Beyond dbt: Modern SQL Transformation and Lineage

Tomas Peluritis

Tomas Peluritis

Head of Data @ Mediatech

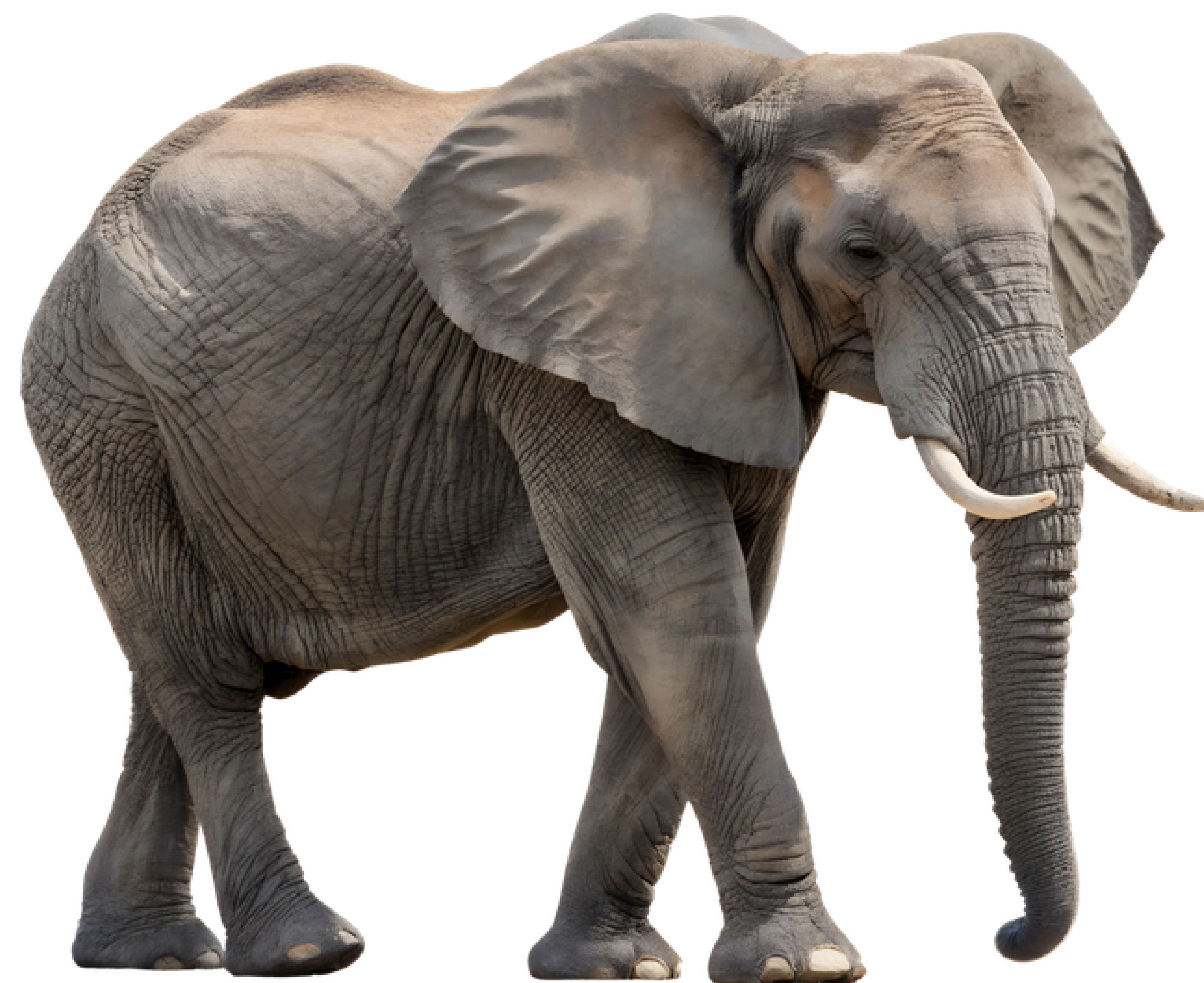
AKA Uncle Data

https://www.linkedin.com/in/tomaspeluritis/

https://podcasters.spotify.com/pod/show/duomenu-dede

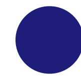
https://uncledata.substack.com

Let´s talk about elephant in the room



SQL Transformation Timeline

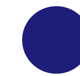
Aug 2025



Slides polished and ready
for November

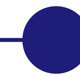
SQL Transformation Timeline

Aug 2025



Slides polished and ready for November

Sept 2025
Acquisition

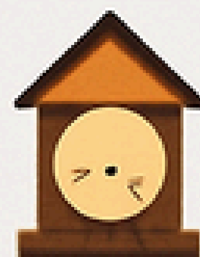


Fivetran acquires Tobiko.

SQL Transformation Timeline

Agenda

- Data Lineage Market Overview
- Quick intro to dbt
- What's SQLGlot and SQLMesh
- Common approaches to implement Data Lineage



Phase 1	Phase 2	Profit
Collect Data Lineage		Profit

$1.5-2B

Lineage market size

The **growing demand** for data lineage solutions reflects the increasing complexity of data ecosystems, with businesses recognizing the need for improved **visibility and accountability** in data management.

Forrester Report

Challengers
Contenders
Strong Performers
Leaders

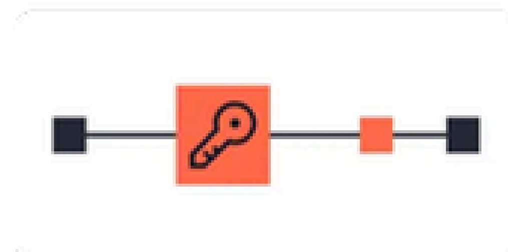
Stronger current offering
Weaker current offering

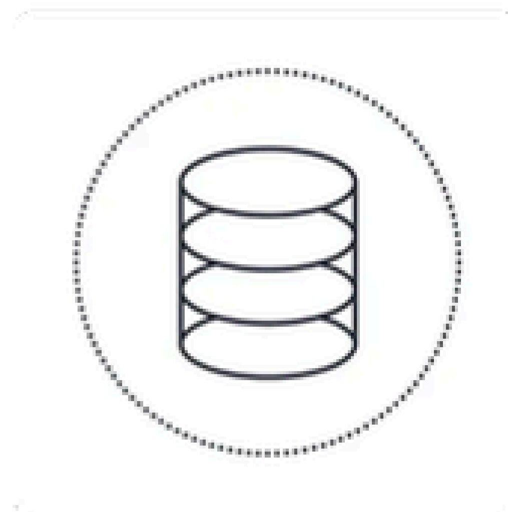
Atlan
Informatica
Alation
BigID
data.world
Collibra
Ataccama
SAS
OvalEdge
Precisely
Qlik
Quest Software

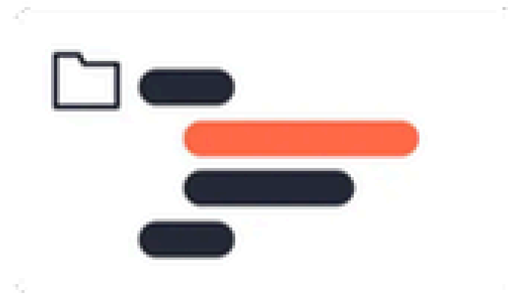
Weaker strategy
Stronger strategy

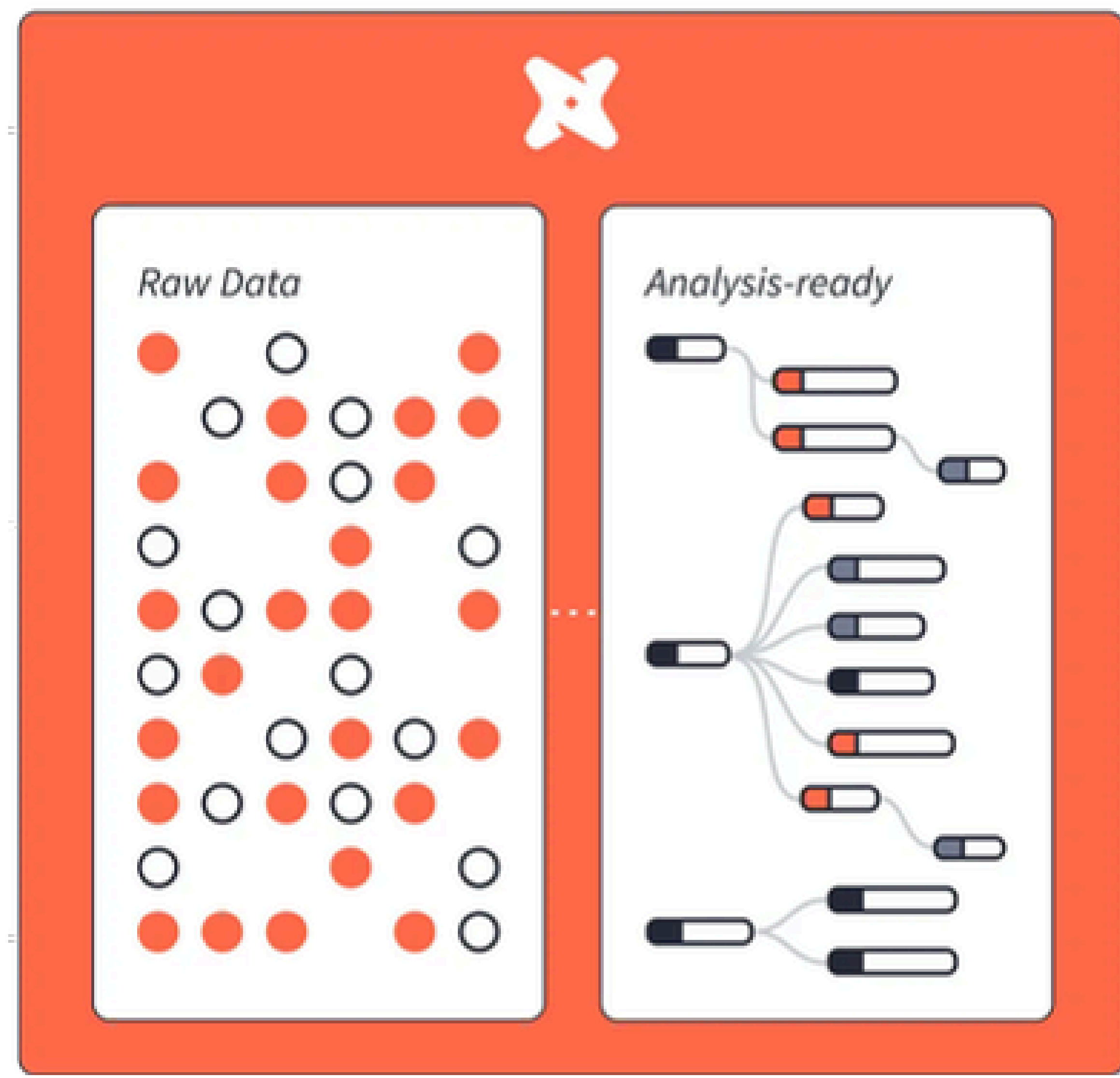
Market presence*

Introduction to dbt









getdbt.com

SQLGlot Deep Dive

SQLGlot Deep Dive

SQL parser, transpiler, optimizer & engine

Zero dependencies

Supports 24+ dialects

SQLGlot Deep Dive

SQL parser, transpiler, optimizer & engine

Zero dependencies

Supports 24+ dialects

Comprehensive SQL parsing with robust testing

Syntactically & semantically correct output

High performance despite pure Python

Format & translate SQL effortlessly

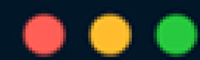
SQLGlot Parser and AST

Abstract Syntax Tree (AST)

Understanding SQLGlot's Abstract Syntax Tree

The Abstract Syntax Tree (AST) visually represents SQL queries, showcasing the structure and relationships within the code, enabling easier parsing, optimization, and transformation for various SQL engines.

Abstract Syntax Tree (AST)



```sql
select first_name, last_name from users where city = 'New York'
```

Abstract Syntax Tree (AST)

SELECT

SQL: SELECT first_name, last_name FROM users WHERE city = 'New York'

Columns

FROM

WHERE

first_name

last_name

users

BinaryOp: =

city

'New York'

Abstract Syntax Tree (AST)

```
Select(
  expressions=[
    Column(
      this=Identifier(this=first_name, quoted=False)),
    Column(
      this=Identifier(this=last_name, quoted=False))],
  from=From(
    this=Table(
      this=Identifier(this=users, quoted=False))),
  where=Where(
    this=EQ(
      this=Column(
        this=Identifier(this=city, quoted=False)),
      expression=Literal(this='New York', is_string=True))))
```

Why ASTs Are Important

Transpilation

Transpilation

input dialect

output dialect

BigQuery

DuckDB

Snowflake

parse

generate

AST

Source

Introduction to SQLMesh

Raw Data

Fivetran & other ingestion tools

CRMs
Transaction databases
SaaS applications
REST APIs

SQLMesh

Transformation and Modeling Framework

Impact Analysis
Unit Tests
Transform
Audit
Blue/Green Deployment

Analytics / BI

AI / ML

Data Sharing

Compatible Query Engines

sqlmesh.com

SQLMesh by Tobiko

- audits
- jaffle-data
- logs
  - sqlmesh_2025_04_11_12_09_20.log
  - sqlmesh_2025_04_11_12_11_51.log
  - sqlmesh_2025_04_11_12_11_59.log
  - sqlmesh_2025_04_11_12_12_19.log
  - sqlmesh_2025_04_11_12_12_38.log
  - sqlmesh_2025_04_11_12_14_02.log
  - sqlmesh_2025_04_11_12_14_18.log
  - sqlmesh_2025_04_11_12_14_57.log
  - sqlmesh_2025_04_11_12_15_05.log
  - sqlmesh_2025_04_11_12_15_41.log
  - sqlmesh_2025_04_11_12_15_44.log
  - sqlmesh_2025_04_11_12_15_49.log
  - sqlmesh_2025_04_11_12_16_10.log
  - sqlmesh_2025_04_11_12_16_14.log
  - sqlmesh_2025_04_11_14_05_48.log
  - sqlmesh_2025_04_11_14_09_49.log
  - sqlmesh_2025_04_11_14_09_56.log
  - sqlmesh_2025_04_11_14_10_12.log
  - sqlmesh_2025_04_11_14_15_16.log
  - sqlmesh_2025_04_11_14_16_29.log
- macros
- models
  - marts
  - seeds
  - staging
    - stg_customers.sql
    - stg_locations.sql
    - stg_order_items.sql
    - stg_orders.sql
    - stg_products.sql
    - stg_supplies.py
- _dummy_sqlmesh.yaml
- config.yaml

Custom SQL 1 | stg_supplies.py

```python
from sqlmesh.core.model import model
import pandas as pd
import hashlib

@model(
    "jaffle_shop.staging.stg_supplies",
    kind="FULL",
    cron="@daily",
    columns={
        "supply_uuid": "STRING",
        "supply_id": "STRING",
        "product_id": "STRING",
        "supply_name": "STRING",
        "supply_cost": "FLOAT",
        "is_perishable_supply": "BOOLEAN"
    },
    column_descriptions={
        "supply_uuid": "The unique key of our supplies per cost",
        "supply_id": "Supply identifier",
        "product_id": "Product identifier",
        "supply_name": "Supply name",
        "supply_cost": "Supply cost in dollars",
        "is_perishable_supply": "Whether the supply is perishable"
    },
    description="""
        List of our supply expenses data with basic cleaning and transformation applied.
        One row per supply cost, not per supply. As supply costs fluctuate they receive
        a new row with a new UUID. Thus there can be multiple rows per supply_id.
    """
)
def stg_supplies(context, **kwargs):
    # Get the raw supplies data
    raw_supplies = context.fetchdf(f"select * from {context.resolve_table('jaffle_shop.raw.raw_supplies')}")

    # Apply transformations
    df = pd.DataFrame(raw_supplies)

    # Generate surrogate key directly (not using the macro)
    df['supply_uuid'] = df.apply(lambda row: hashlib.md5(f"{row['id']}-{row['sku']}".encode()).hexdigest(), axis=1)

    # Convert cents to dollars
    df['supply_cost'] = df['cost'] / 100

    # Rename columns
    df = df.rename(columns={
        'id': 'supply_id',
        'sku': 'product_id',
        'name': 'supply_name',
        'perishable': 'is_perishable_supply'
    })

    # Select only needed columns
    df = df[['supply_uuid', 'supply_id', 'product_id', 'supply_name', 'supply_cost', 'is_perishable_supply']]

    return df
```

Saved: ● Language: Python Dialect: postgres SQLMesh Type: Model

Lineage

jaffle_shop.staging.stg_supplies

SQLMesh Capabilities

SQLMesh State Database

It has a DB for state

```
{% if is_incremental() %}

  -- this filter will only be applied on an incremental run
  -- (uses >= to include records whose timestamp occurred since the last run of this model)
  -- (If event_time is NULL or the table is truncated, the condition will always be true and load all records
where event_time >= (select coalesce(max(event_time),'1900-01-01') from {{ this }} )

{% endif %}
```

It has a DB for state

```
{% if is_incremental() %}

  -- this filter will only be applied                    run
  -- (uses >= to include records whose               rred since the last run of this model)
  -- (If event_time is NULL or the table is            d, the condition will always be true and load all records
where event_time >= (select coalesce(max(e                 00-01-01') from {{ this }} )

{% endif %}
```

```
sqlmesh
  Tables 7
    _auto_restatements -1 rows, 16KB
    _environment_statements -1 rows, 16KB
    _environments -1 rows, 16KB
    _intervals -1 rows, 32KB
    _plan_dags -1 rows, 24KB
    _snapshots -1 rows, 24KB
    _versions -1 rows, 16KB
```

It has a DB for state

Modified Directly

staging__dev.stg_orders

Breaking Change

- marts__dev.orders
- marts__dev.customers
- marts__dev.order_items

- (•) **Breaking Change** — It will rebuild all models
- ( ) **Non-Breaking Change** — It will exclude all indirect models caused by this change
- ( ) **Forward-Only Change** — The change requires no rebuilding

```
---
+++

@@ -2,7 +2,13 @@

  name jaffle_shop.staging.stg_orders,
  start '2016-09-01',
  dialect postgres,
- kind FULL,
+ kind INCREMENTAL_BY_TIME_RANGE (
+   time_column ("ordered_at", 'YYYY-MM-DD'),
+   partition_by_time_column TRUE,
+   forward_only FALSE,
+   disable_restatement FALSE,
+   on_destructive_change 'ERROR'
+ ),
  columns (
```

It has a DB for state

```
  @cents_to_dollars(order_total) AS order_total,
  CAST(ordered_at AS DATE) AS ordered_at
FROM jaffle_shop.raw.raw_orders
+WHERE
+  ordered_at BETWEEN @start_ds AND @end_ds
```

marts__dev.orders

Breaking Change

Modified Indirectly

- marts__dev.customers
- marts__dev.order_items

Backfills

Models 4

- marts__dev.customers
- marts__dev.order_items
- marts__dev.orders
- staging__dev.stg_orders

dbt vs SQLMesh Models

Model Definition

```sql
select
    id as customer_id,
    name as customer_name
from {{ source('ecom', 'raw_customers') }}
```

```yaml
models:
  - name: stg_customers
    description: Customer data with basic cleaning and transformation applied, one row per customer.
    columns:
      - name: customer_id
        description: The unique key for each customer.
        data_tests:
          - not_null
          - unique
```

SQLMesh

```sql
MODEL (
  name jaffle_shop.staging.stg_customers,
  kind FULL,
  cron '@daily',
  description 'Customer data with basic cleaning and transformation applied, one row per customer',
  columns (
    customer_id STRING COMMENT 'The unique key for each customer',
    customer_name STRING COMMENT 'Customer full name'
  )
);

SELECT
  id AS customer_id,
  name AS customer_name
FROM
  jaffle_shop.raw.raw_customers
```

Testing Approaches Comparison

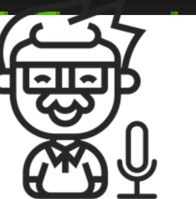
dbt

```yaml
models:
  - name: customers
    description: Customer overview data mart, offering key details for each unique customer. One row per customer.
    data_tests:
      - dbt_utils.expression_is_true:
          expression: "lifetime_spend_pretax + lifetime_tax_paid = lifetime_spend"
    columns:
      - name: customer_id
        description: The unique key of the orders mart.
        data_tests:
          - not_null
          - unique
      - name: customer_name
        description: Customers' full name.
      - name: count_lifetime_orders
        description: Total number of orders a customer has ever placed.
      - name: first_ordered_at
        description: The timestamp when a customer placed their first order.
      - name: last_ordered_at
        description: The timestamp of a customer's most recent order.
      - name: lifetime_spend_pretax
        description: The sum of all the pre-tax subtotals of every order a customer has placed.
      - name: lifetime_tax_paid
        description: The sum of all the tax portion of every order a customer has placed.
      - name: lifetime_spend
        description: The sum of all the order totals (including tax) that a customer has ever placed.
      - name: customer_type
        description: Options are 'new' or 'returning', indicating if a customer has ordered more than once or has only placed their first order to date.
        data_tests:
          - accepted_values:
              values: ["new", "returning"]
```

SQLMesh

```sql
audits (
  /* Primary key validation */
  not_null(columns := (customer_id)),
  unique_values(columns := (customer_id)),

  /* Data integrity checks */
  forall(criteria := (
    ABS(lifetime_spend_pretax + lifetime_tax_paid - lifetime_spend) < 0.01
  )),

  /* Categorical values validation */
  accepted_values(column := customer_type, is_in := ('new', 'returning')),

  /* Logical consistency checks */
  forall(criteria := (
    (customer_type = 'returning' AND count_lifetime_orders > 1) OR
    (customer_type = 'new' AND count_lifetime_orders <= 1)
  )),

  /* Temporal logic checks */
  forall(criteria := (
    (first_ordered_at IS NULL) OR (last_ordered_at IS NULL) OR (first_ordered_at <= last_ordered_at)
  )),

  /* Non-negative values for monetary fields */
  forall(criteria := (
    lifetime_spend_pretax >= 0
  )),
  forall(criteria := (
    lifetime_tax_paid >= 0
  )),
  forall(criteria := (
    lifetime_spend >= 0
  )),

  /* Check repeat buyer flag consistency */
  forall(criteria := (
    (is_repeat_buyer = TRUE AND count_lifetime_orders > 1) OR
    (is_repeat_buyer = FALSE AND count_lifetime_orders <= 1)
  ))
)
```

```
Found 10 audit(s).
not_null on model jaffle_shop.marts.customers ✅ PASS.
unique_values on model jaffle_shop.marts.customers ✅ PASS.
forall on model jaffle_shop.marts.customers ✅ PASS.
accepted_values on model jaffle_shop.marts.customers ✅ PASS.
forall on model jaffle_shop.marts.customers ✅ PASS.
forall on model jaffle_shop.marts.customers ✅ PASS.
forall on model jaffle_shop.marts.customers ✅ PASS.
forall on model jaffle_shop.marts.customers ✅ PASS.
forall on model jaffle_shop.marts.customers ✅ PASS.
forall on model jaffle_shop.marts.customers ✅ PASS.

Finished with 0 audit errors and 0 audits skipped.
Done.
```



Breaking Changes Comparison

Differences

```
Differences from the `prod` environment:

Models:
└── Metadata Updated:
    └── marts.customers
---

+++

@@ -14,6 +14,62 @@

     lifetime_tax_paid DOUBLE PRECISION,
     lifetime_spend DOUBLE PRECISION,
     customer_type TEXT
+  ),
+  audits (
+    NOT_NULL('columns' = (
+      customer_id
+    )),
+    UNIQUE_VALUES('columns' = (
+      customer_id
+    )),
+    FORALL(
+      'criteria' = (
+        ABS(lifetime_spend_pretax + lifetime_tax_paid - lifetime_spend) < 0.01
+      )
+    ),
+    ACCEPTED_VALUES('column' = customer_type, 'is_in' = ('new', 'returning')),
+    FORALL(
+      'criteria' = (
+        (
+          customer_type = 'returning' AND count_lifetime_orders > 1
+        )
+        OR (
+          customer_type = 'new' AND count_lifetime_orders <= 1
+        )
+      )
+    ),
+    FORALL(
```

Breaking Changes

```
Models:
├── Directly Modified:
│   └── staging.stg_customers
└── Indirectly Modified:
    └── marts.customers

---

+++

@@ -9,6 +9,5 @@

   )
 )
 SELECT
-  id AS customer_id,
-  name AS customer_name
+  id AS customer_id
 FROM jaffle_shop.raw.raw_customers

Directly Modified: staging.stg_customers (Breaking)
└── Indirectly Modified Children:
    └── marts.customers (Indirect Breaking)
Models needing backfill:
├── marts.customers: [full refresh]
└── staging.stg_customers: [full refresh]
Apply - Backfill Tables [y/n]:
```

*Bonus: suggests to backfill as well

Breaking Changes

```
Directly Modified: staging.stg_customers (Breaking)
└── Indirectly Modified Children:
    └── marts.customers (Indirect Breaking)
Models needing backfill:
├── marts.customers: [full refresh]
└── staging.stg_customers: [full refresh]
Apply - Backfill Tables [y/n]: y

Updating physical layer ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100.0% • 2/2 • 0:00:00

✔ Physical layer updated

Executing model batches ━━━━━━━━━━━━━━━━━━━━╺━━━━━━━━━━━━━━━━━━━ 50.0% • 1/2 • 0:00:00

Skipped models

  "jaffle_shop"."marts"."customers"

Failed models

  "jaffle_shop"."staging"."stg_customers"

    UndefinedColumn:
      column "customer_name" does not exist
      LINE 1: ...tomer_id", "customer_name") SELECT "customer_id", "customer_...
                                                                  ^
      HINT:  There is a column named "customer_name" in table "staging__stg_customers__2030330001", but it cannot be referenced from this part of the query.

Error: Plan application failed.
```

Detection Methods Comparison

Manual vs. Automatic Processes

dbt Manual Detection

Manual detection in dbt involves **extensive code review**, which can be time-consuming and prone to human error, leading to delays in identifying breaking changes during the development process.

SQLMesh Automatic Detection

SQLMesh streamlines the detection process by offering **automatic identification** of breaking changes, which accelerates development cycles and enhances overall reliability in managing SQL transformation projects.

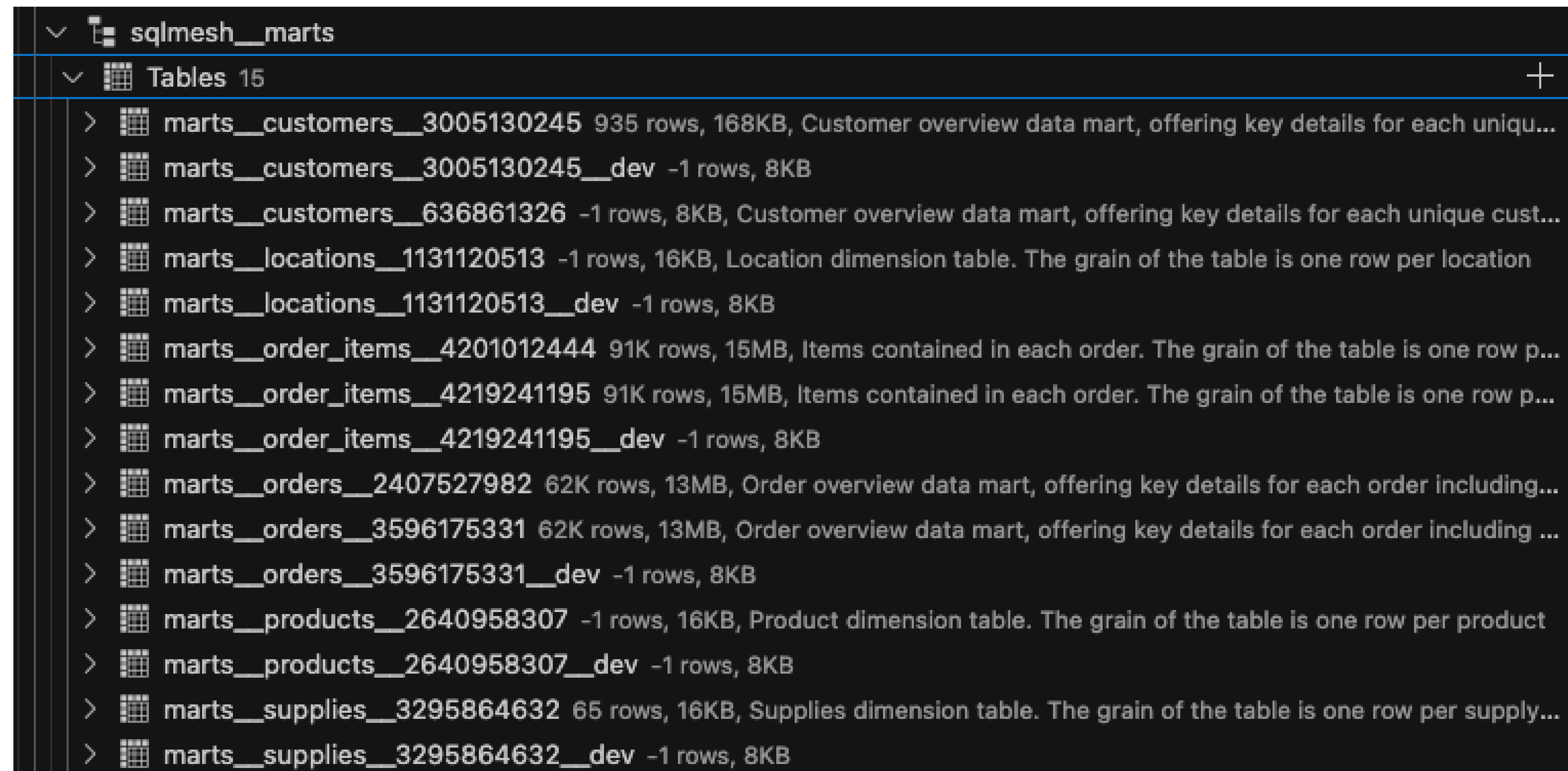
Efficiency of Processes

The shift from manual to automatic detection not only improves **efficiency** but also fosters a more collaborative environment, allowing teams to focus on innovation rather than burdening themselves with manual checks.

Physical vs. Virtual Versioning

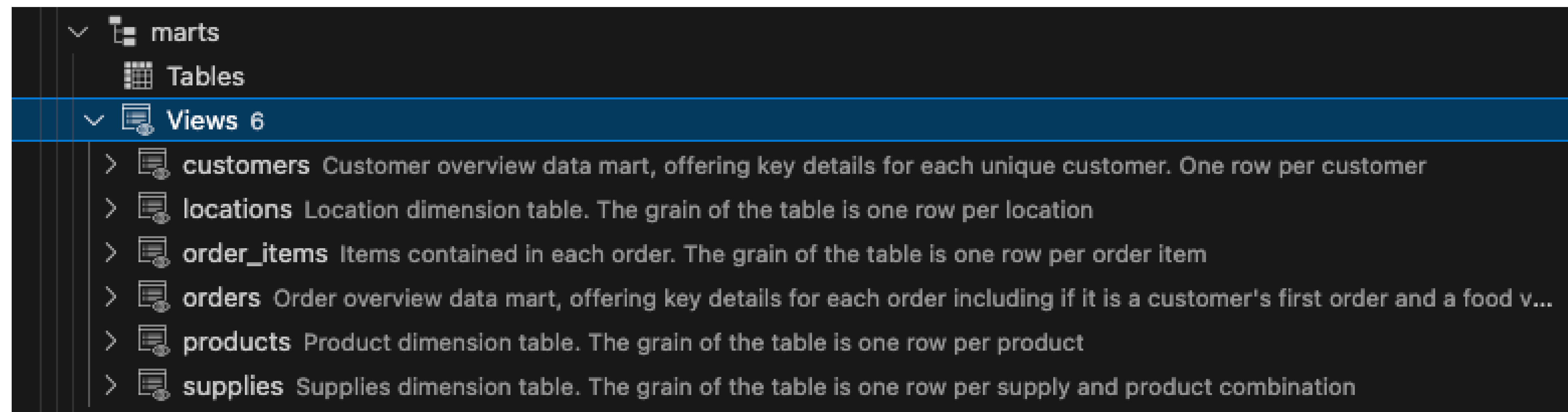
Data Versioning

Physical layer



Data Versioning

Virtual layer



*Bonus adds metadata to DB if possible

Data Versioning

Virtual layer

```sql
CREATE OR REPLACE VIEW marts.customers
AS SELECT marts__customers__3005130245.customer_id,
    marts__customers__3005130245.customer_name,
    marts__customers__3005130245.count_lifetime_orders,
    marts__customers__3005130245.is_repeat_buyer,
    marts__customers__3005130245.first_ordered_at,
    marts__customers__3005130245.last_ordered_at,
    marts__customers__3005130245.lifetime_spend_pretax,
    marts__customers__3005130245.lifetime_tax_paid,
    marts__customers__3005130245.lifetime_spend,
    marts__customers__3005130245.customer_type
   FROM sqlmesh__marts.marts__customers__3005130245;
```

Lineage Approaches

Lineage Approaches: Just Buy a Tool that fits your need

Lineage Approaches: dbt Native

Just Use dbt

```sql
with orders as (

    select * from {{ ref('int_order_payments_pivoted') }}

)
,
customers as (

    select * from {{ ref('int_customer_order_history_joined') }}

)
,
final as (

    select
        *
    from orders
    left join customers using (customer_id)

)

select * from final
```

Source

...but

WHERE NON-SQL SOURCES?

imgflip.com

...messy UI

Source

...extra features = extra money

Column
level lineage
in dbt core

Column-level
lineage in dbt
cloud ($$$)

Lineage Approaches: Airflow OpenLineage

openlineage-airflow

OL

MARQUEZ

Airflow executes a set of DAGs that operate on various data sources (e.g., Snowflake, Redshift, Bigquery, PostgreSQL)

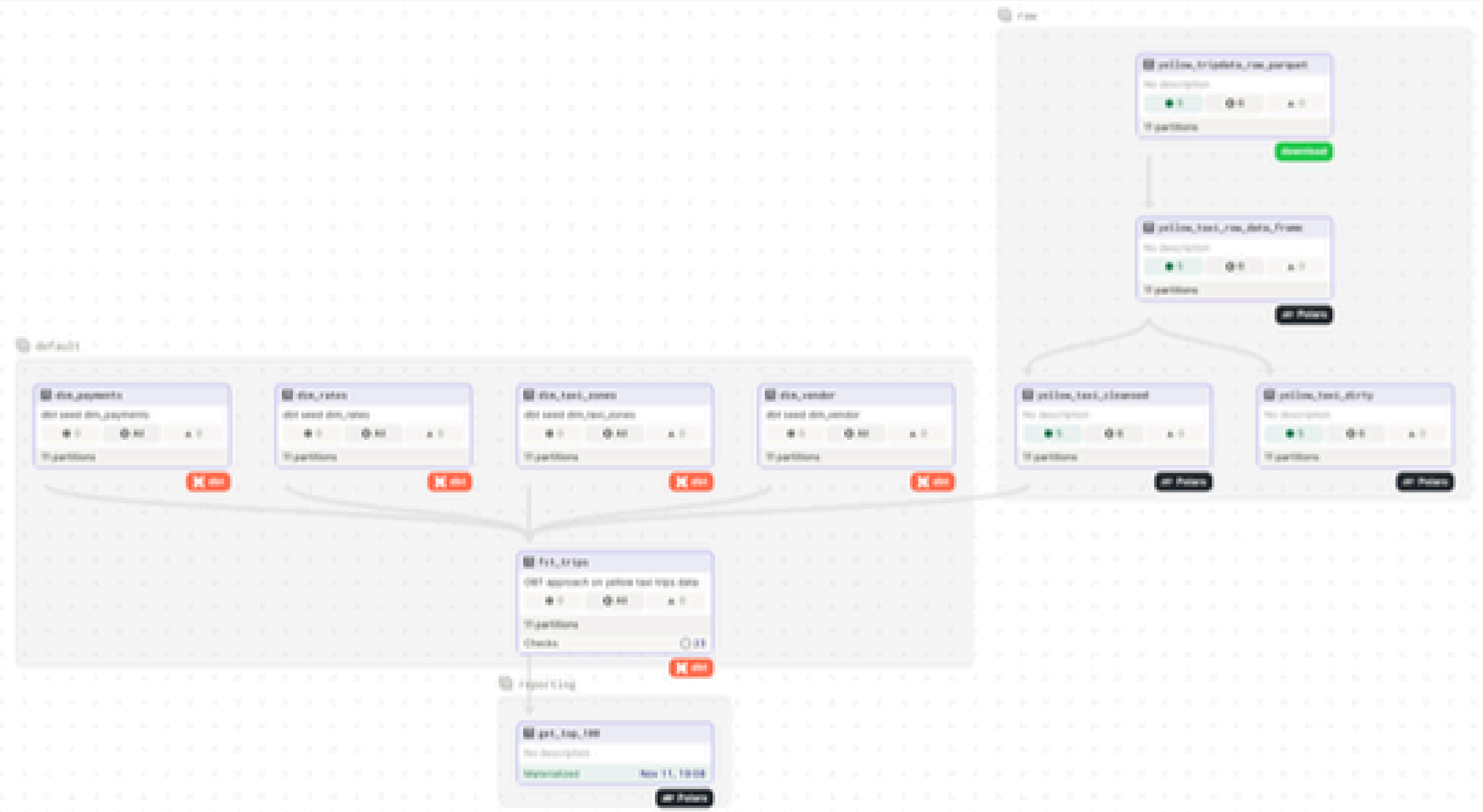
The openlineage-airflow Python module automatically captures metadata and calls the OpenLineage Collection API

Metadata is stored in an OpenLineage-compatible repository like Marquez, where it can be analyzed

...but

- Not all operators have OpenLineage integrations
- Custom Operators require manual OL implementation
- SQL Parsing doesn´t work well on custom cases
- Column level lineage is not ready yet
- Dynamic Tasks + SQL

Lineage Approaches: Dagster



Overview
Runs
Assets
Deployment
Search...
yellow_taxi
Global Asset Lineage
Reload definitions
Jobs
all_assets_job
Asset groups
dbt_seeds
raw
reporting
yellow_taxi_model
Resources
dbt
file_io_manager
s3
Filter
Type an asset subset... (ex: yellow_taxi_raw/yellow_taxi_raw_data_frame+)
Clear query
Materialize all...
yellow_taxi

...but

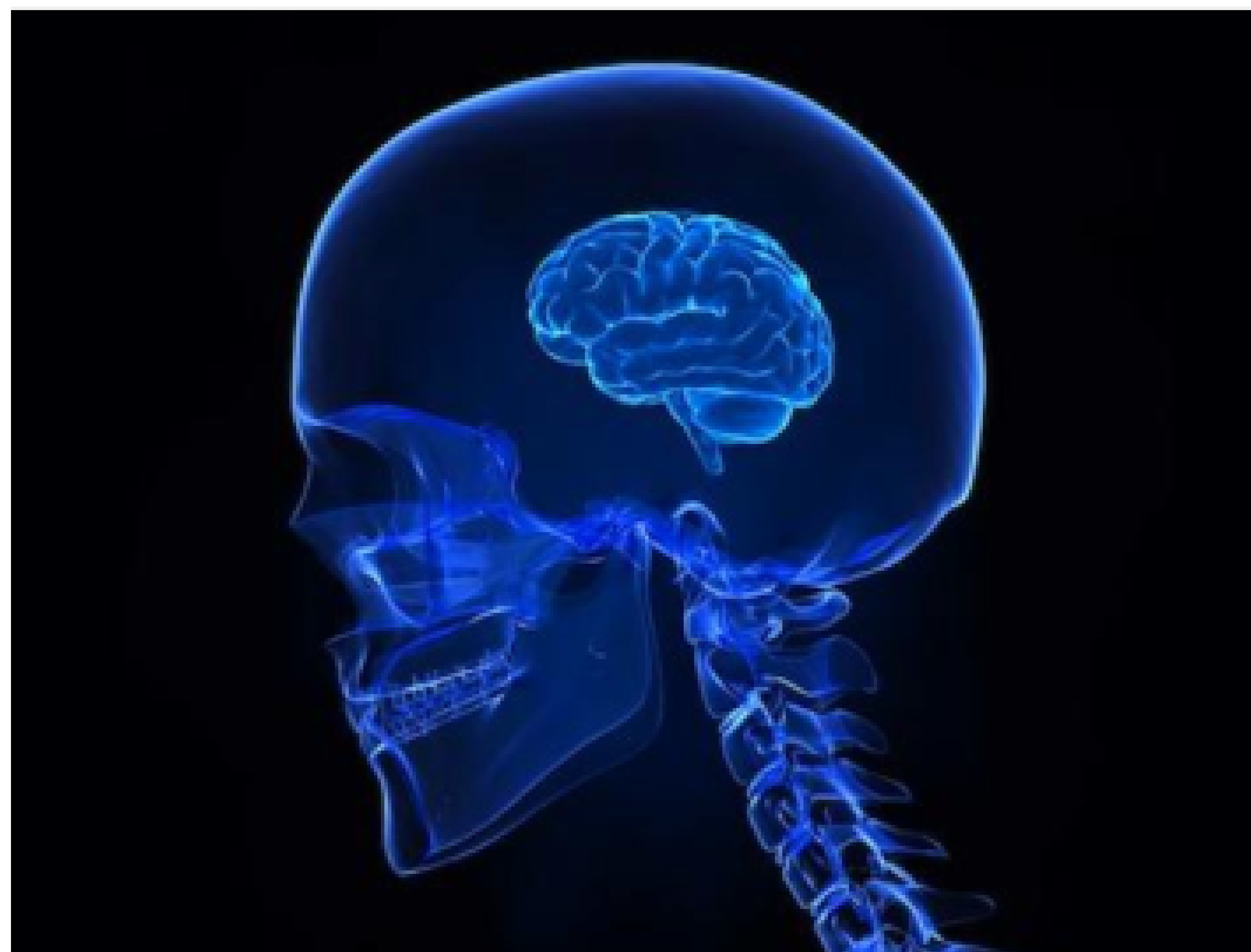
- Relatively small community
- Requires mindset shift
- and...

...extra features = extra money

Column
level lineage
in dagster

Column-level
lineage
in dagster
cloud ($$$)

I'm brave ~~stupid~~ enough to DIY



Let's start simple

```python
from sqlglot import parse_one
sql = """
    CREATE TABLE mart.customer_orders AS
    SELECT
        c.customer_name,
        c.email,
        o.order_id,
        o.order_date,
        o.amount,
        o.price,
        o.amount * o.price AS total_price
    FROM staging.orders o
    JOIN dim.customers c ON o.customer_id = c.customer_id"""

parse_one(sql)
```

```
Create(
  this=Table(
    this=Identifier(this=customer_orders, quoted=False),
    db=Identifier(this=mart, quoted=False)),
  kind=TABLE,
  expression=Select(
    expressions=[
      Column(
        this=Identifier(this=customer_name, quoted=False),
        table=Identifier(this=c, quoted=False)),
      Column(
        this=Identifier(this=email, quoted=False),
        table=Identifier(this=c, quoted=False)),
      Column(
        this=Identifier(this=order_id, quoted=False),
        table=Identifier(this=o, quoted=False)),
      Column(
        this=Identifier(this=order_date, quoted=False),
        table=Identifier(this=o, quoted=False)),
      Column(
        this=Identifier(this=amount, quoted=False),
        table=Identifier(this=o, quoted=False)),
      Column(
        this=Identifier(this=price, quoted=False),
        table=Identifier(this=o, quoted=False)),
      Alias(
        this=Mul(
          this=Column(
            this=Identifier(this=amount, quoted=False),
            table=Identifier(this=o, quoted=False)),
          expression=Column(
            this=Identifier(this=price, quoted=False),
            table=Identifier(this=o, quoted=False))),
        alias=Identifier(this=total_price, quoted=False))],
    from=From(
      this=Table(
        this=Identifier(this=orders, quoted=False),
        db=Identifier(this=staging, quoted=False),
        alias=TableAlias(
          this=Identifier(this=o, quoted=False)))),
    joins=[
      Join(
        this=Table(
          this=Identifier(this=customers, quoted=False),
          db=Identifier(this=dim, quoted=False),
          alias=TableAlias(
            this=Identifier(this=c, quoted=False))),
        on=EQ(
          this=Column(
            this=Identifier(this=customer_id, quoted=False),
            table=Identifier(this=o, quoted=False)),
          expression=Column(
            this=Identifier(this=customer_id, quoted=False),
            table=Identifier(this=c, quoted=False))))]))
```

Let's increase complexity a bit

```
# Example SQL queries
sql_queries = {
    "staging": """
    CREATE TABLE staging.orders AS
    SELECT order_id, customer_id, order_date, amount, price
    FROM raw_data.orders
    """,

    "mart": """
    CREATE TABLE mart.customer_orders AS
    SELECT
        c.customer_name,
        c.email,
        o.order_id,
        o.order_date,
        o.amount,
        o.price,
        o.amount * o.price AS total_price
    FROM staging.orders o
    JOIN dim.customers c ON o.customer_id = c.customer_id
    """
}
```

```
Analyzing staging query:
Source tables: {'raw_data.orders'}
Target tables: {'staging.orders'}

Column lineage:
order_id <- ['order_id']
customer_id <- ['customer_id']
order_date <- ['order_date']
amount <- ['amount']
price <- ['price']

Analyzing mart query:
Source tables: {'dim.customers AS c', 'staging.orders AS o'}
Target tables: {'mart.customer_orders'}

Column lineage:
customer_name <- ['c.customer_name']
email <- ['c.email']
order_id <- ['o.order_id']
order_date <- ['o.order_date']
amount <- ['o.amount']
price <- ['o.price']
total_price <- ['o.amount', 'o.price']
```

more...

```
sql_queries = {
    "staging": """
    CREATE TABLE staging.enriched_orders AS
    WITH daily_rates AS (
        SELECT date, currency, exchange_rate
        FROM raw_data.exchange_rates
        WHERE currency = 'USD'
    ),
    order_metrics AS (
        SELECT
            o.order_id,
            o.customer_id,
            o.order_date,
            o.amount * COALESCE(r.exchange_rate, 1.0) as usd_amount,
            COUNT(*) OVER (PARTITION BY o.customer_id) as customer_order_count
        FROM raw_data.orders o
        LEFT JOIN daily_rates r ON o.order_date = r.date
    )
    SELECT
        om.*,
        c.customer_segment,
        c.country,
        CASE
            WHEN usd_amount > 1000 AND customer_order_count > 5 THEN 'VIP'
            WHEN usd_amount > 500 THEN 'Premium'
            ELSE 'Standard'
        END as customer_tier
    FROM order_metrics om
    JOIN dim.customers c ON om.customer_id = c.customer_id
    """,

    "mart": """
    CREATE TABLE mart.customer_analytics AS
    SELECT
        c.customer_segment,
        c.country,
        DATE_TRUNC('month', o.order_date) as month,
        COUNT(DISTINCT o.customer_id) as unique_customers,
        COUNT(*) as total_orders,
        SUM(o.usd_amount) as total_revenue,
        AVG(CASE WHEN o.customer_tier = 'VIP' THEN o.usd_amount ELSE 0 END) as avg_vip_order_value
    FROM staging.enriched_orders o
    JOIN dim.customers c ON o.customer_id = c.customer_id
    GROUP BY 1, 2, 3
    HAVING COUNT(*) > 10
    """
}
```

```
Analyzing staging query:
Source tables: {'raw_data.orders AS o', 'order_metrics AS om', 'dim.customers AS c', 'raw_data.exchange_rates', 'daily_rates AS r'}
Target tables: {'staging.enriched_orders'}

Column lineage:

Table: staging.enriched_orders

Target column: *
Source columns: [('om', '*')]

Target column: customer_segment
Source columns: [('c', 'customer_segment')]

Target column: country
Source columns: [('c', 'country')]

Target column: customer_tier
Source columns: [(None, 'usd_amount'), (None, 'usd_amount'), (None, 'customer_order_count')]

Analyzing mart query:
Source tables: {'staging.enriched_orders AS o', 'dim.customers AS c'}
Target tables: {'mart.customer_analytics'}

Column lineage:

Table: mart.customer_analytics

Target column: customer_segment
Source columns: [('c', 'customer_segment')]

Target column: country
Source columns: [('c', 'country')]

Target column: month
Source columns: [('o', 'order_date')]

Target column: unique_customers
Source columns: [('o', 'customer_id')]

Target column: total_orders
Source columns: []

Target column: total_revenue
Source columns: [('o', 'usd_amount')]

Target column: avg_vip_order_value
Source columns: [('o', 'usd_amount'), ('o', 'customer_tier')]
```

MORE

```python
sql_queries = {
    "staging": """
    CREATE TABLE staging.enriched_orders AS
    WITH daily_rates AS (
        SELECT date, currency, exchange_rate
        FROM raw_data.exchange_rates
        WHERE currency = 'USD'
    ),
    order_metrics AS (
        SELECT
            o.order_id,
            o.customer_id,
            o.order_date,
            o.amount * COALESCE(r.exchange_rate, 1.0) as usd_amount,
            COUNT(*) OVER (PARTITION BY o.customer_id) as customer_order_count
        FROM raw_data.orders o
        LEFT JOIN daily_rates r ON o.order_date = r.date
    )
    SELECT
        om.*,
        c.customer_segment,
        c.country,
        CASE
            WHEN usd_amount > 1000 AND customer_order_count > 5 THEN 'VIP'
            WHEN usd_amount > 500 THEN 'Premium'
            ELSE 'Standard'
        END as customer_tier
    FROM order_metrics om
    JOIN dim.customers c ON om.customer_id = c.customer_id
    """,

    "mart": """
    CREATE TABLE mart.customer_analytics AS
    SELECT
        c.customer_segment,
        c.country,
        DATE_TRUNC('month', o.order_date) as month,
        COUNT(DISTINCT o.customer_id) as unique_customers,
        COUNT(*) as total_orders,
        SUM(o.usd_amount) as total_revenue,
        AVG(CASE WHEN o.customer_tier = 'VIP' THEN o.usd_amount ELSE 0 END) as avg_vip_order_value
    FROM staging.enriched_orders o
    JOIN dim.customers c ON o.customer_id = c.customer_id
    GROUP BY 1, 2, 3
    HAVING COUNT(*) > 10
    """
}
```

```
Analyzing staging query:
Source tables: {'"dim"."customers" AS "c"', '"raw_data"."exchange_rates" AS "exchange_rates"', '"order_metrics" AS "om"', '"raw_data"."orders" AS "o"', '"daily_rates" AS "r"'}
Target tables: {'"staging"."enriched_orders"'}

Column lineage:

Table: "staging"."enriched_orders"

Target column: order_id
Source columns: [('om', 'order_id')]

Target column: customer_id
Source columns: [('om', 'customer_id')]

Target column: order_date
Source columns: [('om', 'order_date')]

Target column: usd_amount
Source columns: [('om', 'usd_amount')]

Target column: customer_order_count
Source columns: [('om', 'customer_order_count')]

Target column: customer_segment
Source columns: [('c', 'customer_segment')]

Target column: country
Source columns: [('c', 'country')]

Target column: customer_tier
Source columns: [('om', 'usd_amount'), ('om', 'usd_amount'), ('om', 'customer_order_count')]

Analyzing mart query:
Source tables: {'"staging"."enriched_orders" AS "o"', '"dim"."customers" AS "c"'}
Target tables: {'"mart"."customer_analytics"'}

Column lineage:

Table: "mart"."customer_analytics"

Target column: customer_segment
Source columns: [('c', 'customer_segment')]

Target column: country
Source columns: [('c', 'country')]

Target column: month
Source columns: [('o', 'order_date')]

Target column: unique_customers
Source columns: [('o', 'customer_id')]

Target column: total_orders
Source columns: []

Target column: total_revenue
Source columns: [('o', 'usd_amount')]

Target column: avg_vip_order_value
Source columns: [('o', 'usd_amount'), ('o', 'customer_tier')]
```

People who use SELECT *
in production

```python
sql = "SELECT orders.* FROM orders JOIN customers ON orders.customer_id = customers.id"
print(parse_one(sql).sql(pretty=True))
```

```
SELECT
  orders.*
FROM orders
JOIN customers
  ON orders.customer_id = customers.id
```

```python
schema = {"orders": {"id": "INT", "order_date": "DATE", "amount": "FLOAT", "price": "FLOAT", "customer_id": "int"},
          "customers": {"id": "INT", "name": "varchar", "email": "varchar"},
          "exchange_rates": {"date": "DATE", "currency": "STRING", "exchange_rate": "FLOAT"}}
expression = parse_one(sql)
ast = qualify(expression, schema=schema, expand_stars=True).sql(pretty=True)
print(ast)
```

```
SELECT
  "orders"."id" AS "id",
  "orders"."order_date" AS "order_date",
  "orders"."amount" AS "amount",
  "orders"."price" AS "price",
  "orders"."customer_id" AS "customer_id"
FROM "orders" AS "orders"
JOIN "customers" AS "customers"
  ON "orders"."customer_id" = "customers"."id"
```

What if I use dbt?

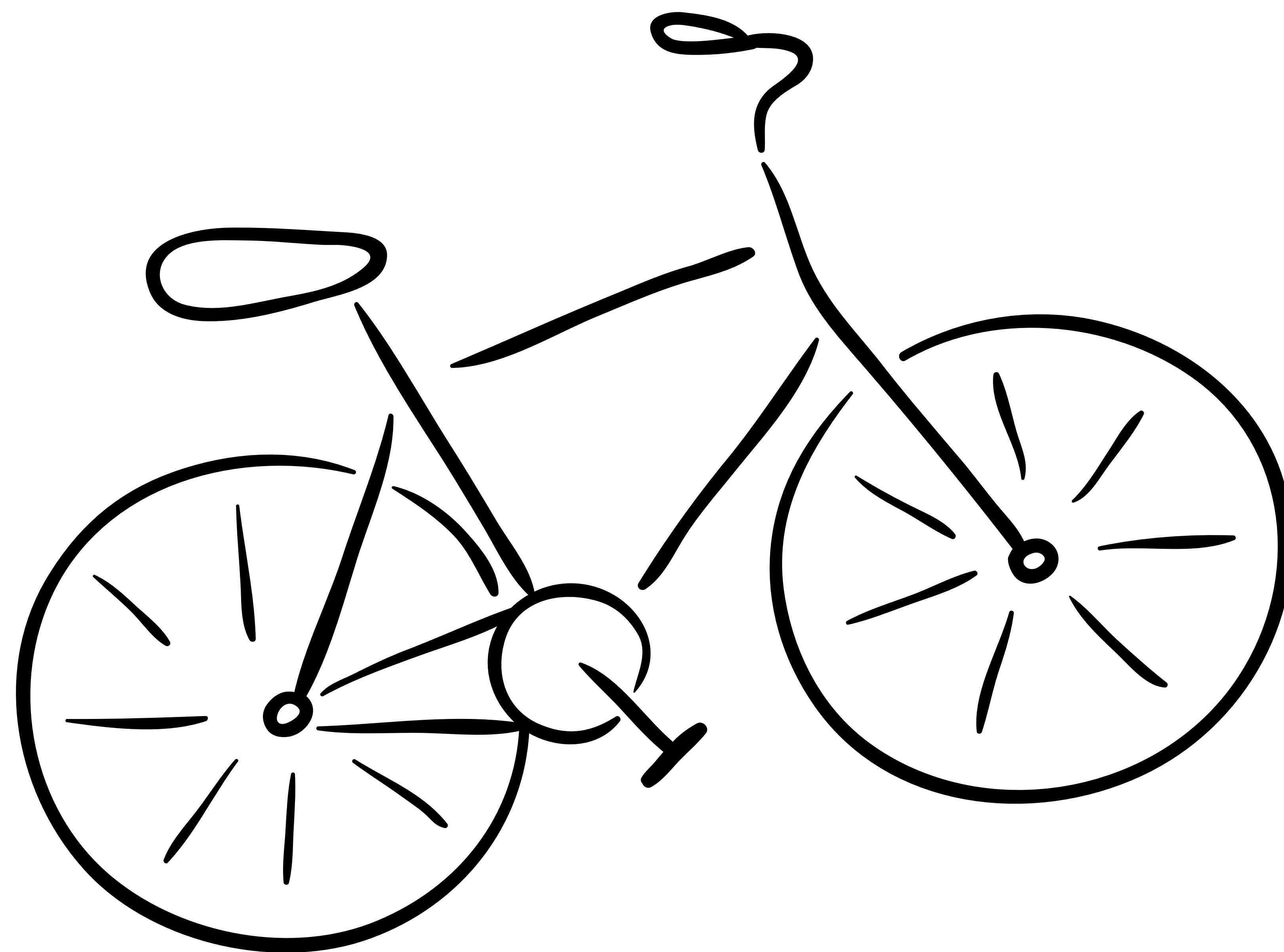
Compile your files

Run parsing over them

What´s next?

Process Flow: Extract SQL → Parse → Push to Database → Add API on top

Congratulations!



Q&A