Lab 2

Data Pipeline with dbt & DuckDB
January 2026

Objectives

- Install and Set up our environment
- Explore Data modeling for Analytics
- Structure a data pipeline using dbt models and layers
- Separate raw data, transformation logic, and serving tables
- Apply data quality tests using dbt
- Prepare a stable serving layer for dashboards

A. Environment Setup

For this second lab, we will create a Data Pipeline based on dbt and Duckdb. We can reuse the virtual environment we used for lab 1.

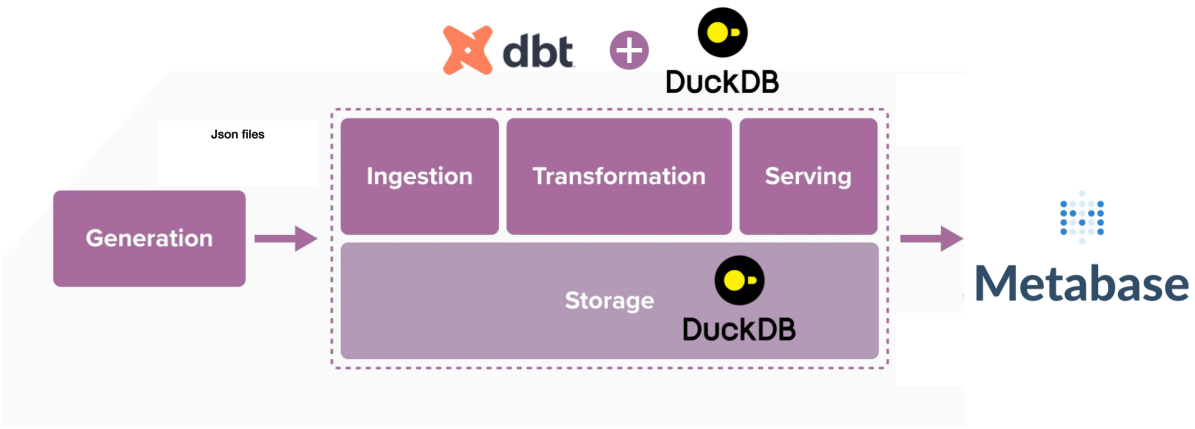
✓ Install Duckdb, dot-core, and the dbt-duckdb adapter in your virtual environment.

When running your instructions/code, make sure you are running it in the right virtual environment (e.g. in the terminal you need to conda activate your environment, in visual Code IDE you can choose your environment by clicking on the bottom right banner).

If your installation goes well, you should see, upon running **dbt --version**, duckdb in dbt Plugins.

B. High-level architecture

The objective of this lab is to re-engineer the pipeline implemented in Lab 1 by replacing the ad-hoc Python logic with standard data engineering frameworks. To this end, DuckDB will be used as the analytical database, while dbt will be employed to model, transform, and validate the data pipeline.



We will retain our **Python** code for accessing Google play store's API for data ingestion, which handle data extraction and initial loading. We will use **dbt Core** (an open-source transformation framework to define, version, test, and document data transformations using SQL) to orchestrate the transformation logic, structuring the data through modeling, cleaning, and business rules, and preparing it for analytics. The data will be stored in **DuckDB** (an open-source database management system designed for analytical workloads), which acts as the unified analytical storage and execution engine. Finally, the transformed and analytics-ready data is served directly from DuckDB to downstream tools such as PowerBI, Tableau, or Metabase enabling efficient exploration and visualization without the need for heavy infrastructure.

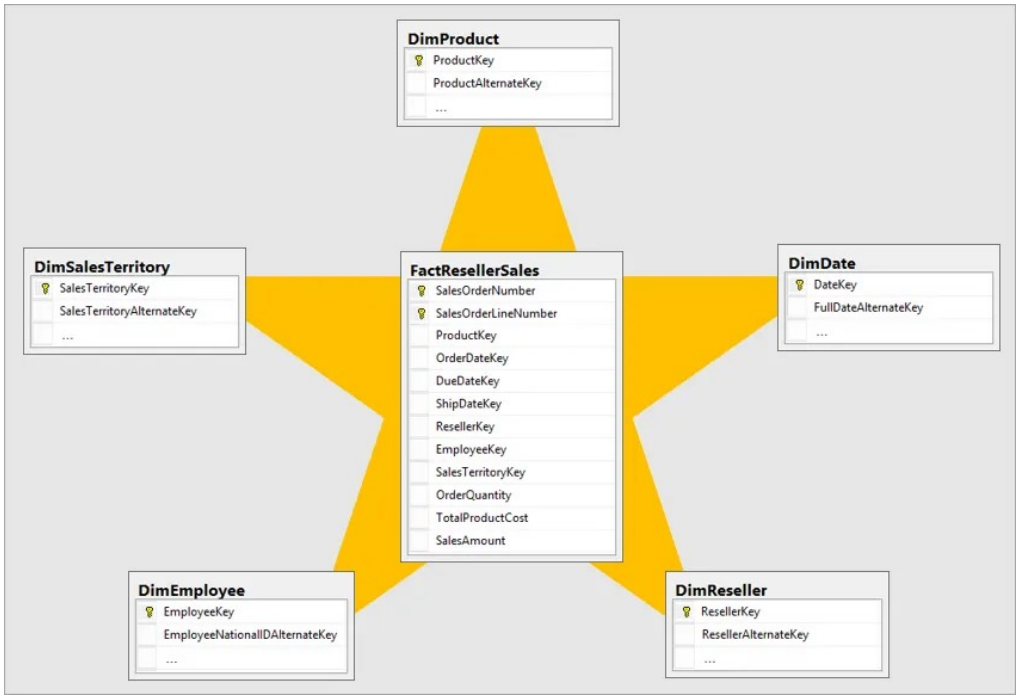
C. Data Modeling

Storing, manipulating, and leveraging data effectively relies on a good structured and organized environment. **Data modeling** plays a relevant role in achieving this objective by providing the blueprint for representing a specific reality or a business and supporting its processes and rules.

Understanding the business's operations, terminology, and processes is essential for creating accurate and meaningful data models. By gathering requirements through interviews, document analysis, and process studies, we gain insights into the business's needs and data requirements. Once the understanding phase is complete, we move to the modeling of our data, and specifically dimensional modeling.

Dimensional data modeling is an approach to structuring data that is specifically designed for analytics and decision-making, rather than for transactional processing. It organizes data around business processes (facts) and their context (dimensions), making analytical queries easier to write, easier to understand, and more efficient to execute. In this paradigm, facts capture measurable events (such as app reviews), while dimensions provide descriptive attributes (such as apps, categories, or developers) used to analyze those events.

The **star schema** is the most common physical representation of dimensional data modeling, translating its core principles—facts, dimensions, and grain—into a simple and efficient table structure optimized for analytical queries.



The star schema is designed specifically to support efficient querying, reporting, and analysis. It organizes data into fact tables, which capture measurable business events or observations (such as sales, reviews, or inventory levels), and dimension tables, which provide the descriptive context needed to analyze those events (such as time, products, people, or categories). **Dimension tables** contain unique identifiers and rich descriptive attributes, often including a dedicated date dimension to enable flexible time-based analysis. **Fact tables** store foreign keys to these dimensions along with numeric measures, and their **granularity** is defined by the combination of dimension keys they contain. By clearly separating descriptive information from measurable events, the star schema offers an intuitive structure that simplifies aggregations, improves query performance, and helps analysts easily understand and explore relationships within the data to derive meaningful insights.

A **snowflake schema** is more normalized than in a star schema. It contains additional levels of normalization by splitting dimension tables into multiple contiguous tables: A dimension table product for instance, that contains product information such as ID, name, and category, will be split table into several contiguous tables (one for product and one for product category).

The Kimball approach [1] is the most widely adopted methodology for dimensional modeling. It advocates starting from the **business process** (step 1), defining a clear **grain** (step 2), and then identifying the **dimensions** (step 3) and the **facts** (step 4). Once done, we can design the **star schemas** composed of the fact tables and the denormalized and conformed dimension tables. This approach emphasizes simplicity, performance, and usability for analysts and BI tools.

In this lab, we adopt the Kimball methodology because our objective is not transactional consistency but analytical exploration of the Google Play data. To do that, we will start from the **analytical use case** and progressively identify the core elements of our star schema.

- ✓ To identify the business process, we need to inspect our raw datasets with purpose. A business process represents a set of measurable events that occur in the system and that we want to analyze. What recurring event or activity do we want to measure and analyze?

- ✓ To declare our grain, which represent the level of detail of our fact table, complete the following sentence: “One row in the fact table represents …”

- ✓ Identify the dimensions that provide context to analyze the business process. Dimensions describe the *who, what, where, and when* of the business process and are used to filter, group, and aggregate facts. List the dimensions that answer questions such as: Which app? Which category or developer? When did the event occur? And for each dimension, specify the business meaning and the source dataset.

- ✓ To identify the facts (facts are quantitative values that can be aggregated (e.g., summed, averaged, counted)) associated with the business process, List the measurable attributes related to the event, and which aggregation functions make sense for each measure (e.g., average, count).

Once the business process, grain, dimensions, and facts have been identified, the next step in the Kimball methodology is to formalize the design using a **Bus Matrix**. The Bus Matrix acts as a bridge between business requirements and the physical star schema.

- ✓ Create the Bus Matrix where each cell indicates whether the facts produced by a given business process can be analyzed using a given dimension.

Facts \ Dimensions	Dim 1	Dim 2	Dim3
Fact 1	X (when Fact 1 is analyzable by Dim 1)		
Fact 2			

✓ Use the completed Bus Matrix to design the **star schema**. Create **one fact table** for each combination of rows that are analyzable by the same dimensions. Create **one dimension table** for each column marked in the matrix.

Ensure that:
- The fact table contains the declared grain, the measures, and the foreign keys,
- Each dimension table contains descriptive attributes used for analysis,
- The model remains denormalized and analysis-oriented.

✓ Validate the Model Against Analytical Needs.

Validate your star schema by ensuring that:
- all analytical questions defined earlier can be expressed as aggregations on the fact table,
- required filters and groupings are provided by the dimensions,
- the declared grain is respected in all joins.

D. Dbt & Duckdb-based Pipeline

Now that we have designed our dimensional star schema for analyzing Google Play Store data, we will start creating the data pipeline.

In this step, we will build a modular analytical data pipeline that **reuses the Python-based data extraction code** developed in Lab 1 and refactors it into a more robust data engineering architecture.

The Python code will remain responsible for data ingestion, collecting data from the Google Play Store API and persisting it in raw form (e.g., JSON or CSV files). These raw datasets constitute a landing or **staging area**, where data is stored in an immutable format and preserved without business transformations.

On top of this raw layer, **dbt** will be used to **implement all transformation logic**, progressively cleaning, standardizing, and reshaping the data into a dimensional star schema.

DuckDB will serve as both the execution engine and the analytical storage layer, materializing staging tables, dimensions, and fact tables.

The final outcome is a pipeline that clearly separates ingestion from transformation and serving, enabling reproducibility, data quality checks, and efficient analytical querying by downstream tools such as BI dashboards or data applications.

N.B. You can refer to the documentation of both dbt Core and DuckDB for detailed information and commands.

1. dbt + DuckDB Configuration

First, we will focus on setting up the tooling that will allow dbt to manage transformations and DuckDB to execute and store analytical data locally. We will not perform any transformations yet, we will only establish the execution environment.

- ✓ Initializes a new dbt project. When prompted select DuckDB as the database adapter. This should create a dbt configuration folder with your project name and sample files, and a connection profile on your local machine.
- ✓ Navigate to the newly created project directory and explore its content.

- ✓ Create data folders for your raw data (where we will store the output of our data extraction Python code) and for the duckdb database file.
- ✓ A common best-practice structure organizes models logically rather than a single models/ folder. We will thus create folders for staging and marts within the models folder.

- ✓ Locate the profiles.yml file in your project root folder. A **dbt profile** is a configuration file that tells dbt where your data warehouse is and how to connect to it. To find where dbt expects profiles, you can run: dbt debug --config-dir
- ✓ Modify the profiles.yml file to add a DuckDB profile that tells dbt to use the DuckDB adapter, the path to the database file, the number of execution threads to use etc.
- ✓ Verify the connection by running: dbt debug (make sure to run dbt from the project root)

- ✓ Locate the project **yml file** created by dbt. Verify that the profile is linked to your dbt project. Modify We also want to define how dbt should build models by default depending on their folder. Here, we want to set dbt **model conventions**. We will define model materialization conventions to enforce a clear separation between lightweight staging transformations (**views**) and analytics-ready mart tables (**materialized** tables).

- ✓ To make sure our configuration is correct, we will run a connection sanity check. Create a small model to prove dbt can create objects in DuckDB: create and run a sql file in your staging folder with the following instructions: select 1 as ok, current_timestamp as created_at
- ✓ Verify that the db file was created. You can open the DuckDB CLI and run a select on your table.

2. Building the staging layer

In this step we will create a staging layer that will make the **raw JSONL files** (from Lab 1) queryable through dbt, and produce two **clean staging models**:

- stg_playstore_apps
- stg_playstore_reviews

- ✓ Move your raw files into the raw data folder of your dbt project
- ✓ Create two models for ingesting the raw data. They will expose the JSON as relations dbt can reference. Use Duckdb's json reader.

N.B. A dbt model should represent one logical transformation at a single grain. It is not recommended to combine datasets with different grains in the same staging model.

At the staging stage, we do not apply business logic or aggregations; instead, we focus on renaming columns consistently, casting fields to the correct data types, handling obvious null or malformed values, and ensuring that each dataset respects its declared grain. The staging layer acts as a controlled interface between raw ingestion and downstream transformations, improving data quality, readability, and reproducibility throughout the pipeline.

- ✓ We want to prepare the raw Play Store data for further transformation. Create two models for standardizing naming (e.g., appId > app_id) and types e.g., text > integer, string > timestamp) of both apps and reviews, applying minimal cleaning (removing null keys, basic filtering), creating surrogate keys, flattening nested structures etc.

A powerful functionality of dbt is its native testing features. **Generic dbt tests** are out-of-the-box tests that you can apply across multiple data models. Dbt also enables you to create custom tests that allow you to develop tests that are customized for a specific data model.

- ✓ To ensure that the staging models produce clean, reliable, and structurally sound data before building the dimensional models, we will create some basic schema tests. In the same directory as your model, create a `schema.yml` file. Define the tests associated with the appropriate column of your model: You can for instance test your data against constraints such as **not_null** and **unique** on key fields, or enforce the relational integrity between apps and reviews, value range testing etc.

- ✓ Execute your staging code then Run the tests on your staging tables.

3. Building dimensional tables

In this step we will create the dimension tables that will be used by the fact table. We'll build them from the staging models created in the previous step: from stg_playstore_apps, our output will be: dim_developers, dim_categories, dim_apps, and dim_date.

- ✓ Create a folder for your data marts code files and your dimensions within: models > marts > dimensions
- ✓ Create a model for each dimension where:
  - ✓ Each dimension must have a **surrogate key**.

A **natural key** is a primary key that is innate to the data. Perhaps in some tables there's a unique id field in each table that would act as the natural key.

A **surrogate key** is a unique identifier derived from the data itself. It often takes the form of a **hashed** value of multiple columns that will create a **uniqueness constraint** for each row.

- ✓ Each dimension retrieves attributes from the stg_playstore_apps table
- ✓ We will create a hierarchy for dim_apps > dim_categories

When building dimensions, it is crucial to think of how our data warehouse will evolve in time. As we add more fact tables, they must share standardized and consistent dimension tables (e.g., Date, Apps, etc.) to ensure uniform definitions, keys, and attributes for accurate cross-functional analysis and drilling across disparate business processes. These dimensions are called **conformed dimensions**.

On **dim_date**: Every data warehouse needs a conformed date dimension which is a single, standardized, and shared table used across multiple fact tables, ensuring standardized time-based analysis and enableing consistent grouping (day/week/month/quarter) across dashboards and metrics. Date dimension tables are commonly **populated using SQL scripts** to generate a continuous range of dates with attributes like fiscal years, holidays, and weekdays etc.

- ✓ We will create a reusable **date dimension** table that supports time-based analysis (daily trends, month/quarter rollups, weekday/weekend effects) and generate a continuous date range. We can use first and last review dates as anchors because time is defined by review events.

N.B. In Kimball dimensional modeling, the date key (or date dimension primary key) is typically an integer in **YYYYMMDD** format (e.g., 20260211) rather than a strictly sequential surrogate key. This intelligent key allows for easy partitioning, faster joins, and human-readable data directly in the fact table.

- ✓ Add schema tests for the dimensions. Make sure to test for referential integrity in the apps hierarchy.
- ✓ Execute your dimensions code then Run the tests on your tables.

4. Building the fact table

Once our dimensions are created, we will create the analytics-ready fact table fact_reviews at the declared grain: One row per review event for one app at a given point in time. A fact table consists of **foreign keys** to the dimensions and the **measures** needed for analysis.

- ✓ Create a folder for your fact tables code files within the data mart folder: models > marts > facts

- ✓ Create the model for fact_reviews where:
  - ✓ Data is retrieved from stg_playstore_reviews
  - ✓ The joins in the fact table must be done on the dimensions' surrogate keys. We will thereby need to use the natural keys to retrieve them.
  - ✓ To join with the dim_date, we will need to retrieve the dates from the reviews' timestamps
  - ✓ Dimension keys that are missing have to be filtered out because the fact table must have valid foreign keys.

- ✓ Validated your fact table with referential integrity tests. You can also add range test on some measures etc.
- ✓ Execute your fact table's code then Run the tests to ensure the viability of your data.

At this stage, our dbt pipeline has produced an **analytics-ready star (snowflake) schema**.

5. Serving Layer & BI Consumption

The objective of this step is to demonstrate how a properly modeled star schema can be consumed by a Business Intelligence (BI) tool for reporting and interactive analysis. You may use: Power BI (for Windows users) or Metabase (local) (for macOS/Linux users) etc.

- ✓ Connect your BI platform with the Duckdb database (you mind need to Install a DuckDB Driver for Power Bi or Tableau)
- ✓ Explore your tables on the BI's platform logical model. Make sure the foreign keys were identified correctly. Re-Map them when needed
- ✓ Create data visualization to analyze: Monthly Average Rating Trend or Developer Performance Comparison etc.

E. Chaos Engineering for data pipelines

In the previous lab, we manually handled stress scenarios for our Python-only pipeline such as: New reviews batch, Schema drift, Dirty or inconsistent records etc. In this section, we will revisit some of these stress cases and analyze how a dbt-based architecture addresses them more robustly and systematically.

1. Incremental loading on dbt

In this section, we will transform the pipeline from a **full refresh architecture** into an **incremental data pipeline** capable of handling new review batches, daily ingestion, efficient updates, and idempotent reruns. In fact, our current pipeline stores all reviews from raw data in stg_playstore_reviews and entirely rebuilds fact_reviews on each run (since we materialized it as table).

Dbt provides **incremental models** to solve performance issues, duplicate insertions, and full-table recomputation. Incremental models are built as tables in the data warehouse. The first time a model is run, the table is built by transforming all rows of source data. On subsequent runs, dbt transforms only the rows in your source data that you tell dbt to filter for, inserting them into the target table which is the table that has already been built.

- ✓ Update the reviews_fact model so that the table is incremental and the checks are done on the unique key (review_id).
- ✓ We will also need to implement Incremental Logic where we only load data with a review date is superior to the date of existing reviews.
- ✓ To test the incremental loading, append new records to the reviews.jsonl.
- ✓ Rerun your pipeline and observe how it will react to the new batch.

2. Slowly Changing Dimensions (SCD Type 2)

Slowly Changing Dimensions (SCDs) are a critical concept in data warehousing that deal with how data changes over time. In many cases, business questions can be related to changes of the truth over time (e.g. Which customers moved? How often do customers change their e-mail address? etc.). There are many SCD types, each responding to specific business needs. Type 2 is particularly useful when we want to retain the entire history of changes.

Dbt allows to implement type 2 SCD using **snapshots**. A snapshot is a copy of a dataset saved at a specific point in time (whether at regular intervals or on demand) and might contain all the data (full data export) or only the most recent data (created or changed since a known time). The snapshot name recalls that the data is “as is” at a specific time.

Snapshots will allow us to easily capture changes as a type 2 SCD without writing any code for that purpose, as the logic is implemented by dbt itself using merge or delete/insert statements. If we get reliable full exports from the source, the snapshot can also capture deletions.

- ✓ Until now, our dim_apps has represented only the current state of each app as our dimension simply overwrites the previous values. We want to implement SCD2 on this dimension to track changes in categories for instance. In the project root folder, if you kept the default structure, you will see a snapshots folder. If not, create the snapshots folder.
- ✓ Verify that your project yaml file includes snapshots (it does by default).

- ✓ In the snapshots folder, create a sql file where we will update the code for dim_apps so that instead of updating a row when attributes change, we will create a new version of the row and preserve the old version. We will use dbt's snapshotting **check strategy**.

In fact, to track historical changes, dbt provides two snapshot strategies:

- A **Timestamp Strategy**: generally used when we have a reliable column, like updated_at or last_modified_date, that tracks when a record changes. On the first run, dbt takes a full snapshot of the table. On subsequent runs, it will compare the updated_at value in the source column with the existing snapshot updated_at field. If the timestamp has changed, dbt will mark the existing record as expired and insert a new version.
- A **Check Strategy**: where we define a list of columns (via check_cols) whose changes should be tracked, and dbt will compare these columns on every run. If any value differs from the previous snapshot, dbt will record the change.

- ✓ Execute the file and check the saved data in snapshots. You should see that dbt automatically created dbt_valid_from and dbt_valid_to columns.

- ✓ The snapshot table contains metadata columns. We will need now to transform it into a clean dimensional model. Create a new dimension model for dim_apps_scd that retrieves the data from the snapshots rather than the staging area. Make sure to keep the validity timestamps. You can also add a is_current column for easier querying of current values (where dbt_valid_to is null).

- ✓ To test the SCD2, we will simulate some changes in our source data. Modify for instance the category_name of an app in the raw apps JSON file. Run your staging to dimensions models and check for multiple versions of the corresponding app row: One historical (closed) and one one current version (open).

- ✓ Currently, our fact_reviews links to the old app dimension and thereby can only access the current version. To perform true historical analysis, we need to link reviews to the version of the app that was valid at review time.