Documentation

Goal and workflow

The project showcases an ETL pipeline that process and moves raw data from external source system/CRM through staging into an aggregated data mart. The workflow covers four main areas:

- Data modeling
- Data processing (incremental, idempotent)
- Data quality and logging
- Analytical queries

Mock data with predefined, intentional DQ issues is used during implementation. The pipeline can execute repeatedly without producing duplicates and processes only new/changed records.

Tech stack

The PostgreSQL and Docker are used. Docker is used just to simplify environment setup and it is not required to launch the project. The *docker-compose.yml* launches a single container (*pg-storage*) on port 5433 with the *project_data* database.

The pipeline - including schemas, data processing, and queries - is written in PostgreSQL.

More specifically, alongside the docker-compose.yml file, following scripts are presented in GitHub repository:

- *sql/schema_init.sql* - initializes schemas and tables
- *sql/first_mock_data_insert.sql* - insertion of initial, mock data sample
- *sql/etl_pipeline.sql* - main pipeline script
- *sql/second_mock_data_incremental_insert.sql* - insertion of incremental data sample
- *sql/analytical_queries.sql* - analytical queries

The outcome

Besides SQL scripts, the documentation presented here aims to cover rationale behind the utilized approach and to provide some context around decisions made during the implementation of the project solution. In the following sub-sections, main aspects of the pipeline and workflow will be described more precisely.

Data Modeling

Two schemas with tables are introduced to separate concerns across the pipeline. Code is presented in *sql/schema_init.sql* script. Namely, these schemas are:

- *source* schema – for storing raw data, in „as-is" format
- *dwh* schema – structured DWH schema, stores staging tables, helper/ETL control tables and data mart

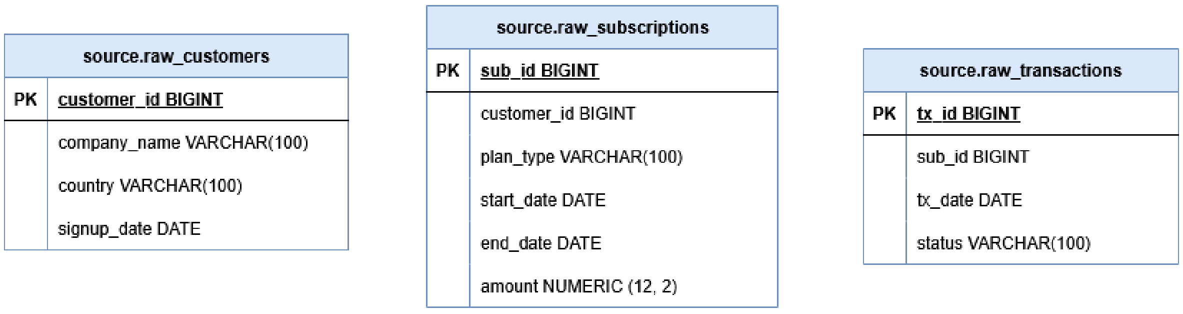
Description of these schemas is provided below.

Source schema

Stores data exactly as received, no business rules or FK constraints are enforced. Still, logical 1:M relationships exist (between customer, subscriptions and transactions tables) but they are not enforced at this layer. The idea is to store data exactly as it arrives from the external source CRM system, even if these records are flawed.

The overview of *source* tables and the composite/natural keys later used for deduplication is shown in table below:

Table name	**Composite/natural key**
source.raw_customers	company_name + country + signup_date
source.raw_subscriptions	customer_id + plan_type + start_date
source.raw_transactions	sub_id + tx_date + status



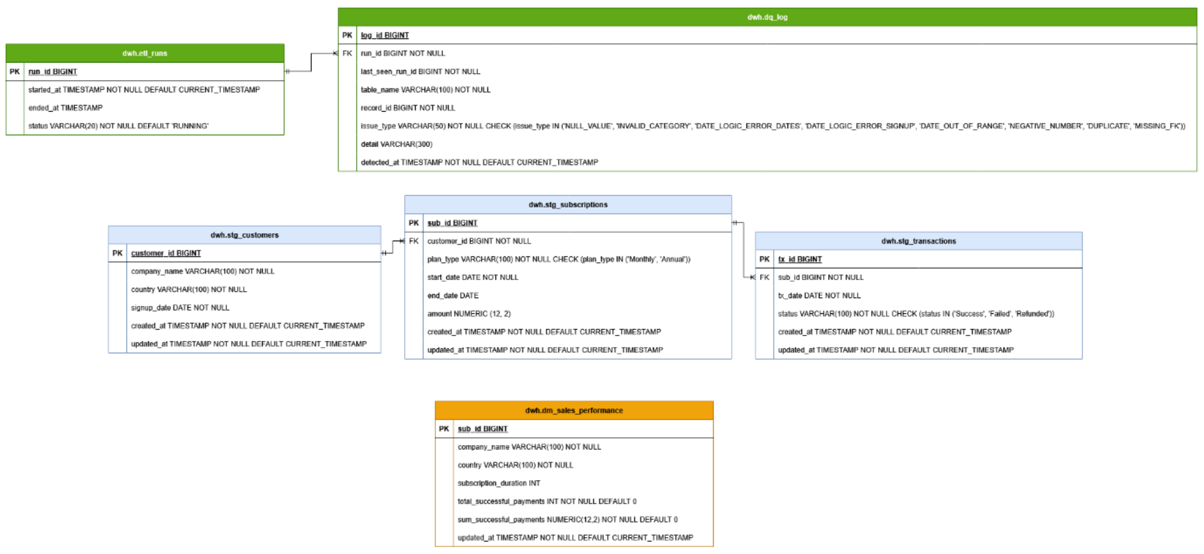
Picture 1: ERD diagram of the tables belonging to source schema

DWH Schema

The DWH schema actually represents the structured warehouse layer for data. It is consisted of 3 logical groups:

1. **Staging tables** (*stg_customers*, *stg_subscriptions*, *stg_transactions*) - they mirror structure of source tables but enforce NOT NULL, CHECK constraints, FK constraints, and date logic. Metadata/technical columns *created_at*/*updated_at* are included.

2. **Helper tables** (*etl_runs*, *dq_log*) - they support pipeline observability. Precisely, *etl_runs* tracks each execution with *started_at*, *ended_at*, and *status* (RUNNING or COMPLETED). On the other hand, *dq_log* captures DQ issues with first-seen (*run_id*) and last-seen (*last_seen_run_id*) tracking, a *detail* column for easier root-cause investigation. The *issue_type* column equals to 8 predefined types of data quality issues: NULL_VALUE, INVALID_CATEGORY, DATE_LOGIC_ERROR_DATES, DATE_LOGIC_ERROR_SIGNUP, DATE_OUT_OF_RANGE, NEGATIVE_NUMBER, DUPLICATE, MISSING_FK.
3. **Data mart** (*dm_sales_performance*) - represents the final, aggregated output, consisting of one row per subscription, covering the required metrics.
   Note that task description requires a column related to *total successful payments* – since it somewhat ambigious whether this refers to the total number of successful payments or total sum of money on basis of successful payments, both metrics have been implemented - *total_successful_payments* and *sum_successful_payments*.


dwh.etl_runs
PK run_id BIGINT
started_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
ended_at TIMESTAMP
status VARCHAR(20) NOT NULL DEFAULT 'RUNNING'
dwh.dq_log
PK log_id BIGINT
FK run_id BIGINT NOT NULL
last_seen_run_id BIGINT NOT NULL
table_name VARCHAR(100) NOT NULL
record_id BIGINT NOT NULL
issue_type VARCHAR(50) NOT NULL CHECK (issue_type IN ('NULL_VALUE', 'INVALID_CATEGORY', 'DATE_LOGIC_ERROR_DATES', 'DATE_LOGIC_ERROR_SIGNUP', 'DATE_OUT_OF_RANGE', 'NEGATIVE_NUMBER', 'DUPLICATE', 'MISSING_FK'))
detail VARCHAR(300)
detected_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
dwh.stg_customers
PK customer_id BIGINT
company_name VARCHAR(100) NOT NULL
country VARCHAR(100) NOT NULL
signup_date DATE NOT NULL
created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
dwh.stg_subscriptions
PK sub_id BIGINT
FK customer_id BIGINT NOT NULL
plan_type VARCHAR(100) NOT NULL CHECK (plan_type IN ('Monthly', 'Annual'))
start_date DATE NOT NULL
end_date DATE
amount NUMERIC (12, 2)
created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
dwh.stg_transactions
PK tx_id BIGINT
FK sub_id BIGINT NOT NULL
tx_date DATE NOT NULL
status VARCHAR(100) NOT NULL CHECK (status IN ('Success', 'Failed', 'Refunded'))
created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
dwh.dm_sales_performance
PK sub_id BIGINT
company_name VARCHAR(100) NOT NULL
country VARCHAR(100) NOT NULL
subscription_duration INT
total_successful_payments INT NOT NULL DEFAULT 0
sum_successful_payments NUMERIC(12,2) NOT NULL DEFAULT 0
updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP

Picture 2: ERD diagram of the tables belonging to dwh schema. Note: green – helper/ETL control tables; blue – staging tables, orange – DM table

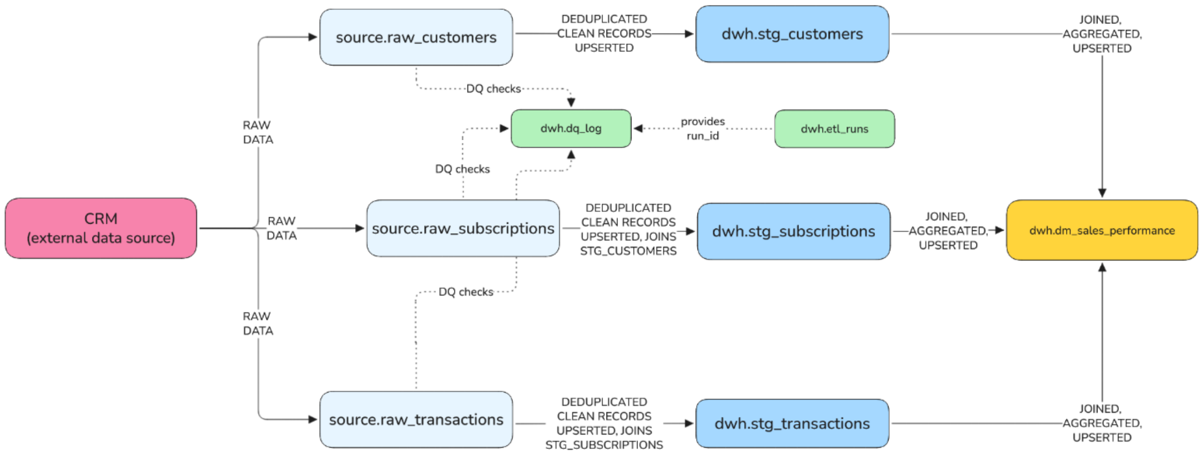
Pipeline overview

The code is provided in *sql\etl_pipeline.sql* file placed in GitHub repo. The pipeline runs in sequential steps. The broad overview of these steps is provided below:

1. **Run initialization** - inserts a new *etl_runs* record (status: *RUNNING*) and captures the *run_id* via INSERT ... RETURNING clause into a temp table *current_run*. The temp table is dropped and recreated at the start of each execution.
2. **DQ logging** - scans all source tables against predefined checks and writes issues to *dq_log*. Uses ON CONFLICT ... DO UPDATE clause so new issues get both *run_id*

and *last_seen_run_id* set to the current run, while previously detected issues have only *last_seen_run_id* updated. This should increase observability across the pipline (e.g. *run_id* = 1, *last_seen_run_id* = 3 means the specific DQ issue persisted across 3 pipeline runs).

3. **Upserts into staging tables** - loads clean records into *stg_customers*, *stg_subscriptions* and *stg_transactions* tables. Each upsert deduplicates, filters out problematic records, and uses ON CONFLICT ... DO UPDATE clause.
4. **Upsert into data mart table** - recalculates all metrics from staging tables and upserts into *dm_sales_performance*.
5. **Pipeline run completion** - updates *etl_runs* with *ended_at* and *status* = *COMPLETED*. If the pipeline fails before this step, the run keeps the *RUNNING* marker.


CRM (external data source)
RAW DATA
RAW DATA
RAW DATA
source.raw_customers
source.raw_subscriptions
source.raw_transactions
DQ checks
DQ checks
DQ checks
dwh.dq_log
provides run_id
dwh.etl_runs
DEDUPLICATED CLEAN RECORDS UPSERTED
DEDUPLICATED CLEAN RECORDS UPSERTED, JOINS STG_CUSTOMERS
DEDUPLICATED CLEAN RECORDS UPSERTED, JOINS STG_SUBSCRIPTIONS
dwh.stg_customers
dwh.stg_subscriptions
dwh.stg_transactions
JOINED, AGGREGATED, UPSERTED
JOINED, AGGREGATED, UPSERTED
JOINED, AGGREGATED, UPSERTED
dwh.dm_sales_performance

Picture 3: High-level overview of the pipeline

The main aspects of the pipeline are described below.

Incremental loading and upsert strategy

Each staging upsert applies window function (ROW_NUMBER) and partition by the composite key, retaining only the row with the lowest PK/ID (meaninng it is earliest inserted; and thus original record). Exclusion criteria for records per staging table is set as follows:

- *stg_customers* - NULL in any required field, *signup_date* outside 2020–2030 (since it is predefined as reasonable year window, excluding records with unreasonable dates/years e.g. 2626). UPPER is applied to *company_name* and *country* columns during deduplication to normalize possible case differences in VARCHAR/text fields (e.g. expected „Company X“ vs possible typos coming from source, like „CoMpANY X“).

- *stg_subscriptions* - NULL in required fields, *plan_type* not in list of allowed categories (e.g. „Quarterly“), *end_date* before *start_date*, dates outside 2020–2030, negative *amount* values, linked customer *signup_date* before *start_date*.
- *stg_transactions* - NULL in required fields, *status* not in list of allowed categories (e.g. „Processing“), *tx_date* before *start_date* of linked subscription, dates outside 2020–2030.

Note that usage of UPPER is applied as comparison „safeguard“ to text/VARCHAR columns (*company_name*, *country*), since *plan_type* and *status* columns are validated separately by INVALID_CATEGORY DQ checks.

The pipeline uses ON CONFLICT ... DO UPDATE for all upserts. No history is retained and old values are replaced with new/current ones during pipeline re-run. The incremental logic of the pipeline is demonstrated by second mock data sample, and its insertion closes a few active subscriptions and adds new customers, subscriptions, and transactions. During the next pipeline run, these changes propagate through staging into the data mart; e.g. previously NULL *subscription_duration* values are calculated, and payment metrics are updated accordingly.

Approach to data quality and logging

The pipeline logs DQ issues before moving data to staging. Issues are logged but data processing does not stop, flawed records are simply excluded from downstream processing.

Brief overview of DQ check coverage per table is shown below.

DQ check	**Table name**		
	raw_customers	*raw_transactions*	*raw_subscriptions*
DUPLICATE	COVERED	COVERED	COVERED
NULL_VALUE	COVERED	COVERED	COVERED
DATE_OUT_OF_RANGE	COVERED	COVERED	COVERED
DATE_LOGIC_ERROR_DATES		COVERED	COVERED
INVALID_CATEGORY		COVERED	COVERED
MISSING_FK		COVERED	COVERED
NEGATIVE_NUMBER			COVERED
DATE_LOGIC_ERROR_SIGNUP			COVERED

Note that there is distinction introduced when it comes to DATE-related DQ issues. Namely, DATE_LOGIC_ERROR_DATES covers direct date violations (e.g. *end_date* before *start_date*, *tx_date* before *start_date*). On the other hand, DATE_LOGIC_ERROR_SIGNUP covers the cross-table cases where a customer's *signup_date* is after a subscription's *start_date*. The separation should allow for more precise root-cause analysis when it comes to data quality investigation.

Also, when it comes to *dq_log* table, UNIQUE constraint (*table_name*, *record_id*, *issue_type*) ensures each DQ issue is stored once. ON CONFLICT DO UPDATE refreshes *last_seen_run_id* and *detail* on every run where the issue is re-detected, while preserving the original *run_id*. The idea is to allow distinction between new DQ issues and persisting ones during post-hoc DQ investigations. Shortly, the whole purpose of *dq_log* table is to capture DQ issues detected during/across pipeline runs. Each pipeline execution inserts a new record into *etl_runs* and captures the *run_id*. This ID is referenced throughout all DQ logging blocks, enabling consistency and traceability and the persistence of detected issues.

Ad-hoc summary queries are also provided to show quick summary of record counts across all pipeline layers and DQ issue counts, providing breakdown by table and DQ issue type. Please note, these queries are placed at the very end of the main pipeline script (*sql/etl_pipeline.sql*) and commented.

Mock data samples

As mentioned, two scripts with mock data are used to test initial and incremental loading. Some of records within each mock data sample contain problematic records, predefined to trigger DQ check and upsert paths.

Analytical queries

The code for queries is provided in *sql/analytical_queries.sql* script.

1st query – MRR/monthly recurring revenue

This query calculates monthly revenue from subscriptions. It uses generate_series to expand each subscription into one row per month it spans. As it is mentioned in task description, monthly plans contribute with the full amount; while annual plans should be divided by 12 and spread across the months. Active subscriptions (NULL *end_date*) are capped at MAX(COALESCE(*end_date*, *start_date*)) with aim to follow the timeline covered by the mock data samples. If this is real use-case with business data, CURRENT_DATE should replace this cap as a meaningful approach.

2nd query - cumulative LTV/lifetime value of the customer

This query calculates actual customer spending over time from their successful transactions. The month series starts from *signup_date* to the customer's last transaction month. Each successful transaction provides the linked subscription amount. Months without transactions show *monthly_spend* = 0; and *cumulative_ltv* relies on a running SUM window function and carries forward.