Testing Data Pipelines

Amitosh Mahapatra - PyCon US 2024

$ about

- “Computer Whisperer” / Data Platform Engineer @ Toplyne
- 🎨 Paints and rides 🚴 bikes for fun, (both human powered and motorbikes)
- Also active in OSS communities in Python & NodeJS (maintains a very popular NodeJS module)
- Sometimes codes for fun

🧑‍💻 linkedin.com/in/amitosh-swain

📝 github.com/recrsn

✉️ recursivefunction.blog

Everyone tells you should test because…

- Your changes won’t drain your data lake
- (Not saying that you should), but you can refactor to your heart's-content, with a whiskey-sour in your hand and still not break prod.
- You really, really, really can’t trust how data comes to you

You will thank yourself later

But how?

- How to begin?
- What do you test?
- Can I do something better?

⚠️ Caveats

The rest of the talk is going to refer Airflow, but the concepts are generic enough for you to apply anywhere, with any tool.

The testing double-pyramid for data pipelines

- Many software engineers preach TDD, unit testing. Many of those concepts aren’t straightforward to apply for data engineering.
- But … don’t discard those yet.
- With careful consideration, we can apply much of those principles and a few data specific ones to make sure our data pipelines keep churning pristine and accurate data

Testing during SDLC

Automated end-to-end testing

- Low effort, high gain
- Comprehensive and end-to-end, will touch a lot of moving parts
- Slow feedback loop
- Costly; Uses real resources
- Difficult to set up
- Handle differences due to variations in date-time, RNGs etc could be difficult
- Run these before deployment

You take several snapshots of real input data and the corresponding output of your production pipeline.

You run your test pipeline and compare your output.

Snapshot-based end-to-end testing

How to setup test suites?

- Test suites are part of your pipeline code, executing on a test infrastructure, pipeline failure => test failure. Imperative to set-up alerting.
- Run queries like SELECT … EXCEPT to compare output to a known stage
- https://github.com/datafold/data-diff is a great tool to simplify comparing tables
- Pandas provide dataframe.diff() to ease data diffing

Unit testing

- Quick and focused
- Fast; uses mocked dependencies
- Treats pipelines as code with behavior
- Uses familiar tools like Pytest, Coverage
- Run these every time you commit some code

Decompose your data pipelines into smaller chunks of Python code, test the behavior with Pytest.

Separate the orchestration bits from the business logic and test them independently

Unit testing

- You can test on how queries are generated for certain inputs
- You can assert what tasks get invoked with what arguments
- You can assert various conditional steps
- You can test the behavior of your orchestration layer, without invoking actual resources
- You can assert your pipeline structure

For example, if using airflow, assert that your DAG is loaded in the DagBag without errors and has all expected tasks.

A very simple unit test

```
def test_crm_users_entity(
    self,
    _mock_load_meta: MagicMock,
    task_factory: HubspotSchemaTransformTaskFactory,
) -> None:
    actual = task_factory.create_crm_entity(entity=Entity.USERS)

    expected = (
        "CREATE OR REPLACE TRANSIENT TABLE TOPLYNE_SCHEMA.TOPLYNE_CRM_USERS AS "
        "SELECT CAST(CONTACT.ID AS VARCHAR) AS CRM_user_id, "
        "SYSDATE() AS _TOPLYNE_CREATED_AT FROM "
        "HUBSPOT.CONTACT AS CONTACT"
    )

    sql_compare(
        expected=expected,
        actual=actual,
    )
```

Another unit test

```
def dag():
    ...

def assert_dag_dict_equal(source, dag):
    assert dag.task_dict.keys() == source.keys()
    for task_id, downstream_list in source.items():
        assert dag.has_task(task_id)
        task = dag.get_task(task_id)
        assert task.downstream_task_ids == set(downstream_list)


def test_dag(dag):
    assert_dag_dict_equal(
        {
            "DummyInstruction_0": ["DummyInstruction_1"],
            "DummyInstruction_1": ["DummyInstruction_2"],
            "DummyInstruction_2": ["DummyInstruction_3"],
            "DummyInstruction_3": [],
        },
        dag,
    )
```

Functional testing

- Testing individual pipeline components (like airflow tasks) in isolation
- Runs on a test environment
- Uses real resources
- Tests if your pipeline input (like queries, dataframe operations) result in correct transformations
-

Functional testing how-to?

- Setup a test environment
- Setup input and output conditions like data, tables as your task needs
- Use a test-runner like pytest and execute your pipeline code
- You can also utilize APIs of your orchestrator to execute a task directly, if possible

Functional testing

```
def test crm users entity(self, snowflake executor: SnowflakeExecutor, task_factory: TaskFactory,
hubspot: HubspotSchemaTransformTaskFactory,) -> None:
    # Setup
    with snowflake executor.connection() as c:
        c.execute sql(
            "CREATE OR REPLACE TRANSIENT TABLE HUBSPOT.CONTACT AS SELECT 1 AS ID, '2022-01-01' AS
CREATED AT"
        )

    task = task factory.query task("hubspot create_crm_entity",
hubspot.create_crm_entity(entity=Entity.USERS))

    with snowflake executor.connection() as c:
        c.execute(task)

    # Validate
    with snowflake executor.connection() as c:
        actual = c.fetch pandas df("SELECT * FROM HUBSPOT.CONTACT")
        expected = pd.DataFrame({"ID": [1], "CREATED_AT": [date_parse("2022-01-01")],})

          pd.testing.assert_frame_equal(actual, expected)
```

Continuous Testing & Monitoring

Why?

- Data is unclean and unpredictable
- Your users need to trust the data you generate
- You may have written a perfect pipeline, but it will break down if data does not adhere to your assumptions
- Having a strongly consistent data model is not possible as scale

Hence, data pipelines must also guarantee data quality

Data Quality Testing

- Check the data model
  - Are all key-references valid?
- Check invariants
  - Is the input and output as expected?
  - Was the input cleaned and sanitized as expected?
  - Do we have unexpected nulls or duplicates?
- Run statistical checks
  - Did the distribution of values change (too many new entries from a new country)
  - Did the input and output rows change drastically
  - Did number of orphaned records change?

Data Quality Testing: How-to?

- Testing in pipeline
  - At the end of each pipeline add a validation task
    - validating your data model
    - Asserting invariants like not-null, duplicates, counts
- Testing outside the pipeline
  - Run a dedicated job checking for distributions, input / output count and report deviations
- Tools:
  - Soda Core
  - Great expectations

What do we do at Toplyne?

- Toplyne processes several hundred terabytes of data from many different customers with wildly different data sources to generate predictions for sales and marketing teams
- Sales and marketing teams use information from Toplyne to power their sales and growth strategies
- Having inaccurate data has direct business implications for us and our customers

...

- We have a framework for abstracting Airflow related code from data pipeline logic
- We unit test for every component of our pipelines with a mocked Airflow executor
- Before merging code, we run functional tests for each component on a test environment
- We run end-to-end tests for every pipeline before deployment
- We have an in-house framework called Litmus, build over soda core to run data quality tests on demand and in schedule