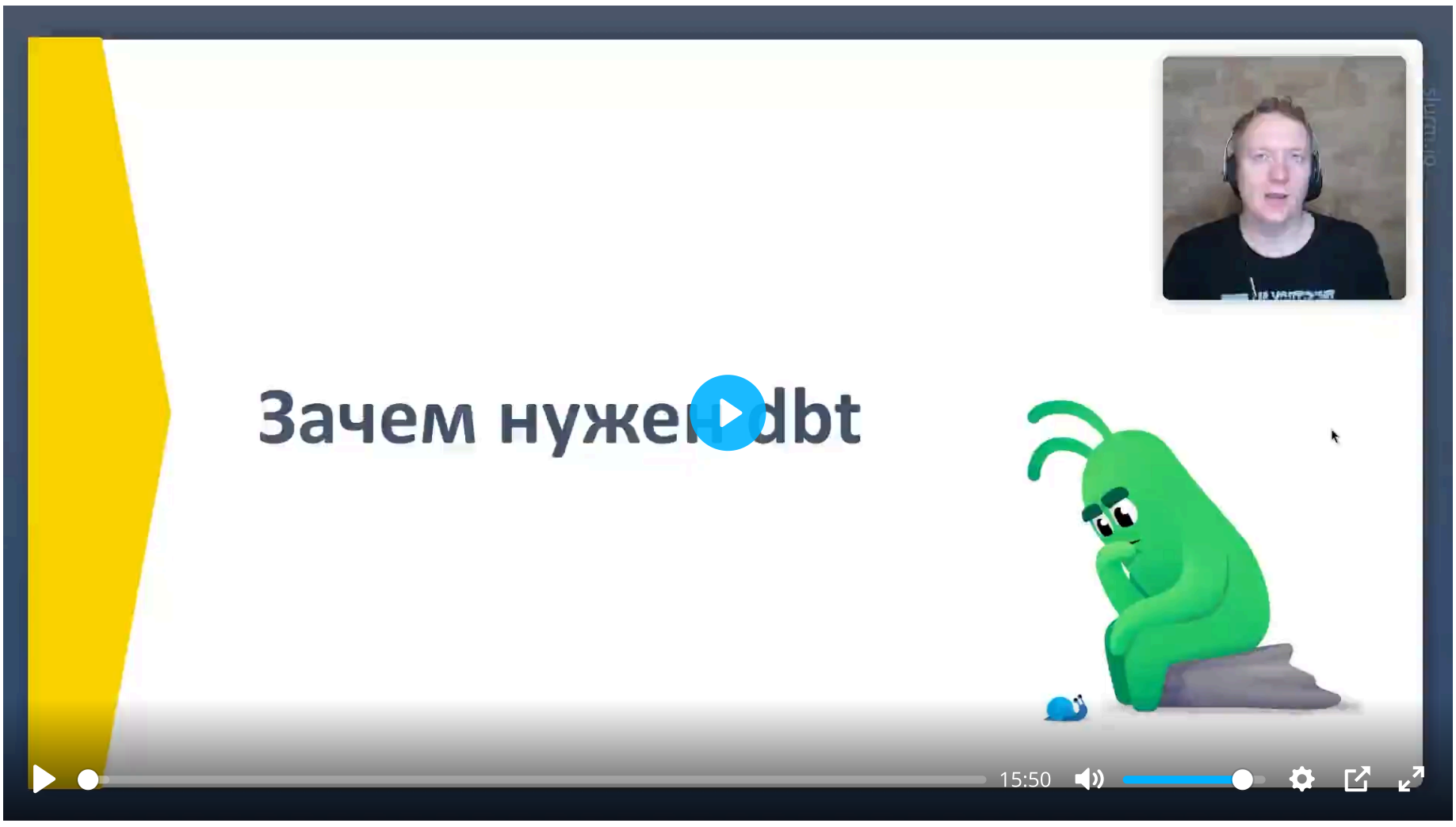


Тексовая расшифровка видео:

ЗАЧЕМ НУЖЕН DBT

План:

- Что такое dbt;
- Первые шаги с dbt;
- Агентство путешествий во времени;
- Базовое Data Quality.

Что такое dbt

Ранее мы говорили о ETL и ELT, а также о том, что инфраструктура для работы с данными строится по слоям.

Если у нас есть ELT, сначала данные мы заливаем в сыром виде, а дальше применяем определенные **трансформации**, чтобы эти данные из сырого слоя разложить по структурированному хранилищу. Чтобы это сделать нам нужно сгенерировать большое количество SQL-запросов. Они сервируют новый слой на основе предыдущего. **Это одна из функций dbt.**

Если же прилетают новые данные, мы хотим иметь возможность их извлекать и записывать в конец (в зависимости от методологии), дописывать в определенные таблицы для того, чтобы они появились в аналитическом хранилище по всем правилам.

Отслеживать весь процесс достаточно тяжело.

Также, например, залив данные, мы хотим **протестировать** то, что получилось. **Это dbt тоже умеет.**

Первые шаги с dbt

Чтобы начать с этим работать, нам достаточно поставить dbt как стандартный набор Python-пакетов.

Обычно ставятся:

```
~$ pip install dbt dbt-core dbt-postgres
```

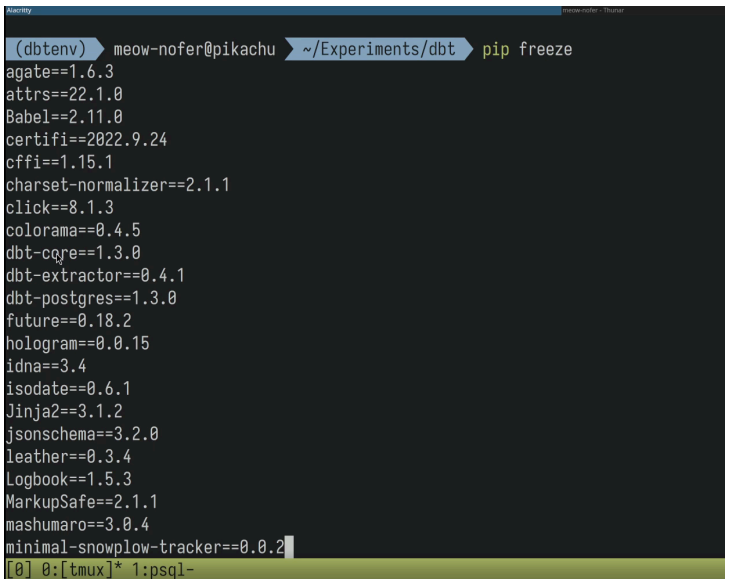
Из коробки dbt поддерживает несколько клаудных хранилищ: Snowflake, Google BigQuery и т.д.

Есть большое количество комьюнити-плагинов, которые добавляют поддержку разных БД.

С помощью команды `~$ dbt init` можем создать структуру проекта.

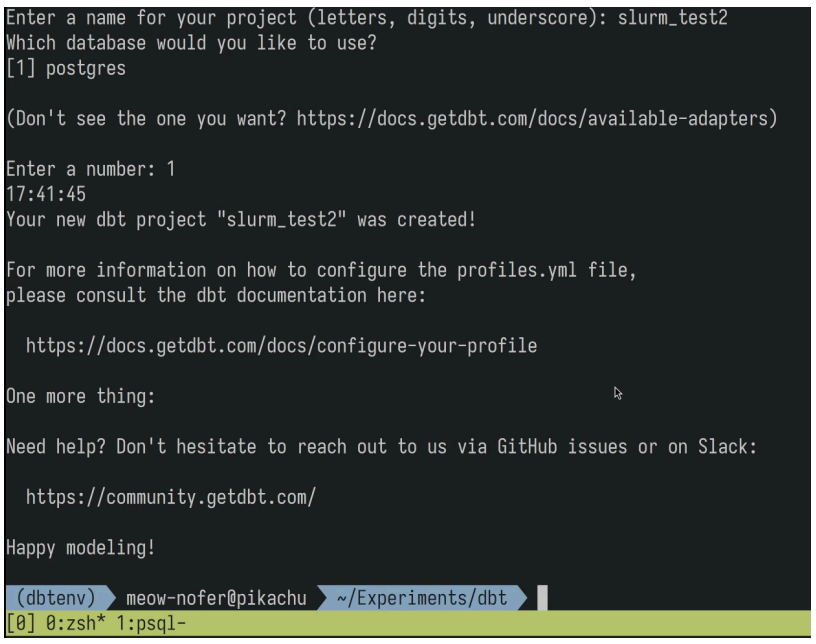
Посмотрим, как это может выглядеть.

Здесь уже установлен dbt со всеми необходимыми зависимостями:

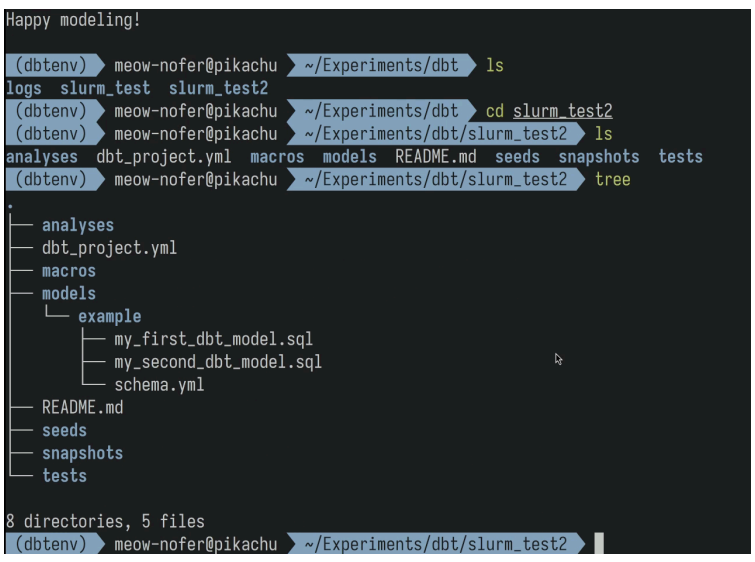


Выполним `~$ dbt init`. Он задаст ряд вопросов:

Также спрашивает, какую базу мы будем использовать. В данный момент у нас стоит плагин для PostgreSQL:



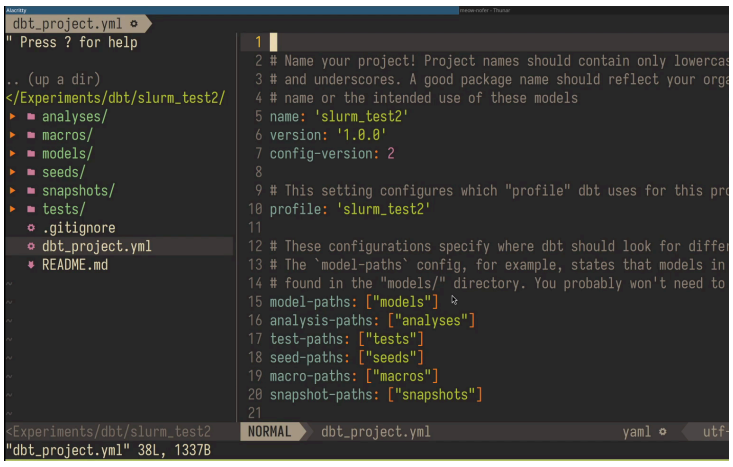
Пойдем в «Тест 2». Здесь видим структуру:



Здесь есть:

- Основной файл «dbt_project.yml».
- Небольшое количество базовых моделей, на основе которых можно что-либо делать.
- Схема, описывающая то, что модели могут создавать (она же выступает источником проверок грамотности заливки).

Если мы посмотрим на dbt_project, то увидим, что в нем перечислено множество разных метаданных:



В первую очередь нас интересуют тесты, модели и макросы (то, куда мы можем писать собственные дополнительные обертки):

```
12 # These configurations specify where dbt should look for differ
13 # The `model-paths` config, for example, states that models in
14 # found in the "models/" directory. You probably won't need to
15 model-paths: ["models"]
16 analysis-paths: ["analyses"]
17 test-paths: ["tests"]
18 seed-paths: ["seeds"]
19 macro-paths: ["macros"]
20 snapshot-paths: ["snapshots"]
21
22 target-path: "target"  # directory which will store compiled SQ
23 clean-targets:         # directories to be removed by `dbt clea
24   - "target"
25   - "dbt_packages"
26
27
28 # Configuring models
```

Видим также дополнительные зависимости и разные плагины, которые можем ставить:

```
28 # Configuring models
29 # Full documentation: https://docs.getdbt.com/docs/configuring-
30
31 # In this example config, we tell dbt to build all models in th
32 # as tables. These settings can be overridden in the individual
33 # using the `{{ config(...) }}` macro.
34 models:
35   slurm_test2:
36     # Config indicated by + and applies to all files under mode
37     example:
38       +materialized: view
```

Однако в первую очередь хотим создавать модельки. Они будут лежать в «slurm_test2»:

```
</Experiments/dbt/slurm_test2/
▸ analyses/
▸ macros/
▾ models/example/
    my_first_dbt_model.sql
    my_second_dbt_model.sql
    schema.yml
▸ seeds/
▸ snapshots/
▸ tests/
  .gitignore
  dbt_project.yml
  README.md
```

Здесь мы можем указать базовый стандартный вариант того, как результат того, что мы напишем в модели будет материализован:

```
models:
  slurm_test2:
    # Config indicated by + and applies to all files under models/example/
    example:
      +materialized: view
```

Для того, чтобы система могла подключиться к БД, нужно где-то задать реквизиты (логин, пароль и т.д.). По умолчанию для этого используется файл в домашнем каталоге:

```
~$ vim ~/.dbt/profiles.yml
```

Также этот файл можно самостоятельно создать в каталоге с самим приложением (**только не забудьте добавить в gitignore, чтобы не залить в систему контроля версий со всеми паролями!**).

Посмотрим на файл с реквизитами:

```
outputs:

  dev:
    type: postgres
    threads: [1 or more]
    host: [host]
    port: [port]
    user: [dev_username]
    pass: [dev_password]
    dbname: [dbname]
    schema: [dev_schema]

  prod:
    type: postgres
    threads: [1 or more]
    host: [host]
    port: [port]
    user: [prod_username]
    pass: [prod_password]
    dbname: [dbname]
    schema: [prod_schema]
```

В случае «slurm_test2» здесь ничего нет. Перед этим мы создали небольшой «slurm_test» и вручную прописали следующее:

```
slurm_test:
  outputs:

    dev:
      type: postgres
      threads: 4
      host: 127.0.0.1
      port: 5432
      user: slurm
      pass: slurm123
      dbname: test_db
      schema: public

    prod:
      type: postgres
      threads: [1 or more]
      host: [host]
      port: [port]
      user: [prod_username]
      pass: [prod_password]
```

Этого достаточно, чтобы dbt смог подключиться к базе.

Зайдем в проект «slurm_test». Он такой же, однако в нем все настроено заранее. По материалам, указанным здесь, вы сможете самостоятельно все произвести.

Проверим, видит ли dbt базу. Введем следующее:

```
(dbtenv) meow-nofer@pikachu ~/Experiments/dbt/slurm_test/slurm_test dbt debug
```

Смотрим:

```
Configuration:
  profiles.yml file [OK found and valid]
  dbt_project.yml file [OK found and valid]

Required dependencies:
 - git [OK found]

Connection:
  host: 127.0.0.1
  port: 5432
  user: slurm
  database: test_db
  schema: public
  search_path: None
  keepalives_idle: 0
  sslmode: None
  Connection test: [OK connection ok]

All checks passed!
 (dbtenv)  meow-nofer@pikachu  ~/Experiments/dbt/slurm_test/slurm_test
[0] 0:nvim- 1:psql  2:zsh*
```

Используя реквизиты, dbt успешно подключился к базе.

Мы можем приступать к созданию моделей.

Агентство путешествий во времени

Опишем две модели.

Если создать базово, то там есть следующие модели:

```
.
├── analyses
├── dbt_project.yml
├── logs
│   └── dbt.log
├── macros
├── models
│   └── example
│       ├── my_first_dbt_model.sql
│       ├── my_second_dbt_model.sql
│       └── schema.yml
├── README.md
├── seeds
├── snapshots
└── tests

9 directories, 6 files
```

Создадим что-то приближенное к реальности, например, агентство путешествий во времени.

У нас будет следующая табличка:

```sql
{{ config(materialized='table') }}

SELECT
    'TARDIS' AS machine_name,
    'Gallifrey' AS origin_planet,
    5 AS passenger_capacity,
    '1963-11-23'::DATE AS debut_date
UNION ALL
SELECT
    'DeLorean',
    'Earth',
    2,
    '1985-07-03'::DATE
UNION ALL
SELECT
    'Hot Tub',
    'Earth',
    4,
    '2010-03-26'::DATE
```

Вне зависимости от написанного в конфиге, мы можем в модели переопределить параметр «materialized» и указать, например, «`materialized='table'`».

Мы создадим табличку с набором машин времени:

- Tardis;

- Delorean;
- Hot tub.

Модель назовем «time_machines.sql» и сможем дальше ее использовать.

Сделаем еще одну табличку:

```
{{ config(materialized='incremental') }}

WITH base AS (
    SELECT
        'TARDIS' AS machine_name,
        'Ancient Rome' AS destination_era,
        '44 BC' AS target_year,
        4 AS passengers,
        '2023-01-01'::DATE AS tour_date
    UNION ALL
    SELECT
        'DeLorean',
        'Wild West',
        '1885 AD',
        2,
        '2023-01-10'::DATE
    UNION ALL
    SELECT
        'Hot Tub',
        'Renaissance Italy',
        '1503 AD',
        4,
        '2023-02-14'::DATE
SELECT *
FROM base
{% if is_incremental() %}
WHERE tour_date > (SELECT MAX(tour_date) FROM {{ this }})
{% endif %}
```

«`materialized='incremental'`» означает, что данная модель может использоваться для добавления новых данных в хранилище.

Если присмотреться, то можно увидеть, что используется CTE'шка (Common Table Expression) и, если мы вызовем эту модель на данных, которые уже существуют, то автоматически их отфильтруем по дате.

Данные объединяются по имени машины времени. У нас будут три путешествия:

- В древний Рим;
- На Дикий Запад;
- В Италию эпохи Возрождения.

Соответственно, это будет еще одна модель «**time_tours.sql**».

Посмотрим, как это будет выглядеть в коде.

Если откроем проект «slurm_test», то увидим файлы с моделями:

```
</Experiments/dbt/slurm_test/
▸ logs/
▾ slurm_test/
  ▸ analyses/
  ▸ dbt_packages/
  ▸ logs/
  ▸ macros/
  ▾ models/example/
      schema.yml
      time_machines.sql
      time_tours.sql
  ▸ seeds/
  ▸ snapshots/
  ▸ target/
  ▸ tests/
    .gitignore
    dbt_project.yml
    README.md
```

Чтобы все это работало вместе, нужно эти коды запустить.

У нас уже есть запущенный PostgreSQL, где на текущий момент ничего нет:

```
Alacritty                                                          meow-nofer - Thunar
 meow-nofer@pikachu  ~/Experiments/dbt  psql -U slurm -d test_db
psql (15.4)
Type "help" for help.

test_db=# \d
Did not find any relations.
test_db=#
```

Запустим наши модели. Видим, что dbt отработал:

```
17:53:12  [WARNING]: Test 'test.slurm_test.unique_my_first_dbt_model_id.16e066b321' (models/exam
a node named 'my_first_dbt_model' which was not found
17:53:12  [WARNING]: Test 'test.slurm_test.not_null_my_first_dbt_model_id.5fb22c2710' (models/ex
n a node named 'my_first_dbt_model' which was not found
17:53:12  [WARNING]: Test 'test.slurm_test.unique_my_second_dbt_model_id.57a0f8c493' (models/exa
 a node named 'my_second_dbt_model' which was not found
17:53:12  [WARNING]: Test 'test.slurm_test.not_null_my_second_dbt_model_id.151b76d778' (models/e
on a node named 'my_second_dbt_model' which was not found
17:53:13  Found 2 models, 0 tests, 0 snapshots, 0 analyses, 289 macros, 0 operations, 0 seed fil
 0 metrics
17:53:13
17:53:13  Concurrency: 4 threads (target='dev')
17:53:13
17:53:13  1 of 2 START sql table model public.time_machines .............................. [RUN]
17:53:13  2 of 2 START sql incremental model public.time_tours ........................... [RUN]
17:53:13  2 of 2 OK created sql incremental model public.time_tours ...................... [SELE
17:53:13  1 of 2 OK created sql table model public.time_machines ......................... [SELE
17:53:13
17:53:13  Finished running 1 incremental model, 1 table model in 0 hours 0 minutes and 0.69 seco
17:53:13
17:53:13  Completed successfully
17:53:13
17:53:13  Done. PASS=2 WARN=0 ERROR=0 SKIP=0 TOTAL=2
```

Для каждой модели вывелся какой-то вывод.

Посмотрим, что поменялось в базе:

```
 meow-nofer@pikachu  ~/Experiments/dbt  psql -U slurm -d test_db
psql (15.4)
Type "help" for help.

test_db=# \d
Did not find any relations.
test_db=# \d
           List of relations
 Schema |     Name      | Type  | Owner
--------+---------------+-------+-------
 public | time_machines | table | slurm
 public | time_tours    | table | slurm
(2 rows)

test_db=#
```

Появились две таблички. Посмотрим, что у них внутри:

```
                                    Table "public.time_machines"
      Column       |  Type   | Collation | Nullable | Default | Storage  | Compression | Stats tar
-------------------+---------+-----------+----------+---------+----------+-------------+----------
 machine_name      | text    |           |          |         | extended |             |
 origin_planet     | text    |           |          |         | extended |             |
 passenger_capacity | integer |           |          |         | plain    |             |
 debut_date        | date    |           |          |         | plain    |             |
Access method: heap

test_db=# \d+ time_tours
                                    Table "public.time_tours"
     Column      |  Type   | Collation | Nullable | Default | Storage  | Compression | Stats target
-----------------+---------+-----------+----------+---------+----------+-------------+--------------
 machine_name    | text    |           |          |         | extended |             |
 destination_era | text    |           |          |         | extended |             |
 target_year     | text    |           |          |         | extended |             |
 passengers      | integer |           |          |         | plain    |             |
 tour_date       | date    |           |          |         | plain    |             |
Access method: heap

test_db=# \d+ time_tours
```

Видим, что все, что мы записывали осталось:

```
test_db=# select * from time_machines;
 machine_name | origin_planet | passenger_capacity | debut_date
--------------+---------------+--------------------+------------
 TARDIS       | Gallifrey     |                  5 | 1963-11-23
 DeLorean     | Earth         |                  2 | 1985-07-03
 Hot Tub      | Earth         |                  4 | 2010-03-26
(3 rows)

test_db=# select * from time_tours;
 machine_name |  destination_era  | target_year | passengers | tour_date
--------------+-------------------+-------------+------------+------------
 TARDIS       | Ancient Rome      | 44 BC       |          4 | 2023-01-01
 DeLorean     | Wild West         | 1885 AD     |          2 | 2023-01-10
 Hot Tub      | Renaissance Italy | 1503 AD     |          4 | 2023-02-14
(3 rows)

test_db=#
```

С одной стороны, мы сами написали SQL-запросы и их выполнили, с другой – мы их описали в определенном виде, указали, в каком виде данные будут храниться, а также сделали первый шаг к тому, чтобы сделать модель инкрементальной.

Базовое Data Quality

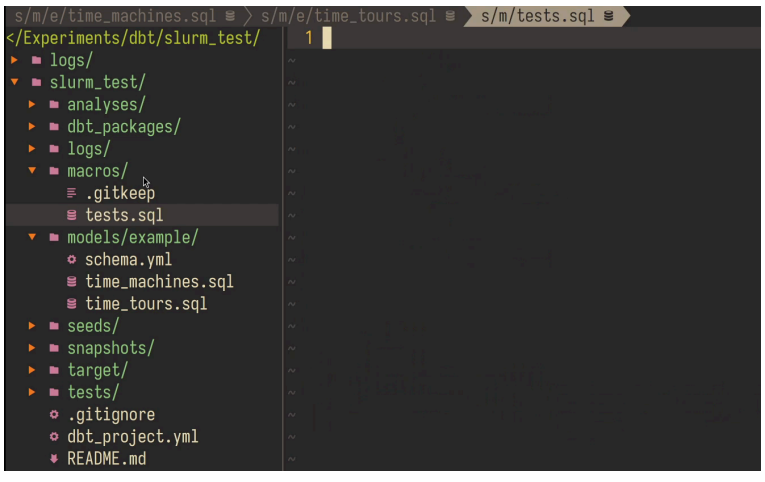
Нам осталось проверить, что данные создаются правильно. Соответственно, нужно проверить некоторые аспекты этих данных.

Рассмотрим пример:

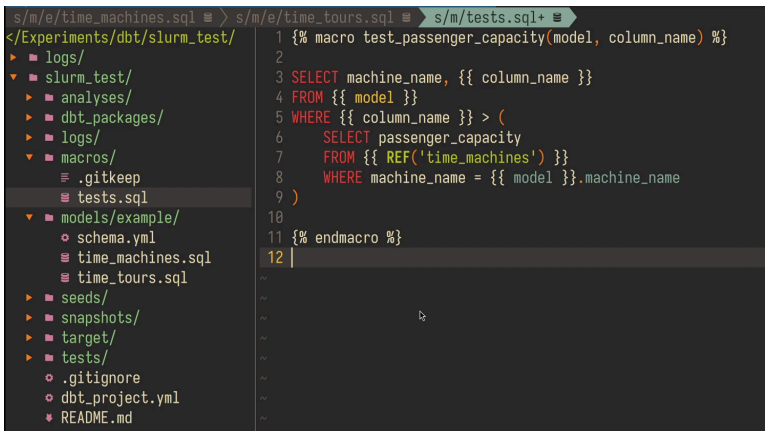
```
{% macro test_passenger_capacity(model, column_name) %}

SELECT machine_name, {{ column_name }}
FROM {{ model }}
WHERE {{ column_name }} > (
    SELECT passenger_capacity
    FROM {{ ref('time_machines') }}
    WHERE machine_name = {{ model }}.machine_name
)

{% endmacro %}
```

Мы можем захотеть проверить превышается ли количество пассажиров в конкретной машине времени. Мы можем описать соответствующий SQL-код в виде запроса.

Создадим следующий файл:

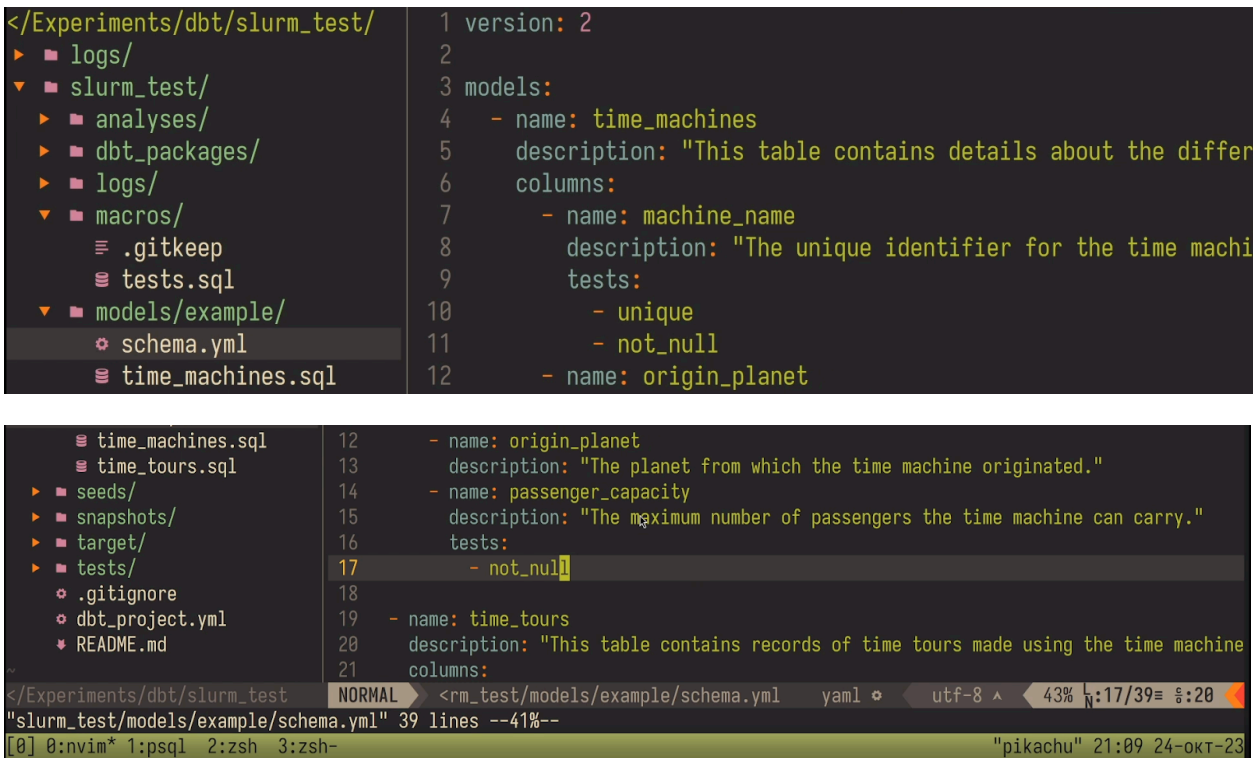


И скопируем содержимое макроса:



Чтобы запустить тесты, нужно описать проверки в файле «tests.sql + schema.yml» (ссылка кликабельная).

Видим файл со схемой, в котором мы перечисляем наборы проверок:



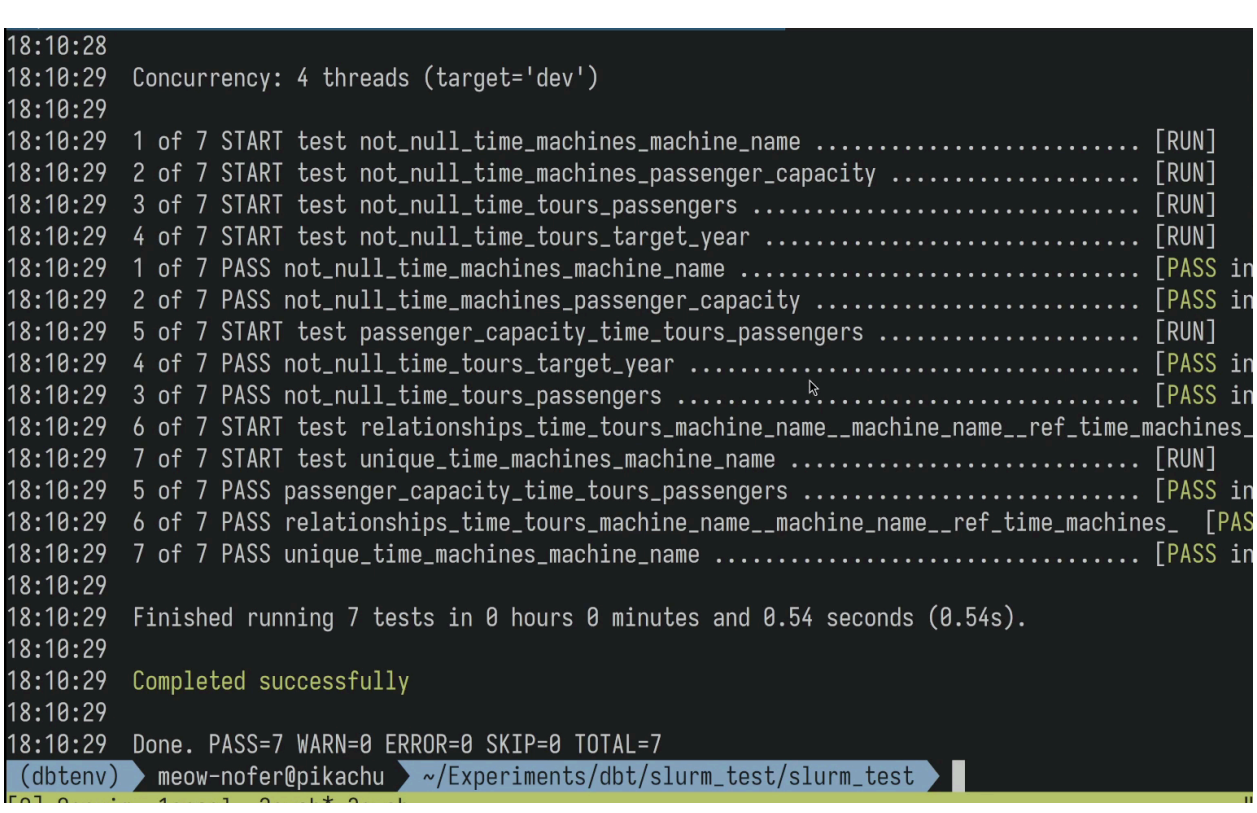
В самом конце мы добавляем тест:

Это и есть вызов макроса. Туда автоматически передаются два аргумента, модель, в которой все это запущено и колонка.

Чтобы это выполнить набираем:

```
(dbtenv) meow-nofer@pikachu ~/Experiments/dbt/slurm_test/slurm_test dbt test
```

Видим следующее:



Все проверки, которые мы описали, успешно пройдены.

Можно использовать более сложные данные, также можно анализировать свежезалитые данные.