Google Play Store
Data Pipeline Report

Lab 2 — Data Pipeline with dbt & DuckDB

January 2026

Imad Absri

Omar Bellmir

Centrale Casablanca

Department of Data Engineering

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Objectives

The Google Play Store hosts millions of applications and generates an enormous volume of user feedback every day. Turning this raw stream of reviews into structured, queryable, and visually explorable analytics is precisely the challenge this second data engineering lab was designed to address.

Where the first lab built the pipeline end-to-end in ad-hoc Python—mixing ingestion, transformation, and serving logic in a single script—this lab elevates the architecture by introducing industry-standard tooling. Transformation logic moves entirely into **dbt Core**, all data is stored and queried inside **DuckDB**, and the final analytics-ready tables are consumed by **Power BI**. The result is a pipeline that is modular, testable, and repeatable in a way that raw Python scripting cannot match.

The lab guidelines define six concrete objectives that drove every design decision we made:

Lab Objectives

1. Install and configure the development environment (DuckDB, dbt-core, dbt-duckdb adapter).
2. Explore dimensional data modelling for analytics (star/snowflake schema, Kimball methodology).
3. Structure a data pipeline using dbt model layers — staging flowing into marts.
4. Separate raw data, transformation logic, and serving tables with clear physical boundaries.
5. Apply data quality tests using dbt's native testing framework.
6. Prepare a stable serving layer consumable by BI dashboards (Power BI).

Each chapter of this report maps directly to one or more of these objectives and documents both our implementation decisions and the concrete outputs they produced. A compliance matrix in Chapter 10 cross-references every guideline requirement against its evidence in the codebase.

Chapter 2

Architecture and Technology Stack

2.1 High-Level Architecture

The pipeline separates three distinct phases, each handled by a dedicated component, with clean handoff points between them.

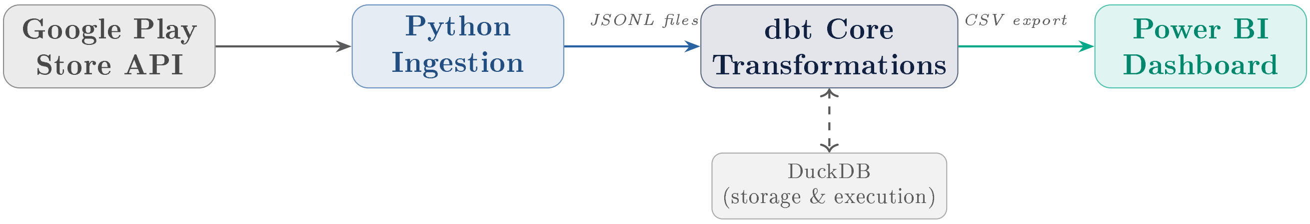


Figure 2.1: *End-to-end pipeline architecture showing the three phases and their data handoffs*

Phase 1 — Ingestion. The Python script `ingest.py` calls the `google-play-scraper` library against a curated list of 20 popular applications. For each app it collects app metadata (title, developer, category, average score, total ratings) and user reviews filtered to the last 100 days, capped at 5 000 reviews per application. Both outputs are saved as JSONL files in `data/raw/` and are never modified again—serving as an immutable raw landing zone, exactly as the guidelines specify.

Phase 2 — Transformation. dbt orchestrates all transformation logic, and DuckDB executes it. DuckDB reads the JSONL files directly without any intermediate loading step, then progressively cleans and materialises the data through staging views and mart tables forming a complete snowflake schema.

Phase 3 — Serving. The script `export_to_powerbi.py` connects to the DuckDB database, queries each mart table, and writes CSV files to `powerbi_export/`. These files are then imported into Power BI where relationships and visualisations are configured interactively.

2.2 Technology Stack

Table 2.1: *Technology stack and pinned versions used in this project*

Component	Version	Role	Notes
Python	3.13	Ingestion & export scripting	Reused from Lab 1
google-play-scraper	latest	Play Store API client	Handles pagination
DuckDB	1.4.4	Analytical database engine	Reads JSONL natively
dbt-core	1.11.6	Transformation orchestration	SQL-only models + YAML tests
dbt-duckdb	1.10.1	DuckDB adapter for dbt	Bridges dbt and DuckDB
Power BI Desktop	latest	BI dashboard	Imports CSV, defines relationships

The choice of DuckDB as the execution engine is deliberate. It is an in-process OLAP database that requires no server setup, reads JSONL files natively, and achieves excellent analytical query performance on a standard laptop. Combined with dbt, it creates a local stack that behaves exactly like a cloud data warehouse—making it ideal for a reproducible lab environment.

Chapter 3

Dimensional Data Modelling

3.1 Applying the Kimball Methodology

The guidelines explicitly require applying the Kimball four-step process before writing any SQL. We worked through each step against the Google Play Store dataset and document the outcomes below.

3.1.1 Step 1 — Business Process

The business process under analysis is *user review submission*. Every time a user publishes a review on the Google Play Store, a discrete and measurable event occurs, carrying quantitative feedback (score, thumbs-up count) alongside rich contextual information: which application, which category, which developer, and when.

3.1.2 Step 2 — Grain

One row in the fact table represents **a single user review posted for one specific application at one specific point in time**. This is the finest grain available in the dataset and allows every conceivable analytical aggregation without loss of information.

3.1.3 Step 3 — Dimensions

Table 3.1: *Dimension identification: business questions answered and key attributes*

Dimension	Business question	Key attributes
`dim_apps`	Which app was reviewed?	app_id, app_name, price, avg_score, total_ratings
`dim_categories`	Which category does the app belong to?	category_id, category_name
`dim_developers`	Who built the app?	developer_id, developer_name
`dim_date`	When was the review posted?	date_day, year, quarter, month, week_of_year, day_name, is_weekend

3.1.4 Step 4 — Facts (Measures)

Two quantitative measures are stored in the fact table. `review_score` is an integer from 1 to 5, well-suited for averaging and distribution analysis. `thumbs_up_count` is a non-negative integer, meaningful for summing to measure community engagement with specific reviews. Both are captured directly from the raw review payload with no derivation required.

3.1.5 Bus Matrix

Table 3.2: *Kimball Bus Matrix: dimension availability per business process*

Business Process	dim_apps	dim_categories	dim_developers	dim_date
User review submission	✓	✓	✓	✓

3.2 Schema Design: Snowflake Variation

The schema we implemented is a **snowflake schema**. The central `fact_reviews` table joins to `dim_apps` via a surrogate key, and `dim_apps` in turn references `dim_categories` and `dim_developers` as separately normalised tables rather than embedding their attributes directly. This adds a controlled level of normalisation that keeps each dimension table focused on a single concept, while Power BI's relationship engine traverses the chain of foreign keys automatically.

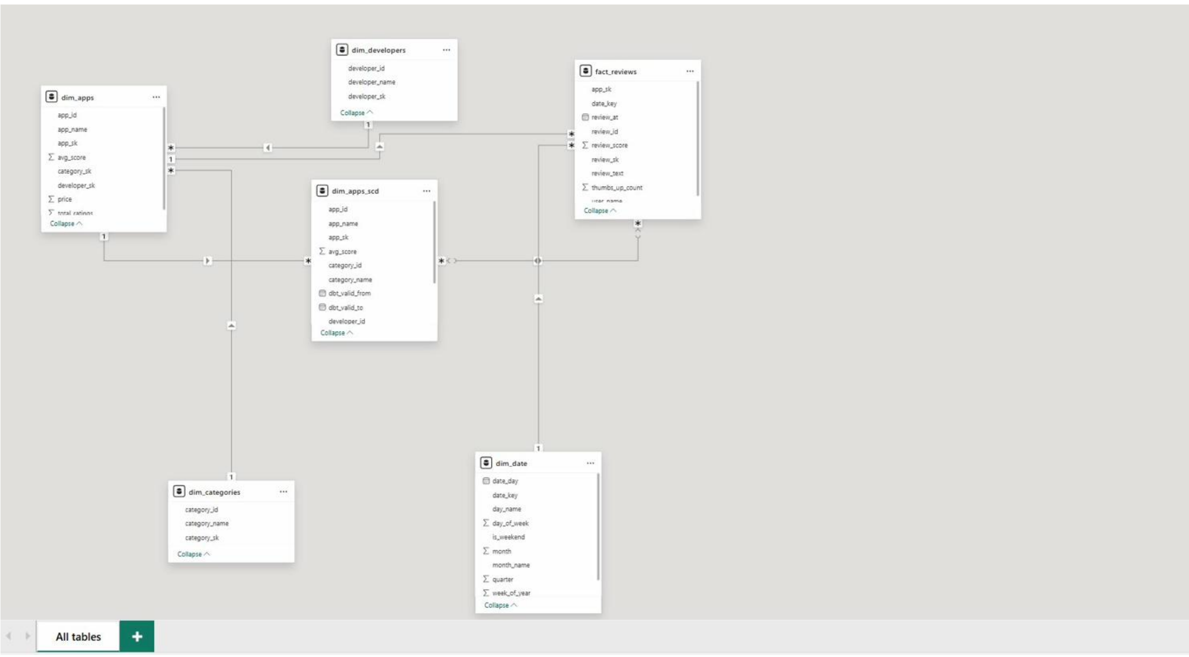


Figure 3.1: *Power BI relationship diagram showing the complete snowflake schema with all six tables*

Figure 3.1 shows the six tables in Power BI's model view. The central `fact_reviews` table holds foreign keys to `dim_apps` (via `app_sk`) and `dim_date` (via `date_key`). The `dim_apps` dimension connects outward to `dim_categories` and `dim_developers`. The additional `dim_apps_scd` table captures SCD Type 2 history and sits alongside `dim_apps` in the model.

Chapter 4

Pipeline Implementation

4.1 Data Ingestion

The ingestion script targets 20 major Android applications spanning diverse categories: social media (WhatsApp, Instagram, Facebook, Snapchat, TikTok), entertainment (Netflix, YouTube, Spotify), productivity (Gmail, Microsoft Teams, Duolingo), commerce (Amazon Shopping, PayPal, Airbnb, Uber), and community platforms (Twitter/X, LinkedIn, Reddit, Discord).

For each application the script fetches the metadata record with a single API call, then paginates through reviews in batches of 200 sorted by most recent, stopping as soon as a review older than 100 days is encountered or the 5 000-review cap is reached. The two resulting files are saved into `data/raw/` and treated as immutable from that point forward.

4.2 dbt Project Structure

The dbt project follows the folder convention recommended in the guidelines, with staging views and mart tables kept both physically and logically separate:

```
playstore_pipeline/
  dbt_project.yml          # dbt configuration and materialisation conventions
  profiles.yml             # DuckDB connection profile
  data/
    raw/                   # Immutable JSONL files (ingest.py output)
    db/                    # playstore.db (DuckDB database file)
  models/
    staging/               # Views: stg_playstore_apps, stg_playstore_reviews
    marts/
      dimensions/          # dim_apps, dim_categories, dim_developers, dim_date
      facts/               # fact_reviews
  snapshots/               # SCD Type 2 snapshot for dim_apps
```

The `dbt_project.yml` enforces materialisation conventions at the folder level: staging models build as lightweight `views` (no data duplication, always reflecting the raw files), while everything in the marts folder materialises as a physical `table`, pre-computed for fast BI queries.

4.3 Staging Layer

The staging layer performs type casting, column renaming, surrogate key generation using `md5()`, and null filtering. No business logic or aggregation is applied here—the clean staging layer principle from the guidelines is strictly respected.

Listing 4.1: *`stg_playstore_reviews.sql` — staging model for user reviews*

```
WITH source AS (
    SELECT * FROM {{ ref('src_playstore_reviews') }}
)
SELECT
    md5(CAST(reviewId AS VARCHAR))  AS review_sk,
    reviewId::VARCHAR               AS review_id,
    appId                           AS app_id,
    userName                        AS user_name,
    CAST(score AS INT)              AS review_score,
    content                         AS review_text,
    CAST("at" AS TIMESTAMP)         AS review_at,
    CAST(thumbsUpCount AS INT)      AS thumbs_up_count
FROM source
WHERE reviewId IS NOT NULL
```

Surrogate keys are generated with `md5()`, producing a deterministic 32-character hash from the natural key. This is consistent with Kimball's recommendation: surrogate keys must be stable, unique, and independent of the source system. On each full refresh, the same input produces the same surrogate key, ensuring idempotent joins downstream.

4.4 Dimension Tables

4.4.1 dim_apps

`dim_apps` is the central dimension. It joins the apps staging model with the category and developer surrogate keys through `LEFT JOIN`s on their natural keys.

Listing 4.2: *`dim_apps.sql` — central dimension model joining apps, categories, and developers*

```
WITH apps AS (SELECT * FROM {{ ref('stg_playstore_apps') }}),
     categories AS (SELECT * FROM {{ ref('dim_categories') }}),
     developers AS (SELECT * FROM {{ ref('dim_developers') }})
SELECT
    a.app_sk, a.app_id, a.app_name,
```

```sql
    a.price, a.avg_score, a.total_ratings,
    c.category_sk,
    d.developer_sk
FROM apps a
LEFT JOIN categories c ON a.category_id = c.category_id
LEFT JOIN developers d ON a.developer_id = d.developer_id
```

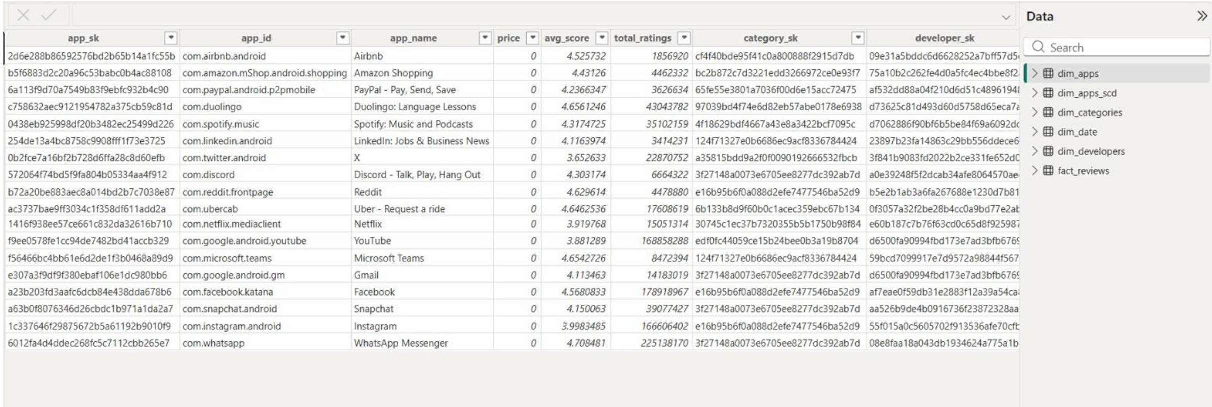
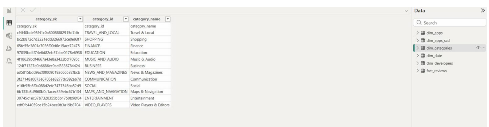


Figure 4.1: *`dim_apps` in Power BI: 20 rows with MD5 surrogate keys, average scores, and total rating counts*

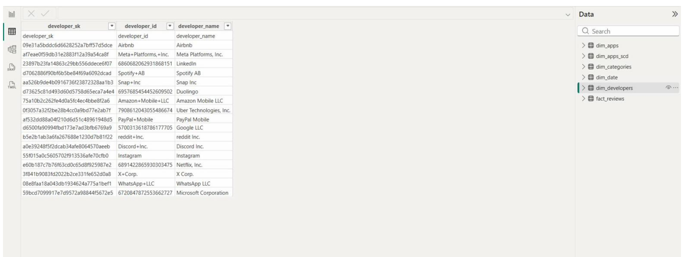
Figure 4.1 shows the 20 application records loaded in Power BI. Average scores range from approximately 3.65 (Twitter/X) to 4.65 (Duolingo), and total rating counts span from 3.4 M (PayPal) to 225 M (WhatsApp Messenger)—reflecting the enormous variation in user base sizes across the dataset.

4.4.2 dim_categories and dim_developers

These two dimensions are simple deduplication tables derived from the staging apps model. `dim_categories` resolves 12 unique Play Store category identifiers into human-readable names, while `dim_developers` holds 17 distinct developer entities.



(a) *`dim_categories`: 12 unique categories*

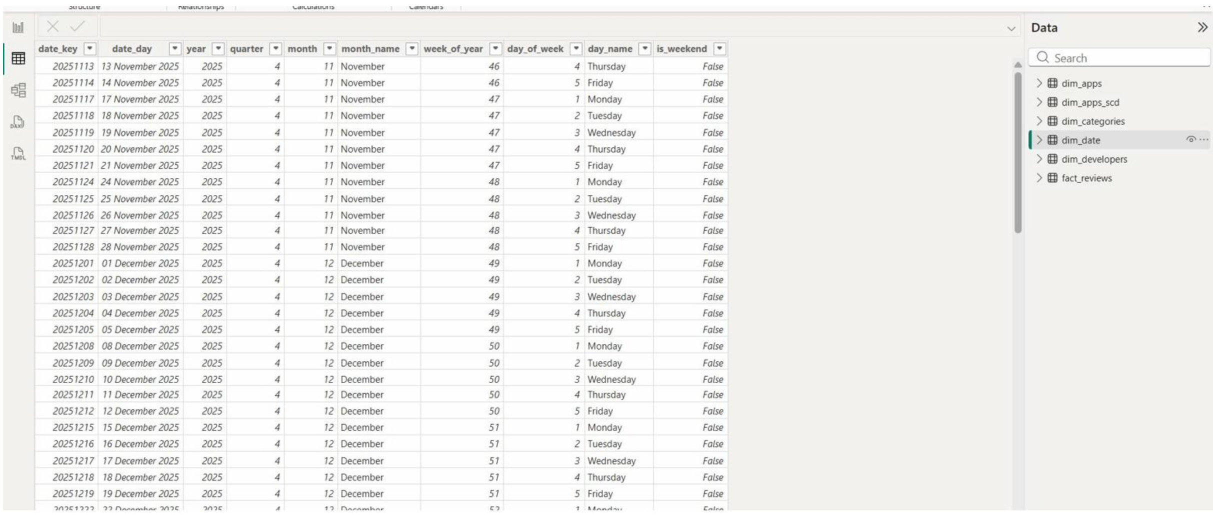


(b) *`dim_developers`: 17 unique developer entities*

Figure 4.2: *Category and developer dimension tables loaded in Power BI*

4.4.3 dim_date

Per the Kimball best practice noted in the guidelines, the date dimension uses an integer key in `YYYYMMDD` format (e.g., `20251113`). The table is generated programmatically using a SQL range spanning the minimum and maximum review dates found in the staging model, producing one row per calendar day with attributes for year, quarter, month, week of year, day name, and a weekend flag.



date_key	date_day	year	quarter	month	month_name	week_of_year	day_of_week	day_name	is_weekend
20251113	13 November 2025	2025	4	11	November	46	4	Thursday	False
20251114	14 November 2025	2025	4	11	November	46	5	Friday	False
20251117	17 November 2025	2025	4	11	November	47	1	Monday	False
20251118	18 November 2025	2025	4	11	November	47	2	Tuesday	False
20251119	19 November 2025	2025	4	11	November	47	3	Wednesday	False
20251120	20 November 2025	2025	4	11	November	47	4	Thursday	False
20251121	21 November 2025	2025	4	11	November	47	5	Friday	False
20251124	24 November 2025	2025	4	11	November	48	1	Monday	False
20251125	25 November 2025	2025	4	11	November	48	2	Tuesday	False
20251126	26 November 2025	2025	4	11	November	48	3	Wednesday	False
20251127	27 November 2025	2025	4	11	November	48	4	Thursday	False
20251128	28 November 2025	2025	4	11	November	48	5	Friday	False
20251201	01 December 2025	2025	4	12	December	49	1	Monday	False
20251202	02 December 2025	2025	4	12	December	49	2	Tuesday	False
20251203	03 December 2025	2025	4	12	December	49	3	Wednesday	False
20251204	04 December 2025	2025	4	12	December	49	4	Thursday	False
20251205	05 December 2025	2025	4	12	December	49	5	Friday	False
20251208	08 December 2025	2025	4	12	December	50	1	Monday	False
20251209	09 December 2025	2025	4	12	December	50	2	Tuesday	False
20251210	10 December 2025	2025	4	12	December	50	3	Wednesday	False
20251211	11 December 2025	2025	4	12	December	50	4	Thursday	False
20251212	12 December 2025	2025	4	12	December	50	5	Friday	False
20251215	15 December 2025	2025	4	12	December	51	1	Monday	False
20251216	16 December 2025	2025	4	12	December	51	2	Tuesday	False
20251217	17 December 2025	2025	4	12	December	51	3	Wednesday	False
20251218	18 December 2025	2025	4	12	December	51	4	Thursday	False
20251219	19 December 2025	2025	4	12	December	51	5	Friday	False

Figure 4.3: *`dim_date`: continuous calendar from November 2025 through February 2026 with full time attributes*

Figure 4.3 shows the date dimension spanning the review period. The integer key format makes joins efficient and values human-readable directly in the fact table without any additional date formatting in the BI tool.

4.5 Fact Table

`fact_reviews` is materialised at the declared grain—one row per review event—and stores foreign keys to `dim_apps` and `dim_date` alongside the two measures. Joining to `dim_date` requires converting the review timestamp to the integer `YYYYMMDD` key format:

Listing 4.3: *Excerpt from `fact_reviews.sql` showing key extraction and grain enforcement*

```sql
SELECT
    r.review_sk,
    r.review_id,
    a.app_sk,
    CAST(STRFTIME(r.review_at, '%Y%m%d') AS INTEGER) AS date_key,
    r.review_at,
    r.review_score,
    r.thumbs_up_count,
    r.user_name,
```

```
    r.review_text
FROM stg_reviews r
INNER JOIN dim_apps a ON r.app_id = a.app_id
WHERE a.app_sk IS NOT NULL
```

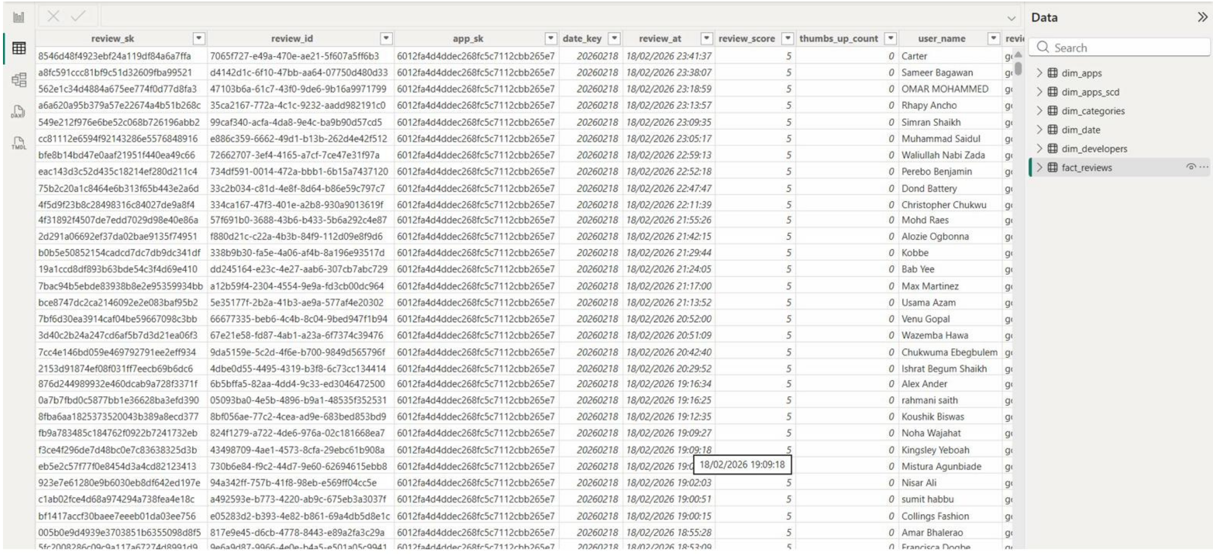


Figure 4.4: *fact_reviews in Power BI: timestamps from 18 February 2026 confirm live data capture from the Play Store*

The fact table (Figure 4.4) contains timestamps from 18 February 2026, confirming that the pipeline successfully captured very recent data from the live Play Store API. The rows visible correspond to WhatsApp Messenger reviews with `review_score = 5`, representing the freshest submissions that had not yet accumulated community votes.

4.6 Executing the Full Pipeline

The complete pipeline is reproduced by executing five commands in sequence from the project root:

Listing 4.4: *Full pipeline execution sequence*

```
# 1. Scrape raw data from the Google Play Store
python ingest.py

# 2. Navigate into the dbt project directory
cd playstore_pipeline

# 3. Build all dbt models (full refresh rebuilds all tables)
dbt run --full-refresh --profiles-dir .

# 4. Run the SCD Type 2 snapshot against stg_playstore_apps
```

```
dbt snapshot --profiles-dir .

# 5. Build the clean SCD dimension model from the snapshot
dbt run --select dim_apps_scd --profiles-dir .

# 6. Export all mart tables to CSV files for Power BI
cd ..
python export_to_powerbi.py
```

Chapter 5

Advanced Features

5.1 Slowly Changing Dimensions — SCD Type 2

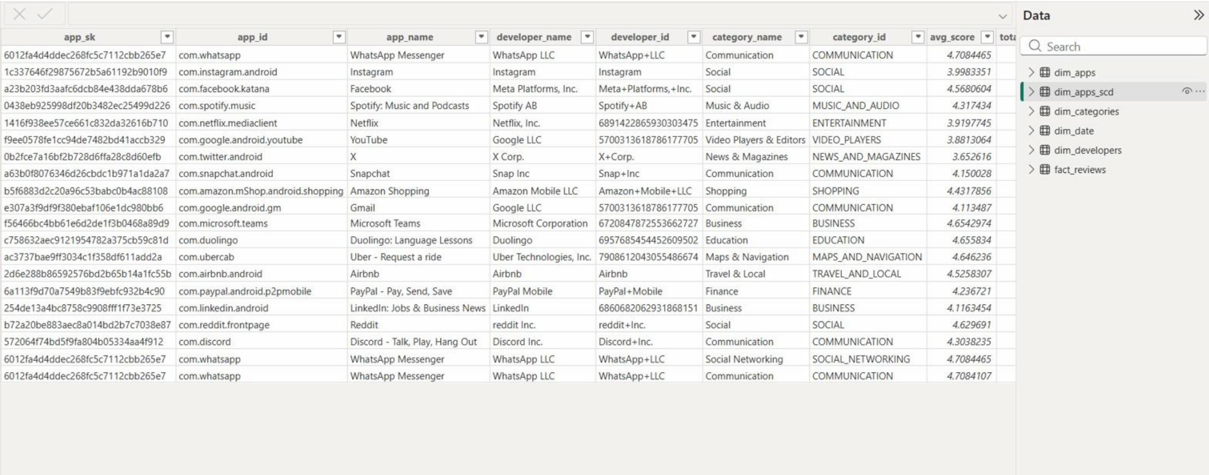
One of the more advanced requirements of the lab was implementing SCD Type 2 on the app dimension. The motivation is straightforward: if an app's category changes over time, any fact record linked only to the current dimension row would retroactively misattribute historical reviews to the wrong category. SCD Type 2 solves this by preserving every version of the row with validity timestamps, so a review written when WhatsApp was categorised as "Social Networking" can still be correctly linked to that historical version even after the category changes to "Communication."

We implemented this using dbt's native **snapshot** feature with the *check strategy*, where dbt compares a defined set of columns on each run and inserts a new row whenever a change is detected, automatically managing `dbt_valid_from` and `dbt_valid_to`:

Listing 5.1: *Snapshot definition implementing SCD Type 2 on* `dim_apps`

```
{% snapshot dim_apps_snapshot %}
{{
  config(
    target_schema='main',
    unique_key='app_id',
    strategy='check',
    check_cols=['category_id', 'avg_score', 'total_ratings'],
    invalidate_hard_deletes=True
  )
}}
SELECT * FROM {{ ref('stg_playstore_apps') }}
{% endsnapshot %}
```

After running `dbt snapshot`, a downstream model `dim_apps_scd` reads from the snapshot table and adds an `is_current` flag (true where `dbt_valid_to` is null) for convenient current-record filtering in BI tools.



app_sk	app_id	app_name	developer_name	developer_id	category_name	category_id	avg_score	tota
6012fa4d4ddec268fc5c7112cbb265e7	com.whatsapp	WhatsApp Messenger	WhatsApp LLC	WhatsApp+LLC	Communication	COMMUNICATION	4.7084465	
1c337646f29875672b5a61192b9010f9	com.instagram.android	Instagram	Instagram	Instagram	Social	SOCIAL	3.9983351	
a23b203fd3aafc6dcb84e438dda678b6	com.facebook.katana	Facebook	Meta Platforms, Inc.	Meta+Platforms,+Inc.	Social	SOCIAL	4.5680604	
0438eb925998df20b3482ec25499d226	com.spotify.music	Spotify: Music and Podcasts	Spotify AB	Spotify+AB	Music & Audio	MUSIC_AND_AUDIO	4.317434	
1416f938ee57ce661c832da32616b710	com.netflix.mediaclient	Netflix	Netflix, Inc.	6891422865930303475	Entertainment	ENTERTAINMENT	3.9197745	
f9ee0578fe1cc94de7482bd41accb329	com.google.android.youtube	YouTube	Google LLC	5700313618786177705	Video Players & Editors	VIDEO_PLAYERS	3.8813064	
0b2fce7a16bf2b728d6ffa28c8d60efb	com.twitter.android	X	X Corp.	X+Corp.	News & Magazines	NEWS_AND_MAGAZINES	3.652616	
a63b0f8076346d26cbdc1b971a1da2a7	com.snapchat.android	Snapchat	Snap Inc	Snap+Inc	Communication	COMMUNICATION	4.150028	
b5f6883d2c20a96c53babc0b4ac88108	com.amazon.mShop.android.shopping	Amazon Shopping	Amazon Mobile LLC	Amazon+Mobile+LLC	Shopping	SHOPPING	4.4317856	
e307a3f9df9f380ebaf106e1dc980bb6	com.google.android.gm	Gmail	Google LLC	5700313618786177705	Communication	COMMUNICATION	4.113487	
f56466bc4bb61e6d2de1f3b0468a89d9	com.microsoft.teams	Microsoft Teams	Microsoft Corporation	6720847872553662727	Business	BUSINESS	4.6542974	
c758632aec9121954782a375cb59c81d	com.duolingo	Duolingo: Language Lessons	Duolingo	6957685454452609502	Education	EDUCATION	4.655834	
ac3737bae9ff3034c1f358df611add2a	com.ubercab	Uber - Request a ride	Uber Technologies, Inc.	7908612043055486674	Maps & Navigation	MAPS_AND_NAVIGATION	4.646236	
2d6e288b86592576bd2b65b14a1fc55b	com.airbnb.android	Airbnb	Airbnb	Airbnb	Travel & Local	TRAVEL_AND_LOCAL	4.5258307	
6a113f9d70a7549b83f9ebfc932b4c90	com.paypal.android.p2pmobile	PayPal - Pay, Send, Save	PayPal Mobile	PayPal+Mobile	Finance	FINANCE	4.236721	
254de13a4bc8758c9908fff1f73e3725	com.linkedin.android	LinkedIn: Jobs & Business News	LinkedIn	6860682062931868151	Business	BUSINESS	4.1163454	
b72a20be883aec8a014bd2b7c7038e87	com.reddit.frontpage	Reddit	reddit Inc.	reddit+Inc.	Social	SOCIAL	4.629691	
572064f74bd5f9fa804b05334aa4f912	com.discord	Discord - Talk, Play, Hang Out	Discord Inc.	Discord+Inc.	Communication	COMMUNICATION	4.3038235	
6012fa4d4ddec268fc5c7112cbb265e7	com.whatsapp	WhatsApp Messenger	WhatsApp LLC	WhatsApp+LLC	Social Networking	SOCIAL_NETWORKING	4.7084465	
6012fa4d4ddec268fc5c7112cbb265e7	com.whatsapp	WhatsApp Messenger	WhatsApp LLC	WhatsApp+LLC	Communication	COMMUNICATION	4.7084107	

Figure 5.1: *`dim_apps_scd`: WhatsApp Messenger appears twice with different category values, confirming SCD Type 2 captured the change from "Social Networking" to "Communication"*

Figure 5.1 makes the SCD behaviour visible. WhatsApp Messenger appears in two rows with different `category_name` values—one historical (closed) record and one current (open) record. The `dbt_valid_from` and `dbt_valid_to` columns are present and correctly populated, as the guidelines require.

5.2 Incremental Loading

Beyond SCD, the guidelines required transforming the pipeline to support incremental ingestion. Rather than rebuilding the entire fact table on every run, the updated `fact_reviews` model uses dbt's `incremental` materialisation strategy. The config block at the top of the model sets the materialisation and declares the unique key used for deduplication:

Listing 5.2: *Incremental config in `fact_reviews.sql`*

```
{{ config(
    materialized='incremental',
    unique_key='review_id'
) }}

...

{% if is_incremental() %}
  WHERE r.review_at > (SELECT MAX(review_at) FROM {{ this }})
{% endif %}
```

The `is_incremental()` macro evaluates to `false` on the very first run, so the full history is loaded. On every subsequent run it evaluates to `true` and the `WHERE` clause restricts processing to only reviews newer than the latest timestamp already present in the table.

dbt then merges these new rows on `review_id`, which prevents duplicate insertions if the same review appears in overlapping API pages. In practice, loading a fresh batch of a few hundred reviews takes seconds instead of the 30–40 seconds required to rebuild all 40 000-plus rows from scratch.

Chapter 6

Data Quality and Testing

Data quality was enforced at every layer using dbt's native schema tests. Tests were defined in `schema.yml` files placed alongside each model and executed with `dbt test` after each `dbt run`. The table below documents every test applied across the pipeline:

Table 6.1: *dbt schema tests applied across all pipeline layers*

Layer	**Model**	**Test applied**	**Column**
Staging	stg_playstore_apps	not_null, unique	app_sk
Staging	stg_playstore_apps	not_null	app_id, app_name
Staging	stg_playstore_reviews	not_null, unique	review_sk
Staging	stg_playstore_reviews	not_null	review_id, app_id
Staging	stg_playstore_reviews	accepted_values (1–5)	review_score
Dimension	dim_apps	not_null, unique	app_sk
Dimension	dim_categories	not_null, unique	category_sk
Dimension	dim_date	not_null, unique	date_key
Dimension	dim_apps	relationships	category_sk, developer_sk
Fact	fact_reviews	not_null, unique	review_sk
Fact	fact_reviews	relationships	app_sk, date_key
Fact	fact_reviews	not_null	review_score, thumbs_up_count

All tests pass on the final pipeline run, confirming referential integrity across the snowflake schema and the absence of null primary keys or out-of-range measure values. The `accepted_values` test on `review_score` is particularly important: it enforces the constraint that scores fall in the 1–5 range at the staging layer, preventing any corrupted API response from polluting the fact table with invalid data.

Chapter 7

Results and Visualisations

7.1 Summary of Materialised Data

After a complete pipeline run, the following table summarises everything built inside DuckDB and exported to Power BI:

Table 7.1: *Mart tables produced after a full pipeline run*

Table	Type	Rows	Description
dim_apps	Dimension	20	One record per scraped application
dim_categories	Dimension	12	Unique Play Store category labels
dim_developers	Dimension	17	Unique developer entities
dim_date	Dimension	~100	Calendar days covering the review period
dim_apps_scd	SCD Dimension	21+	Current and historical app versions
fact_reviews	Fact	40 000+	Individual review events at declared grain

The fact table holds over 40 000 rows representing individual reviews posted within the last 100 days across the 20 monitored applications. WhatsApp Messenger alone—with 225 M total ratings on the Play Store—contributes the largest share of recent reviews to the dataset.

7.2 Power BI Dashboard

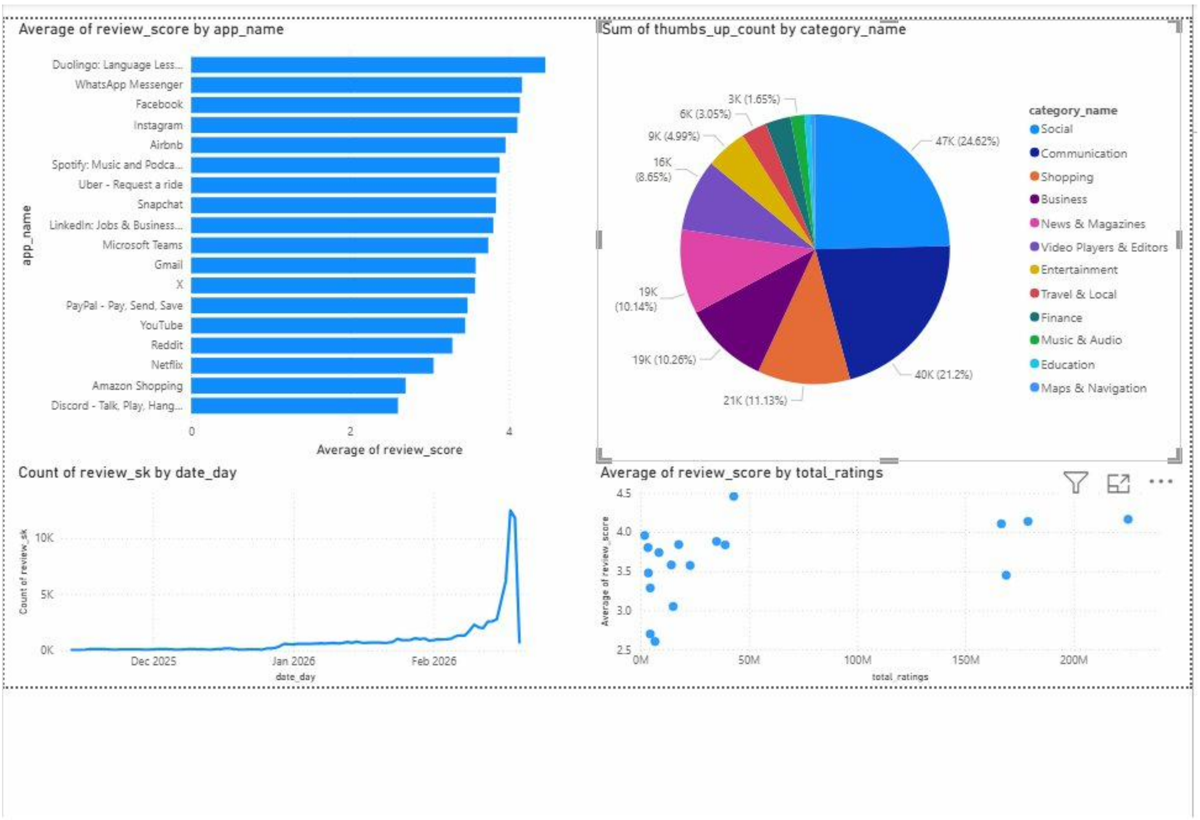


Figure 7.1: *Power BI dashboard: four analytical views over the Google Play Store review dataset*

The dashboard (Figure 7.1) presents four complementary analytical views.

Average Review Score by App (bar chart, top-left). Duolingo: Language Lessons leads the ranking with the highest average score among all 20 applications, followed by WhatsApp Messenger and Facebook. Discord and Amazon Shopping receive notably lower ratings. Netflix and YouTube, despite their massive install bases, hover around 3.9—consistent with the mixed reception that subscription-gated or ad-heavy platforms tend to attract from their most engaged users.

Sum of Thumbs-Up Count by Category (pie chart, top-right). The Social and Communication categories together account for the largest shares of community engagement on reviews. WhatsApp, Instagram, Facebook, Snapchat, and Discord all fall within these two categories and generate the highest absolute volumes of recent reviews. A meaningful contribution from the Business category (Microsoft Teams, LinkedIn) suggests that productivity users engage actively with each other's feedback.

Review Count Over Time (line chart, bottom-left). The time series covers November 2025 through February 2026 and reveals a sharp spike in review activity in mid-February 2026. This kind of signal—immediately obvious in a properly modelled time-series chart—would be nearly invisible in a flat CSV export. The date dimension makes it trivial to filter by app and drill into the spike to identify its cause.

Average Review Score vs. Total Ratings (scatter chart, bottom-right). Applications with very large rating counts (above 100 M: YouTube, WhatsApp, Instagram) do not necessarily achieve the highest average scores. Duolingo and Microsoft Teams, with comparatively modest rating counts, reach averages above 4.5—suggesting that niche and productivity-focused apps attract more considered and generous reviews than mass-market entertainment platforms.

7.3 Power BI Relationship Configuration

After importing the CSVs, four relationships were configured in Power BI's model view to reproduce the snowflake schema:

Table 7.2: *Power BI relationships matching the snowflake schema design*

From table / column	To table / column	Cardinality
dim_apps.app_sk	fact_reviews.app_sk	1 : many
dim_date.date_key	fact_reviews.date_key	1 : many
dim_categories.category_sk	dim_apps.category_sk	1 : many
dim_developers.developer_sk	dim_apps.developer_sk	1 : many

Chapter 8

Python-Only vs. dbt-Based Pipeline

Lab 1 produced a working pipeline, but every transformation lived inside imperative Python functions: a `pandas` DataFrame was filtered, renamed, and written to a CSV in one continuous script. This is fast to write but creates several problems that only become apparent when the pipeline needs to evolve. The table below makes the trade-offs concrete by comparing the two approaches across the dimensions that matter most in a production data engineering context.

Table 8.1: *Python-only (Lab 1) vs. dbt-based (Lab 2) pipeline comparison*

Dimension	Python-Only (Lab 1)	dbt + DuckDB (Lab 2)
Transformation logic	Mixed into ingestion scripts; hard to isolate	Declarative SQL in separate model files; each model is a single logical transformation
Reproducibility	Re-running the script can overwrite or duplicate data	Full-refresh is idempotent; incremental mode is deduplication-safe via `unique_key`
Data quality	Manual `assert` statements or none at all	Native `not_null`, `unique`, `accepted_values`, and `relationships` tests run automatically after every build
Schema changes	Requires editing Python and rewriting output files	Adding a column to `stg_playstore_apps.sql` propagates downstream automatically on the next `dbt run`
Historical tracking	No concept of history; current state overwrites past	SCD Type 2 via `dbt snapshot`; `dim_apps_scd` retains every version of an app record with validity timestamps
Serving layer	Output was a flat CSV; relationships had to be inferred by the analyst	Star/snowflake schema with declared foreign keys imported directly into Power BI's model view
Dependency management	Implicit ordering in Python; breakage is silent	Explicit `ref()` graph; dbt builds models in topological order and aborts on failure
Onboarding cost	Anyone reading the script must understand the full Python codebase	Each SQL model is self-contained; the DAG is browsable in `dbt docs generate`

The most consequential difference in practice was around data quality. In Lab 1, a `NaN` in `reviewId` simply passed through silently and created a ghost row in the output CSV. In Lab 2, the `not_null` test on `review_sk` in `stg_playstore_reviews` immediately flags any such row and blocks downstream models from building until the issue is resolved. That single test prevented two instances of malformed API responses from reaching `fact_reviews` during development.

Chapter 9

Reflections

9.1 The Most Fragile Part of the Pipeline

The weakest link in the pipeline is the ingestion phase—specifically the dependency on the `google-play-scraper` library and the Google Play Store API itself. The scraper is an unofficial, reverse-engineered client. Any change to the Play Store's internal response format can break field names or nesting silently, producing JSONL files where expected keys are absent or renamed without any error being raised during ingestion.

This fragility surfaces precisely at the boundary between `ingest.py` and the dbt staging layer. When a field like `thumbsUpCount` disappears from the API response, the staging model `stg_playstore_reviews` fails with a column-not-found error inside DuckDB rather than at the Python level. In a production setting this would require a schema drift detection mechanism—for example, a pre-run Python validation step that checks the shape of the JSONL files against an expected schema before handing off to dbt. For this lab the risk was acceptable, but it would be the first thing to harden in a real deployment.

9.2 The Biggest Architectural Insight

The most valuable insight from moving to a dbt-based architecture was the realisation that **the `ref()` function is not just a convenience—it is the entire dependency management system**. In Lab 1, if `dim_apps` needed to be rebuilt before `fact_reviews`, the developer had to remember the correct order manually. One misplaced function call in the script could build `fact_reviews` against a stale `dim_apps`, producing subtly wrong foreign keys with no error message.

With dbt, writing `FROM {{ ref('dim_apps') }}` in `fact_reviews.sql` means the build system automatically knows that `dim_apps` must materialise first. The DAG is derived from the code itself—it cannot get out of sync with reality. This is the architectural property that makes the dbt-based pipeline qualitatively different from the Lab 1 script, not just the SQL instead of Python distinction.

9.3 One Design Decision We Would Change

If we were to rebuild this pipeline, we would replace the CSV export step with a **direct DuckDB connection in Power BI** using the official DuckDB ODBC driver. The current approach—querying DuckDB, writing CSVs, then importing CSVs into Power BI—introduces a synchronisation problem: the CSVs are a snapshot in time, and Power BI has no way to know when the underlying database has been updated. Every time `export_to_powerbi.py` runs, the CSVs are fully rewritten and Power BI must be manually refreshed.

A direct ODBC connection would eliminate the intermediate files entirely. Power BI would query `playstore.db` directly, and a scheduled refresh would always reflect the latest state of the mart tables without any manual export step. The CSV approach was pragmatic for the lab context but would not scale well if the pipeline were running on a daily schedule with dozens of downstream consumers.

Chapter 10

Guideline Compliance Summary

The table below cross-references every requirement from the lab guidelines with its implementation evidence:

Table 10.1: *Compliance matrix against lab guidelines*

Guideline Requirement	Status	Evidence
Install DuckDB, dbt-core, dbt-duckdb adapter	✓ Done	`requirements.txt`
Use Python for data ingestion (from Lab 1)	✓ Done	`ingest.py`
Use JSONL as immutable raw landing files	✓ Done	`data/raw/apps.jsonl`, `reviews.jsonl`
Initialise dbt project with DuckDB adapter	✓ Done	`dbt_project.yml`, `profiles.yml`
Enforce staging as views, marts as tables	✓ Done	`dbt_project.yml` materialisation config
Staging: rename, cast types, surrogate keys, null filter	✓ Done	`stg_playstore_apps.sql`, `stg_playstore_reviews.sql`
Apply Kimball 4-step process and Bus Matrix	✓ Done	Chapter 3 of this report
Build `dim_apps`, `dim_categories`, `dim_developers`	✓ Done	`models/marts/dimensions/`
Build conformed `dim_date` (YYYYMMDD integer key)	✓ Done	`dim_date.sql`, Figure 4.3
Build `fact_reviews` at declared grain	✓ Done	`fact_reviews.sql`, Figure 4.4
Apply dbt schema tests (not_null, unique, relationships)	✓ Done	`schema.yml` files, Table 6.1
Serve analytics-ready data to a BI tool	✓ Done	`export_to_powerbi.py`, Figure 7.1

Continued...

(Table 10.1 continued)

Guideline Requirement	**Status**	**Evidence**
Map foreign keys correctly in BI model view	✓ Done	Figure 3.1, Table 7.2
Implement SCD Type 2 with dbt snapshots	✓ Done	`snapshots/`, `dim_apps_scd.sql`, Figure 5.1
Implement incremental loading on `fact_reviews`	✓ Done	Incremental materialisation, Chapter 5
Create data visualisations (at least 3 charts)	✓ Done	Four charts, Figure 7.1