Project: Data Warehouse with PostgreSQL

Table of Contents

This document accompanies the *SQL Data Warehouse project*, created while following along with the 30-hour SQL Full course on the *Data with Baraa* YouTube channel. The project is to build a Medallion-structured data warehouse with a primary focus on using just SQL to extract from CSV files, perform silver transformations, and restructure and integrate the gold layer into a star schema.

Find the project at https://github.com/arkadianriver/de-dwh-sql.

Preparing the environment

This project was created on Pop_OS! Linux, with PostgreSQL as the DBMS.

Before running the data warehouse scripts, ensure that you have PostgreSQL installed and running on your system. You will also need to have access to the PostgreSQL user with sufficient privileges to create databases and tables. In this project, the scripts use the PostgreSQL root user `postgres`.

To install PostgreSQL:

```
sudo apt install postgresql
```

By default, the service will start automatically after installation. If it doesn’t, you can start it manually using the following command:

```
sudo service postgresql start
```

Documentation

The project documentation is written with AsciiDoc (source in the `src/docsrc` folder). If you want to modify and rebuild the documentation, install Asciidoctor and a couple of supporting gems for PDF and diagrams.

```
sudo apt install ruby
sudo gem install asciidoctor asciidoctor-diagram asciidoctor-pdf
```

Building the documentation

To build the documentation, navigate to the `src/docsrc` directory and run:

```
docbuild.sh all
```

Diagram support

For diagrams, some additional packages are required.

```
sudo apt install graphviz plantuml
```

For mermaid, rather than use `mermaid-cli` from the package manager, I wanted the latest version of mermaid-cli via npm (after installing nvm for the latest version of nodejs).

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh | bash
source ~/.bashrc || source ~/.zshrc
nvm install --lts
```

```
npm install -g @mermaid-js/mermaid-cli
```

An unsuccessful tangent

I like Baraa's specifications table but didn't want to copy it into Asciidoc, so I thought I might extract the text from his image and reformat it into Asciidoc. I ended up letting AI have at it.

OCR image text extraction support

For OCR image text extraction, Tesseract and the Python Pillow and pytesseract libraries are required.

```
sudo apt install tesseract-ocr
pip install Pillow pytesseract
```

Preparing the specifications table

1. Extract the text:

   ```
   pushd src/docsrc/img-extract
   python ocr-extract.py > ocr-output.txt
   popd
   ```

2. …
3. Okay, nevermind. I let ChatGPT have at it and do it all for me. I uploaded the image and gave it this prompt:

   ```
   I will upload an image that is basically a fancy table.
   Extract the text from the image and convert it to a table in AsciiDoc.
   Let me know when you're ready.
   ```

But the OCR was a good exercise anyway.

Preparing the ETL mindmap

Let's try ChatGPT again.

Well done, ChatGPT! Here's the prompt and you can find the source image in this repository under `src/docsrc/img-extract/etl-map.png`.

```
New task: Now I'm going to upload an image of a fancy mindmap diagram.
I'd like you to extract the information from the image and recreate it
using mermaid syntax. Let me know when you're ready.
```

I'll clean it up with colors and flags myself. Can't let AI rob me of all the fun!

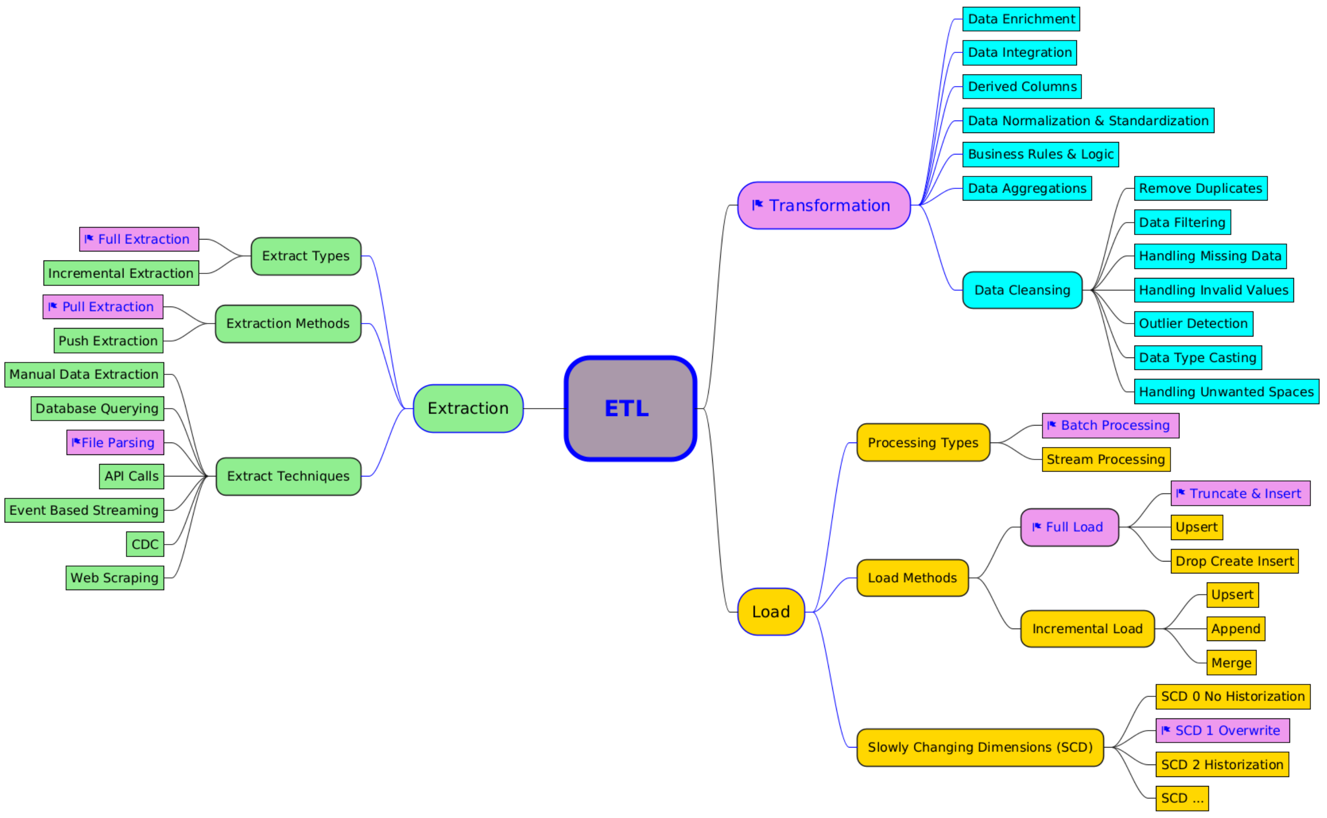
(I later asked ChatGPT to convert to a Drawio file instead, but it had an error about IDs. After debugging with `xmllint` and fixing the IDs, I got a valid Drawio file, but it was only two node levels deep and didn't follow the style anyway. So, I'm sticking with mermaid.)

Data architecture

Before defining the data flow, Baraa helps us map out clear specifications, as if we spent time with stakeholders to understand their needs.

ETL task breakdown

These are the tasks in every ETL process. I'd go over this with stakeholders to see what's required. For this exercise, Baraa wants us to do the tasks highlighted in pink. (We'll do the entire *Transformation* section and its subtasks.)



Specifications

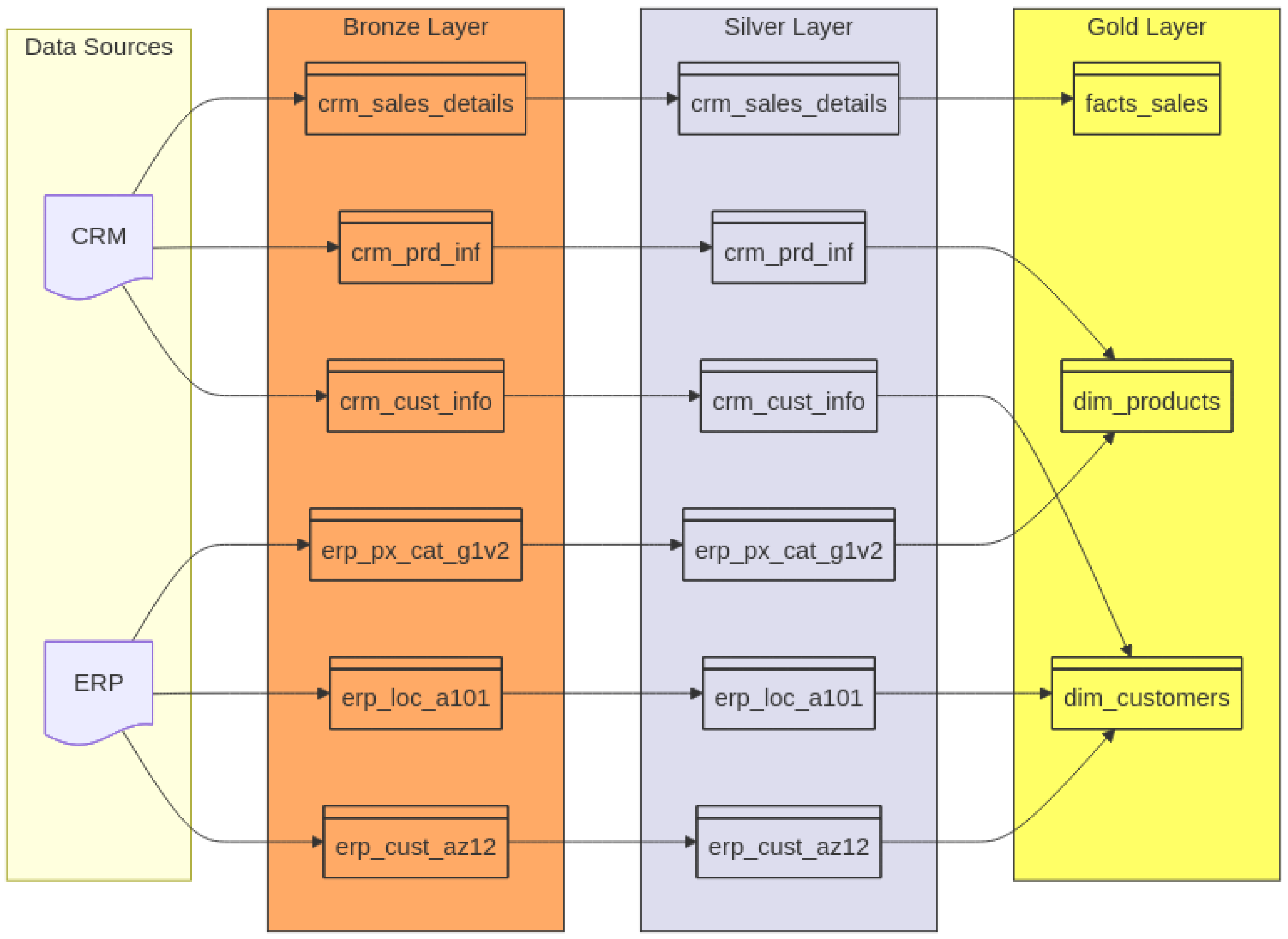
Given the chosen tasks in the ETL process and an understanding of the basic medallion flow, we have the following specifications.

Category	Bronze Layer	Silver Layer	Gold Layer
Definition	Raw, unprocessed data as-is from sources	Clean & standardized data	Business-Ready data

Category	Bronze Layer	Silver Layer	Gold Layer
Objective	Traceability & Debugging	(Intermediate Layer) Prepare Data for Analysis	Provide data to be consumed for reporting & Analytics
Object Type	Tables	Tables	Views
Load Method	Full Load (Truncate & Insert)	Full Load (Truncate & Insert)	None
Data Transformation	None (as-is)	Data Cleaning Data Standardization Data Normalization Derived Columns Data Enrichment	Data Integration Data Aggregation Business Logic & Rules
Data Modeling	None (as-is)	None (as-is)	Star Schema Aggregated Objects Flat Tables
Target Audience	Data Engineers	Data Analysts Data Engineers	Data Analysts Business Users

Data Flow

Looking at the source data we have, this will be the flow to address our specifications.



Code

Pipeline

> Location in the repository: /pipeline.sh

Baraa’s project doesn’t have any kind of automated pipeline, so I was a bit unsure about how to run the scripts in a clean and automated way. After briefly looking at Airflow, I decided to write the ETL pipeline with Bash (`pipeline.sh`), since I started testing things with `psql` on the command line and am not ready to dive into learning the Airflow syntax just yet. Simple.

The `pipeline.sh` script calls SQL scripts run with psql, with output to the command line and a log in `/tmp/sql-data-warehouse/`.

To run the full pipeline:

```
sudo -u postgres ./pipeline.sh full
```

There are other options besides 'full', if you want to run isolated stages of the pipeline. Here are the stages each option runs:

```
case $1 in
    "init")
        init_db
        ;;
    "bronze-create")
        bronze_create
        ;;
    "bronze-load")
        bronze_load
        ;;
    "bronze-all")
        bronze_create
        bronze_load
        ;;
    "silver-create")
        silver_create
        ;;
    "silver-load")
        silver_load
        ;;
    "silver-validate")
        silver_validate
        ;;
    "silver-all")
        silver_create
        silver_load
        silver_validate
        ;;
    "gold-create")
        gold_create
        ;;
    "gold-validate")
        gold_validate
        ;;
    "gold-all")
        gold_create
        gold_validate
        ;;
    "full")
        init_db
        bronze_create
        bronze_load
        silver_create
        silver_load
        silver_validate
        gold_create
        gold_validate
        ;;
    *)  echo -ne "\nUsage: $0 {init\n  |bronze-create|bronze-load|bronze-all\n
|silver-create|silver-load|silver-validate|silver-all\n  |gold-create|gold-
validate|gold-all|full}\n"
esac
```

Bronze layer

Location in the repository: /src/bronze/

The Bronze layer is the initial storage area for raw data ingested from various sources. This layer is designed to store data in its original format, preserving its integrity and providing a foundation for further processing and transformation.

Silver layer

Location in the repository: /src/silver/

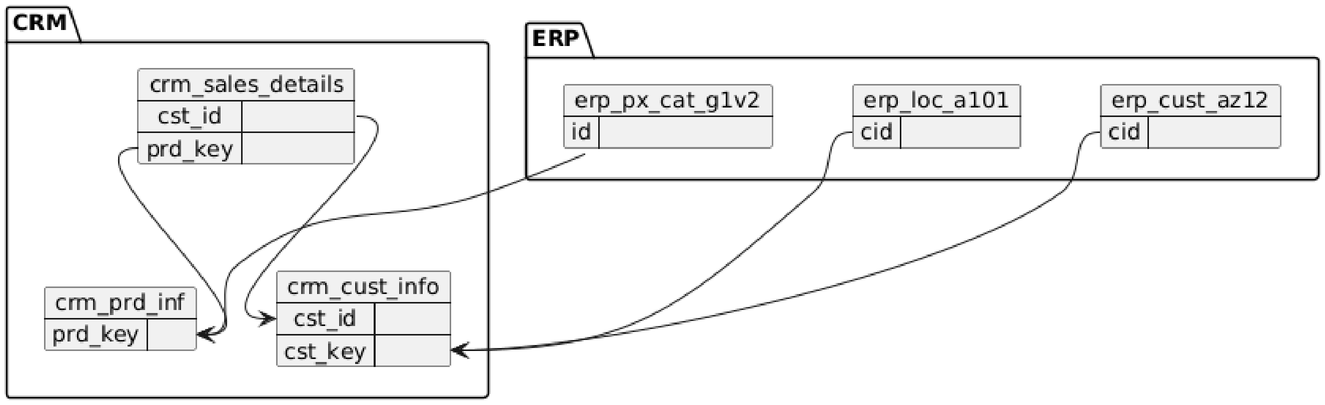
The Silver layer serves as the intermediate processing stage in the data pipeline. In this layer, raw data from the Bronze layer is cleaned, transformed, and enriched to enhance its quality and usability for analysis.

The key functions of the Silver layer include:

- Data Cleaning: Removing duplicates, handling missing values, and correcting inconsistencies in the data.
- Data Transformation: Converting data into a more structured format, applying business rules, and aggregating information as needed.
- Data Enrichment: Integrating additional data sources to provide more context and insights.

By implementing these processes in the Silver layer, we ensure that the data is reliable and ready for advanced analytics and reporting in the subsequent Gold layer.

Data integration analysis and sketch



Coding the transformations

Add metadata columns to track data lineage and help with debugging. We'll add a `created` column to each silver table. Following the data engineer naming conventions (details in Baraa's video), and Postgres syntax, we add this column to the create script:

```
dwh_created     TIMESTAMP DEFAULT CURRENT_TIMESTAMP;
```

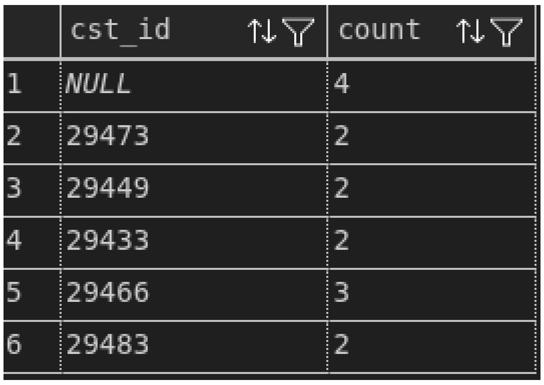
CRM customer info table

Next, we gotta check the data quality of the bronze tables.

First, we'll check for nulls in the primary key columns.

```
select cst_id, count(*) from bronze.crm_cust_info
group by cst_id having count(*) > 1 or cst_id is null
```

Reveals some duplicates.

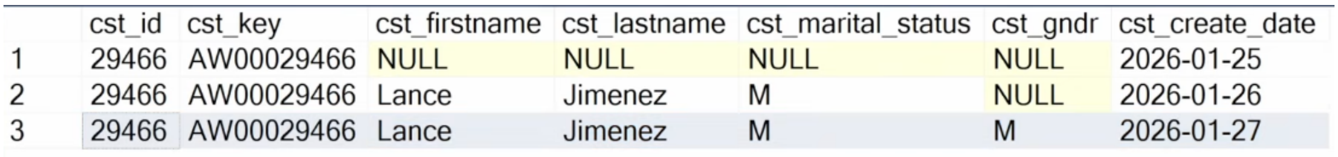


	cst_id	count
1	NULL	4
2	29473	2
3	29449	2
4	29433	2
5	29466	3
6	29483	2

Focus on one to see what the problem is.

```
select * from bronze.crm_cust_info
where cst_id = '29466';
```

Results:



	cst_id	cst_key	cst_firstname	cst_lastname	cst_marital_status	cst_gndr	cst_create_date
1	29466	AW00029466	NULL	NULL	NULL	NULL	2026-01-25
2	29466	AW00029466	Lance	Jimenez	M	NULL	2026-01-26
3	29466	AW00029466	Lance	Jimenez	M	M	2026-01-27

This shows that there are multiple entries with different dates. Since this data warehouse specification doesn't need historical data, we'll take the latest.

But remember, we leave bronze the same and transform (clean) as we go. To do this, let's grab the latest record per `cst_id` and rank them.

```
select *, row_number() over (
    partition by cst_id order by cst_create_date desc
) as flag_last
from bronze.crm_cust_info
where cst_id = 29466;
```

We'll apply this to the whole table to see if this same procedure flags all duplicates.

```
select * from (
    select *, row_number() over (
        partition by cst_id order by cst_create_date desc
    ) as flag_last
    from bronze.crm_cust_info
) sub
where flag_last != 1;
```

Checking against the duplicates found earlier, when we change `flag_last = 1`, we see that all duplicates are removed.

That's just one example of data cleaning in the silver layer. See the `src/silver` folder for the full SQL scripts.

In addition to **removing duplicates in primary keys**, we will do things like:

- **Remove unwanted spaces in text fields**
- **Data normalization or standardization**, e.g. consistency of values in low cardinality columns
- **Handling missing values appropriately**, e.g. replacing NULLs with 'Unknown' or default values

CRM product info and sales tables

For the `crm_prd_info` table, we will, in addition:

- **Derive new columns**, e.g. extracting category from product codes
- **Data type conversions**, e.g. casting timestamps with `00:00:00` times to date
- **Data enrichment**, e.g. making sales end dates make sense and not overlap with start dates

For the `crm_sales_details` table, the sales, quantity, and price columns are tricky. After doing some quality checks, we discover quite a few places where things don't add up. Before doing **data enrichment**, we would verify with the data owner what to do with these bad values.

The rules Baraa came up with are:

- If sales is negative, zero, null, or not equal to derived sales from quantity * price, set it to derived value
- If price is zero or null, derive it from sales / quantity (or null if 0 quantity to prevent div by 0)
- If prices is negative, set it to absolute value (positive)

ERP tables

In addition to deriving a valid foreign key to `crm_cust_info`, there are birthdays that are in the future and as old as 1917! After consulting the data owner, we decide to set future birthdays to null and leaving old birthdays as is.

For the `erp_loc_a101` table, besides deriving the foreign key, we also normalize the country fields.

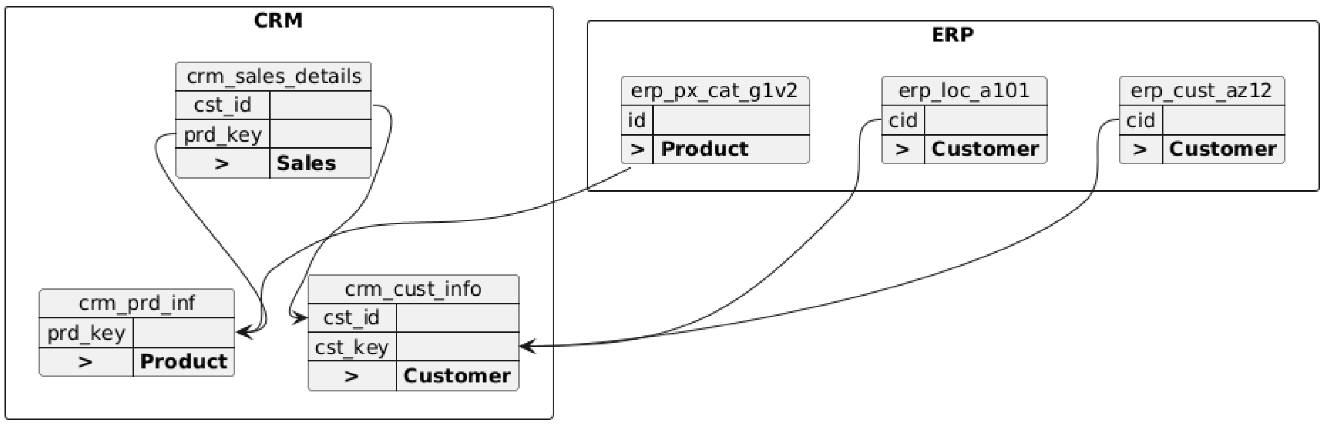
For the `erp_px_cat_g1v2` table, we already derived a category id column in the cust_prod_info silver table that matches the id, and when checking spaces, standardization, and consistency, we find no issues.

Gold layer

> Location in the repository: /src/gold/

The Gold layer represents the final stage in the data pipeline, where the data is refined and optimized for business intelligence and analytics, separated into facts and dimensions. For this project, the star schema model is sufficient as there are only a couple of product and customer *dimensions* surrounding the sales orders *facts*.

First step on how to determine how to build our gold data model is to examine the silver data model and add some labels to group things logically.



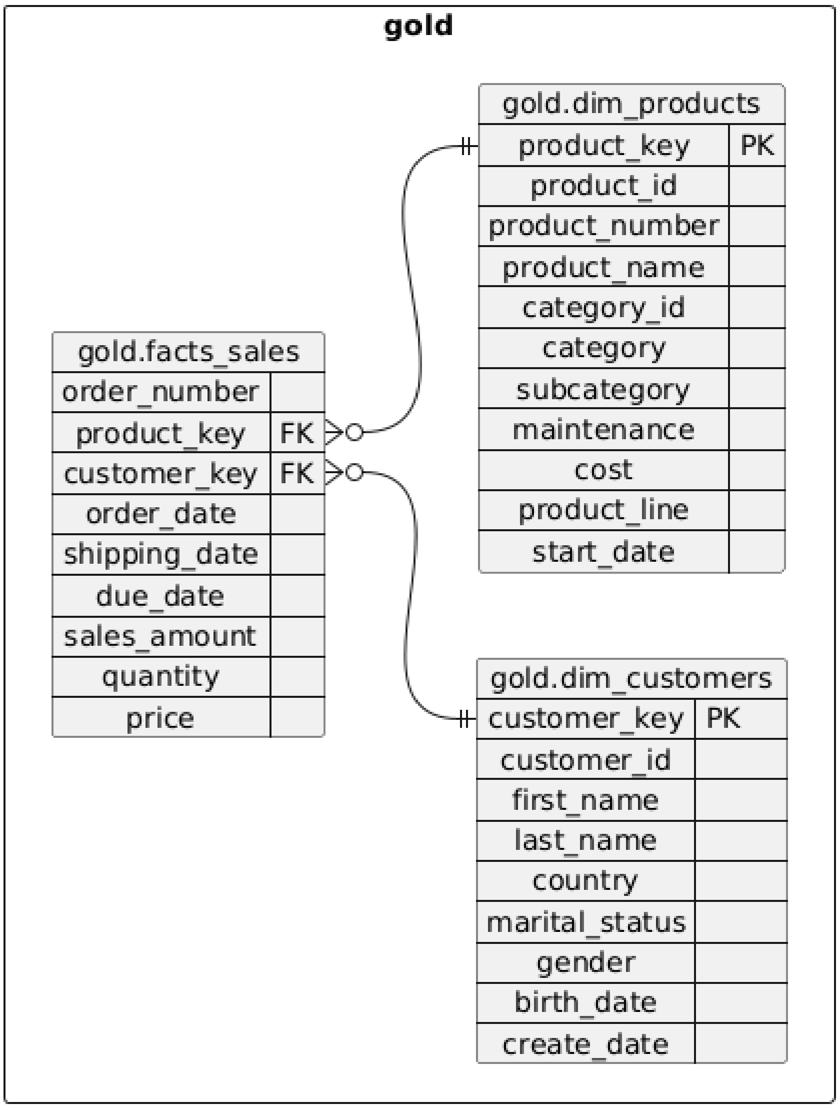
Referring back to the Specifications, we see that rather than loading data, we'll be creating views into the cleaned silver tables, views to integrate and possibly aggregate information.

Another thing to add to the views are surrogate keys to use as the primary keys. TIL that these are exposed in the fact view SELECTs, while it's fact view JOINs that create the links between the relevant columns in the gold views (rather than defining those columns as foreign keys). Only after JOINing the fact table are the foreign key relationships revealed.

To verify the joins (foreign key integrity) execute joins on the keys and to ensure none of them are NULL:

```
SELECT COUNT(*) FROM gold.fact_sales f
LEFT JOIN gold.customers c ON f.customer_key = c.customer_key
LEFT JOIN gold.products c ON f.product_key = c.product_key
WHERE p.product_key IS NULL OR c.customer_key IS NULL;
```

After coding the gold layer, we end up with this data model for our star schema:


gold
gold.dim_products
product_key PK
product_id
product_number
product_name
category_id
category
subcategory
maintenance
cost
product_line
start_date
gold.facts_sales
order_number
product_key FK
customer_key FK
order_date
shipping_date
due_date
sales_amount
quantity
price
gold.dim_customers
customer_key PK
customer_id
first_name
last_name
country
marital_status
gender
birth_date
create_date

The Data Catalog

Distribute this *Data Catalog* to business stakeholders (analysts, data scientists, etc.).

Overview

This catalog describes the Gold layer, the business-level data representation, structured to support analytical and reporting use cases. It consists of **dimension tables** and **fact tables** for specific business metrics.

gold.dim_customers

- **Purpose:** Stores customer details enriched with demographic and geographic data.
- **Columns:**

Column Name	Data Type	Description
customer_key	INT	Surrogate key uniquely identifying each customer record in the dimension table.
customer_id	INT	Unique numerical identifier assigned to each customer.
customer_number	VARCHAR(50)	Alphanumeric identifier representing the customer, used for tracking and referencing.
first_name	VARCHAR(50)	The customer’s first name, as recorded in the system.
last_name	VARCHAR(50)	The customer’s last name or family name.
country	VARCHAR(50)	The country of residence for the customer (See Known countries).
marital_status	VARCHAR(50)	The marital status of the customer (e.g., 'Married', 'Single').
gender	VARCHAR(50)	The gender of the customer (Valid values: 'Male', 'Female', 'Unknown').
birthdate	DATE	The date of birth of the customer, formatted as YYYY-MM-DD. (Oldest possible customer is 115 years old.)
create_date	DATE	The date and time when the customer record was created in the system

Known countries

The following countries are currently the only values that exist in the data.

- Australia
- Canada
- France
- Germany

- United Kingdom
- United States
- Unknown

gold.dim_products

- **Purpose:** Provides information about the products and their attributes.
- **Columns:**

Column Name	Data Type	Description
product_key	INT	Surrogate key uniquely identifying each product record in the product dimension table.
product_id	INT	A unique identifier assigned to the product for internal tracking and referencing.
product_number	VARCHAR(50)	A structured alphanumeric code representing the product, often used for categorization or inventory.
product_name	VARCHAR(50)	Descriptive name of the product, including key details such as type, color, and size.
category_id	VARCHAR(50)	A unique identifier for the product's category, linking to its high-level classification.
category	VARCHAR(50)	The broader classification of the product (e.g., Bikes, Components) to group related items.
subcategory	VARCHAR(50)	A more detailed classification of the product within the category, such as product type.
maintenance_required	VARCHAR(50)	Indicates whether the product requires maintenance (Valid values are 'Yes', 'No', 'Unknown').
cost	INT	The cost or base price of the product, measured in monetary units.
product_line	VARCHAR(50)	The specific product line or series to which the product belongs (e.g., Road, Mountain).
start_date	DATE	The date when the product became available for sale or use, stored in.

gold.facts_sales

- **Purpose:** Stores transactional sales data for analytical purposes.
- **Columns:**

Column Name	Data Type	Description
order_number	VARCHAR(50)	A unique alphanumeric identifier for each sales order (e.g., 'SO54496').

Column Name	Data Type	Description
product_key	INT	Surrogate key linking the order to the product dimension table.
customer_key	INT	Surrogate key linking the order to the customer dimension table.
order_date	DATE	The date when the order was placed.
shipping_date	DATE	The date when the order was shipped to the customer.
due_date	DATE	The date when the order payment was due.
sales_amount	INT	The total monetary value of the sale for the line item, in whole currency units (e.g., 25).
quantity	INT	The number of units of the product ordered for the line item (e.g., 1).
price	INT	The price per unit of the product for the line item, in whole currency units (e.g., 25).