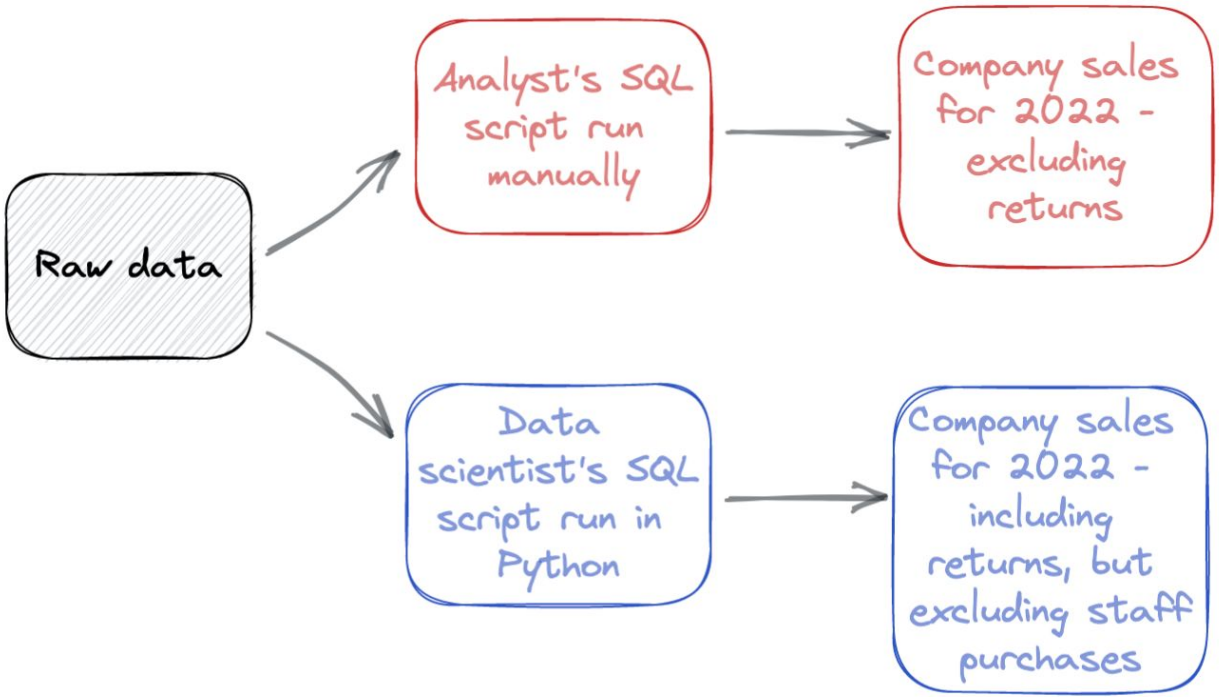
A brief history of data & dbt

How most data pipelines used to work

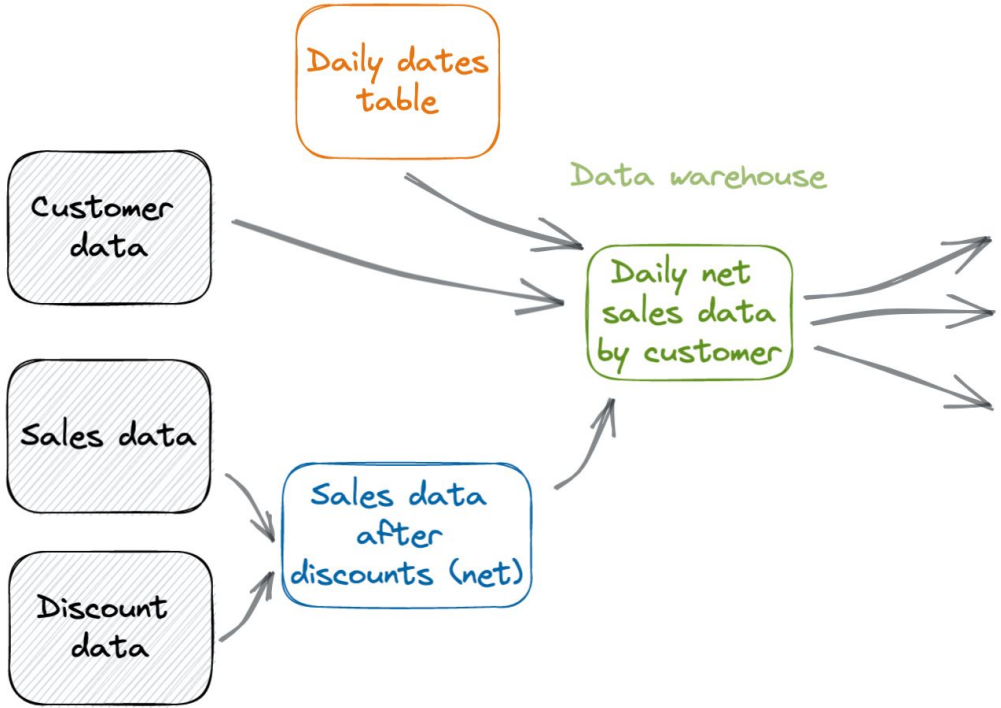
Lots of manual effort, lots of overlap, little consistency



Same metric, different definitions, run manually and not reusable

Data warehousing

Centralised data models



The problems

- The people creating the data models were very separate from the people using them
- You have to manually specify what order to run your SQL scripts in
- Testing for data quality is a manual process (1 SQL file per check)
- Documenting data models is typically done in a different tool
- SQL has limitations compared to other tools like Python

The main benefits of using dbt

1. Inferring dependencies

2. Documentation & testing

3. Python-like functionality

1. Inferring dependencies

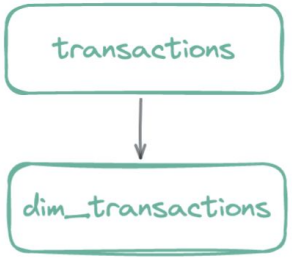
dbt SQL file
(dim_transactions.sql)

Compiled dbt SQL file (easy to switch database/schema!)

Run dbt SQL file

```sql
SELECT
    date,
    transaction_id,
    amount

FROM {{ ref('transactions')}}
```

Infers dependencies (DAG)




```sql
SELECT
    date,
    transaction_id,
    amount

FROM `my_database.my_schema.transactions`
```

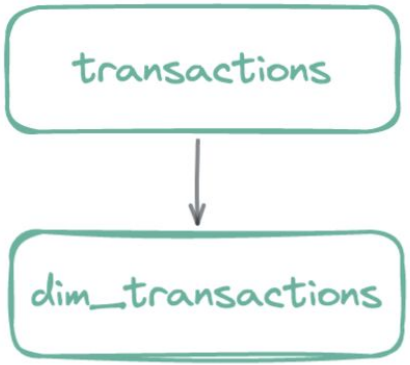
```sql
CREATE OR REPLACE TABLE `my_database.my_schema.dim_transactions` AS

SELECT
    date,
    transaction_id,
    amount

FROM `my_database.my_schema.transactions`
```

1a. What's a DAG?

- **D**irected **A**cyclic **G**raph
  - **Directed**: arrows pointing from one node to another
  - **Acyclic**: no circular dependencies
  - **Graph**: points (nodes) and arrows (edges)

- Fancy way of saying a set of data models (SQL files) with arrows that show what order to run them, without any circular dependencies

- "What order do I need to run my SQL files?"



2. Document & test SQL data models

Using YAML (.yml) files

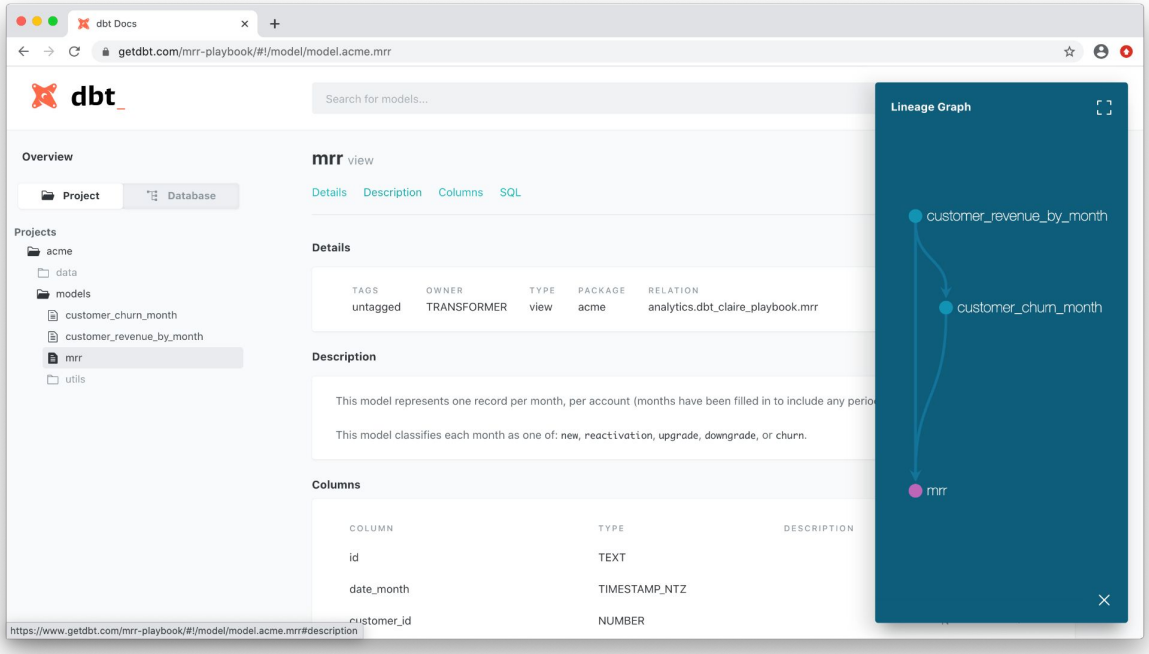
```yaml
version: 2

models:
  - name: dim_transactions
    description: One row per transaction_id, with an amount column
    columns:
      - name: date
        description: "Transaction date"
        tests:
          - not_null

      - name: transaction_id
        description: "Unique identifier for this transaction"
        tests:
          - not_null
          - unique
          - relationships:
              to: ref('transactions')
              field: transaction_id

      - name: amount
        description: "Transaction amount"
        tests:
          - not_null
          - dbt_utils.expression_is_true:
              expression: "> 0"
```

You can have:

- One .yml file per model (e.g. dim_transactions.yml),
- Or have multiple models in one file (e.g. core_models.yml)

2. Document & test SQL data models

Using the YAML files to generate (and host) documents



- Easy to share
- Shows model lineage

Source: https://docs.getdbt.com/docs/collaborate/documentation

3. Python like functionality - variables

Using `set` to create a list which we then loop through

Compiled SQL

```
{% set categories = ["electronics", "clothes"] %}

SELECT
    date,
    {% for category in categories %}
    SUM(IF(item_category = '{{ category }}', amount, 0)) AS amount_sold_{{ category }},
    {% endfor %}
    SUM(amount) AS amount_sold

FROM {{ ref('transactions') }}
GROUP BY 1
```

```
SELECT
    date,
    SUM(IF(item_category = 'electonics', amount, 0)) AS amount_sold_electronics,
    SUM(IF(item_category = 'clothes', amount, 0)) AS amount_sold_clothes,
    SUM(amount) AS amount_sold

FROM `my_database.my_schema.transactions`
GROUP BY 1
```

3. Python like functionality - macros

Writing a macro, stored in a separate SQL file

```
{% macro divide_by_100(column_name, precision=2) %}
    ROUND({{ column_name }} / 100), {{ precision }}
{% endmacro %}
```

Applying it in a SQL file

```
SELECT
    {{ divide_by_100(amount_cents, precision=2) }} AS amount_usd

FROM {{ ref('transactions') }}
```

The compiled SQL

```
SELECT
    ROUND(amount_cents / 100, 2) AS amount_usd

FROM `my_database.my_schema.transactions`
```

3. Python like functionality - tests

Writing a column level test, stored in a separate SQL file

Applying it to a column in a YAML file

```
{% test all_values_equal_to(model, column_name, equal_to) %}

-- Returns any records where the column doesn't match a given value
SELECT
*
FROM {{ model }}
WHERE {{ column_name }} != {{ equal_to }}

{% endtest %}
```

```
- name: store_id
  description: "Unique identifier for the store"
  tests:
    - test_all_values_equal_to:
        equal_to: 1234
```

The problems (recap)

- The people creating the data models were very separate from the people using them
- You have to manually specify what order to run your SQL scripts in
- Testing for data quality is a manual process (1 SQL file per check)
- Documenting data models is typically done in a different tool
- SQL has limitations compared to other tools like Python

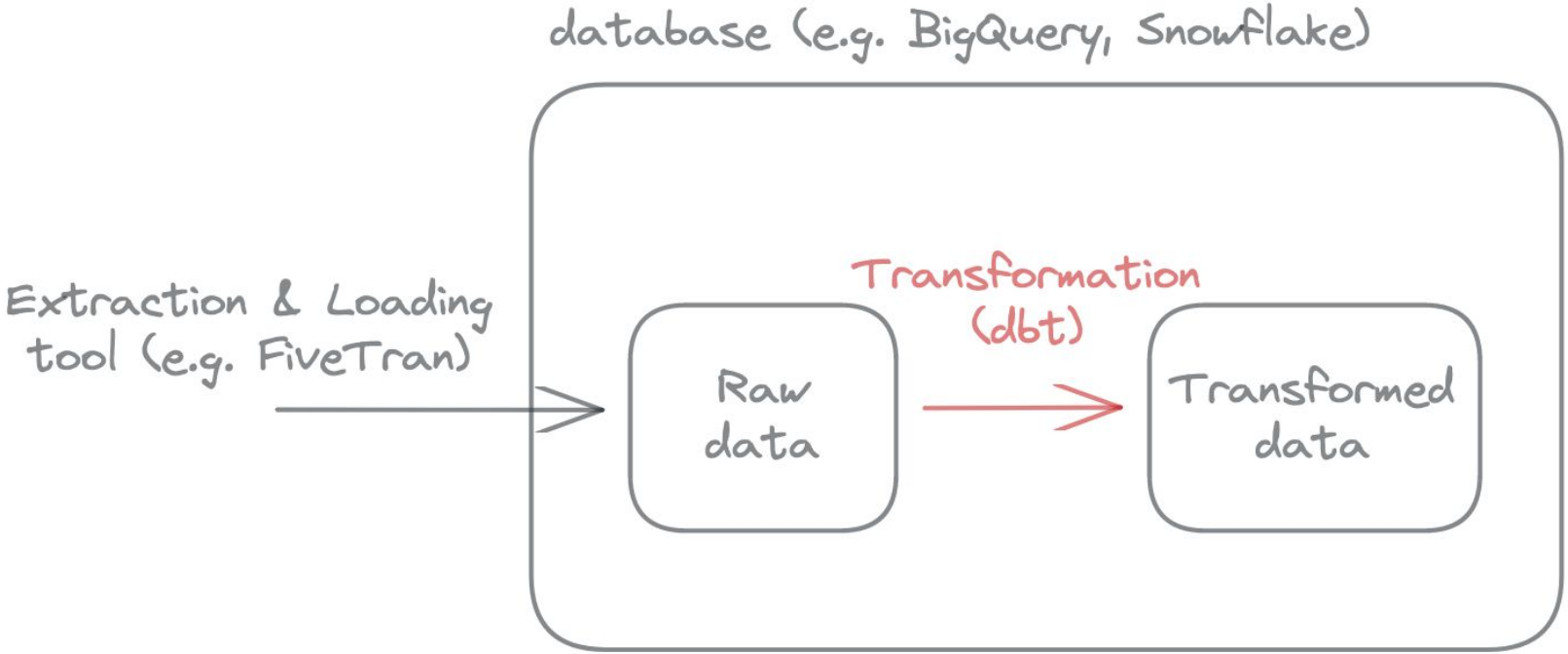
How dbt solved common data problems

- Lowered the technical barrier for creating data pipelines & models
- Automatically infers what order to run SQL models
- Documenting and testing data models is very simple
- Jinja allows for functionality not found in native SQL

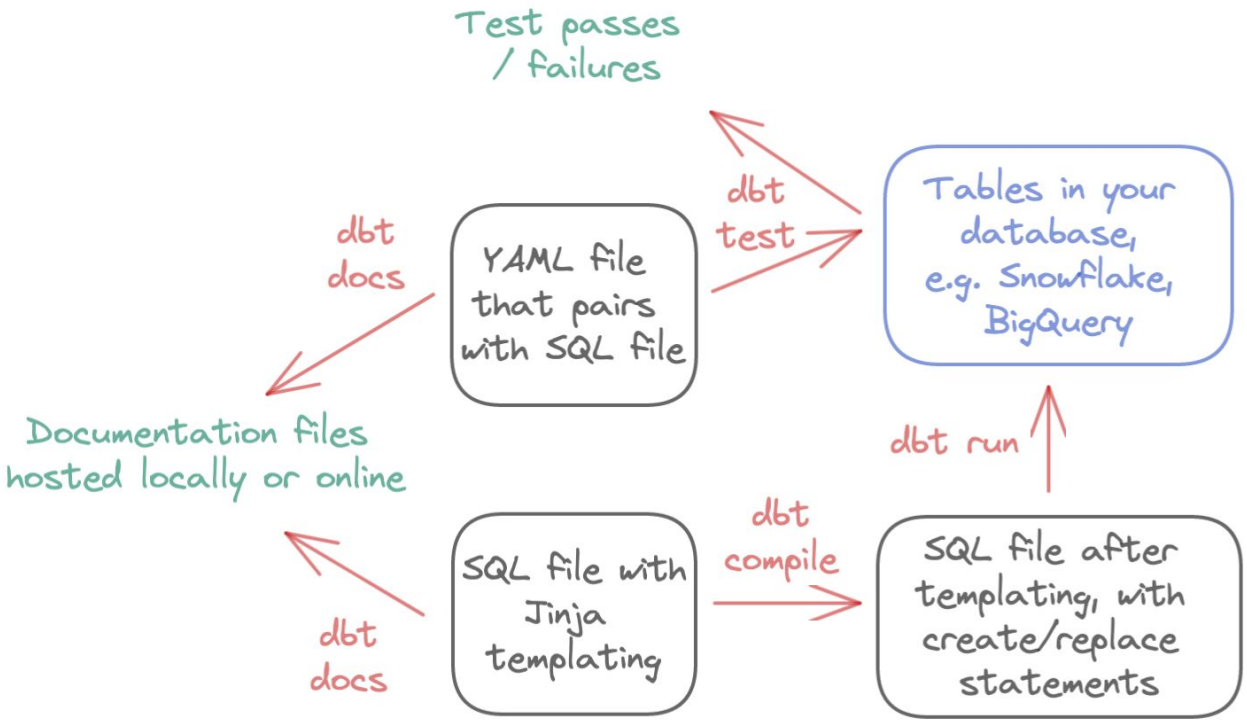
How dbt fits in

Dbt is the “T” in “ELT” (Extract, Load, Transform)

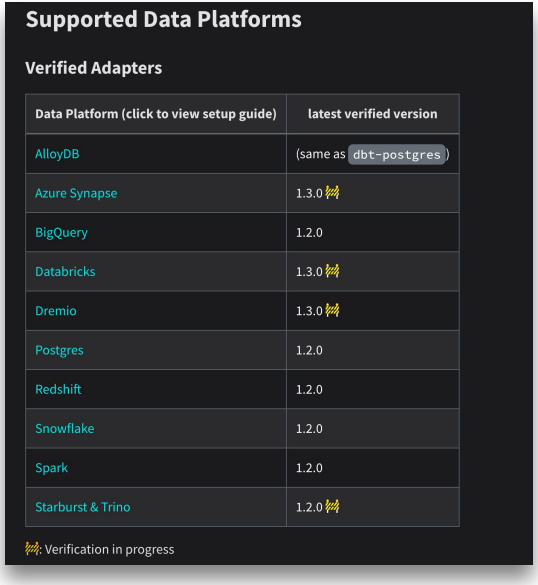
It transforms data within your database



dbt takes care of transforming, testing, and documenting your data



What databases can you use dbt with?



Supported Data Platforms

Verified Adapters

Data Platform (click to view setup guide)	latest verified version
AlloyDB	(same as `dbt-postgres`)
Azure Synapse	1.3.0 🚧
BigQuery	1.2.0
Databricks	1.3.0 🚧
Dremio	1.3.0 🚧
Postgres	1.2.0
Redshift	1.2.0
Snowflake	1.2.0
Spark	1.2.0
Starburst & Trino	1.2.0 🚧

🚧: Verification in progress

Adapter - thing that plugs dbt into a specific type of database, e.g. dbt-bigquery

Source: https://docs.getdbt.com/docs/supported-data-platforms