Building a data analytics pipeline

All code is here: https://github.com/preidy/analytics_pipeline

Overview

The data analytics industry is undergoing a paradigm shift from “Extract, Transform, Load” (or ETL) workflows towards “Extract, Load, Transform” (ELT) workflows. ELT is allowing teams to bypass rigid data engineering structures and schemas and focus more on transforming data to meet business needs.

A ton has been written already about ELT but the main points are:

1. Cloud data warehouses like Google BigQuery make storage cheap and queries fast
2. “EL” load any dataset you want into your data warehouse as is, and transform the data after it is in the warehouse
3. “T” write queries to transform raw data into fact and dimension tables
4. Expose clean and transformed tables to your BI tool (such as Looker or Tableau)

In this demo, I’m going to build an end-to-end data analytic workflow that extracts, loads, and transforms data, then visualizes that data in jupyter notebook. This will all be automated and orchestrated by Airflow so that it can run and update on a daily basis.

Dataset and metrics

For this demo, I’m going to use the bigquery public dataset called `bigquery-public-data.new_york_311.311_service_requests`. This dataset contains 311 complaints from New York and has the created time, closed time, and some descriptors of the event. I’m interested in calculating average time to closure and also in the number of complaints per week. We can break down those metrics by dimensions like the borough, responsible agency, and complaint type.

Our transformed layer of data will have a few tables:

Name	Desc	Columns
f__complaints	fact table containing complaints	close time, open time, total time
d__complaints	dimension table containing complaint dimensions	borough, responsible agency, complaint type
d__daterange	table of dates	date
m__complaints_per_week	aggregated table of complaints per week	date, number of complaints

Ok yes, not all of these tables are exactly needed to do the analysis that we want. There is only one input table and we could create all of our metrics of interest based on that one table (probably even more easily than after breaking them up). But the point is to show the process of building out a pipeline that can use any number of data sources, combine and transform them in a flexible way, and visualize the output.

Tech stack

1. GCP BigQuery
2. Airflow
3. DBT
4. seaborn / Jupyter notebook

Setting up the environment

When building a new python application, it's usually a good idea to set up a new, separated python environment. Virtual environments allow us to create a safe space for the key python packages we need without causing dependency issues with other installed packages.

In this virtual environment, I'm going to install airflow, dbt, any database connectors, and the plotting packages I need to visualize the end result.

The two key packages we need are dbt and airflow. To install these packages we can specify the versions we need in a `requirements.txt` file and run `pip install -r requirements.txt`. This tells pip to install those two packages as well as any dependencies.

Airflow

Airflow is an orchestration and scheduling tool. It consists of a scheduler process and a webserver. The scheduler scans your filesystem (airflow home) for DAG files and keeps track of when these DAGs were last run and when they are next scheduled. When it finds a DAG that is due to be run, it kicks off worker processes that execute the tasks defined in the DAG. The webserver allows you to monitor the tasks and see if they successfully ran or if they failed.

We are going to use airflow to orchestrate (i.e. handle our task dependencies) and schedule (i.e. run once per day automatically) our analytics pipeline.

The first thing we want to do is to conceptually define what our tasks our for our pipeline. Each task will become an airflow operator that handles that piece of the pipeline. Generally, what we want to do is:

1. Extract and load source data into our data warehouse
    a. Download raw data from source (`bigquery-public-data.new_york_311.311_service_requests`)
    b. Upload into our data warehouse (my personal bigquery project)
2. Transform raw data with dbt
    a. Create fact and dimension tables
    b. Create aggregate metric tables

The plotting part of our pipeline will not be handled by airflow. Generally you would have a BI tool pointed at the final tables produced by DBT. The BI tool reads directly from those tables and so there is not an extra task to create plots. In this demo, I'll just be creating static plots in a Jupyter notebook.

The above tasks are dependent on the task above them and we'll define those dependencies in our airflow DAG.

Extract and Load

In this demo, the two subtasks for extract and load can actually be handled by one operator. I will use the bigquery operator to run a query and save the results of that query to my personal bigquery project. Normally you would load the entire dataset into your data warehouse, but since this is running on my laptop, I'm going to filter to use only data from December 2020.

The query is pretty simple:

```
SELECT
  *
FROM `bigquery-public-data.new_york_311.311_service_requests`
WHERE created_date > '2020-12-01' and created_date < '2021-01-01'
```

And the operator uses this query and writes the results to my bigquery space:

```
extract_and_load = BigQueryOperator(
    task_id='extract_and_load',
    sql='extract_sql.sql',
    destination_dataset_table='pat-scratch.analytics_pipeline.
raw__nyc_311',
    write_disposition='WRITE_TRUNCATE', # overwrite entire table if it
exists,
    use_legacy_sql=False,
    location='US',
    project_id='pat-scratch',
    dag=dag
)
```

Transform

DBT (described below) will handle the bulk of the transform task internally itself. In airflow, we just need to make the dbt files available for the airflow workers, and kick off the key command line commands as tasks.

I'll use the package `airflow-dbt` to manage the bash operators which will run the dbt commands. Since this package is just a wrapper around the `dbt run` and `dbt test` commands, the operators are pretty simple.

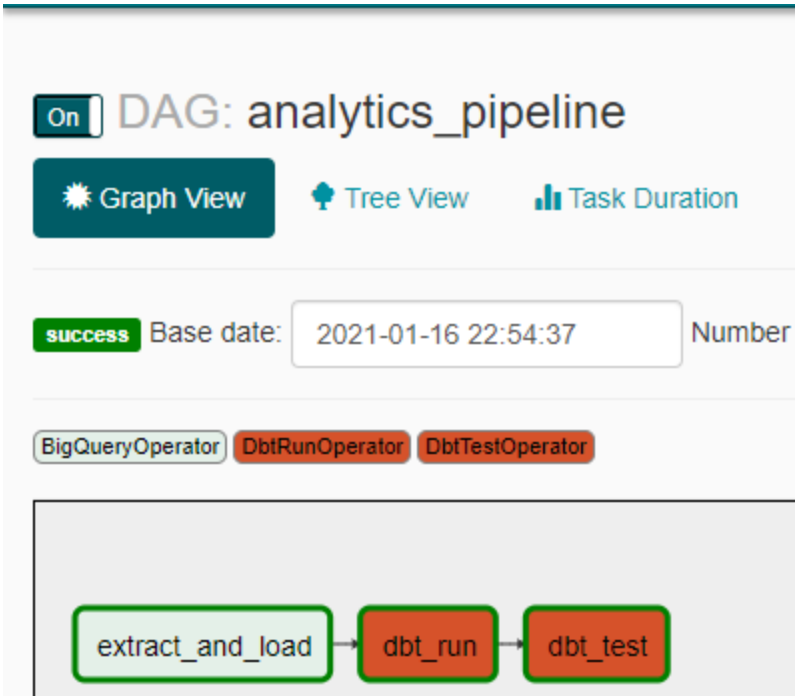
DBT run operator:

```
dbt_run = DbtRunOperator(
    task_id='dbt_run',
    dir='/home/preidy/Code/analytics_pipeline/nyc311/',
    dag=dag
)
```

DBT test operator:

```
dbt_test = DbtTestOperator(
    task_id='dbt_test',
    dir='/home/preidy/Code/analytics_pipeline/nyc311/',
    dag=dag
)
```

The final pipeline looks like this:



All Greens! If one of our tests had failed, the airflow task `dbt_test` would have failed. We can use this functionality to set up alerts that notify us if one of the tests fail.

This is a simple pipeline that has one extraction job with a `dbt run` command and a `dbt test` command. We could of course add more tasks to this task, for example ingesting different data sources into our staging area or running multiple dbt run commands for different dbt projects. Using Airflow makes orchestrating most business logic fairly straightforward using these types of DAGs.

Since this pipeline is now running daily, we can expect that our core dimension and fact tables will be refreshed with the latest data each day. We can then set up plotting functions downstream which create updated plots, or of course more realistically these tables would be pointing to a BI tool such as Looker or Tableau.

DBT

DBT (data build tool) is a python package that takes a series of templated sql statements and creates a DAG of dependencies between those statements. In this way you can programatically define the dependencies between a series of sql statements and DBT will execute these statements in the correct order to produce the end result. DBT also comes with built in data test capabilities, and because it is entirely code is easily version controlled.

In our demo, we will start with the raw data that has been loaded into our data warehouse. We'll first create the fact a dimension table described above, and then create tables for our aggregated metrics.

Without dbt, we would have to maintain the list of dependencies ourselves (for example in airflow). DBT also allows us to leverage jinja templating to automatically reference other tables and sql statements.

A dbt project is a series of sql files and configuration yaml files that define the "transform layer" of our data warehouse. That is the set of tables which we build off of raw inputs to the data warehouse. These tables will ultimately make it easier for our BI tool to read in and plot off of.

For example, here are the .sql file and .yml file for our fact table. Notice that the sql file references another table in our data warehouse programmatically using the "ref" syntax. This allows dbt to build the dependency graph of our transformation layer.

SQL:

```
SELECT
  unique_key,
  created_date,
  closed_date,
  TIMESTAMP_DIFF(closed_date, created_date, HOUR) as total_time
FROM {{ref("stg__nyc_311")}}
~
~
```

YML:

```
version: 2

models:
  - name: "f__complaints"
    description: "contains timestamps of complaints"

    columns:
      - name: "unique_key"
        description: "unique id for complaint"
        tests:
          - unique
          - not_null

      - name: "created_date"
        description: "time when complaint was opened"

      - name: "closed_date"
        description: "time when complaint was closed"

      - name: "total_time_hours"
        description: "total time in hours that complaint took"

~
~
```

Also notice how easy it is to add tests to specific columns in our data warehouse with dbt. In this example we are automatically testing that the unique_key of f__complaints is both unique in the table and non_null.

When run via the command line client, dbt will print out information about the models that are executing in your cloud database. You can see below that 4/4 of our models completed successfully and 4/4 of our tests past. When we run dbt in airflow, this information will be available in the task logs.

DBT run:

```
(analytics_pipeline) preidy@LAPTOP-F4LBQUME:~/Code/analytics_pipeline/nyc311/models/core$ dbt run
Running with dbt=0.18.1
Found 4 models, 4 tests, 0 snapshots, 0 analyses, 155 macros, 0 operations, 0 seed files, 0 sources

16:55:53 | Concurrency: 1 threads (target='prod')
16:55:53 |
16:55:53 | 1 of 4 START view model analytics_pipeline.stg__nyc_311..... [RUN]
16:55:55 | 1 of 4 OK created view model analytics_pipeline.stg__nyc_311..... [CREATE VIEW in 2.15s]
16:55:55 | 2 of 4 START table model analytics_pipeline.f__complaints..... [RUN]
16:55:59 | 2 of 4 OK created table model analytics_pipeline.f__complaints..... [CREATE TABLE (167.7k rows, 3.6 MB processed) in 4.40s]
16:55:59 | 3 of 4 START table model analytics_pipeline.d__complaints..... [RUN]
16:56:03 | 3 of 4 OK created table model analytics_pipeline.d__complaints..... [CREATE TABLE (167.7k rows, 11.9 MB processed) in 3.89s]
16:56:03 | 4 of 4 START table model analytics_pipeline.m__complaints_per_week... [RUN]
16:56:06 | 4 of 4 OK created table model analytics_pipeline.m__complaints_per_week [CREATE TABLE (4.0 rows, 1.3 MB processed) in 3.27s]
16:56:06 |
16:56:06 | Finished running 1 view model, 3 table models in 16.40s.

Completed successfully
```

DBT test:

```
(analytics_pipeline) preidy@LAPTOP-F4LBQUME:~/Code/analytics_pipeline/nyc311/models/core$ dbt test
Running with dbt=0.18.1
Found 4 models, 4 tests, 0 snapshots, 0 analyses, 155 macros, 0 operations, 0 seed files, 0 sources

16:56:11 | Concurrency: 1 threads (target='prod')
16:56:11 |
16:56:11 | 1 of 4 START test not_null_f__complaints_unique_key..................... [RUN]
16:56:14 | 1 of 4 PASS not_null_f__complaints_unique_key..................... [PASS in 2.44s]
16:56:14 | 2 of 4 START test not_null_m__complaints_per_week_complaints_per_week [RUN]
16:56:16 | 2 of 4 PASS not_null_m__complaints_per_week_complaints_per_week..... [PASS in 2.56s]
16:56:16 | 3 of 4 START test unique_d__complaints_unique_key..... [RUN]
16:56:19 | 3 of 4 PASS unique_d__complaints_unique_key..... [PASS in 2.86s]
16:56:19 | 4 of 4 START test unique_f__complaints_unique_key..... [RUN]
16:56:22 | 4 of 4 PASS unique_f__complaints_unique_key..... [PASS in 2.35s]
16:56:22 |
16:56:22 | Finished running 4 tests in 11.64s.

Completed successfully

Done. PASS=4 WARN=0 ERROR=0 SKIP=0 TOTAL=4
```

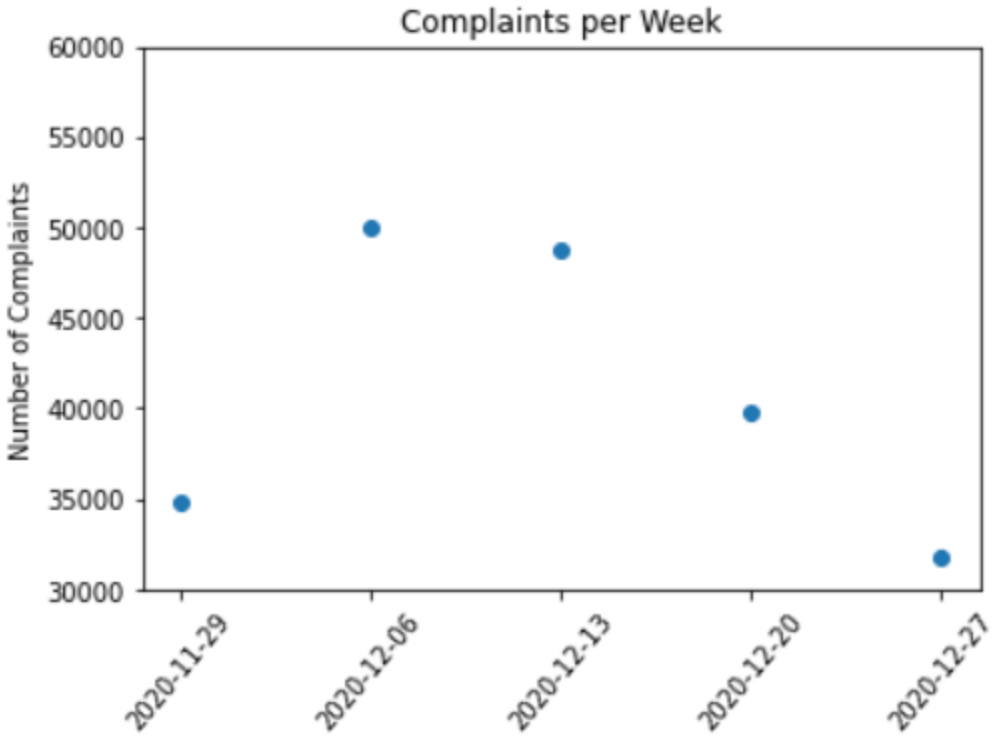
This information is helpful when debugging. If we had had failures in our transform layer, we could see exactly which models had failed, and with extensive testing we can catch failures before they propagate to downstream tables.

seaborn (+ Jupyter notebook)

Now that we are confident that our data is updating daily, let's create some simple plots that could form the basis of a daily dashboard of what the NYC 311 data looks like.

Because we have already transformed the data into the structure we need, plotting is as simple as downloading the data directly to the notebook, and creating the plot we want. No need to transform or merge in python.

Here is the result for the number of complaints per week in December:



The lower number of complaints for the week of 11/29 and 11/27 could be due to the holidays. Alternatively this could be due to those being shortened weeks in our dataset, which spans from 12/1 - 12/31.

Wrap-up

In this demo I showed how to set up a modern, cloud based data analytic pipeline. Starting with ingestion of outside data, transforming that data for my “business” need, and plotting the results. Structuring your analytic pipeline this way makes it easy for analysts to easily add new transforms to the data warehouse and allows a data team to build reliable and useful data assets.

Troubleshooting

Auth BQ and airflow

1. used application default credentials
2. manually removed “quota_project_id”: “myproject” line (https://stackoverflow.com/questions/59858003/using-airflow-with-bigquery-and-cloud-sdk-gives-error-user-must-be-authenticate)
3. Changed the “bigquery_default” connection in airflow admin to use my project (https://stackoverflow.com/questions/46166015/bigquery-with-airflow-missing-projectid)