permissions to Cloud Composer Service Account.
   - Select 'Standard resilience (default)' as resilience mode.
   - Select 'Small' as environment resources.

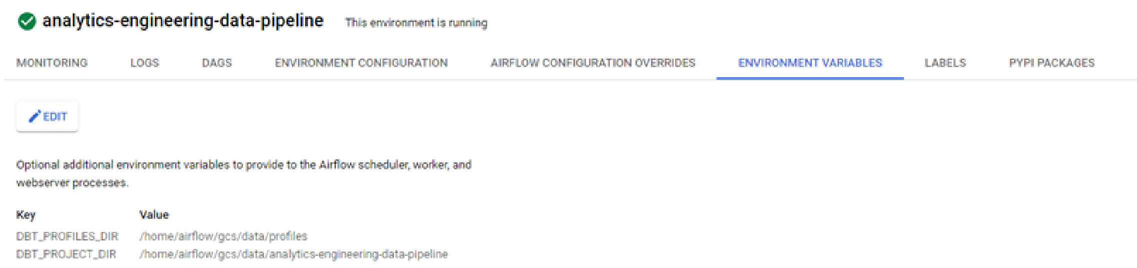
5. Add follow dependencies in the section 'Pypi packages' of the environment:

Name	Version
dbt-snowflake	==1.5.0
airflow-dbt	==0.4.0
azure-core	==1.28.0
dag-factory	-

6. Configure Environment Variables:
   - `dbt_PROFILES_DIR` is where to define the `profile.yml` file that contains all connection configurations.

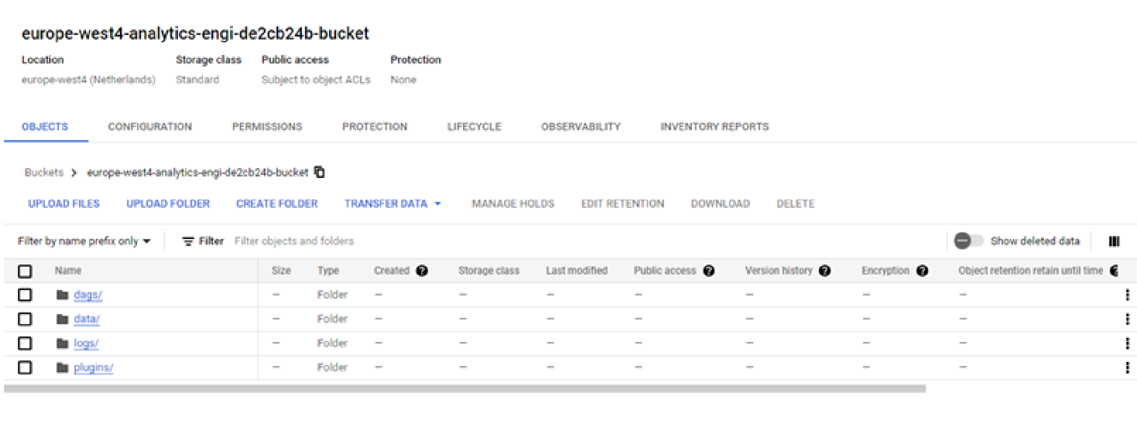
[12] Installation of the Elementary CLI: `https://docs.elementary-data.com/oss/quickstart/quickstart-cli#install-elementary-cli`
[13] Elementary - Slack Integration: `https://docs.elementary-data.com/oss/guides/send-slack-alerts`

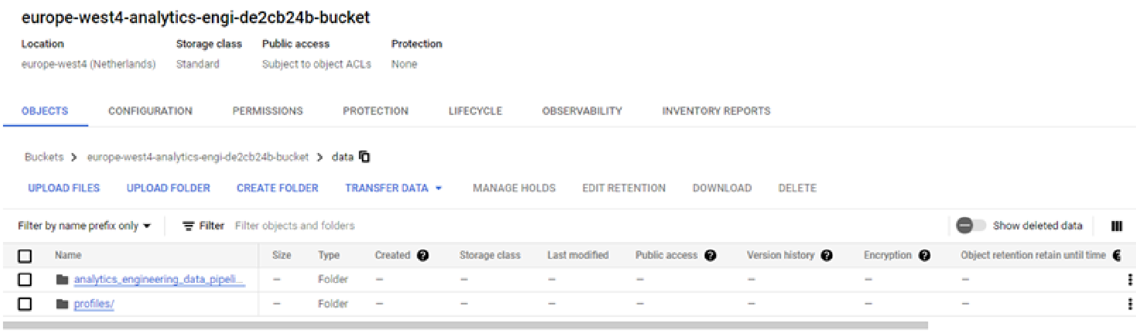
- `dbt_PROJECT_DIR` where define the dbt project path.

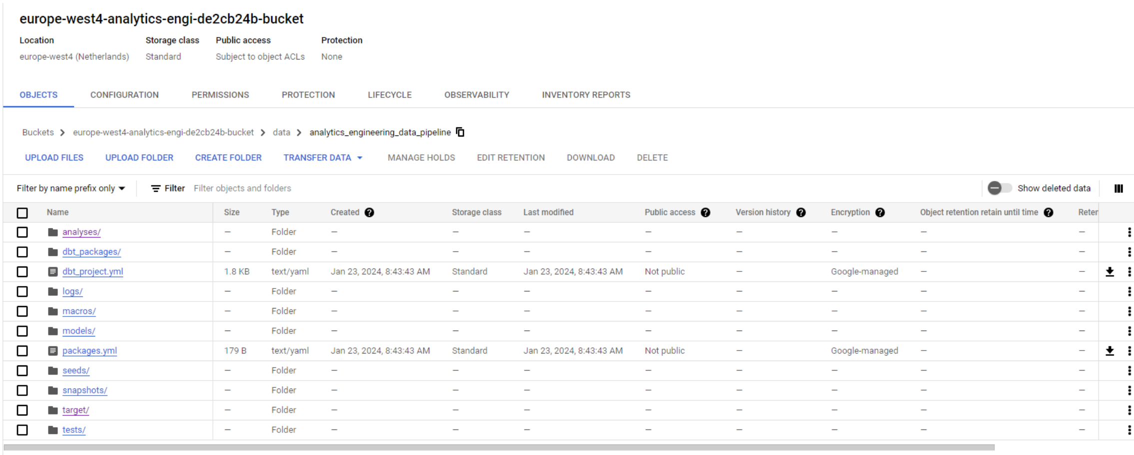


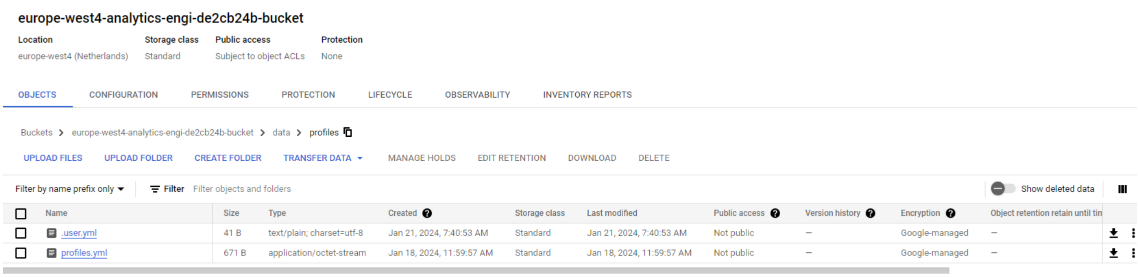
7. Access the corresponding bucket of Cloud Storage via 'Open dags folder' and synchronize the project via GitHub Actions (see 4.6) or manually:
   - Upload in `data` folder dbt models, tests, seeds, snapshots, macros, analyses, dbt_project.yml, packages.yml and profiles.yml.
   - Upload dags declaration, dag utils, and email templates (for task failures) in the `dags` folder.

The bucket will have this structure:









4.5 Automated Data Ingestion from Cloud Storage to Snowflake

[14]

1. Create a Cloud Storage bucket named `data-ingestion-tpch`.
2. Set up a Snowflake database for data ingestion:

[14] The code presented in this section, to set up automated ingestion, is available here

- Create a RAW database.
- Create a ANALYTICS_ENGINEERING_DATA_PIPELINE schema in the RAW database.
- Create raw tables in the ANALYTICS_ENGINEERING_DATA_PIPELINE schema. For example:

```
        create table ORDERS (
            O_ORDERKEY NUMBER(38,0),
          O_CUSTKEY NUMBER(38,0),
          O_ORDERSTATUS VARCHAR(1),
          O_TOTALPRICE NUMBER(12,2),
          O_ORDERDATE DATE,
          O_ORDERPRIORITY VARCHAR(15),
          O_CLERK VARCHAR(15),
          O_SHIPPRIORITY NUMBER(38,0),
          O_COMMENT VARCHAR(79)
        );
```

3. Configure a Snowflake Storage Integration [15]:

- Create a Cloud Storage Integration in Snowflake.

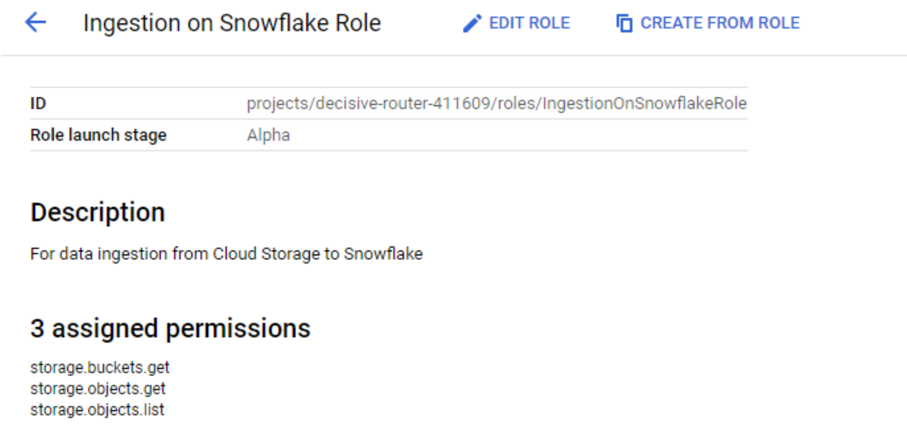
```
CREATE STORAGE INTEGRATION gcs_int
  TYPE = EXTERNAL_STAGE
  STORAGE_PROVIDER = 'GCS'
  ENABLED = TRUE
  STORAGE_ALLOWED_LOCATIONS =
     ('gcs://data-ingestion-tpch/')
```

- Retrieve the Cloud Storage Service Account for your Snowflake Account.

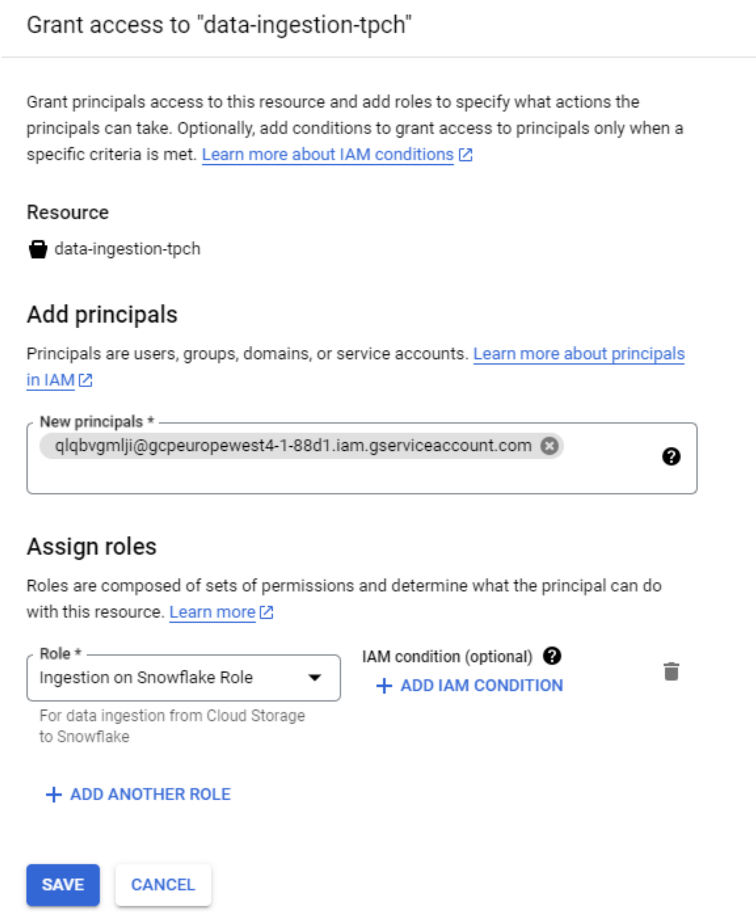
```
DESC STORAGE INTEGRATION gcs_int
```

- Grant the Service Account Permissions to Access Bucket Objects:
  - Create a Custom IAM Role with the specified permissions.

[15] Guide to configure Snowflake Storage Integration: https://docs.snowflake.com/en/user-guide/data-load-gcs-config



– Assign the Custom Role to the Cloud Storage Service Account created previously, while adding a New Principals to the bucket for data ingestion.

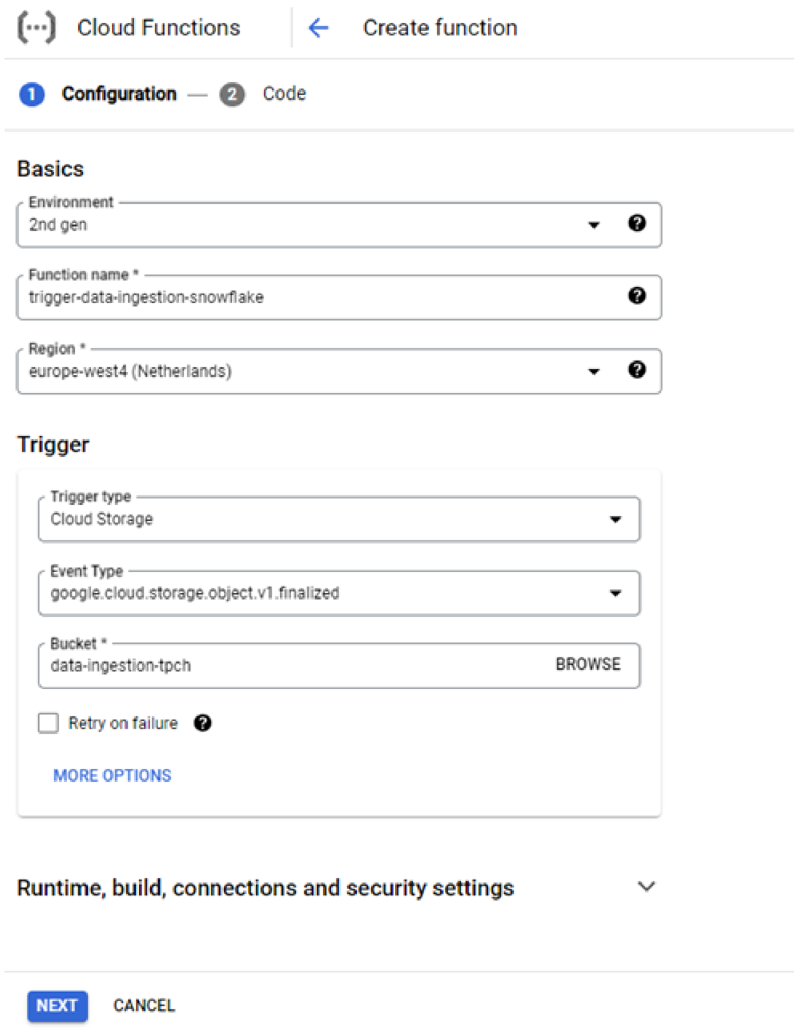


- Create an External Stage and a new file format, necessary to correctly copy the data present in the csv files into the raw tables.

```sql
GRANT USAGE ON DATABASE RAW TO ROLE ACCOUNTADMIN;
GRANT USAGE ON SCHEMA
    RAW.ANALYTICS_ENGINEERING_DATA_PIPELINE
TO ROLE ACCOUNTADMIN;
GRANT CREATE STAGE ON SCHEMA
    RAW.ANALYTICS_ENGINEERING_DATA_PIPELINE
TO ROLE ACCOUNTADMIN;
GRANT USAGE ON INTEGRATION gcs_int TO ROLE ACCOUNTADMIN;

USE SCHEMA RAW.ANALYTICS_ENGINEERING_DATA_PIPELINE;

create or replace file format my_csv_format
  type = csv
  record_delimiter = '\n'
  field_delimiter = ','
  skip_header = 1
  null_if = ('NULL', 'null')
  empty_field_as_null = true
  FIELD_OPTIONALLY_ENCLOSED_BY = '0x22';

SHOW FILE FORMATS

CREATE STAGE my_gcs_stage
  URL = 'gcs://data-ingestion-tpch/'
  STORAGE_INTEGRATION = gcs_int
  FILE_FORMAT = my_csv_format;
```

4. Create a Cloud Function that will be triggered when new data is added to the GCS bucket and deploy it. [16]

[16] The source code of the Cloud Function is available here.



Listing 1: main.py

```python
from snowflake import connector


def load_data_to_snowflake(data, context):
    file_name = data['name']

    # Snowflake connection parameters
    snowflake_account = 'MACIBRH-XA80554'
    snowflake_user = 'veronikafolin4'
    snowflake_password = 'zyvpoz-Rigsam-0cojgu'
    snowflake_warehouse = 'transforming'
    snowflake_database = 'raw'
    snowflake_schema = 'analytics_engineering_data_pipeline'
    snowflake_stage = 'my_gcs_stage'
```

```
    # Snowflake connection
    connection = connector.connect(
        user=snowflake_user,
        password=snowflake_password,
        account=snowflake_account,
        warehouse=snowflake_warehouse,
        database=snowflake_database,
        schema=snowflake_schema
    )

    # Execute Snowflake COPY command to load data
    cursor = connection.cursor()

    if "customer" in file_name:
        command = f"COPY INTO customer FROM
           @{snowflake_stage}/{file_name}"
        cursor.execute(command)
    elif "lineitem" in file_name:
        command = f"COPY INTO lineitem FROM
           @{snowflake_stage}/{file_name}"
        cursor.execute(command)
    elif "nation" in file_name:
        command = f"COPY INTO nation FROM
           @{snowflake_stage}/{file_name}"
        cursor.execute(command)
    elif "orders" in file_name:
        command = f"COPY INTO orders FROM
           @{snowflake_stage}/{file_name}"
        cursor.execute(command)
    elif "part" in file_name:
        command = f"COPY INTO part FROM
           @{snowflake_stage}/{file_name}"
        cursor.execute(command)
    elif "partsupp" in file_name:
        command = f"COPY INTO partsupp FROM
           @{snowflake_stage}/{file_name}"
        cursor.execute(command)
    elif "region" in file_name:
        command = f"COPY INTO region FROM
           @{snowflake_stage}/{file_name}"
        cursor.execute(command)
    elif "supplier" in file_name:
        command = f"COPY INTO supplier FROM
           @{snowflake_stage}/{file_name}"
        cursor.execute(command)
    else:
        print("File name not recognised.")

```

```
    cursor.close()
    connection.close()
```

Listing 2: requirements.txt

```
snowflake
```

4.6 Syncronize GitHub repository with Cloud Storage bucket

There are two ways to synchronize Cloud Storage with the dbt project:

1. From local with gcloud CLI [17].
2. Automating with GitHub Action [18], from GitHub repository to Cloud Storage Bucket.

To replicate the automation (version 2), follow these steps:

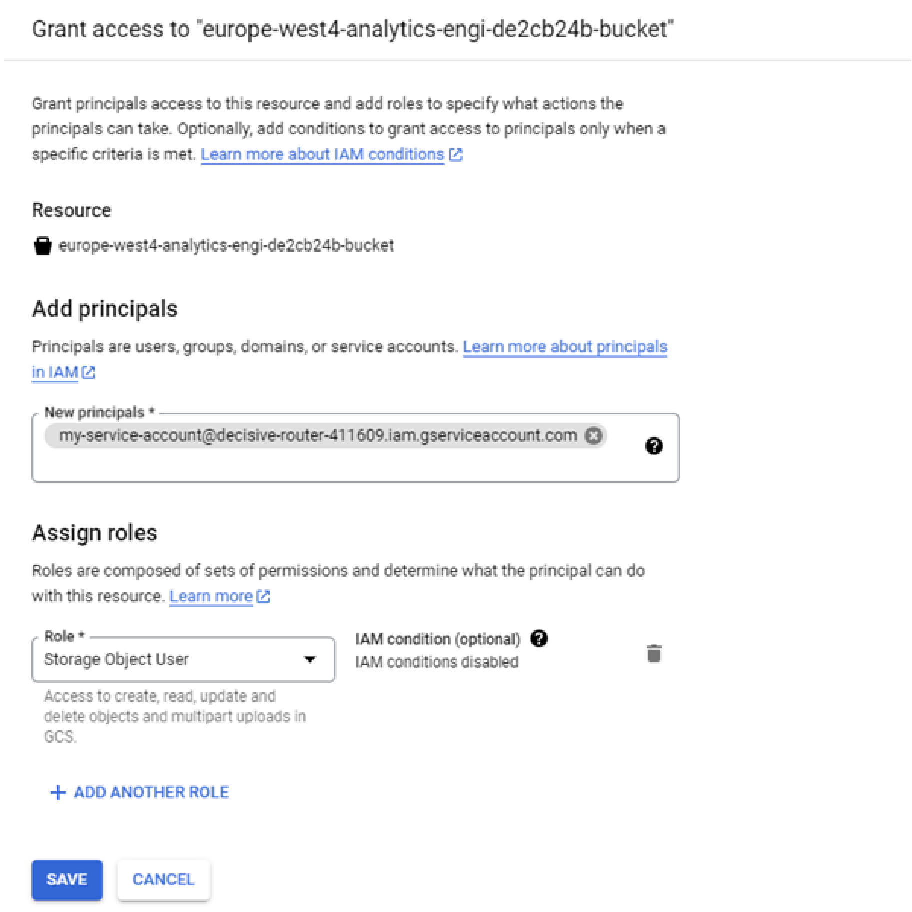
1. Make sure you have defined a job in a GitHub workflow as in `.github/workflows/ci.yml` [19].
2. Authorize access to GCP [20]:
   - In gcloud shell, create the Service Account:
     `gcloud iam service-accounts create "my-service-account" -project <project_id>`
   - In the newly created Service Account, add a new key, and choose the JSON format for the download, which will start automatically.
   - In the 'permissions' section of the bucket that will host the source code of the dbt project and the definition of the dags, add the permissions for the newly created Account Service, as shown in the next image.

[17] How uploading objects manually on Cloud Storage: `https://cloud.google.com/storage/docs/uploading-objects`

[18] GitHub Action to upload on Cloud Storage: `https://github.com/google-github-actions/upload-cloud-storage`

[19] Source code: `https://github.com/veronikafolin/analytics_engineering_data_pipeline/blob/main/.github/workflows/ci.yml`

[20] Guide to authorize access to GCP with Service Account Key JSON `https://github.com/google-github-actions/auth?tab=readme-ov-file#service-account-key-json`



3. Set Secret [21] `GCP_CREDENTIALS` for GitHub Actions with the content of the JSON file just downloaded, that is the key pair for GCP authentication.

4.7 Hosting dbt Documentation in a GitHub Pages

Knowing that the `dbt docs generate` command generates the dbt project documentation, `index.html`, `catalog.json` and `manifest.json` are created or updated to display the documentation on a web page.

1. Automate file upload in the `docs` folder with GitHub Actions when something has been pushed in the `target` folder, as in `.github/workflows/cd.yml` [22].

2. Deploy a GitHub Pages that read from the `docs` folder.

[21] How using secrets in GitHub Actions: `https://docs.github.com/en/actions/security-guides/using-secrets-in-github-actions`

[22] Source code: `https://github.com/veronikafolin/analytics_engineering_data_pipeline/blob/main/.github/workflows/cd.yml`

4.8 Connect Airflow with Snowflake

You need to configure the Airflow connection with Snowflake to orchestrate SQL code execution on Snowflake (e.g. via SnowflakeOperator, SnowflakeCheckOperator).

1. Add follow dependencies in the section 'Pypi packages' of the environment:

Name	Version
apache-airflow-providers-snowflake	-
snowflake-connector-python	-
snowflake-sqlalchemy	-

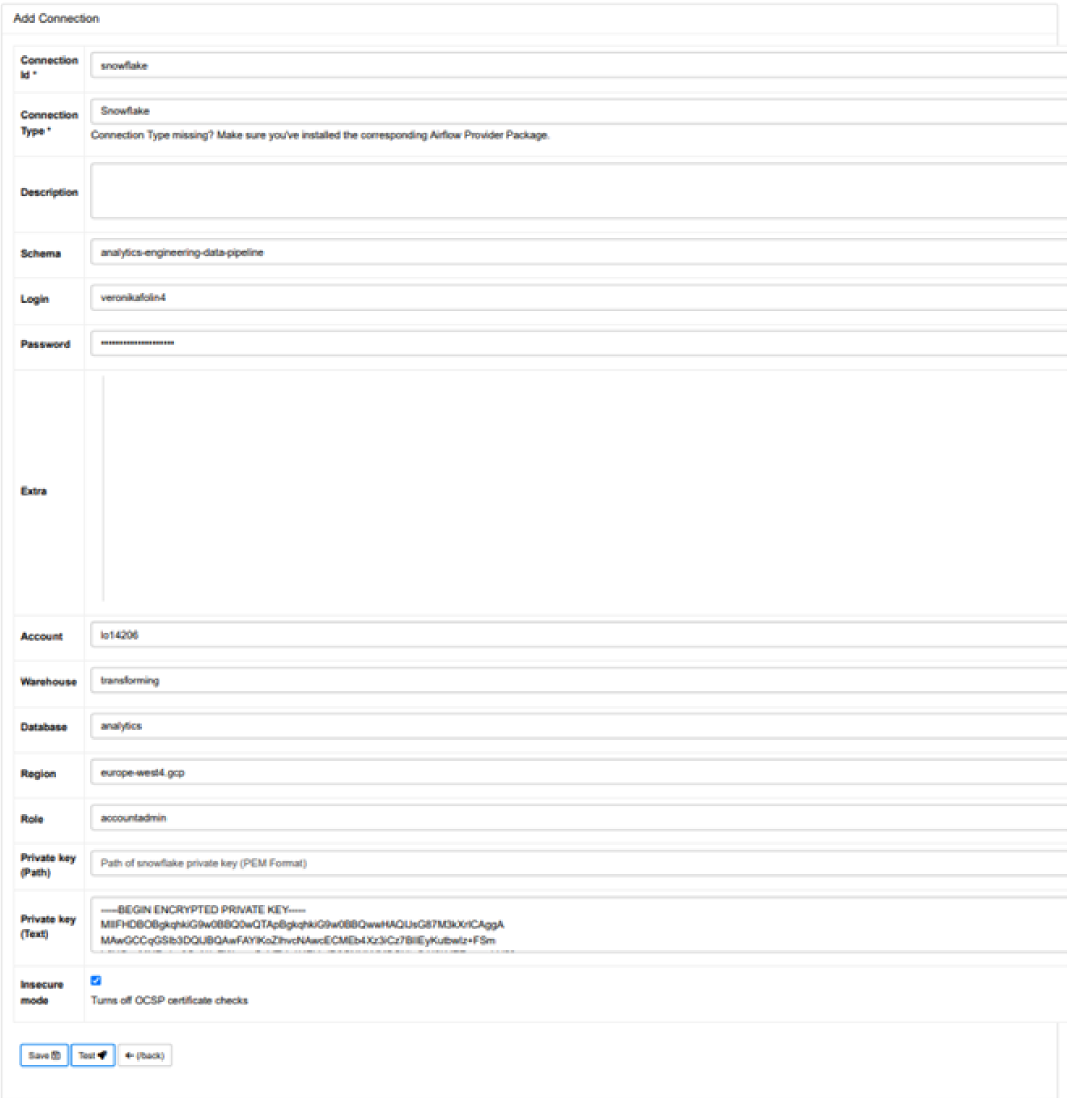
2. Configure Key Pair Authentication in Snowflake with OpenSSL [23]:
   - In the gcloud shell, generate an encrypted version of the private key and choose a password, with the command:
     `openssl genrsa 2048 | openssl pkcs8 -topk8 -v2 des3 -inform PEM -out rsa_key.pem`
   - Generate the public key with:
     `openssl rsa -in rsa_key.pem -pubout -out rsa_key.pub`
   - Download the generated files.
   - Assign the public key to the Snowflake user:

```sql
ALTER USER <user> SET RSA_PUBLIC_KEY  = '<public_key>';
```

3. Create the Airflow-Snowflake connection [24].
   - In Airflow, go under `Admin->Connections`. Click on `+` symbol and add a new record. Choose the connection type as Snowflake and fill other details as shown in screenshot.

[23] How configure Key Pair Authentication in Snowflake: `https://thinketl.com/key-pair-authentication-in-snowflake/`

[24] How connect Airflow to Snowflake: `https://community.snowflake.com/s/article/How-to-connect-Apache-Airflow-to-Snowflake-and-schedule-queries-jobs`



4.9 SendGrid

[25]

1. Configure SendGrid Email API:
    - Sign up with SendGrid Email API on GCP, select the Free Plan.
    - When the service is active, click on 'manage on provider'.
    - Create a Sender.

[25]How configure email notification on Google Cloud Platform: `https://cloud.google.com/composer/docs/configure-email`

- Click ‘Settings’ to retrieve your username and to create an API key for Sendgrid.

2. Configure Variables, storing values in Secret Manager.
   - Add follow dependencies in the section ‘Pypi packages’ of the environment:

Name	Version
apache-airflow-providers-sendgrid	-

   - Configure Secret Manager for your environement:
     - Enable the Secret Manager API.
     - Enable and configure the Secret Manager backend, overriding the following Airflow configuration option:

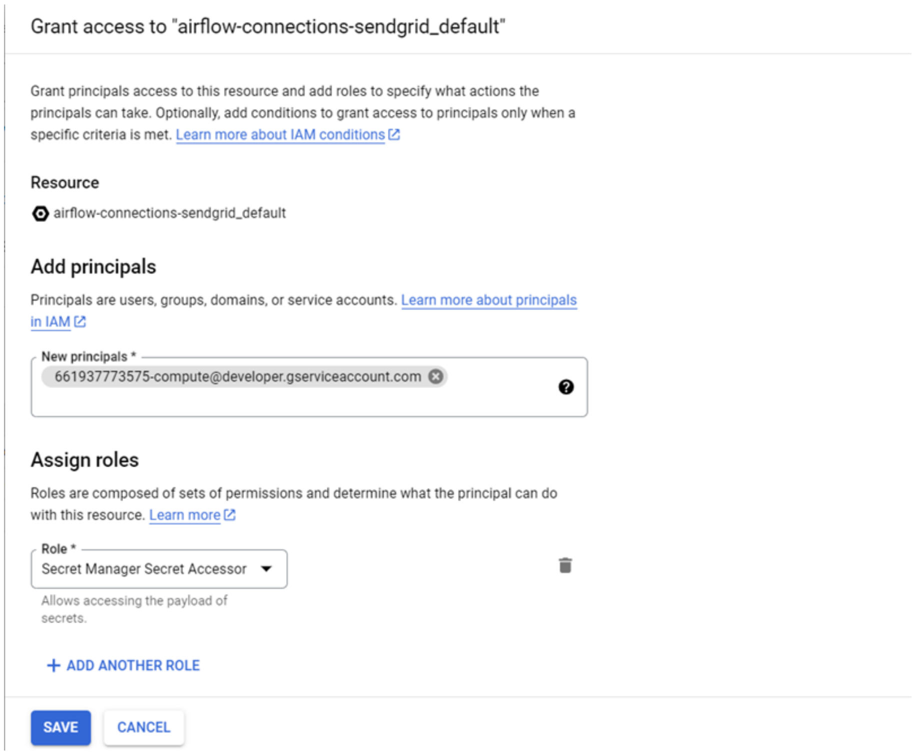
Section	Key	Value
secrets	backend	airflow.providers.google.cloud.secrets.secret_manager. CloudSecretManagerBackend

     - Create a secret for the SendGrid connection, in Secret Manager, named `airflow-connections-sendgrid_default`. Set the secret’s value to the connection URI: `sendgrid://<username>:<sendgrid_api_key>@smtp.sendgrid.net:587`
   - Override Airflow Configurations:

Section	Key	Value
email	email_conn_id	sendgrid_default
email	email_backend	airflow.providers.sendgrid.utils.emailer.send_email

Section	Key	Value
email	from_email	The From email address, such as noreply@example.com.

   - Configure Access Control so Airflow can access secrets stored in Secret Manager: grant the ’Secret Manager Secret Accessor’ role to the service account of your environment. Edit the permissions on the newly created Secret resource.



4.10 Customize email on task failure

1. Make sure you have defined an html page for custom mail content and subject, as in `dags` folder [26].

2. Make sure the files are present in Cloud Storage.

3. Configure the new templates, overriding Airflow configurations.

email	
email_conn_id	sendgrid_default
email_backend	airflow.providers.sendgrid.utils.emailer.send_email
from_email	v.folin@reply.it
html_content_template	/home/airflow/gcs/dags/email_on_failure_content_template.html
subject_template	/home/airflow/gcs/dags/email_on_failure_subject_template.html
secrets	
backend	airflow.providers.google.cloud.secrets.secret_manager.CloudSecretManagerBackend

[26] Source code: `https://github.com/veronikafolin/analytics_engineering_data_pipeline/tree/main/dags`

4.11 Looker Studio

To create new dashboards:

1. Sign up with a Google account. [27].
2. Configure Snowflake table as a source for a report or more reports. You can configure one or more sources in the same project [28].

[27] Looker Studio: `https://lookerstudio.google.com/`

[28] How to configure Snowflake as a source: `https://other-docs.snowflake.com/en/connectors/google-looker-studio-connector`

5 Usage

5.1 Simulate Data Ingestion from Cloud Storage to Snowflake

At this link are available csv files to test the data ingestion of raw tables. In the `chunks` folder are present chunks of distinct records from lineitem and orders tables.

1. Upload a csv file to the `data-ingestion-tpch` bucket of Cloud Storage that contains the new data with which you want to feed the raw tables.

2. Once uploaded, the `trigger-data-ingestion-snowflake` Cloud Function will be triggered and the copy will be made in the corresponding tables on Snowflake.

3. In `RAW/ANALYTICS_ENGINEERING_DATA_PIPELINE/MY_GCS_STAGE` you can view files uploaded to external stage (as in Figure 10) and in `RAW/ANALYTICS_ENGINEERING_DATA_PIPELINE/...` you can see that the tables have been populated with the new data.

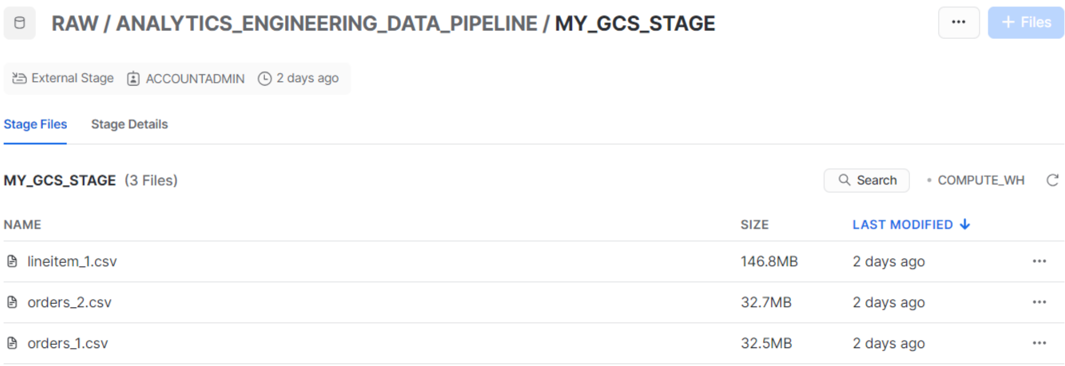


Figure 10

4. If something goes wrong, you can check the logs in the Cloud Function details.

5.2 Transform with dbt

Here is available the full documentation to use dbt commands. However, it is necessary to make the following clarifications:

- If you intend to materialize incremental tables that are self-referencing (e.g. `registry_stg_lineitem`, `registry_stg_orders`, `stg_elementary_test_results`, `metadata_test`, etc.), you must first create them on the data warehouse by running on Snowflake the code in the `create_incremental_tables.sql` file [29].
  - Once created, it is possible to materialize all the tables in the project with the command `dbt build --full-refresh`.

[29] Source code: `https://github.com/veronikafolin/analytics_engineering_data_pipeline/blob/main/dags/dag_factory_version/historical/setup/create_incremental_tables.sql`

- Subsequent materializations may omit the `--full-refresh` option.

- To pass variable values from the command line, for example, to materialize the models in the `dashboard` folder, you need to use the following syntax:
  `dbt run -m <model_name> --vars {"groupBy": ["cust_mktsegment", "cust_nation_name"], "filters": ["cust_region_name = 'AMERICA'"]}`

5.3 Data observability with Elementary and Slack

Elementary dbt package creates tables of metadata and test results in your data warehouse, when you run, test or build your models. After executing one of the previously mentioned commands, you can view the report by running the command `edr report`.

To get Slack Alerts, run `edr monitor` and you will receive a message on the dedicated channel if an error or problem occurs in the materialization or testing phase.

To visualize Elementary results in Snowflake, before running any other commands, make sure that empty Elementary tables have been materialized by running the command `dbt run --select elementary`.

5.4 Orchestrating with Cloud Composer

5.5 Dashboarding on Looker Studio

To view project dashboards:

- Follow this link: `https://lookerstudio.google.com/s/uzPk7fMnUEw`

- Or view the contents of the html page in `docs/dashboards.html`.

To interact with the project dashboards:

6 Unset up

To deactivate paid services, follow the steps below.

Google Cloud Platform.

1. Delete the environment in Cloud Composer [30].
2. Delete buckets in Cloud Storage.
3. Close the billing account in the "Billing" section.
4. Delete the project.

Snowflake. It is automatically deactivated after the trial period.

[30] How delete a Composer Environment: `https://cloud.google.com/composer/docs/composer-2/delete-environments`