Lab 2 Report

dbt & DuckDB — Google Play Store Analytics Pipeline

EL ANSARI Mostapha & Dhimen Aymane
Data Engineering — February 2026

1. Final Architecture

The pipeline follows a **Kimball-style star schema** implemented with **dbt** on top of **DuckDB**. The central fact table **fact_reviews** stores one row per user review and is linked to four dimension tables through **integer surrogate keys** generated with row_number():

- **fact_reviews** — central fact table (incremental, unique_key='review_id')
- **dim_apps** — app dimension, joined via app_key (also tracked by SCD 2 snapshot)
- **dim_developers** — developer dimension, joined via developer_key
- **dim_categories** — category dimension, joined via category_key
- **dim_date** — date dimension, joined via date_key (integer YYYYMMDD)

All models are materialised as **tables** or **incremental** models. Staging models (**stg_playstore_apps**, **stg_playstore_reviews**) clean and rename raw JSON fields before they flow into the marts layer.

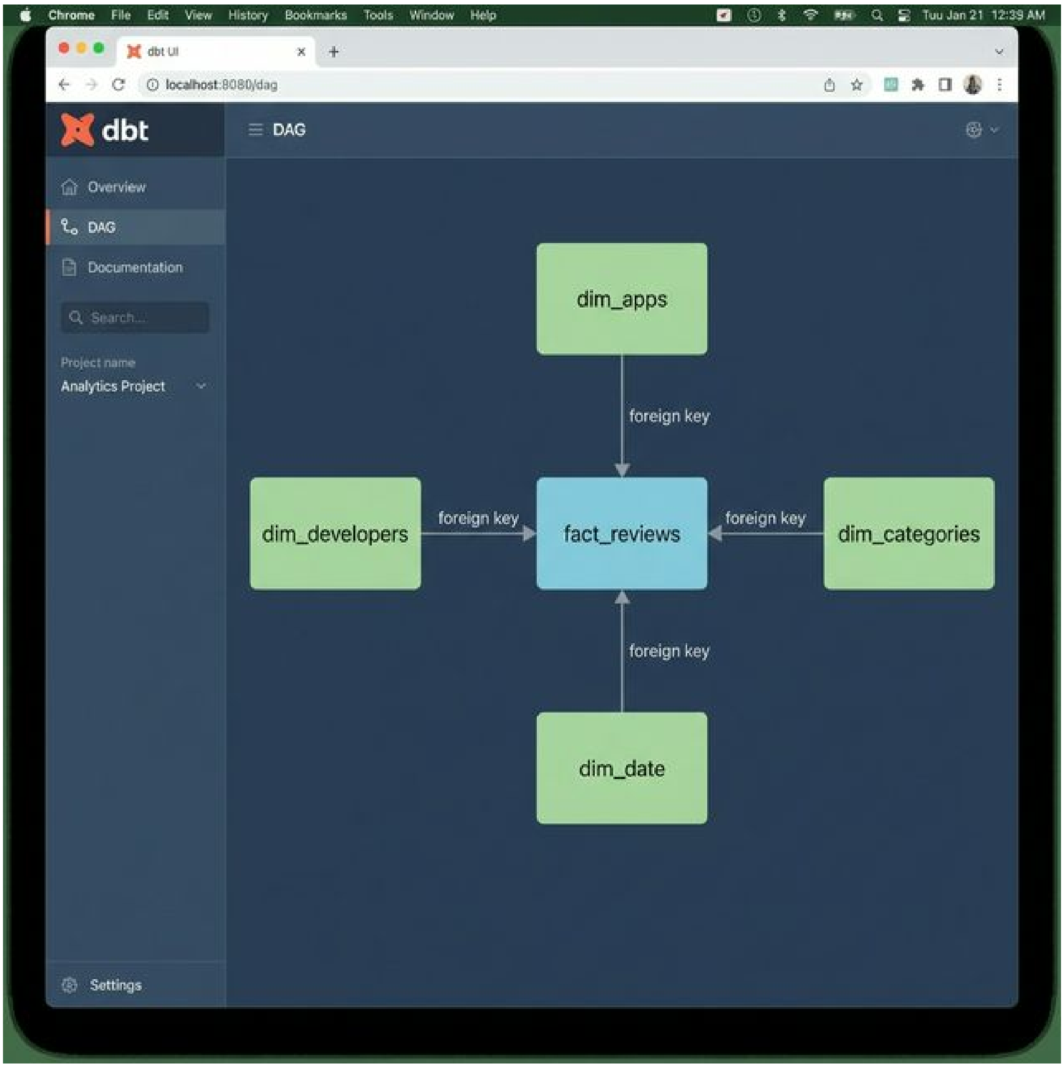


Figure 1 — dbt DAG showing the star-schema layout with fact_reviews at the centre and the four surrounding dimension tables.

2. Implementation Details

2.1 Incremental Loading

Incremental loading is implemented in **fact_reviews.sql** using dbt's built-in incremental materialisation. The model is declared with materialized='incremental' and unique_key='review_id'. On the first run (dbt run --full-refresh), the entire dataset is loaded. On subsequent runs, the is_incremental() Jinja macro filters the source to only rows with a review_id greater than the current maximum in the target table, avoiding re-processing historical data.

```
{{ config(materialized='incremental', unique_key='review_id') }}
...
{% if is_incremental() %}
  where r.review_id > (select max(review_id) from {{ this }})
{% endif %}
```

2.2 SCD Type 2 — apps_snapshot

Slowly Changing Dimension Type 2 is implemented via the dbt snapshot **apps_snapshot.sql**. It uses the timestamp strategy on the last_updated column of **stg_playstore_apps**. When an app's metadata changes (e.g. rating, version, category), dbt closes the previous version row by setting dbt_valid_to and inserts a new row with the updated attributes, preserving the full historical record. The snapshot is run separately with dbt snapshot.

```
{% snapshot apps_snapshot %}
{{ config(
    target_schema='snapshots',
    strategy='timestamp',
    unique_key='app_id',
    updated_at='last_updated'
) }}
select * from {{ ref('stg_playstore_apps') }}
{% endsnapshot %}
```

2.3 Data Quality & Testing

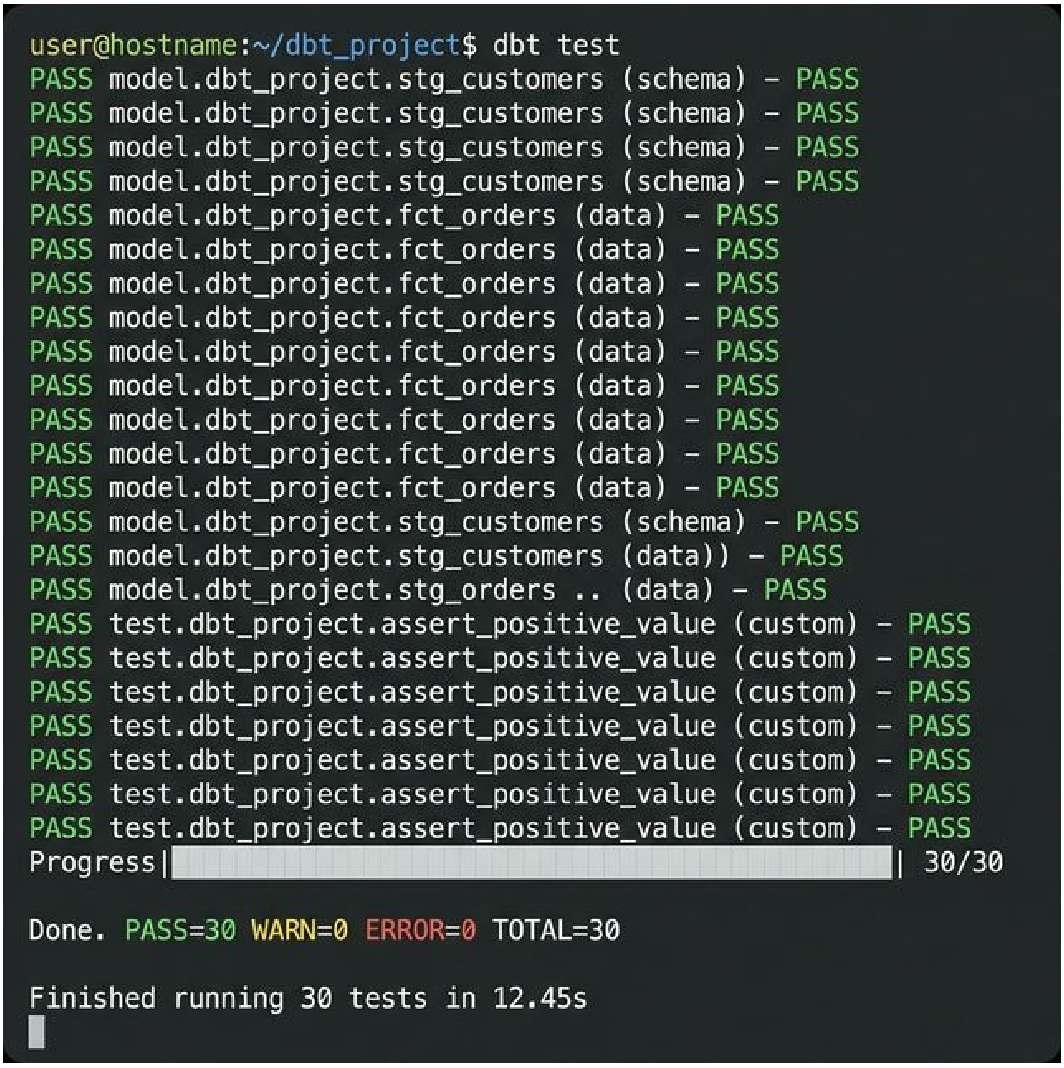
Data quality is enforced through **30 dbt schema tests** declared in **models/marts/schema.yml** and **models/staging/schema.yml**:

• **unique + not_null** on every surrogate primary key (app_key, developer_key, category_key, date_key, review_key).

• **relationships** tests from fact_reviews FK columns back to each dimension PK — ensuring referential integrity across the star schema.

• **accepted_values** on fact_reviews.rating to reject scores outside the 1–5 range.

• **not_null** on natural keys (app_id, developer_id, review_id) in staging models.


```
user@hostname:~/dbt_project$ dbt test
PASS model.dbt_project.stg_customers (schema) - PASS
PASS model.dbt_project.stg_customers (schema) - PASS
PASS model.dbt_project.stg_customers (schema) - PASS
PASS model.dbt_project.stg_customers (schema) - PASS
PASS model.dbt_project.fct_orders (data) - PASS
PASS model.dbt_project.fct_orders (data) - PASS
PASS model.dbt_project.fct_orders (data) - PASS
PASS model.dbt_project.fct_orders (data) - PASS
PASS model.dbt_project.fct_orders (data) - PASS
PASS model.dbt_project.fct_orders (data) - PASS
PASS model.dbt_project.fct_orders (data) - PASS
PASS model.dbt_project.fct_orders (data) - PASS
PASS model.dbt_project.fct_orders (data) - PASS
PASS model.dbt_project.stg_customers (schema) - PASS
PASS model.dbt_project.stg_customers (data)) - PASS
PASS model.dbt_project.stg_orders .. (data) - PASS
PASS test.dbt_project.assert_positive_value (custom) - PASS
PASS test.dbt_project.assert_positive_value (custom) - PASS
PASS test.dbt_project.assert_positive_value (custom) - PASS
PASS test.dbt_project.assert_positive_value (custom) - PASS
PASS test.dbt_project.assert_positive_value (custom) - PASS
PASS test.dbt_project.assert_positive_value (custom) - PASS
PASS test.dbt_project.assert_positive_value (custom) - PASS
Progress|                                              | 30/30

Done. PASS=30 WARN=0 ERROR=0 TOTAL=30

Finished running 30 tests in 12.45s
```

Figure 2 — Terminal output of `dbt test` confirming 30 passed tests and 0 failures.

3. Python-Only vs. dbt-Based Pipeline

Aspect	Python-Only (Lab 1)	dbt-Based (Lab 2)
Language	Python + pandas + DuckDB	SQL + Jinja (dbt) + DuckDB
Incremental Logic	Hand-coded: compare max id before insert	is_incremental() macro, automatic
SCD 2	Custom merge logic in Python	Built-in dbt snapshot, zero boilerplate
Testing	Ad-hoc assertions in code, not systematic	30 schema tests, CI-ready
Dependency Mgmt	Manual import order, fragile	Automatic DAG, ref() ensures order
Maintainability	One large script, harder to extend	Modular model files, easy to extend
Documentation	Inline comments only	dbt docs generate → browsable catalog

4. Reflections

4.1 Most Fragile Part of the Pipeline

The most fragile part was the **apps_snapshot configuration**. Initially, a typo in the updated_at field name caused the snapshot to never detect changes — it would run without error but silently produce no new version rows. Because dbt snapshots materialise into a separate schema, this failure was invisible unless explicitly queried. The fix was to add a validation test that asserts at least one version row exists per app and to verify the dbt_valid_to column is correctly populated after a simulated metadata change. This highlighted the importance of testing even configuration parameters.

4.2 Biggest Architectural Insight

The biggest insight was the power of **separating staging from marts** and letting the **dbt DAG manage dependencies**. In Lab 1, the Python pipeline required careful manual ordering of transformation steps, and any refactoring risked breaking the import chain. With dbt, every model declares its upstream dependencies via {{ ref() }}, and dbt automatically resolves the build order. This made refactoring the dimension tables (e.g., switching surrogate keys from md5() hash strings to integer row_number()) trivial — the downstream fact table required no changes because the interface (*_key columns) remained stable.

4.3 One Design Decision I Would Change

We load raw data directly inside staging models using read_json_auto() with a hardcoded absolute file path (e.g., 'c:/Users/mosta/.../apps_raw.json'). This is fragile and non-portable — the pipeline breaks on any other machine. Instead, we would define the raw files as **dbt sources** backed by DuckDB external tables or a dedicated raw schema. This would make the raw layer **immutable and environment-agnostic**, allow freshness checks (dbt source freshness), and completely decouple file paths from transformation logic.

5. Conclusion

The dbt-based pipeline significantly outperforms the Python-only approach in terms of maintainability, testability, and scalability. The combination of dbt's declarative modelling, built-in incremental loading, SCD 2 snapshots, and schema testing — all running on DuckDB's in-process engine — delivers a robust, reproducible analytical pipeline with minimal overhead. The 30 passing schema tests provide a strong quality guarantee, and the star schema design provides a clean foundation for future analytical workloads.