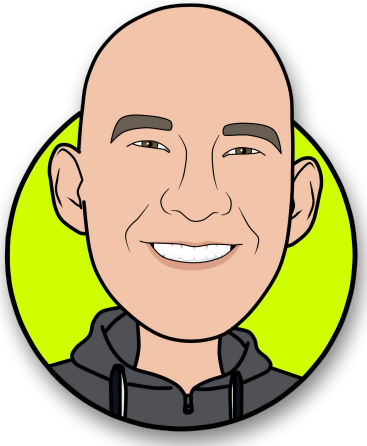
Unleashing Confidence in SQL Development through Unit Testing

Tobias Lampert, Lotum GmbH

December 2024

Lotum

About me



Tobias Lampert

Analytics Engineer, Team BI

Lotum GmbH

Back end developer turned data+AI guy

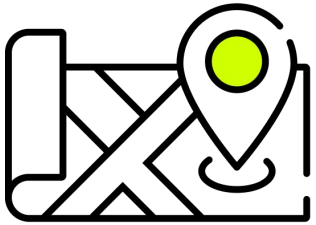
- Data Engineering
- Data Science
- Data Architecture
- Data Platforms

lotum

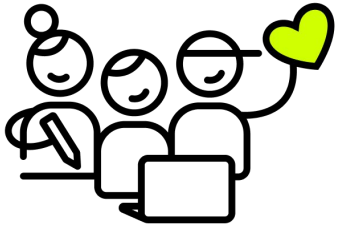
About Lotum

Play together.

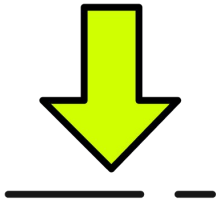
We create mobile games that millions of friends and families play together every day.



Bad Nauheim,
Germany



50
Employees



900M+
Downloads

Lotum franchises

NEW!

4 Pics 1 Word

>400M
Downloads

Quiz Planet

>100M
Downloads

Word Blitz

>100M
Downloads

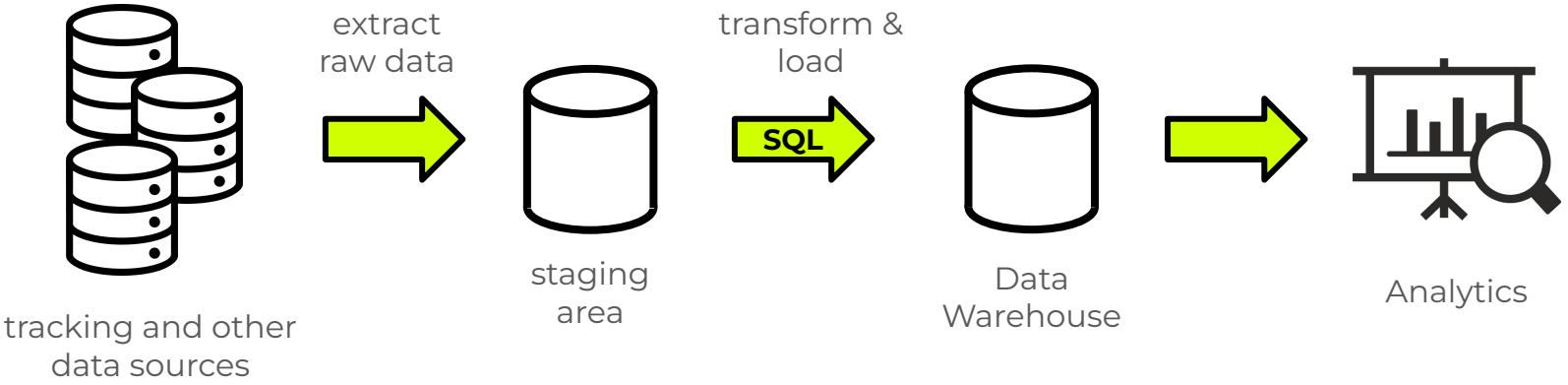
The Test

>100M
Downloads

Classics: Word Games & More

Lotum

Analytics Engineering at Lotum



~300M events per day

many data transformations
performed by
complex SQL statements

lotum

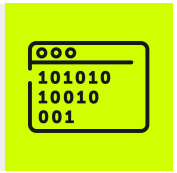
Analytics Engineering at Lotum

How can we make sure the pipeline **produces the expected output** after a code change?

Lotum

The Ripple Effect of a Small Change

1. Imagine making a **minor tweak to a SQL statement...**



2. ...it seems **harmless**, yet it quietly **introduces an error.**

3. ...weeks pass before anyone notices...

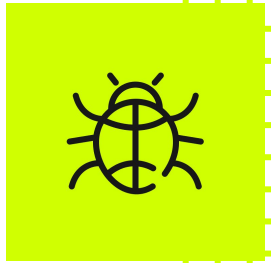
4. ...by then, the error has already **compromised th data insights**...

5. correcting this requires orchestrating a **costly and time-consuming data backfill.**

How can this be prevented?

lotum

Testing SQL Statements



Why does it matter?

ensure
accuracy & **reliability**

meet defined
requirements and
specifications

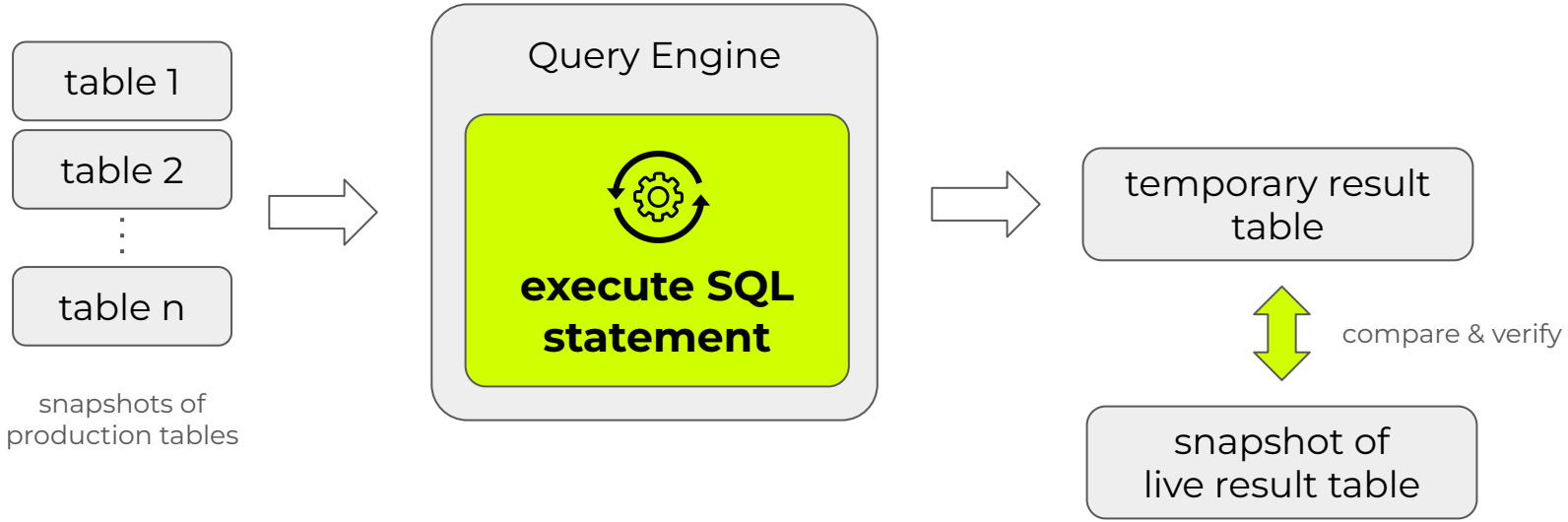
detect errors early in the
development cycle

catch **stealthy errors**

Testing makes sure data is trustworthy!

lotum

Traditional SQL Testing Approach



Disadvantages

- **Stateful**
  Relies on specific data states
- **Data duplication**
  Requires a copy of production data
- **Limited scope**
  Works with existing data only
- **Difficult to isolate**
  Requires a dedicated test environment
- **Non-atomic**
  Pinpointing issues can be challenging
- **Can be slow**
  and resource-intensive

lotum

Unit Tests

What is a Unit Test?

Unit testing

Article Talk

From Wikipedia, the free encyclopedia

In computer programming, **unit testing**, a.k.a. **component** or **module** testing, is a form of software testing by which isolated source code is tested to validate expected behavior.[1]

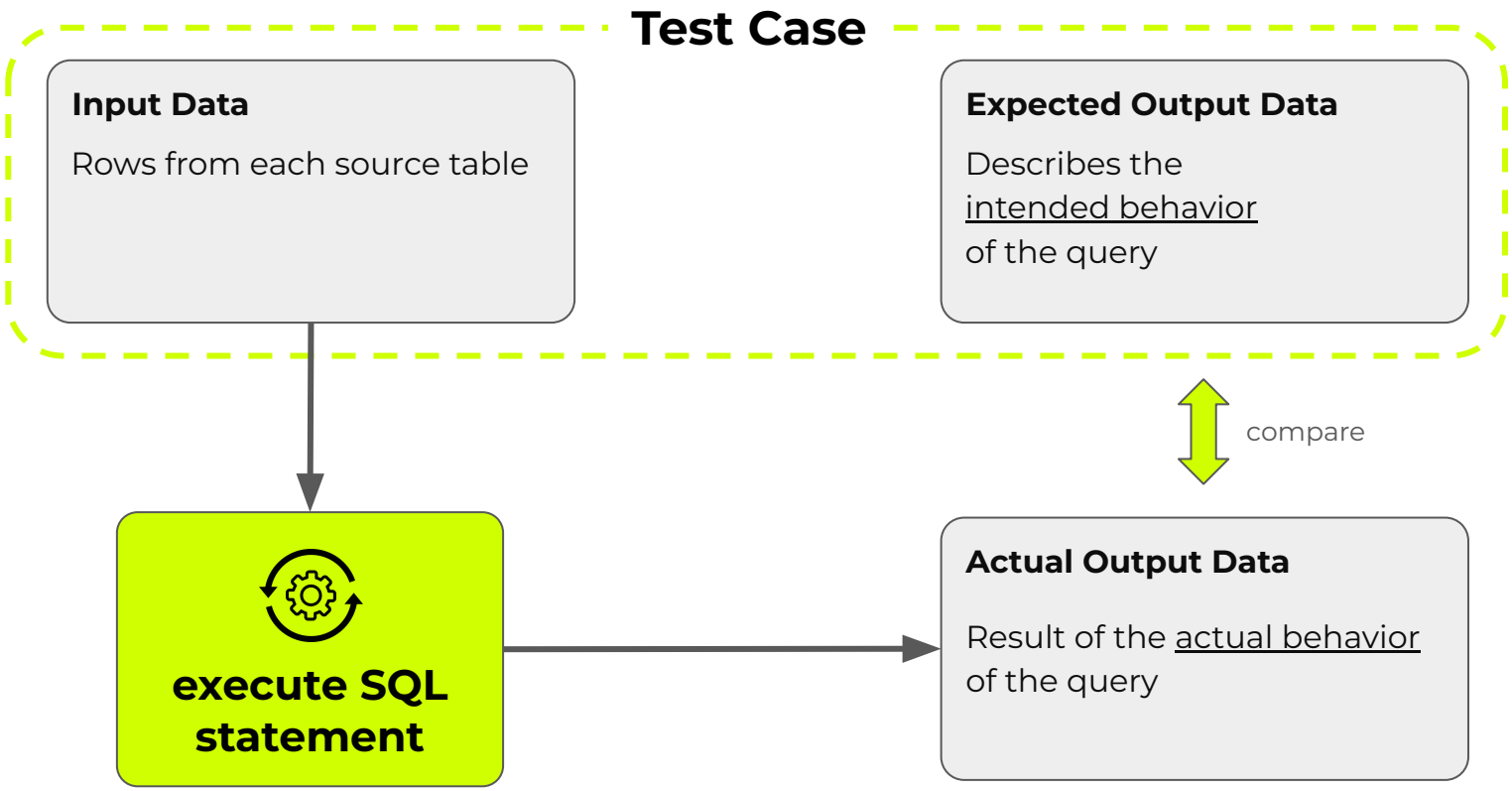
Test case [edit]

Main article: Test case

A test case describes the expected behavior (i.e. output) of the code under test for a particular setup (i.e. input).

lotum

Unit Tests for SQL statements



Test Cases for SQL statements

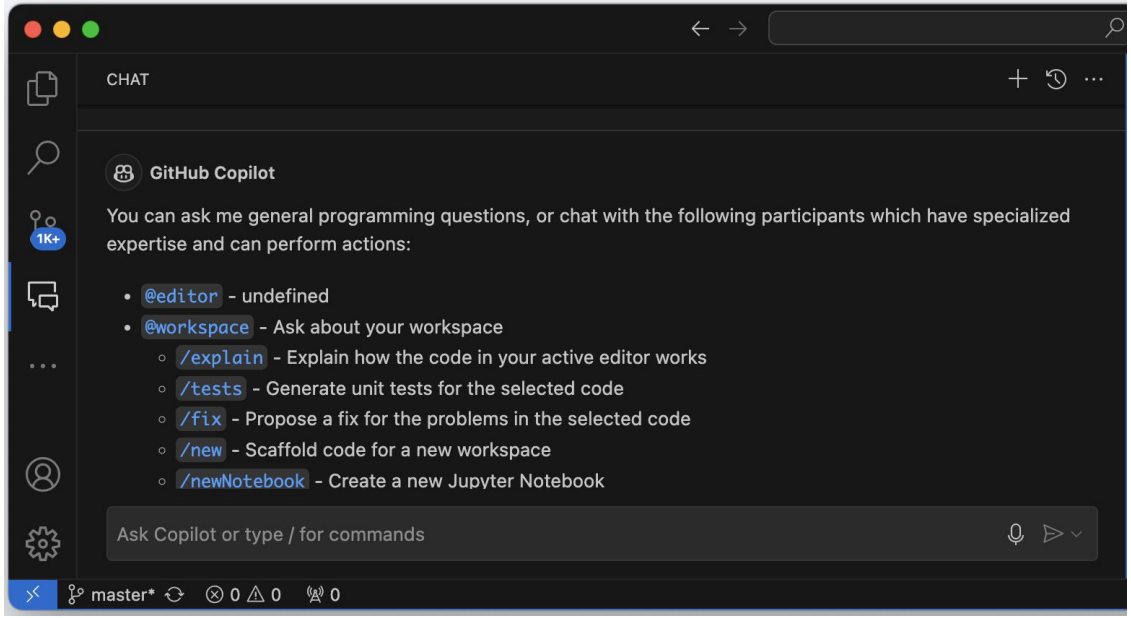
How should a test case look like?

Atomic	Only required fields	Compact

A test case should have **as few input rows** with **as many empty fields** as possible

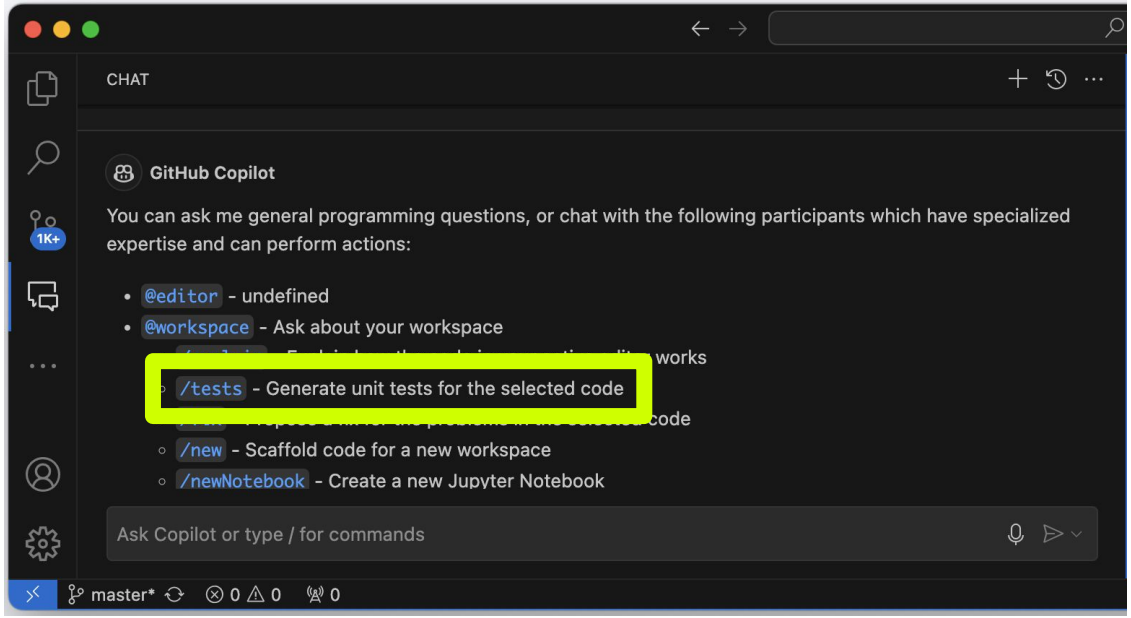
Let’s write some tests

GitHub Copilot



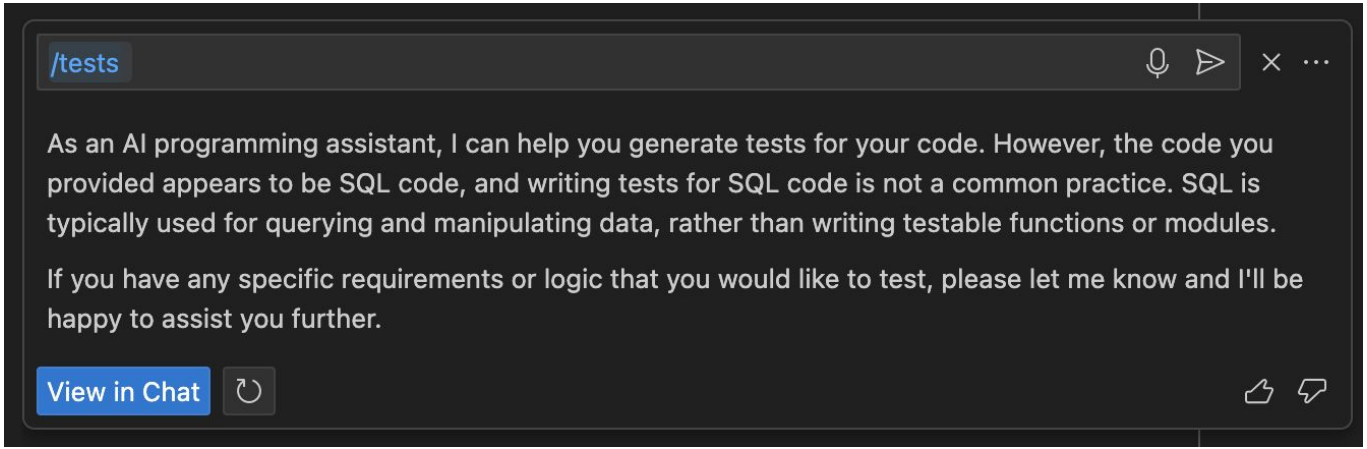
Let’s write some tests

GitHub Copilot



Let’s write some tests

GitHub Copilot



Let’s write some tests

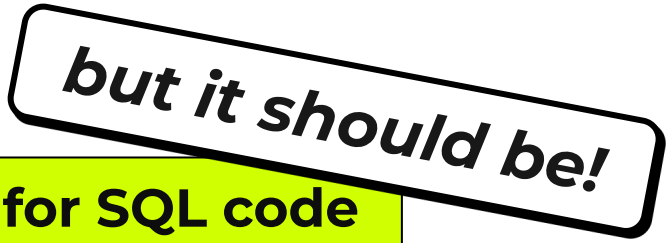
GitHub Copilot

“writing tests for SQL code is not a common practice”

lotum

Let’s write some tests

GitHub Copilot



“writing tests for SQL code is not a common practice”

SQL Unit Testing

Our requirements

No test data persisted in the database

Production-like Testing

Test definition **independent from SQL statement**

Test cases **as code**

lotum

Defining Test Cases as Python Dicts

Each table row is represented as a dict

Employees

name	department	salary
John	1	100
Jack	2	50
Jill	3	200

```
{ "name": "John", "department": 1, "salary": 100 }
{ "name": "Jack", "department": 2, "salary":  50 }
{ "name": "Jill", "department": 3, "salary": 200 }
```

Defining Test Cases as Python Dicts

SQL statement

```
SELECT
  department,
  AVG(salary) AS avg_salary
FROM
  employees
GROUP BY
  department
```

Simple Test case #1

Input data:

```
[
  { "department": 1, "salary": 100 }
]
```

Expected output data:

```
[
  { "department": 1, "avg_salary": 100 }
]
```

Defining Test Cases as Python Dicts

SQL statement

```
SELECT
  department,
  AVG(salary) AS avg_salary
FROM
  employees
GROUP BY
  department
```

Simple Test case #2

Input data:

```
[
  { "department": 1, "salary": 100 },
  { "department": 1, "salary": 50 }
]
```

Expected output data:

```
[
  { "department": 1, "avg_salary": 75 }
]
```

Defining compact test cases

```
[
  {
    "id":          12345,
    "first_name": "John",
    "last_name":   "Doe",
    "hire_date":   date(2000, 1, 1),
    "department": 1,
    "salary":      100,
    "position":    "Engineer",
    "email":       "john.doe@example.com",
    "manager_id": 98765,
    "full_time":   True,
    "address":     "123 Main Street",
    "city":        "Springsville"
  }
]
```

Defining compact test cases

```
default_employee = {
    "id":          12345,
    "first_name": "John",
    "last_name":   "Doe",
    "hire_date":   date(2000, 1, 1),
    "department": 1,
    "salary":      100,
    "position":    "Engineer",
    "email":       "john.doe@example.com",
    "manager_id": 98765,
    "full_time":   True,
    "address":     "123 Main Street",
    "city":        "Springsville"
}
```

Defining compact test cases

Test Case: Average salary calculation

```
[
  default_employee | { "id": 1, "salary": 100 },
  default_employee | { "id": 2, "salary": 50 }
]
```

Test Case: Duplicate email address

```
[
  default_employee | { "id": 1, "email": "john.doe@example.com" },
  default_employee | { "id": 2, "email": "john.doe@example.com" }
]
```

Test Case: Hire date in the future

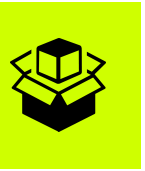
```
[
  default_employee | { "hire_date": date(2025, 1, 1) }
]
```

Verifying Test Cases

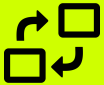
How can we **execute a SQL statement without having the test data stored** in our database tables?

Combine SQL logic and input data

Insert test data into the SQL statement itself

1. insert **virtual tables into the statement** for each reference 

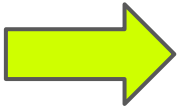
2. **populate virtual tables** with static data 

3. **substitute original table references** by virtual tables 

Transforming Python dicts into SQL

Input data for table “employees”:

```
[
 {
  department: 1,
  salary: 100
 }
]
```



SQL CTE:

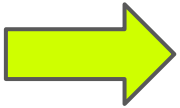
```
WITH mockdata_employees AS (
  SELECT
    1 AS department,
    100 AS salary
)
```

Transforming Python dicts into SQL

Input data for table “employees”:

```
[
 {
  department: 1,
  salary: 100
 },


 {
  department: 1,
  salary: 50
 }
]
```



SQL CTE:

```
WITH mockdata_employees AS (
  SELECT
    1 AS department,
    100 AS salary

  UNION ALL

  SELECT
    1 AS department,
    50 AS salary
)
```

lotum

Injecting test data

```
WITH mockdata_employees AS (
  SELECT
    1 AS department,
    100 AS salary
)
```

CTE with static test case input data

```
SELECT
  department,
  AVG(salary) AS avg_salary
FROM
  employees
GROUP BY
  department
```

Original SQL statement

Injecting test data

```
WITH mockdata_employees AS (
  SELECT
    1 AS department,
    100 AS salary
)
SELECT
  department,
  AVG(salary) AS avg_salary
FROM
  employees
GROUP BY
  department
```

Original SQL statement
including CTE with static test data

Injecting test data

```
WITH mockdata_employees AS (
  SELECT
    1 AS department,
    100 AS salary
)
SELECT
  department,
  AVG(salary) AS avg_salary
FROM
  employees
GROUP BY
  department
```

Original SQL statement
including CTE with static test data

Injecting test data

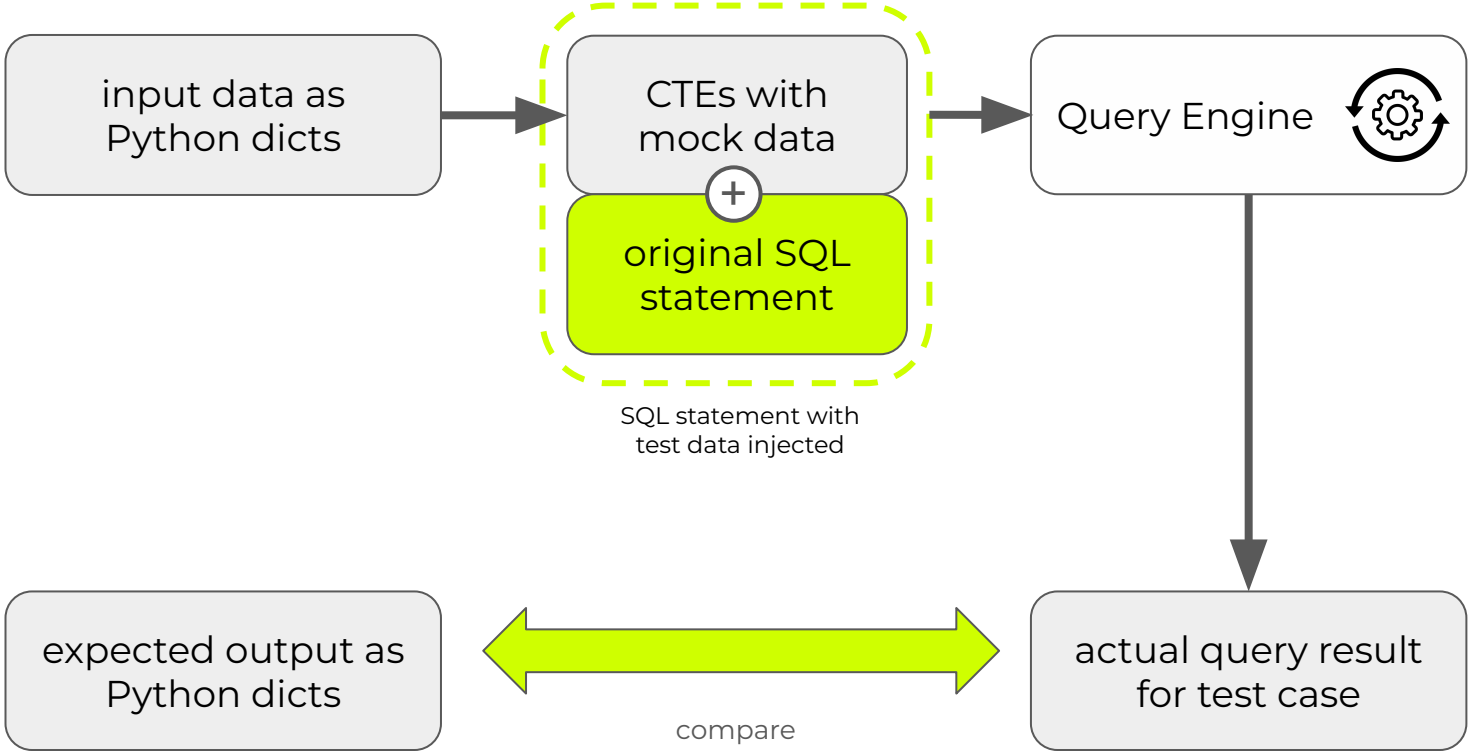
```
WITH mockdata_employees AS (
  SELECT
    1 AS department,
    100 AS salary
)
SELECT
  department,
  AVG(salary) AS avg_salary
FROM
  mockdata_employees
GROUP BY
  department
```

SQL statement with test data injected via CTE

No references to live database tables

Can be executed by the query engine without needing real database tables

Putting it all together



Integration with pytest

```
def test_single_case():
    employees = MockedTable(
        name="employees",
        data=employees_input_rows
    )
    expected = MockedTable(
        name="report",
        data=expected_output_rows
    )
    actual = MockedTable(
        name="report",
        references=[employees]
    )
    assert expected == actual
```

```
[
 { department: 1, salary: 100 },
 { department: 1, salary: 50 }
]
```

```
[
 { department: 1, avg_salary: 75 }
]
```

Running the test suite

```
❯ pytest tests/unit/dm -p no:warnings
================================== test session starts ==================================
platform darwin -- Python 3.8.12, pytest-8.3.2, pluggy-1.5.0
rootdir: /Users/tobiaslampert/_Projects/business-intelligence
configfile: pyproject.toml
plugins: time-machine-2.15.0, anyio-4.5.2
collected 148 items

tests/unit/dm/test_cohort.py ...                                                [  2%]
tests/unit/dm/test_entry_point_log.py ..................                        [ 14%]
tests/unit/dm/test_events.py .....                                              [ 17%]
tests/unit/dm/test_experiment_log.py ......                                     [ 21%]
tests/unit/dm/test_install_log.py ........                                      [ 27%]
tests/unit/dm/test_kohort_io.py ..........                                      [ 33%]
tests/unit/dm/test_mapping.py ......                                            [ 37%]
tests/unit/dm/test_purchases.py ......                                          [ 41%]
tests/unit/dm/test_raw.py ..............................................        [ 74%]
tests/unit/dm/test_reduced_user_segments.py .....                               [ 77%]
tests/unit/dm/test_report.py ..                                                 [ 79%]
tests/unit/dm/test_sessions.py .....                                            [ 82%]
tests/unit/dm/test_staging.py ...................                              [ 95%]
tests/unit/dm/test_user.py ...                                                  [ 97%]
tests/unit/dm/test_user_activity.py .                                           [ 98%]
tests/unit/dm/test_user_segments.py ..                                          [100%]

============================== 148 passed in 389.28s (0:06:29) ==============================
```

Traditional Approach vs. Unit Testing

Traditional approach	Unit testing
Stateful	**Executed on the fly**
Data duplication	**No data needed in the database**
Limited scope	**Can handle test cases with unseen data**
Difficult to isolate	**Run tests at any time, in any environment**
Non-atomic	**Exact test failure diagnosis**
Can be slow	**Light weight test cases, executed quickly**

Frameworks for SQL unit testing

SQLMesh (sqlmesh.com)

- data transformation and modeling framework, backwards compatible with dbt
- can do **tests with mock data as CTEs** (test cases defined in YAML)

SQL Mock (github.com/DeepLcom/sql-mock)

- Python library for mocking SQL Queries with dictionary inputs
- **replaces table references with CTEs** and runs query in the database engine

Conclusion

Improves Code Quality

- Ensures expected behavior
- Can verify correctness for unencountered data scenarios

Minimizes Errors

- Find issues during development
- Safeguards against regressions

Boosts Developer Confidence

- No need for manual verifications
- Makes extensive data comparisons unnecessary
- **Deploying with peace of mind!**

Thank you for your attention!

Contact me:

linkedin.com/in/tlampert

We’re hiring:

Analytics Engineer (w/m/d)

lotum.com/jobs

lotum